Proximity 4.3 QGraph Guide

Proximity 4.3 QGraph Guide

Published November 15, 2007

This effort is or has been supported by AFRL, DARPA, NSF, and LLNL/DOE under contract numbers F30602-00-2-0597, F30602-01-2-0566, HR0011-04-1-0013, EIA9983215, and W7405-ENG-48 and by the National Association of Securities Dealers (NASD) through a research grant with the Univeristy of Massachusetts. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of AFRL, DARPA, NSF, LLNL/DOE, NASD, the University of Massachusetts Amherst, or the U.S. Government.

General inquiries regarding Proximity should be directed to:

Knowledge Discovery Laboratory
c/o Professor David Jensen, Director
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003-9264

Table of Contents

1. Introduction 1
    - Using the QGraph Guide 3
2. Query Basics 5
    - Query Structure 5
    - Query Results 5
    - Undirected Edges 9
    - Handling Database Self-Links 10
    - Implementation in Proximity 10
    - Summary 11
3. Conditions 13
    - Attribute Value Conditions 13
    - Existence Conditions 14
    - Evaluating Conditions 16
    - Complex Conditions 19
    - Implementation in Proximity 21
    - Summary 23
4. Numeric Annotations 25
    - The Need for Counting 25
    - Annotation Basics 26
    - Understanding Multiple Annotations 32
    - Negated and Optional Elements 36
    - Adjacency Requirements 42
    - Implementation in Proximity 45
    - Summary 46
5. Constraints 47
    - Identity Constraints 47
    - Attribute Constraints 51
    - Multiple Constraints 52
    - Evaluating Constraints 53
    - Constraints and Annotations 55
    - Implementation in Proximity 57
    - Summary 58
6. Subqueries 59
    - Subqueries and Annotations 62
    - Subqueries and Constraints 63
    - Multiple Subqueries 67
    - Implementation in Proximity 69
    - Summary 70

A. Using the Proximity Query Editor 73
    - Introduction 73
    - Working with the Query Editor 74
    - Working with Vertices and Edges 76
    - Working with Updates 78
    - Working with Subqueries 78
    - Working with Queries 79

B. XML Representation 81
    - Declarations 81
    - The `graph-query` root element 81
    - Query Body 82
    - Vertices 82
    - Edges 83
    - Conditions 84
    - Numeric Annotations 84

Subqueries 85
Constraints 86
Test Expressions 87
Update Functionality 88
Editor data 89
References 91
Glossary 93
Index 97

Chapter 1. Introduction

QGraph is a visual query language designed to support knowledge discovery in large graph databases. Such databases are used to represent *relational* data, that is, data that explicitly represent both objects and the relationships among them.

QGraph has some important differences from standard relational database query languages such as SQL. Most obviously, QGraph is a visual query language. Queries are created by drawing the desired graph structure and adding restrictions in the form of conditions, constraints, and numeric annotations. QGraph includes a grouping and counting mechanism that lets users move beyond specifying an exact database structure in a query. Queries can contain generalized structural descriptions, enabling a single query to match a variety of database structures.

Unlike SQL, which returns individual records, QGraph queries return *subgraphs*, complex objects that correspond to connected sets of objects and relations in the queried database. These subgraphs retain the full details of the database objects and links rather than just providing aggregations such as count or average. Subgraphs give users access to both the content and structure of the matches when analyzing and exploring relational data.

QGraph is used in Proximity, an open-source system for relational knowledge discovery. Although QGraph is a full-fledged query and update language for graph databases, Proximity currently only implements a large subset (but not all) of QGraph functionality. This document focuses on the features of QGraph that have been implemented in Proximity. See the technical report *A Visual Language for Querying and Updating Graphs* [Blau, Immerman, and Jensen, 2002] for a full description of QGraph's power, including functionality not yet available in the current Proximity implementation.[1]

QGraph is designed to work with graph databases such as those used by Proximity. Proximity databases are directed, attributed graphs where nodes correspond to objects (typically people, places, and things) and binary links represent the relationships among objects. Both objects and links can have zero or more attributes and all attributes are set valued. For example, in a database containing information about movies, an object representing an actor might have a name attribute that has multiple values for the actor's given and stage names. Similarly, a link connecting an actor to a movie might have a role attribute that could contain multiple values when the actor plays multiple roles in a film.

QGraph queries return a collection of subgraphs from the database that match the structure specified in the query. In Proximity, this result set is called a *container*. A container's subgraphs include the matching objects and links as well as all associated attributes for these elements.

To get a sense of QGraph's power and flexibility, consider a graph database containing information on the movie industry. Such a database might represent movies, actors, directors, producers, studios, and awards as objects, with links representing the relationships among these entities. We might query such a database to identify simple subgraphs such as movies and their actors, or more complex subgraphs such as all the actors who have worked with a particular director, following links through the associated movies. The query in Figure 1.1 finds just such subgraphs.

[1] How QGraph evaluates conditions and constraints on attributes with multiple values has changed since the publication of this technical report. On this issue, the *Proximity QGraph Guide* is definitive.

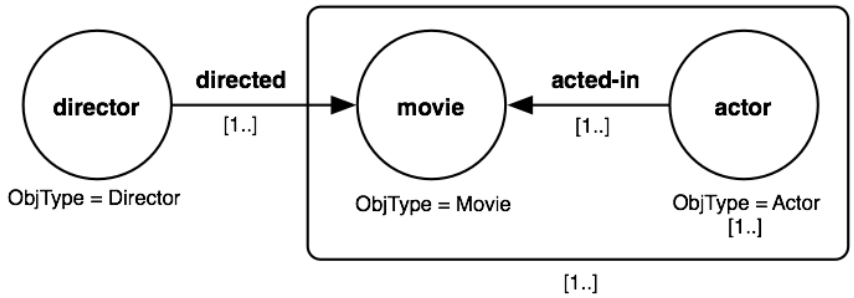


Figure 1.1. Example query [Intro_DB01_Q01.qg2.xml]

Without explaining the query's notation at this point, we observe that this query finds directors connected to one or more movies, which are in turn are connected to one or more actors. When the query finds a match in the database, it returns the entire matching subgraph.

Figure 1.2 shows the kind of subgraph that might get returned as a match for such a query. This subgraph shows the director Steven Spielberg and the links from him to the movies that he's directed. Those movies are in turn linked to the actors that appeared in those movies. (Like other examples in this guide, data for this example is intended to be representative, but not necessarily complete.)

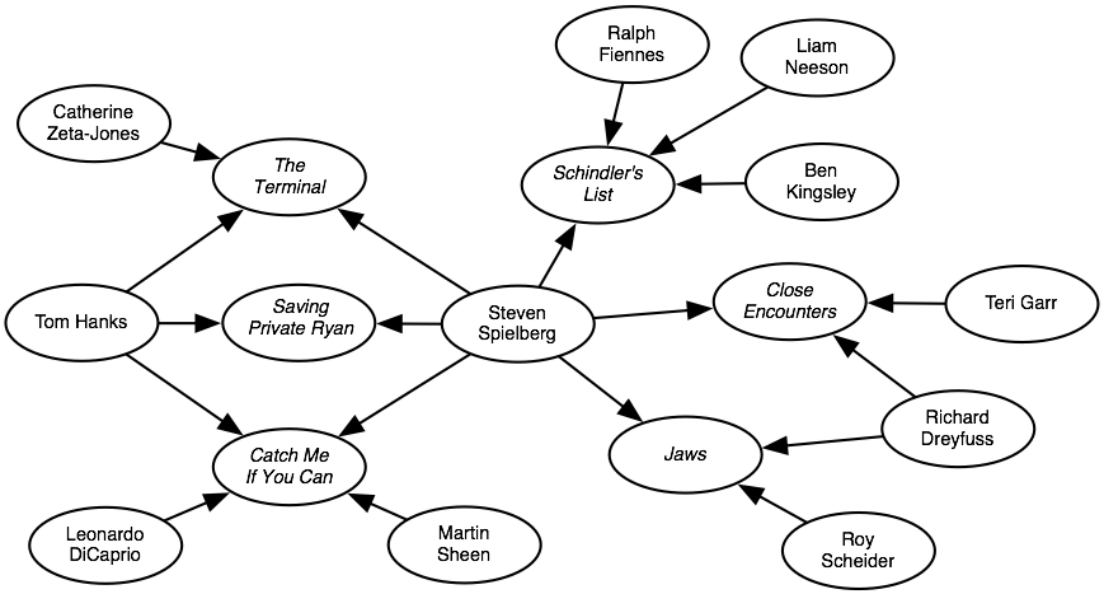


Figure 1.2. Example query match

An important feature of this subgraph is that movies are linked to different numbers of actors. We do not need to know, and the query does not have to specify, how many actors are linked to each movie. Similarly, we do not need to know how many movies Steven Spielberg has directed. The matching subgraph for another director is likely to include a different number of movies, and those movies are also likely to be linked to varying numbers of actors. QGraph lets us create a simple, easily understood query that matches varying graph structures.

By returning complete subgraphs, QGraph provides results particularly well suited for use in relational knowledge discovery. Unlike query languages that require propositionalizing attribute data from related objects, QGraph preserves both the explicit relationships among objects and the specific attribute values for all objects and links in the subgraph. Simplistic approaches to propositionalizing relational data can result in bias due to hidden autocorrelation or degree disparity. By preserving the full relationship and attribute value data, QGraph facilitates the development of models that can detect and adjust for such sources of bias.

QGraph includes several mechanisms that together provide a powerful system for specifying the graph structures that match a QGraph query: These mechanisms are introduced in the following chapters:

- Chapter 2, *Query Basics* (p. 5) describes the basic structure and results of a QGraph query
- Chapter 3, *Conditions* (p. 13) introduces conditions, which place restrictions on the attribute values of matching objects and links.
- Chapter 4, *Numeric Annotations* (p. 25) describes numeric annotations, a powerful and elegant counting feature that serves to limit and group query results.
- Chapter 5, *Constraints* (p. 47) describes constraints, a global mechanism that relates one object or link to another.
- Chapter 6, *Subqueries* (p. 59) describes subqueries, which enable the treatment of connected subgraphs as a logical unit in defining more complex queries.
- Appendix B, *XML Representation* (p. 81) describes the Proximity XML representation for queries.

Using the QGraph Guide

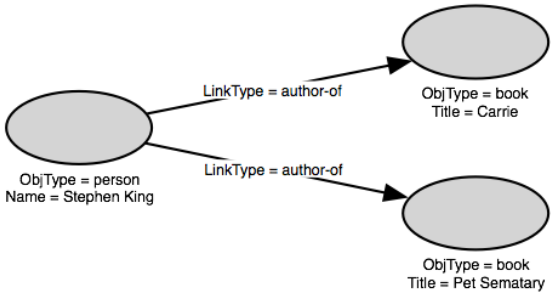
A QGraph query is a labeled graph designed to correspond to selected structures in the target database. The vertices in the query correspond to objects in the database and the edges in the query correspond to database links. This distinction in terminology is important and will be maintained throughout this guide—we use the terms *object* and *link* when referring to database entities, and the terms *vertex* and *edge* when referring to query elements.

Because QGraph is currently only implemented in Proximity, this *Guide* includes a discussion of how that implementation may affect the use and interpretation of QGraph queries and results. When applicable, each chapter includes a section that describes factors you should be aware of when using QGraph in Proximity, such as how query results are stored in the database, efficiency considerations, or implementation restrictions. Information presented in these sections is not part of the QGraph specification, but can be useful when understanding how QGraph works in Proximity.

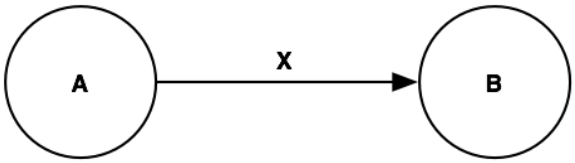
An important characteristic of Proximity databases is the lack of an inherent sense of object or link types. Type information is stored as an attribute value, just like any other attribute. When we refer to an actor object or role link, we are using a convenient shorthand for “an object whose ‘type’ attribute has a value of ‘actor’ ” or “a link whose ‘kind-of’ attribute has a value of ‘role.’ ” Users are free to include or not include type information in a database’s object and link attributes and to use whatever name they chose for an attribute that stores type information. Descriptions in this document may refer to objects and links by type (e.g., an actor object or role link) or value (e.g., the Peter Sellers object) with the understanding that this is a shorthand for database objects and links containing attributes that represent this “type” information or that have the specified attribute value.

Databases, QGraph queries, and matches identified by executing those queries are all graphs. To help distinguish these items, the *Guide* follows the following graphic conventions:

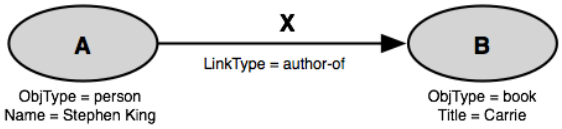
Database fragments use shaded ovals to represent the objects in the database. Attributes are shown next to their corresponding object or link.



Queries use white circles for vertices. Query vertices and edges are labeled.



Query results use shaded ovals to represent the objects in the resulting subgraphs. Subgraph objects and links are labeled with the corresponding query labels. Because QGraph queries return complete subgraphs that include all attribute values for the component objects and links, we also show attribute values in our graphical representations of subgraphs.



The *QGraph Guide* is primarily intended for readers with little to no experience using QGraph. The major chapters are heavily oriented around a series of examples, making the *Guide* a relatively quick read. More experienced users may want to take advantage of the summary at the end of each chapter or return to the *Guide* to improve their understanding of specific issues.

Most of the examples in the *QGraph Guide*, including both database fragments and queries, are available in the `$PROX_HOME/doc/user/qgraph/examples` directory. When such files are available, the figure caption includes the appropriate file name. See the *Proximity Tutorial* for complete instructions for importing databases and running queries.

Chapter 2. Query Basics

Query Structure

A QGraph query is a labeled graph that describes the structure and content of the database entities (objects and links) to be matched. Vertices and edges in the query correspond to objects and links in the database. Every query must have at least one vertex and every edge must have vertices at both ends. A QGraph query must be a connected graph, with a single vertex being the simplest QGraph query.

QGraph queries are represented graphically. The example below shows a query that matches two objects connected by a directed link.

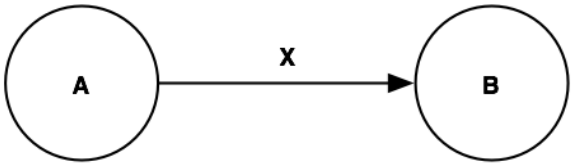


Figure 2.1. Example query structure

Each vertex and edge in a QGraph query has a unique name within the context of the query. These names have no intrinsic meaning and serve mostly to provide meaningful labels to the user in query results.

For the example queries in this document, we use either arbitrary alphabetic labels or semantically meaningful names for ease in understanding the query. For example, if we define a query vertex that matches database objects with the attribute value ObjType = movie, we might call this the *movie* vertex. Similarly, a query edge defined to match links with the attribute value LinkType = acted-in might be named *appeared-in* in the query. By convention, when using alphabetic labels, we use letters at the beginning of the alphabet such as *A*, *B*, and *C* to label vertices, and letters at the end of the alphabet such as *X*, *Y*, and *Z* to label edges. When context makes the intent clear, we will also use an informal shorthand to refer to database entities within this document. For example, we might refer to an object with the attribute Name = James Dean as the James Dean object.

Although the underlying graph model uses only directed links, QGraph supports both directed and undirected edges. That is, QGraph requires that the database use directed, binary links, but queries can either specify a required direction for edges and thus for corresponding links in the database, or indicate that the directionality of a link doesn't matter when matching the query.

Query Results

Evaluating a QGraph query returns a *container*, a collection of all the matching subgraphs from the database. To match the query, a subgraph must have the correct structure and satisfy all of the query's requirements (conditions, numeric annotations, and constraints). This chapter covers how query structure matches database structure; conditions, numeric annotations and constraints are covered in subsequent chapters.

It's important to understand how the structure of a QGraph query matches database structures. Let's look at an example database fragment and see how a simple query matches that fragment. Figure 2.2 shows a fragment of a database containing information about books, movies, and writers. We can see that this fragment includes three different types of objects, as represented by the different values for the ObjType attribute. Similarly, the LinkType attribute describes the type of relationship that links two objects in the database. Additional object attributes provide more details about the objects, such as a person's name or a book's publication date.

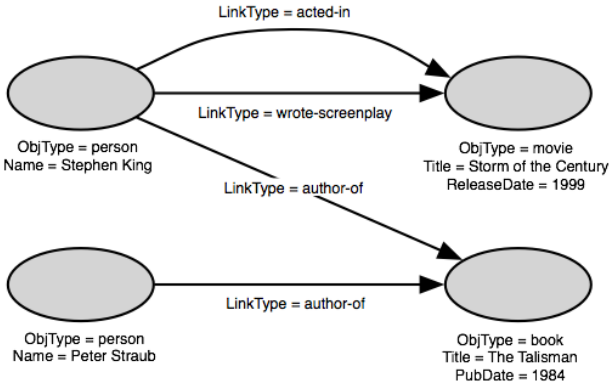


Figure 2.2. Database fragment [Basics_DB01.xml]

The query shown in Figure 2.3 finds two objects connected by a directed link. It will match any database subgraph with this structure.

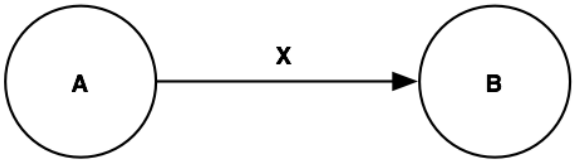


Figure 2.3. Basic structural query [Basics_DB01_Q01.qg2.xml]

Executing the query on the database fragment shown above yields the four subgraphs shown in Figure 2.4.

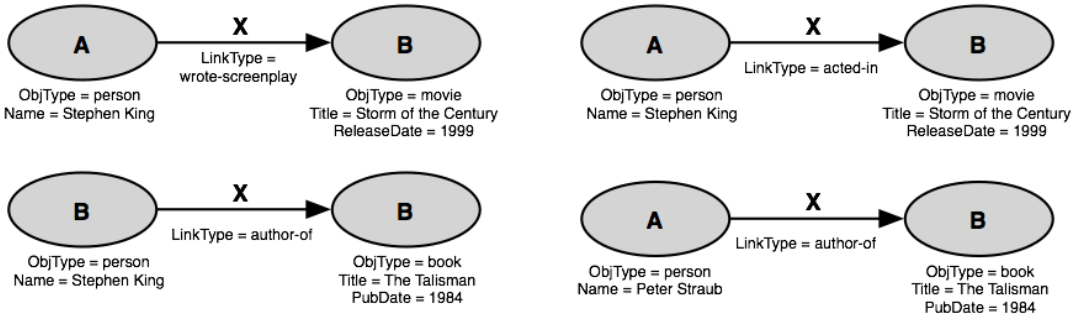


Figure 2.4. Query results

The database fragment in Figure 2.2 includes four different instances of two objects connected by a directed link, which correspond to the four subgraphs in the query results. Each match to the query yields a separate subgraph in the results container, for example, the two links between Stephen King and *Storm of the Century* create two different matches to the query and therefore two different subgraphs in the query results. Importantly, the query returns subgraphs that include all related object and link attributes. Query results retain the vertex and edge names used in the query. These names are used only in the context of the result subgraphs; they are not added to the database objects or links themselves.

Let's examine the results of a slightly more complicated query run on another database fragment, shown in Figure 2.5. This time our fragment contains information on movies and their casts. In particular, we see that Elizabeth Taylor appeared in *Butterfield 8*, that both Elizabeth Taylor and and Richard Burton appeared in *Cleopatra*, and that the two actors were married to each other.

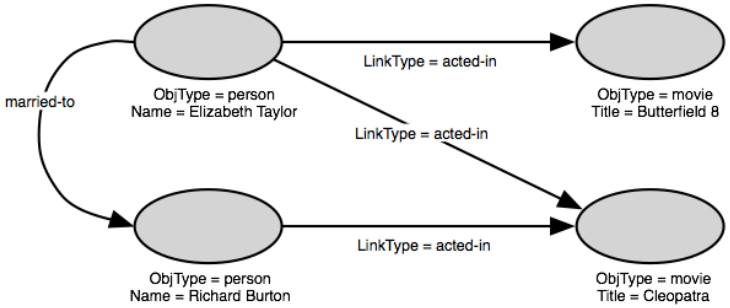


Figure 2.5. Database fragment [Basics_DB02.xml]

This time, our query finds all subgraphs that have two objects linked to a third object. Both links must be directed and must point to the third object. The new query is shown in Figure 2.6.

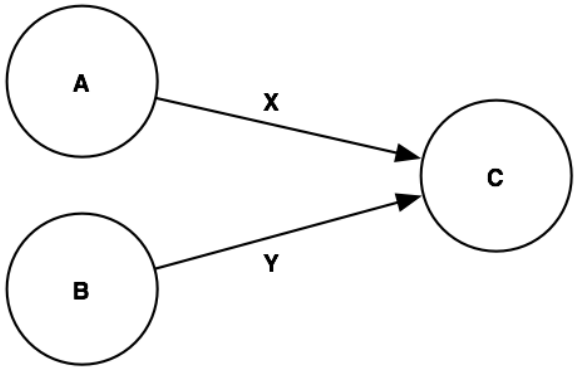


Figure 2.6. Three-vertex query [Basics_DB02_Q01.qg2.xml]

The results of executing this query on the above database fragment are shown in Figure 2.7.

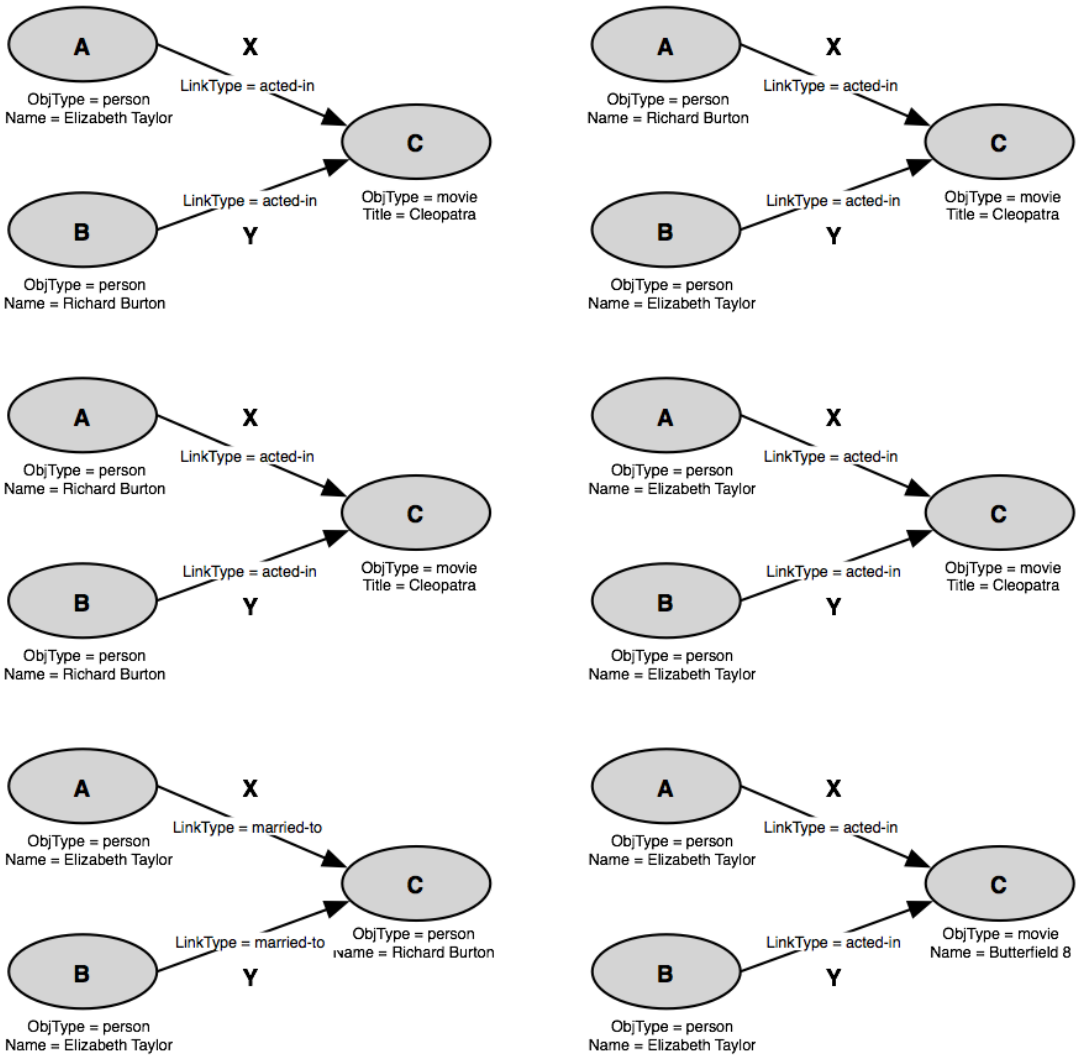


Figure 2.7. Query results

Because QGraph finds each unique match of the query's structure, the query's results can include subgraphs that contain the same *database* objects and links but which are mapped to different *query* elements. This can be seen in the top two subgraphs in Figure 2.7. Both subgraphs contain the objects Elizabeth Taylor, Richard Burton, and *Cleopatra* and their associated acted-in links, but they correspond to different vertices and edges in the query.

Additionally, QGraph does not require that distinct query elements match distinct database entities, therefore database elements may appear more than once in an individual subgraph if they correctly match the query's structure. In this example, the last four subgraphs include duplicated objects and links. Although they correspond to the same database entity, such duplicated objects and links are treated as distinct entities within the subgraph because they match distinct query elements. Query constructs such as conditions, numeric annotations, and constraints, introduced later in this guide, can be used to restrict matches and avoid this kind of duplication in query results when desired, but the query structure alone carries no such requirement.

Just as important as what a query returns, is what it doesn't return. Note that the first two subgraphs in Figure 2.7 do not include the married-to link from Richard Burton to Elizabeth Taylor. QGraph does not return links in the database that are not specified in the query, even when both link ends are in the same subgraph.

Undirected Edges

Sometimes we don't care about the direction of a link when identifying matching subgraphs. For example, we might care only that a path exists between two objects, but not about the direction of that path. In such a case, we want to create a query that requires that two objects be linked, but that doesn't require a specific direction for that link.

An *undirected edge* in a QGraph query indicates that there must be a link between the corresponding database objects without concern for the direction of that link. To see how this affects query results, let's look at the following sample database and query. Figure 2.8 shows a fragment of a database containing information about Web pages and their connections.

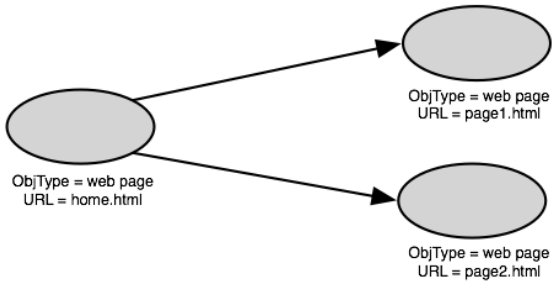


Figure 2.8. Database fragment [Basics_DB03.xml]

Figure 2.9 is a simple query designed to find sets of two linked objects. Because we don't care whether the link goes from *A* to *B* or vice versa, we use an undirected edge to represent that connection.

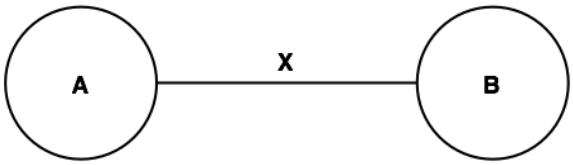


Figure 2.9. Query [Basics_DB03_Q01.qg2.xml]

The results of running this query on the above database fragment are shown in Figure 2.10.

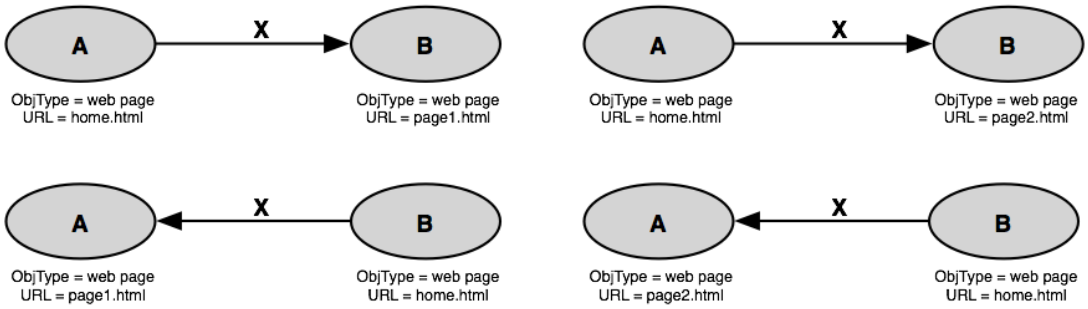


Figure 2.10. Query results

Ignoring link direction lets this query match any pair of objects as long as those objects have a link connecting them. As we saw before, QGraph finds as many matches as it can, including subgraphs that duplicate the elements of another match but that map those database elements to different query elements. In this example, we have two subgraphs that include the linked objects home.html and page1.html. In one subgraph, home.html matches vertex *A*; in the other it matches vertex *B*. The link matching edge *X* always follows its direction in the database (from home.html to page1.html), even though it points from *A* to *B* in one subgraph and from *B* to *A* in the other.

Handling Database Self-Links

Some relational databases contain objects that link back to themselves. For example, a database where the objects represents Web pages is likely to include such a construct because many Web pages include links from one part of the page to another part of the same page. It's important to understand how QGraph handles self links (loops) when matching queries against the database structure.

Consider the database fragment shown in Figure 2.11.

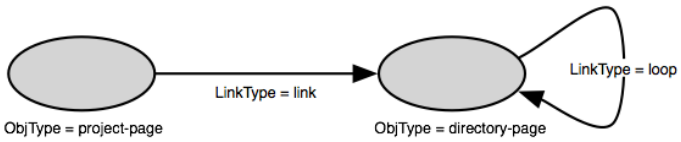


Figure 2.11. Database structure containing a loop [Basics_DB04.xml]

This fragment contains a directed link that connects the directory-page object to itself, a common phenomenon in long Web pages. QGraph treats this link and object the same as any other in the database, including matching them to more than one query element when appropriate.

To see how this works, let's see what happens when we execute the query shown in Figure 2.12 on the database fragment shown above.

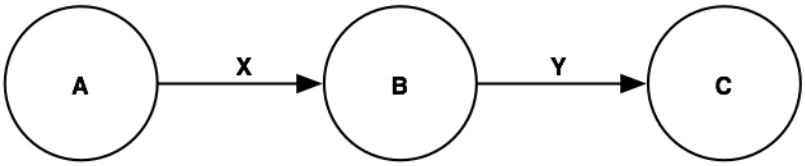


Figure 2.12. Query [Basics_DB04_Q01.qg2.xml]

The results of executing this query on the database fragment are shown in Figure 2.13.

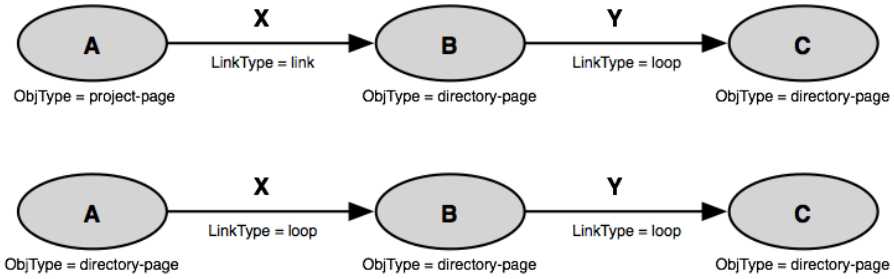


Figure 2.13. Results

The loop connecting the directory-page object to itself is treated the same as any other link. Like all other links, it can match more than one query edge and thus appear in more than one place in a subgraph. Again, query constructs such as conditions and constraints can be used to restrict which database elements a query edge or vertex may match and to avoid such duplications in query results when desired.

Implementation in Proximity

Query results

Query matches are returned as a collection of subgraphs called a container. Containers and subgraphs are structural views of the data that are added as persistent items to the database.

Subgraphs include the database objects and links that match the originating query. Importantly, these are the actual objects and links in the database and not copies. If you modify a database entity, every subgraph that includes that entity will be correspondingly modified. Similarly, if you modify an object or link in a subgraph, you are modifying the underlying database entity. Deleting an object or link from the database will similarly impact existing subgraphs, potentially resulting in disconnected or invalid subgraphs.

Changes that delete database entities or alter attribute values may change subgraphs so that they no longer satisfy the original query. Proximity does not re-evaluate the query to update containers when the underlying database changes. Any subgraphs rendered invalid by database changes remain in the container until the query is re-executed.

Efficiency considerations

In general, the efficiency considerations described in this *Guide* may improve processing speed for extremely complex queries or queries run against an extremely large database. They are unlikely to have a noticeable effect on query processing speed under normal circumstances. Taking advantage of some of these efficiency improvements may require perfect information about a database's data and design. However, in practice, we frequently do not have thorough enough information about a database to be able to use efficiency improvements with confidence that any modifications made for efficiency do not also affect query results.

Directed versus undirected edges

Queries containing undirected edges can be much more computationally expensive than those that contain only directed edges. Because the underlying database model uses only directed links, Proximity must check for links in both directions when the query specifies an undirected edge, increasing the processing time accordingly.

For those instances where a directed and undirected edge would return identical matches (such as when some links only occur in a specific direction), use the directed edge for more efficient processing. For example, in a database where links only go from actors to movies (and never from movies to actors), queries should use directed edges to match these links, even though an undirected edge would return the same results.

Summary

Query format

- Every query must have at least one vertex and every edge must have vertices at both ends.
- A QGraph query must be a connected graph.
- QGraph queries can be as simple as a single vertex.
- QGraph supports both directed and undirected edges but requires that the underlying database use directed links.
- Each vertex and edge in a QGraph query has a unique name (label) used within the context of the query. These names are used to identify the elements in the query and resulting subgraphs but have no effect on what the query matches.

Query results

- QGraph queries return their results in the form of a collection of subgraphs called a container.
- Query results include all attributes for the objects and links in the container's subgraphs, even if those attributes are not explicitly included in the query.
- QGraph does not require that distinct query elements match distinct database entities, therefore individual database items may appear more than once in a subgraph.
- QGraph does not return links in the database that are not specified in the query, even when both link ends are in the same subgraph.
- Loops (links that point back to the originating object) in the database are treated identically to other links.

Proximity implementation

- Subgraphs and containers are added as persistent items to the database.
- Subgraphs include the actual database objects and links that match the originating query, not copies.
- Queries containing undirected edges are more computationally expensive than those that contain only directed edges.

Chapter 3. Conditions

Being able to only match structure is of limited usefulness in querying the database. We must also be able to consider the semantic information contained in object and link attributes when defining and matching queries. *Conditions* allow queries to move beyond purely structural matches to consider database content. Specifically, conditions serve to restrict the number of query matches by placing requirements on the attributes of an object or link in the database.

Conditions are commonly, but certainly not exclusively, used to enforce type restrictions in query matches. However, QGraph only tests and compares attribute values; it makes no assumptions about how type information is stored. It's up to you to specify the appropriate attribute to examine.

QGraph provides two types of conditions:

- *Attribute value conditions* restrict matches to only those database entities that match the attribute values specified in the query.
- *Existence conditions* test only to see whether the corresponding object or link has a specified attribute, but does not require that the attribute have any particular value.

Attribute Value Conditions

To see attribute value conditions in action, let's revisit the database originally shown in Figure 2.2 and repeated below. This database contains information on authors, books, movies, and the relationships among these entities.

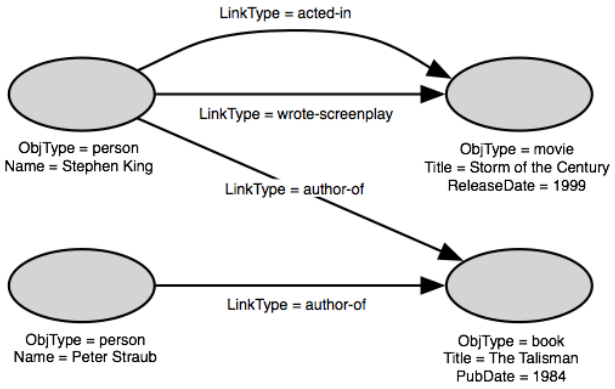


Figure 3.1. Database fragment [Basics_DB01.xml]

In this database, object type information is represented as the value of the ObjType attribute and link type information is represented as the value of the LinkType attribute. The query in Figure 3.2 identifies all author-book pairs in the data by matching database structures that connect person objects to book objects via author-of links.

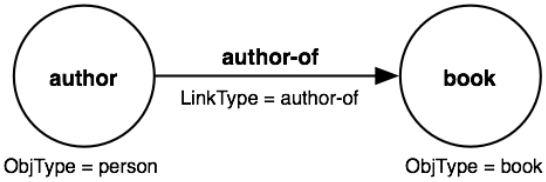


Figure 3.2. Using conditions for type matching [Basics_DB01_Q02.qg2.xml]

The text under each query element specifies a condition on that element. The *author* vertex has the condition ObjType = person. Only objects having the value person for the ObjType attribute will match this vertex. Similarly, the query will only match links whose LinkType attribute have the value author-of.

Once we add conditions, it's common to use semantically meaningful names for the vertices and edges in a query. Such names usually reflect the restrictions imposed by the conditions, such as we see above. Although these names may duplicate required attribute values, it is the condition, not the label, that determines query behavior. Executing this query on the database fragment above yields the results shown in Figure 3.3.

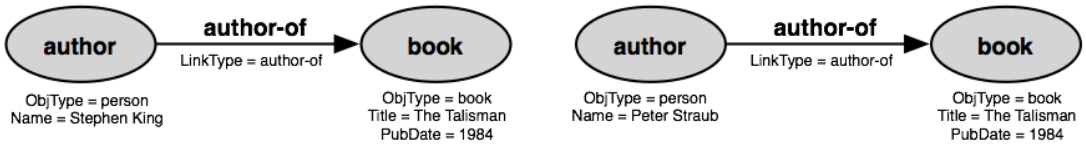


Figure 3.3. Results of using conditions for type matching

Compare these results to the purely structural matches shown in Figure 2.4. Adding conditions to the query limits the results to only those subgraphs that have the required attribute values. Specifically, the new query requires that the *author* vertex match only person objects, the *book* vertex matches only book objects, and the *author-of* edge matches only author-of links. The new query yields only two matches versus the four subgraphs found by the original query.

As we noted earlier, unlike the `SELECT` statement in SQL, a QGraph query does not specify which attributes of matching objects or edges are included in the query results (subgraphs). This continues to hold even when queries include conditions that mention only some of these attributes. All the object and link attributes, not just those mentioned in the query, are available for inspection in the query results.

In addition to testing for equality, conditions can compare attribute values to the specified values using the comparison operators <>, <, <=, >, and >=. For example, to find authors who have published before 1990, we create the following query:

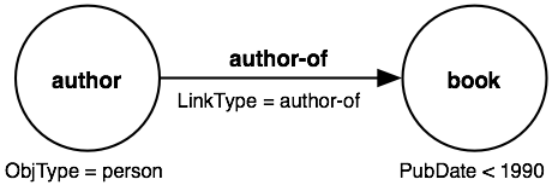


Figure 3.4. Query [Basics_DB01_Q03.qg2.xml]

Because the book *The Talisman* is the only object with a PubDate before 1990, this query returns the same results as the earlier example.

> The general form of an attribute value condition is
>
> ```
> attribute operator value
> ```
>
> where
>
> - *`attribute`* is the name of an attribute for the corresponding database object or link
> - *`value`* is a legal value for *`attribute`*
> - *`operator`* is one of =, <>, <, <=, >, or >=.

Existence Conditions

Proximity supports a heterogeneous database structure, in which objects and links can have variable numbers of attributes, including none. For example, if a database contains information about movies, some movie objects might have a value for a Genre attribute while other movie objects might omit this attribute. QGraph lets you test whether an object or link has a value for a given attribute by using an *existence condition*.

The graph in Figure 3.5 represents part of a database containing information about books and authors. Because some authors write books under pseudonyms, we represent that information with a WrittenAs attribute on the link connecting authors and books they've written. As shown below, the author Ed McBain has written books both under his own name as well as under multiple pseudonyms.

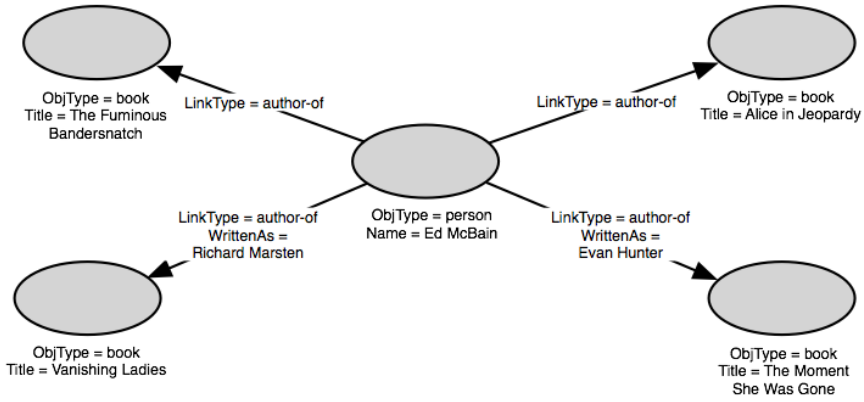


Figure 3.5. Database fragment [Cond_DB01.xml]

The query in Figure 3.6 finds author-book pairs where the author wrote the book under an assumed name. We use an existence condition to capture the fact that the link must have a WrittenAs attribute but that we do not care about its value.

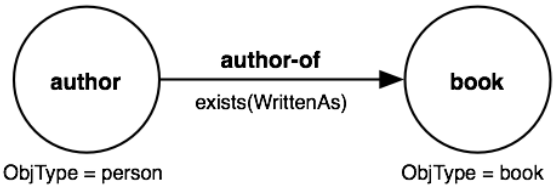


Figure 3.6. Query with an existence condition [Cond_DB01_Q01.qg2.xml]

The results of executing the query on this database fragment are shown in Figure 3.7.

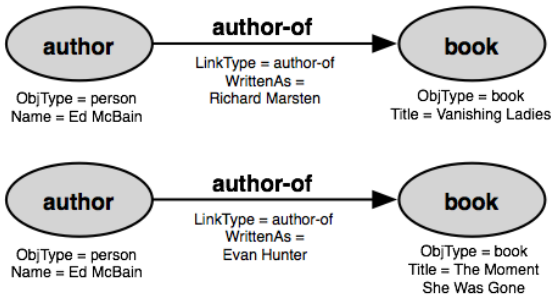


Figure 3.7. Query results

Existence conditions test only whether the matching database entity does or does not have the specified attribute. The attribute's value is not specified in the query and is not examined when evaluating the query. The query returns all matching subgraphs where the author-of link has a WrittenAs attribute, regardless of that attribute's value.

The general form of an existence condition is

`exists(`*`attribute`*`)`

where

`attribute` is the name of an attribute for the corresponding database object or link

Evaluating Conditions

QGraph is designed for use with databases that permit set-valued attributes. Therefore if an attribute has more than one value for an individual object or link, we must consider all of its values when evaluating conditions. And because the set of values may be empty, we also need to understand how QGraph evaluates attribute value conditions when the object or link has no values for the given attribute.

Evaluating attribute value conditions

A condition is satisfied when the value of the corresponding attribute satisfies the test established in the condition. When an object or link has more than one value for a particular attribute, a condition on that entity is satisfied if *any member* of the set of attribute values for that entity satisfies the condition.[1]

Consider a movie database that links actors to the movies they appear in. Each of these links includes a Role attribute that lists the characters the actor played in the movie. In *Monty Python and the Holy Grail*, Graham Chapman played several roles, so the edge connecting Graham Chapman to *Monty Python and the Holy Grail* has a Role attribute with multiple values: King Arthur, Voice of God, Middle Head and Hiccuping Guard. A relevant fragment of such a database is shown in Figure 3.8.

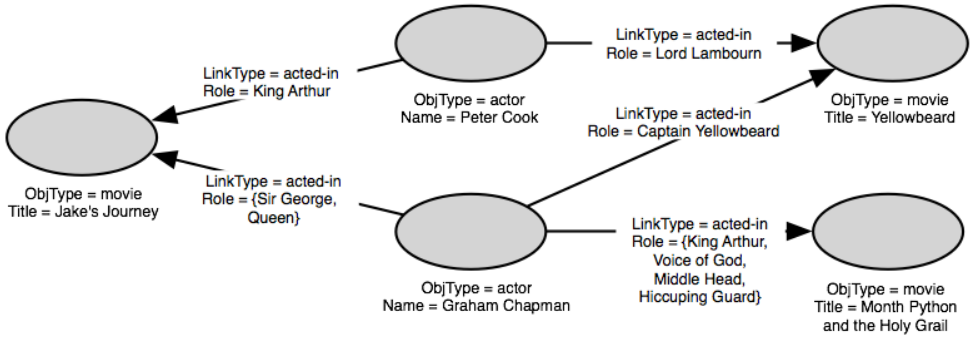


Figure 3.8. Database with multiple attribute values [Cond_DB02.xml]

To find actors and the movies in which they played King Arthur we create a query that requires an edge condition `Role = King Arthur`, as shown in Figure 3.9.

Figure 3.9. Query [Cond_DB02_Q01.qg2.xml]

The pair of Peter Cook and *Jake's Journey* is returned as a match because the link connecting them has a

[1]This reflects a change in the QGraph language since the publication of the technical report *A Visual Language for Querying and Updating Graphs* [Blau, Immerman, and Jensen, 2002]. The *QGraph Guide* provides the current and correct rules for how QGraph evaluates conditions on set-valued attributes.

Role attribute with a single value of King Arthur. The link connecting Graham Chapman to *Monty Python and the Holy Grail* has multiple values for the Role attribute. Because King Arthur is one of these values, this link also matches the condition on the query edge. The results of the query are shown in Figure 3.10.

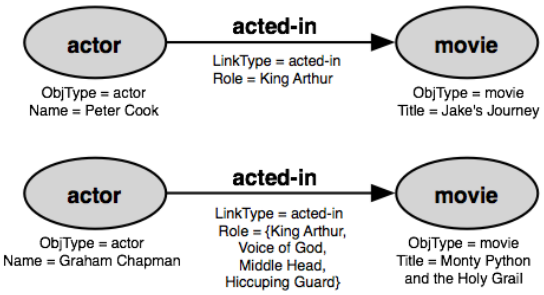


Figure 3.10. Results

As we have seen before, even though the condition only requires matching a single attribute value, the matching subgraph includes all values for the Role attribute.

The rule for evaluating attribute value conditions can be more formally expressed as:

Given

- a database entity (object or link) *E*
- an attribute *attribute*
- a set of values *Values* for *attribute* on *E*
- an attribute value condition *attribute operator value*

The condition is satisfied if there exists a value *v* a member of *Values*, such that *v operator value* is true.

Because QGraph considers the condition to be satisfied if any attribute value for the corresponding database entity satisfies the condition, care must be taken in defining queries and interpreting results when objects or links have multiple values for that attribute. For example, books are often released in different formats over time: hard cover, trade paperback, mass market paperback, etc. As a result, a single book can have multiple publication dates, as shown in Figure 3.11.

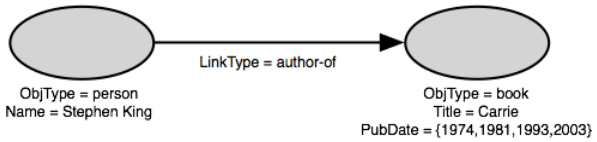


Figure 3.11. Book with multiple publication dates

Consider a condition that tests the publication date for a book, e.g.,

```
PubDate = 1981
```

Because 1981 is a member of the set of values for *Carrie*'s PubDate attribute, the condition is satisfied and *Carrie* matches a query vertex with this condition.

But because *Carrie*'s PubDate attribute contains other values, the condition

```
PubDate <> 1981
```

is also satisfied. The other values for this attribute (1974, 1993, and 2003) make the expression true. Similarly, the conditions

`PubDate < 1990`, `PubDate <= 1990`, `PubDate > 1990`, and `PubDate >= 1990`

are also satisfied because at least one PubDate value satisfies each of these tests.

Because objects and links in a Proximity database can be structurally heterogeneous, some entities may not have a value for the attribute listed in a condition. In such cases, an attribute condition's test always fails because there are no values to compare against the condition's requirements. Consider the database fragment shown in Figure 3.12.

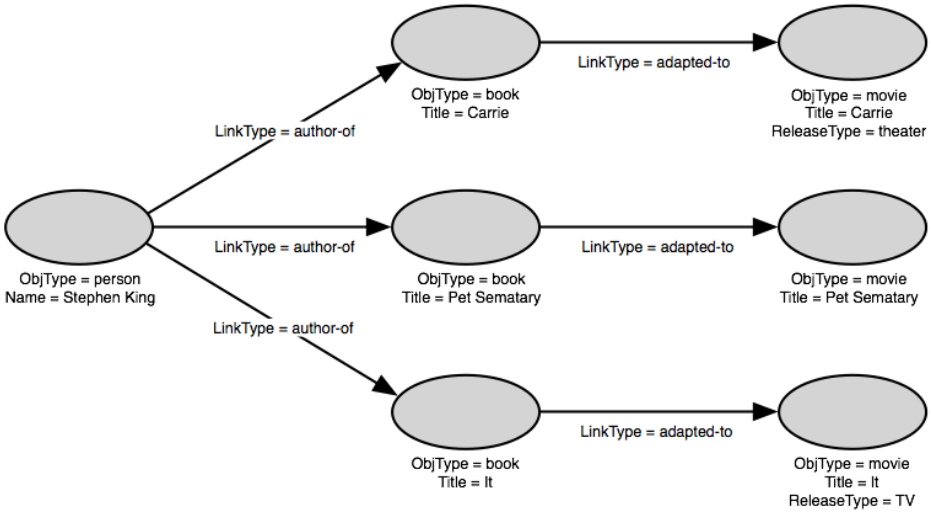


Figure 3.12. Books adapted to movies [Cond_DB03.xml]

Two of the movies, *Carrie* and *It*, have a ReleaseType attribute that indicates whether the movie was released in theaters or made for TV. The third movie, *Pet Sematary*, does not have a value for this attribute. If we execute the query shown below

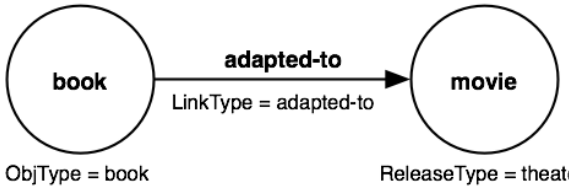


Figure 3.13. Query for books made into theatrical movies [Cond_DB03_Q01.qg2.xml]

we get the results shown in Figure 3.14.

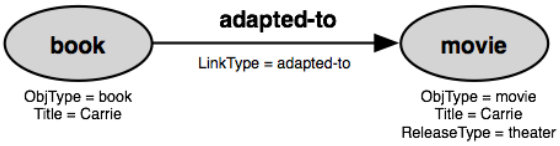


Figure 3.14. Results for books made into theatrical movies

The query finds only a single matching movie, *Carrie*, which was released theatrically. Both *It*, which was made into a TV movie, and *Pet Sematary*, which does not include a release type, do not match the query's condition.

Objects and links that do not have the specified attribute similarly fail to match inequality conditions. The query shown in Figure 3.15, below, also finds books made into theatrical movies. For the given database fragment, where the ReleaseType attribute can only take a single value drawn from the choices theater or TV, this query is equivalent to the query shown earlier in Figure 3.13.

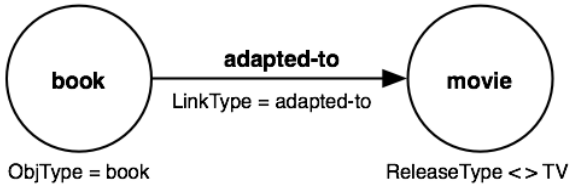


Figure 3.15. Alternate query for books made into theatrical movies [Cond_DB03_Q02.qg2.xml]

This new query yields the same results as the original. In addition to excluding the movie *It* because its value for the ReleaseType attribute is TV, it also excludes the movie *Pet Sematary* because this object has no value for the ReleaseType attribute. It's important to note that the equivalence of these two queries depends on the database enforcing the restriction of legal values for the ReleaseType attribute to those specified. If, for example, the database also lets ReleaseType assume values of video and DVD, this equivalence no longer holds.

Evaluating existence conditions

An existence condition is satisfied if the corresponding object or link has at least one value for the specified attribute. The condition fails if the database entity has no values for the attribute. The number of values (as long as it is more than zero) and actual attribute values have no impact on whether an existence condition is satisfied.

> The rule for evaluating existence conditions against set-valued attributes can be more formally expressed as:
>
> Given
>
> - a database entity (object or link) *E*
> - an attribute *attribute*
> - a set of values *Values* for *attribute* on *E*
> - an existence condition exists(*attribute*)
>
> The condition is satisfied if *Values* is not empty.

Complex Conditions

Sometimes you cannot capture the necessary query restrictions using a single condition. For example, suppose we have a database that contains information not just on authors and books, but that also includes other published materials such as magazine articles and screenplays. If we want to not just restrict matches to items published before 1990, but to also ensure that we only match book objects, we have to place two conditions on the *book* vertex to correctly restrict the matching objects:

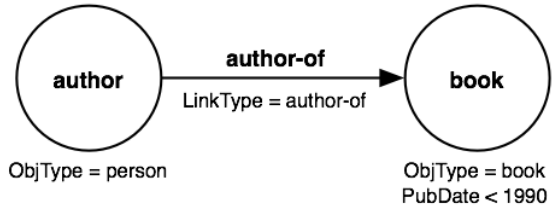


Figure 3.16. Using multiple conditions

Both of the conditions ObjType = book and PubDate < 1990 must be satisfied in order for the corresponding database object to match this vertex. Because both conditions must be satisfied, we can more correctly write this as the logical conjunction of two conditions, as shown in Figure 3.17.

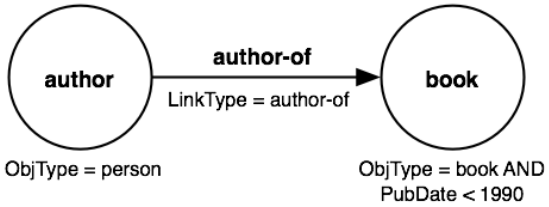


Figure 3.17. Complex condition using AND

In this example, the two conditions are combined using the logical AND operator. For clarity, query diagrams use the English words AND and OR instead of the symbols ∧ and ∨. In principle, complex conditions in QGraph can be any logical combination of simple conditions. For example, we could to modify the above query to find both books and screenplays published before 1990 by creating the following query. (However, see “Complex conditions in Proximity” (p. 21) for important Proximity limitations on the form of complex conditions.)

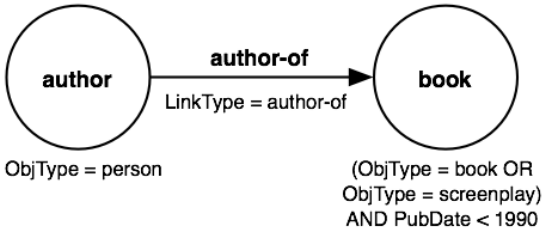


Figure 3.18. Example of a more complex condition

QGraph also permits the use of the logical NOT operator in complex conditions. The NOT operator has the expected effect of requiring that the condition’s test must fail for the corresponding database entity to match the query. For example, to find items not published in 1990, we can create the query

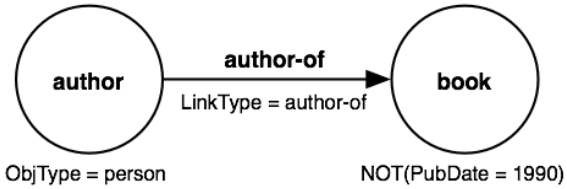


Figure 3.19. Example of a negated condition

This query matches objects whose PubDate attribute does not include the value 1990.

Remember that QGraph must consider all of an attribute’s values for an object or link when evaluating a condition. Therefore, when database items have multiple attribute values, we cannot assume that Attribute <> value is equivalent to NOT(Attribute = value). For example, in the query above, if an object’s PubDate attribute has the the value {1984, 1990}, then the condition PubDate <> 1990 will be satisfied because 1984 <> 1990. But the condition NOT(PubDate = 1990) will not be satisfied because 1990 is included in the value for PubDate.

Similarly, the two conditions PubDate <> 1990 and NOT(PubDate = 1990) do not yield the same results when applied to objects or links having *no* values for the specified attribute. In the case of an inequality condition, if an item has no PubDate attribute, no value exists to satisfy the inequality, therefore the condition cannot be satisfied. In the case of a negated condition, no value exists to satisfy the PubDate = 1990 equality, therefore the condition is satisfied.

Implementation in Proximity

Case sensitivity

Proximity makes some decisions about obeying or ignoring case sensitivity that have an impact on how you write conditions, specifically

- Proximity ignores case when matching attribute *names*.
- Proximity obeys case when matching attribute *values*.

Proximity converts all attribute names to lower case for its internal representation both when importing data and when evaluating conditions and constraints on a query. You can therefore ignore case when specifying the name of an attribute in a query's condition or constraint.

Proximity does not change the case of any attribute value, however. For example, you can write the condition `ObjType = Movie` with the attribute specified as objtype, Objtype, or OBJTYPE because all of these will be converted to the lower case form. But Proximity will not match objects with a value of movie for this attribute because the condition requires the capitalized value Movie to be a match.

Attribute values in Proximity

In addition to supporting set-valued attributes, Proximity also allows the use of multi-dimensional attributes, such as having both *x* and *y* values for an area attribute. Proximity currently supports the use of multi-dimensional attributes for data representation, import, and display, but does not yet support the use of multi-dimensional attributes in queries.

Proximity does not have a mechanism for assigning an attribute to a database entity without also assigning it a value. The statement that an object (or link) does not have an attribute is equivalent to stating that it does not have a value for that attribute.

Comparison operators

QGraph does not explicitly restrict the comparison operators allowed for use in query conditions. In general, a condition can be any boolean combination of restrictions on attribute values. However, Proximity's implementation of QGraph only supports the =, <>, <, <=, >, and >= operators in simple conditions.

Complex conditions in Proximity

Proximity requires that each query entity (vertex or edge) have only a single, possibly complex, condition. Proximity requires the use of disjunctive normal form for combining simple conditions into a single, complex condition.

Disjunctive normal form requires that logical expressions consist of a disjunction of conjunctions. That is, you can combine statements using `OR` as long as those statements only use `AND`. For example, to state the logical expression

```
(A OR B) AND C
```

in disjunctive normal form you would use

```
(A AND C) OR (B AND C).
```

The Proximity Query Editor requires prefix notation for complex conditions. The condition above must be entered as `OR (AND (A, C), AND (B, C))` where `A`, `B`, and `C`, are placeholders for simple conditions.

Efficiency considerations

The efficiency considerations described below are unlikely to have a noticeable effect on query

processing speed under normal circumstances. You may find some of these points helpful, however, if you need to use extremely complex queries, or execute the query against an extremely large database.

Redundant information in conditions

In many cases, databases contain redundant information. For example, an *acted-in* link might only be legally used to connect actors to movies. Thus if we require this link type in a query edge condition, we know that the linked objects will always be actors and movies. Given such limits on the how the database is structured, the queries shown in Figure 3.20 are equivalent.

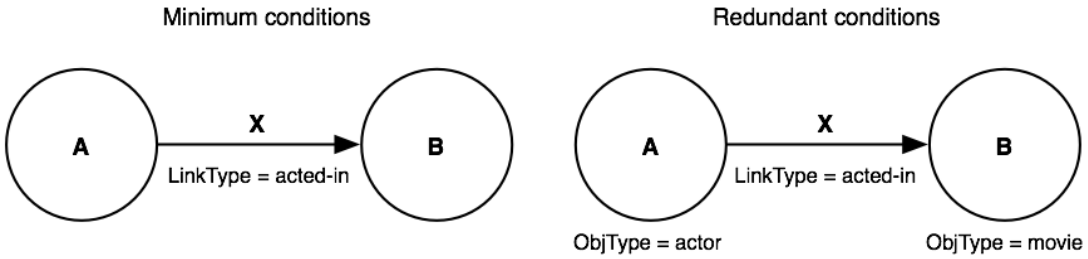


Figure 3.20. Potentially equivalent queries

Although both forms yield the same results, it is generally more efficient to include more rather than less information when adding conditions. The additional information prunes the candidate entities for the join needed to execute the query and thus reduces the number of items involved in the join.

In general, reducing the number of items involved in a join improves processing speed, so adding additional information in the form of conditions to your query is usually a good idea, even when such information doesn't change the anticipated query results. Even if you know that acted-in links only connect actors to movies, it's still worthwhile to include the conditions for these objects when query efficiency is a concern.

Existence conditions versus attribute value conditions

Checking for the existence of an attribute in Proximity is extremely efficient. Use existence conditions instead of attribute value conditions when they match the same data.

For example, suppose our database uses a Series attribute to indicate whether a book is part of a series, as shown in Figure 3.21.

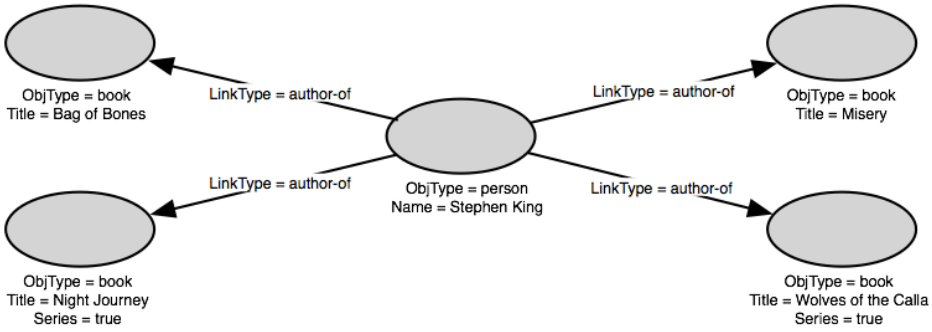


Figure 3.21. Database fragment [Cond_DB04.xml]

As shown in this example, *Wolves of the Calla* is part of the *Dark Tower* series, and *Night Journey* is part of the *Green Mile* series, so both have the attribute value Series = true. To find authors and their books that are part of a series, we might initially create the query shown in Figure 3.22.

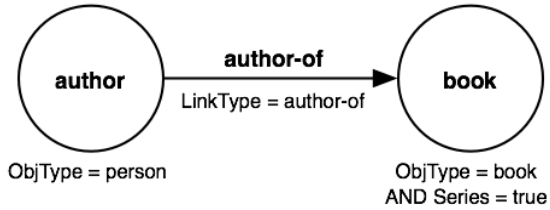


Figure 3.22. Query using attribute value condition [Cond_DB04_Q01.qg2.xml]

This typical query looks for book objects whose Series attribute has a value of true. But because of how this database is designed, all books that are part of a series have this same attribute value, while books that are not part of a series have no value for this attribute. This means that, for this database, the query above is equivalent to that shown in Figure 3.23, below.

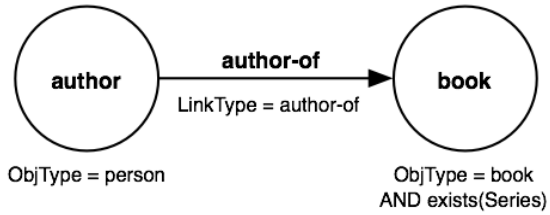


Figure 3.23. Query using existence condition [Cond_DB04_Q02.qg2.xml]

Because existence conditions are more efficiently processed than attribute value conditions, the second query is faster than the first, while yielding identical results.

It's important to note that the efficiency considerations described above are only applicable when choosing which of a set of equivalent queries to execute. Such a determination often relies on a thorough understanding of the target database. In practice, we are often faced with unknown or ill-defined database schemas making this kind of choice moot—we simply have to design a query that finds the matches we want without being able to take advantage of such efficiency considerations.

Summary

Conditions

- Conditions serve to restrict the number of query matches by placing requirements on the attributes of an object or link in the database.
- Conditions are commonly, but not exclusively, used to enforce type restrictions in query matches.
- Attribute value conditions restrict matches to only those database entities that match the attribute values specified in the query.
- Existence conditions test only to see whether the corresponding object or link has an instance of the specified attribute, but do not require that the attribute have any particular value.
- Attribute value conditions can use the comparison operators =, <, >, <=, >=, and <>
- Simple conditions can be combined using disjunctive normal form.
- An attribute value condition is satisfied if any member of the set of attribute values for the corresponding database entity satisfies the condition.
- If an object or link does not have a value for the attribute listed in an attribute value condition, the condition's expression is false and the condition fails (unless negated by `NOT`).
- The use of set-valued attributes means that different attribute values for a single object or link may satisfy different condition tests, yielding the result that `NOT(attr = value)` is not necessarily equivalent to `attr <> value`.

Case sensitivity

- Proximity ignores case when matching attribute names.
- Proximity obeys case when matching attribute values.

Efficiency considerations

- It is generally more efficient to include more rather than fewer restrictions in the form of conditions.
- Use existence conditions instead of attribute value conditions when they match the same data.
- Taking advantage of some of these efficiency improvements may require perfect information about a database's schema. In practice, we frequently do not have thorough enough information about a database to be able to use efficiency improvements with confidence.

Chapter 4. Numeric Annotations

The Need for Counting

The queries we've examined so far work fine when we know the exact structure of the subgraphs we want to find in the database. For example, if we want to find movies produced by two different studios, we create a query that includes two studio vertices, one for each studio credited with producing the movie, as shown in Figure 4.1

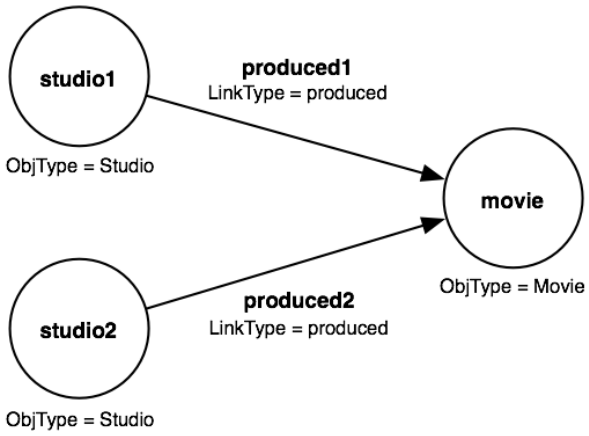


Figure 4.1. Movies produced by two studios [Annot_DB01_Q01.qg2.xml]

But this query has some problems. As we saw in Chapter 2, in addition to returning the desired subgraphs, this query's results will include subgraphs with duplicated elements, that is, with the same studio matching both the *studio1* and *studio2* vertices. And what if we want to instead find movies produced by two or more studios? We have to create separate queries for movies produced by three studios, by four studios, and so on. How high do we go? In many cases, we won't know the upper bound ahead of time. How can we create a query that finds all movies and their associated studios, without including duplicated elements, regardless of the number of studios involved?

Recall, as well, that the queries described so far return separate subgraphs for each match. Consider the author-book query shown in Figure 4.2.

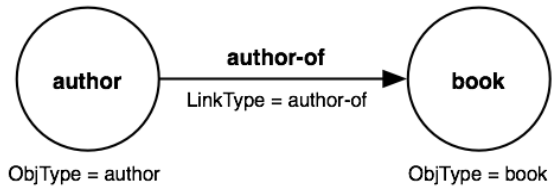


Figure 4.2. Simple author-book query

If our database contains 40 different books written by Stephen King, the query will return 40 different subgraphs, one for each author-book pair, even though all contain the same author. How can we create a query that collapses all the resulting subgraphs into a structure that more closely resembles the underlying structure of the data?

These cases are handled by *numeric annotations*. Numeric annotations place limits on the number of isomorphic structures that can occur in matching portions of the database. Limits can involve lower bounds, upper bounds, or both. Numeric annotations also serve to group isomorphic structures into a single subgraph that would otherwise produce multiple matches in the query results. QGraph does not provide any mechanism for limiting the number of matching substructures without grouping the results.

Annotation Basics

A numeric annotation specifies how many database entities must match a particular query element. Both vertices and edges may be annotated. (Subqueries are also annotated, as described in Chapter 6, *Subqueries*.) Numeric annotations can specify a *range* of values, giving them a great deal more flexibility than the alternative of specifying exact structural matches.

> Numeric annotations take the form
>
> **`[min..max]`**
>
> where
>
> - ***`min`*** specifies the minimum number of database elements required to match the query
> - ***`max`*** specifies the maximum number of database elements that can be present to match the query
>
> That is, to match the query the database must contain at least ***`min`*** and at most ***`max`*** of the annotated entity. The value for ***`max`*** can be left unspecified.

There are three legal forms of numeric annotation:

- An *unbounded range* [`i`..] on a vertex or edge means that at least `i` instances of the corresponding database element must be present to match the query. An unbounded range will match any number greater than or equal to `i` of database elements.
- A *bounded range* [`i`..`j`] means that at least `i` and no more than `j` instances are required for a match. The query will not match database structures that have fewer than or more than the specified number of elements.
- An *exact annotation* means that exactly the specified number of database elements must be present to match the query. For example, if you specify a vertex annotation of `[2]`, the query will not match database structures that have one, three, or more matching vertices.

We can use numeric annotations to restate the query with which we began this chapter, finding movies produced by exactly two studios:

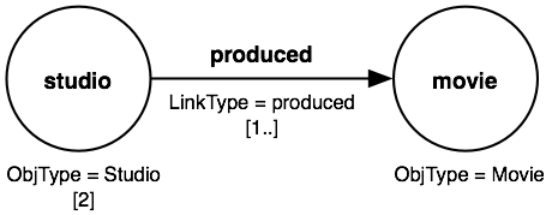


Figure 4.3. Movies produced by exactly two studios [Annot_DB01_Q02.qg2.xml]

Figure 4.3 includes two numeric annotations, one on the vertex representing studio objects, and one on the adjacent edge. The `[2]` annotation on the *studio* vertex indicates that the query can only match subgraphs containing exactly two studio objects. Of course, all the other parts of the query must also be satisfied—those two studio objects must be linked to the same actor object by produced links. This annotation also serves to group the two studio-movie pairs in a single subgraph with one movie object and two linked studio objects, rather than returning the multiple subgraphs we saw in Chapter 2.

The `[1..]` edge annotation is included because we cannot assume that linked objects in a database are connected by only a single link. If a database contains multiple links between objects, then we usually want to group these links, in addition to grouping the objects, in the query results. Because we may not know how many links connect one object to another, we use the unbounded annotation `[1..]` on the edge. For now, we'll note that this is usually the correct annotation for an adjacent edge and simply follow this convention in defining the next several queries. The section on "Understanding Multiple Annotations" later in this chapter provides a more complete explanation of this edge annotation.

The QGraph language includes specific requirements for annotated vertices and their adjacent edges.

- QGraph requires that edges adjacent to annotated vertices must themselves be annotated.
- QGraph requires that, at most, only one of two adjacent vertices may be annotated.

Edges adjacent to annotated vertices must be annotated for the reason cited above. Only one of two adjacent vertices may be annotated because annotating adjacent vertices can result in ambiguities in interpreting the query. Proximity enforces these requirements and will not execute queries with illegal annotations. See “Adjacency Requirements” (p. 42) later in this chapter for a more detailed explanation of the reasons behind these requirements.

When executing an annotated query, the vertex annotation takes precedence over the edge annotation. That is, the query processor first satisfies requirements on the vertex and then checks to see if it can satisfy requirements on the corresponding edges.

To see how Proximity handles the query shown in Figure 4.3, consider the database fragment shown in Figure 4.4. This fragment contains information about studios that produced some recent Academy Award winning pictures.

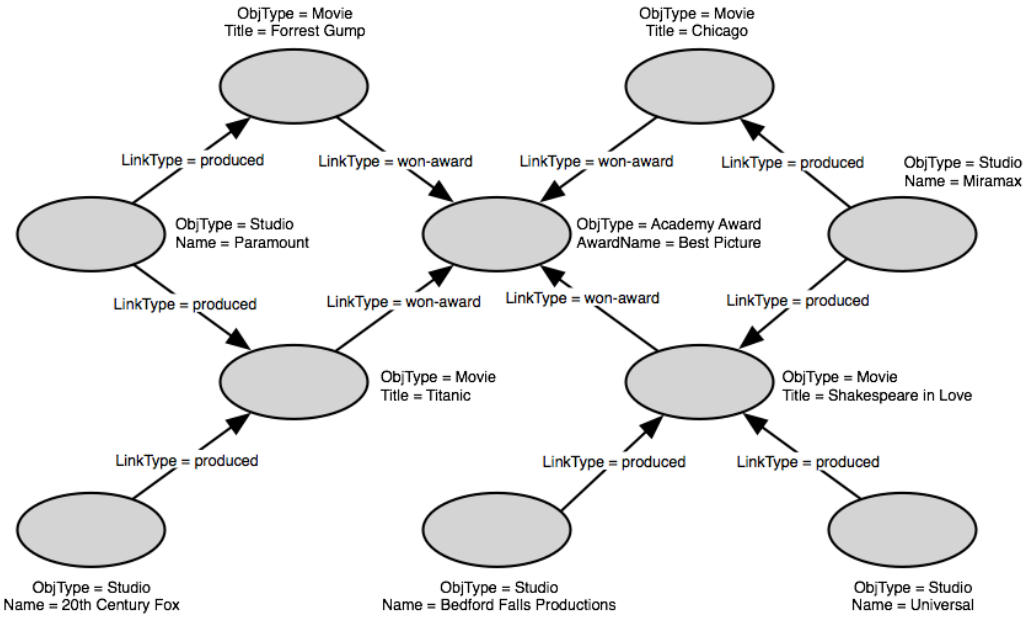


Figure 4.4. Database fragment [Annot_DB01.xml]

The above fragment includes four different movies: two produced by a single studio (*Forrest Gump* and *Chicago*), one produced by two studios (*Titanic*), and one produced by three studios (*Shakespeare in Love*). Executing the query shown in Figure 4.3 on this database fragment yields the matching subgraph shown in Figure 4.5.

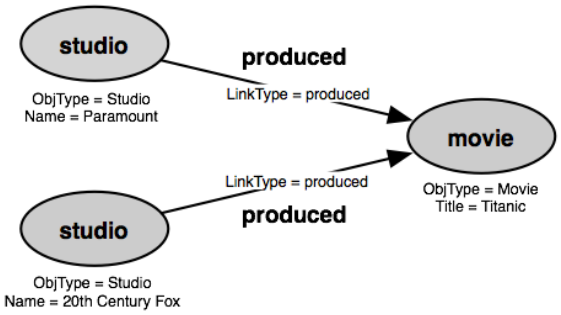


Figure 4.5. Query results

Rather than returning two subgraphs, each with one movie and one studio, this query returns a single subgraph containing the same data that would have been spread across multiple matches had we omitted the annotations from the query. Because we used an exact annotation of `[2]` on the *studio* vertex, the query does not match subgraphs containing movies connected to a single studio or to more than two studios. If we want to instead find all the movies produced by two or more studios, we need to change the numeric annotation on the *studio* vertex to use the unbounded range `[2..]`, as shown in Figure 4.6.

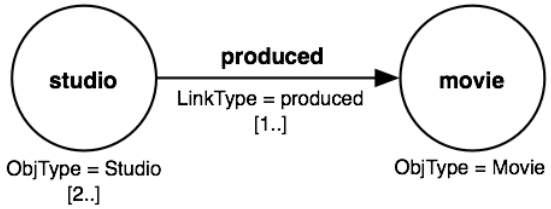


Figure 4.6. Movies produced by two or more studios [Annot_DB01_Q03.qg2.xml]

The results of executing this modified query on the data shown in Figure 4.4 are shown below:

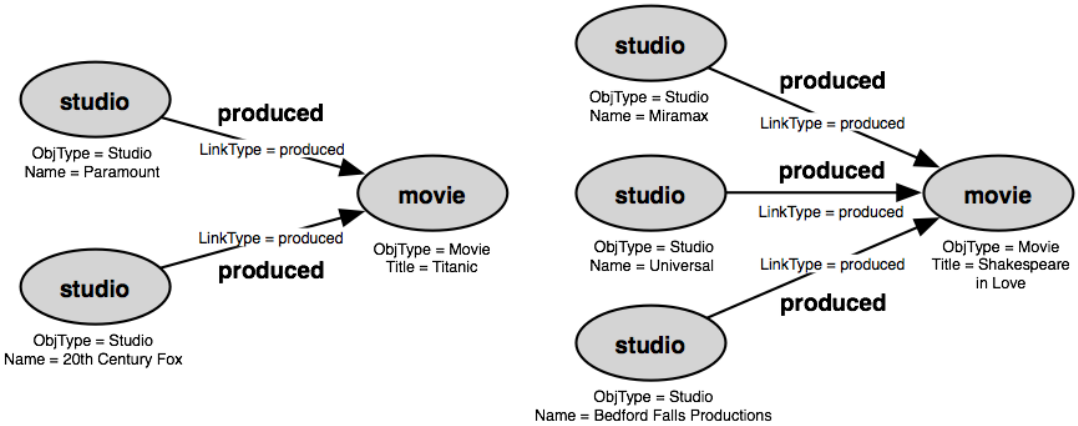


Figure 4.7. Query results

This time, the unbounded annotation `[2..]` on the *studio* vertex matches both the subgraph containing the two studios that produced *Titanic* and the subgraph containing the three studios that produced *Shakespeare in Love*.

A variation on this query structure forms one of the most common QGraph queries, the *star query*. Star queries find all database elements linked to a core object. Star queries are typically used to find subgraphs such as “all actors in a movie” or “all authors for a paper” (assuming the corresponding database contains the appropriate objects and links).

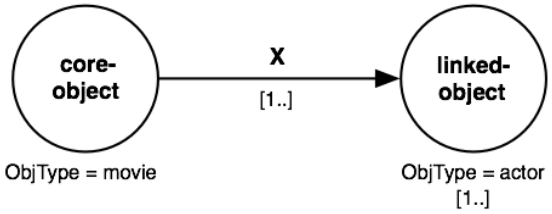


Figure 4.8. General star query

Star queries can use either directed or undirected edges.

To create a star query for the movie and studio database, we need to determine which type of objects should serve as the core vertex for the query. Because this database links multiple studios to a single movie, we make the core vertex match movie objects.

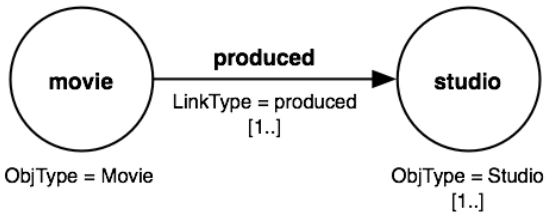


Figure 4.9. Star query with movies as core objects [Annot_DB01_Star.qg2.xml]

The query above finds and returns a subgraph for each movie in the database. Each subgraph includes all the actors linked from that movie. The results of executing this query on the database fragment in Figure 4.4 are shown below:

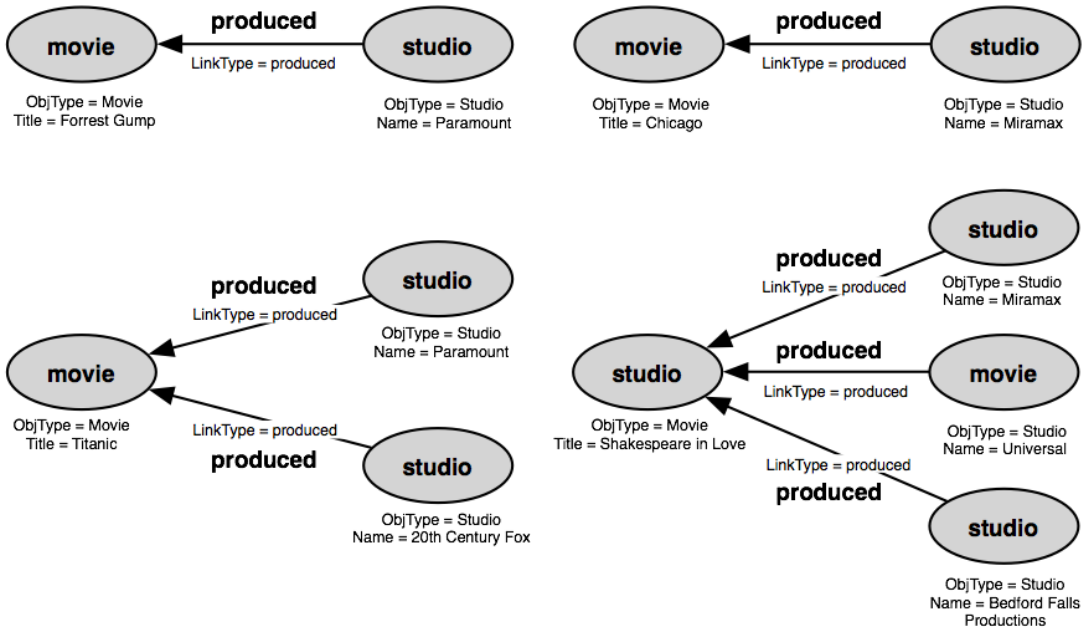


Figure 4.10. Query results

Just as we can annotate vertices so that they match more than one object, we can also annotate edges so that they match more than one link. For example, the database fragment shown in Figure 4.11 contains information on several actors and the roles they played in the movie *Angels in America*.

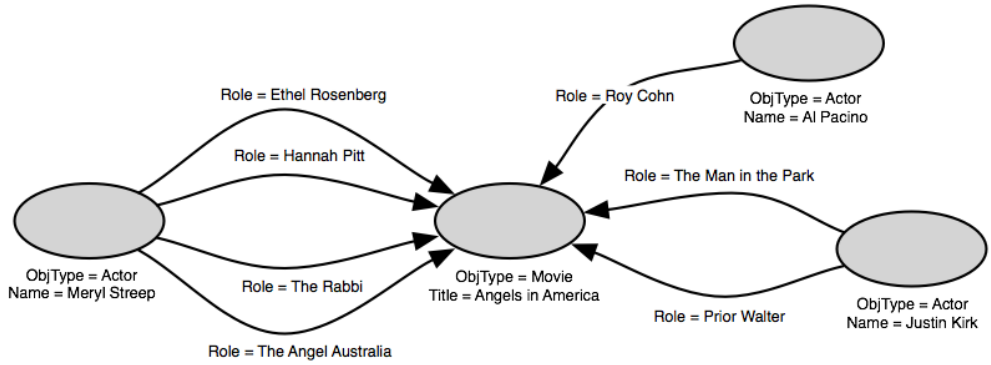


Figure 4.11. Database fragment [Annot_DB02.xml]

The database fragment indicates that Al Pacino played a single role, Justin Kirk played two different roles, and Meryl Streep played four different roles in this movie.

It's worth noting that the database fragment shown in Figure 4.11 uses a different schema to represent actors and roles from that used in Figure 3.8. The example in Chapter 3 used multiple attribute values on a single link to indicate that an actor played multiple roles in a movie. The example in this chapter uses multiple links to represent the same kind of information. Proximity does not requires any particular representational schema for a given dataset, although consistency within a dataset is important. You can determine the appropriate schema for your data.

A query that uses edge annotations to find actors playing multiple roles is shown in Figure 4.12.

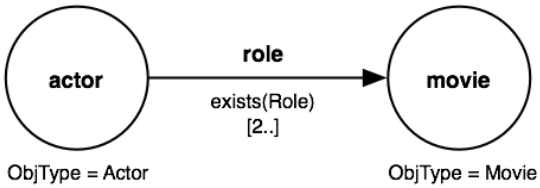


Figure 4.12. Actors playing multiple roles [Annot_DB02_Q01.qg2.xml]

Here we include the annotation `[2..]` on the edge connecting the *actor* vertex to the *movie* vertex, indicating that the query matches actor-movie pairs connected by two or more role links. Annotated edges can stand alone; they do not require that any adjacent vertices be annotated. The existence condition on the *role* edge requires that matching edges have a Role attribute, but doesn't place any requirements on the specific value of this attribute.

The results of executing this query on the database fragment shown in Figure 4.11 are shown below.

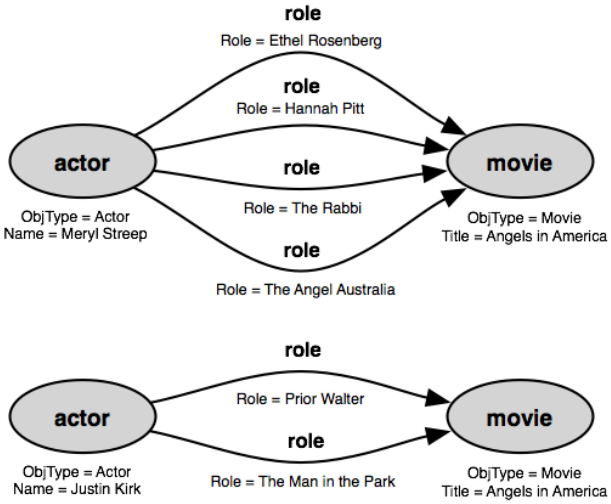


Figure 4.13. Query results

Just as vertex annotations group matching objects, the query's edge annotation groups matching links into a single subgraph in the query's results. Without the edge annotation, this query returns seven subgraphs—one for each unique actor-role-movie subgraph in the database.

An annotation of `[1]` is not equivalent to no annotation. A `[1]` annotation requires that the query only match subgraphs that contain exactly one of the annotated entities. A query with no annotation will match each appropriate database entity regardless of number, although it will not group the matches into a single subgraph. This can be seen by comparing the results for the two queries below.

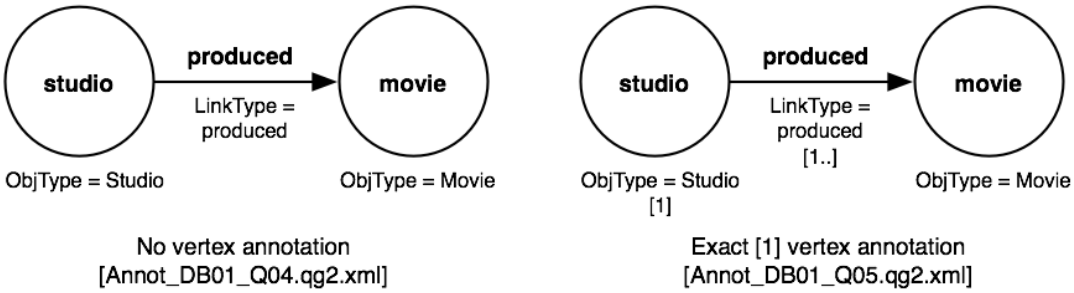


Figure 4.14. Annotated and unannotated queries

The query on the right includes an exact `[1]` annotation on the *studio* vertex (and the standard `[1..]` annotation on the incident edge to satisfy QGraph's adjacency requirements for annotations). The query on the left has no annotations. Executing these queries on the database fragment shown in Figure 4.4 yields distinctly different results. Figure 4.15 shows the results from the unannotated query.

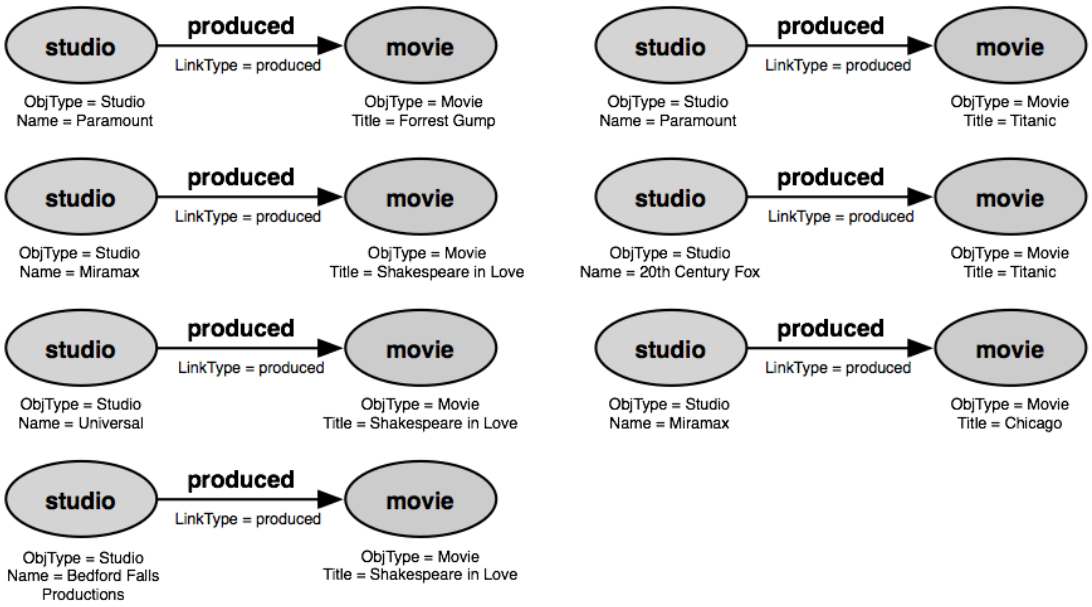


Figure 4.15. Query results

The query without annotations matches all the studio-movie pairs in the database. Studios are not grouped; each match forms a separate subgraph. Compare these results to that for the query containing the `[1]` vertex annotation.

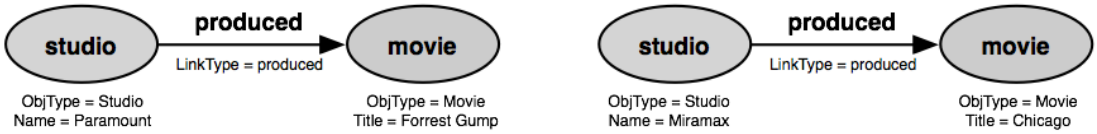


Figure 4.16. Query results

The results of executing the annotated query include just two subgraphs, matching the two instances in the database where a movie is linked to exactly one studio.

Understanding Multiple Annotations

It's important to understand how vertex and link annotations combine to create the desired query. This section examines several specific examples to see how these annotations work together.

Recall that the annotation `[1..]` means that the query matches if the database contains one or more of the annotated elements. We've seen the common usage of annotating a vertex and adjacent edge with this annotation to find simple clusters (stars) around a central object in the database. But what happens if one or both of these annotations has a different value?

We consider such queries in the context of finding clusters of actors and the movies they've appeared in. We know that actors usually appear in more than one movie, and that actors occasionally play multiple roles in a single movie. The database fragment in Figure 4.17 illustrates the type of data we might find in a target database for such queries. As we can see from this fragment, Peter Sellers, Alec Guinness, and Dennis Price have all played multiple roles in a single movie. Alec Guinness played three roles in *Kind Hearts and Coronets*, Dennis Price played two roles in *Kind Hearts and Coronets*, and Peter Sellers played multiple roles in two movies, *Dr. Strangelove* and *The Mouse That Roared*. Both Peter Sellers and Alec Guinness have also appeared in other movies where they played only a single role, but this fragment doesn't tell us whether Dennis Price ever did the same.

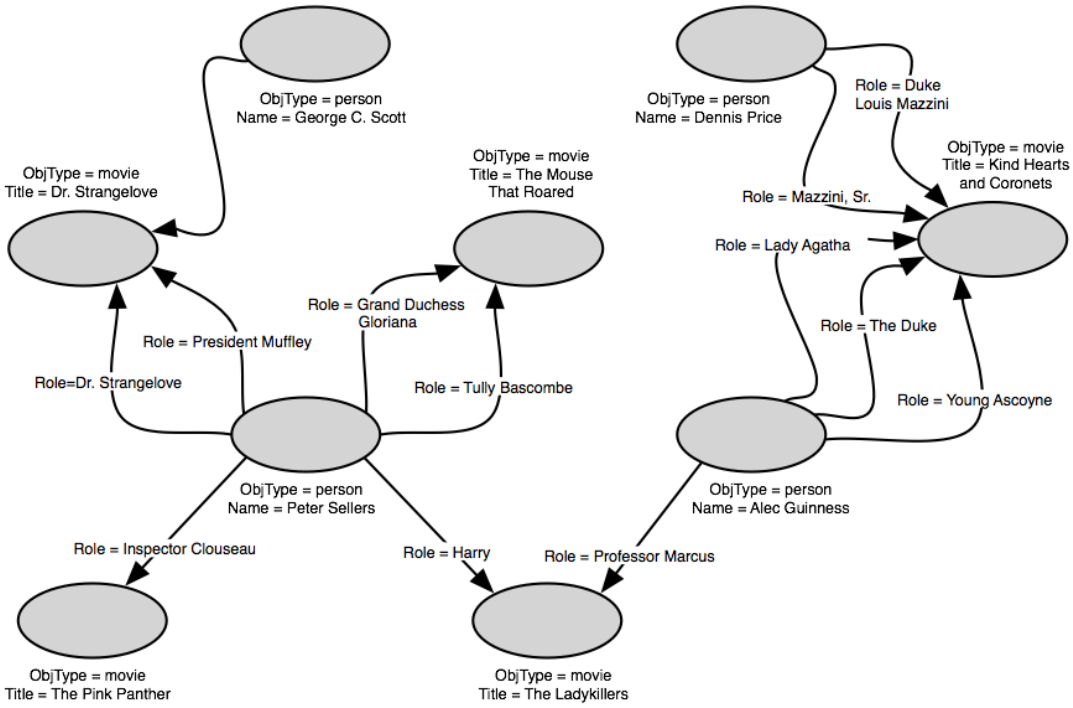


Figure 4.17. Database fragment [Annot_DB03.xml]

As we've seen before, we can use a query like that shown in Figure 4.18 to find actors connected to at least two movies.

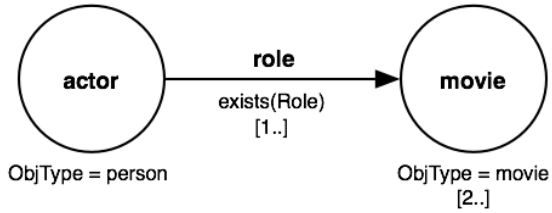


Figure 4.18. Requiring multiple object matches [Annot_DB03_Q01.qg2.xml]

The vertex annotation of `[2..]` tells the query to match clusters with actors as core objects, where those actor objects are linked to at least two movies. Our use of the standard `[1..]` edge annotation groups multiple links connecting a particular actor and movie in the same subgraph, as shown in Figure 4.19.

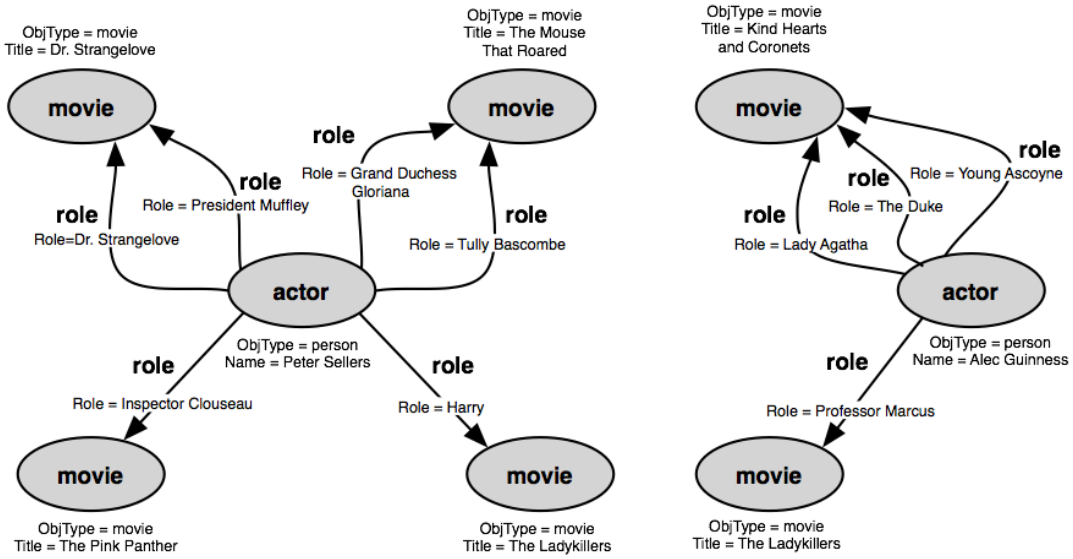


Figure 4.19. Query results

Two subgraphs are returned, one with Peter Sellers as the core actor object and one with Alec Guinness as the core actor. Both of these actors are linked to two or more movies. The subgraphs surrounding the actors George C. Scott and Dennis Price are not included in the query results because the actor is only linked to a single movie in both of these instances.

In contrast, if we reverse these annotations to create the query shown in Figure 4.20, we now must find subgraphs where each movie is connected to an actor by at least two role links.

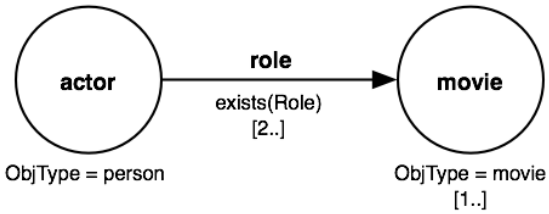


Figure 4.20. Requiring multiple link matches [Annot_DB03_Q02.qg2.xml]

This query identifies subgraphs containing an actor and any movies in which that actor played at least two roles. Any movies in which the actor played only a single role are not included in the query results, shown in Figure 4.21. The `[1..]` annotation on the *movie* vertex groups the matching movies for a specific actor into a single subgraph.

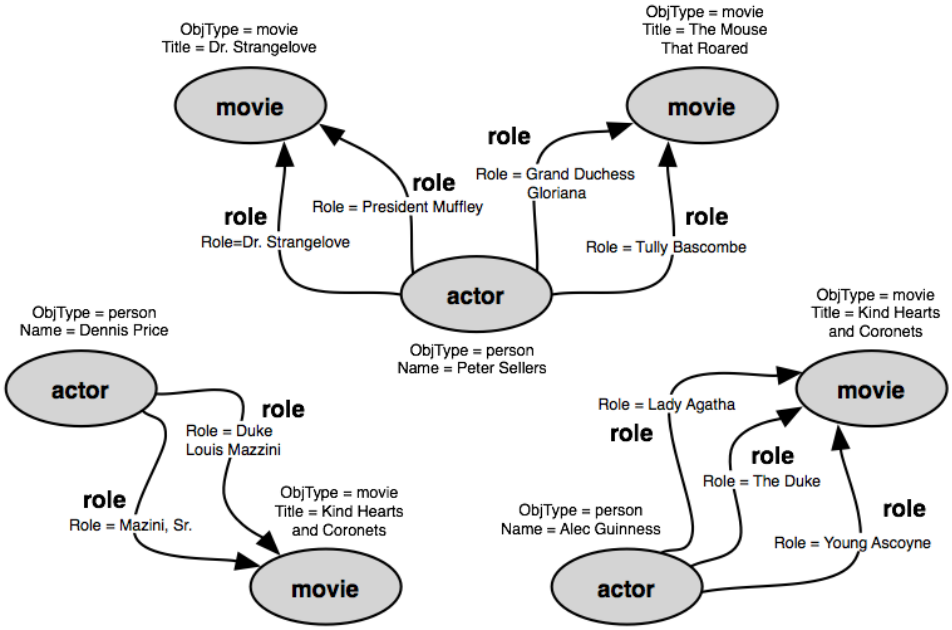


Figure 4.21. Query results

The new annotation results in the elimination of the movie *The Pink Panther* and *The Ladykillers* from the subgraph containing Peter Sellers, and the elimination of *The Ladykillers* from the subgraph containing Alec Guinness. The actors played only one role in these movies. And because he played two roles in *Kind Hearts and Coronets*, the subgraph containing Dennis Price is now included in the results. This subgraph was omitted from the first query's results because the first query required that each actor be linked to two or more movies but this database only includes information about one movie in which Dennis Price acted.

If we change the query again to include a numeric annotation of `[2..]` for both the *movie* vertex and its adjacent edge, as shown in Figure 4.22, then matching subgraphs must include at least two movie objects connected to an actor object, and each of those movie objects must be connected to their corresponding actor by at least two distinct role links.

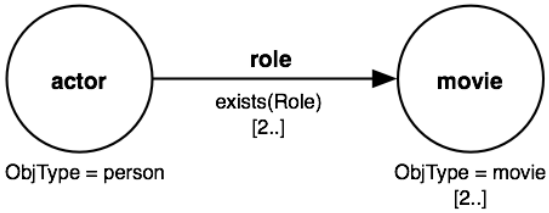


Figure 4.22. Requiring multiple object and link matches [Annot_DB03_Q03.qg2.xml]

As we can see in Figure 4.23, this query finds only a single subgraph that satisfies the query's requirements. Only one actor in this database fragment is linked to at least two movies in which he played two or more roles.

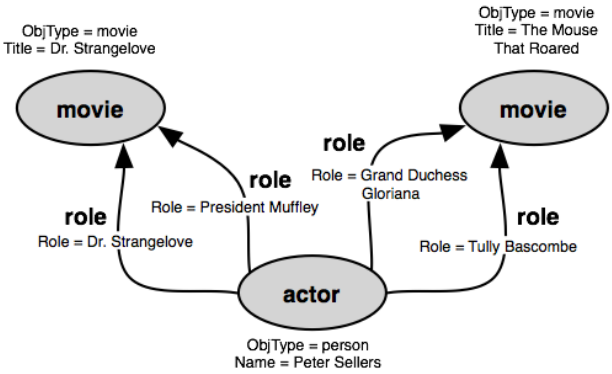


Figure 4.23. Query results

Negated and Optional Elements

Numeric annotations can also be used to indicate that a database element must not or need not be present to match the query.

- *Negated elements* must not be present in the matching subgraph. Negated elements are specified by the numeric annotation `[0]`, indicating that exactly zero database elements can match this query element.
- *Optional elements* may be but are not required to be present in the matching subgraph. Optional elements are specified by the numeric annotation `[0..]` or `[0..j]`, indicating that zero or more database elements can match this query element.

To be well formed, a query must remain a connected graph after removing any negated or optional elements.

The database fragment shown in Figure 4.24 includes information on movies and any awards they have won.

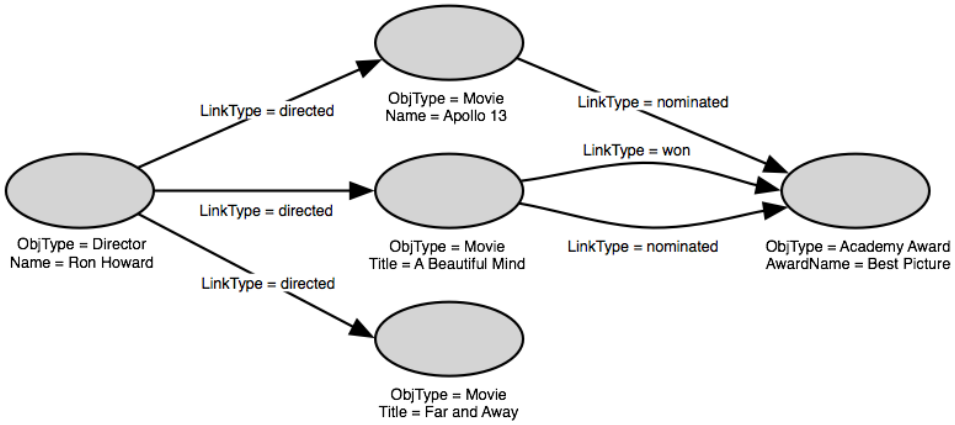


Figure 4.24. Database fragment [Annot_DB04.xml]

This fragment contains information on three movies, all directed by Ron Howard. *A Beautiful Mind* was nominated for and won the Academy Award for Best Picture for 2002; *Apollo 13* was nominated for the Academy Award for Best Picture for 1995, but the Oscar was awarded to *Forrest Gump* that year; and *Far and Away* was not nominated for any Academy Awards.

Suppose we want to find all movies that have *not* been nominated for an Academy Award. One way to do so is to only match subgraphs that do not include any Academy Award objects.

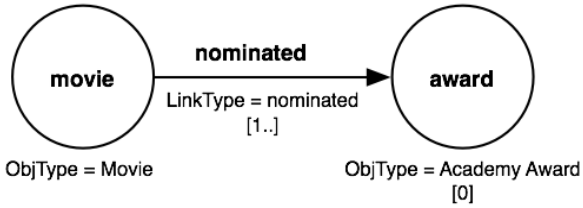


Figure 4.25. Query [Annot_DB04_Q01.qg2.xml]

The `[0]` annotation on the *award* object means that this query matches only those movies that are not linked to any Academy Awards. For now we'll simply note that the `[1..]` edge annotation is usually the correct annotation for edges adjacent to negated or optional vertices and proceed accordingly. Additional information on choosing the correct annotation for edges adjacent to negated or optional vertices is included in the section "Annotating edges adjacent to negative and optional vertices" later in this chapter.

The result of executing this query on the database fragment shown in Figure 4.24 is shown below.

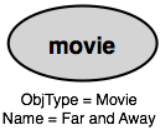


Figure 4.26. Query results

Only a single movie, *Far and Away*, matches the query. Note that negated query elements do not appear in the query results. Because the query includes a `[0]` annotation on the *award* vertex, there can be no corresponding object in order to match the query, and therefore no award objects can appear in the matching subgraphs.

Compare this query to a similar query where we don't care whether the movie has been nominated for an Academy Award.

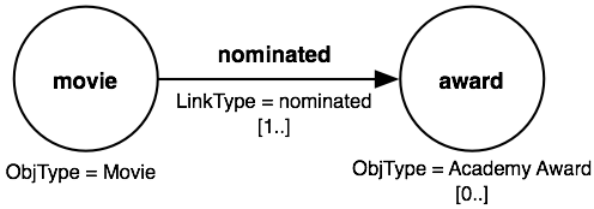


Figure 4.27. Query [Annot_DB04_Q02.qg2.xml]

In this example, the *award* vertex is annotated with `[0..]`, making matches to this vertex optional. That is, subgraphs in the database will match this query regardless of whether or not the corresponding movie has been nominated for an Academy Award. The results of executing the new query with the optional *award* vertex are shown below.

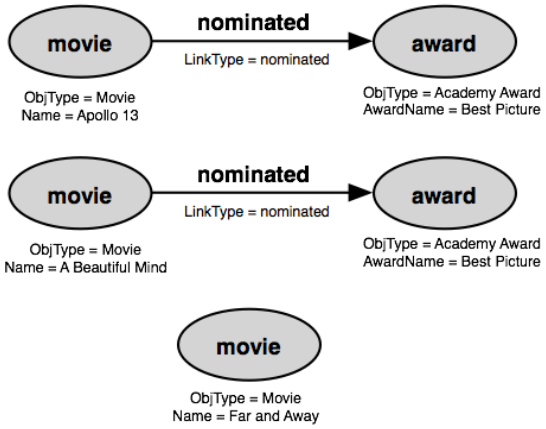


Figure 4.28. Query results

This time, all three movies appear in the query results. The optional *award* vertex lets the query match both movies that have received nominations and movies that have not received any nominations. Because *Far and Away* has no nominations, its subgraph does not include any award objects.

Although not yet supported by Proximity, QGraph queries can include negated and optional edges in addition to negated and optional vertices. Care must be taken to avoid disconnected queries; queries with negated or optional edges must also include required alternate paths to all query elements.

Figure 4.29 illustrates a query that includes a negated edge. This query finds movies that have been nominated for, but not won an Academy Award. The *won* edge is annotated with `[0]` indicating that matching subgraphs must not include a matching link.

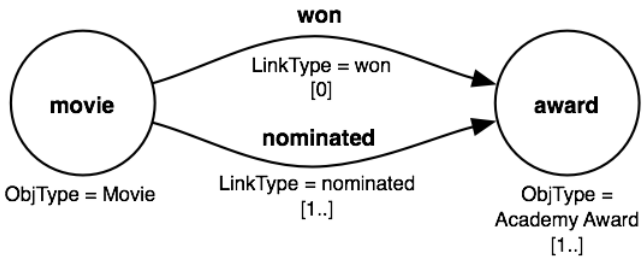


Figure 4.29. Query

To prevent the query from becoming disconnected, the *nominated* edge is required. The `[1..]` edge annotation means that the query only matches subgraphs that contain one or more nominated links connecting the movie to any Academy Award objects.

The expected results, had we been able to execute this query in Proximity, are shown below.

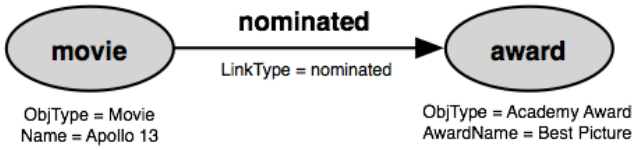


Figure 4.30. Query results

Only one movie in our database fragment, *Apollo 13* has been nominated for but not won an Academy Award.

Annotating edges adjacent to negative and optional vertices

Because edges adjacent to annotated vertices must themselves be annotated, how are we to annotate the edge adjacent to a negated or optional vertex? In general, the usual annotation of `[1..]` remains appropriate. We require the presence of the query edge to avoid having the query become disconnected. For example, an optional edge connected to an optional vertex could result in a disconnected subgraph when the matching vertex is present but the matching edge is not. Optional or negated edges are usually inappropriate in most queries, although they may be safely used if other elements in the query ensure that the query remains connected after removing any optional or negated elements.

For example, consider the database fragment shown in Figure 4.31.

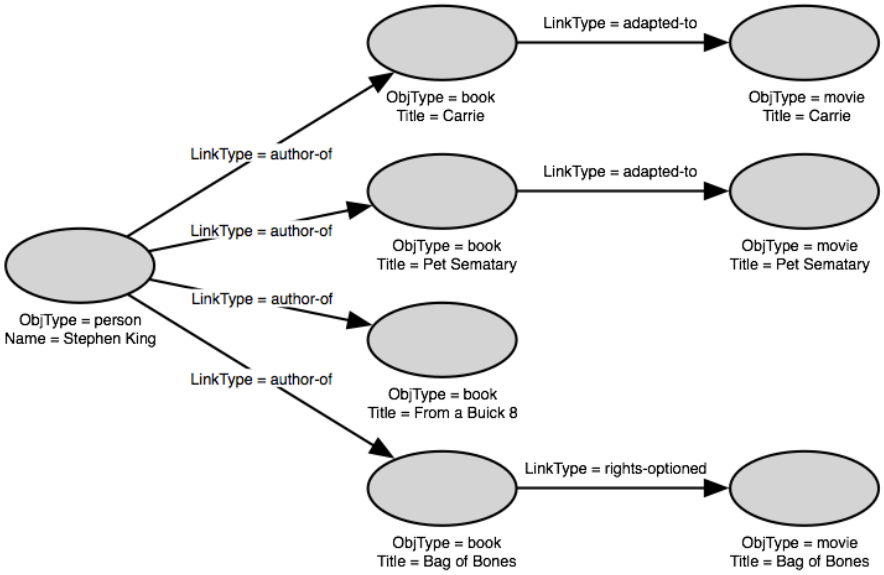


Figure 4.31. Stephen King books and movies

If we wanted to identify author-book pairs and also include movie adaptations, if any, we might initially (and incorrectly) create a query that looks like that shown in Figure 4.32.

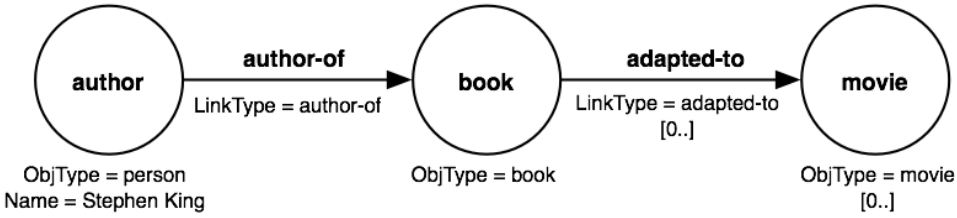


Figure 4.32. Improperly annotated query

This query uses the improper annotation `[0..]` for both the *movie* vertex and adjacent *adapted-to* edge. Such a query can match disconnected subgraphs. Consider the highlighted subset of elements from this database shown in Figure 4.33.

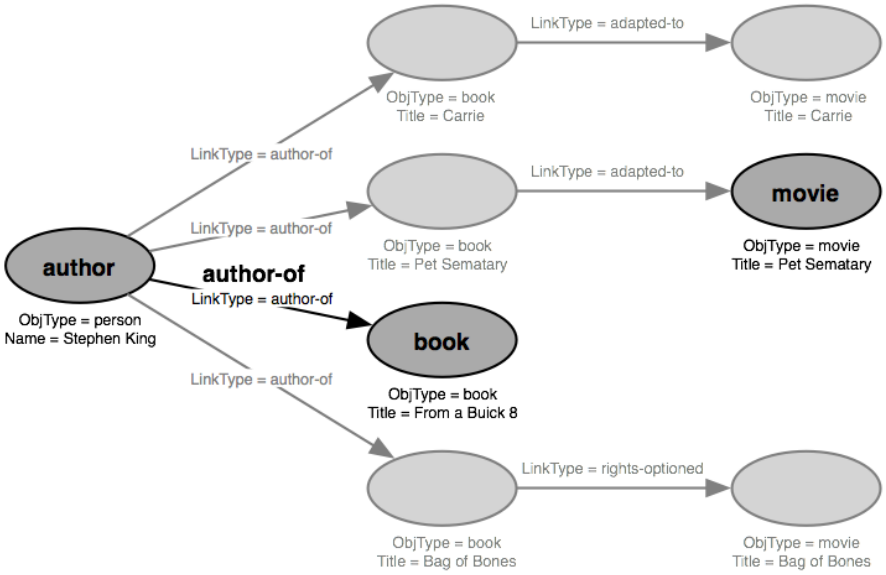


Figure 4.33. Disconnected subgraph

The highlighted set of objects and links matches the query shown in Figure 4.32. The `[0..]` annotation on the *movie* vertex and `[0..]` annotation on the *adapted-to* edge mean that the query is allowed to match database structures that include a matching movie object but that omit the matching adapted-to link. This can produce disconnected subgraphs like the highlighted elements shown above, which clearly isn't what was intended by the query. To prevent the possibility of matching such disconnected subgraphs, use the `[1..]` annotation on edges adjacent to a vertex annotated with `[0]` or `[0..]`.

Negated elements versus inequality conditions

Although they represent conceptually similar concepts, negated elements cannot substitute for conditions that express inequalities. Each has a different effect on how the query matches the database. Consider the database fragment shown in Figure 4.34.

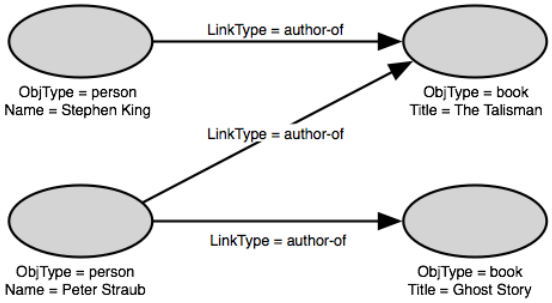


Figure 4.34. Database fragment [Annot_DB05.xml]

If we want to find books not written by Stephen King, we have at least two options, depending on our intent. As our first option, the query in Figure 4.35 seems to satisfy this goal.

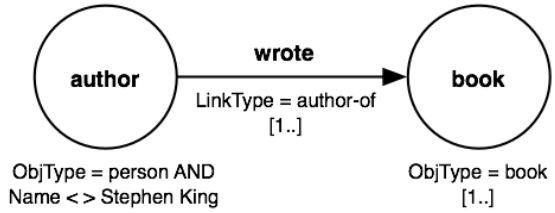


Figure 4.35. Query using inequality condition [Annot_DB05_Q01.qg2.xml]

The inequality in the condition on the *author* vertex, Name <> Stephen King, specifies that only books linked to persons that are not Stephen King are to be included in the query results. The results of running this query on the database fragment in Figure 4.34 are shown below:

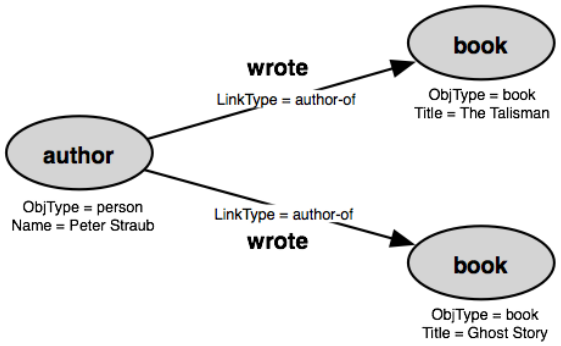


Figure 4.36. Query results

The results of this query include both person and book objects. As specified in the query, we only match books linked to person objects with a Name attribute not equal to Stephen King. To be more precise, because QGraph works with set-valued attributes, the query's *author* vertex matches person objects if their Name attribute includes at least one value other than Stephen King. The query uses numeric annotations to group the results into a single subgraph, so both books written by Peter Straub are included in the subgraph.

The query shown in Figure 4.37 also appears to satisfy our goal of finding books not written by Stephen King.

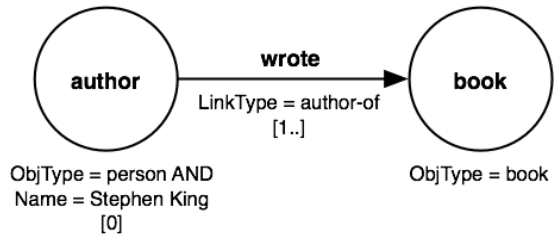


Figure 4.37. Query using inequality condition [Annot_DB05_Q02.qg2.xml]

In this case, the negated *author* vertex specifies that we can only match book objects that are not connected to a person named Stephen King. This is different than matching books connected to people not named Stephen King, as seen in the results of the second query, below.

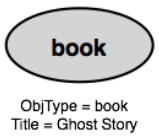


Figure 4.38. Query results

Because Stephen King is one of the authors of *The Talisman*, this book is not included in the query results. *The Talisman* is included in the results of the first query because it is connected to a second author, satisfying the inequality condition. Also, remember that negated elements are not included in query results, therefore the second query's results do not include an author object (and associated links).

Both queries can be viewed as satisfying our goal of identifying books not written by Stephen King. Each makes us aware of an unstated assumption in this initial goal—do we want books written exclusively by someone else or books that are not solely written by Stephen King? And each query also has a different impact on the form of the resulting subgraphs. The inequality condition must still match an object in the database so the corresponding objects appear in the results. But a negated element cannot match a database entity and therefore there can be no corresponding element in the query results.

Adjacency Requirements

We saw before that an edge adjacent to an annotated vertex must itself be annotated. To illustrate the reason for this rule, let's examine what happens without such a requirement. Figure 4.39 shows a part of the database shown in Figure 4.17. This fragment includes multiple links from an actor to a single movie, representing Alec Guinness's multiple roles in *Kind Hearts and Coronets*.

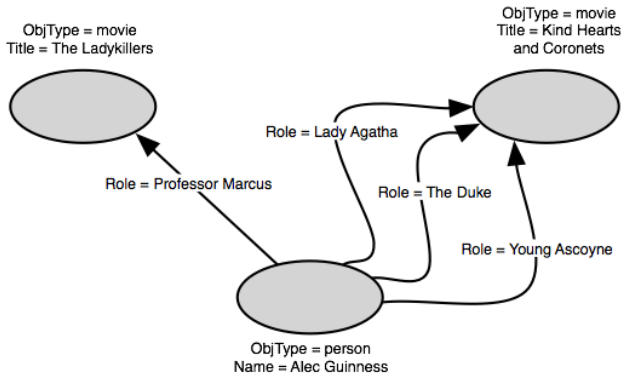


Figure 4.39. Database fragment [Annot_DB06.xml]

The (illegal) query shown in Figure 4.40 finds actor-movie pairs but omits the required edge annotation.

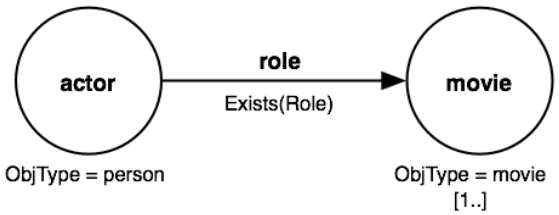


Figure 4.40. Illegal query omitting adjacent edge annotation

How are we to interpret this query? One interpretation would be to retain the ability of numeric annotations to collapse the subgraphs containing the same actor and different movies but let each *role* edge define a distinct match, much as we saw in Chapter 2. If we could execute such a query on the database fragment in Figure 4.39, we would see the results shown in Figure 4.41.

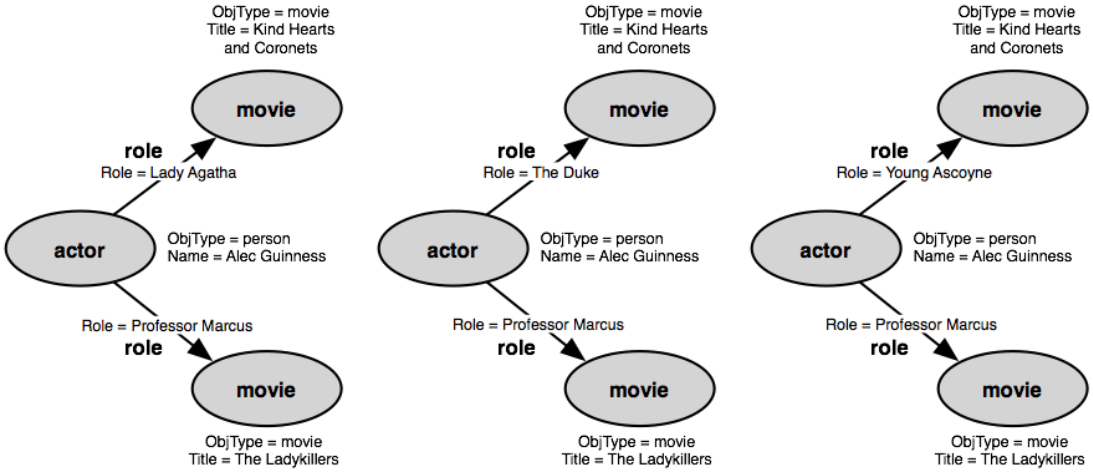


Figure 4.41. Hypothetical results for improperly annotated query

In this interpretation, the unannotated edge results in three different matching subgraphs instead of one subgraph that groups the three links. Each contains the same objects, but they differ in the link connecting the actor Alec Guinness with the movie *Kind Hearts and Coronets*. But annotating the edge as shown in Figure 4.42

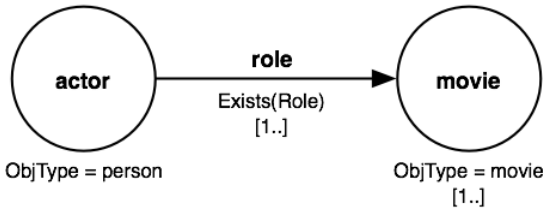


Figure 4.42. Legal query including adjacent edge annotation [Annot_DB06_Q02.qg2.xml]

collapses these three subgraphs into the single subgraph shown below:

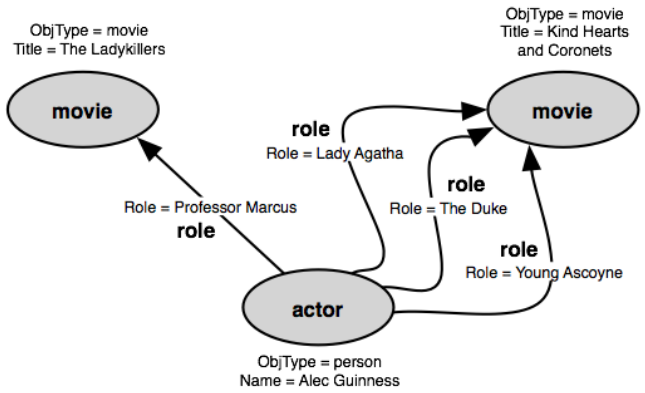


Figure 4.43. Results for properly annotated query

This subgraph best corresponds to our intuitive understanding of the query as identifying actor-movie clusters, regardless of how many roles the actor played in each movie.

QGraph also requires that, at most, only one of two adjacent vertices may be annotated. Another way of stating this is that there can be no edge connecting two annotated vertices in a query. To see why this is the case, consider the database fragment and (illegal) query shown in Figure 4.44.

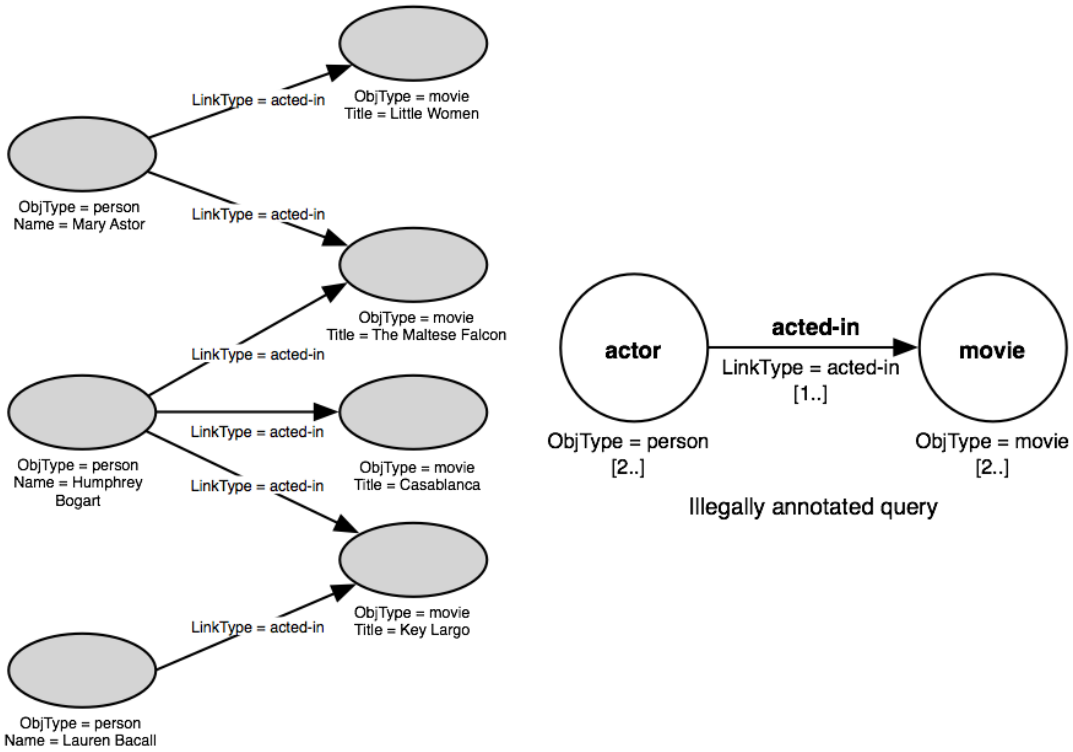


Figure 4.44. Database fragment and illegal query

How are we to match the query against this database fragment? We have many instances where an actor is connected to two or more movies, and where a movie is connected to two or more actors, but matching these database elements to the query is far from straightforward.

Can we include the actor Mary Astor in the query results? She is connected to two movies, satisfying the annotation on the *movie* vertex, but those movies must also be part of a subgraph that satisfies the query's other conditions and annotations, thus each movie she is connected to must be connected to two or more actors (one of which can be Mary Astor).

One of the movies linked to Mary Astor, *Little Women*, is only linked to a single actor, Mary Astor herself, and therefore does not satisfy the *actor* vertex annotation, so it does not match the query's requirements. And if we cannot include *Little Women*, then Mary Astor cannot be included because she no longer links to two movies that satisfy the query.

A similar situation arises with respect to the movie *The Maltese Falcon*. At first glance, it appears to satisfy the query—it is linked to two actors, and each of those actors is linked to at least two movies. But as we've seen, at least one of those actors, Mary Astor, does not satisfy the query's requirements, so it cannot be included, and therefore *The Maltese Falcon* also cannot be included in the query's results.

This query is ambiguous because neither vertex annotation takes precedence over the other. In effect, we don't know where to start when evaluating the query's match to a specific part of the database, and the choice of a starting point effects how we determine whether an object in the database satisfies the requirements of the query. To avoid this ambiguity of interpretation, QGraph requires that at most one of two adjacent vertices be annotated.

QGraph also places additional restrictions in numeric annotations when used in queries that include constraints (Chapter 5, *Constraints*) or subqueries (Chapter 6, *Subqueries*). These rules are discussed in later chapters.

Implementation in Proximity

Implementation restrictions

Proximity places an additional implementation restriction on the use of numeric annotations in a query. Proximity cannot process queries in which an annotated vertex is connected to more than one unannotated vertex. For example, the query shown below is a legal QGraph query, but it cannot currently be executed in Proximity.

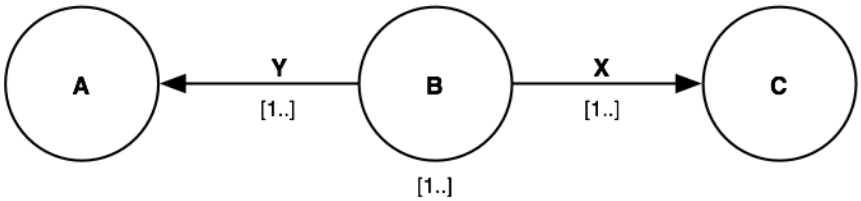


Figure 4.45. Unimplemented query

Proximity currently prohibits edge annotations with a lower bound of `0`. Query edges may not be annotated with `[0]`, `[0..]`, or `[0..`*`j`*`]`. This prohibition applies regardless of whether the edge annotation is used alone, or is applied to an edge adjacent to an annotated vertex.

Efficiency considerations

For some database schemas, annotations may duplicate other query requirements. For example, consider a database containing information about movies and their DVD releases. If we want to find movie-DVD subgraphs we might normally create a query like that shown in Figure 4.46.

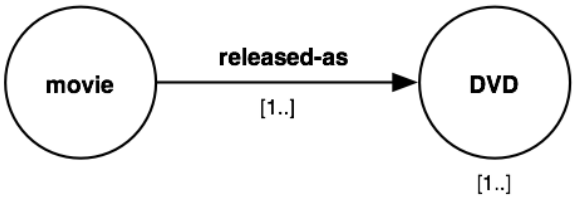


Figure 4.46. Annotated query

If the target database always contains exactly one link from a movie object to its corresponding DVD object, then annotating the *DVD* vertex and adjacent link will have no effect on the query's matches. The annotation will, however, require additional computation and is therefore less efficient than the equivalent unannotated query, shown below.

Figure 4.47. Potentially equivalent unannotated query

Importantly, the second query is only equivalent if the database enforces the link restrictions described above. If we later add additional objects or links for subsequent DVD releases, the annotated and unannotated queries will no longer match the same set of subgraphs.

In most cases, the difference in processing speed will not be of concern, but such efficiency modifications may be worthwhile for complex queries or queries run on extremely large databases. Taking advantage of such efficiency improvements can require perfect knowledge of the corresponding database schema and data, however, and thus may not be practical in many situations.

Summary

Annotation uses

- Numeric annotations serve to place limits on the number of isomorphic structures that can occur in matching portions of the database.
- Numeric annotations also serve to group isomorphic structures that would otherwise produce multiple matches into a single subgraph.
- QGraph does not provide any mechanism for limiting the number of matching substructures without grouping the results.
- A numeric annotation can be specified on a vertex or edge in a QGraph query.

Annotation format

- An unbounded range [*i*..] on a vertex or edge means that at least *i* instances of the corresponding database element must be present to match the query. An unbounded range will match any number greater than or equal to *i* of database elements.
- A bounded range [*i*..*j*] means that at least *i* and no more than *j* instances are required for a match. The query will not match database structures that have fewer than *i* or more than *j* of the annotated element.
- An exact annotation [*i*] means that exactly the specified number of database elements must be present to match the query. The query will not match database structures that have fewer than or more than the specified number of elements.
- To group substructures without limit, we use the annotation `[1..]`.

Adjacency requirements

- Edges adjacent to annotated vertices must themselves be annotated.
- Annotated edges can stand alone; they do not require that any adjacent vertices be annotated.
- The `[1..]` annotation is often the appropriate choice for annotating an edge adjacent to an annotated vertex.
- The vertex annotation takes precedence over the edge annotation.
- At most, only one of two adjacent vertices can be annotated.

Negated and optional elements

- A negated element is annotated with `[0]`, indicating that the corresponding database element must not be present in the matching subgraph.
- An optional element is annotated with `[0..]` or `[0..`*j*`]`, indicating that the corresponding database element may be present but is not required in the matching subgraph.
- To be well formed, a query must remain a connected graph after removing any negated or optional elements.
- The annotation `[1..]` is usually appropriate for annotating edges adjacent to a negated or optional vertex.
- Negated query elements do not appear in the query's results.
- Negated elements cannot substitute for conditions that express inequalities.

Implementation restrictions

- Proximity cannot process queries in which an annotated vertex is connected to more than one unannotated vertex.
- Proximity prohibits edge annotations with a lower bound of `0`.

Chapter 5. Constraints

Conditions let you specify restrictions on individual items in a query. To place restrictions across different items we use *constraints*. Constraints compare one vertex or edge in the query to another vertex or edge.

QGraph provides two types of constraints:

- *Identity constraints* compare the identity of the corresponding database objects, for example, making sure that the same object does not match multiple query elements.
- *Attribute constraints* compare attribute values for the corresponding database objects, for example, requiring that objects have the same attribute value.

Both types of constraints are described in more detail below.

Because constraints involve more than one query element, they apply to the query as a whole rather than to a specific vertex or edge. We therefore draw constraints separate from any query element. Figure 5.1 shows an example query with an identity constraint that requires that the object matching vertex *A* must be different from the object matching vertex *C*.

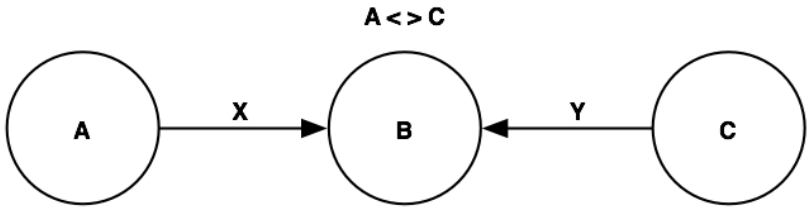


Figure 5.1. Example query with an identity constraint

Identity Constraints

Identity constraints are commonly used to ensure that the same database element does not match two different query elements. Consider the database fragment shown in Figure 5.2. This database represents interconnected web pages.

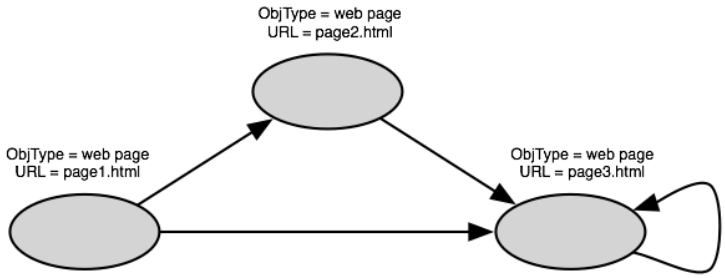


Figure 5.2. Database fragment [Const_DB01.xml]

Like many long web pages, page3.html links to itself, creating a loop. Notice, also, that this database does not have any link attributes. Neither QGraph nor Proximity requires the use of attributes for either objects or links in a database.

Figure 5.3 shows a query designed to find the cluster of pages (star) linked from each web page in the database.

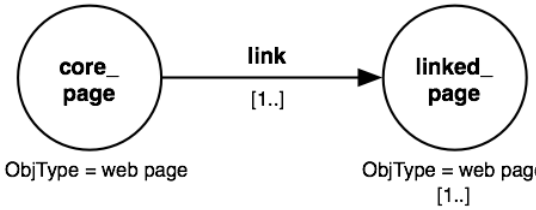


Figure 5.3. 1-dimensional star query [Const_DB01_Q01.qg2.xml]

With no constraints, we get the following results when the query is run on the database fragment shown in Figure 5.2.

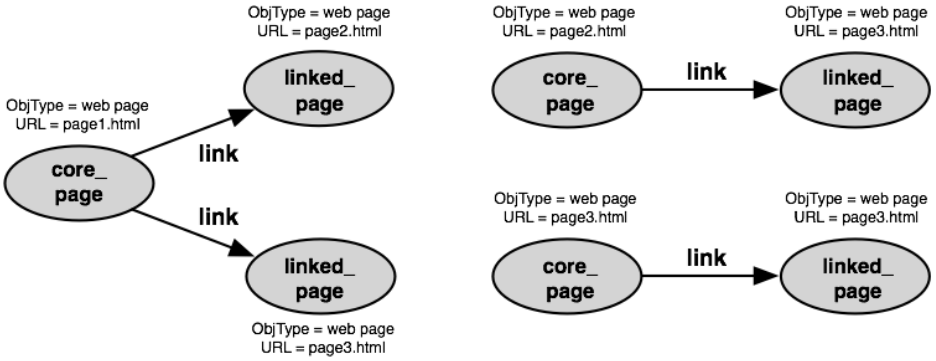


Figure 5.4. Query results

The subgraph with page3.html as the *core_page* shows how Proximity used the link from page3.html to itself to match the query. As we saw in Chapter 2, QGraph does not require that distinct query elements be matched by distinct database elements. Because this object matches both the *core_page* and the *linked_page* vertices in the query, it appears twice in the query results, once for each corresponding query vertex.

To eliminate such duplicated elements, we use an identity constraint that specifies that the object that matches the *core_page* vertex must not be the same as the object that matches the *linked_page* vertex.

The general form of an identity constraint is

```
element1 operator element2
```

where

- `element1` and `element2` are the names of two vertices or two edges in the query
- `operator` is one of =, <>, <, <=, >, and >=

The revised query with the identity constraint is shown in Figure 5.5.

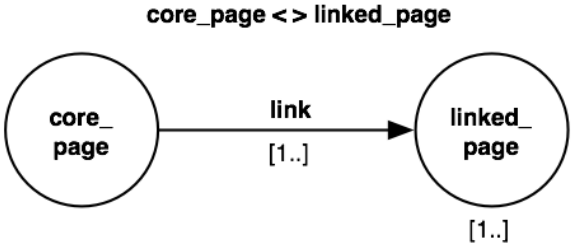


Figure 5.5. Query with identity constraint [Const_DB01_Q02.qg2.xml]

The query's constraint, core_page < > linked_page, prohibits matching the same object to both the *core_page* and *linked_page* vertices. The results of executing this query on the database fragment shown in Figure 5.2 are shown below:

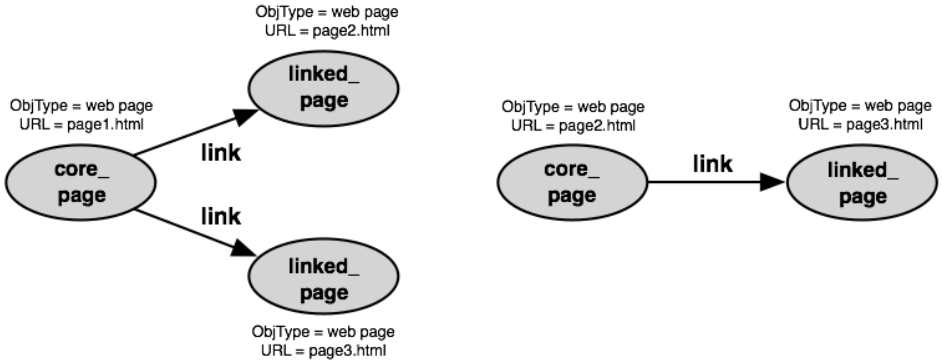


Figure 5.6. Query results

The subgraph with page3.html as the *core_page* is no longer included in the query results. The constraint ensures that the same object cannot match both of the query's vertices.

Another common use for constraints is to remove equivalent subgraphs from a query's results. Consider the fragment of a genealogy database shown below:

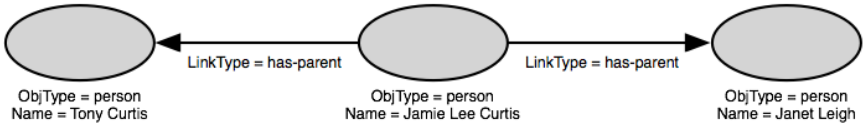


Figure 5.7. Database fragment [Const_DB02.xml]

A query (without constraints) that finds both parents of an individual is shown in Figure 5.8.

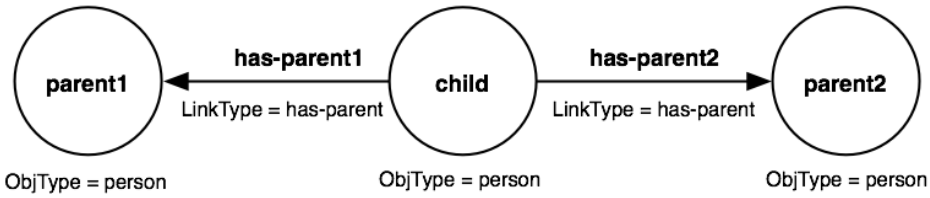


Figure 5.8. Query [Const_DB02_Q01.qg2.xml]

As we saw before, a query without constraints can return matches that include repeated elements as well as equivalent subgraphs where the same objects match different vertices, but with the order reversed. We call such subgraphs *mirror matches* because one often looks like the mirror image of the other when graphed.

The results of executing this query on the database fragment in Figure 5.7 are shown below:

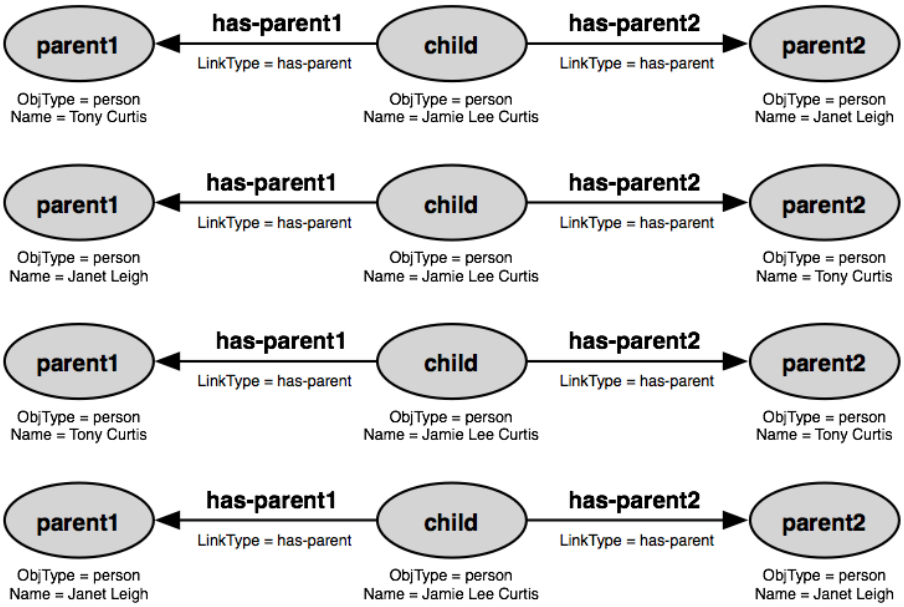


Figure 5.9. Query results

The top two subgraphs are mirror matches; they differ only in how Tony Curtis and Janet Leigh correspond to the *parent1* and *parent2* vertices. The bottom two subgraphs contain repeated elements. In one Tony Curtis matches both the *parent1* and *parent2* vertices; in the other, Janet Leigh matches both vertices.

As described before, we can add an identity constraint to the query to remove the subgraphs containing duplicated elements:

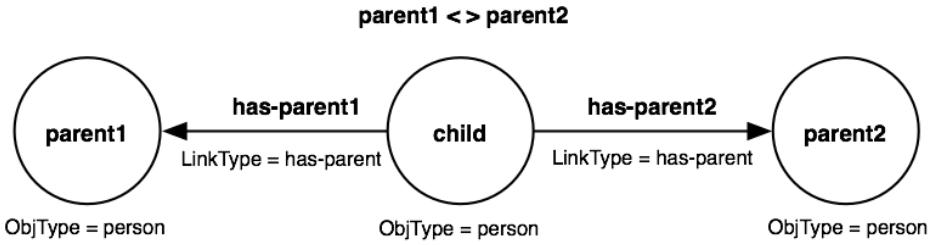


Figure 5.10. Query [Const_DB02_Q02.qg2.xml]

The constraint, parent1 < > parent2, ensures that the same object cannot match both vertices.

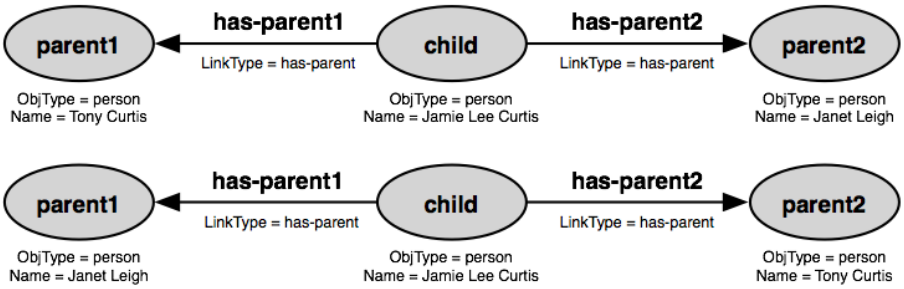


Figure 5.11. Query results

The inequality constraint parent1 < > parent2 eliminates the subgraphs where the same object matches

both the *parent1* and *parent2* vertices, but the results still include a mirror match. These two subgraphs include the same database objects and links but matches them to different query elements. To remove the mirror match from the results, we modify the constraint as shown below.

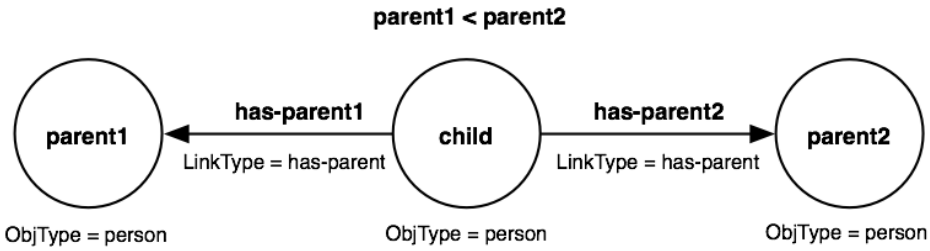


Figure 5.12. Query [Const_DB02_Q03.qg2.xml]

The new constraint, parent1 < parent2, enforces an ordering on the object IDs matching these vertices. Because each object has a unique ID, this ensures that different subgraphs do not contain the same objects in a different order. The results of the modified query can be seen below:

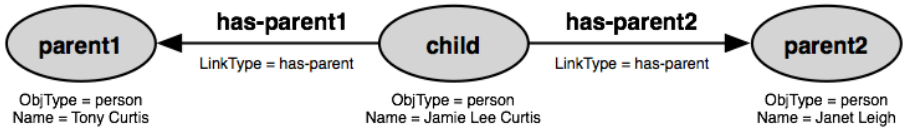


Figure 5.13. Query results

As you can see, this container includes just a single subgraph. The revised constraint removed the mirror match.

This constraint works because the underlying database provides a unique ID for each object and link. Proximity provides unique IDs; therefore, Proximity queries can take advantage of such constraints to eliminate mirror matches.

Attribute Constraints

Attribute constraints are used to compare attribute values across different database entities. (See "Implementation in Proximity" later in this chapter, however, for important restrictions regarding mixing object and link attributes in a constraint in Proximity.)

The general form of an attribute constraint is

```
element1.attribute1 operator element2.attribute2
```

where

- *element1* and *element2* are the names of two vertices or two edges in the query
- *attribute1* is the name of an attribute for *element1*
- *attribute2* is the name of an attribute for *element2*
- *operator* is one of =, <>, <, <=, >, and >=

Attribute1 and *attribute2* must be comparable types, but need not be the same attribute name.

Figure 5.14 shows a fragment of a database containing information about movies released as DVDs. Note that Proximity databases do not necessarily have to be fully connected; as this example illustrates, discontinuities are permitted.

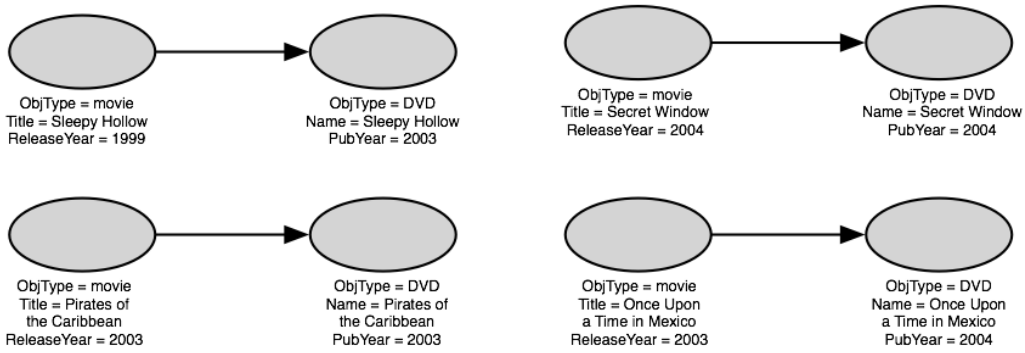


Figure 5.14. Sample database fragment [Const_DB03.xml]

Suppose that we want to find movies released as DVDs in the same year as their theatrical release. Figure 5.15 shows the QGraph query for this task.

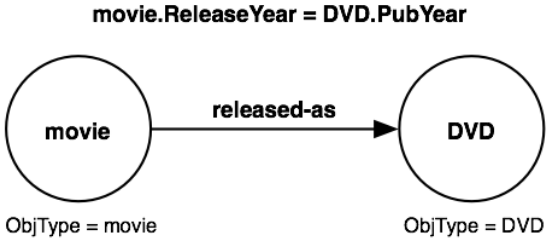


Figure 5.15. Query: Movies and DVDs released in the same year [Const_DB03_Q01.qg2.xml]

The query's constraint requires that the value of the ReleaseYear attribute of the object corresponding to the *movie* vertex be the same as the value of the PubYear attribute of the object corresponding to the *DVD* vertex. Figure 5.16 shows the results of executing the above query on the database fragment shown in Figure 5.14.

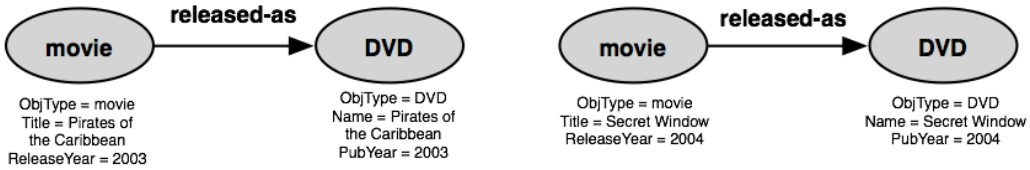


Figure 5.16. Query results: Movies and DVDs released in the same year

The query finds two subgraphs where a movie's DVD release occurred in the same year as its theatrical release.

Multiple Constraints

In principle, QGraph permits any boolean combination of constraints. In query diagrams, like that shown in Figure 5.17, we simply list all the constraints with the understanding that they must all be satisfied. (Proximity, however, requires that multiple constraints always be `AND`ed together. You cannot combine constraints with `OR` or use `NOT` in a constraint.)

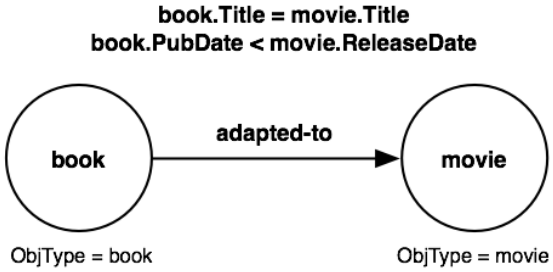


Figure 5.17. Query with multiple constraints

This query finds books that have been adapted to movies where both the book and movie have the same name, and where the book was published before the movie was released.

Queries may include both identity and attribute constraints, but an individual constraint may not attempt to compare identity with attribute values. For example, if a database includes objects with an age attribute, a query for that database might include both an identity constraint such as vertex1 < > vertex2 and an attribute constraint such as vertex1.age < vertex2.age, but it may not include a constraint such as vertex1 < vertex2.age that attempts to compare identity to an attribute value.

Evaluating Constraints

As we saw with conditions, the process of evaluating attribute constraints must take into account that attributes are set valued. Constraints follow the same evaluation principle as do conditions: A constraint is satisfied if any member of the set of attribute values satisfies the constraint.

The rule for evaluating attribute constraints can be more formally expressed as:

Given

- two database entities (objects or links) *A* and *B*
- an attribute *attr1* on *A*
- an attribute *attr2* on *B*, where *attr1* and *attr2* must be comparable types and may be the same attribute
- a set of values *V1* for *attr1* on *A*
- a set of values *V2* for *attr2* on *B*
- an attribute constraint *A.attr1 operator B.attr2*

The constraint is satisfied if there exists a value *a1* a member of *V1*, and a value *a2* a member of *V2*, such that *a1 operator a2* holds.

As an example, let's look at a fragment of a database describing books and related movies, shown below.

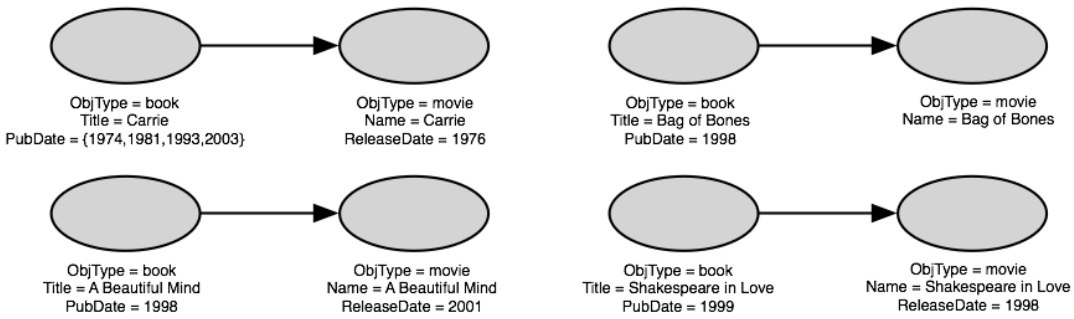


Figure 5.18. Database fragment [Const_DB04.xml]

Because books are often reissued or issued in different formats, they can have multiple publication dates, as we see with *Carrie*. Also, books and movies can have varying temporal relationships. Frequently, movies are adapted from an existing book, as we see for *A Beautiful Mind*, but occasionally a screenplay or novelization is published after a successful movie, as is the case for *Shakespeare in Love*. And sometimes we know that a book's movie rights have been optioned, but the movie has not yet been released, as we see for *Bag of Bones*.

This database has no attributes on the links connecting books and their associated movies. Although this is a directed link, as Proximity requires, the lack of any attributes means that we have no semantics for this link and thus cannot assign a temporal ordering based on the arrow's direction. We could have just as easily reversed the links without changing the information represented by these objects and links.

Suppose we want to find both categories of books—those released before their movie adaptation, and those initially published after the movie's release. We need two queries to do so:

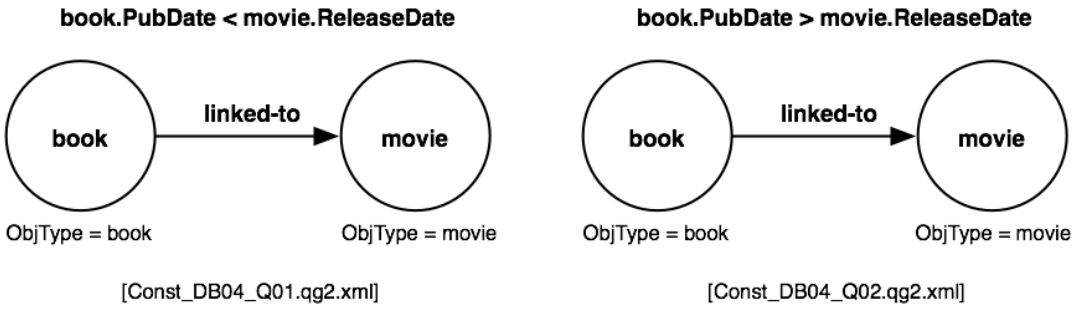


Figure 5.19. Queries

The results of executing the first query are shown below:

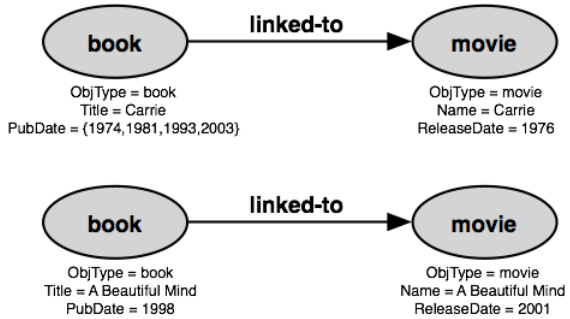


Figure 5.20. Query results

The resulting container includes two subgraphs. The movie *A Beautiful Mind* was based on the book written by Sylvia Nasar, so it came out after the book's publication. *Carrie* was originally published before the movie adaptation, then re-released in various formats after the movie's release.

Compare these matches to those for the second query:

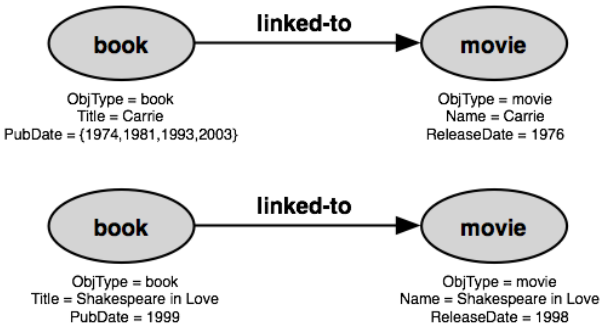


Figure 5.21. Query results

This time, the movie *Shakespeare in Love* appears in the results; its associated book is a screenplay based on the movie. But *Carrie* appears again, this time because re-releases of the book occurred after the movie's release.

Because objects and links in a Proximity database can be structurally heterogeneous, some entities may not have a value for an attribute used in a constraint. In our example database fragment, because the movie version of *Bag of Bones* has not been released yet, the corresponding database object has no value for the ReleaseDate attribute. In such cases, an attribute constraint's test always fails because there are no values to compare against the constraint's requirements. Therefore *Bag of Bones* does not appear in any of our query results.

As we saw for conditions, the ability to have set-valued attributes can result in satisfying apparently contradictory constraints. In this case, the book-movie pair for *Carrie* appears in the results for both queries because the multiple book publication dates contain values that satisfy both constraints. Specifically, the values for the book's publication date attribute include a value (1974) that is before the movie's release date of 1976, and several values (1981, 1993, and 2003) that are after the movie's release date.

Constraints and Annotations

Constraints on annotated elements

QGraph prohibits constraints between two annotated query elements. As we saw in Chapter 4, *Numeric Annotations*, annotating adjacent vertices can result in ambiguities of interpretation. Placing a constraint on two annotated elements results in similar ambiguities.

QGraph admits one exception to the above prohibition against constraints between two annotated elements: Constraints between an annotated vertex and its adjacent annotated edge are permitted. This exception has not been implemented in Proximity, however. Proximity prohibits all constraints between two annotated elements, whether or not they involve this special case.

Proximity imposes additional restrictions on the use of constraints for annotated elements. See "Implementation in Proximity" later in this chapter for details on these limitations.

Constraints involving negated and optional elements

What does it mean to have a constraint involving optional or negated elements (elements that may not or must not be present)? Interpreting such queries can require careful attention to how each query component influences how database elements match the query. This is most easily seen by walking through a relevant example.

The database fragment below represents selected college and university student Internet home pages and the pages they connect to:

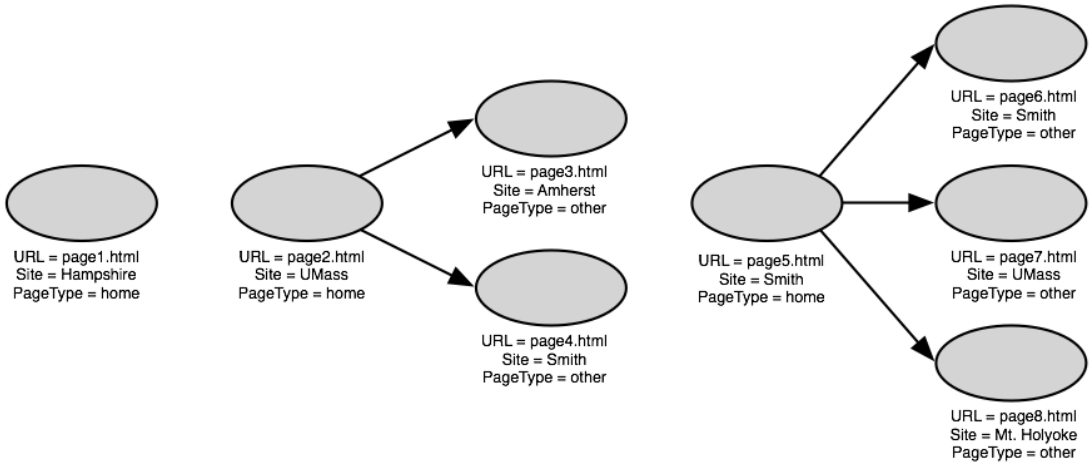


Figure 5.22. Database fragment [Const_DB05.xml]

The query in Figure 5.23 finds home pages that do not link to any other pages from the same site.

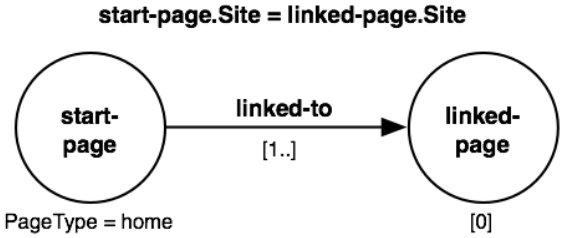


Figure 5.23. Query [Const_DB05_Q01.qg2.xml]

The constraint start-page.Site = linked-page.Site requires that the query only consider linked pages from the same site as the start page. The `[0]` annotation on the *linked_page* vertex tells the query to only match subgraphs that contain no such linked pages. In other words, the query will only match student home pages that do not link to any other pages from the same site. Such home pages may or may not link to pages on other sites. Linked pages from different sites are irrelevant for this query.

The results of executing this query are shown below:

Figure 5.24. Query results

This query found two home pages that do not connect to other pages on the same site. Page1.html doesn't link to any other page; therefore, it satisfies the query. Page2.html links to two other pages, but neither of the linked pages is from the same site, so it also satisfies the query. The third home page in the database, page5.html, links to another page from Smith and thus satisfies the constraint, but the `[0]` annotation excludes it from the query results.

If we modify the query to use `[0..]` instead of `[0]` for the *linked-page* vertex, we change the meaning of the query and thus its results.

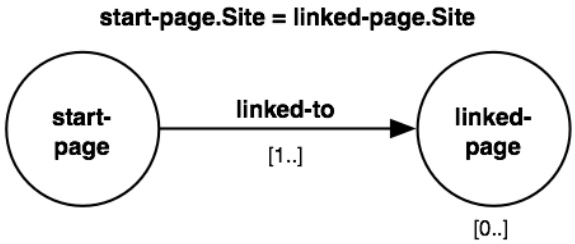


Figure 5.25. Query [Const_DB05_Q02.qg2.xml]

This modification says that a home page may, but is not required to, link to a page from the same site to match the query. Executing the modified query yields different results than those we saw for the query in Figure 5.23.

Figure 5.26. Query results

Because home pages that do not link to other pages from the same site match the query, the results of this query include the previous results. Page5.html links to a page from the same site and therefore satisfies the constraint. But this time the `[0..]` annotation on the *linked-page* vertex tells us to include such pairs in the results.

Implementation in Proximity

Proximity imposes some additional restrictions on how constraints can be used in QGraph queries. This section describes the current requirements for using constraints in Proximity.

Constraint elements

QGraph permits constraints between an edge and a vertex, as long as they involve comparable types and obey the annotation requirements discussed above, but this functionality is not currently implemented in Proximity. Proximity restricts constraints to either two vertices or two edges; mixing is not allowed.

As we saw above, constraints do not have to use the same attribute on both sides of the expression, as long as the attribute data types can be legally compared. Proximity allows you to compare `DBL` with `FLT` and `INT`, and `FLT` with `INT`.

Annotation restrictions

Proximity adds the additional restriction that you cannot have a constraint between two edges if one of them is annotated. Proximity admits one exception to this rule: Constraints between an annotated edge and an unannotated edge are permitted only if the annotated edge is connected to a vertex annotated with `[0..]` or `[0..`*`n`*`]`.

Multiple constraints

Although QGraph permits arbitrary boolean combinations of constraints, Proximity requires that multiple constraints only be combined with `AND`; the boolean operators `OR` and `NOT` are not supported.

Summary

Constraint uses

- Constraints compare one vertex or edge in the query to another vertex or edge.
- Constraints apply to the query as a whole, rather than to a specific vertex, edge, or subquery.
- Identity constraints compare the identity of the corresponding database objects.
- Attribute constraints compare attribute values for the corresponding database objects.
- Constraints can compare either identity or attributes; mixing is not allowed.

Identity constraints

- Identity constraints can use the comparison operators =, <>, <, <=, >, and >=.
- Identity constraints are commonly used to ensure that the same database element does not match two different query elements.
- Another typical use for identity constraints is to remove equivalent subgraphs (mirror matches) from a query's results.

Attribute constraints

- Attribute constraints can use the comparison operators =, <>, <, <=, >, and >=.
- The attributes on both sides of a constraint must be comparable types, but need not be the same attribute.
- A constraint is satisfied if any member of the set of attribute values for the corresponding database entities satisfies the constraint.
- When an object or link has multiple values for an attribute, different constraints may match different values resulting in matching apparently contradictory constraints.

Constraints and annotations

- QGraph prohibits constraints between two annotated query elements.
- QGraph admits one exception to the prohibition against constraints between two annotated elements: Constraints between an annotated vertex and its adjacent annotated edge are permitted.

Implementation restrictions

- Proximity does not permit comparing object and edge attributes in the same constraint.
- Proximity limits the allowed boolean operations on constraints. Multiple constraints are always `AND`ed together. You cannot combine constraints with `OR` or negate constraints with `NOT` in Proximity.
- Proximity allows you to compare `DBL` with `FLT` and `INT`, and `FLT` with `INT`.
- Proximity does not permit any constraint between annotated elements, even those between an annotated vertex and its adjacent annotated edge, which are allowed in QGraph.
- Proximity prohibits constraints between two edges if one of them is annotated, with the exception that such constraints are allowed if the vertex adjacent to the annotated edge is itself annotated with `[0..]` or `[0..`*`j`*`]`.

Chapter 6. Subqueries

A subquery is a connected subgraph of vertices and edges that can be treated as a logical unit. Using subqueries expands the expressive power of QGraph, enabling you to identify more complex structures than could be found otherwise.

The example with which we opened this *Guide*, repeated in Figure 6.1, shows a query that finds subgraphs containing a director, all the movies he or she has directed, and all the actors who have appeared in those movies. This example contains a subquery, denoted by the box surrounding the *movie* and *actor* vertices and the *acted-in* edge.

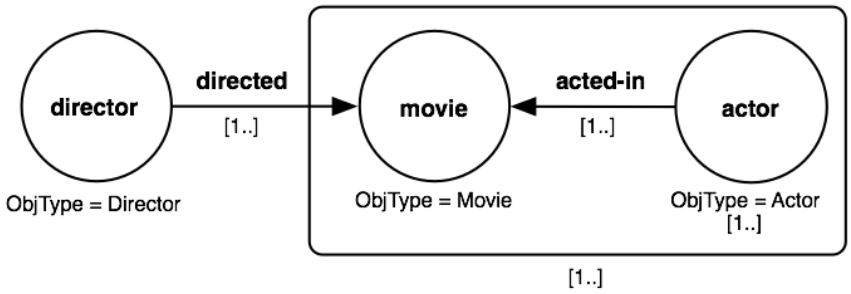


Figure 6.1. Example query with subquery [Intro_DB01_Q01.qg2.xml]

The subquery in Figure 6.1 is linked to its parent query by an edge connecting the *movie* vertex to the *director* vertex. We call this edge (the edge labeled *directed* in the above query) a *boundary edge* of the subquery. All subqueries must be connected to one or more vertices in the main query. (However, see "Implementation in Proximity" later in this chapter for restrictions on how subqueries can be connected to the main query in Proximity.)

Subqueries expand QGraph's expressive power by letting you attach a numeric annotation to a connected set of vertices and edges instead of just a single vertex or edge. This effectively lets you treat a more complex structure as if it were a single vertex. For example, if we replace the subquery in Figure 6.1 with a single vertex, we see the familiar star query structure.

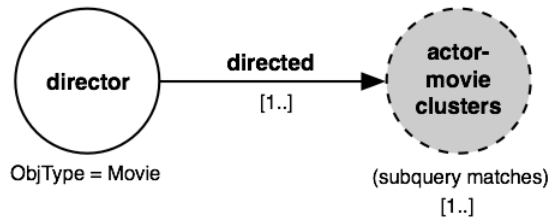


Figure 6.2. Conceptual structure of query in Figure 6.1

This diagram shows the subquery as a single vertex. The `[1..]` annotation on the subquery means that the complex structures matching the subquery are grouped in the same way that objects are grouped when matching an annotated vertex. Thus all the movies and their linked actors for a specific director will be included in a single subgraph. Executing this query on the sample database contained in `Intro_DB01.xml` returns six subgraphs, one for each director object in the database. The subgraph where the *director* vertex matches Steven Spielberg is shown in Figure 6.3 (Edge labels have been removed for space reasons.)

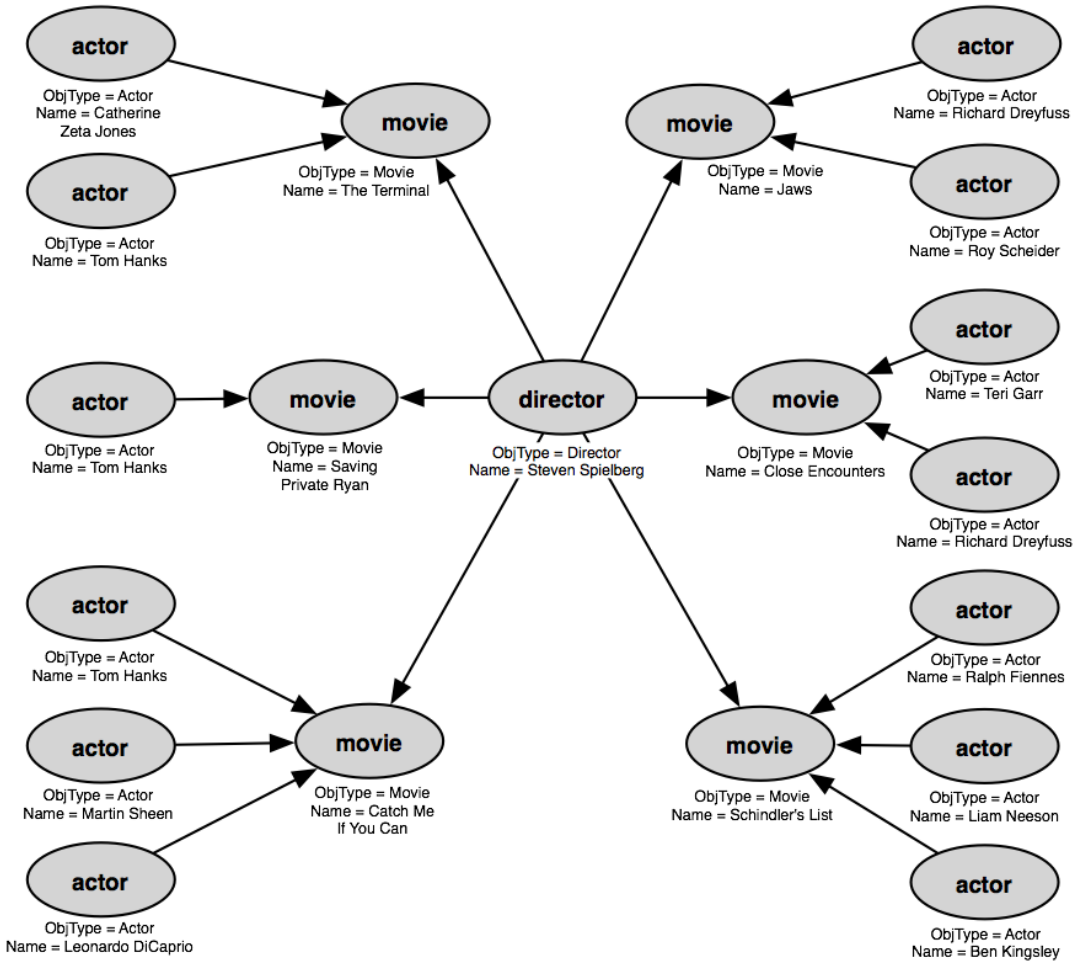


Figure 6.3. Query results for director = Steven Spielberg (edge labels omitted)

All the movies directed by Steven Spielberg, as well as all the actors linked to those movies, are included in this single subgraph.

Compare the subgraph shown in Figure 6.3 to the results of a similar query that does not use subqueries. Figure 6.4 shows the query from Figure 6.1, but without the subquery box and subquery annotation.

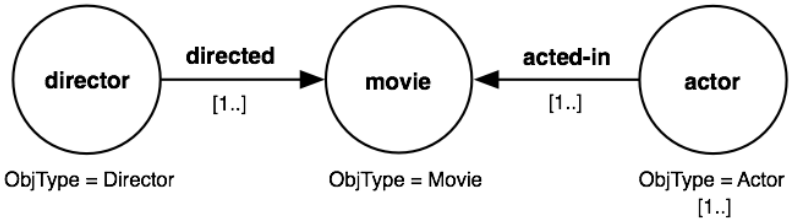


Figure 6.4. Similar query without subquery [SubQ_DB01_Q02.qg2.xml]

Executing this query on the same data returns 23 subgraphs, one for each movie in the database. The subgraphs in which the *director* vertex matches Steven Spielberg are shown in Figure 6.5.

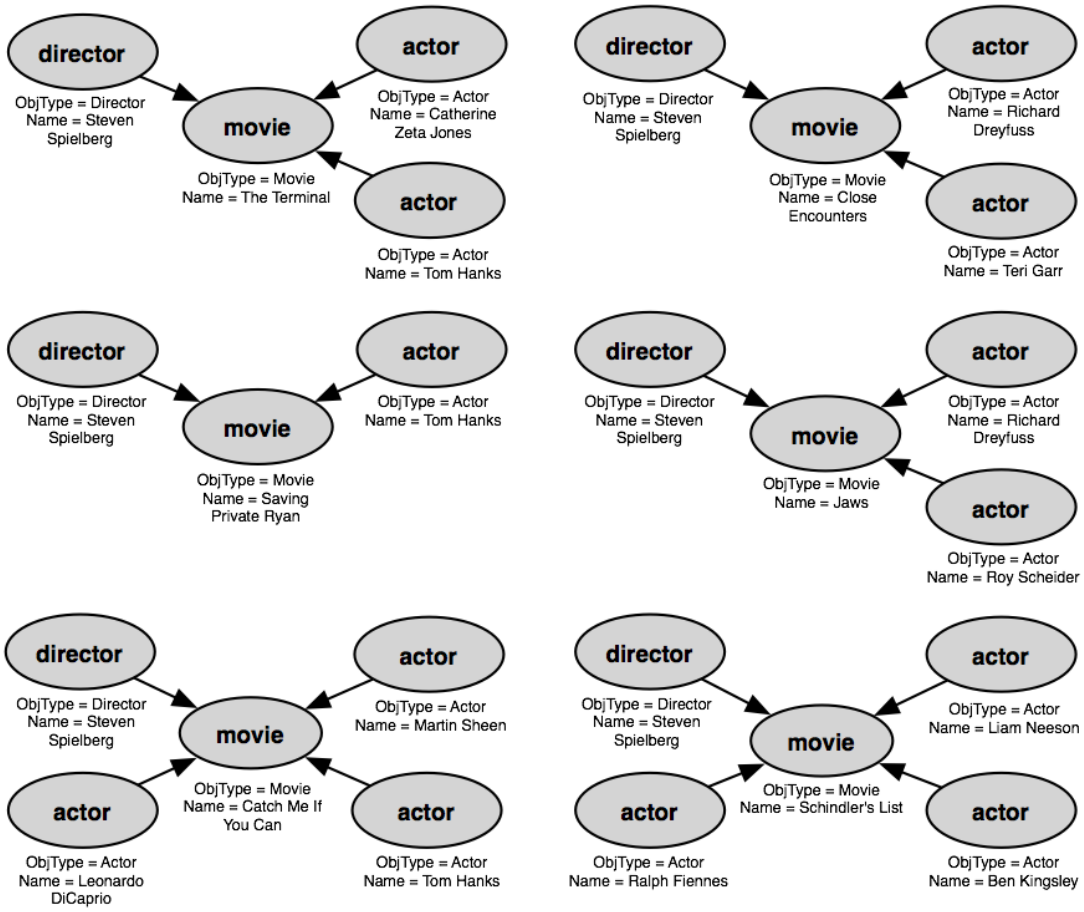


Figure 6.5. Query results for director = Steven Spielberg (edge labels omitted)

Because the *movie* vertex is not annotated, the query cannot group all the movies for a single director and must return a separate subgraph for each director-movie pair in the data.

The *inner structure* of a subquery must be a well-formed query in its own right. (The inner structure of a subquery is the part that remains after removing the boundary edges and the subquery box with its annotation.) For example, the inner structure of the subquery shown in Figure 6.1, shown below, forms a valid query in its own right.

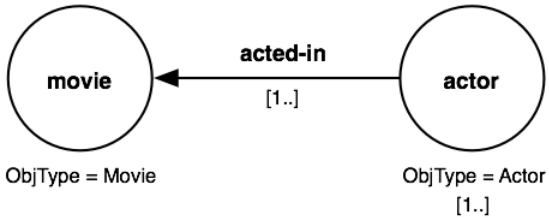


Figure 6.6. Inner structure of subquery [SubQ_DB01_Q01_SubQ.qg2.xml]

The inner structure of this subquery is the familiar star query that finds all actors linked to a single movie. If we choose, we can create and execute a new query containing just this structure.

In particular, because disconnected queries are not well-formed, subqueries cannot be disconnected. That is, all the vertices and edges inside the subquery box must be connected in a single graph. For example, the query shown in Figure 6.7 contains a disconnected subquery.

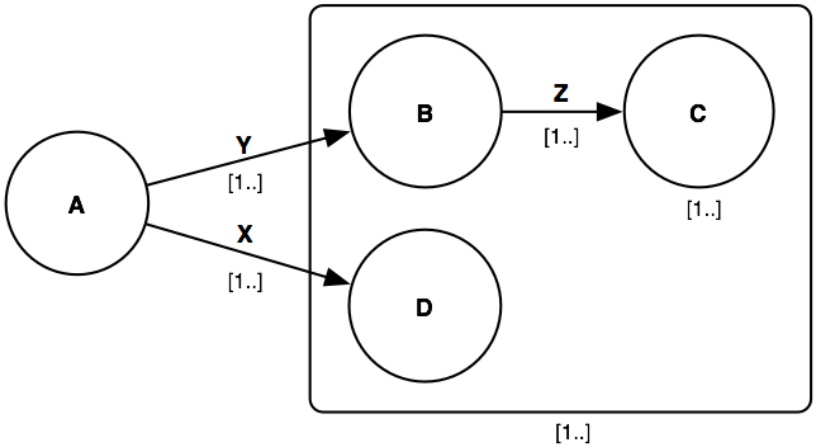


Figure 6.7. Illegal query containing disconnected subquery

If we look only at the elements inside the subquery box, we see that the *D* vertex is not connected to the other subquery components (the *B* and C vertices and the *Z* edge). Although the the query as a whole is connected, because it contains a disconnected subquery, the query shown in Figure 6.7 is illegal.

Subqueries and Annotations

QGraph requires that all subqueries must be annotated. An unannotated subquery is equivalent to the same query structure without the subquery box. If we could remove the annotation from the subquery box and then run the query shown in Figure 6.1, we would see the same results we saw in Figure 6.5. Because an unannotated subquery duplicates capabilities available via other QGraph elements, unannotated subqueries add nothing to QGraph's expressive power and QGraph therefore requires that all subqueries must be annotated.

Because subqueries are annotated, they must obey all the QGraph rules that apply to annotated query elements. For example, QGraph requires that the edge adjacent to an annotated element must itself be annotated. Therefore the boundary edge(s) of an annotated subquery must always be annotated.

The query shown in Figure 6.1 illustrates the proper annotation of a subquery's boundary edge. As we saw earlier in Chapter 4, although other annotations are also legal, most queries will probably use the `[1..]` annotation on the boundary edge.

Because QGraph prohibits edges connecting two annotated elements, no numeric annotation is allowed on a vertex adjacent to an annotated subquery. For example, the query structure shown below is illegal:

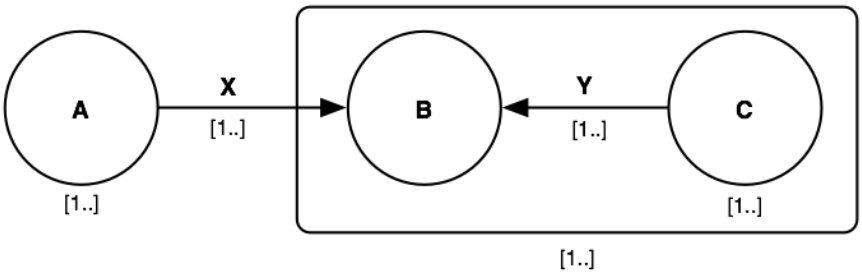


Figure 6.8. Illegal annotation (vertex *A*)

As we saw in the first part this chapter, we can mentally substitute a single vertex for a subquery to better understand the conceptual structure of a query. This heuristic also works well for seeing potential annotation problems involving subqueries. For example, we can visualize the query shown in Figure 6.8 as the structure shown below:

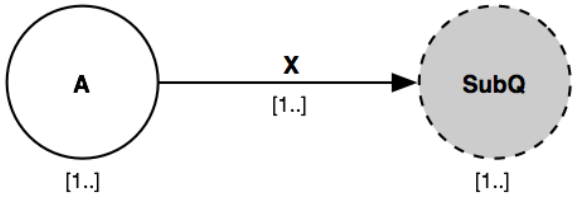


Figure 6.9. Conceptual structure of illegally annotated query

This conceptual view helps us see that the query includes an edge connecting two annotated elements, the subquery and the *A* vertex. Because this conceptual structure includes an illegal annotation, we can more easily see that the corresponding query is also illegal.

Subqueries and Constraints

In this section we examine the ways in which subquery elements can be used in constraints. A constraint may compare two elements within the same subquery, or it may compare elements that span the subquery boundary. Constraints that involve elements inside a subquery must obey the same rules that apply to any QGraph constraint. Proximity's current implementation of QGraph imposes additional restrictions for constraints that cross the subquery boundary. See "Implementation in Proximity" later in this chapter for information on these restrictions.

Constraints within a subquery

Let's look first at a constraint that compares two items within the same subquery. Our example uses a database containing information on student and faculty web pages and their interconnecting links. A fragment of such a database is shown in Figure 6.10.

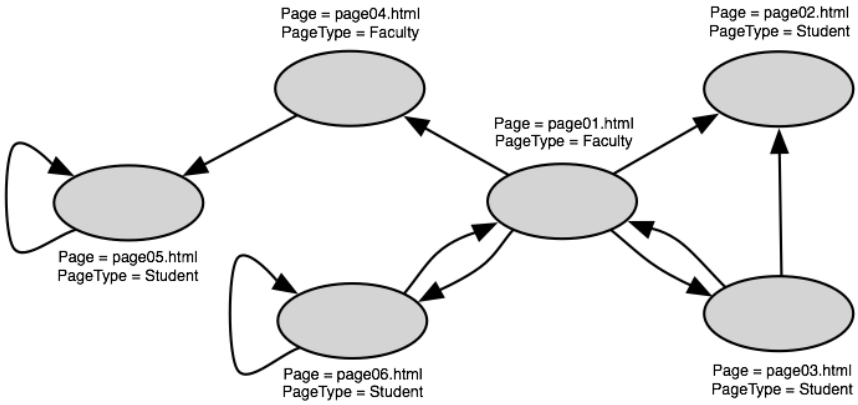


Figure 6.10. Database fragment [SubQ_DB02.xml]

It's common for two web pages to link to each other and for pages to link to themselves. We see this reflected in our database fragment where, for example, page01.html and page03.html point to each other and where page06.html links to itself. Our goal is to find all the student pages that we can reach by following exactly two links (hops) from a faculty page. Figure 6.11 shows a first pass at creating such a query.

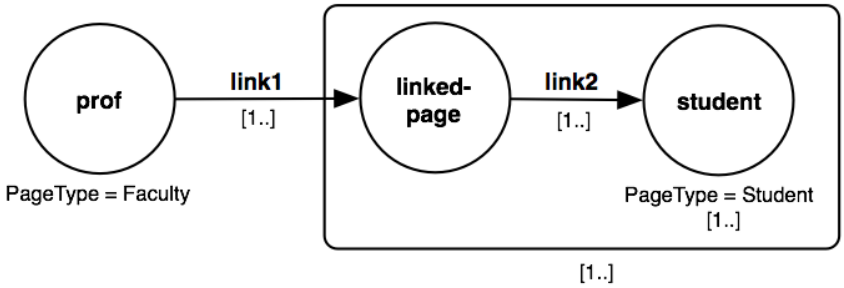


Figure 6.11. Query [SubQ_DB02_Q01.qg2.xml]

We want our results to include a single subgraph for each faculty member, so the query uses a subquery to group the cluster of pages linked from each *linked-page* vertex.

Because the database contains objects that link to themselves, this query will incorrectly identify some pages as being two hops away when the second hop follows this self link. We see this in the results of executing the query on our database fragment:

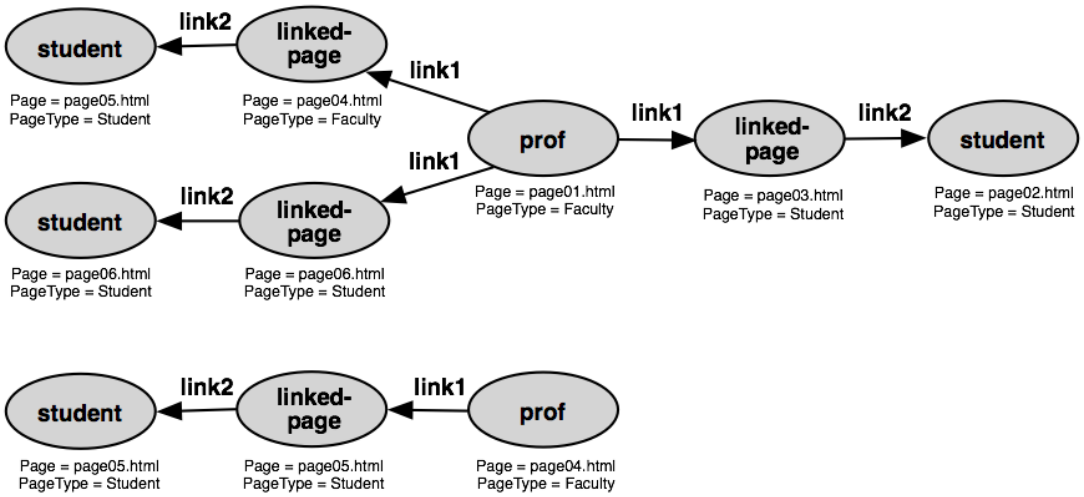


Figure 6.12. Query results

Because page06.html links to itself, the top subgraph shows page06.html as being both one and two hops away from page01.html. Similarly, because page05.html links to itself, the bottom subgraph shows that page05.html matches both the *linked-page* and *student* vertices. To eliminate these matches, we add an identity constraint that requires that the *linked-page* and *student* vertices not match the same object.

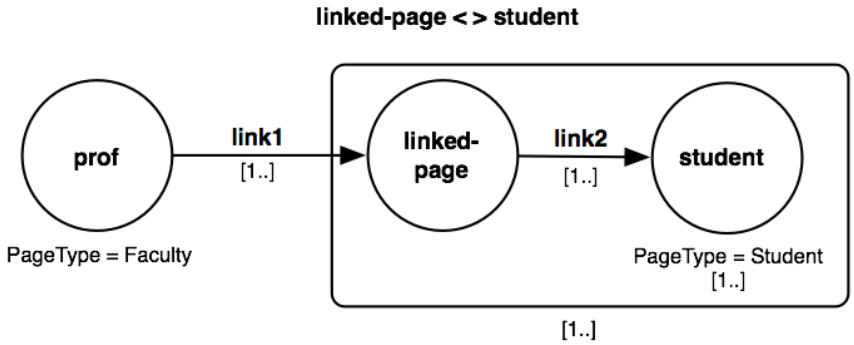


Figure 6.13. Revised query with constraint [SubQ_DB02_Q02.qg2.xml]

The results of executing this modified query on our database fragment are shown below:

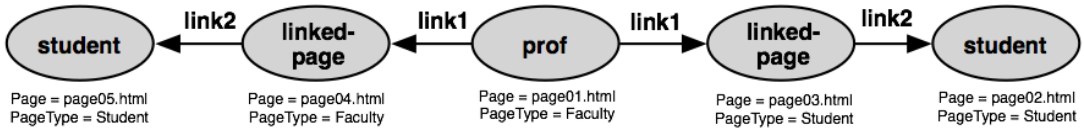


Figure 6.14. Revised query results

The two student pages, page05.html and page06.html, that were included as a result of the self links are no longer included in the query results.

Constraints crossing the subquery boundary

Constraints can also cross subquery boundaries. That is, constraints can compare a query element inside a subquery with a query element outside the subquery, either in the main query or in a different subquery. (Proximity currently places limitations on the the circumstances under which constraints can cross the subquery boundary. See “Implementation in Proximity” later in this chapter for details on these restrictions.)

The “Six Degrees” game finds paths between pairs of actors based on the shared movies in which they and other actors have appeared. The length of the shortest path between actors becomes the “degree of separation” between these two actors. The database fragment shown in Figure 6.15 contains information on actors and the other actors with whom they have co-starred. Links between actors indicate that those two actors appeared in the same movie. Proximity requires that all links be directed, so the database includes two links, one in each direction, between each pair of connected actors.

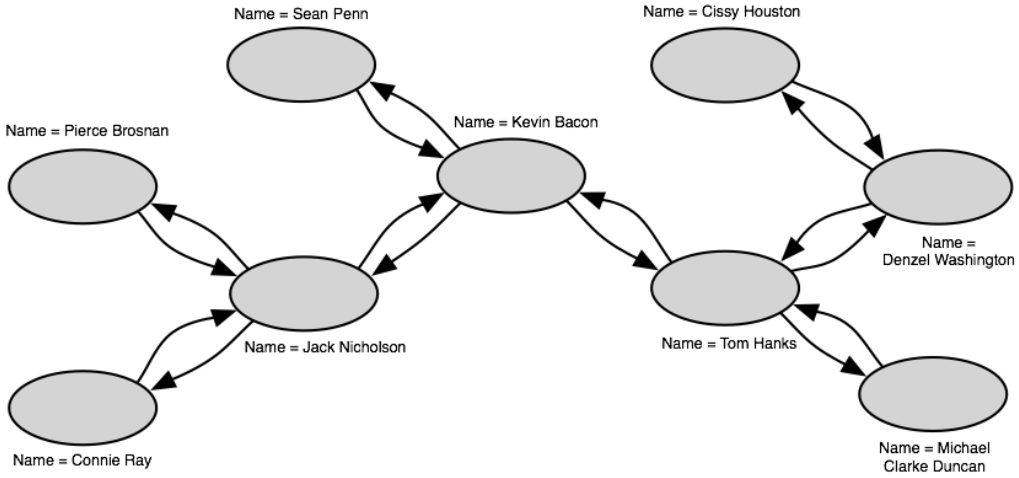


Figure 6.15. Database fragment [SubQ_DB03.xml]

The query in Figure 6.16 finds actors who are one or two degrees of separation from another actor in the database.

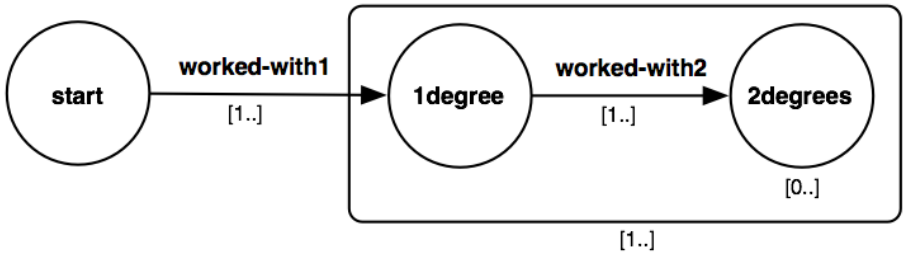


Figure 6.16. Two degrees of separation query without constraint [SubQ_DB03_Q01.qg2.xml]

A query without constraints, like the one shown in Figure 6.16, suffers from a problem similar to that which we encountered in our previous example. In this case, because the database includes links in both directions, this query can match the same object to two different vertices. Specifically, the same database object can match both the *start* and *2degrees* vertices, as can be seen in the sample subgraph drawn from the query's results, shown below:

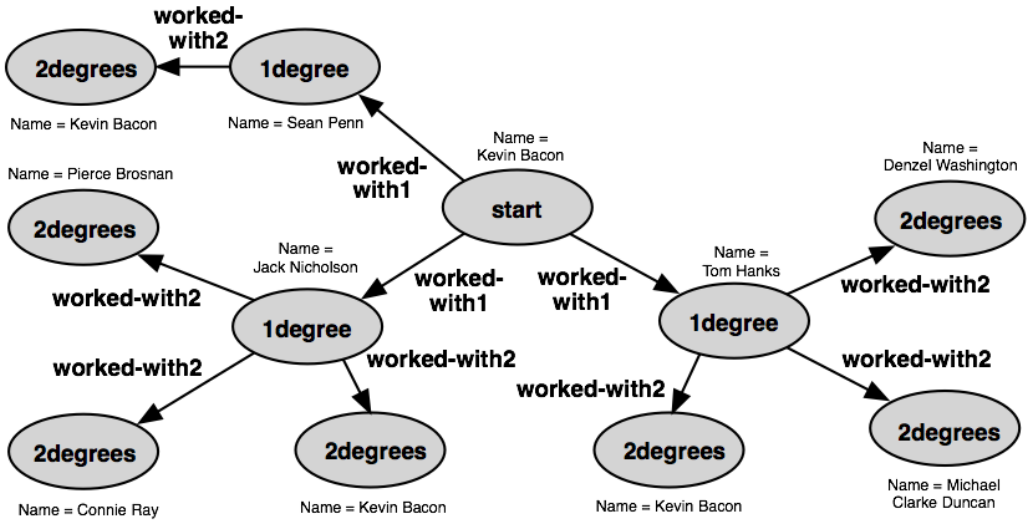


Figure 6.17. Two degrees of separation results for Kevin Bacon (no constraint)

Figure 6.17 shows the subgraph in which the *start* vertex matches the Kevin Bacon object. This subgraph includes three instances of Kevin Bacon also matching the *2degrees* vertex. In these instances, the query matched the *worked-with2* edge to the links pointing back towards the Kevin Bacon object.

To eliminate these unwanted matches, we add an identity constraint to the query, as shown in Figure 6.18. The constraint states that the same object may not match both the *start* and *2degrees* vertices.

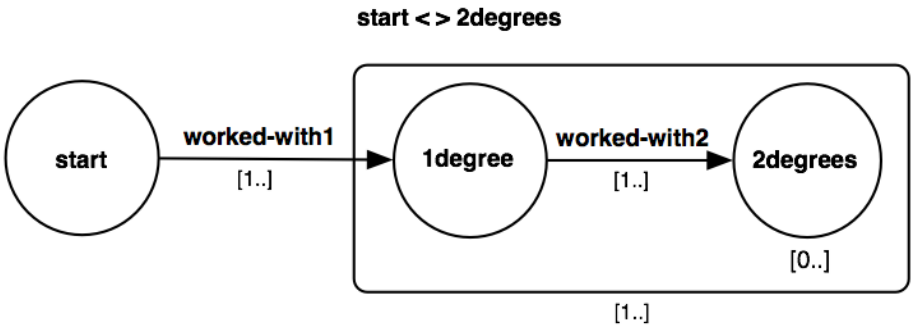


Figure 6.18. Two degrees of separation query with constraint [SubQ_DB03_Q02.qg2.xml]

Again, we examine the subgraph in which the *start* vertex matches the Kevin Bacon object to examine.

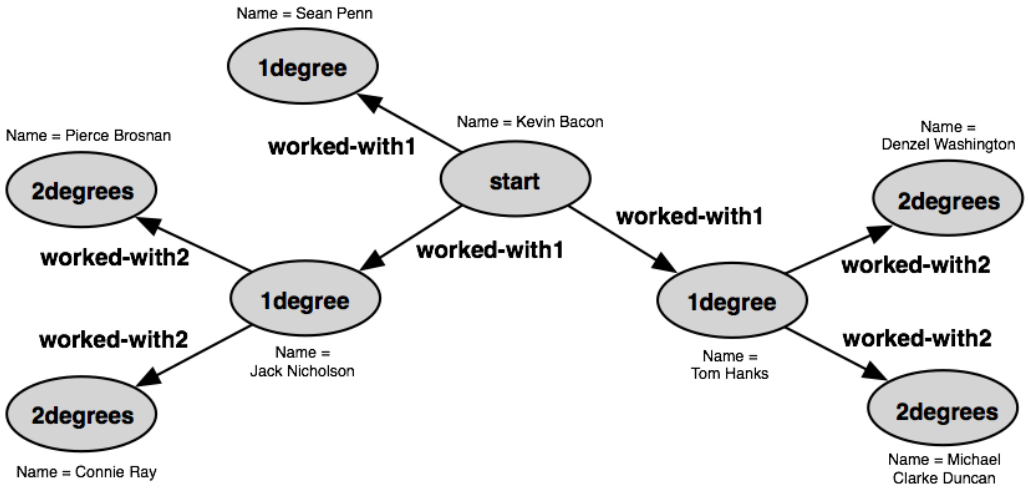


Figure 6.19. Two degrees of separation results for Kevin Bacon

The constraint successfully eliminated the three instances of Kevin Bacon matching the *2degrees* vertex that we saw in Figure 6.17.

Multiple Subqueries

You can include more than one subquery in a QGraph query. Because subqueries are always annotated, you must ensure that queries containing subqueries obey QGraph's annotation rules. For example, because QGraph prohibits edges connecting two annotated elements, you cannot connect two subqueries by a single edge.

In the query shown in Figure 6.20, two subqueries are connected by the *Z* edge. Because the *Z* edge connects two annotated elements, this query is illegal.

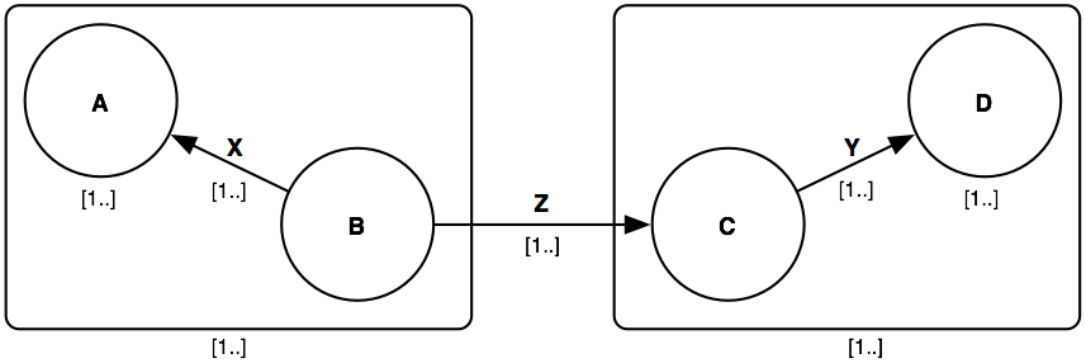


Figure 6.20. Illegal query

It may be easier to see whether your query obeys all annotation rules by visualizing any subqueries as vertices. Figure 6.21 shows such a view of the above query:

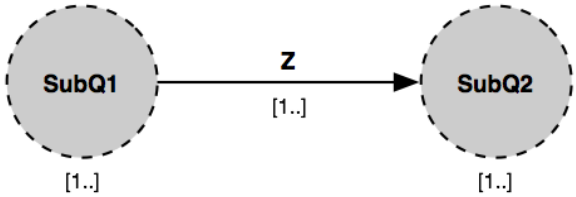


Figure 6.21. Conceptual structure of illegal query

Visually collapsing the subqueries into single vertices helps to illustrate how the *Z* edge connects two annotated query elements.

Compare the query in Figure 6.20 to the one shown below.

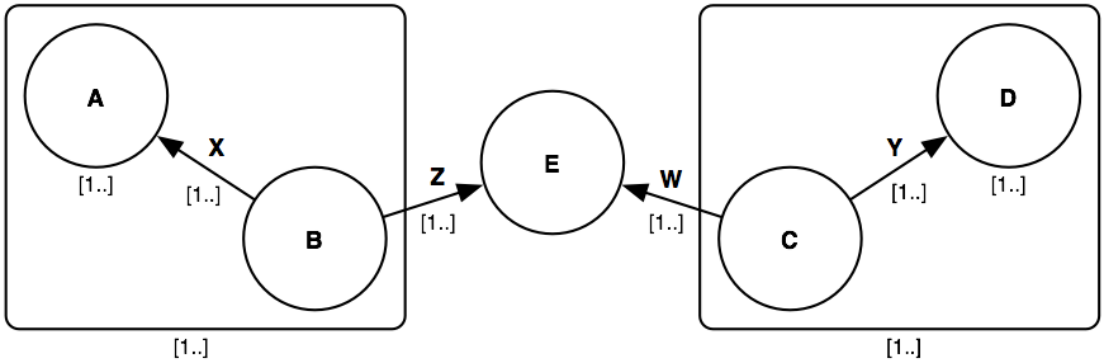


Figure 6.22. Legal query

This query adds another vertex (vertex *E*) between the two subqueries. Visualizing the subqueries as vertices yields the structure shown in Figure 6.23.

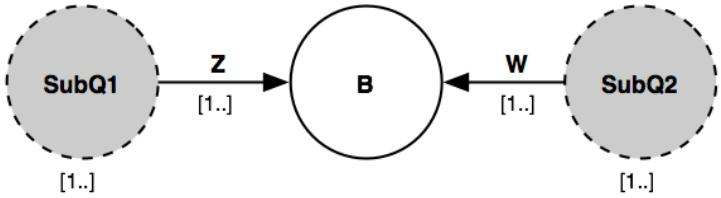


Figure 6.23. Conceptual structure of legal query with two subqueries

This view helps us see that this structure obeys QGraph's requirements for annotations, and is thus a legal query.

Nested subqueries

Although not yet supported by Proximity, QGraph also lets you nest subqueries inside other subqueries. This further increases QGraph's expressive power and lets you match more complex structures than would be possible with a single level of subquery. The query shown in Figure 6.24 shows a query that includes a nested subquery.

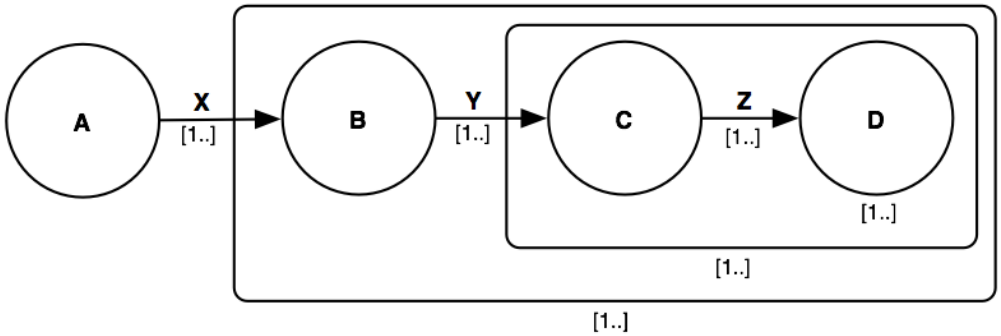


Figure 6.24. Nested subqueries

Nested subqueries must obey the same annotation requirements that apply to any query element. In the above query, both the top-level subquery and nested subquery are annotated, as are the *X* and *Y* incident edges.

This query extends the pattern we saw in the query shown in Figure 6.1, which returned 2d-star subgraphs—clusters of linked objects two hops away from a central object. In this case, using nested subqueries lets us find and return 3d-star subgraphs. Deeper nesting offers the opportunity to match even more complex structures.

We can continue our technique of visualizing subqueries as vertices to help us understand more complex queries. Visualizing the nested subquery in Figure 6.24 as a single vertex gives us the structure shown below.

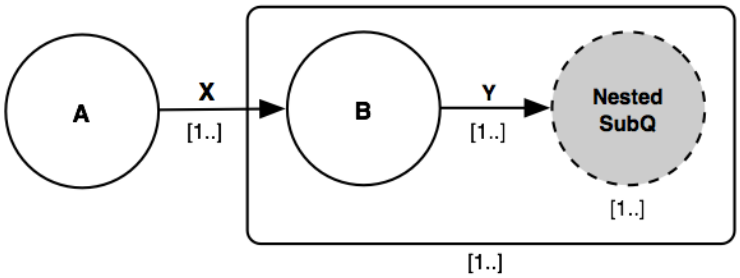


Figure 6.25. Conceptual structure of nested subqueries

By treating a complex structure (the nested subquery) as a vertex inside the top-level subquery, we can more clearly see how the structure of this query resembles the 2d-star query we saw in Figure 6.1.

Implementation in Proximity

Proximity imposes some additional restrictions on how subqueries can be used in QGraph queries. This section describes the current requirements for using subqueries in Proximity.

Edge requirements

Subqueries can potentially be connected to the main query via multiple edges in several ways. Proximity, however, requires that a subquery can only be connected to the main query via a single edge. Figure 6.26 shows two queries that cannot currently be executed in Proximity.

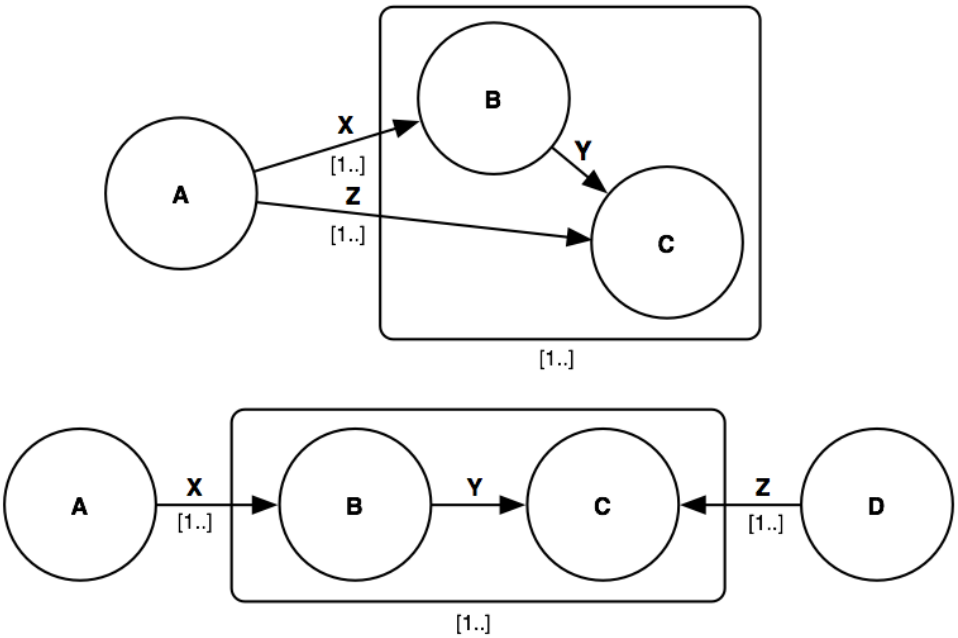


Figure 6.26. Unimplemented Proximity queries

Proximity cannot execute queries that have multiple boundary edges connecting the subquery to a single boundary vertex, or that have multiple boundary edges connecting the subquery to more than one boundary vertex. Queries can contain more than one subquery as long as each subquery has only a single boundary edge.

Annotation requirements

Proximity requires that the boundary vertex of a subquery not be annotated. The boundary vertex of a subquery is that vertex which is connected to elements outside the subquery box by the boundary edge.

Nested subqueries

Proximity does not currently support executing queries containing nested subqueries.

Constraint restrictions

Proximity requires that constraints that cross subquery boundaries must obey specific annotation requirements. When a constraint compares items that do not fall in the same subquery, one of the elements being compared in the constraint must be optional, that is, it must be annotated with either `[0..]` or `[0..j]`. This annotation restriction is illustrated in the example query in Figure 6.18. This query compares an element in the subquery (the *2degrees* vertex) with an element in the main query (the *start* vertex). The *2degrees* vertex is annotated with `[0..]`, making the match to this vertex optional and thus making this a valid Proximity query.

The restriction that one of the constrained elements be optional applies whether one of the elements is in a subquery and the other is in the main query, or both elements are in two distinct subqueries. Constraints that compare elements within the same subquery are not subject to such annotation requirements.

Another Proximity implementation restriction addresses constraints when one of the constrained items is outside the subquery boundary while the other item is inside the subquery box. In this case, the item outside the subquery boundary cannot be annotated. Proximity prohibits constraints between an annotated item and an item within an annotated subquery.

Summary

Subquery structure

- Subqueries let you attach a numeric annotation to a connected set of vertices and edges instead of just a single vertex or edge.
- A subquery must be connected to one or more vertices in the main query.
- A subquery's *boundary edge* connects a vertex inside the subquery (the boundary vertex) to a vertex outside of the subquery.
- A subquery's *boundary vertex* is a vertex inside the subquery box that is connected to the main query by a boundary edge.
- The *inner structure* of a subquery is the set of objects and links inside the subquery box. A boundary edge is not part of a subquery's inner structure; a boundary vertex is included in the inner structure.

Subquery requirements

- The inner structure of a subquery must be a well-formed query in its own right.
- A subquery's inner structure may not be disconnected.
- All subqueries must be annotated.
- The boundary edge of a subquery must be annotated.
- The vertex outside the subquery connected to the boundary edge may not be annotated.
- Draw subqueries as single vertices to better understand query structure and annotation requirements.
- Constraints on elements within a subquery must obey the same rules that apply to any QGraph constraint.
- Two subqueries may not be connected by a single edge.

Implementation restrictions

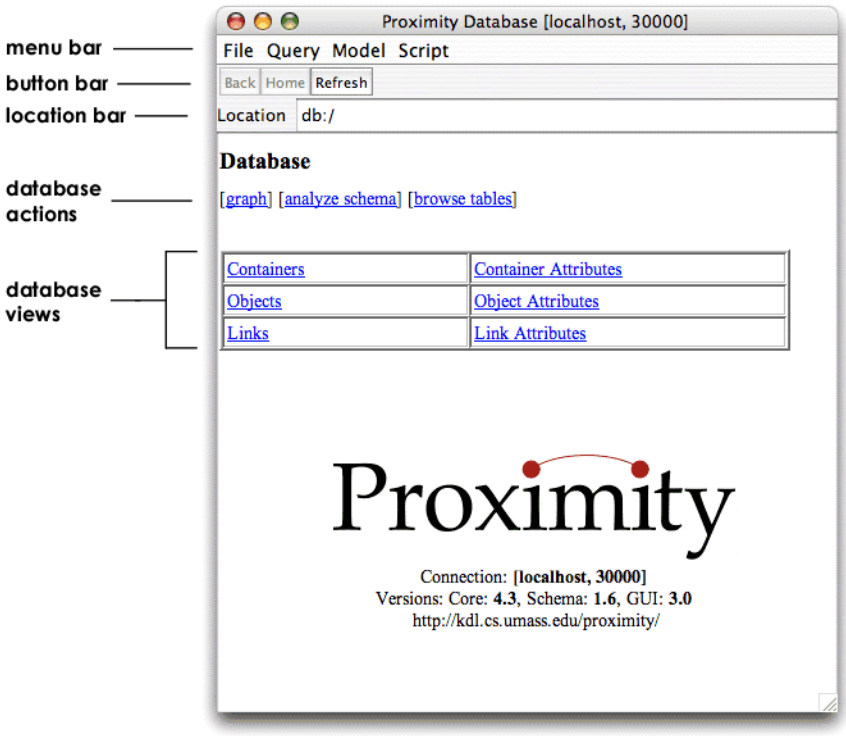
- A subquery must be connected to the main query by a single edge; multiple boundary edges are not permitted.
- The boundary vertex of a subquery may not be annotated.
- Constraints that cross subquery boundaries require that one of the items being compared be optional (have an annotation of `[0..]` or `[0..`*`j`*`]`).
- Proximity prohibits constraints between an annotated item and an item within an annotated subquery.
- Nested subqueries are not permitted.

Appendix A. Using the Proximity Query Editor

Introduction

QGraph is implemented in Proximity, an open source system for relational knowledge discovery. Proximity represents queries in an XML format described in Appendix B, *XML Representation*. Although you can create queries by editing the XML file, Proximity provides a Query Editor that lets you work with queries using a more intuitive graphical representation similar to that used in this guide. You can also edit, save, validate, and execute queries directly from the Proximity Query Editor. This chapter provides a brief overview of how to use the Proximity Query Editor. See the Proximity Tutorial for more information on using the Query Editor and on creating and executing queries in Proximity.

The Proximity Query Editor is available from the Proximity Database Browser, a browser-based interface to Proximity's functionality. To access the Query Editor, select **New Query** from the **Query** menu of the Proximity Database Browser, shown below.



Other options in the **Query** menu apply to existing query files:

- Select **Edit Query** to display a previously saved query in the Query Editor for editing.
- Select **Run Query** to execute a saved query.

This chapter describes how to use the Query Editor to create, edit, save, and execute QGraph queries. See the Proximity Tutorial for information on other methods of executing queries in Proximity.

Creating a query consists of

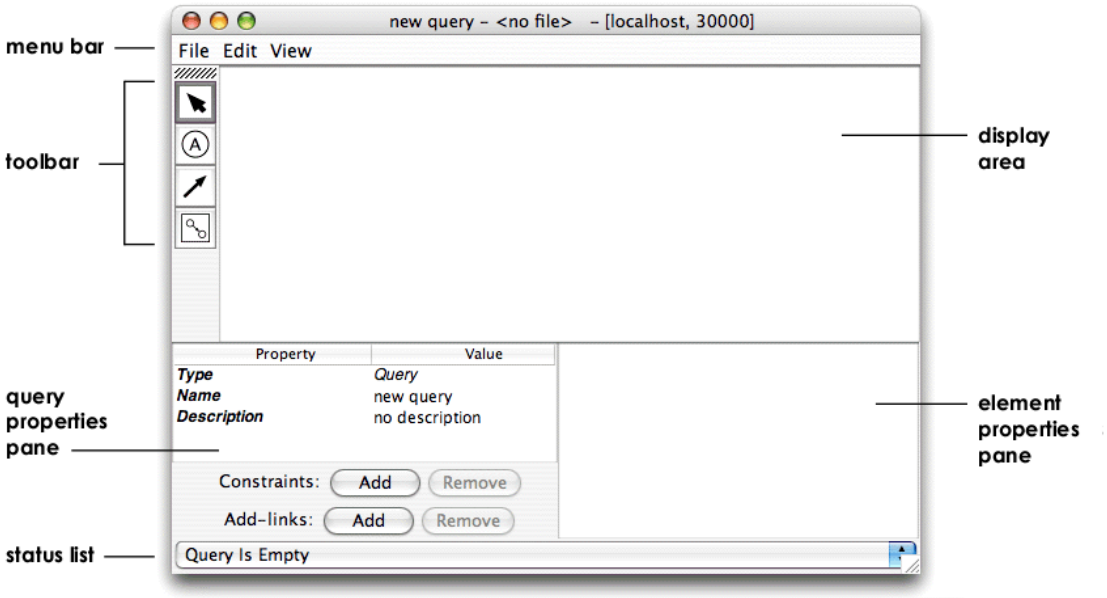
- defining one or more vertices
- defining edges linking those vertices
- adding any conditions and annotations to the vertices and edges

- defining any subqueries
- adding any constraints between separate query elements
- adding any database update (new link) specifications
- providing a name and description for the query
- making sure the query is valid

A query can be as simple as a single vertex. Therefore, with the exception of defining at least one vertex and validating the query, all of these steps are optional. Invalid queries will not execute and may not be correctly saved—we recommend that you check the status list for errors before executing or saving a query. The Query Editor imposes no ordering on these steps.

Working with the Query Editor

The figure below shows the Proximity Query Editor interface.



The Query Editor provides a large display area where you graphically create the query structure of vertices, edges, and subqueries. The Query Editor includes two properties panes at the bottom of the window for displaying and editing query properties such as conditions, annotations, and constraints. The query properties pane displays properties of the query as a whole. The element properties pane displays the properties of the selected query element. The element properties pane is blank when no query element is selected. The status list at the bottom of the window indicates whether the query is valid, and if invalid, provides a drop-down list of errors.

The Query Editor is mode based. Each toolbar button selects a tool, which in turn determines the current mode for the Query Editor. For example, clicking the vertex tool places the Query Editor in vertex mode. All subsequent clicks in the Query Editor display area place a new vertex at the indicated location. Selecting a new tool by clicking the corresponding toolbar button changes the mode and changes how clicks in the display area are interpreted.


Selection mode—clicking selects existing query elements for editing or deletion


Vertex mode—clicking creates a new vertex at the indicated location


Edge mode—clicking and dragging creates a new edge


Subquery mode—dragging a surrounding box indicates which vertices belong to a subquery

The sections below provide instructions for creating queries and working with the Query Editor.

To start the Query Editor:

You can start the Query Editor with no specified query (to create a new query), or specify an existing query to edit that query in the Query Editor.

- Choose the appropriate action from the **Queries** menu:
  - To create a new query, select **New Query**.
  - To edit an existing query, select **Edit Query**.

  If you are editing a previously saved query, Proximity display the Open dialog so that you can select the query to edit.

Proximity allows you to open multiple Query Editor windows. All Query Editor windows are closed when you close the Proximity Database Browser.

To close the Query Editor:

Proximity asks whether to save any unsaved queries before closing the Query Editor.

- In the **File** menu, select **Exit Query Editor**.

To change zoom level:

Changing the zoom level can be helpful when working with large queries. Zooming out lets you see the entire query structure at once while zooming in lets you easily work with individual query elements.

- To zoom in: In the **View** menu, select **Zoom In** or press Ctrl-=.
- To zoom out: In the **View** menu, select **Zoom Out** or press Ctrl- – (Ctrl-minus).
- To zoom contents to fit the display area: In the **View** menu, select **Zoom To Fit** or press Ctrl-0 (Ctrl-zero).
- To reset the zoom to its default setting: In the **View** menu, select **Reset Zoom** or press Ctrl-Backspace.

Keyboard shortcuts:

Many Query Editor commands can be invoked using keyboard shortcuts instead of using the mouse. You can open, close, save, and run queries, select tools, and change the Query Editor's mode (select tools) using keyboard shortcuts. Keyboard shortcuts are shown next to the corresponding command in the Query Editor's menus. The table below summarizes these shortcuts.

Command	**Keyboard shortcut**	**Command**	**Keyboard shortcut**
New query	Ctrl-N	Selection tool	Ctrl-1
Open query file	Ctrl-O	Vertex tool	Ctrl-2
Close query	Ctrl-W	Edge tool	Ctrl-3
Save query	Ctrl-S	Subquery tool	Ctrl-4
Run query	Ctrl-R	Flip edge direction	Ctrl-F
Select next item	Ctrl-→	Select previous item	Ctrl-←
Zoom in	Ctrl-=	Zoom to fit	Ctrl-0 (zero)
Zoom out	Ctrl-Minus	Reset zoom	Ctrl-Backspace

Working with Vertices and Edges

Vertices and edges are the foundation of QGraph queries. All queries must contain at least one vertex. Queries with multiple vertices must form a connected graph.

To add a vertex:

To add new vertices to a query, enter vertex mode and use the mouse to position the new vertex.

1. In the Query Editor toolbar, click  or press Ctrl-2 to choose the vertex tool and enter vertex mode.

   The Query Editor remains in vertex mode (all clicks in the Query Editor display area create a new vertex) until you change modes by choosing a different tool from the toolbar.
2. Click in the Query Editor display area. The Query Editor creates a new vertex.
3. [Optional] Provide a name for the new vertex. (Alternatively, you can continue using the default name provided by the Query Editor.)
   a. In the Query Editor toolbar, click  or press Ctrl-1 to choose the selection tool and enter selection mode.
   b. Click the vertex to be re-named. The Query Editor displays the vertex's properties in the element properties pane.
   c. In the element properties pane, click the current (default) name for this vertex. Edit the name and press Tab or Return.

To add an edge:

To add new edges to a query, enter edge mode and use the mouse to draw the new edge.

1. In the Query Editor toolbar, click  or press Ctrl-3 to choose the edge tool and enter edge mode.

   The Query Editor remains in edge mode (clicks in the Query Editor display area create a new edge) until you change modes by choosing a different tool from the toolbar.
2. Drag the mouse from one vertex to another vertex. The Query Editor creates a directed edge from the starting vertex to the ending vertex.
3. [Optional] Provide a name for the new edge. (Alternatively, you can continue using the default name provided by the Query Editor.)
   a. In the Query Editor toolbar, click  or press Ctrl-1 to choose the selection tool and enter selection mode.
   b. Click the edge to be re-named. The Query Editor displays the edge's properties in the element properties pane.
   c. In the element properties pane, click the current (default) name for this edge. Edit the name and press Tab or Return.

Edges may be directed or undirected. Directionality is an edge property and is set in the element properties pane.

To add an undirected edge:

1. Create a directed edge.
2. Click  or press Ctrl-1 to choose the selection tool.
3. Click the edge to be modified
4. In the element properties pane, change the value of Is Directed from **`true`** to **`false`**.
5. Press Tab or Return.

To change edge direction:

1. Click  or press Ctrl-1 to choose the selection tool.
2. Click the edge to be modified.
3. In the **Edit** menu, choose **Flip Edge Direction** or press Ctrl-F.

A self-link edge connects a single vertex to itself and matches links in the database that connect an object to itself.

To add a self-link (loop) edge:

1. Click  to choose the edge tool.
2. Click the target vertex without dragging. The Query Editor creates an edge that begins and ends at the target vertex (a self-link or loop edge).

To select a query element:

Use selection mode to re-position query elements in the display and to select existing query elements for editing, such as adding or modifying conditions and annotations on that element.

1. In the Query Editor toolbar, click  or press Ctrl-1 to choose the selection tool and enter selection mode.
2. Click the query element you want to select. The Query Editor draws handles around the selected element.

The Query Editor remains in selection mode (clicks in the Query Editor display select query elements) until you change modes by choosing a different tool from the toolbar.

To delete a vertex or edge:

Deleting a vertex also deletes any edges linked to that vertex. Make sure that your query remains connected after deleting any elements.

1. Click  or press Ctrl-1 to choose the selection tool.
2. Click the element to be deleted.
3. Press Delete.

To add or edit a condition:

Conditions and annotations are properties of a specific vertex or edge. Select the element to be modified and use the element properties pane to add or edit any conditions or annotations on that element.

Complex conditions must be entered using disjunctive normal form in prefix notation. See “Complex conditions in Proximity” (p. 21) for additional information on entering complex conditions in the Query Editor.

1. Click  or press Ctrl-1 to choose the selection tool.
2. Click the vertex or edge that requires a condition.
3. In the element properties pane, click in the Value column for Condition.
4. Enter the new condition or edit the value of the current condition.
5. Press Tab or Return.

To add or edit an annotation:

All edges adjacent to an annotated vertex must also be annotated. To help avoid validation errors due to unannotated edges, you can elect to automatically add a `[1..]` annotation to any new edge (see To change automatic annotation preferences, below, for instructions).

1. Click  to choose the selection tool and select the vertex or edge to be annotated.
2. In the element properties pane, click in the value column for Annotation.
3. Enter the annotation text.
4. Press Tab or Return.

To change automatic annotation preferences:

The Query Editor provides the option of automatically annotating every new edge with `[1..]`. Because edges adjacent to annotated vertices must be annotated, adding this annotation automatically can help to prevent validation errors. The `[1..]` annotation is the most common edge annotation and is typically the correct edge annotation for most queries involving annotated vertices. Adding a `[1..]` annotation to edges that are not adjacent to annotated vertices is rarely a problem for most datasets, although care should be taken to understand how this differs from unannotated edges when the dataset contains instances of multiple links connecting two objects.

- In the **Edit** menu, choose **Add [1..] To New Edges**.

Selecting the **Add [1..] To New Edges** command toggles the automatic edge annotation behavior:

- If automatic edge annotation is disabled, choosing this command turns **on** automatic edge annotation.
- If automatic edge annotation is enabled, choosing this command turns **off** automatic edge annotation.

Working with Updates

You can use QGraph queries to update (add, delete, or modify) database elements. However, implementation in Proximity is currently limited to using queries to add new links to the database.

To create new links:

You must specify an attribute value for all new links created by queries. The attribute must currently exist in the database and all new links (defined by a single link specification) must be assigned the same value for this attribute.

1. Create a query that matches the desired objects and links.
2. In the Add-link area at the bottom of the query properties pane, click **Add**. The Query Editor adds a temporary, example link specification, shown in the query properties pane.
3. Edit the temporary, example link specification to show the correct link information. Specify the vertices corresponding to the starting and ending objects and attribute value for the new links.
4. Press Tab or Return.

Working with Subqueries

The Query Editor identifies subqueries by drawing a box around the subquery elements and changing the color of the subquery's vertices from blue to red. You must define at least some of the subquery's component elements by creating vertices and edges before you can define the subquery itself. You can add additional elements to the subquery by creating new vertices and edges within the subquery box.

To add a subquery:

1. Create the vertices and edges that will form the subquery.
2. In the Query Editor toolbar, click  or press Ctrl-4 to choose the subquery tool.

3. Drag a rectangle around the subquery elements. The Query Editor draws a rectangle around the target elements and changes subquery vertices to red.

To delete a subquery:

1. Click  or press Ctrl-1 to choose the selection tool.
2. Select the subquery by clicking the subquery rectangle's boundary edge.
3. Press Delete.

The Query Editor removes the subquery rectangle but leaves the subquery's vertices and edges in place. Vertices that were in the subquery are changed back to blue.

Working with Queries

To edit an existing query:

You can load an existing (previously saved) query in the Query Editor. Editing an existing query is often an efficient way to create similar queries. You can elect to edit a saved query from either the Proximity Database Browser or from within the Query Editor.

To edit a saved query from the Proximity Database Browser:

1. From the **Query** menu, select **Edit Query**. Proximity displays the Open dialog.
2. Navigate the the directory containing the query file. Select the query to be edited and press **OK**.

To edit a saved query from the Query Editor:

1. In the **File** menu, choose **Open** or press Ctrl-O. Proximity displays the Open dialog.
2. Navigate the the directory containing the query file. Select the query to be edited and press **OK**.

Proximity displays the selected query in the Query Editor display area.

To add a constraint:

Because constraints compare the identity or attribute values of two different query elements, constraints are properties of a query as a whole rather than of any individual query element.

1. In the Constraints area at the bottom of the query properties pane, click **Add**. The Query Editor adds a temporary, example constraint, shown in the query properties pane.
2. Edit the temporary, example constraint to show the correct constraint.
3. Press Tab or Return.

To delete a constraint:

1. Select the constraint you want to delete by clicking the corresponding entry in the query properties pane's Value column.
2. In the Constraints area, click **Remove**. The Query Editor deletes the selected constraint.

To add query information:

To help you remember the purpose of or other information about a query, you can save meta-information with your query. The XML query format supports saving a name and description for each query.

To add a query name:

1. In the query properties pane, click the value for Name.
2. Change the value to the new query name.
3. Press Tab or Return.

To add a query description:

1. In the query properties pane, click the value for Description.
2. Change the value to the correct query description.
3. Press Tab or Return.

To save a query:

It is frequently useful to save a query for future use. Saved queries are written out to an XML file. The Query Editor understands this XML format and can reload saved queries for future use or editing. Make sure that your query is valid before saving it—invalid queries may not be saved correctly or may not display correctly in the Query Editor when reloaded.

- In the **File** menu, choose **Save** or press Ctrl-S.
  - If the query has been saved previously, the new information is written out to the associated file.
  - If the query has not been saved previously, Proximity displays the Open dialog.
    1. Navigate to the target directory.
    2. Enter a filename in the Save As box.
    3. Click **Save**.

Once a query has been saved to an XML file, you can save any new changes by selecting **Save** or pressing Ctrl-S.

To validate a query:

The Query Editor checks only that the query is syntactically correct; it cannot verify semantic information. For example, it reports an error if the query is disconnected but cannot check whether attributes listed in conditions or constraints actually exist in the database.

A query must be valid before it can be executed. Additionally, invalid queries may not be correctly displayed in the Query Editor when reloaded after saving. The Query Editor's status list shows whether the currently displayed query is valid or not. Check the status bar before executing or saving queries.

- Check the message displayed in the status list at the bottom of the Query Editor window.
  - If the query contains no syntactic errors, the status list shows `Query is Valid`.
  - If the query contains errors, the status list shows the number of validation errors.

If the status list shows one or more errors, click the up/down arrows to the right of the status list to display the specific errors for this query. Fix any errors before executing or saving the query.

To execute a query:

You can execute queries from the Proximity Database Browser or from within the Query Editor.

To execute the current query (Query Editor):

1. In the **File** menu, choose **Run** or press Ctrl-R. Proximity asks you to provide a name for the container that will hold the query's results.
2. Enter a name for the container and press OK. Proximity prints a trace of the query's execution in a new window.
3. Close the trace window after query execution completes and you have finished examining the trace.

To execute a saved query (Proximity Database Browser):

1. From the **Query** menu, select **Run Query**. Proximity asks you to provide a name for the container that will hold the query's results.
2. Enter a name for the container and press OK. Proximity prints a trace of the query's execution in a new window.
3. Close the trace window after query execution completes and you have finished examining the trace.

Appendix B. XML Representation

This appendix describes the major elements used in the query XML format. The Query Editor saves queries in this XML format. You can create queries by using the Query Editor or by creating an XML file in a text editor. The DTD for the Proximity XML query format is located in `$PROX_HOME/resources/graph-query.dtd`.

Empty elements as well as elements whose content consists solely of PCDATA text are not included in this appendix. The appendix also omits detailed descriptions of `and`, `or`, and `not` as their use is reasonably intuitive and a full description adds little useful information. Interested readers are encouraged to examine the examples under “Conditions”, “Constraints”, and “Test Expressions” or the DTD file.

Declarations

Document type declaration

The XML import file must include the following document type declaration:

```
<!DOCTYPE graph-query SYSTEM "graph-query.dtd">
```

Note that the document type declaration uses a system identifier for the DTD. Proximity expects to find the `graph-query.dtd` file in the same directory as the XML data file.

The `graph-query` root element

The root element for a query file is `graph-query`. The `graph-query` tag must appear immediately after the document type declaration. The query XML file ends with the closing `</graph-query>` tag.

The `graph-query` specifies the name of the query in the required `name` attribute. Queries must include a `query-body` element, and may optionally include a description and layout information for the Query Editor.

`<graph-query>`

Attributes

`name`	required	The name of the query

Children

`description`	zero or one	String describing the query
`query-body`	exactly one	Specifies the graph pattern to be matched
`editor-data`	zero or one	Specifies layout information for the query editor

Content Model

```
(description?, query-body)
```

Example

```
<graph-query name="1d-clusters">
  <description>Finds 1d-clusters</description>
  <query-body> see query-body </query-body>
  <editor-data> see editor-data </editor-data>
</graph-query>
```

Query Body

The required `query-body` element specifies the structure and requirements for the pattern to be matched to structures in the database.

<query-body>

Attributes None.

Children

`vertex`	one or more	Describes a vertex in the query; matches an object in the data
`edge`	zero or more	Describes an edge in the query; matches a link in the data
`subquery`	zero or more	Specifies a subquery within the current query
`constraint`	zero or one	Specifies a constraint between query elements
`add-link`	zero or more	Specifies new links to be added to the database

Content Model

```
(vertex, (edge | vertex)*, subquery*, constraint?, add-link*)
```

Example

```
<query-body>
  <vertex name="A"> see vertex </vertex>
  <edge name="Y"> see edge </edge>
  <subquery> see subquery </subquery>
  <constraint> see constraint </constraint>
  <add-link> see add-link </add-link>
</query-body>
```

Vertices

The `vertex` element defines a vertex in the query. The name of the vertex, as defined in the query, is provided in the required `name` attribute. The `vertex` element may contain an optional `condition` element and an optional `numeric-annotation` element. Vertices in the query are matched to objects in the database.

<vertex>

Attributes

`name`	required	Vertex name as used in the query; vertex names are case sensitive

Children

`condition`	zero or one	Restrictions on the link's attribute values
`numeric-annotation`	zero or one	Cardinality requirements for matching links

Content Model

```
(condition?, numeric-annotation?)
```

Example

```
<vertex name="A">
  <condition> see condition </condition>
  <numeric-annotation> see numeric-annotation </numeric-annotation>
</vertex>
```

Edges

The `edge` element defines an edge in the query. The name of the edge, as defined in the query, is provided in the required `name` attribute. The `edge` element may contain an optional `condition` element and an optional `numeric-annotation` element. Edges in the query are matched to links in the database.

Edges must indicate which vertices they connect and whether or not the direction of the edge is taken into account when matching the query to database structures.

<edge>

Attributes

`name`	required	Edge name as used in the query; edge names are case sensitive

Children

`vertex1`	exactly one	The starting vertex; must match the name (including case) of a vertex in this query
`vertex2`	exactly one	The ending vertex; must match the name (including case) of a vertex in this query
`directed`	exactly one	Whether to consider link direction in matching; must be "true" or "false"
`condition`	zero or one	Restrictions on the link's attribute values
`numeric-annotation`	zero or one	Cardinality requirements for matching links

Content Model

```
(vertex1, vertex2, directed, condition?, numeric-annotation?)
```

Example

```
<edge name="Y">
  <vertex1>A</vertex1>
  <vertex2>B</vertex2>
  <directed>true</directed>
  <condition> see condition </condition>
  <numeric-annotation> see numeric-annotation </numeric-annotation>
</edge>
```

Conditions

A `condition` is a boolean combination of `test` elements, each of which defines a condition on the vertex (or link). Condition `test` elements can be combined with the `and`, `or`, and `not` elements using disjunctive normal form.

> Legal operators for conditions are `eq`, `lt`, `le`, `gt`, `ge`, and `ne`. You can also use the `exists` operator to define an existence condition, requiring only that an object or link have a specific attribute without concern for its value.

<condition>

Attributes None.

Children

`test`	member of a required set	A requirement for the corresponding vertex's or edge's attribute values
`and`	member of a required set	Requires all of the included tests be satisfied
`or`	member of a required set	Requires one of the included tests be satisfied
`not`	member of a required set	Requires that the specified test not be satisfied

Content Model

```
(or | and | not | test)
```

Example

```
<condition>
  <and>
    <test>
      <operator>eq</operator>
      <attribute-name>objtype</attribute-name>
      <value>person</value>
    </test>
    <not>
      <test>
        <operator>exists</operator>
        <attribute-name>cluster-coeff</attribute-name>
      </test>
    </not>
  </and>
</condition>
```

Numeric Annotations

The `numeric-annotation` element includes a required `min` element indicating the annotation's lower bound and an optional `max` element indicating the annotation's upper bound. You must specify both upper and lower bounds for exact annotations. Numeric annotations are optional child elements of `vertex`, `edge`, and `subquery` elements.

<numeric-annotation>

Attributes None.

Children

`min`	exactly one	Lower bound of the annotation
`max`	zero or one	Upper bound of the annotation

Content Model

```
(min, max?)
```

Example

```
<numeric-annotation>
  <min>0</min>
  <max>3</max>
</numeric-annotation>
```

Subqueries

Subqueries must be well-formed queries in their own right. They must have one or more edges that connect the subquery to a vertex or vertices in the main query. Subqueries can be nested. (Nested subqueries are allowed by QGraph and the DTD, but are prohibited by the current Proximity implementation.)

Subqueries are always annotated, as shown in the example below. The boundary edge of a subquery must be annotated. The boundary vertex of a subquery cannot be annotated.

<subquery>

Attributes None.

Children

`vertex`	one or more	A vertex within the subquery
`edge`	one or more	An edge within the subquery
`subquery`	zero or more	A nested subquery
`constraint`	zero or one	Specifies a constraint between query elements
`numeric-annotation`	exactly one	Cardinality requirements for the subquery

Content Model

```
(vertex, (edge | vertex)*, subquery*, numeric-annotation)
```

Example

```
<vertex name="B">
</vertex>
<subquery>
  <vertex name="C">
  </edge<
  <edge name="Z">
    <vertex1>B</vertex1>
    <vertex2>C</vertex2>
    <directed>true</directed>
    <numeric-annotation>
      <min>1</min>
    </numeric-annotation>
  </edge>
  <numeric-annotation>
    <min>1</min>
  </numeric-annotation>
</subquery>
```

Constraints

A `constraint` defines a `test` element that compares the identity or attribute values of two vertices or two edges. You cannot mix vertices and edges within a constraint, but queries may include both vertex constraints and edge constraints. Because constraints apply to two different query elements, they occur at the top level within a query, rather than under any specific element. Specifically, you cannot include constraints within `subquery` elements. You can combine multiple constraints with `and` elements.

> Legal operators for constraints are `eq`, `lt`, `le`, `gt`, `ge`, and `ne`.

Constraints take slightly different forms depending on whether they are comparing attributes or identity. In the example below, the first `test` requires that query items (vertices or edge) A and B have the same value for attribute *`attr`*. The second `test` requires that query items B and C do not map to the same entity in the database.

Vertex and edge names used in the constraint, as specified by the `item-name` element, must match the name and case of vertices and edges defined for this query.

<constraint>

Attributes None.

Children

`test`	member of a required set	Requirement for query elements' attribute values
`and`	member of a required set	Requires all of the included tests be satisfied
`or`	member of a required set	Requires one of the included tests be satisfied; allowed by DTD but prohibited by current Proximity implementation
`not`	member of a required set	Requires that the specified test not be satisfied; allowed by DTD but prohibited by current Proximity implementation

Content Model

Although the DTD does not enforce this, Proximity requires that constraint `test` elements may only be combined with `and`.

```
(or | and | not | test)
```

Example

```
<constraint>
  <and>
    <test>
      <operator>eq</operator>
      <item>
        <item-name>A</item-name>
        <attribute-name>objtype</attribute-name>
      </item>
      <item>
        <item-name>B</item-name>
        <attribute-name>objtype</attribute-name>
      </item>
    </test>
    <test>
      <operator>ne</operator>
      <item>
        <item-name>B</item-name>
        <id/>
      </item>
      <item>
        <item-name>C</item-name>
        <id/>
      </item>
    </test>
  </and>
</constraint>
```

Test Expressions

Both conditions and constraints use test expressions, represented by the `test` element, to describe the required comparison. Conditions test attribute values for a single database entity whereas constraints compare attribute values or IDs for two database entities. Proximity uses the same DTD elements to represent these query constructs, although each will be realized in somewhat different forms in the XML representation. See the examples under “Conditions” and “Constraints” for examples of each form.

`<test>`

Attributes None.

Children

The cardinality of `test`’s children, listed below, reflect the legal uses of these elements based on a query’s semantics rather than the restrictions imposed by the DTD.

`operator`	exactly one	The comparison operator; used in both conditions and constraints
`attribute-name`	zero or one	Name of attribute(s) to be compared; used in `condition` elements
`value`	zero or one	Value to be compared to that for the specified attribute; used in `condition` elements
`item`	zero or two	Provides data for one of the items being compared; used in `constraint` elements

Content Model

```
(operator, (attribute-name | value | item),
           (attribute-name | value | item)?)
```

Example See “Conditions” and “Constraints”.

<item>

The item element describes one of the two items being compared in a constraint. The name of the item is contained in the required item-name element. Attribute constraints use the attribute-name element to indicate which of this item’s attribute’s values are to be compared. Identity constraints use the id element to indicate that the constraint compares IDs rather than attribute values.

Attributes None.

Children

item-name	exactly one	Name of vertex or edge whose attribute is being compared
attribute-name	exactly one	Name of attribute to be compared
id	zero or one	Indicates constraint is to compare IDs rather than attribute values

Content Model

```
(item-name, (attribute-name | id))
```

Example See “Constraints”.

Update Functionality

<add-link>

QGraph provides complete update capabilities, however, only a portion has been implemented in Proximity to date. Proximity permits adding links to the database through the use of QGraph queries.

The add-link element describes how and where to add new links to the database. The element has two children, representing the query vertices that match the starting and ending objects for the new links. Each new link must be given a specified attribute value, specified by the element’s attrname and attrval attributes. Attribute values for new links must be the same for all links created by a single specification.

Attributes

attrname	required	The name of the attribute to be added to each newly created link; the attribute must already exist in the database
attrval	required	The value to be assigned to the new link’s attribute; value must be static

Children

`vertex1`	exactly one	Name of vertex matching starting objects for the new links; vertex names are case sensitive
`vertex2`	exactly one	Name of vertex matching ending objects for the new links; vertex names are case sensitive

Content Model

```
(vertex1, vertex2)
```

Example

```
<add-link attrname="linktype" attrval="newlink">
  <vertex1>start-obj</vertex1>
  <vertex2>end-obj</vertex2>
</add-link>
```

Editor data

Proximity stores layout information when you save a query in the Query Editor, letting the Query Editor re-display the query as currently laid out the next time it's opened. Layout information, in the form of *x* and *y* coordinates, is stored in the `editor-data` element in the query XML file.

The `editor-data` element contains one `vertex-location` element for each vertex in the query. Each `vertex-location` element includes three required attributes, representing the name and coordinates of the vertex.

Proximity currently only saves the location of vertices in the Query Editor; edge location is determined by vertex location and is thus not saved.

<editor-data>

Attributes None.

Children

`vertex-location`	one or more	Vertex location in the query editor

Content Model

```
(vertex-location+)
```

Example See `vertex-location`.

<vertex-location>

Attributes

`name`	required	Vertex name as used in the query; vertex names are case sensitive
`x`	required	The x coordinate describing the vertex's location in the Query Editor
`y`	required	The y coordinate describing the vertex's location in the Query Editor

Children None.

Content Model

```
EMPTY
```

Example

```
<editor-data>
   <vertex-location name="vertex1" x="514" y="99" />
   <vertex-location name="vertex2" x="314" y="232" />
</editor-data>
```

References

[Blau, Immerman, and Jensen, 2002] H. Blau, N. Immerman, and D. Jensen. *A Visual Language for Querying and Updating Graphs*. University of Massachusetts Amherst, Computer Science Department Technical Report 2002-037. 2002.

Glossary

aggregation	Aggregation is the process of grouping data. For example, in Proximity the models aggregate attribute values to create features. Example aggregation functions include average, count, degree, and proportion.
ambiguous	A query is ambiguous if it admits more than one interpretation. For example, a query with adjacent annotated vertices is ambiguous because neither annotation takes precedence over the other. Ambiguous queries are not permitted in QGraph.
attribute	An attribute is a name-value pair that represents additional information about the database entity on which it appears. Proximity attribute values are sets, which may contain zero or more values and which may include specific values more than once. Proximity supports attributes for objects, links, subgraphs, and containers. Objects and links can have a variable number of attributes and attributes can have a variable number of values.
attribute constraint	Attribute constraints compare the attribute values of two database entities, such as two objects or two links.
attribute value condition	Attribute value conditions restrict query matches to objects or links having the attribute value specified in the condition.
bias	Bias is the systematic difference between the values predicted by the learned model and actual values.
boundary edge	A boundary edge is a query edge that crosses a subquery box.
boundary vertex	A boundary vertex is a vertex within a subquery box that is connected to query elements outside the subquery box by a boundary edge.
bounded range	A numeric annotation of the form [*i*..*j*], a bounded range specifies that there must be at least *i* and no more than *j* corresponding elements to match the query.
comparable types	Conditions and constraints can compare attribute values of the same type (e.g., `STR` with `STR`). In addition, you can compare attributes of type `DBL` with `FLT` and `INT`, and attributes of type `FLT` with `INT`.
complex condition	In principle, a complex condition can be any boolean combination of simple conditions. Proximity's implementation of QGraph restricts complex conditions to disjunctive normal form.
condition	A condition restricts query matches by requiring that database items match specified attribute values. For example, the condition ObjType = person on a vertex requires that that vertex only match objects with a ObjType attribute having a value of person. Existence conditions work similarly, but only require that the corresponding object or link have *any* value for the specified attribute without caring what that value is.
constraint	Constraints compare the attribute values or identities of two distinct query elements. Only pairs of objects or links that satisfy the constraint match the corresponding query.
container	A container is a collection of subgraphs usually created as the result

	of executing a query.
core vertex	The core vertex is the vertex from a QGraph query that corresponds to the object to be classified. Objects and links connected to the core vertex define the *local neighborhood* to be used in classifying the core object.
directed edge	A directed edge in a query requires matching links in the database follow the same direction as the query edge.
disconnected query	A query must be a single connected graph. A query containing more than one connected component is considered to be disconnected and thus not allowed.
disjunctive normal form	Boolean formulae expressed as a disjunction of conjunctions are said to be in disjunctive normal form. Such formulae consist of a series of disjunctions (expressions `OR`ed together) where each expression is either a terminal expression (a simple proposition in the case of boolean logic or a simple condition in the case of Proximity conditions), the negation of a terminal expression, or the conjunction (expressions `AND`ed together) of terminal expressions.
edge	Proximity uses the terms *vertex* and *edge* to refer to entities in a query and the terms *object* an *link* to refer to entities in the data. An edge in a query matches corresponding links in the data.
exact annotation	An exact annotation requires a specific number of matches for example, [2], rather than a range in the number of matches such as [2-4] or [2..].
existence condition	Existence conditions check to see if the corresponding object or link has a value for the specified attribute. An existence condition is satisfied if the corresponding item has any value for the attribute, regardless of what that value is.
identity constraint	Identity constraints compare the identity (OID) of two database entities.
inner structure	The inner structure of a subquery is the set of vertices and edges that fall entirely within the subquery box. The boundary edge and subquery annotation are not part of the inner structure.
isomorphic	Isomorphic subgraphs have the same structure in terms of nodes and edges but may have different member objects and links.
knowledge discovery	Knowledge discovery seeks to find useful patterns in large and complex databases. More specifically, relational knowledge discovery focuses on constructing useful statistical models from data about complex relationships among people, places, things, and events.
link	A link is a directed binary relation connecting two objects in a Proximity database. We use *link* to refer to relations in a database and *edge* to refer to relations in a query.
loop	see *self link*
mirror match	A mirror match to a QGraph query is a pair of otherwise identical subgraphs that differ only in terms of how the subgraph's objects and links match the query's vertices and edges. Specifically, in a mirror match, two objects reverse positions from their original locations in the other matching subgraph.

multi-dimensional attribute	A multi-dimensional (or multi-column) attribute in Proximity contains more than one value. For example, a location attribute might contain two values corresponding to the *x* and *y* coordinates of the item's position. Proximity permits the inclusion of multi-dimensional attributes in data but does not yet support their use in queries or models.
name	Each vertex and edge in a Proximity query is assigned a name (label) that is used to identify the corresponding items in the matching subgraphs.
negated element	A negated query element (vertex or edge) has a numeric annotation of `[0]`; there must be no corresponding element in the data when matching the query.
numeric annotation	Numeric annotations place limits on the number of isomorphic substructures that can occur in matching portions of the database. Annotations also serve to group isomorphic structures into a single subgraph rather than producing multiple matches.
object	An object is a Proximity database entity that represents things in the world such as people, places, and events.
optional element	Optional query elements define structures that can be, but are not required to be present in the data in order to match the query. They are annotated with `[0..`*n*`]` or `[0..]`.
precedence	Precedence determines the order in which query elements are considered in matching the database. An annotated vertex has precedence over an annotated edge because the match process first finds objects that match the annotated vertex and then finds links from that object to match the corresponding edges in the query.
prefix notation	Prefix notation places an expression's operator before its operands. For example, the expression *a* + *b* becomes + *a b* when using prefix notation.
propositionalizing data	Propositionalizing data "flattens" relational data by moving attributes of related items to the objects of interest. For example, in a system that reasons about movies, propositionalizing an attribute of a director, such as date-of-birth, might place that attribute on related movie objects as director-date-of-birth.
QGraph	QGraph is a visual query language designed to support knowledge discovery in large graph databases.
relational data	As used in this document, relational data refers to data that explicitly represent relations among objects as first-class entities. Relational data are represented by a directed graph in which nodes represent objects from the domain of interest and links represent relationships between pairs of objects.
schema	A database's schema determines how the data are represented, i.e., which data entities are mapped to objects, which are mapped to links, and what constitutes attributes of those objects and links. Proximity also uses an internal schema that determines how Proximity database structures map to MonetDB data structures.
self link	A self-link connects a node with itself. For example, web pages often contain hyperlinks that jump to another part of the current page, linking the web page to itself.

star query	A one-dimensional star query includes a *core vertex* and one or more neighboring vertices, each connected to the core vertex by a single edge. Typically, the neighboring vertices (and therefore the corresponding edges as well) are annotated with an unbounded range, permitting any number of matching neighbor objects and links. Star queries can be extended to additional dimensions through the use of subqueries.
subgraph	A subgraph is a connected portion of a graph. QGraph queries return subgraphs as matches to the query.
subquery	A subquery is a connected subgraph of vertices and edges that can be treated as a logical unit. Subqueries allow grouping and limiting of complex query structures rather than just individual query elements.
type	A type is a label that categorizes instances in a data set, usually represented as an attribute-value pair assigned to an object or link. For example, a data set might contain objects that represent three types of entities: actors, movies, and studios. Proximity does not require a type attribute, but users may specify zero, one, or many attributes that provide type information. These attributes can be practical for the user, but in fact Proximity does not distinguish attributes representing type information from attributes representing other kinds of information.
unbounded range	A numeric annotation of the form [*i*..], an unbounded range specifies that there must be at least *i* corresponding element(s) to match the query.
undirected edge	An undirected edge in a query matches links in the database regardless of the link's direction.
validation	Validation is the process of ensuring that an XML document obeys the structure specified in the associated DTD. In Proximity, queries (which are represented internally in XML) must validate against the DTD in `graph-query.dtd`. Because DTDs cannot specify semantic content or enforce all potential syntactic requirements, a syntactically valid query may still be illegal under the rules of QGraph.
vertex	Proximity uses the terms *vertex* and *edge* to refer to entities in a query and the terms *object* an *link* to refer to entities in the data. A vertex in a query matches corresponding objects in the data.
well formed	A well-formed query conforms to all rules governing how queries may be legally structured.

Index

A

adjacency requirements, 26, 30, 32, 42-44, 78
  for negated/optional vertices, 37, 39
ambiguous queries, 44, 55
AND (see logical operators)
annotations (see numeric annotations)
attribute constraints, 51-52
attribute value conditions, 13-14
  evaluating, 16
  XML representation, 84
attributes
  graphical representation, 3
  missing, 16, 18, 55
  multi-dimensional values, 21
  multiple values, 1, 16, 53
  propositionalizing, 2
  representation in Proximity, 21, 47
  type information, 3, 13
autocorrelation, 2

B

bias, 2
boolean operators (see logical operators)
boundary edges (of a subquery), 59, 62, 70
  multiple, 69
boundary vertices, 70, 70
bounded ranges for numeric annotations, 26

C

case matching in Proximity, 21
changing (see editing)
closing the Query Editor, 75
comparable data types, 57
comparison operators
  in constraints, 48, 51
  in conditions, 14, 21
complex conditions, 19-20, 21, 77
conditions
  attribute value, 13-14
  compared to negated annotations, 40
  comparison operators in, 14, 21
  complex, 19-20, 21
  creating in Query Editor, 77
  evaluating, 16, 19, 19
  existence, 14-16
  graphical representation, 13
  implementation restrictions, 21
  logical operators in, 20, 21
  multiple (see complex conditions)
  XML representation, 84
constraints
  on annotated elements, 55
  attribute, 51-52
  comparison operators in, 48, 51
  creating in Query Editor, 79
  evaluating, 53-55
  general form, 51
  graphical representation, 47
  identity, 47-51
  implementation restrictions, 55, 57
  mixing vertices and edges, 57
  multiple, 52, 57
  on negated/optional elements, 55, 70
  subqueries and, 63-67, 70
  XML representation, 86
containers, 1, 5
  names of, 80
  updating, 11
core object or vertex, 28, 29
creating queries, 73-80

D

data type, 57
databases
  disconnected, 51
  editing, 11
  graphical representation, 3
  heterogeneous, 14
  loops in, 10
  Proximity, 1, 30
    attribute requirements, 14, 47
    link requirements, 5, 51, 65
  schema, 30
degree disparity, 2
deleting
  database elements, 11
  query elements, 77
descriptions of queries, 80
directed edges (see edges)
disconnected databases, 51
disconnected queries, 36, 38, 39, 61
disconnected subgraphs, 11
disconnected subqueries, 61
disjunctive normal form, 21, 77, 84
DTD for query files, 81
duplicates in subgraphs, 8, 25, 48

E

edge mode, 74
edges
  automatic annotation of, 78
  changing direction, 77
  creating in Query Editor, 76
  directed vs. undirected, 5, 9, 11, 76, 83
  and links, 3
  names of, 5, 76, 83
  negated, 38
  on subquery boundaries, 85
  optional, 38, 45

self-link (loop), 77
XML representation, 83
editing
database elements, 11
queries, 73, 75, 79
efficiency in Proximity, 11, 21, 45
element properties pane, 74, 76, 77
equivalent subgraphs, 8, 49
errors in queries, 74, 78, 80
evaluating
attribute value conditions, 16
complex conditions, 19
constraints, 53-55
existence conditions, 19
exact annotation, 26
examples, 4
executing queries, 73, 75, 80
existence conditions, 14-16
evaluating, 19
XML representation, 84
exiting the Query Editor, 75

G

graph-query.dtd, 81
graphical representation
conditions, 13
constraints, 47
databases, 3
numeric annotation, 26
queries, 4, 5
subgraphs, 4
subqueries, 59
grouping in query results, 25, 26, 59

I

identity constraints, 47-51
implementation (see Proximity QGraph implementation)
incident edges (see subqueries, boundary edges)
inner structure of a subquery, 61

K

keyboard shortcuts, 75

L

labels (see names)
layout of queries, 89
limiting query matches, 14, 25
links, 1, 3
adding with queries, 78
directionality, 5, 9
grouping in query results, 26
loops, 10
multiple
and annotations, 33
matching queries, 6
logical operators
in constraints, 52, 57
in conditions, 20, 21
loops in databases, 10

M

matching queries
grouping isomorphic structures, 25
with negated elements, 37, 40, 42
negated elements vs. conditions, 40
with optional elements, 37
requirements, 5
satisfying conditions, 16
satisfying constraints, 53-55
structure, 13
meta-information in queries, 79
mirror matches, 49
missing attribute values, 16, 18, 55
modes in Query Editor, 74, 75
modifying (see editing)
multi-dimensional attribute values, 21
multiple annotations, 32
multiple conditions (see complex conditions)
multiple constraints, 52, 57
multiple subqueries, 67

N

names
of attributes, 21
of containers, 80
conventions, 5, 14
of edges, 5, 76, 83
of queries, 79
in subgraphs, 6
of vertices, 5, 76, 82
negated
edges, 38
query elements, 36
in constraints, 55
effect on query results, 37, 42
vs. conditions, 40
vertices, 39
nested subqueries, 68-69
NOT (see logical operators)
numeric annotations
adding automatically, 78
adjacency requirements, 26, 30, 32, 42-44, 78
for negated/optional vertices, 37, 39
bounded ranges, 26
and constraints, 55
creating in Query Editor, 78
on edges, 29, 45
exact annotation, 26, 31
general form, 26
graphical representation, 26
implementation restrictions, 45
multiple, 32
negated elements, 36, 55

omitted, 31
optional elements, 36, 45, 55
on subqueries, 59, 62, 67, 68, 70
unbounded ranges, 26
on vertices, 26, 43
XML representation, 85

O

objects, 1, 3
opening queries, 75, 79
operators
comparison, 14, 21
logical, 20, 21
optional
edges, 38, 45
query elements, 36
in constraints, 55, 70
effect on query results, 37
vertices, 39
OR (see logical operators)

P

precedence of vertex and edge annotations, 27, 44
prefix notation, 21, 77
properties panes, 74
propositionalizing attribute values, 2
Proximity
Database Browser, 73, 79, 80
databases, 1, 30
attribute requirements, 14, 47
link requirements, 5, 51, 65
QGraph implementation, 1, 3, 10
conditions, 21
constraints, 55, 57
numeric annotations, 45
subqueries, 69
Query Editor, 73-80
query efficiency, 11, 21, 45
query results in, 10
representation
attributes, 21, 47
type information, 3, 13
system, 1

Q

queries
ambiguous, 44, 55
core vertex, 29
creating in Query Editor, 73
definition, 5
descriptions of, 80
disconnected, 36, 38, 39, 61
editing in Query Editor, 73, 75, 79
errors in, 74, 78, 80
executing, 73, 75, 80
graphical representation, 4, 5
layout, 89
limiting matches, 14, 25
meta-information, 79
names of, 79
results, 1, 14, 25, 37, 37, 42, 59
(see also subgraphs)
rule summaries, 11, 23, 46, 58, 70
star, 28, 61, 69
update, 88
validity of, 74, 78, 80
well-formedness, 36, 61, 85
XML representation, 73, 80, 81-90
Query Editor, 73-80
closing queries, 75
creating annotations, 78
creating conditions, 77
creating edges, 76
creating vertices, 76
deleting elements, 77
editing queries, 73, 75, 79
element properties pane, 74, 76, 77
executing queries, 73, 75, 80
exiting, 75
keyboard shortcuts, 75
modes, 74, 75
opening queries, 75, 79
query properties pane, 74
saving queries, 75, 80
selecting elements, 77
starting, 75, 79
status list, 74, 80
zooming in display, 75
query properties pane, 74

R

ranges in numeric annotations, 26
removing (see deleting)
representation in Proximity
attributes, 21, 47
database design, 30
type information, 3, 13
results of queries, 1, 14, 25, 59
(see also subgraphs)
and negated elements, 37, 40, 42
and optional elements, 37

S

satisfying (see evaluating)
saving queries, 75, 80
schema (see database schema)
SELECT (in SQL), 14
selection mode, 74, 77
self-links, 10, 77
SQL, 1, 14
star queries, 28, 61, 69
starting the Query Editor, 75, 79
status list, 74, 80
structure of queries, 5

subgraphs, 1
  contents, 14
  disconnected, 11
  duplicated elements in, 8, 25, 48
  equivalent subgraphs, 8, 49
  graphical representation, 4
  names in, 6
subqueries
  adding elements to, 78
  annotation rules, 67
  boundary edges, 59, 62, 69, 70
  boundary vertices, 70, 70
  constraints and, 63-67, 70
  creating in Query Editor, 78
  disconnected, 61
  graphical representation, 59
  implementation restrictions, 69
  inner structure of, 61
  multiple, 67, 68
  nested, 68-69
  numeric annotations on, 59, 62
  requirements, 85
  visualizing, 59, 62, 67, 69
  XML representation, 85
subquery mode, 74

T

test expression, XML representation, 87
type of an object or link, 3, 13
  (see also data type)

U

unbounded ranges for numeric annotations, 26
undirected edges (see edges)
updating data, 78

V

validity of a query, 74, 78, 80
vertex mode, 74
vertices
  creating in Query Editor, 76
  names of, 5, 76, 82
  negated (and adjacent edges), 37
  and objects, 3
  on subquery boundaries, 85
  XML representation, 82

X

XML representation of queries, 73, 80, 81-90

Z

zooming, 75