OPTIMIZATION OF SPARQL BY USING coreSPARQL

Jinghua Groppe, Sven Groppe and Jan Kolbaum

IFIS, University of Luebeck, Ratzeburger Allee 160, D-23538 Luebeck, Germany

Keywords: Semantic Web, RDF, SPARQL, coreSPARQL, Query Optimization, Query Rewriting.

Abstract: SPARQL is becoming an important query language for RDF data. Query optimization to speed up query processing has been an important research topic for all query languages. In order to optimize SPARQL queries, we suggest a core fragment of the SPARQL language, which we call the coreSPARQL language. coreSPARQL has the same expressive power as SPARQL, but eliminates redundant language constructs of SPARQL. SPARQL engines and optimization approaches will benefit from using coreSPARQL, because fewer cases need to be considered when processing coreSPARQL queries and the coreSPARQL syntax is machine-friendly. In this paper, we present an approach to automatically transforming SPARQL to coreSPARQL, and develop a set of rewriting rules to optimize coreSPARQL queries. Our experimental results show that our optimization of SPARQL speeds up RDF querying.

1 INTRODUCTION

The Semantic Web uses the Resource Description Framework (RDF) (Beckett, 2004) as its data format to describe information in the web. RDF provides a model and syntax for describing data, but it does not provide querying functionalities. A number of RDF querying languages have been developed, e.g. RQL, N3, Versa, SeRQL, Triple and RDQL. When (Haase et al., 2004.) compares these six languages, SPARQL (Prud’hommeaux and Seaborne, 2007) has not emerged. SPARQL was first proposed on 12th October 2004 and became an official W3C Recommendation on 15th January 2008. Many RDF stores support or plan to support SPARQL, e.g. Jena (Wilkinson et al., 2003) and Sesame (Broekstra et al., 2002). SPARQL becomes increasingly important as an RDF query language.

The optimization of queries has been an active research topic for improving the performance of query processing. An important optimization technique is rewriting of queries. While query rewriting has been extensively studied in the relational databases and XML areas, there is no complete and thorough work on rewriting of SPARQL queries. Therefore, we focus on the rewriting and simplification of SPARQL queries. In this paper we develop a core fragment of the SPARQL language to simplify SPARQL, which we name coreSPARQL, and a set of rules to optimize coreSPARQL queries.

SPARQL supports a large number of different language constructs, which brings flexibility of expressiveness, but also redundancy of expressions. For example, the three expressions of SPARQL in Figure 1 have the same semantics. Redundant expressive power increases the difficulties of query processing. It is also obvious that the syntax for Expression 1 is user-friendly, but Expression 3 is more easily to be interpreted by a machine.

<table>
<thead>
<tr>
<th>Expression 1</th>
<th>Expression 2</th>
<th>Expression 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Figure 1: Three SPARQL expressions with same semantics." /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In order to reduce the number of cases, which must be considered when processing SPARQL queries, and in order to make SPARQL queries more machine-processable, we suggest the coreSPARQL language, which is a core fragment of the SPARQL language. coreSPARQL possesses the same expressive power as SPARQL, but does not contain redundant

This work is funded by the German Research Foundation (DFG) project GR 3435/1-1 LUPOSDATE.
language constructs of SPARQL and only allows machine friendly syntax. We develop an approach, which automatically transforms SPARQL queries to coreSPARQL queries.

SPARQL queries written by users or generated by applications are often un-optimized, and thus sub-optimal. Sub-optimal queries impact query processing performance. Based on coreSPARQL, we develop a set of simplification rules to rewrite coreSPARQL queries, and transform a sub-optimal query into an optimal query by eliminating redundant parts and optimizing sub-expressions. Our performance study shows that after our optimization, SPARQL can be processed more efficiently, and the transformation of SPARQL to coreSPARQL has a low overhead. Due to the limitation of space, we do not present our experiment results in this paper.

**Related Work.** (Pérez et al., 2006) suggests several rules for rewriting AND, UNION and OPTIONAL expressions in SPARQL queries. The purpose of the rewriting is constructing a critical fragment of UNION-free graph pattern expressions for the study of evaluation complexity. (Bernstein et al., 2007), (Groppe et al., 2007a), (Broekstra et al., 2002) and (Groppe et al., 2009) reorder triple patterns in order to reduce the size of intermediate results. (Groppe et al., 2007a) pushes a filter expression upward if all the variables in the filter expression has already been bound. (Bernstein et al., 2007) reorders triple patterns according to their selectivity, which is estimated based on schemas. (Broekstra et al., 2002) and (Groppe et al., 2009) both observe that the number of variables might impact the sizes of the intermediate resultant data. (Broekstra et al., 2002) reorders the triple patterns according to the number of variables, while (Groppe et al., 2009) considers the number of the new variables, which have not been bound so far, because the occurred variables are bound with the result of previous triple patterns.

An amount of work contributes to the rewriting of relational algebra, and develops a number of equivalency rules (Araujo et al., 2006) (Chaudhuri, 1998) (Ioannidis, 1996) (Jarke and Koch, 1984). Some of our and other equivalency rules for rewriting SPARQL queries are adapted from the equivalency rules for relational algebra, e.g. the rules for comparison operators.

Several contributions are dedicated to the transformation of SPARQL queries to SQL queries, and the storage of RDF data in relational databases, and thus use proven database technologies, e.g. (Chong et al., 2005), (Chebotko et al., 2007) and (Cyganiak, 2005).

(Groppe et al., 2007b), (Weiss et al., 2008) and (Groppe et al., 2009) suggest different indices for fast data access. (Groppe et al., 2009) develops a new approach to compute join of triple patterns by dynamically restricting triple patterns.

## 2 RDF AND SPARQL

Figure 2 presents an example of RDF data and of a SPARQL query.

RDF data is a set of triples of the form Subject Predicate Object, which are RDF terms, e.g. IRIs, literals or blank nodes. Figure 2 provides an example of RDF data with 3 triples. SPARQL selects RDF data based on graph pattern matching, where the core component of SPARQL graph patterns is a set of triple patterns $s p o$. The triple pattern matches the result of each triple pattern.

The SPARQL query `Book.sparql` in Figure 2 consists of the `SELECT` clause and the `WHERE` clause. The `SELECT` clause identifies the variables to appear in the query results, and the `WHERE` clause contains two triple patterns, which identify the constraints on RDF data. The triple pattern `?x ex:title ?y` matches the first two triples of `Book.rdf`, such that its result is `{?x=ex:book1, ?y="XML">, ?x=ex:book2, `?y="Index">}). The triple pattern `?x ex:pages ?z` matches the last triple of `Book.rdf`, such that the result is `{?x=ex:book2, `?z=90}>}. The two triple patterns impose a join over the common variable `?x`, such that the result of the two triple patterns is `{?x=ex:book2, `?y="Index">, `?z=90}>}. The final query result is `{?y="Index">, `?z=90}>}.

SPARQL provides rich capabilities to select and filter data, and we refer the interested reader to...
(Prud’hommeaux and Seaborne, 2007) for a complete description of SPARQL.

3 CORESPARQL

SPARQL allows redundant language constructs and supports abbreviated syntax. The redundancy brings the flexibility of expressiveness and abbreviations bring the simplification of expressions, but they do not increase the expressive power of the language. That a SPARQL query can be expressed in different forms increases the number of cases to be processed; the abbreviated syntaxes are not machine-friendly. In order to make SPARQL queries more machine-processable, and to reduce the number of cases, which must be considered when processing SPARQL queries, we abstract a subset from the SPARQL language, and name the subset coreSPARQL.

3.1 Defining coreSPARQL

In Definition 1, we describe coreSPARQL in terms of the common and different properties with SPARQL. Figure 3 demonstrates several SPARQL and corresponding coreSPARQL components.

<table>
<thead>
<tr>
<th>component</th>
<th>SPARQL</th>
<th>coreSPARQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>triple pattern</td>
<td>s1 p1 o1; p2 ?x.</td>
<td>s1 p1 o1; s2 p2 ?x.</td>
</tr>
<tr>
<td>blank node</td>
<td>[ p o].</td>
<td>_b p o.</td>
</tr>
<tr>
<td>group graph pattern</td>
<td>{ s1 p1 o1 } s2 p2 o2.</td>
<td>{ s1 p1 o1 } s2 p2 o2.</td>
</tr>
<tr>
<td>&amp;&amp; operator</td>
<td>Filter(A &amp;&amp; B).</td>
<td>Filter(A).</td>
</tr>
</tbody>
</table>

Figure 3: SPARQL and corresponding coreSPARQL components.

Definition 1 (coreSPARQL). CoreSPARQL is a core fragment of SPARQL. A coreSPARQL query is also a SPARQL query. CoreSPARQL has the same expressive power as SPARQL, but allows only machine-friendly syntax, and eliminates many redundant language constructs. Especially in coreSPARQL,

- all triple patterns are only in the form: s p o;
- a group graph pattern cannot directly nest another group graph pattern;
- variable names start only with ?;
- blank nodes [ ] are not allowed;
- RDF collections of the form (…) are not allowed;
- neither prefixed IRIs nor IRIs, which are relative to a BASE-declaration, are allowed.
- the keyword a is not allowed;
- the && operator is not allowed.

3.2 Transforming SPARQL to coreSPARQL

SPARQL provides user-friendly syntax to write RDF queries, and coreSPARQL queries are easy to program. Therefore, the next task for us is to find a way to automatically transform SPARQL queries to coreSPARQL queries. We develop a set of transformation rules, such that a SPARQL query can be transformed into a coreSPARQL query by recursive application of these rules, i.e. if the expression of a left-hand side of a rule occurs in a SPARQL query, it is replaced with the right-hand side of the rule.

We use the following notation to describe these rules: we write s (s1, s2, …), p (p1, p2, …), o (o1, o2, …) for the subject, predicate, and object of a triple pattern, os (os1, os2, …) for a list of objects, e.g. os = o1, o2, …, om, where m ≥ 1, and pos (pos1, pos2, …) for predicate-object-lists, e.g., pos=p1 os1; p2 os2; …; pm osm, where m ≥ 1. A blank node [ ] is replaced by a blank node label, e.g. _b, where b must be not used elsewhere in the query. Note that some patterns in the following rules may be not supported by SPARQL. Such patterns are intermediate results of the transformation, and will be translated to standard language constructs after the transformation.

- Rule 1: eliminating Object-Lists:
  1.1 s1 p1 o1, os. => s1 p1 o1. s1 p1 os.
- Rule 2: eliminating Predicate-Object-Lists:
  2.1 s1 p1 os1; pos. => s1 p1 os1. s1 pos.
- Rule 3: eliminating blank nodes []:
  3.1 [ ] => _b
  3.2 [ pos ]. => _b pos.
  3.3 [ pos ] p1 os1. => _b pos. _b p1 os1.
  3.4 s1 p1 [ pos ]. => s1 p1 _b. _b pos.
- Rule 4: eliminating RDF collections ( ), where e (e1, e2, …) is an element of the collection, i.e. a variable, a literal, a blank node, or a collection. Here, we introduce a variant of the collection, e.g. (e)_b. to restrict that the blank node, which is allocated for the collection (e), must be _b.
  4.1 (e) pos. => _b rdf:first e. _b rdf:rest rdf:nil.
  4.2 (e). => _b rdf:first e. _b rdf:rest rdf:nil.
  4.3 (e1 e2 e3…). => _b rdf:first e1. _b rdf:rest (e2 e3…).
4.4 \( s p (e1 e2 \ldots) \Rightarrow s p \_b. (e1 e2 \ldots)_s-b \).
4.5 \( (e1 e2 \ldots)_s-b \Rightarrow \_b rdffirst e1. \)
4.6 \( (e)_s-b \Rightarrow \_b rdffirst e. \_b rdfrest (e2 \ldots) \).
4.6 () \Rightarrow rdfnil.

- Rule 5: eliminate the keyword \( a \):
  5.1 \( a \Rightarrow rdf:type \).
- Rule 6: eliminate directly nested group graph patterns
  6.1 \{ \{ A \} \ldots \} \Rightarrow \{ A \ldots \},
where \( A \) is not a part of a \( OPTIONAL \), or a \( UNION \), or a \( GRAPH \) graph pattern; \( A \) does not consist of only Filter expressions either.

  6.2 \{ \{ Filter(e) \} \ldots \} \Rightarrow \{ \{ Filter(true) \} \ldots \},
if the result of the static analysis of \( e \) is true.

  6.3 \{ \{ Filter(false) \} \ldots \},
if the result of the static analysis of \( e \) is false or a type error.

For example, the expression \( 10>1 \) is statically analyzed to true, and thus \( \text{Filter}(10>1)) = \text{Filter}(true) \). In the group graph pattern \( \text{Filter}(\text{bound}(?x)) \), the variable \( x \) will never be bound. Therefore, the static analysis of \( \text{bound}(?x) \) detects a type error, and thus \( \text{Filter}(\text{bound}(?x)) = \text{Filter}(false) \). For the details on the static analysis and type errors, see Section 11.2 Filter Evaluation in the SPARQL specification (Prud’hommeaux and Seaborne, 2007).

- Rule 7: eliminating \&\& operator, where \( A, B \) and \( C \) are conditional expressions.
  7.2 \text{Filter}(A || C) \Rightarrow \text{Filter}(A). || \text{Filter}(C).
  7.3 \text{Filter}(A || B) \Rightarrow \text{Filter}(A) \& \& \text{Filter}(B).
  7.4 \text{Filter}(A && B) \Rightarrow \text{Filter}(A).

- Rule 8: eliminating prefixes and \text{BASE} declarations.
  8.1 p:a \Rightarrow <prefix(p) a>.
where \text{prefix}(p) is a function to resolve the prefixed IRI \( p:a \) according to defined \text{PREFIX} and \text{BASE} declarations. The \text{PREFIX} and \text{BASE} declarations are deleted in the coreSPARQL query.

**Example 1.** Using this example, we demonstrate how to transform a SPARQL expression \( t1 = \{ [ p o1] (2) \} \) into the corresponding coreSPARQL expression by recursively applying the rules above.

1. Applying Rule 4.3 on \( t1: t1 \Rightarrow t2. t3. \)
   \( b1 \) rdffirst 1. \( b2 \) rdfrest \( [ p o1] (2) \). \( (t2) \)
2. Applying Rule 4.4 on \( t3: t3 \Rightarrow t4. t5. \)
   \( b1 \) rdfrest \( b2 \). \( (t3) \)
3. Applying Rule 4.5 on \( t5: t5 \Rightarrow t6. t7. \)
   \( b2 \) rdffirst \( [ p o1] \). \( (t4) \)
   \( b2 \) rdfrest \( (2) \). \( (t5) \)
4. Applying Rule 3.4 on \( t6: t6 \Rightarrow t8. t9. \)
   \( b2 \) rdffirst \( b3 \). \( (t6) \)
   \( b3 \) p o1. \( (t7) \)
5. Applying Rule 4.4 on \( t7: t7 \Rightarrow t10. t11. \)
   \( b2 \) rdfrest \( b4 \). \( (t8) \)
6. Applying Rule 4.6 on \( t8: t8 \Rightarrow t10. t11. \)
   \( b4 \) rdffirst \( b5 \). \( (t9) \)
7. Applying Rule 4.6 on \( t9: t9 \Rightarrow t12. t13. \)
   \( b4 \) rdffirst \( b5 \). \( (t10) \)
8. Applying Rule 4.6 on \( t10: t10 \Rightarrow t12. t13. \)
   \( b5 \) rdffirst \( 2 \). \( (t11) \)
9. Applying Rule 4.6 on \( t11: t11 \Rightarrow t12. t13. \)
   \( b5 \) rdfrest \( (2) \). \( (t12) \)
10. Applying Rule 4.6 on \( t12: t12 \Rightarrow t14. t15. \)
    \( b5 \) rdfrest \( rdf:nil. \) \( (t13) \)
11. Applying Rule 4.6 on \( t13: t13 \Rightarrow t14. t15. \)
    \( b5 \) rdfrest \( rdf:nil. \) \( (t14) \)
12. Applying Rule 4.6 on \( t14: t14 \Rightarrow t16. t17. \)
    \( b5 \) rdffirst \( 2 \). \( (t15) \)
13. Applying Rule 4.6 on \( t15: t15 \Rightarrow t16. t17. \)
    \( b5 \) rdfrest \( rdf:nil. \) \( (t16) \)
14. Applying Rule 4.6 on \( t16: t16 \Rightarrow t18. t19. \)
    \( b5 \) rdfrest \( rdf:nil. \) \( (t17) \)

The result of transformation consists of the triple patterns \( t2, t4, t8, t9, t10, t13, t14, t16 \) and \( t17 \).

Note that there are further redundancies, which we allow in coreSPARQL, as they can be processed in a machine-friendly way. For example, the wildcard \( * \) in \text{SELECT} \{ \text{DISTINCT} \ | \text{REDUCED} \} \ * \) and \text{DESCRIBE} \ *, can be replaced by the concrete variables in triple patterns. \text{REDUCED} keyword can be replaced by \text{DISTINCT} or can be deleted. Any operations on constants can be replaced by the result of their applications.

## 4 REWRITING CORESPARQL QUERIES

While the coreSPARQL query does not contain redundant language constructs, a coreSPARQL query may not be optimal, e.g. containing redundant constraints. For example, if we have two constraints \text{bound}(\text{x}) \&\& \text{Filter}(\text{x}==1) \). In a SPARQL query, then the constraint \text{bound}(\text{x}) \) is redundant: \text{bound}(\text{x}) requires that the variable \( x \) is bound with a value, and \text{Filter}(\text{x}==1) \) implies that \( x \) is bound to the value 1. The reason for sub-optimal SPARQL queries is that queries written by users or generated in applications are often non-optimized. The sub-optimal queries impact significantly query processing performance.

As well as being sub-optimal, queries are also possibly unsatisfiable. A query is unsatisfiable if the query selects the empty result for any RDF data. Therefore, if we can detect that a query is unsatisfiable, we can avoid the submission and evaluation of the unsatisfiable query, and thus save
processing time and query cost. A query is unsatisfiable, if it contains conflicting constraints. For example, two constraints IsIRI(\( ?x \)) and Filter(\( ?x = \text{"http://example.com"} \)) contradict each other: IsIRI(\( ?x \)) requires that \( ?x \) is an IRI, but Filter(\( ?x = \text{"http://example.com"} \)) requires that \( ?x \) is a string.

In order to optimize queries and improve the evaluation performance, we develop a set of equivalency rules to detect conflicting and redundant constraints. By recursive application of these rules, a coreSPARQL query can be optimized to a more simple expression, or even to an empty expression, i.e. the query is unsatisfiable.

We use the rewriting rules in Table 1 to simplify coreSPARQL queries, where \( C \), \( C1, C2, \ldots \) represents a literal, \( G \) a graph pattern or the query pattern, i.e. the outer-most graph pattern, and \( E \), \( E1, E2, \ldots \) an expression. Additionally, we introduce a new graph pattern: void graph pattern, denoted by \( \bot \). Contrary to the empty group pattern \( \{ \} \) in SPARQL, which matches any RDF graph, a void graph pattern \( \bot \) does not match any RDF graph. If a SPARQL query is simplified to the void graph pattern, the query is unsatisfiable. Note that \( \bot \) is an intermediate result during simplification, and any satisfiable SPARQL expressions will not contain \( \bot \) after optimization.

### 5 CONCLUSIONS

We suggest the coreSPARQL language, which is a core fragment of SPARQL, but has the same expressiveness as SPARQL. Optimization approaches, SPARQL engines and all applications, which process SPARQL queries, benefit from coreSPARQL, because coreSPARQL possesses machine-friendly syntax and thus is easy to program, contains less language constructs and thus reduces the number of cases to be considered.

We develop a set of transformation rules to translate SPARQL queries to coreSPARQL queries, and a set of rewriting rules to further optimize coreSPARQL queries. We develop a prototype of our approach, which shows that our optimization speeds up SPARQL query processing.
Table 1: Rewriting rules for optimizing coreSPARQL queries (cont.).

<table>
<thead>
<tr>
<th>Comparison operators:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• FILTER(?x op1 C1). FILTER(?x op2 C2). =&gt;</td>
</tr>
<tr>
<td>• FILTER(?x op1 C1), if ((op1=op2 ∧ (C1 op1 C2) ∧ op1 e (c1, c2, ..., cn)) ∨ ((C1 op1 C2) ∧ C1#C2 ∧ (op1, op2 e (c1, c2, ..., cn)) ∨ (C1=C2 ∧ ((op1=op2 e (c1, c2, ..., cn)) ∨ (op1=op2 e (c1, c2, ..., cn))))</td>
</tr>
<tr>
<td>• FILTER(?x op2 C2), if ((op1=op2 ∧ (C2 op1 C1) ∧ op1 e (c1, c2, ..., cn)) ∨ ((C2 op1 C1) ∧ C1#C2 ∧ (op1, op2 e (c1, c2, ..., cn)) ∨ (C2=C1 ∧ ((op1=op2 e (c1, c2, ..., cn)) ∨ (op2=op1 e (c1, c2, ..., cn))))</td>
</tr>
<tr>
<td>• FILTER(false), if ((C1&gt;C2 ∧ op1 e (c1, c2, ..., cn)) ∨ (C1=C2 ∧ op1 e (c1, c2, ..., cn)) ∨ (C2 op1 C1) ∧ op1 op2 e (c1, c2, ..., cn))</td>
</tr>
<tr>
<td>• FILTER(?x op1 C1), FILTER(?x op2 C2), otherwise.</td>
</tr>
</tbody>
</table>

Operators || .! and ¬
---|---
E || true => true • false || false => false • E || E => E • !A1 op A2) => A1 - (op) A2
?x op1 C1 || ?x op2 C2 =>
?x op1 C1, if ((op1=op2 ∧ (C2 op1 C1) ∧ op1 e (c1, c2, ..., cn)) ∨ ((C2 op1 C1) ∧ C1#C2 ∧ (op1, op2 e (c1, c2, ..., cn)) ∨ (C2=C1 ∧ ((op1=op2 e (c1, c2, ..., cn)) ∨ (op1=op2 e (c1, c2, ..., cn)))) |
?x op2 C2, if ((op1=op2 ∧ (C1 op2 C2) ∧ op1 e (c1, c2, ..., cn)) ∨ ((C1 op2 C2) ∧ C1#C2 ∧ (op1, op2 e (c1, c2, ..., cn)) ∨ (C1=C2 ∧ ((op1=op2 e (c1, c2, ..., cn)) ∨ (op1=op2 e (c1, c2, ..., cn)))) |
| Bound(?x), if (op1=-(op2) ∧ C1=C2), |
?x op1 C1 || ?x op2 C2, otherwise.
• ¬(e) => ≠ • ¬(=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>=) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => ≠ • ¬(>) => ≠ • ¬(<) => ≠ • ¬(<=) => !=