International

Virtual

Observatory

Alliance

IVOA Astronomical Data Query Language

Version 2.0

IVOA Proposed Recommendation 30 April 2008

This version:
2.0-20080430
Latest version:
http://www.ivoa.net/Documents/latest/ADQL.html
Previous version(s):
none

Editor(s):
Pedro Osuna and Inaki Ortiz
Author(s):
Inaki Ortiz, Jeff Lusted, Pat Dowler, Alexander Szalay, Yuji Shirasaki, Maria A. Nieto-Santisteban, Masatoshi Ohishi, William O'Mullane, Pedro Osuna, the VOQL-TEG and the VOQL Working Group.

Abstract

This document describes the Astronomical Data Query Language (ADQL). ADQL has been developed based on SQL92. This document describes the subset of the SQL grammar supported by ADQL. Special restrictions and extensions to SQL92 have been defined in order to support generic and astronomy specific operations.

Status of This Document

This is a Proposed Recommendation. The first release of this document was 2006 January 22.

This is an IVOA Proposed Recommendation made available for public review. Comments on this document - for consideration in the next version of this document - should be sent to vogl@ivoa.net, a mailing list with a public archive. It is appropriate to reference this document only as a recommended standard that is under review and which may be changed before it is accepted as a full recommendation.

A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/Documents/.

Contents

1 Introduction

The Astronomical Data Query Language (ADQL) is the language used by the International Virtual Observatory Alliance (IVOA) to represent astronomy queries posted to VO services. The IVOA has developed several standardized protocols to access astronomical data, e.g., SIAP and SSAP for image and spectral data respectively. These protocols might be satisfied using a single table query. However, different VO services have different needs in terms of query complexity and ADQL arises in this context.

The ADQL specification pretends to avoid any distinction between core and advanced or extended functionalities. Hence ADQL has been built according to a single language definition (BNF based [1]). Any service making use of ADQL would then define the level of compliancy to the language. This would allow the notion of core and extension to be service-driven and it would decouple the language from the service specifications.

ADQL is based on the Structured Query Language (SQL), especially on SQL 92. The VO has a number of tabular data sets and many of them are stored in relational databases, making SQL a convenient access means. A subset of the SQL grammar has been extended to support queries which are specific to astronomy.

The exact meaning of keywords indicating requirement levels can be found in the References section [2].

2 Astronomical Data Query Language (ADQL)

This section describes the ADQL language specification. We will define in subsequent sections the syntax for the special characters, reserved and non-reserved words, identifiers and literals and then, finally, the syntax for the query expression.

The formal notation for syntax of computing languages is often expressed in the “Backus Naur Form” BNF. This syntax is used by popular tools for producing parsers. Appendix A to this document provides the full BNF grammar for ADQL.

The following conventions are used through this document:

- Optional items are enclosed in meta symbols [and]
- A group of items is enclosed in meta symbols { and }
- Repetitive item (zero or more times) are followed by …
- Terminal symbols are enclosed by < and >
- Terminals of meta-symbol characters (=,[,],(,),<,>,*) are surrounded by quotes (“) to distinguish them from meta-symbols
- Case insensitiveness otherwise stated.

2.1 Characters, Keywords, Identifiers and Literals

2.1.1 Characters

The language allows simple Latin letters (lower and upper case, i.e. {aA-zZ}), digits ({0-9}) and the following special characters:

- space
- single quote (’)
- double quote (“)
- percent (%)
- left and right parenthesis
- asterisk (*)
- plus sign (+)
- minus sign (-)
- comma (,)
- period (.)
- solidus (/)
- colon (:)
- semicolon (;)
- less than operator (<)
- equals operator (=)
- greater than operator (>)
- underscore (_)
- ampersand (&)
- question mark (?)
- vertical bar (|)

2.1.2 Keywords and Identifiers

Besides the character set, the language provides a list of reserved keywords plus the syntax description for regular identifiers.

A reserved keyword has a special meaning in ADQL and cannot be used as an identifier. These keywords must be enforced and should be extensive as an escaping mechanism is already in place. We can extend the list of SQL92 reserved keywords to accommodate those useful for astronomical purposes and/or present in a subset of vendor specific languages only (e.g. TOP). This leads to the following list:

- SQL reserved keywords:

ABSOLUTE, ACTION, ADD, ALL, ALLOCATE, ALTER, AND, ANY, ARE, AS, ASC, ASSERTION, AT, AUTHORIZATION, AVG, BEGIN, BETWEEN, BIT, BIT_LENGTH, BOTH, BY, CASCADE, CASCADED, CASE, CAST, CATALOG, CHAR, CHARACTER, CHARACTER_LENGTH, CHAR_LENGTH, CHECK, CLOSE, COALESCE, COLLATE, COLLATION, COLUMN, COMMIT,

CONNECT, CONNECTION, CONSTRAINT, CONSTRAINTS, CONTINUE, CONVERT, CORRESPONDING, COUNT, CREATE, CROSS, CURRENT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, CURSOR, DATE, DAY, DEALLOCATE, DECIMAL, DECLARE, DEFAULT, DEFERRABLE, DEFERRED, DELETE, DESC, DESCRIBE, DESCRIPTOR, DIAGNOSTICS, DISCONNECT, DISTINCT, DOMAIN, DOUBLE, DROP, ELSE, END, END-EXEC, ESCAPE, EXCEPT, EXCEPTION, EXEC, EXECUTE, EXISTS, EXTERNAL, EXTRACT, FALSE, FETCH, FIRST, FLOAT, FOR, FOREIGN, FOUND, FROM, FULL, GET, GLOBAL, GO, GOTO, GRANT, GROUP, HAVING, HOUR, IDENTITY, IMMEDIATE, IN, INDICATOR, INITIALLY, INNER, INPUT, INSENSITIVE, INSERT, INT, INTEGER, INTERSECT, INTERVAL, INTO, IS, ISOLATION, JOIN, KEY, LANGUAGE, LAST, LEADING, LEFT, LEVEL, LIKE, LOCAL, LOWER, MATCH, MAX, MIN, MINUTE, MODULE, MONTH, NAMES, NATIONAL, NATURAL, NCHAR, NEXT, NO, NOT, NULL, NULLIF, NUMERIC, OCTET_LENGTH, OF, ON, ONLY, OPEN, OPTION, OR, ORDER, OUTER, OUTPUT, OVERLAPS, PAD, PARTIAL, POSITION, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC, READ, REAL, REFERENCES, RELATIVE, RESTRICT, REVOKE, RIGHT, ROLLBACK, ROWS, SCHEMA, SCROLL, SECOND, SECTION, SELECT, SESSION, SESSION_USER, SET, SIZE, SMALLINT, SOME, SPACE, SQL, SQLCODE, SQLERROR, SQLSTATE, SUBSTRING, SUM, SYSTEM_USER, TABLE, TEMPORARY, THEN, TIME, TIMESTAMP, TIMEZONE_HOUR, TIMEZONE_MINUTE, TO, TRAILING, TRANSACTION, TRANSLATE, TRANSLATION, TRIM, TRUE, UNION, UNIQUE, UNKNOWN, UPDATE, UPPER, USAGE, USER, USING, VALUE, VALUES, VARCHAR, VARYING, VIEW, WHEN, WHENEVER, WHERE, WITH, WORK, WRITE, YEAR, ZONE

- ADQL reserved keywords:

ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, DEGREES, EXP, FLOOR, LOG, LOG10, MODE, PI, POWER, RADIANS, RAND, ROUND, SIN, SQRT, SQUARE, TAN, TOP, TRUNCATE

The identifiers are used to express, for example, a table or a column reference name.

Both the identifiers and the keywords are case insensitive. They SHALL begin with a letter {aA-zZ}. Subsequent characters shall be letters, underscores or digits {0-9} as follows:

<Latin_letter> [{ <underscore> | {<Latin_letter> | <digit>} }]

For practical purposes the language specification should be able to address reserved keyword and special character conflicts. To do so the language

provides a way to escape a non-compliant identifier by using the double quote character as a delimiter.

ADQL allows making use of the same quoting mechanism to handle the case sensitiveness if needed.

2.1.3 Literals

Finally we define the syntax rules for the different data types: string and number.

A string literal is a character expression delimited by single quotes.

Literal numbers are expressed in BNF as follows:

```
<unsigned_numeric_literal> ::=
        <exact_numeric_literal> | <approximate_numeric_literal>
<exact_numeric_literal> ::=
        <unsigned_integer> [<period> [<unsigned_integer>]]
        | <period><unsigned_integer>
<unsigned_integer> ::= <digit>…
<approximate_numeric_literal> ::= <mantissa> E <exponent>
<mantissa> ::= <exact_numeric_literal>
<exponent> ::= <signed_integer>
<signed_integer> ::= [<sign>] <unsigned_integer>
<sign> ::= <plus_sign> | <minus_sign>
```

2.2 Query syntax

A full and complete syntax of the select statement can be found in “Appendix A: BNF Grammar” at the <query_specification> construct. Follows a simplified syntax for the SELECT statement showing the main constructs for the query specification:

```
SELECT [ ALL |  DISTINCT ]
        [ TOP unsigned_integer ]
        { * | { value_expression [ [AS] column_name ] }, …  }
        FROM {
          { table_name [ [AS] identifier ]      |
            ( SELECT ….) [ [AS] identifier ]  |
            table_name [NATURAL] [ INNER | { LEFT | RIGHT | FULL
              [OUTER] } ] JOIN table_name
              [ON  search_condition | USING ( column_name,…) ] }
        , …}
        [ WHERE search_condition ]
        [ GROUP BY column_name, … ]
        [ HAVING search_condition  ]
```

[ORDER BY { *column_name* | *unsigned_integer* } [ASC | DESC],…]

The SELECT statement defines a query to some derived table(s) specified in the FROM clause. As a result of this query, a subset of the table(s) is returned. The order of the rows MAY be arbitrary unless ORDER BY clause is specified. The order of the columns to return SHALL be the same as the order specified in the selection list, or the order defined in the original table if asterisk is specified.

TOP n construct is used to return the first n-rows.

The selection list MAY include any numeric, string or geometry value expression,.

In the following sections some constructs requiring further description are presented.

2.2.1 Table subqueries and Joins

Table subqueries are present and can be used by some existing predicates within the search condition (IN and BETWEEN most likely) or as an artifact of building derived tables..

Among the different types of joins ADQL supports qualified ones only. These are INNER and OUTER ones (LEFT, RIGHT and FULL). All of these can be NATURAL or not. The join condition does not support embedded sub joins.

2.2.2 Search condition

The search condition can be part of several other clauses: JOIN, HAVING and, obviously, WHERE. Standard logical operators are present in its description (AND, OR and NOT). Five different types of predicates are present in which different types of reserved keywords or characters are used:

- Standard comparison operators: =, !=, <>, <, >, <=, >=
- BETWEEN
- LIKE
- NULL
- EXISTS

2.3 Functions

ADQL declares a list of non reserved keywords (section 2.1.2) which defines a set of special functions to enhance the astronomical usage of the language. These can be split into three different types:

2.3.1 Mathematical Functions

The next table shows the description of the mathematical functions.

Name	Return type	Comment
acos(x)	double	Inverse cosine.
asin(x)	double	Inverse sine.
atan(x)	double	Inverse tangent.
atan2(x,y)	double	Inverse tangent of x/y.
cos(x)	double	Cosine.
sin(x)	double	Sine.
tan(x)	double	Tangent.
abs(x)	double	Absolute value.
ceiling(x)	double	Smallest integer not less than argument.
degrees(x)	double	Radians to degrees.
exp(x)	double	Exponential.
floor(x)	double	Larger integer not greater than argument.
log(x)	double	Natural logarithm.
log10(x)	double	Base 10 logarithm.
mod(x, y)	double	Remainder of y/x.
pi()	double	Pi constant.
power(x, y)	double	X raised to the power of Y.
radians(x)	double	Degree to radians.
sqrt(x)	double	Square root.
rand(x)	double	Random value between 0.0 and 1.0. It can take a seed value.
round(x, n)	double	Round to nearest integer.
truncate(x, n)	double	Truncate to n decimal places.

2.3.2 Region

2.3.2.1 Additional ADQL Reserved words

The region specification introduces new reserved words for geometry functions:

AREA, CENTROID, CIRCLE, CONTAINS, DISTANCE, INTERSECTS, LATITUDE, LONGITUDE, POINT, POLYGON, RECTANGLE, REGION

2.3.2.2 Region Functions

The functions cover four general topics: Data Types, Predicates, Utility Calculations, and Manipulation. Each of these are covered below.

2.3.2.2.1 Data Type Functions

Certain functions represent geometry data types. These data types are Point, Circle, Polygon and Rectangle together with a generalized Region data type. The functions are similarly named and return a variable-length binary value.

Geometry data types are centred around the bnf construct <value_expression> which is central to data types within SQL.

<value_expression> ::=
    <numeric_value_expression>
  | <string_value_expression>
  | <geometry_value_expression>

A <geometry_value_expression> does not simply cover data type functions (POINT, CIRCLE etc) but must also allow
 (i) for column values where a geometry data type is stored in a column; and
 (ii) for the manipulation of one or more geometry data types (CENTROID at present, see below under Manipulative Functions).

So <geometry_value_expression> is expanded as

 <geometry_value_expression> ::= <geometry_expression> | <geometry_value> | <centroid>

, where

 <geometry_expression> ::= <point> | <circle> | <rectangle> | <polygon> | <region>

 and

 <geometry_value> ::= <column_reference>

2.3.2.2.2 Predicate Functions

Functions CONTAINS and INTERSECTS each accept two geometry data types and return 1 or 0 according to whether the relevant verb (eg: "contains") is satisfied against the two input geometries; 1 represents true and 0 represents false. Each of these functions can be assembled into a predicate:

SELECT * FROM PhotoObj as p WHERE CONTAINS(POINT(...), CIRCLE(...)) = 1

, where the ... would represent the constituent parts of a circle and point geometry.

One would expect later additions to ADQL to add to this range of functions. For example: equals, disjoint, touches, crosses, within, overlaps, and relate are possibilities.

2.3.2.2.3 Utility Calculations

Functions AREA, DISTANCE, LONGITUDE and LATITUDE accept a geometry (or two geometries in the case of DISTANCE) and return a calculated numeric value.

Predicate and Utility Calculation functions have been included as <numeric_value_functions> because they return simple numeric values. Thus

```
<numeric_value_function> ::=
    <trig_function>
  | <math_function>
  | <user_defined_function>
  | <system_defined_function>
```

, where

```
<system_defined_function> ::=
    <distance_function>
  | <region_function>
  | <longitude>
  | <latitude>
  | <area>
```

and

<region_function> ::= <contains_function> | <intersects_function>

2.3.2.2.4 Manipulative Functions

There is only one at present. This is CENTROID, which accepts a geometry as an input parameter and returns the geometry's centroid. This is manipulative from the viewpoint that it accepts a geometry and returns a geometry.

We would expect this to be the first of a number of manipulative functions to be added to ADQL. CENTROID is associated with geometry data types because of returning a geometry, and so is included as a <geometry_value_expression> (See section 2.3.2.2.1 above).

2.3.2.2.5 Over-view of Each Region Function in alphabetical order

2.3.2.2.5.1 AREA

Computes the area of a given geometry, for example:

```
AREA( CIRCLE(...) )
```

2.3.2.2.5.2 CENTROID

Computes the centroid of a given geometry, for example:

```
CENTROID( CIRCLE(...) )
```

and returns a Point.

2.3.2.2.5.3 CIRCLE

Expresses a circular region on the sky (a cone in space). The arguments specify the coordinate system, the longitude and latitude of the centre, and the radius, for example:

```
CIRCLE('ICRS', 1, 2, 3)
```

, where numeric values are in degrees.

2.3.2.2.5.4 CONTAINS

A numeric function that determines if one geometry is wholly contained within another. This is most commonly used to express the "point-in-shape" condition, for example:

```
CONTAINS( POINT(...), POLYGON(...) )
```

, where the contains function returns 1 (true) if the first argument is in or on the boundary of the polygon and 0 (false) otherwise. Thus, contains is not symmetric in the meaning of the arguments. When used in the where clause of a query, the value must be compared to 0 or 1 to form an SQL predicate:

```
CONTAINS(...) = 1
```

for "does contain" and

CONTAINS(...) = 0

for "does not contain".

The arguments to the contains function can be (literal) values created from the geometry types or they can be single column names or aliases (for geometry stored in a database table). Since the two argument geometries may be expressed in different coordinate systems, the function is responsible for converting one (or both). If it cannot do so, it should return NULL.

2.3.2.2.5.5 DISTANCE

Computes the arc length along a great circle between two points, for example:

DISTANCE(POINT(...), POINT(...))

, where all numeric values and the returned arc-length are in degrees.

Since the two argument points may be expressed in different coordinate systems, the function is responsible for converting one (or both). If it cannot do so, it should return NULL.

2.3.2.2.5.6 INTERSECTS

A numeric function that determines if two geometry values overlap. This is most commonly used to express a "shape-vs-shape" intersection test, for example:

INTERSECTS(CIRCLE(...), POLYGON(...))

, where the intersects function returns 1 (true) if the two arguments overlap and 0 (false) otherwise. When used in the where clause of a query, the value must be compared to 0 or 1 to form an SQL predicate:

INTERSECTS(...) = 1

for "does intersect" and

INTERSECTS(...) = 0

for "does not intersect".

The arguments to the intersects function can be (literal) values created from the geometry types or they can be single column names or aliases (for geometry

stored in a database table). Note that if one of the arguments is a point, intersects is equivalent to contains (with the point argument first). Unlike contains, the intersect function is symmetric in its arguments, e.g. INTERSECTS(a, b) is equivalent to INTERSECTS(b, a). Since the two argument points may be expressed in different coordinate systems, the function is responsible for converting one (or both). If it cannot do so, it should return NULL.

2.3.2.2.5.7 LONGITUDE

Extracts the longitude of a given point. For example:

```
LONGITUDE( POINT( ... ) )
```

2.3.2.2.5.8 LATITUDE

Extracts the latitude of a given point. For example:

```
LATITUDE( POINT( ... ) )
```

2.3.2.2.5.9 POINT

Expresses a single location on the sky. The arguments specify the coordinate system, longitude, and latitude, for example:

```
POINT('ICRS', 1, 2)
```

, where numeric values are in degrees.

2.3.2.2.5.10 POLYGON

Expresses a region on the sky with sides denoted by great circles passing through specified coordinates. The arguments specify the coordinate system and three or more sets of coordinates (longitude,latitude pairs), for example the triangle:

```
POLYGON('ICRS', 1, 2, 4, 5, 3, 3)
```

, where all numeric values are in degrees. Thus, the polygon is a list of vertices in a single coordinate system, with each vertex connected to the next along a great circle and the last vertex implicitly connected to the first vertex.

2.3.2.2.5.11 RECTANGLE

Expresses a coordinate-system aligned box on the sky. The arguments specify the coordinate system and the longitude and latitude of two opposite corners of the box, for example:

RECTANGLE('ICRS', 1, 2, 4, 5)

, where all numeric values are in degrees. The two longitude values specify the range in longitude; the two latitude values specify the range in latitude. The longitude and latitude ranges may differ in magnitude and direction. Transforming a rectangle to another coordinate system will generally result in a polygon.

2.3.2.2.5.12 REGION

A generic way of expressing a region represented by a single string input parameter.

2.3.2.2.6 Geometry Output in the Select Clause

Geometry values (literals or columns containing geometry values) may be listed in the select clause, in which case they must be converted into a text form. This text form will be identical to the way a literal value would be specified in a query, including the geometry type (point, circle, rectangle, or polygon) and all arguments but excluding the required quotes around the coordinate system string. For example, the query

SELECT circle('ICRS', 1, 2, 0.5)

could return

CIRCLE(ICRS, 1.0, 2.0, 0.5)

or equivalent. The case of the coordinate system string should be preserved; the geometry type string is case insensitive. The output may alter the numeric format by converting whole numbers to floating point (as in the example above) but should not gartuitiously add digits. Otherwise, numeric output must conform to the rules for numeric expressions in the ADQL BNF.

2.3.3 User Defined Functions

ADQL provides a placeholder to define user specific functions. Such construct supports a variable list of parameters as input in the following way:

```
<user_defined_function> ::=
   <user_defined_function_name>
      <left_paren>
         [ <user_defined_function_param> [ { <comma>
            <user_defined_function_param> }... ] ]
      <right_paren>
```

The function names can be qualified with a prefix to ease parsing of the ADQL statement

```
<user_defined_function_name> ::=
    [ <default_function_prefix> ] <regular_identifier>
```

, while the function parameters are generic enough to support string, numeric and geometrical expressions

```
<user_defined_function_param> ::= <value_expression>
```

If metadata on a user defined function is available, this should be used. For example function names and cardinality of arguments should be checked against metadata where available.

Appendix A: BNF Grammar

An easier to navigate version of the BNF grammar can be found here

```
<ADQL_language_character> ::=
    <simple_Latin_letter>
  | <digit>
  | <SQL_special_character>

<ADQL_reserved_word> ::=
    ABS
  | ACOS
  | AREA
  | ASIN
  | ATAN
  | ATAN2
  | CEILING
  | CENTROID
  | CIRCLE
  | CONTAINS
  | COS
  | DEGREES
  | DISTANCE
  | EXP
  | FLOOR
  | INTERSECTS
  | LATITUDE
  | LOG
  | LOG10
  | LONGITUDE
  | MODE
  | PI
  | POINT
  | POLYGON
  | POWER
  | RADIANS
  | RECTANGLE
  | REGION
  | RAND
  | ROUND
  | SIN
  | SQUARE
  | SQRT
  | TOP
```

| TAN
| TRUNCATE

<SQL_embedded_language_character> ::=
<left_bracket>
| <right_bracket>

<SQL_reserved_word> ::=
ABSOLUTE | ACTION | ADD | ALL
| ALLOCATE | ALTER | AND
| ANY | ARE
| AS | ASC
| ASSERTION | AT
| AUTHORIZATION | AVG
| BEGIN | BETWEEN | BIT | BIT_LENGTH
| BOTH | BY
| CASCADE | CASCADED | CASE | CAST
| CATALOG
| CHAR | CHARACTER | CHAR_LENGTH
| CHARACTER_LENGTH | CHECK | CLOSE | COALESCE
| COLLATE | COLLATION
| COLUMN | COMMIT
| CONNECT
| CONNECTION | CONSTRAINT
| CONSTRAINTS | CONTINUE
| CONVERT | CORRESPONDING | COUNT | CREATE | CROSS
| CURRENT
| CURRENT_DATE | CURRENT_TIME
| CURRENT_TIMESTAMP | CURRENT_USER | CURSOR
| DATE | DAY | DEALLOCATE
| DECIMAL | DECLARE | DEFAULT | DEFERRABLE
| DEFERRED | DELETE | DESC | DESCRIBE | DESCRIPTOR
| DIAGNOSTICS
| DISCONNECT | DISTINCT | DOMAIN | DOUBLE | DROP
| ELSE | END | END-EXEC | ESCAPE
| EXCEPT | EXCEPTION
| EXEC | EXECUTE | EXISTS
| EXTERNAL | EXTRACT
| FALSE | FETCH | FIRST | FLOAT | FOR
| FOREIGN | FOUND | FROM | FULL
| GET | GLOBAL | GO | GOTO
| GRANT | GROUP
| HAVING | HOUR
| IDENTITY | IMMEDIATE | IN | INDICATOR
| INITIALLY | INNER | INPUT
| INSENSITIVE | INSERT | INT | INTEGER | INTERSECT

| INTERVAL | INTO | IS
| ISOLATION
| JOIN
| KEY
| LANGUAGE | LAST | LEADING | LEFT
| LEVEL | LIKE | LOCAL | LOWER
| MATCH | MAX | MIN | MINUTE | MODULE
| MONTH
| NAMES | NATIONAL | NATURAL | NCHAR | NEXT | NO
| NOT | NULL
| NULLIF | NUMERIC
| OCTET_LENGTH | OF
| ON | ONLY | OPEN | OPTION | OR
| ORDER | OUTER
| OUTPUT | OVERLAPS
| PAD | PARTIAL | POSITION | PRECISION | PREPARE
| PRESERVE | PRIMARY
| PRIOR | PRIVILEGES | PROCEDURE | PUBLIC
| READ | REAL | REFERENCES | RELATIVE | RESTRICT
| REVOKE | RIGHT
| ROLLBACK | ROWS
| SCHEMA | SCROLL | SECOND | SECTION
| SELECT
| SESSION | SESSION_USER | SET
| SIZE | SMALLINT | SOME | SPACE | SQL | SQLCODE
| SQLERROR | SQLSTATE
| SUBSTRING | SUM | SYSTEM_USER
| TABLE | TEMPORARY
| THEN | TIME | TIMESTAMP
| TIMEZONE_HOUR | TIMEZONE_MINUTE
| TO | TRAILING | TRANSACTION
| TRANSLATE | TRANSLATION | TRIM | TRUE
| UNION | UNIQUE | UNKNOWN | UPDATE | UPPER | USAGE
| USER | USING
| VALUE | VALUES | VARCHAR | VARYING | VIEW
| WHEN | WHENEVER | WHERE | WITH | WORK | WRITE
| YEAR
| ZONE

<SQL_special_character> ::=
  <space>
| <double_quote>
| <percent>
| <ampersand>
| <quote>
| <left_paren>

```
  | <right_paren>
  | <asterisk>
  | <plus_sign>
  | <comma>
  | <minus_sign>
  | <period>
  | <solidus>
  | <colon>
  | <semicolon>
  | <less_than_operator>
  | <equals_operator>
  | <greater_than_operator>
  | <question_mark>
  | <underscore>
  | <vertical_bar>

<ampersand> ::= &

 <approximate_numeric_literal> ::= <mantissa>E<exponent>

 <area> ::= AREA <left_paren> <geometry_value_expression> <right_paren>

 <as_clause> ::= [ AS ] <column_name>

<asterisk> ::= *

<between_predicate> ::=
    <value_expression> [ NOT ] BETWEEN
    <value_expression> AND <value_expression>

  <boolean_factor> ::= [ NOT ] <boolean_primary>

 <boolean_primary> ::=
    | <left_paren> <search_condition> <right_paren>
      <predicate>

 <boolean_term> ::=
     <boolean_factor>
   | <boolean_term> AND <boolean_factor>

<catalog_name> ::= <identifier>

 <centroid> ::= CENTROID <left_paren> <geometry_expression> <right_paren>

 <character_factor> ::= <character_primary>
```

```
 <character_primary> ::=
     <value_expression_primary>
   | <user_defined_function>

<character_representation> ::= <nonquote_character> | <quote_symbol>

 <character_string_literal> ::=
    <quote> [ <character_representation>... ] <quote>
    [ { <separator>... <quote> [ <character_representation>... ] <quote> }... ]

 <character_value_expression> ::= <concatenation> | <character_factor>

  <circle> ::=
     CIRCLE <left_paren> <coord_sys>
                <comma> <coordinates>
                <comma> <radius> <right_paren>

<colon> ::= :

<column_name> ::= <identifier>

<column_name_list> ::= <column_name> [ { <comma> <column_name> }... ]

 <column_reference> ::= [ <qualifier> <period> ] <column_name>

<comma> ::= ,

<comment> ::= <comment_introducer> [ <comment_character>... ] <newline>

<comment_character> ::= <nonquote_character> | <quote>

<comment_introducer> ::= <minus_sign><minus_sign> [<minus_sign>...]

 <comp_op> ::=
     <equals_operator>
   | <not_equals_operator>
   | <less_than_operator>
   | <greater_than_operator>
   | <less_than_or_equals_operator>
   | <greater_than_or_equals_operator>

 <comparison_predicate> ::=
     <value_expression> <comp_op> <value_expression>

  <concatenation> ::= <character_value_expression> <concatenation_operator>
<character_factor>
```

<concatenation_operator> ::= ||

<contains_function> ::= CONTAINS <left_paren>
<geometry_value_expression> <comma> <geometry_value_expression>
<right_paren>

<coord_lat> ::= <numeric_value_expression>

<coord_lon> ::= <numeric_value_expression>

<coord_sys> ::= <string_value_expression>

<coordinates> ::= <coord_lon> <comma> <coord_lat>

<correlation_name> ::= <identifier>

<correlation_specification> ::= [AS] <correlation_name>

<default_function_prefix> ::=

<delimited_identifier> ::= <double_quote> <delimited_identifier_body>
<double_quote>

<delimited_identifier_body> ::= <delimited_identifier_part>...

<delimited_identifier_part> ::= <nondoublequote_character> |
<double_quote_symbol>

<delimiter_token> ::=
    <character_string_literal>
  | <delimited_identifier>
  | <SQL_special_character>
  | <not_equals_operator>
  | <greater_than_or_equals_operator>
  | <less_than_or_equals_operator>
  | <concatenation_operator>
  | <double_period>
  | <left_bracket>
  | <right_bracket>

<derived_column> ::= <value_expression> [<as_clause>]

<derived_table> ::= <table_subquery>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<distance_function> ::=
DISTANCE <left_paren> <point> <comma> <point> <right_paren>

<double_period> ::= ..

<double_quote> ::= "

<double_quote_symbol> ::= <double_quote><double_quote>

<equals_operator> ::= =

<exact_numeric_literal> ::=
<unsigned_integer> [<period> [<unsigned_integer>]]
| <period> <unsigned_integer>

<exists_predicate> ::= EXISTS <table_subquery>

<exponent> ::= <signed_integer>

<factor> ::= [<sign>] <numeric_primary>

<from_clause> ::= FROM <table_reference>
[{ <comma> <table_reference> }...]

<general_literal> ::= <character_string_literal>

<general_set_function> ::=
<set_function_type> <left_paren> [<set_quantifier>] <value_expression>
<right_paren>

<geometry_expression> ::= <point> | <circle> | <rectangle> | <polygon> |
<region>

<geometry_value> ::= <column_reference>

<geometry_value_expression> ::= <geometry_expression> | <geometry_value>
| <centroid>

<greater_than_operator> ::= >

<greater_than_or_equals_operator> ::= >=

<group_by_clause> ::= GROUP BY <grouping_column_reference_list>

<grouping_column_reference> ::= <column_reference>

<grouping_column_reference_list> ::=
<grouping_column_reference> [{ <comma>
<grouping_column_reference> }...]

<having_clause> ::= HAVING <search_condition>

<identifier> ::= <regular_identifier> | <delimited_identifier>

<in_predicate> ::=
<value_expression> [NOT] IN <in_predicate_value>

<in_predicate_value> ::=
<table_subquery> | <left_paren> <in_value_list> <right_paren>

<in_value_list> ::=
<value_expression> { <comma> <value_expression> } ...

<intersects_function> ::= INTERSECTS <left_paren>
<geometry_value_expression> <comma> <geometry_value_expression>
<right_paren>

<join_column_list> ::= <column_name_list>

<join_condition> ::= ON <search_condition>

<join_specification> ::= <join_condition> | <named_columns_join>

<join_type> ::=
INNER
| <outer_join_type> [OUTER]

<joined_table> ::=
<qualified_join>
| <left_paren> <joined_table> <right_paren>

<keyword> ::= <SQL_reserved_word> | <ADQL_reserved_word>

<latitude> ::= LATITUDE <left_paren> <point> <right_paren>

<left_bracket> ::= [

<left_paren> ::= (

<less_than_operator> ::= <

<less_than_or_equals_operator> ::= <=

```
 <like_predicate> ::=
    <match_value> [ NOT ] LIKE <pattern>

 <longitude> ::= LONGITUDE <left_paren> <point> <right_paren>

<mantissa> ::= <exact_numeric_literal>

<match_value> ::= <character_value_expression>

 <math_function> ::=
    ABS <left_paren> <numeric_value_expression> <right_paren>
          | CEILING <left_paren> <numeric_value_expression> <right_paren>
          | DEGREES <left_paren> <numeric_value_expression> <right_paren>
          | EXP <left_paren> <numeric_value_expression> <right_paren>
          | FLOOR <left_paren> <numeric_value_expression> <right_paren>
          | LOG <left_paren> <numeric_value_expression> <right_paren>
          | PI <left_paren><right_paren>
          | POWER <left_paren> <numeric_value_expression> <comma>
             <unsigned_integer> <right_paren>
          | RADIANS <left_paren> <numeric_value_expression> <right_paren>
          | SQUARE <left_paren> <numeric_value_expression> <right_paren>
          | SQRT <left_paren> <numeric_value_expression> <right_paren>
          | LOG10 <left_paren> <numeric_value_expression> <right_paren>
          | RAND <left_paren> [ <unsigned_integer> ] <right_paren>
          | ROUND <left_paren> <numeric_value_expression>
             [ <comma> <signed_integer>] <right_paren>
          | TRUNCATE <left_paren> <numeric_value_expression>
             [ <comma> <signed_integer>] <right_paren>

<minus_sign> ::= -

 <named_columns_join> ::= USING <left_paren> <join_column_list>
<right_paren>

 <newline> ::=

 <nondelimiter_token> ::=
    <regular_identifier>
  | <keyword>
  | <unsigned_numeric_literal>

  <nondoublequote_character> ::=

 <nonquote_character> ::=
```

```
<not_equals_operator> ::= <not_equals_operator1> | <not_equals_operator2>

<not_equals_operator1> ::= <>

<not_equals_operator2> ::= !=

 <null_predicate> ::= <column_reference> IS [ NOT ] NULL

 <numeric_primary> ::=
     <value_expression_primary>
   | <numeric_value_function>

 <numeric_value_expression> ::=
     <term>
   | <numeric_value_expression> <plus_sign> <term>
   | <numeric_value_expression> <minus_sign> <term>

 <numeric_value_function> ::=
    <trig_function>
  | <math_function>
  | <user_defined_function>
  | <system_defined_function>

 <order_by_clause> ::= ORDER BY <sort_specification_list>

<ordering_specification> ::= ASC | DESC

<outer_join_type> ::= LEFT | RIGHT | FULL

 <pattern> ::= <character_value_expression>

<percent> ::= %

<period> ::= .

<plus_sign> ::= +

  <point> ::= POINT <left_paren> <coord_sys> <comma> <coordinates>
<right_paren>

  <polygon> ::=
     POLYGON <left_paren> <coord_sys>
                 <comma> <coordinates>
                 <comma> <coordinates>
               { <comma> <coordinates> } ?
             <right_paren>
```

```
<predicate> ::=
   <comparison_predicate>
 | <between_predicate>
 | <in_predicate>
 | <like_predicate>
 | <null_predicate>
 | <exists_predicate>

<qualified_join> ::=
    <table_reference> [ NATURAL ] [ <join_type> ] JOIN
    <table_reference> [ <join_specification> ]

<qualifier> ::= <table_name> | <correlation_name>

 <query_expression> ::=
     <query_specification>
   | <joined_table>

<query_specification> ::=
    SELECT [ <set_quantifier> ] [ <set_limit> ] <select_list> <table_expression>

<question_mark> ::= ?

<quote> ::= '

<quote_symbol> ::= <quote> <quote>

<radius> ::= <numeric_value_expression>

 <rectangle> ::=
    RECTANGLE <left_paren> <coord_sys>
                 <comma> <coordinates> <comma> <coordinates>
              <right_paren>

<region> ::= REGION <left_paren> <string_value_expression> <right_paren>

 <region_function> ::= <contains_function> | <intersects_function>

<regular_identifier> ::=
    <simple_Latin_letter>...
    [ { <digit> | <simple_Latin_letter> | <underscore> }... ]

<right_bracket> ::= ]

<right_paren> ::= )
```

<schema_name> ::= [<catalog_name> <period>] <unqualified_schema name>

<search_condition> ::=
<boolean_term>
| <search_condition> OR <boolean_term>

<select_list> ::=
<asterisk>
| <select_sublist> [{ <comma> <select_sublist> }...]

<select_sublist> ::= <derived_column> | <qualifier> <period> <asterisk>

<semicolon> ::= ;

<separator> ::= { <comment> | <space> | <newline> }...

<set_function_specification> ::=
COUNT <left_paren> <asterisk> <right_paren>
| <general_set_function>

<set_function_type> ::= AVG | MAX | MIN | SUM | COUNT

<set_limit> ::= TOP <unsigned_integer>

<set_quantifier> ::= DISTINCT | ALL

<sign> ::= <plus_sign> | <minus_sign>

<signed_integer> ::= [<sign>] <unsigned_integer>

<simple_Latin_letter> ::=
<simple_Latin_upper_case_letter>
| <simple_Latin_lower_case_letter>

<simple_Latin_lower_case_letter> ::=
a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
| p | q | r | s | t | u | v | w | x | y | z

<simple_Latin_upper_case_letter> ::=
A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
| P | Q | R | S | T | U | V | W | X | Y | Z

<solidus> ::= /

<sort_key> ::= <column_name> | <unsigned_integer>

<sort_specification> ::=
    <sort_key> [<ordering_specification>]

<sort_specification_list> ::=
    <sort_specification> [{ <comma> <sort_specification> }...]

<space> ::=

<string_value_expression> ::=
    <character_value_expression>

  <subquery> ::= <left_paren> <query_expression> <right_paren>

<system_defined_function> ::=
   <distance_function>
 | <region_function>
 | <longitude>
 | <latitude>
 | <area>

<table_expression> ::=
    <from_clause>
    [<where_clause>]
    [<group_by_clause>]
    [<having_clause>]
    [<order_by_clause>]

<table_name> ::= [<schema_name> <period>] <identifier>

<table_reference> ::=
   <table_name> [<correlation_specification>]
 | <derived_table> <correlation_specification>
 | <joined_table>

<table_subquery> ::= <subquery>

<term> ::=
    <factor>
  | <term> <asterisk> <factor>
  | <term> <solidus> <factor>

<token> ::=
    <nondelimiter_token>
  | <delimiter_token>

```
<trig_function> ::=
    ACOS <left_paren> <numeric_value_expression> <right_paren>
  | ASIN <left_paren> <numeric_value_expression> <right_paren>
  | ATAN <left_paren> <numeric_value_expression> <right_paren>
  | ATAN2 <left_paren> <numeric_value_expression> <comma>
<numeric_value_expression> <right_paren>
  | COS <left_paren> <numeric_value_expression> <right_paren>
  | COT <left_paren> <numeric_value_expression> <right_paren>
  | SIN <left_paren> <numeric_value_expression> <right_paren>
  | TAN <left_paren> <numeric_value_expression> <right_paren>

<underscore> ::= _

<unqualified_schema name> ::= <identifier>

<unsigned_integer> ::= <digit>...

<unsigned_literal> ::= <unsigned_numeric_literal> | <general_literal>

<unsigned_numeric_literal> ::=
    <exact_numeric_literal>
  | <approximate_numeric_literal>

<unsigned_value_specification> ::= <unsigned_literal>

<user_defined_function> ::=
    <user_defined_function_name>
        <left_paren>
            [ <user_defined_function_param> [ { <comma>
<user_defined_function_param> }... ] ]
        <right_paren>

<user_defined_function_name> ::=
    [ <default_function_prefix> ] <regular_identifier>

<user_defined_function_param> ::= <value_expression>

<value_expression> ::=
    <numeric_value_expression>
  | <string_value_expression>
  | <geometry_value_expression>

<value_expression_primary> ::=
    <unsigned_value_specification>
  | <column_reference>
```

| <set_function_specification>
| <left_paren> <value_expression> <right_paren>

<vertical_bar> ::= |

<where_clause> ::= WHERE <search_condition>

References

[1] BNF Grammar for ISO/IEC 9075:1992 – Database Language SQL (SQL-92)
http://savage.net.au/SQL/sql-92.bnf.html

[2] RFC-2119 Keywords to indicate Requirement Levels
http://rfc.net/rfc2119.html