Title: Adding Structured Text to SQL/MM Part 2: Full Text

Author: Ian Davis

Project: SQL/MM Part 2: Full Text

Description: Change Proposal

Status: Adopted by SQL/MM for late progression.

References:

1) Working Draft SQL3


4) Ian Davis, Saman Farazdaghi, Frank Tompa and Paul Cotton: Integrating SQL and SGML YOW-028

5) David Beech: Comments on Integrating SQL and SGML. YOW-034

6) ISO 8879: Standard Generalized Markup Language (SGML) 1986


Introduction

The current draft of the SQL/MM Part 2: Full Text specification provides for the construction of a Full-Text Abstract Data Type. Instances of this type can be tested to determine if they contain a specific textual pattern within them, but they cannot be internally marked or decomposed. They also cannot be assigned any structure beyond that suggested by the keywords character, word, sentence, and paragraph.

This proposal defines a Structured Full-Text ADT that is based on the Full-Text ADT. No changes are required to the underlying Full-Text ADT in order to support this proposal, and future changes to Full-Text will have little or no impact on the definition of the proposed Structured Full-Text ADT.

Structured Full-Text encapsulates instances of text having some hierarchical structure defined by an implicit or explicit grammar. When encapsulated within the Structured Full-Text ADT, such instances of text can be selected on the basis of their grammar, Full-Text content, and/or hierarchical structure. Structural nodes (subtexts) within instances of Structured Full-Text can be marked and these marks later used: to count subtexts that matched patterns, to qualify searches, to navigate within structural information, and to assist in fragmenting structured text into subtexts. Marks may also be used to preserve knowledge of the context from which subtexts were extracted, and to later reconstruct instances of structured text in which only specific sets of subtexts (identified using arbitrary SQL3 operations) remain marked.

The grammar associated with an instance of structured text provides the schema for the internal structure of that text. Methods are proposed for extracting from this grammar the valid node names associated with the structure of an instance of structured text, the relationships between node names in structures and substructures, and specific language dependent information contained within an explicit grammar. These grammars may also be used to validate updates performed against instances of structured text, etc.

Although one of the principal goals of this proposal is to provide support for SGML, this proposal provides the same support for instances of structured text that adhere to other standards. Nothing is assumed about how structured text or its associated grammar is externally represented or encoded.

Background

Reference [3] was presented to the SQL/MM Rapporteur Group Meeting in Ottawa July 1995 as a discussion paper [4]. The methods described within that discussion paper for integrating SQL and Structured Text have been implemented and explored within a prototype SQL2 federated database system developed by the Centre for the New Oxford English Dictionary and Text Research at the University of Waterloo. Demonstrations of this software can be conducted on request. The SQL language extensions encapsulated within this prototype are formulated in this document as SQL3 Abstract Data Types.

This paper was reviewed as an expert contribution by the Canadian Advisory Committee in Victoria B.C on November 17th, 1995. It was accepted as an approved Canadian position at the December 20, 1995 CAC/WG3 meeting held in Toronto, Ontario, and the proposed changes to SQL/MM approved for late progression January 18th, 1996 by the SQL/MM Rapporteur Group meeting in London, England.

The research leading to the creation of this change proposal forms part of our contribution to the Canadian Strategic Software Consortium (CSSC). Open Text Corporation, Fulcrum Technologies Inc., Dataware Technologies Inc., Grafnetix Systems Inc., Public Sector Systems Ltd., InContext Corporation, SoftQuad Inc., and the University of Waterloo are the members of this consortium.
Outstanding Issues

A  Coding Issues

A1) Further work (and guidance) is needed to ensure that the Abstract Data Type definitions presented in this document conform precisely with evolving SQL3 and SQL/MM standards and syntax.

A2) It is assumed that some method exists for generating unique node identifiers, and that such unique identifiers can be represented by integer values.

B. Full-Text Opportunities

B1) Structured text may contain embedded figures, footnotes, references, markup, entity references etc. Some Full-Text searches will depend on the assumption that all text within an instance of structured text is visible. Other types of Full-Text search will not be possible unless text within structured text can be treated as absent or replaced with alternative Full-Text fragments when performing Full-Text queries against fragments of structured text.

B2) The current definition of Full-Text provides no ability to mark fragments of Full-Text containing specific information. This has already been raised as Issue Number 2-009 of the current SQL/MM draft. It is desirable that fragments of Full-Text can be marked, since this would make Full-Text operations more consistent with the structured text operations described in this proposal (which do allow structured text nodes to be marked), and might potentially simplify the task of integrating structured and content based text searching.

B3) This proposal provides mechanisms for extending the contains language described in Section 5 of SQL/MM Part 2: Full Text so that structural information present in text can be used to identify specific fragments of Full-Text that are to be tested to see if they contain specific information. The contains language used to perform such tests should be extended so that it is capable of defining proximity and potentially other operations not just in terms of implementation dependent units such as characters, words, sentences, and paragraphs, but in terms of identifiable units of structured subtext occurring within text. Such extensions would only apply when operating on instances of structured text. This would help address Issue Number 2-013 of the current SQL/MM draft, and would greatly increase the potential to describe, locate, mark and extract specific instances of both full and structured text.

B4) It is not clear if it should be possible to convert an instance of Structured Text directly into an instance of Full-Text.

C. Encoding issues

C1) This proposal deliberately avoids making any suggestion as to how to encode structured text. The choice of how structured text is to be conceptually represented within a structured text tree may have a significant impact on the power of the pattern matching language proposed.

C2) It should not be assumed that there is necessarily a one to one correspondence between nodes in a structured text tree, and marked up structures in the input text.

C3) If Structured Text contains ‘X’ then the pattern %‘X’ will match every node in the structured text that contains ‘X’ up to an including the root of the structured text tree. If it is useful to be able to identify the smallest subtext(s) containing ‘X’, the encoding should provide an easy mechanism for identifying leaves in the structured text tree. One such mechanism would be to assign leaves in the structured text tree a reserved node label.
C4) It is not clear if node labels within structured text should be case sensitive or case insensitive. If it is established that node labels are always case insensitive, then the LIKE comparisons performed on these node labels should also be case insensitive.

C5) It is desirable that standards be developed for how texts and grammars might be encoded within structured text trees. The DSSSL standard [9] provides details of how SGML documents can be encoded within tree structures.

D. Structured Text Pattern Matching

D1) The style of the proposed pattern matching language departs from the style of the Full-Text pattern matching language. This may or may not be of concern.

D2) Within node label patterns, the default LIKE escaping mechanism is assumed. Within Full-Text pattern matching, the default CONTAINS escaping mechanism is assumed. If other escaping mechanisms are desired, this can be handled by applying string conversion functions to input patterns.

E. Grammars

E1) This proposal assumes that a single external grammar is associated with structured text which imposes a hierarchical structure on this text. Not all structure present within instances of structured text can necessarily be described using a single hierarchical grammar.

E2) The StructuredTextGrammar ADT provides access to the values of (and the relationships between) node labels in a Structured Text Tree. This information may be based either on the current node labels or the node labels allowed within this Structured Text Tree.

E3) Extracted fragments of structured text may have an undefined external grammar. SGML subtexts typically do not have a grammar associated with them.

E4) Operations which test that an instance of Structured Text conforms with a given internal or external grammar need to be defined, so that columns containing Structured Text can be suitably constrained, using the constraint mechanisms supported within SQL3.

E5) It is not clear if permission to create an instance of StructuredTextGrammar directly from a string should be granted to the public. If not, then such StructuredTextGrammar instances can only be derived indirectly from valid instances of StructuredText.

E6) The GrammarToText functionality should perhaps be removed from the StructuredTextGrammar ADT described in section 7.2. Instances of StructuredText describing external grammars (DTD’s, etc.) can be directly created by invoking the StructuredText ADT constructor with an external grammar and a suitably identified parser. It would then be the responsibility of applications to create and preserve (when appropriate) the relationships between structured texts values and their external grammars. This would simplify and help resolve some of the issues, E1, E2, E3 and E4, above.

F. SQL Issues

F1) This proposal does not yet address how structured text, or an associated internal or external grammar might be updated.

F2) Several public functions described within this proposal return relations having multiple rows and columns. It is assumed for the present that the undefined function UNNESTED_TABLE will perform conversion from sets of lists to corresponding sets of rows having suitable column names when necessary, and then convert sets of rows into tables having these rows.
F3) The set aggregation function *AggregateMarks* is implemented as proposed in LHR DBL-076. This proposal uses procedures to implement user-defined set aggregation functions. Since each of these procedures always has exactly one output they might have been better expressed as functions.

F4) Some of the functionality within this proposal depends on recursion. It is assumed that SQL3 allows self referencing within ADT value and function definitions. This may not currently be the case.

F5) There is an assumption that keys used in the SQL ‘order by’ clause can be computed by functions. This may not be valid within SQL3.

**Proposal**


A) Change Section 1 Scope by adding:

f) defines the structured full-text abstract data type,

B) Insert after Section 6 the new Section 7 and 8 provided below.

C) Renumber all later sections (currently Section 7) appropriately.

D) Adjust the SQLSTATE values as deemed appropriate, and register their usage.

E) Attach suitably chosen prefixes if appropriate to functions, domains, etc.

F) Update Table of Contents, Indices, etc.

**Electronic availability**

This document is available as a Word 6 document, and/or as ASCII text. To obtain this document electronically please contact ijdavis@solo.uwaterloo.ca. Earlier discussion papers are available via http://bluebox.uwaterloo.ca/OED/trdbms.html.
7 StructuredText Abstract Data Type Family

The types in this family provide for the construction of structured text, their associated internal grammars and the text pattern rules used to operate on structured text.

7.1 StructuredText Abstract Data Type

Function

This type provides for the construction of structured text, for testing whether structured text contains specified patterns, for marking subtexts, for decomposing structured text into substructures, for aggregating structured text marks and for converting structured text into character strings.

Definition

CREATE VALUE TYPE StructuredText (  
  CAST  
    (StructuredText AS CHARACTER VARYING(max_text_length)  
      WITH TextToVarying);  
  CAST  
    (StructuredText AS StructuredTextGrammar  
      WITH TextToGrammar);  

PRIVATE Grammar  StructuredTextGrammar,  
  Tree StructuredTextTree,  
  Marks SetOfNodeIds,

PUBLIC FUNCTION  
  StructuredText(string CHARACTER VARYING(max_text_length),  
    parser CHARACTER VARYING(max_parser_length))  
  RETURNS StructuredText  
BEGIN  
  DECLARE new StructuredText;  
  IF string IS NULL OR parser IS NULL THEN  
    RETURN NULL;  
  END IF;  
  SET new          = StructuredText();  
  SET new..Grammar = StructuredTextGrammar(string, parser);  
  SET new..Tree    = ParseText(string, parser);  
  SET new..Marks   = SetOfNodeIds(); -- !! An empty set of marks  
  RETURN new;  
END,

GRANT EXECUTE ON FUNCTION StructuredText TO PUBLIC

PUBLIC FUNCTION  
  TextToString(text StructuredText)  
  RETURNS CHARACTER VARYING(max_text_length)  
BEGIN  
  RETURN TextToString(text, 'DEFAULT');  
END,

PUBLIC FUNCTION
BEGIN
  DECLARE new CHARACTER VARYING(max_text_length);

  IF text IS NULL OR method IS NULL THEN
    RETURN NULL;
  END IF;

  CASE lowercase(method)
    WHEN 'root' THEN SET new = text..Tree..RootLabel;
    WHEN 'clear' THEN SET new = text..Tree..RawText;
      SET new = text..Grammar..ClearText(new);
    WHEN 'default' THEN SET new = ...
    WHEN ...
      -- ! See Description
    END CASE;

  RETURN new;
END,

PUBLIC FUNCTION TextToGrammar(text StructuredText)
  RETURNS TextGrammar
BEGIN
  IF text IS NULL THEN
    RETURN NULL;
  ELSE
    RETURN text..Grammar;
  END IF;
END,

PUBLIC FUNCTION CountMarks(text StructuredText)
  RETURNS INTEGER
BEGIN
  IF text IS NULL THEN
    RETURN NULL;
  END IF;

  RETURN text..Marks..Cardinality;
END,

PUBLIC FUNCTION KeepMarks(text StructuredText, start INTEGER, count INTEGER)
  RETURNS StructuredText
BEGIN
  DECLARE nodeid   NodeID;
  DECLARE newmarks SetOfNodeIds;
  DECLARE new      StructuredText;

  IF text IS NULL OR start IS NULL OR count IS NULL THEN
    RETURN NULL;
  END IF;

  RETURN new;
END,
END IF;

SET newmarks = SetOfNodeIds();

FOR nodeid AS SELECT m FROM TABLE(text..Tree..Ids..Members) T(m)
ORDER BY text..Tree..Preorder(m)
DO -- preorder traversal --
  IF text..Marks..HasMember(nodeid) THEN
    IF start < 1 and count > 0 THEN
      SET newmarks = newmarks..Union(node..NodeId);
      SET count    = count - 1;
    END IF;
  END IF;
END FOR;

SET new = text;
SET new..Marks = newmarks;

RETURN new;
END,

GRANT EXECUTE ON FUNCTION KeepMarks TO PUBLIC

PUBLIC FUNCTION
TextMatch(text StructuredText, pattern StructuredTextPattern)
RETURNS Boolean
BEGIN
  DECLARE search  StructuredTextSearch;
  DECLARE new     Boolean;

  IF text IS NULL OR pattern IS NULL THEN
    RETURN NULL;
  END IF;

  SET search = StructuredTextSearch(pattern);
  SET new    = search..Matches(text);

  RETURN new;
END,

GRANT EXECUTE ON FUNCTION TextMatch TO PUBLIC

PUBLIC FUNCTION
MarkSubtexts(text StructuredText, pattern StructuredTextPattern)
RETURNS StructuredText
BEGIN
  DECLARE search  StructuredTextSearch;
  DECLARE new     StructuredText;

  IF text IS NULL OR pattern IS NULL THEN
    RETURN NULL;
  END IF;

  SET search = StructuredTextSearch(pattern);
  SET new    = text;
  SET new..Marks = search..Nodeids(text);

  RETURN new;
END,

GRANT EXECUTE TO FUNCTION MarkSubtexts TO PUBLIC
PUBLIC FUNCTION
MarkUnion(text1 StructuredText, text2 StructuredText)
RETURNS StructuredText
BEGIN
  DECLARE DifferentTexts EXCEPTION FOR SQLSTATE 'G2001';
  DECLARE new StructuredText;

  IF text1 IS NULL OR text2 IS NULL THEN
    RETURN NULL;
  END IF;

  IF text1..Tree..NodeId <> text2..Tree..NodeId THEN
    SIGNAL DifferentTexts;
  ELSE
    SET new = text1;
    SET new..Marks = text1..Marks..Union(text2..Marks);
    RETURN new;
  END IF;
END,

GRANT EXECUTE ON FUNCTION MarkUnion TO PUBLIC

PUBLIC FUNCTION
MarkIntersect(text1 StructuredText, text2 StructuredText)
RETURNS StructuredText
BEGIN
  DECLARE DifferentTexts EXCEPTION FOR SQLSTATE 'G2002';
  DECLARE new StructuredText;

  IF text1 IS NULL OR text2 IS NULL THEN
    RETURN NULL;
  END IF;

  IF text1..Tree..NodeId <> text2..Tree..NodeId THEN
    SIGNAL DifferentTexts;
  ELSE
    SET new = text1;
    SET new..Marks = text1..Marks..Intersect(text2..Marks);
    RETURN new;
  END IF;
END,

GRANT EXECUTE ON FUNCTION MarkIntersect TO PUBLIC

PUBLIC FUNCTION
MarkExcept(text1 StructuredText, text2 StructuredText)
RETURNS StructuredText
BEGIN
  DECLARE DifferentTexts EXCEPTION FOR SQLSTATE 'G2003';
  DECLARE new StructuredText;

  IF text1 IS NULL OR text2 IS NULL THEN
    RETURN NULL;
  END IF;

  IF text1..Tree..NodeId <> text2..Tree..NodeId THEN
    SIGNAL DifferentTexts;
  ELSE
    SET new = text1;
    SET new..Marks = text1..Marks..Except(text2..Marks);
    RETURN new;
  END IF;
END,
PUBLIC FUNCTION
IsolateSubtexts(text StructuredText)
RETURNS UNNESTED_TABLE(
    SET(ROW(C1 StructuredText NOT NULL,
            C2 StructuredText NOT NULL)))
-- !! i.e. a two column relation !!
BEGIN
DECLARE sametext StructuredText;
DECLARE subtext StructuredText;
DECLARE nodeid NodeID;
DECLARE tuple ROW(C1 StructuredText NOT NULL,
                 C2 StructuredText NOT NULL);
DECLARE new SET(ROW(C1 StructuredText NOT NULL,
                     C2 StructuredText NOT NULL));

SET new = CAST(EMPTY AS
    SET(ROW(C1 StructuredText NOT NULL,
             C2 StructuredText NOT NULL)));

IF text is NULL THEN
    RETURN UNNESTED_TABLE(new); -- !! i.e. empty table !!
END IF;

FOR nodeid AS SELECT m from TABLE(text..Marks..Members) T(m) DO
    SET sametext = text;
    SET sametext..Marks = SetOfNodeIds(nodeid);
    SET subtext = text..RootedAt(nodeid);
    SET subtext..Marks = text..Marks..Intersect(
        subtext..Tree..Ids..Except(nodeid));
    SET tuple = ROW{sametext, subtext};
    SET new = new S_UNION SET{tuple};
END FOR;

RETURN UNNESTED_TABLE(new);
END,

PUBLIC FUNCTION
ExtractSubtexts(text StructuredText,
columns INTEGER,
pattern StructuredTextPattern)
RETURNS UNNESTED_TABLE(SET(LIST(StructuredText)))
-- !! i.e. A relation having columns columns !!
BEGIN
DECLARE IllegalColumns EXCEPTION FOR SQLSTATE 'G2010';
DECLARE IncorrectColumns EXCEPTION FOR SQLSTATE 'G2011';
DECLARE search StructuredTextSearch;
DECLARE nodeid NodeID;
DECLARE mark NodeID;
DECLARE match LIST(NodeID);
DECLARE matches SetOfListsOfNodeIds;
DECLARE undermark Boolean;
DECLARE rest LIST(NodeID);
DECLARE markedtree StructuredTextTree;
DECLARE subtext StructuredText;
DECLARE tuple LIST(StructuredText);
DECLARE new SET(LIST(StructuredText)); -- !! relation !!
SET new = CAST(EMPTY AS SET(LIST(StructuredText)));

IF columns IS NULL OR columns < 2 THEN
  SIGNAL IllegalColumns;
END IF;

IF text IS NULL OR pattern IS NULL THEN
  RETURN UNNESTED_TABLE(new);  -- !! ie empty table !
END IF;

SET search  = StructuredTextSearch(pattern);
SET matches = search..MatchesNodeids(text);
IF matches..Present = FALSE THEN
  RETURN UNNESTED_TABLE(new);  -- !! ie empty table !
END IF;

IF columns <> matches..Degree + 1 THEN
  SIGNAL IncorrectColumns;
END IF;

FOR match AS SELECT m from TABLE(matches..Members) T(m)
DO
  SET tuple          = CAST(EMPTY AS LIST(StructuredText));
  SET subtext        = text;
  SET subtext..Marks = SetOfNodeIds();
  WHILE match <> EMPTY DO
    -- !! Add direct descendants of subtext in match !
    -- !! to subtext..Marks !
    SET rest = match;
    WHILE rest <> EMPTY DO
      -- !! Extract nodeid from head of list !
      -- !! Set rest to the tail of the list !
      SET nodeid = ELEMENT(rest, 1);
      SET rest   = SUBLIST(rest, 2, ELEMENT_LENGTH(rest)-1);
      IF subtext..Tree..InTree(nodeid) THEN
        -- i.e. nodeid is a marked node under subtext
        SET undermark = FALSE;
        FOR mark AS
          SELECT m FROM TABLE(subtext..Marks..Members) T(m)
        DO
          SET markedtree = subtext..Tree..Subtree(mark);
          IF markedtree..InTree(nodeid) THEN
            SET undermark = TRUE;
          END IF;
        END FOR;
        IF NOT undermark THEN
          -- i.e. nodeid is not under any marked node which
          -- is itself under subtext
          subtext..Marks = subtext..Marks..Union(nodeid);
        END IF;
      END IF;
    END WHILE;
    -- !! Add subtext to next column in tuple !
    SET tuple   = CONCATENATE(tuple, LIST(subtext));
  END WHILE;
END FOR;

-- !! Move on to next nodeid in match !!
-- !! Set nodeid to head of match (earliest in list) !!
-- !! Set match to tail of match (rest of list) !!

SET nodeid = ELEMENT(match, 1);
SET match = SUBLIST(match, 2, ELEMENT_LENGTH(match)-1);
SET subtext = text..RootedAt(nodeid);
SET subtext..Marks = SetOfNodeIds();
END WHILE;

SET tuple = CONCATENATE(tuple, LIST(subtext));
SET new = new S_UNION SET{tuple};
END FOR;

RETURN UNNESTED_TABLE(new);
END,

GRANT EXECUTE ON FUNCTION ExtractSubtexts TO PUBLIC

PUBLIC FUNCTION
AggregateMarks(text StructuredText)
RETURNS StructuredText
STATE row(new StructuredText, first Boolean)
INITIALIZE AggregateMarks_initialize
ITERATE AggregateMarks_iterate
TERMINATE AggregateMarks_terminate;

GRANT EXECUTE ON FUNCTION AggregateMarks TO PUBLIC

PRIVATE PROCEDURE
AggregateMarks_initialize(OUT work ROW(first Boolean,
new   StructuredText))
BEGIN
SET work = ROW{TRUE, NULL};
END,

PRIVATE PROCEDURE
AggregateMarks_iterate(IN    text StructuredText,
INOUT work ROW(first Boolean,
new   StructuredText))
BEGIN
IF work..first = TRUE THEN
SET work..new    = text;
SET work..first  = FALSE;
ELSE
SET work..new    = work..new..MarkUnion(text);
END IF;
END,

PRIVATE PROCEDURE
AggregateMarks_terminate(IN work ROW(first Boolean,
new   StructuredText))
RETURNS StructuredText
BEGIN
RETURN work..new;
END,

PRIVATE FUNCTION
ParseText(string CHARACTER VARYING(max_text_length),
parser CHARACTER VARYING(max_parser_length))
RETURNS StructuredTextTree
BEGIN
DECLARE UnableToParseText EXCEPTION FOR SQLSTATE 'G2100';
DECLARE new StructuredTextTree;
CASE lower_case(parser)
   --
   -- !!! See Description
   --
END CASE;
RETURN new;
END,
PRIVATE FUNCTION
RootedAt(text StructuredText, nodeid NodeID)
RETURNS StructuredText
BEGIN
   DECLARE subtree StructuredTextTree;
   DECLARE new     StructuredText;
   SET new        = text;
   SET new..Tree  = text..Tree..Subtree(nodeid);
   SET new..Marks = SetOfNodeIds();
   RETURN new;
END,

Definitional rules

The values of max_text_length, max_parser_length and max_method_length are implementation defined.

Description

StructuredText

StructuredText is any text that has a hierarchical structure defined by an implicit or explicit grammar. Arbitrary sets of nodes (structural components) within instances of structured text may be marked.

Grammar

The private attribute Grammar which is a StructuredTextGrammar abstract data type is introduced in section 7.2.

Tree

The private attribute Tree which is a StructuredTextTree abstract data type is introduced in section 7.4.

Marks

The private attribute Marks are a set of zero or more node identifiers referencing a set of nodes in a StructuredText..Tree. These are introduced as an abstract data type in section 7.5.

TextToString

The public function TextToString uses the specified method to convert an abstract structured text into a string. As a convenience and to facilitate casting of Structured Text into character strings a 'DEFAULT' method for converting structured text into a character string may be requested either implicitly or explicitly. This 'DEFAULT' method is defined by the implementation. The 'CLEAR' method translates text into a string containing no markup. The 'ROOT' method returns the node label (for example an SGML generic identifier) associated with the root of the StructuredText..Tree. Other methods of translating an abstract structured text into an appropriately formatted string may exist. Such methods are dependent both on the implementation of this standard and on the parser(s) used to construct instances of structured text.
TextToGrammar
The public function *TextToGrammar* returns the internal grammar associated with the specified text.

CountMarks
Returns a count of the number of marked nodes in the structured text.

KeepMarks
Preserves the specified marks within the structured text.

TextMatch
Tests to see if a structured text pattern exists in the structured text. The *StructuredTextPattern* is introduced in section 7.3

MarkSubtexts
Associates marks with the structured text according to a given structured text pattern.

MarkUnion
Forms the union of marks present in two instances comparable of structured text.

MarkIntersect
Forms the intersection of marks present in two instances of comparable structured text.

MarkExcept
Eliminates marks in the first instance of comparable structured text that occur in a second instance of structured text.

IsolateSubtexts
Forms a two column relation in which marked subtexts within an instance of structured text may be related to this instance of structured text while being used and manipulated independently of this instance of structured text.

Specifically for each mark within the input text a row is formed. The first column in this row contains the input text in which only this mark is present. The second column contains the corresponding marked subtext. Within this marked subtext the root node will not be marked. However marks present in the input text which refer to nodes below the root of this subtree will be preserved in this subtext.

ExtractSubtexts
Forms a relation in which nodes within the text instance that collectively match a provided pattern, are (if themselves flagged within the pattern) emitted as new subtext elements of a tuple.

Specifically the first column in this relation will always contain the entire text operated on. Later columns will contain subtexts corresponding to flagged *node rule>*s within the text pattern, in the order that these *node rule>*apos; occurred within the text pattern. The subtexts in the second and subsequent columns of the tuple will necessarily be contained within some earlier text(s) with the tuple. The nearest ancestor text
within this same tuple will contain a mark for this subtext, allowing the context of this subtext to be preserved, even after this subtext has been extracted from its larger context.

The parameter *columns* is constrained to be either a constant or a value computable when *ExtractSubtexts* is encountered within a SQL expression. It provides a mechanism for the number of columns expected to be returned by an invocation of *ExtractSubtexts* to be determined when SQL statements are parsed.

**AggregateMarks**

Forms the union of marks present in a grouped column of instances of structured text.

**ParseText**

Uses the specified parser to parse the input character string, verify that this string conforms with input accepted by this parser, and to convert this string into an appropriate instance of *StructuredTextTree*. An exception condition is raised if the input string is not accepted by the specified parser.

**RootedAt**

Returns a new instance of structured text corresponding to the subtext rooted at a specified node in the parse tree associated with the specified structured text.
7.2 StructuredTextGrammar Abstract Data Type

Function

This abstract data type provides for the construction of structured text grammars that can be stored and manipulated independently of the structured text(s) that they define.

Definition

CREATE DISTINCT TYPE NodeLabel
    AS CHARACTER VARYING(max_node_label_length);

CREATE DISTINCT TYPE NodeDescription
    AS CHARACTER VARYING(max_node_description_length);

CREATE VALUE TYPE StructuredTextGrammar (PRIVATE Parser CHARACTER VARYING(max_parser_length), Grammar InternalGrammar, -- !! Implementation defined !!

PUBLIC FUNCTION StructuredTextGrammar(string CHARACTER VARYING(max_text_length), parser CHARACTER VARYING(max_parser_length)) RETURNS StructuredTextGrammar
BEGIN
DECLARE new StructuredTextGrammar;

IF string IS NULL OR parser IS NULL THEN
    RETURN NULL;
END IF;
SET new.Parser = StructuredTextGrammar();
SET new.Grammar = ParseGrammar(string, parser);

RETURN new;
END,

PUBLIC FUNCTION ParserUsed(grammar StructuredTextGrammar) RETURNS CHARACTER VARYING(max_parser_length)
BEGIN
    IF grammar IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN grammar.Parser;
END,

GRANT EXECUTE ON FUNCTION ParserUsed TO PUBLIC

PUBLIC FUNCTION GrammarToText(grammar StructuredTextGrammar) RETURNS StructuredText
BEGIN
DECLARE new StructuredText;

IF grammar IS NULL THEN
    RETURN NULL;
END IF;
CASE lower_case(grammar.Parser)
WHEN
```
--
-- !! See Description
--
END CASE;

RETURN new;
END,

GRANT EXECUTE ON FUNCTION GrammarToText TO PUBLIC

PUBLIC FUNCTION
  GrammarRoot(grammar StructuredTextGrammar)
  RETURNS NodeLabel
  BEGIN
    -- See description
    END,

GRANT EXECUTE ON FUNCTION GrammarRoot TO PUBLIC

PUBLIC FUNCTION
  GrammarElements(grammar StructuredTextGrammar)
  RETURNS UNNESTED_TABLE(
    SET(ROW(C1 NodeLabel NOT NULL,
             C2 NodeDescription)))
  BEGIN
    DECLARE nodelabel     NodeLabel;
    DECLARE info          NodeDescription;
    DECLARE new SET(ROW(C1 NodeLabel NOT NULL,
                         C2 NodeDescription));

    SET new = CAST(EMPTY AS SET(
                      ROW(C1 Nodelabel NOT NULL,
                          C2 NodeDescription)));

    IF grammar IS NULL THEN
      RETURN UNNESTED_TABLE(new);
    END IF;

    FOR nodelabel AS
      SELECT m FROM TABLE(grammar..NodeLabels) T(m) DO
        SET info = grammar..description(nodelabel);
        SET new  = new S_UNION SET{ROW{nodelabel, info}};
      END FOR;

    RETURN UNNESTED_TABLE(new);
  END,

GRANT EXECUTE ON FUNCTION GrammarElements TO PUBLIC

PUBLIC FUNCTION
  GrammarHierarchy(grammar StructuredTextGrammar)
  RETURNS UNNESTED_TABLE(
    SET(ROW(c1 NodeLabel NOT NULL,
            c2 NodeLabel NOT NULL,
            c3 Boolean)))
  BEGIN
    DECLARE e1     NodeLabel;
    DECLARE e2     NodeLabel;
    DECLARE e3     Boolean;
    DECLARE new    SET(ROW(c1 NodeLabel NOT NULL,
                            c2 NodeLabel NOT NULL,
                            c3 Boolean));

    SET new = CAST(EMPTY AS SET(ROW(
```
c1 NodeLabel NOT NULL,
c2 NodeLabel NOT NULL,
c3 Boolean));

IF grammar IS NULL THEN
  RETURN UNNESTED_TABLE(new);
END IF;

FOR e1 AS SELECT m FROM TABLE(grammar..NodeLabels) T(m) DO
  FOR e2 AS SELECT m FROM TABLE(grammar..NodeLabels) T(m) DO
    IF grammar..Ancestor(e1, e2) = TRUE THEN
      SET e3  = grammar..MaybeParent(e1, e2);
      SET new = new S_UNION SET{ROW{e1, e2, e3}};
    END IF;
  END FOR;
END FOR;

RETURN UNNESTED_TABLE(new);
END,

GRANT EXECUTE ON FUNCTION GrammarHierarchy TO PUBLIC

PUBLIC FUNCTION
  ClearText(grammar StructuredTextGrammar,
            string CHARACTER VARYING(max_text_length))
  RETURNS CHARACTER VARYING(max_text_length)
BEGIN
  -- See description
  -- Removes any identifiable internal markup from the input string
END,

PRIVATE FUNCTION
  ParseGrammar(string CHARACTER VARYING(max_text_length),
                parser CHARACTER VARYING(max_parser_length))
  RETURNS InternalGrammar
BEGIN
  DECLARE UnableToParseGrammar EXCEPTION FOR SQLSTATE 'G2101';
  DECLARE new InternalGrammar;
  CASE lower_case(parser)
    --
    -- !! See Description
    --
    END CASE;

  RETURN new;
END,

PRIVATE FUNCTION
  NodeLabels(grammar StructuredTextGrammar)
  RETURNS SET(NodeLabel)
BEGIN
  -- See description
END,

PRIVATE FUNCTION
  Description(grammar   StructuredTextGrammar,
              nodelabel NodeLabel)
  RETURNS NodeDescription
BEGIN
  -- See description
END,
PRIVATE FUNCTION
Ancestor(grammar StructuredTextGrammar, e1 NodeLabel, e2 NodeLabel)
  RETURNS Boolean
BEGIN
  -- See description
  -- Returns TRUE iff e1 may within the grammar be an ancestor of e2
END,

PRIVATE FUNCTION
MaybeParent(grammar StructuredTextGrammar, e1 NodeLabel, e2 NodeLabel)
  RETURNS Boolean
BEGIN
  -- See description
  -- Returns TRUE iff e1 may be a parent of e2, else FALSE.
  -- Returns NULL if it is not known if e1 may be a parent of e2.
END
)

Definition rules

The values of max_node_label_length and max_node_description_length are implementation defined.

Description

StructuredTextGrammar

The StructuredTextGrammar constructor returns an abstract data type encapsulating both the internal
grammar associated with a text, and the operations which may be performed on such a grammar.

ParserUsed

As a convenience the name of the parser used to create each instance of StructuredTextGrammar (and thus
StructuredText) is preserved within that instance of StructuredTextGrammar. This read-only attribute may
be retrieved by using the public function ParserUsed.

GrammarToText

The public function GrammarToText converts the internal grammar associated with a structured text into
an instance of StructuredText, thus providing a StructuredText representation of data preserved within the
implementation defined internal grammar. This new StructuredText may provide additional information
about the internal structure of instances of StructuredText, and/or provide information about the external
grammar used to construct such instances of StructuredText. All instances of Structured Text produced
from a grammar (derived from a common parser) share a common internal meta grammar. If no
StructuredText is associated with the internal grammar, then this function returns null.

GrammarRoot

The public function GrammarRoot returns the root node label within the grammar.

GrammarElements

The public function GrammarElements returns a relation identifying the values of all node labels within the
internal grammar which may be used to query and manipulate any instances of StructuredText having this
internal grammar. Optionally, an informal description of each node label (and potentially other
information) may be returned in tuples within this relation.

GrammarHierarchy
The public function `GrammarHierarchy` returns a relation describing the valid hierarchical relationships between node labels within an internal grammar. Parent/child relationships may also optionally be described within this relation. This information may be used to assist in formulating sensible queries against structured text defined using this internal grammar.

**ClearText**

The public function `ClearText` is intended for use only within the `StructuredText` abstract data type. It removes all identifiable internal structural markup from the input text string, and returns the resulting `ClearText` string. For each grammar produced using a specific parser, the identifiable internal structural markup is implementation defined.

**ParseGrammar**

The private function `ParseGrammar` uses the specified parser to recover the grammar used when parsing the input string. The internal structure of the resulting parsed grammar is not defined. An exception condition is raised if no grammar can be associated with the input string.

In practice it is likely that the functionality of `ParseText` and `ParseGrammar` would be tightly coupled.

**NodeLabels**

The private function `NodeLabels` returns the set of node labels that occur in instances of `StructuredText` conforming to this grammar.

**Description**

The private function `Description` returns a free format description about a given node label within the grammar. The description may be `null`.

**Ancestor**

The private function `Ancestor` returns `true` when the first input node label within a grammar may legitimately be an ancestor of the second input node label, and `false` otherwise.

**MaybeParent**

The private function `MaybeParent` returns `true` when the first input node label within a grammar may legitimately be a parent of the second input node label, and `false` when it may not. Returns `null` if the implementation of this function is unable to distinguish between these two cases.
7.3 StructuredTextSearch Abstract Data Type

Function

This type provides for parsing structured search patterns, and defines how such parsed patterns are matched against nodes in StructuredTextTree's.

Definition

CREATE DISTINCT TYPE StructuredTextPattern
    AS CHARACTER VARYING(maxstructured_pattern_length)

Definition rules

1. The value of maxstructured_pattern_length is an implementation-dependent maximum length for the character representation of an instance of a StructuredTextPattern.

2. Instances of StructuredTextPattern, if valid, can be parsed using the following BNF for <pattern>:

   <pattern> ::= <node_rule> [ <forest> ] | <node_rule>
   <node_rule> ::= <node_pattern> <comma> <node_pattern> | <node_pattern>
   <node_pattern> ::= <node_label> { <text_pattern> } | <node_label>
   <node_label> ::= <characters>
   <characters> ::= character <characters> | <character>
   <character> ::= !! Any character subject to rules below !!
   <text_pattern> ::= <search expression>

   <ampersand> ::= &
   <comma> ::= ,
   <rooted> ::= ^
   <was_marked> ::= @
   <flagged> ::= #

3. The grammar for <search expression> is described in section 5.2.

4. The grammar above is augmented by the operators “..” and “.” which act as syntactic sugar defined by the rewriting rules shown below:

   <node_rule> .. <marked_rule> [ <forest> ] ->
   <node_rule> [ <marked_rule> [ <forest> ] ]
   <node_rule> .. <marked_rule> ->
   <node_rule> [ <marked_rule> ]
   <node_rule> . <marked_rule> [ <forest> ] ->
   <node_rule> [^<marked_rule> [ <forest> ] ]
   <node_rule> . <marked_rule> ->
   <node_rule> [^<marked_rule> ]

5. The grammar above is further augmented by allowing <ampersand> to replace <comma> where ever <comma> is valid in the above syntax.
6. Each of the characters “[”, “]”, “{”, “}”, “#”, “@”, “^”, “{”, “}”, “.” and “&” must be escaped when they occur within an element to avoid ambiguity in the above grammar.

Description

A structured text pattern describes the pattern against which instances of StructuredText are to be matched. When a StructuredText matches a pattern, this pattern also defines what (if anything) is to be marked or extracted from this StructuredText.

A match occurs if all <node_pattern>’s in the StructuredTextPattern can simultaneously be matched against nodes in the StructuredText while satisfying all of the following properties:

1. Each <node_rule> is matched against a distinct node in the StructuredTextTree.
2. The node matching each <node_rule> has a node label which is “like” the <node_label> present within this <node_rule>.
3. When a <text_pattern> is specified within a <node_rule>, the node matching this <node_rule> subsumes FullText which “contains” this <text_pattern>.
4. Each <node_rule> which specifies that the matching node <was_marked> matches a previously marked node within the StructuredText.
5. Any bracketed <forest> of <pattern>’s, following a <node_rule> within a <pattern>, match subtrees which are descendants of the node matching this <node_rule>.
6. Each <pattern> in such a <forest> of <pattern>’s matches a disjoint subtree. Disjoint subtrees share no common nodes.
7. Each <node_rule> which specifies that the matching node is <rooted> matches either the root node of the StructuredTextTree, or a node whose parent is simultaneously matched by a <node_rule>.
8. Any list of <pattern> within a <forest> that are separated by <ampersand> match text without additional regard to the order of that text. The <ampersand> has higher precedence than the <comma>.
9. Any two <pattern>’s (or lists of <pattern>’s separated by <ampersand>) within a <forest> which are separated by <comma> impose an additional ordering constraint on the matched text. Such <comma> separated <pattern>’s match text only if every node simultaneously matched by the left <pattern> occurs before any node simultaneously matched by the right <pattern>.

When a match occurs, all nodes matching <node_rules> which specify that the node is to be <flagged> are marked or extracted as appropriate.

To implement the <ampersand> operator, form the equivalent set of structured text patterns in which every <ampersand> operator has been placed by the <comma> operator. This can be done by iteratively replacing the maximal (longest) lists of <pattern> separated by <ampersand> with every permutation of this list of <pattern> separated by <comma>, thus forming new sets of structured text patterns, until all derivable patterns containing no <ampersand> have been produced.

Then invoke the private function GetMatches (documented below) on each such derivable pattern, reorder the elements in the lists returned (when necessary) so that the order of node identifiers in lists continues to correspond to the order of <flagged> patterns within the original structured text pattern against which they were matched, and construct the union of all such resulting sets of lists.
**Definition**

CREATE VALUE TYPE SearchRuleTree (  
Children    LIST(SearchRuleTree), --!! in left to right child order  
Nodelabel   NodeLabel,            --!! node label to match if any  
Textpattern FT_pattern,           --!! fulltext content pattern if any  
Rooted      Boolean,              --!! true if node must be rooted  
WasMarked   Boolean,              --!! true if node was earlier marked  
Flagged     Boolean               --!! true if should extract/mark node )

**Description**

Each node in a SearchRuleTree has the attributes described above. The values in these attributes are determined by the function ParsePattern when it parses the pattern string, producing an instance of a SearchRuleTree.

**Definition**

CREATE VALUE TYPE StructuredTextSearch (  
PRIVATE Rule SearchRuleTree;  
PUBLIC FUNCTION StructuredTextSearch(pattern StructuredTextPattern)  
RETURNS StructuredTextSearch  
BEGIN  
DECLARE new StructuredTextSearch;  
SET new       = StructuredTextSearch();  
SET new..Rule = ParsePattern(pattern);  
RETURN new;  
END,

PUBLIC FUNCTION Matches(pattern StructuredTextSearch, text StructuredText)  
RETURNS Boolean  
BEGIN  
DECLARE relation  SetOfListsOfNodeIds;  
SET relation = GetMatches(pattern..Rule, text..Tree, text..Marks);  
RETURN relation..Present;  
END,

PUBLIC FUNCTION Nodeids(pattern StructuredTextSearch, text StructuredText)  
RETURNS SetOfNodeIds  
BEGIN  
DECLARE nodeid   NodeID;  
DECLARE tuple    LIST(NodeID);  
DECLARE relation SetOfListsOfNodeIds;  
DECLARE new      SetOfNodeIds;  
relation = GetMatches(pattern..Rule, text..Tree, text..Marks);  
SET new = SetOfNodeIds();

23
IF relation..Degree <> 0 THEN
  FOR tuple in SELECT m from TABLE(relation..Members) T(m) DO
    FOR nodeid in SELECT m from TABLE(tuple) T(m) DO
      SET new = new..Union(nodeid);
    END FOR;
  END FOR;
END IF;

RETURN new;
END,

PUBLIC FUNCTION
MatchesNodeids(pattern StructuredTextSearch, text StructuredText)
RETURNS SetOfListsOfNodeIds
BEGIN
  RETURN GetMatches(pattern..Rule, text..Tree, text..Marks);
END,

PRIVATE FUNCTION
ParsePattern(pattern StructuredTextPattern)
RETURNS SearchRuleTree
BEGIN
  DECLARE UnableToParsePattern EXCEPTION FOR SQLSTATE 'G2102';
  --
  -- !! See Description
  --
END,

PRIVATE FUNCTION
GetMatches(pattern SearchRuleTree, text_tree StructuredTextTree, oldmarks SetOfNodeIds)
RETURNS SetOfListsOfNodeIds
BEGIN
  DECLARE child_tree StructuredTextTree;
  DECLARE relation1 SetOfListsOfNodeIds;
  DECLARE relation2 SetOfListsOfNodeIds;
  DECLARE new SetOfListsOfNodeIds;
  SET new = SetOfListsOfNodeIds(FALSE);
  IF NOT pattern..Rooted THEN
    FOR child_tree AS SELECT m from TABLE(text_tree..Children) T(m) DO
      relation1 = GetMatches(pattern, child_tree, oldmarks);
      IF relation1..Present = TRUE THEN
        SET new = new..Union(relation1);
      END IF;
    END FOR;
  END IF;
  relation1 = MatchRootNode(pattern, text_tree, oldmarks);
  IF relation1..Present THEN
    relation2 = MatchChildren(pattern..Children, text_tree..Children, oldmarks);
    IF relation2..Present THEN
      SET relation1 = relation1..CrossProduct(relation2);
      SET new = new..Union(relation1);
    END IF;
  END IF;
END;
RETURN new;
END,

PRIVATE FUNCTION
MatchRootNode(pattern_node SearchRuleTree,
    tree_node StructuredTextTree,
    oldmarks SetOfNodeIds)
RETURNS SetOfListsOfNodeIds
BEGIN
    IF pattern_node.WasMarked THEN -- !! Ie began with '@' !!
        IF NOT oldmarks.HasMember(tree_node.NodeId) THEN
            RETURN SetOfListsOfNodeIds(FALSE);
        END IF;
    END IF;
    IF NOT tree_node.RootMatches(pattern_node.Nodelabel,
        pattern_node.Textpattern) THEN
        RETURN SetOfListsOfNodeIds(FALSE);
    END IF;
    IF pattern_node.Flagged THEN -- !! Ie terminated with '#' !!
        RETURN SetOfListsOfNodeIds(tree_node.NodeId);
    END IF;
    RETURN SetOfListsOfNodeIds(TRUE);
END,

PRIVATE FUNCTION
MatchChildren(pattern LIST(SearchRuleTree),
    tree LIST(StructuredTextTree),
    oldmarks SetOfNodeIds)
RETURNS SetOfListsOfNodeIds
BEGIN
    DECLARE elements INTEGER;
    DECLARE count INTEGER;
    DECLARE left_pattern SearchRuleTree;
    DECLARE next_pattern SearchRuleTree;
    DECLARE left_group LIST(SearchRuleTree);
    DECLARE rest_pattern LIST(SearchRuleTree);
    DECLARE left StructuredTextTree;
    DECLARE rest LIST(StructuredTextTree);
    DECLARE relation1 SetOfListsOfNodeIds;
    DECLARE relation2 SetOfListsOfNodeIds;
    DECLARE new SetOfListsOfNodeIds;
    SET elements = ELEMENT_LENGTH(pattern);
    IF elements = 0 THEN -- !! No patterns to match against text
        RETURN SetOfListsOfNodeIds(TRUE);
    END IF;
    IF tree = EMPTY THEN -- !! No text to match against pattern(s)
        RETURN SetOfListsOfNodeIds(FALSE);
    END IF;
    SET new = SetOfListsOfNodeIds(FALSE);
    -- !! Test for possibility that left_pattern matches a distinct
    -- !! child subtree from all later subpatterns in pattern
    SET left_pattern = ELEMENT(pattern, 1);
    SET rest_pattern = SUBLIST(pattern, 2, elements - 1);
    SET rest = tree;

-- !! Test each child subtree that left_pattern might match while
-- !! simultaneously matching subsequent subtrees against the
-- !! remaining rest_pattern

WHILE rest <> EMPTY DO
  SET left  = ELEMENT(rest, 1);
  SET rest  = SUBLIST(rest, 2, ELEMENT_LENGTH(rest) - 1);
  SET relation1 = GetMatches(left_pattern, left, oldmarks);

  IF relation1..Present THEN
    relation2 = MatchChildren(rest_pattern, rest, oldmarks);
    IF relation2..Present THEN
      SET relation1 = relation1..CrossProduct(relation2);
      SET new       = new..Union(relation1);
    END IF;
  END IF;
END WHILE;

-- !! Test for possibility that an unrooted left sublist of
-- !! pattern (having two or more subpatterns) matches within
-- !! a common member of tree

IF left_pattern..Rooted THEN
  RETURN new;
END IF;

SET count    = 2;
WHILE count <= elements DO
  SET next_pattern = ELEMENT(pattern, count);
  IF next_pattern..Rooted THEN
    RETURN new;
  END IF;
  SET left_group   = SUBLIST(pattern, 1, count);
  SET rest_pattern = SUBLIST(pattern, count+1, elements - count);
  SET rest         = tree;

  WHILE rest <> EMPTY DO
    SET left  = ELEMENT(rest, 1);
    SET rest  = SUBLIST(rest, 2, ELEMENT_LENGTH(rest) - 1);
    SET relation1 = MatchChildren(left_group, left..Children, oldmarks);

    IF relation1..Present THEN
      relation2 = MatchChildren(rest_pattern, rest, oldmarks);
      IF relation2..Present THEN
        SET relation1 = relation1..CrossProduct(relation2);
        SET new       = new..Union(relation1);
      END IF;
    END IF;
  END WHILE;

  RETURN new;
END WHILE;

)

Description

Instances of type StructuredTextSearch encapsulate the translation of structured text patterns into the StructuredTextSearch abstract data type.

Instances of type StructuredTextSearch also encapsulate the operations performed in identifying nodes (by node identifier) within the StructuredTextTree that match the specified StructuredTextPattern.
The `SearchRuleTree` is a tree whose nodes each contain up to two strings and a collection of flags.

**StructuredTextSearch**

This function validates a structured text pattern and converts it into a parsed structure that can be more easily matched against structured text.

**Matches**

This function returns `true` if the text matches the structured text pattern in at least one way, else `false`.

**Nodeids**

This function returns all of the node identifiers within the text tree which (when the structured text pattern is matched against the tree) occur in nodes matching flagged nodes within the text pattern. The `SetOfNodeIds` abstract data type used to represent and manipulate this set of node identifiers is introduced in section 7.5.

**MatchesNodeids**

This function returns a relation describing how nodes within the structured text tree can be matched against flagged nodes within the structured text pattern. The `SetOfListsOfNodeIds` abstract data type used to represent and manipulate the resulting relation is introduced in section 7.6.

**ParsePattern**

This function parses the structured text pattern against the grammar provided at the beginning of this section and constructs the corresponding `SearchRuleTree` describing this pattern. An exception condition is raised if a `StructuredTextPattern` cannot be parsed.

When parsing the input pattern the parser builds a tree structure having the following properties. For every `<node rule>` encountered within the input pattern, a corresponding node is included within the resulting `SearchRuleTree`. This node contains as attributes the `<node label>` and `<text pattern>` strings associated with this `<node rule>`, and boolean flags identifying if this `<node rule>` contained `<rooted>`, `<was_marked>` and/or `<flagged>` productions. The `<text pattern>` is null if not present within the input pattern.

Nodes derived from `<pattern>`’s that form a `<forest>` are children of the node derived from the `<node rule>` appearing immediately before this `<forest>`. The order in which a preorder traversal visits nodes within the resulting `SearchRuleTree` corresponds to the order in which `<node rule>`’s that generated these nodes were encountered within the input pattern when scanning this pattern from left to right. Thus the structured text pattern language uses a one dimensional representation to describe a specific two dimensional tree pattern structure whose nodes each have an associated node value, text value and collection of boolean flags.

**GetMatches**

This private function returns a boolean result indicating if the structured text pattern matched the structured text tree (in at least one way). If some nodes within the structured text pattern are flagged then this function also returns a relation having as many columns as flagged nodes. For each distinct method of matching flagged nodes in the structured text pattern with specific nodes in the structured text tree (while concurrently matching the entire structured text pattern against some collective set of nodes in the structured text tree) a row is formed within the relation. Each column within this row contains the value of the node identifier of a node matching a flagged portion of the structured text pattern. Node identifiers (and
thus the columns) within each row (when examined from left to right) are ordered so that they occur in the same sequence as encountered when performing a pre-order traversal on the search rule tree.

**MatchRootNode**

This private function matches a given `<node rule>` in the structured text pattern with a given `node` in the structured text tree. The matching may be performed by comparing the label in the structured text tree with a (potentially wild carded) string pattern, by testing to see if the Full-Text subsumed by this structured text node matches the specified contains clause, or both. It is assumed that if structural nodes must be identified by role, category, type, etc. that this information will be encoded in some easily identifiable way within their node label. This function returns a boolean result indicating if a match occurred. If the `<node rule>` was `<flagged>` and a match occurred, a one row, one column relation is returned containing the node identifier of the given node in the text tree.

**MatchChildren**

This private function matches all of the children of a given `<node rule>` in the structured text pattern with corresponding descendants of a given node in the text tree. It returns a boolean result indicating if all children could be concurrently matched. It also returns a relation describing how flagged nodes in the structured text pattern matched nodes in the structured text tree, as described above.
7.4 StructuredTextTree Abstract Data Type

Function

This type provides a mechanism for representing structured text as a hierarchy of Full-Text, and for defining operations that may be performed on the structured text nodes within the hierarchical tree structure. Each node in such a hierarchical tree structure has a unique node identifier, a label which describes the structure represented by this node, and the Full-Text subsumed by this structural node.

Definition

CREATE DISTINCT TYPE NodeID
    AS INTEGER; -- !! Or more appropriate data type !!

CREATE VALUE TYPE StructuredTextTree (  
    -- !! The following variables are present in every node within a  
    -- !! structured text tree  
    PUBLIC NodeId NodeID,  
    Children LIST(StructuredTextTree), -- In left to right order  
    PRIVATE Nodelabel NodeLabel,  
    Subsumed FullText,  
    PUBLIC FUNCTION  
    StructuredTextTree(label NodeLabel,  
        text FullText)  
    RETURNS StructuredTextTree  
    BEGIN  
    DECLARE new StructuredTextTree;
      SET new           = StructuredTextTree();
      SET new..NodeId   = UniqueIdentifier();
      SET new..Nodelabel = label; --!! Structural identifying label !!
      SET new..Subsumed = text;  --!! Text subsumed by this structure !!
      SET new..Children = CAST(EMPTY AS LIST(StructuredTextTree));
    RETURN new;
    END,  
    PUBLIC FUNCTION  
    RootMatches(root StructuredTextTree,  
        nodelabel NodeLabel,  
        textpattern FT_pattern)  
    RETURNS Boolean  
    BEGIN  
    IF nodelabel IS NOT NULL THEN
      IF NOT root..Nodelabel LIKE nodelabel THEN
        RETURN FALSE;
      END IF;
    END IF;
    IF textpattern IS NOT NULL THEN
      IF NOT root..Subsumed..Contains(textpattern) THEN
        RETURN FALSE;
      END IF;
    END IF;
    RETURN TRUE;
    END,
PUBLIC FUNCTION
RawText(tree StructuredTextTree)
    RETURNS CHARACTER VARYING(max_text_length)
BEGIN
    DECLARE new CHARACTER VARYING(max_text_length);
    SET new = CAST(tree..FullText AS
        CHARACTER VARYING(max_text_length));
    RETURN new;
END,

PUBLIC FUNCTION
RootLabel(tree StructuredTextTree)
    RETURNS CHARACTER VARYING(max_text_length)
BEGIN
    RETURN CAST(tree..Nodelabel AS CHARACTER VARYING(max_text_length));
END,

PUBLIC FUNCTION
InTree(tree StructuredTextTree, nodeid NodeID)
    RETURNS Boolean
BEGIN
    DECLARE new Boolean;
    new = tree..Ids..HasMember(nodeid);
    RETURN new;
END,

PUBLIC FUNCTION
Ids(tree StructuredTextTree)
    RETURNS SetOfNodeIds
BEGIN
    DECLARE child StructuredTextTree;
    DECLARE nodeids SetOfNodeIds;
    SET nodeids = SetOfNodeIds(tree..NodeId);
    FOR child AS SELECT m FROM TABLE(tree..Children) T(m) DO
        SET nodeids = nodeids..Union(child..Ids);
    END FOR;
    RETURN nodeids;
END,

PUBLIC FUNCTION
Preorder(tree StructuredTextTree, nodeid NodeID)
    RETURNS INTEGER
BEGIN
    DECLARE IllegalNodeIdentifier EXCEPTION FOR SQLSTATE 'G2020';
    DECLARE new INTEGER;
    SET new = Preorder1(tree, nodeid);
    IF new IS NULL THEN
        SIGNAL IllegalNodeIdentifier;
    END IF;
    RETURN new;
END,
PRIVATE FUNCTION
Preorder1(tree StructuredTextTree, nodeid NodeID)
RETURNS INTEGER
BEGIN
DECLARE seen INTEGER;
DECLARE new INTEGER;
DECLARE child StructuredTextTree;
DECLARE order INTEGER;

SET new = 1;
IF nodeid = tree..NodeId THEN
RETURN new;
END IF;

FOR child,order AS SELECT m1,m2
FROM TABLE(tree..Children) WITH ORDINALITY
T(m1,m2)
ORDER BY m2 DO
SET seen = child..Preorder1(nodeid);
IF seen IS NOT NULL THEN
new = new + seen;
RETURN new;
END IF;
SET new = new + child..Ids..Cardinality;
END FOR;
RETURN NULL;
END,

PUBLIC FUNCTION
Subtree(tree StructuredTextTree, nodeid NodeID)
RETURNS StructuredTextTree
BEGIN
DECLARE IllegalNodeIdentifier EXCEPTION FOR SQLSTATE 'G2020';
DECLARE new StructuredTextTree;
SET new = Subtree1(tree, nodeid);
IF new IS NULL THEN
SIGNAL IllegalNodeIdentifier;
END IF;
RETURN new;
END,

PRIVATE FUNCTION
Subtree1(tree StructuredTextTree, nodeid NodeID)
RETURNS StructuredTextTree
BEGIN
DECLARE child StructuredTextTree;
DECLARE new StructuredTextTree;
IF nodeid = tree..NodeId THEN
RETURN tree; --!!! N.B. Nodeid's do not change !!
END IF;

FOR child AS SELECT m from TABLE(tree..Children) T(m) DO
SET new = Subtree1(child, nodeid);
IF new IS NOT NULL THEN
RETURN new;
END IF;
END FOR;

PRIVATE FUNCTION
UniqueIdentifier()
   RETURNS NodeID
BEGIN
   -- See description
END

Description

StructuredTextTree

Instances of a StructuredTextTree are created by the parser invoked within the ParseText function. When such an instance is first created every node within it is assigned an object identifier distinct from the object identifiers of all other nodes within every StructuredTextTree.

RootMatches

The public function RootMatches hides the mechanics of how a string is matched against the values stored within a node of the StructuredTextTree.

RawText

Returns the raw text subsummed by a StructuredTextTree.

RootLabel

Returns the contents of the label string associated with the root node in the text tree.

InTree

Returns true if the StructuredTextTree contains a node with the specified nodeid. Otherwise returns false.

Ids

Returns as a SetOfNodeIds the node id of every node in the StructuredTextTree.

Preorder

Computes a key from an input node identifier. For any two keys key1 and key2 derived from nodeid1 and nodeid2 respectively (where nodeid1 and nodeid2 both reference nodes in the input StructuredTextTree), key1 will be less than key2 if and only if the node referenced by nodeid1 is visited earlier than that referenced by nodeid2 when performing a preorder traversal of the input StructuredTextTree.

Subtree

The public function Subtree returns the subtree within a StructuredTextTree rooted at the specified node identifier. Within this subtree node identifiers remain unchanged.

UniqueIdentifier
The private function *UniqueIdentifier* returns a unique node identifier. This node identifier is used to
distinguish between distinct nodes within a single *StructuredTextTree* instance, and to determine (when
performing publically defined set operations on *StructuredText marks*) if two nodes (potentially occurring
in distinct *StructuredTextTree* instances) have the same provenance; that is that they were created by the
same invocation of *StructuredTextTree()*, represent the same node within a common conceptual instance of
*StructuredTextTree*, and thus have identical descendants.
7.5 SetOfNodeIds Abstract Data Type

Function

This type provides a mechanism for describing a set of identified nodes in a structured text tree, and for defining valid operations on this set.

Definition

CREATE VALUE TYPE SetOfNodeIds (  
PRIVATE NodeIds SET(NodeID),  
PUBLIC FUNCTION  
  SetOfNodeIds()  
  RETURNS SetOfNodeIds  
  BEGIN  
    DECLARE new SetOfNodeIds;  
    SET     new          = SetOfNodeIds();  
    SET     new..NodeIds = CAST(EMPTY AS SET(NodeID));  
    RETURN  new;  
  END,  
PUBLIC FUNCTION  
  SetOfNodeIds(nodeid NodeID)  
  RETURNS SetOfNodeIds  
  BEGIN  
    DECLARE new SetOfNodeIds;  
    SET     new          = SetOfNodeIds();  
    SET     new..NodeIds = new..NodeIds S_UNION SET{nodeid};  
    RETURN  new;  
  END,  
PUBLIC FUNCTION  
  Union(set1 SetOfNodeIds, nodeid NodeID)  
  RETURNS SetOfNodeIds  
  BEGIN  
    DECLARE new SetOfNodeIds;  
    SET     new          = set1;  
    SET     new..NodeIds = new..NodeIds S_UNION SET{nodeid};  
    RETURN  new;  
  END,  
PUBLIC FUNCTION  
  Union(set1 SetOfNodeIds, set2 SetOfNodeIds)  
  RETURNS SetOfNodeIds  
  BEGIN  
    DECLARE new SetOfNodeIds;  
    SET     new          = set1;  
    SET     new..NodeIds = set1..NodeIds S_UNION set2..NodeIds;  
    RETURN  new;  
  END,  
PUBLIC FUNCTION  
  Intersect(set1 SetOfNodeIds, set2 SetOfNodeIds)  
  RETURNS SetOfNodeIds
BEGIN
DECLARE new SetOfNodeIds;

SET new = set1;
SET new..NodeIds = set1..NodeIds S_INTERSECT set2..NodeIds;
RETURN new;

END,

PUBLIC FUNCTION
Except(set1 SetOfNodeIds, nodeid NodeID)
RETURNS SetOfNodeIds
BEGIN
DECLARE new SetOfNodeIds;

SET new = set1;
SET new..NodeIds = set1..NodeIds S_EXCEPT SET{nodeid};
RETURN new;

END,

PUBLIC FUNCTION
Except(set1 SetOfNodeIds, set2 SetOfNodeIds)
RETURNS SetOfNodeIds
BEGIN
DECLARE new SetOfNodeIds;

SET new = set1;
SET new..NodeIds = set1..NodeIds S_EXCEPT set2..NodeIds;
RETURN new;

END,

PUBLIC FUNCTION
Members(set1 SetOfNodeIds)
RETURNS SET(NodeID)
BEGIN
RETURN set1..NodeIds;
END,

PUBLIC FUNCTION
Cardinality(set1 SetOfNodeIds)
RETURNS INTEGER
BEGIN
RETURN CARDINALITY(set1..NodeIds);
END,

PUBLIC FUNCTION
HasMember(set1 SetOfNodeIds, nodeid NodeID)
RETURNS Boolean
BEGIN
DECLARE found INTEGER;

SET found = (SELECT count(*)
FROM Table(set1..NodeIds) T(member)
WHERE member = nodeid) = 0);

IF found = 0 THEN
RETURN FALSE;
ELSE
RETURN TRUE;
END IF;
END}
**Description**

**SetOfNodeIds**

`SetOfNodeIds` encapsulates a set of zero or more node identifiers identifying nodes within a structured text tree. When presented with an input node identifier or set of node identifiers returns the encapsulated set.

**Union**

The public function `Union` performs set union on two sets of nodeids.

**Intersect**

The public function `Intersect` performs set intersection on two sets of nodeids.

**Except**

The public function `Except` removes every nodeid in set1 that is also a member of set2.

**Members**

The public function `Members` allows access to the set of known nodeids.

**Cardinality**

The public function `Cardinality` returns the number of nodeids in a set of nodeids.

**HasMember**

The public function `HasMember` returns true if and only if the specified node identifier is a member of the indicated set of nodes.
7.6 SetOfListsOfNodeIds Abstract Data Type

Function

This type provides a mechanism for describing and manipulating a potentially absent set of fixed size lists of node identifiers used by the StructuredText and StructuredTextSearch Abstract Data Types to represent dynamically created relations whose degree and cardinality even when present may be zero.

Definition

CREATE VALUE TYPE SetOfListsOfNodeIds (  
  PRIVATE SetExists      Boolean,  
  DegreeOfLists  INTEGER,  
  ListsOfNodeIds  SET(List(NodeID)),  
  PUBLIC FUNCTION  
  SetOfListsOfNodeIds(present Boolean)  
    RETURNS SetOfListsOfNodeIds  
    BEGIN  
      DECLARE new SetOfListsOfNodeIds;  
      SET     new                 = SetOfListsOfNodeIds();  
      SET     new..SetExists      = present;  
      SET     new..DegreeOfLists  = 0;  
      SET     new..ListsOfNodeIds = CAST(EMPTY AS SET(LIST(NodeID)));  
      RETURN  new;  
    END,  
  PUBLIC FUNCTION  
  SetOfListsOfNodeIds(nodeid NodeID)  
    RETURNS SET(LIST(NodeID))  
    BEGIN  
      DECLARE new SetOfNodeIds;  
      SET     new                 = SetOfListsOfNodeIds();  
      SET     new..SetExists      = TRUE;  
      SET     new..DegreeOfLists  = 1;  
      SET     new..ListsOfNodeIds = SET{LIST{nodeid}};  
      RETURN  new;  
    END,  
  PUBLIC FUNCTION  
  Present(set1 SetOfListsOfNodeIds)  
    RETURNS Boolean  
    BEGIN  
      RETURN set1..SetExists;  
    END,  
  PUBLIC FUNCTION  
  Members(set1 SetOfListsOfNodeIds)  
    RETURNS SET(LIST(NodeID))  
    BEGIN  
      RETURN set1..ListsOfNodeIds;  
    END,  
  PUBLIC FUNCTION  
  Cardinality(set1 SetOfListsOfNodeIds)  
    RETURNS INTEGER  
    BEGIN  
      RETURN CARDINALITY(set1..ListsOfNodeIds);  
    END,  
)
PUBLIC FUNCTION
  Degree(set1 SetOfListsOfNodes)
  RETURNS INTEGER
  BEGIN
    RETURN set1..DegreeOfLists;
  END,

PUBLIC FUNCTION
  Union(set1 SetOfListsOfNodeIds, set2 SetOfListsOfNodeIds)
  RETURNS SetOfListsOfNodeIds;
  BEGIN
    DECLARE IllegalListDegree EXCEPTION FOR SQLSTATE 'G2021';
    DECLARE new SetOfListsOfNodeIds;
    new = set1;
    IF new..SetExists = FALSE THEN
      new..SetExists = TRUE;
      new..DegreeOfLists = set2..DegreeOfLists;
    ELSE
      IF set1..DegreeOfLists <> set2..DegreeOfLists THEN
        SIGNAL IllegalListDegree;
      END IF;
    END IF;
    new..ListsOfNodeIds = new..ListsOfNodeIds S_UNION
      set2..ListsOfNodeIds;
    RETURN new;
  END,

PUBLIC FUNCTION
  CrossProduct(relation1 SetOfListsOfNodeIds,
                relation2 SetOfListsOfNodeIds)
  RETURNS SetOfListsOfNodeIds
  BEGIN
    DECLARE tuple LIST(NodeID);
    DECLARE tuples SET(LIST(NodeID));
    DECLARE new SetOfListsOfNodeIds;
    IF relation1..DegreeOfLists = 0 THEN
      RETURN relation2;
    END IF;
    IF relation2..DegreeOfLists = 0 THEN
      RETURN relation1;
    END IF;
    SET new = SetOfListsOfNodeIds();
    Set new..SetExists = TRUE;
    SET new..DegreeOfLists = relation1..DegreeOfLists +
      relation2..DegreeOfLists;
    SET tuples = CAST(EMPTY AS SET(LIST(NodeID)));
    FOR tuple AS SELECT CONCATENATE(m1,m2) -- !! Append the lists !!
      FROM TABLE(relation1..ListsOfNodeIds) T1(m1),
      TABLE(relation2..ListsOfNodeIds) T2(m2) DO
      SET tuples = tuples S_UNION SET{tuple};
    END FOR;
    SET new..ListOfNodeIds = tuples;
    RETURN new;
  END)
Description

SetOfListsOfNodeIds

SetOfListsOfNodeIds encapsulates a set of zero or more pseudo rows of nodeids represented using lists. Each pseudo row has the same degree which may be zero.

Present

Returns true if this instance of SetOfListsOfNodeIds contains a known value else false.

Members

The public function Members returns and allows access to a set of known lists of nodeids.

Cardinality

The public function Cardinality returns the number of lists of nodeids in a set of lists of nodeids.

Degree

The public function Degree returns the common degree of all pseudo rows in the set.

Union

The public function Union returns the union of two sets of pseudo rows.

CrossProduct

The public function CrossProduct returns the cross product of two sets of fixed length lists each list having degree zero or higher.
8. Status Codes

The character string value returned in a SQLSTATE parameter comprises a 2-character class value followed by a 3-character subclass. Potential exception conditions raised by SQL/MM Full Text ADT’s are documented below. The category code “X” means that the class value given corresponds to an exception condition.

<table>
<thead>
<tr>
<th>Category</th>
<th>Condition</th>
<th>Class</th>
<th>Subcondition</th>
<th>Subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Unioning marks in potentially distinct texts</td>
<td>001</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Intersecting marks in potentially distinct texts</td>
<td>002</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Excepting marks in potentially distinct texts</td>
<td>003</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Illegal number of columns specified as parameter to ExtractSubtext</td>
<td>010</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Result returned by ExtractSubtext does not have specified number of columns</td>
<td>011</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Structured text node identifier not found within Structured Text</td>
<td>020</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Two fixed sized lists unexpectedly have differing cardinality</td>
<td>021</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Unable to parse input grammar of text</td>
<td>101</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Unable to parse input structured text</td>
<td>102</td>
</tr>
<tr>
<td>X</td>
<td>SQL/MM Structured Text</td>
<td>G2</td>
<td>Unable to parse structured text pattern</td>
<td>103</td>
</tr>
</tbody>
</table>
APPENDIX A

EXAMPLES

In the examples that follow bard_table and oed_table are tables having one row and one column, whose column name is bard and oed respectively. The bard table contains the Complete Works of Shakespeare as a single StructuredText. The oed table contains the entire text of the Oxford English Dictionary as a single StructuredText. In these examples we assume that structured text is internally encoded as described in references [3] and [4]. Tables are unnested as proposed in LHR-034.

Find the title of the works and speeches by Shakespeare which contain the text 'wherefore art' but are not in the play 'Romeo and Juliet'.

```sql
SELECT TextToString(title, 'clear'),
       TextToString(speech, 'clear')
FROM bard_table,
     ExtractSubtexts(bard, 3, '<work>{title}#,<speech>"wherefore art"#}') t(bard, title, speech)
WHERE NOT TextMatch(title, '^\"Romeo and Juliet\"')
```

Mark all speeches by Macbeth (but not Lady Macbeth) which contain 'bloody' and ('knife' or 'dagger') but not the word cut.

```sql
SELECT MarkExcept(
    MarkIntersect(
        MarkSubtexts(bard, '@\"bloody\"#'),
        MarkUnion(
            MarkSubtexts(bard, '@\"knife\"#'),
            MarkSubtexts(bard, '@\"dagger\"#'))
    ),
    MarkUnion(
        MarkSubtexts(bard, '@\"cut\"#'),
        MarkSubtexts(bard, '<speech>[<speaker>{\"Lady\"},@%#]')
    ))
FROM (SELECT MarkSubtexts(bard,
                          '<speech>[<speaker>{\"Macbeth\"},<said>#]') as bard
      FROM bard_table)
```

Find the title and 10th speech for each Shakespearian play

```sql
SELECT TextToString(title, 'clear'),
       TextToString(speech_10, 'clear')
FROM bard_table,
     ExtractSubtexts(bard, 3, '<work>{title}#&<kind>\"play\"}') t1(bard, play, title),
     IsolateSubtexts(HastenMarks(MarkSubtexts(play, '<speech>#'), 10, 1)) t2(play, speech_10)
```
Count the number of speeches by Romeo and by Juliet

```
SELECT CountMarks(MarkSubtexts(bard,'<speech>..<speaker>{{Romeo}}#'),
                    CountMarks(MarkSubtexts(bard,'<speech>..<speaker>{{Juliet}}#')))
FROM bard_table
```

Produce a single instance of `StructuredText` containing the Complete Works of Shakespeare in which all titles longer than 80 characters are marked.

```
SELECT AggregateMarks(t.bard)
FROM bard_table,
     ExtractSubtexts(bard, 2, '<title>#') t(bard, title)
WHERE LENGTH(TextToString(title,'clear')) > 80
```

Find words attributed to Shakespeare within the Oxford English Dictionary that contain ‘Spirit’. For each such word, find speeches in plays in the Complete Works of Shakespeare which contain that word.

```
SELECT TextToString(lookup, 'with markup'),
       TextToString(sense, 'with markup'),
       TextToString(mark_subtexts(work, '<title>#'), 'marked text'),
       TextToString(mark_subtexts(act, '<actno>#'), 'marked text'),
       TextToString(mark_subtexts(scene, '<sceneno>#'), 'marked text'),
       TextToString(speech, 'with markup')
FROM bard_table,
     (SELECT TextToString(lookup,'contains') as pattern,
          lookup,
          sense
     FROM oed_table,
          ExtractSubtexts(
          oed,
          3,
          '<entry>['
          '  <Headword>..<lookup form>{{Spirit}}#, '
          '  <Sense Level 6>#..<First Quot\.>..<Author>'
          '  {{Shaks.''}'
          ')'
          ) t(oed, lookup, sense)
     ),
      ExtractSubtexts(
      bard,
      5,
      '<WORK>#..<ACT>#..<SCENE>#..<SPEECH>#..<SAID>{{| | pattern || |}}'
      ) t1(bard, work, act, scene, speech)
```

Find the names of all speakers in a Shakespearian play who are not listed in the character section of any play. Report only those characters whose names include the case insensitive string 'ghost'.

```
SELECT speaker
FROM (SELECT TextToString(speaker, 'clear') as speaker
       FROM bard_table,
            ExtractSubtexts(bard, 2, '<speech>..<speaker>#')
       t(bard, speaker)
     ) EXCEPT
     (SELECT TextToString(character, 'clear') as speaker
      FROM bard_table,
           ExtractSubtexts(bard, 2, '<character>#')
      t(bard, character)
     )
GROUP BY speaker
HAVING lower(speaker) like '%ghost%'
```