Relation Component Requirements Specification

1. Scope

1.1 Overview

This component consists of data structures which represent tuples and relations. It also includes a relationally complete set of operations for manipulating those data structures.

1.2 Logic Requirements

1.2.1 Tuples

A tuple is an unordered set of values. Each value is associated with an attribute name and is represented by a specified type. The set of attributes contained in a tuple creates a tuple type.

Example:

The ‘pet’ type may be defined as:

(name:java.lang.String age:int species:com.petstore.common.Species weight:int)

1.2.2 Tuple Constraints

The component will include the ability to define constraints for a tuple which can be used to validate each field in the tuple.

Example constraints:
120 > age > 0
weight > 0

The component will include standard constraints to enforce the following:

1) a given numeric value is within a given range
2) a given numeric value is greater than, less than or equal to another value in the tuple
3) a given string value matches a given regular expression

The component will also allow custom tuple constraints to be added by users.

1.2.3 Relations

A relation is a data structure that is composed of two parts: a header and a body. The header consists of attribute names and their types and defines the type of the relation. The body consists of a set of tuples where each tuple contains a value of the specified type for each name in the header. Each tuple in a relation is unique.

Example:

A type ‘pet’ may have attributes ‘species’, ‘name’, ‘age’, ‘weight’. A relation of type ‘pet’ may contain these tuples:

(name:Fido age:3 species:Dog weight:30)
(name:Fluffy age:2 species:Cat weight 120)
(weight:300 age:5 species:horse name:Trigger)

It is not be legal to add any of the following tuples:

() -- empty tuple is illegal since all fields are null

(name:Rover) -- missing fields
(name: age:4 species:Dog weight:35) -- no value for ‘name’ attribute

Adding a duplicate tuple to the relation has no affect. It’s not an error, but number of tuples contained by the relation does not change.

1.2.4 Relation Constraints

The component will include an ability to define relation-level constraints. These constraints will be used to verify that added tuples don’t contradict each other.

Example:

Rel1 = ((id:101 value:Cat)
(id:101 value:Dog))

Constraint = unique(id)

‘Rel1’ by itself is a valid relation but contradicts itself. The unique constraint is added to help guarantee the integrity of the relation. The component will provide a custom constraint to enforce the uniqueness of one or more attributes. The component will also allow custom relation constraints to be added by users.

1.2.5 Relation Operations

The result of any relational operation is another relation. The resulting relation may be empty, but will never be ‘null’. This component will support the relational operations below.

1.2.5.1 Project Operation

The ‘project’ operation returns a relation containing all tuples in the original relation with the supplied columns removed. Duplicates are removed.

Example:

a projection of [‘weight’, ‘species’] over ‘pet’ will result in:

(name:Fido age:3)
(name:Fluffy age:2)
(age:5 name:Trigger)

1.2.5.2 Restrict Operation

The ‘restrict’ operation accepts a condition and returns a relation containing the tuples for which the condition is true.

Example:
A restriction of [age > 2] will result in:

(name:Fido age:3 species:Dog weight:30)
(weight:300 age:5 species:horse name:Trigger)

1.2.5.3 Union Operation

The ‘union’ operation combines two relations of the same type into a single relation. Duplicates are removed.

Example

Rel1 = ((a:1 b:2 c:3)
(a:2 b:3 c:4))

Rel2 = ((a:10 b:20 c:30)
(a:1 b:2 c:3))

Rel1 union Rel2 = ((a:1 b:2 c:3)
(a:2 b:3 c:4)
(a:10 b:20 c:30))

1.2.5.4 Difference Operation

The ‘difference’ operation accepts two relations of the same type and returns a relation that includes each tuple that is in the first relation, but not the second.

Example:

Rel1 = ((a:1 b:2 c:3)
(a:2 b:3 c:4))

Rel2 = ((a:10 b:20 c:30)
(a:1 b:2 c:3))

Rel1 difference Rel2 = ((a:2 b:3 c:4))

1.2.5.5 Natural Join Operation

The natural join operator combines two relations based on the values of common attributes. Tuples are extended to include all attributes in both relations. Duplicates are removed.

Example:

Rel1 = ((a:1 b:2 c:3)
(a:2 b:3 c:4))

Rel2 = ((a:1 x:0)
(a:2 x:1)
(a:3 x:2))

Rel1 join Rel2 = ((a:1 b:2 c:3 x:0)
(a:2 b:3 c:4 x:1))

1.2.5.6 Conditional Join

The conditional join operator is similar to the natural join operator except that the join criteria is specified by the component user. Duuplicates are removed

Example

Rel1 = ((a:1 b:2 c:3)
(a:2 b:1 c:4))

Rel2 = ((a:1 x:0)
(a:2 x:1)
(a:3 x:2))

Rel1 join Rel2 on (Rel1.a = Rel2.a and Rel1.b = Rel2.x)

= ((a:2 b:1 c:4 x:1))

1.2.5.7 Outer Join Operation

The outer join operation is a special type of join which assumes that when a value is missing in a on one side of the join, then the tuple should be included. For this component a left outer join will be used.

Example:

Rel1 = ((a:1 b:2 c:3)
(a:2 b:3 c:4)
(a:3 b:4 c:5))

Rel2 = ((a:1 x:2)
(a:2 x:3))

Rel1 left outer join Rel2 on (Rel1.a = Rel2.a and Rel1.b = Rel2.x)
= ((a:1 b:2 c:3 x:2)
(a:2 b:3 c:4 x:3)
(a:3 b:4 c:5 x:unknown))

While it's not generally possible for a relation to contain nulls, the outer join is an exception to that rule. The component will not use java's 'null' to represent missing information, but will define a special value that indicates missing information.

1.2.5.8 Intersection Operation

The 'intersection' operation is a special case of join which requires the two input relations to be of the same type. It will return each tuple that appears in both relations.

1.2.5.9 Product Operation

The 'product' operation distributes each element of each tuple in Rel1 over each element of each tuple in Rel2. Duplicates are removed.

Example:

Rel1 = ((a:1 b:2 c:3)
(a:2 b:3 c:4))

Rel2 = ((x:9 y:8 z:7)
(x:6 y:5 z:4))

Rel1 product Rel2 = ((a:1 b:2 c:3 x:9 y:8 z:7)
(a:1 b:2 c:3 x:6 y:5 z:4)
(a:2 b:3 c:4 x:9 y:8 z:7)
(a:2 b:3 c:4 x:6 y:5 z:4))

1.2.6 XML Representation

The component will include a method for creating an XML representation of a given relation. It will also include a method to re-create a relation from the XML.

1.2.7 ResultSet

The component will include a method for creating a relation from a fully populated java.sql.ResultSet object.

1.2.8 Performance

This component will be used with data sets of varying sizes. Algorithms for each operation will be chosen to provide the best average performance for both small (e.g. 10 tuples/relation) data sets and large (e.g. 100000 tuples/relation) data sets. Justification will be given for each selected algorithm.

1.3 Required Algorithms

As described in Logic Requirements.

1.4 Example of the Software Usage

This component could be used in a data abstraction layer to contain data returned from SQL queries and to perform operations on that data. In essence, this may be the returned type from a DAO layer.

1.5 Future Component Direction

None Defined.

2. Interface Requirements

2.1.1 Graphical User Interface Requirements

None Defined

2.1.2 External Interfaces

None Defined

2.1.3 Environment Requirements

- Development language: Java1.4
- Compile target: Java1.4

2.1.4 Package Structure

com.topcoder.persistence.relation

3. Software Requirements

3.1 Administration Requirements

3.1.1 What elements of the application need to be configurable?

The tuple types and constraints will be configurable. The example below is a suggested syntax, but it is not required that the component follow this. The designer will choose the most appropriate syntax.

Example:

```
<relation>
  <attribute>
    <attribute-name>name</attribute-name>
    <attribute-type>java.lang.String</attribute-type>
  </attribute>
  <attribute>
    <attribute-name>weight</attribute-name>
    <attribute-type>java.lang.Integer</attribute-type>
  </attribute>
  <attribute>
    <attribute-name>species</attribute-name>
    <attribute-type>com.petsore.common.Species</attribute-type>
  </attribute>
  <attribute>
    <attribute-name>age</attribute-name>
    <attribute-type>java.lang.Integer</attribute-type>
    <tuple-constraint>
      <constraint-class>com.petstore.constraints.WeightConstraint</constraint-class>
    </tuple-constraint>
  </attribute>
  <relation-constraint>
    <constraint-class>com.topocoder.persistence.constraints.UniqueConstraint</constraint-class>
    <constraint-param>name</constraint-param>
  </relation-constraint>
</relation>
```

3.2 Technical Constraints

3.2.1 Are there particular frameworks or standards that are required?

None defined.

3.2.2 TopCoder Software Component Dependencies:

**Please review the TopCoder Software component catalog for existing components that can be used in the design.

3.2.3 Third Party Component, Library, or Product Dependencies:

None Defined

3.2.4 QA Environment:

- Solaris 7
- RedHat Linux 7.1
- Windows 2000
- Windows 2003

3.3 Design Constraints

The component design and development solutions must adhere to the guidelines as outlined in the TopCoder Software Component Guidelines. Modifications to these guidelines for this component should be detailed below.

3.4 Required Documentation

3.4.1 Design Documentation

- Use-Case Diagram
- Class Diagram
- Sequence Diagram
- Component Specification
- XML Schema for Relation XML representation
- XML Schema for Relation and Tuple definition (configuration)

3.4.2 Help / User Documentation

- Design documents must clearly define intended component usage in the ‘Documentation’ tab of Poseidon.