International
Virtual
Observatory
Alliance

Astronomical Data Query Language

Version 2.1

IVOA Working Draft 2016-05-02

Working group
Data Access Layer Working Group

This version
http://www.ivoa.net/documents/ADQL/20160502

Latest version
http://www.ivoa.net/documents/ADQL

Previous versions
ADQL-2.0

Author(s)
The IVOA Virtual Observatory Query Language (VOQL) working group members, The IVOA Data Access Layer (DAL) working group members

Editor(s)
Dave Morris

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 an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

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

Contents

1 Introduction **5**
1.1 Role within the VO Architecture 6

2 Astronomical Data Query Language (ADQL) **7**
2.1 Characters, Keywords, Identifiers and Literals 8
2.1.1 Characters 8
2.1.2 Keywords and Identifiers 9
2.1.3 Literals 10
2.2 Query syntax 12
2.2.1 Table subqueries and Joins 13
2.2.2 Search condition 13
2.3 Mathematical and Trigonometrical Functions 14

3 ADQL Type System **16**

4 Optional components **18**
4.1 Service capabilities 18
4.2 Geometrical Functions 18
4.2.1 Overview 18
4.2.2 Data Type Functions 19
4.2.3 Predicate Functions 20
4.2.4 Utility Functions 21
4.2.5 AREA 23
4.2.6 BOX 23
4.2.7 CENTROID 24
4.2.8 CIRCLE 24
4.2.9 CONTAINS 25
4.2.10 COORD1 26
4.2.11 COORD2 27
4.2.12 COORDSYS 27

4.2.13 DISTANCE . . . 28
4.2.14 INTERSECTS . . . 29
4.2.15 POINT . . . 30
4.2.16 POLYGON . . . 31
4.2.17 REGION . . . 31
4.2.18 Geometry in the SELECT clause . . . 32
4.3 User Defined Functions . . . 32
4.3.1 Overview . . . 32
4.3.2 Metadata . . . 33
4.4 String functions and operators . . . 34
4.4.1 LOWER . . . 34
4.4.2 ILIKE . . . 34
4.5 Set operators . . . 35
4.5.1 UNION . . . 35
4.5.2 EXCEPT . . . 36
4.5.3 INTERSECT . . . 36
4.6 Common table expressions . . . 37
4.6.1 WITH . . . 37
4.7 Type operations . . . 38
4.7.1 CAST . . . 38
4.8 Unit operations . . . 39
4.8.1 IN_UNIT . . . 39
4.9 Bitwise operators . . . 39
4.9.1 Bit AND . . . 40
4.9.2 Bit OR . . . 40
4.9.3 Bit XOR . . . 40
4.9.4 Bit NOT . . . 41
4.10 Cardinality . . . 41
4.10.1 OFFSET . . . 41

A BNF Grammar **41**

B Language feature support **57**

C Changes from Previous Versions **58**
C.1 Changes from ADQL-2.0 . . . 58

Acknowledgments

The authors would like to acknowledge all contributors to this and previous versions of this standard, especially: P. Dowler, J. Lusted, M. A. Nieto-Santisteban, W. O'Mullane, M. Ohishi, I. Ortiz, P. Osuna, Y Shirasaki, and A. Szalay.

Conformance-related definitions

The words "MUST", "SHALL", "SHOULD", "MAY", "RECOMMENDED", and "OPTIONAL" (in upper or lower case) used in this document are to be interpreted as described in IETF standard, Bradner (1997).

The *Virtual Observatory (VO)* is general term for a collection of federated resources that can be used to conduct astronomical research, education, and outreach. The International Virtual Observatory Alliance (IVOA) is a global collaboration of separately funded projects to develop standards and infrastructure that enable VO applications.

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 makes no 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 that are specific to astronomy. Similarly to SQL, the ADQL language definition is not semantically safe by design and therefore this specification defines syntactical correctness only. Type safety has been achieved as far as it can be done in SQL. The exact meaning of key words indicating requirement levels can be found in the References section [2].

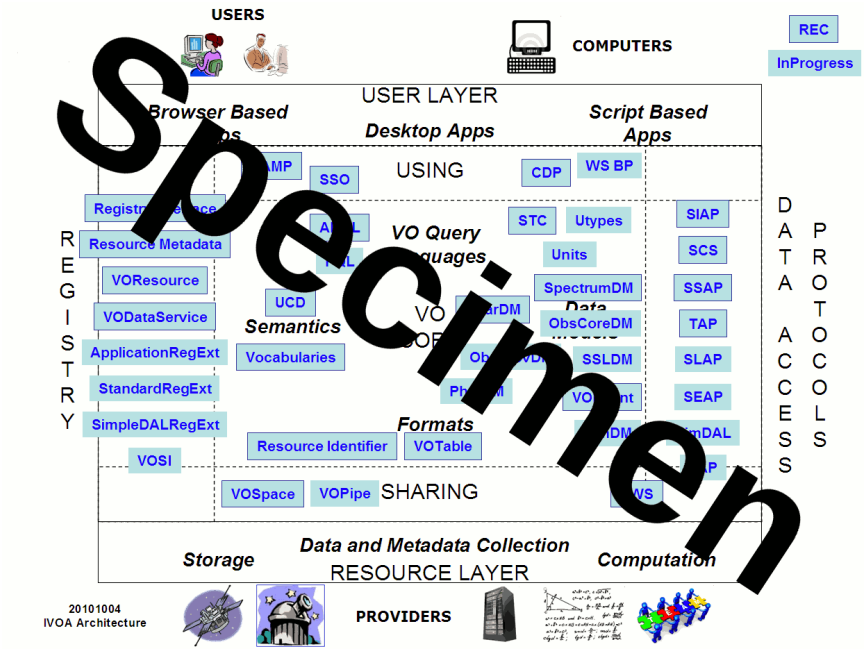

Figure 1: Architecture diagram for this document

1.1 Role within the VO Architecture

Fig. 1 shows the role this document plays within the IVOA architecture (Arviset et al., 2010).

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 unless 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 (`?`)
- circumflex (`^`)
- tilde (`~`)
- 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, MOD, PI, POWER, RADIANS, RAND, ROUND, SIN, SQRT, TAN, TOP, TRUNCATE

AREA, BOX, CENTROID, CIRCLE, CONTAINS, COORD1, COORD2, COORDSYS,
DISTANCE, INTERSECTS, POINT, POLYGON, REGION

ILIKE
```

```
IN_UNIT
```

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

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 case sensitiveness if needed.

2.1.3 Literals

Finally we define the syntax rules for the different data types: string, numeric and boolean.

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

```
<character_string_literal> ::=
    <quote> [ <character_representation>... ] <quote>
```

Literal numbers are expressed in BNF as follows:

```
<signed_numeric_literal> ::= [<sign>] <unsigned_numeric_literal>

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

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

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

<mantissa> ::= <exact_numeric_literal>

<exponent> ::= <signed_decimal>

<signed_decimal> ::= [<sign>] <unsigned_decimal>
```

```
<unsigned_decimal> ::= <digit>...

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

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

Hexadecimal literals are expressed using the 'C' style notation, e.g. `0xFF`, defined in BNF as follows :

```
<unsigned_hexadecimal> ::= 0x<hex_digit>...

hex_digit ::= <digit> | a | b | c | d | e | f | A | B | C | D | E | F
```

Hexadecimal literals are not case sensitive.

Hexadecimal literals can only be used to create integer data types, SMALLINT, INTEGER and BIGINT.

Boolean literals are expressed in BNF as follows:

```
<boolean_literal> ::= True | False
```

Boolean literals are not case sensitive.

Regarding the usage of other data types like datetime and timestamp, ADQL can deal with them similarly to how SQL does: using the string literal construct. As Relation Database Manager Systems (RDBMs) do, a service should be able to implicitly convert strings to internal (datetime or timestamp) form using a variety of techniques, where e.g. ISO 8601 is an acceptable format. Therefore, as with other string representations, it should be up to the service capability to understand such specific formats.

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_decimal ]
    {
         * |
         { 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_decimal } [ ASC | DESC],
        ...
        ]
    [ OFFSET unsigned_decimal ]
```

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 an ORDER BY clause is specified. A TOP clause MAY be specified to limit the number of rows returned. An OFFSET clause MAY be specified to skip a number of rows at the start of the results. If OFFSET is used in combination with a TOP clause then OFFSET is applied first, then the result set is limited by TOP (see 4.10.1).

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. 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 join, ADQL supports INNER and OUTER (LEFT, RIGHT and FULL) joins. If none is specified, the default is INNER. 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`

In addition, some service implementations may also support the optional ILIKE case-insensitive string comparison operator, defined in section 4.4.2.

- `ILIKE`

2.3 Mathematical and Trigonometrical Functions

ADQL declares a list of reserved keywords (section 2.1.2) which defines a set of mathematical and trigonometrical function names. Their syntax, usage and description are detailed in the following tables:

Name	**Argument data type**	**Return data type**	**Description**
abs(x)	double	double	Returns the absolute value of x.
ceiling(x)	double	double	Returns the smallest double value that is not less than the argument x and is equal to a mathematical integer.
degrees(x)	double	double	Converts an angle to degrees. Argument x must be in radians.
exp(x)	double	double	Returns Euler's number e raised to the power of x.
floor(x)	double	double	Returns the largest double value that is not greater than the argument x and is equal to a mathematical integer.
log(x)	double	double	Returns the natural logarithm (base e) of a double value. Value x must be greater than zero.
log10(x)	double	double	Returns the base 10 logarithm of a double value. Value x must be greater than zero.
mod(x, y)	double	double	Returns the remainder of y/x.
pi()	n/a	double	The π constant.
power(x, y)	x double y double	double	Returns the value of the first argument raised to the power of the second argument.
radians(x)	double	double	Converts an angle to radians. Argument x must be in degrees.
sqrt(x)	double	double	Returns the positive square root of a double value.
rand(x)	integer	double	Returns a random value between 0.0 and 1.0, where x is a seed value.
round(x, n)	x double n integer	double	Rounds double value x to n number of decimal places, with the default being to round to the nearest integer. To round to the left of the decimal point, a negative number should be provided.
truncate(x, n)	x double n integer	double	Returns the result of truncating the argument x to n decimal places.

Table 1: Mathematical functions

Name	**Argument data type**	**Return data type**	**Description**
acos(x)	double	double	Returns the arc cosine of an angle, in the range of 0 through π radians. Absolute value of x must be lower or equal than 1.0.
asin(x)	double	double	Returns the arc sine of an angle, in the range of -$\pi/2$ through $\pi/2$ radians. Absolute value of x must be and lower or equal than 1.0.
atan(x)	double	double	Returns the arc tangent of an angle, in the range of -$\pi/2$ through $\pi/2$ radians.
atan2(y,x)	double	double	Converts rectangular coordinates x,y to polar angle. It computes the arc tangent of y/x in the range of $-\pi$ through π radians.
cos(x)	double	double	Returns the cosine of an angle, in the range of -1.0 through 1.0. Argument x must be in radians.
sin(x)	double	double	Returns the sine of an angle, in the range of -1.0 through 1.0. Argument x must be in radians.
tan(x)	double	double	Returns the tangent of an angle. Argument x must be in radians.

Table 2: Trigonometrical functions

3 ADQL Type System

ADQL defines no data definition language (DDL). It is assumed that table definition and data ingestion are performed in the underlying database's native language and type system.

However, column metadata needs to give column types in order to allow the construction of queries that are both syntactically and semantically correct. Examples of such metadata includes VODataService's `TAPType` (VODataService-1.1, Plante et al. (2010)) or TAP's `TAP_SCHEMA` (TAP-1.0, Dowler et al. (2010)).

Services SHOULD, if at all possible, try express their column metadata in these terms even if the underlying database employs different types. Services SHOULD also use the following mapping when interfacing to user data, either by serializing result sets into VOTables or by ingesting user-provided VOTables into ADQL-visible tables. Where non-ADQL types are employed in the underlying database, implementors SHOULD make sure that all operations that are possible with the recommended ADQL type are also possible with the type used in the backend engine. For instance, the ADQL string concatenation operator `||` should be applicable to all columns resulting from VOTable char-typed columns.

VOTable			**ADQL**
datatype	**arraysize**	**xtype**	**type**
boolean	1	-	BOOLEAN
short	1	-	SMALLINT
int	1	-	INTEGER
long	1	-	BIGINT
float	1	-	REAL
double	1	-	DOUBLE
(numeric)	> 1	-	implementation defined
char	1	-	CHAR(1)
char	n	-	CHAR(n)
char	n*	-	VARCHAR(n)
unsignedByte	n	-	BINARY(n)
unsignedByte	n*	-	VARBINARY(n)
unsignedByte	n, *, n*	adql:BLOB	BLOB
char	n, *, n*	adql:CLOB	CLOB
char	n, *, n*	adql:TIMESTAMP	TIMESTAMP
char	n, *, n*	adql:POINT	POINT
char	n, *, n*	adql:REGION	REGION

Table 3: VOTable/ADQL type mapping

"Implementation defined" in the above table means that an implementation is free to reject attempts to (de-)serialize values in these types. They are to be considered unsupported by ADQL, and the language provides no means to manipulate *"native"* representations of them.

References to REGION-typed columns must be valid wherever the ADQL *region* nonterminal is allowed. References to POINT-typed columns must be valid wherever the ADQL *point* nonterminal is allowed.

Comparing the equality of a boolean value or expression with another boolean returns a boolean result.

When comparing the size of a boolean with another boolean, the value True is greater than the value False.

Unless explicitly stated, the result of any other operation on boolean values is undefined.

4 Optional components

In addition to the core components, the ADQL language also includes support for optional features and functions.

The following sections define the optional features that are part of the the ADQL language, but are not required in order to meet the standard for a basic ADQL service.

It is up to each service implementation to declare which optional or additional features it supports.

If a service does not declare support for an optional or additional feature, then a client SHOULD NOT assume that the service supports that feature, and SHOULD NOT make use of that feature in any ADQL queries that it sends.

4.1 Service capabilities

The TAPRegExt-1.0 standard (Demleitner et al., 2012) defines an XML schema that a service SHOULD use to declare which optional or additional features it supports.

In general, each group of langauge features is identified by a `type` URI, and each individual feature within the group is identified by the feature name.

Appendix B contains examples of how to declare support for each of the langauge features defined in this document using the TAPRegExt XML schema.

For full details on the XML schema and how it can be used, please refer to the TAPRegExt (Demleitner et al., 2012) standard.

4.2 Geometrical Functions

4.2.1 Overview

In addition to the mathematical functions, ADQL provides a set of geometrical functions to enhance the astronomical usage of the language.

- AREA
- BOX
- CENTROID
- CIRCLE
- CONTAINS
- COORD1

- COORD2
- COORDSYS
- DISTANCE
- INTERSECTS
- POINT
- POLYGON
- REGION

A special attention has to be paid to the REGION function. As can be seen more in detail in Section 2.4.14, this construct is a general purpose function and it takes a string value expression as argument. The format of the string is to be specified by a service that accepts ADQL by referring to a standard format. Currently STC/s (See [3] and [4]) is the only standardized string representation a service can declare.

As can also be seen in the following sections, all these functions have arguments being a geometrical, a string and/or a numerical value expression. When these values represent spherical coordinates the units MUST be in degrees (square degrees for area). If the cartesian coordinate system is used, the vector coordinates MUST be normalized.

Regarding the legal ranges, for spherical coordinates, these SHOULD be [0, 360] and [-90, 90]. In a cartesian coordinate system, there are no inherent limits but the already mentioned constraint that vectors should be normalized. It remains up to the service making use of ADQL to define the errors that should be raised when using values outside these ranges.

For historical reasons, the geometry constructors (BOX, CIRCLE, POINT, POLYGON) require a string-valued first argument. It was intended to carry information on a reference system or other coordinate system metadata. As of this version of the specification (2.1), this parameter has been marked as deprecated. Services are permitted to ignore this parameter and clients are advised to pass an empty string here. Future versions of this specification may remove this parameter from the listed functions.

Generally speaking, all these geometrical functions cover three different topics: data types, predicates and utility calculations. Each of these are covered below.

4.2.2 Data Type Functions

Certain functions represent geometry data types. These data types are BOX, CENTROID, CIRCLE, POINT and POLYGON together with the generalized REGION data type. The functions are similarly named and return a

variable length binary value. The semantics of these data types are based on the corresponding concepts from the STC data model (See [3]).

Geometry data types are centered around the BNF construct `<value_expression>` which is central to data types within SQL.

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

A `<geometry_value_expression>` does not simply cover data type functions (POINT, CIRCLE etc) but must also allow for user defined functions and column values where a geometry data type is stored in a column.

Therefore, `<geometry_value_expression>` is expanded as

```
<geometry_value_expression> ::=
    <value_expression_primary>
  | <geometry_value_function>
```

, where

```
<geometry_value_function> ::=
    <box>
  | <centroid>
  | <circle>
  | <point>
  | <polygon>
  | <region>
  | <user_defined_function>
```

and `<value_expression_primary>` makes possible to use a column reference.

4.2.3 Predicate Functions

Language feature :

Functions CONTAINS and INTERSECTS each accept two geometry data types and return 1 or 0 according to whether the relevant verb (e.g.: "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 SDSS as s 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.

4.2.4 Utility Functions

Function COORDSYS extracts the coordinate system string from a given geometry. To do so it accepts a geometry expression and returns a calculated string value.

This function has been included as a string value function because it returns a simple string value. Hence

```
<string_value_function> :: =
    <string_geometry_function> | <user_defined_function>

<string_geometry_function> ::= <extract_coordsys>

<extract_coordsys> ::=
    COORDSYS <left_paren> <geometry_value_expression> <right_paren>
```

Note - as of this version of the specification (2.1), the COORDSYS function has been marked as deprecated. This function may be removed in future versions of this specification.

Functions like AREA, COORD1, COORD2 and DISTANCE accept a geometry and return a calculated numeric value.

The specification defines two versions of the DISTANCE function, one that accepts accept two geometries, and one that accepts four separate numeric values, both forms return a numeric value.

The Predicate and most of the Utility functions have been included as numeric value functions because they return simple numeric values. Thus

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

, where

```
<numeric_geometry_function> ::=
    <predicate_geometry_function>
  | <non_predicate_geometry_function>
```

and

```
<non_predicate_geometry_function> ::=
    AREA <left_paren> <geometry_value_expression> <right_paren>
  | COORD1 <left_paren> <coord_value> <right_paren>
  | COORD2 <left_paren> <coord_value> <right_paren>
  | DISTANCE <left_paren>
        <coord_value> <comma>
        <coord_value>
        <right_paren>
  | DISTANCE <left_paren>
        <numeric_value_expression> <comma>
        <numeric_value_expression> <comma>
        <numeric_value_expression> <comma>
        <numeric_value_expression>
        <right_paren>
```

and

```
<predicate_geometry_function> ::= <contains> | <intersects>
```

The following sections provide a detailed description for each geometrical function. In each case, the functionality and usage is described rather than going into the BNF grammar details as above.

4.2.5 AREA

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: AREA`

This function computes the area, in square degrees, of a given geometry.

For example, the area of a circle of one degree radius centered on a position of (25.4, -20.0) degrees would be written as follows:

```
AREA(CIRCLE('', 25.4, -20.0, 1))
```

The coordinates of the circle center could also be directly derived from either a POINT function (See 2.4.12) or the coordinate's column references:

```
AREA(CIRCLE('', t.ra, t.dec, 1))
```

, where t would be the table and *ra*, *dec* the column references for the circle centre.

Inappropriate geometries for this construct (e.g. POINT) SHOULD either return zero or throw an error message, the later to be defined by the service making use of ADQL.

4.2.6 BOX

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: BOX`

This function expresses a box on the sky. A box is a special case of Polygon, defined purely for convenience, and it corresponds semantically to the STC Box region ([3], section 4.5.1.5). It is specified by a center position and size (in both coordinates) defining a cross centered on the center position and with arms extending, parallel to the coordinate axes at the center position, for half the respective sizes on either side. The box's sides are line segments or great circles intersecting the arms of the cross in its end points at right angles with the arms.

The function arguments specify the coordinate system, the center position and both the width and height (arms) values, where

- the coordinate system is a string value expression as defined in Section 2.4.1.
- the center position is a comma separated numeric duple, with units and legal ranges as defined in Section 2.4.1.
- and the arms are numeric value expressions in degrees.

For example, a function expressing a box of ten degrees centered on a position (25.4, -20.0) in degrees would be written as follows:

```
BOX('', 25.4, -20.0, 10, 10)
```

As another example, the coordinates of the center position could also be extracted from either a POINT function (See 2.4.12) or the coordinate's column references:

```
BOX('', t.ra, t.dec, 10, 10)
```

, where *t* would be the table and *ra*, *dec* the column references for the center position.

To see what this function would return when listed in the select clause, see Section 2.4.15.

4.2.7 CENTROID

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: CENTROID`

This function computes the centroid of a given geometry and returns a POINT (See 2.4.11).

For example, the centroid of a circle of one degree radius centered in a position of (25.4, -20.0) degrees would be written as follows :

```
CENTROID(CIRCLE ('', 25.4, -20.0, 1))
```

4.2.8 CIRCLE

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: CIRCLE`

This function expresses a circular region on the sky (a cone in space) and corresponds semantically to the STC Circle region ([3], section 4.5.1.2).. The function arguments specify the coordinate system, the center position, and the radius, where:

- the coordinate system is a string value expression as defined in Section 2.4.1.
- the center position is a comma separated numeric duple, with units and legal ranges as defined in Section 2.4.1.
- and the radius is a numeric value expression in degrees.

For example, a function expressing a circle of one degree radius centered on a position of (25.4, -20.0) degrees would be written as follows:

```
CIRCLE('', 25.4, -20.0, 1)
```

The coordinates of the center position could also be derived from either a POINT function (See 2.4.12) or the coordinate's column references:

```
CIRCLE('', t.ra, t.dec, 1)
```

, where *t* would be the table and *ra*, *dec* the column references for the center position.

To see what this function would return when listed in the select clause, see Section 2.4.15.

4.2.9 CONTAINS

Language feature :

```
type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo
name: CONTAINS
```

This numeric function determines if a geometry is wholly contained within another. This is most commonly used to express the "point-in-shape" condition.

For example, to determine if a point with right ascension of 25 degrees and declination of -19.5 degrees is within a circle of one degree radius centered in a position of (25.4, -20.0) degrees and defined according to the same coordinate system, we would make use of the CONTAINS function as follows:

```
CONTAINS(
    POINT('', 25.0,-19.5),
    CIRCLE('', 25.4, -20.0, 1)
    )
```

, where the CONTAINS function returns 1 (true) if the first argument is in or on the boundary of the circle 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(
    POINT('', 25.0,-19.5),
    CIRCLE('', 25.4, -20.0, 1)
    ) = 1
```

for "does contain" and

```
CONTAINS(
    POINT('', 25.0,-19.5),
    CIRCLE('', 25.4, -20.0, 1)
    ) = 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 throw an error message, to be defined by the service making use of ADQL.

4.2.10 COORD1

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: COORD1`

This function extracts the first coordinate value, in degrees, of a given POINT (See 2.4.12) or column reference.

For example, the right ascension of a point with position (25, -19.5) in degrees would be obtained using the following expression:

```
COORD1(POINT('', 25.0,-19.5))
```

, being the result a numeric value of 25.0 degrees. The first coordinate could also be derived directly from a column reference as follows:

```
COORD1(t.point)
```

, where t is the table and *point* the column reference for the POINT geometry stored in the database table.

4.2.11 COORD2

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: COORD2`

This function extracts the second coordinate value, in degrees, of a given POINT (See 2.4.12) or column reference.

For example, the declination of a point with position (25, -19.5) in degrees, would be obtained using the following expression:

```
COORD2(POINT('', 25.0,-19.5))
```

, being the result a numeric value of -19.5 degrees. The second coordinate could also be derived directly from a column reference as follows:

```
COORD2(t.point)
```

, where *t* is the table and *point* the column reference for the POINT geometry stored in the database table.

4.2.12 COORDSYS

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: COORDSYS`

Note - as of this version of the specification (2.1), the COORDSYS function has been marked as deprecated. This function may be removed in future versions of this specification.

This function extracts the coordinate system string value from a given geometry.

As described in section 2.4.1, the allowed return values must be defined by any service making use of ADQL, and a list of standard coordinate system literals can be found in the STC specification [3].

For example, a function extracting the coordinate system of a point with position (25, -19.5) in degrees according to the ICRS coordinate system with GEOCENTER reference position, would be written as follows:

```
COORDSYS(POINT('ICRS GEOCENTER', 25.0,-19.5))
```

, returning the 'ICRS GEOCENTER' string literal. As other samples above, the coordinate system could also be derived from a column referencing any other geometry data type:

```
COORDSYS(t.circle)
```

, where *t* is the table and *circle* the column reference for the CIRCLE geometry stored in the database table.

4.2.13 DISTANCE

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: DISTANCE`

The DISTANCE function computes the arc length along a great circle between two points and returns a numeric value expression in degrees.

The specification defines two versions of the DISTANCE function, one that accepts two geometries, and one that accepts four separate numeric values.

If an ADQL service implementation declares support for DISTANCE, then it must implement both the two parameter and four parameter forms of the function.

For example, a function computing the distance between two points of coordinates (25,-19.5) and (25.4,-20) would be written as follows:

```
DISTANCE(
    POINT('', 25.0, -19.5),
    POINT('', 25.4, -20.0)
    )
```

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

The equivalent call to the four parameter form of the function would be:

```
DISTANCE(
    25.0,
   -19.5,
    25.4,
   -20.0
    )
```

The distance between two points could also be derived from two columns referencing POINT geometries stored in the database tables as follows:

```
DISTANCE(
    t.p1,
    t.p2
    )
```

, where *t* would be the table and *p1*, *p2* the column references for the POINT geometries.

If the two arguments to the two parameter form are expressed in different coordinate systems, the function is responsible for converting one (or both). If it cannot do so, it SHOULD throw an error message, to be defined by the service making use of ADQL.

It is assumed that the arguments for the four parameter form all use the same coordinate system.

4.2.14 INTERSECTS

Language feature :

```
type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo
name: INTERSECTS
```

This numeric function determines if two geometry values overlap. This is most commonly used to express a "shape-vs-shape" intersection test.

For example, to determine whether a circle of one degree radius centered in a position of (25.4, -20.0) degrees overlaps with a box of ten degrees centered in a position (20.0, -15.0) in degrees, we would make use of the INTERSECTS function as follows:

```
INTERSECTS(
    CIRCLE('', 25.4, -20.0, 1),
    BOX('', 20.0, -15.0, 10, 10)
    )
```

, 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(CIRCLE('', 25.4, -20.0, 1),
BOX('', 20.0, -15.0, 10, 10)
) = 1
```

for "does intersect" and

```
INTERSECTS(
    CIRCLE('', 25.4, -20.0, 1),
    BOX('', 20.0, -15.0, 10, 10)
    ) = 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).

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 throw an error message, to be defined by the service making use of ADQL.

The arguments to INTERSECTS SHOULD be geometric expressions evaluating to either BOX, CIRCLE, POLYGON, or REGION. Previous versions of this specification allow POINTs as well and require servers to interpret the expression as a CONTAINS with the POINT moved into the first position. Servers SHOULD still implement that behaviour, but clients SHOULD NOT expect it. This behaviour will be dropped in the next major version of this specification.

4.2.15 POINT

Language feature :

`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`

`name: POINT`

This function expresses a single location on the sky, and corresponds semantically to an STC SpatialCoord ([3], section 4.4.2). The arguments specify the coordinate system and the position, where:

- the coordinate system is a string value expression as defined in Section 2.4.1.
- the position is a comma separated numeric duple, with units and legal ranges as defined in Section 2.4.1.

For example, a function expressing a point with right ascension of 25 degrees and declination of -19.5 degrees would be written as follows:

```
POINT('', 25.0,-19.5)
```

, where numeric values are in degrees. The coordinates of the POINT could also be derived from the coordinate's column references:

```
POINT('', t.ra, t.dec)
```

, where *t* would be the table and *ra*, *dec* the column references for the position.

The coordinates of a POINT could also be individually extracted using the COORD1 and COORD2 functions (See 2.4.7 and 2.4.8).

To see what this function would return when listed in the select clause, see Section 2.4.15.

4.2.16 POLYGON

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: POLYGON`

This function expresses a region on the sky with sides denoted by great circles passing through specified coordinates. It corresponds semantically to the STC Polygon region ([3], section 4.5.1.4). The arguments specify the coordinate system and three or more sets of 2-D coordinates, where:

- the coordinate system is a string value expression as defined in Section 2.4.1.
- the coordinate sets are comma separated numeric duples, with units and legal ranges as defined in Section 2.4.1.

For example, a function expressing a triangle, whose vertices are (10.0, -10.5), (20.0, 20.5) and (30.0,30.5) in degrees would be written as follows:

```
POLYGON('', 10.0, -10.5, 20.0, 20.5, 30.0, 30.5)
```

, where all numeric values are in degrees,
As for other geometries like BOX, CIRCLE and POINT, one could also derive the coordinates from database column references instead:

```
POLYGON('', t.ra, t.dec, 20.0, 20.5, 30.0, 30.5)
```

, where t would be the table and ra, dec the column references for one of the triangle's corner position.
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.

4.2.17 REGION

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-geo`
`name: REGION`

This function provides a generic way of expressing a region represented by a single string input parameter. The format of the string MUST be specified by a service that accepts ADQL by referring to a standard format. Currently STC/s is the only standardized string representation a service can declare.

For example, given a string serialization of an STC region, the REGION function just embeds such literal within parenthesis in the following way:

```
REGION('Convex ... Position ... Error ... Size')
```

A detailed description on how to use STC/s can be seen in the referenced document [4]. Inappropriate geometries for this construct SHOULD throw an error message, to be defined by the service making use of ADQL.

4.2.18 Geometry 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, BOX, or POLYGON) and all the required numeric arguments.

Previous versions of this specification requred the text form to include the coordinate system string. However, as the coordinate system has been marked as deprecated in this version of the specification (2.1), the text form should contain an empty string `''` in place of the coordinate system. Future versions of this specification may remove the coordinate system parameter from the text form.

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

could return

```
CIRCLE('', 1.0, 2.0, 0.5)
```

or equivalent. The output may alter the numeric format by converting whole numbers to floating point (as in the example above) but should not gratuitously add digits. Otherwise, numeric output must conform to the rules for numeric expressions in the ADQL BNF.

4.3 User Defined Functions

4.3.1 Overview

ADQL also 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.

4.3.2 Metadata

The URI for identifying the language feature for a user defined function is defined as part of the TAPRegExt-1.0 standard (Demleitner et al., 2012).

```
ivo://ivoa.net/std/TAPRegExt#features-udf
```

For user defined functions, the `form` element of the language feature declaration must contain the signature of the function, written to match the signature nonterminal in the following grammar:

```
signature ::= <funcname> <arglist> "->" <type_name>
funcname ::= <regular_identifier>
arglist ::= "(" <arg> { "," <arg> } ")"
arg ::= <regular_identifier> <type_name>
```

For example, the following fragment declares a user defined function that takes two `TEXT` parameters and returns an integer, zero or one, depending on the regular expression pattern matching.

```
<languageFeatures type="ivo://ivoa.net/std/TAPRegExt#features-udf">
    <feature>
        <form>match(pattern TEXT, string TEXT) -> INTEGER</form>
```

```
            <description>
                match returns 1 if the POSIX regular expression pattern
                matches anything in string, 0 otherwise.
            </description>
        </feature>
    </languageFeatures>
```

See the TAPRegExt standard for full details on how to use the XML schema to declare user defined functions.

4.4 String functions and operators

An ADQL service implementation MAY include support for the following optional string manipulation and comparison operators.

- `LOWER()` Lower case conversion
- `ILIKE` Case insensitive comparison

4.4.1 LOWER

Language feature :

`type: ivo://ivoa.net/std/TAPRegExt#features-adql-string`

`name: LOWER`

The LOWER function converts its string parameter to lower case.

Since case folding is a nontrivial operation in a multi-encoding world, ADQL requires standard behaviour for the ASCII characters, and recommends following algorithm R2 described in section 3.13, "Default Case Algorithms" of The Unicode Consortium (2012) for characters outside th ASCII set.

```
LOWER('Francis Albert Augustus Charles Emmanuel')
=>
'francis albert augustus charles emmanuel'
```

4.4.2 ILIKE

Language feature :

`type: ivo://ivoa.net/std/TAPRegExt#features-adql-string`

`name: ILIKE`

The ILIKE string comparison operator performs a case insensitive comparison of its string operands.

```
'Francis'  LIKE 'francis' => False
```

```
'Francis' ILIKE 'francis' => True
```

Since case folding is a nontrivial operation in a multi-encoding world, ADQL requires standard behaviour for the ASCII characters, and recommends following algorithm R2 described in section 3.13, "Default Case Algorithms" of The Unicode Consortium (2012) for characters outside th ASCII set.

4.5 Set operators

An ADQL service implementation MAY include support for the following optional set operators:

- `UNION`
- `EXCEPT`
- `INTERSECT`

4.5.1 UNION

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-sets`
`name: UNION`

"The UNION clause combines the results of two SQL queries into a single table of all matching rows. Any duplicate records are automatically removed unless UNION ALL is used."[1]

For a UNION operation to be valid in ADQL, the following criteria MUST be met:

- The two queries MUST result in the same number of columns
- The corresponding columns in the operands MUST have the same data types
- The corresponding columns in the operands SHOULD have the same metadata, e.g. units, UCD etc.
- The metadata for the results SHOULD be generated from the left-hand operand

Note that the comparison used for removing duplicates is based purely on the column value only and does not take into account the units. This means that row with a numeric value of `2` and units of `m` and a row with a numeric value of `2` and units of `km` will be considered equal.

[1] https://en.wikipedia.org/wiki/Set_operations_%28SQL%29UNION_operator

4.5.2 EXCEPT

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-sets`
`name: EXCEPT`

"The EXCEPT operator takes the distinct rows of one query and returns the rows that do not appear in a second result set. The EXCEPT ALL operator does not remove duplicates. For purposes of row elimination and duplicate removal, the EXCEPT operator does not distinguish between NULLs."[2]

For an EXCEPT operation to be valid in ADQL, the following criteria MUST be met:

- The two queries MUST result in the same number of columns
- The corresponding columns in the operands MUST have the same data types
- The corresponding columns in the operands SHOULD have the same metadata, e.g. units, UCD etc.
- The metadata for the results MUST be generated from the left-hand operand

Note that the comparison used for identifying matching rows and for removing duplicates is based purely on the column value only and does not take into account any units declared in the metadata.

This means that row with a numeric value of `2` and units of `m` and a row with a numeric value of `2` and units of `km` will be considered equal.

4.5.3 INTERSECT

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-sets`
`name: INTERSECT`

"The INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets. For purposes of duplicate removal the INTERSECT operator does not distinguish between NULLs. The INTERSECT operator removes duplicate rows from the final result set. The INTERSECT ALL operator does not remove duplicate rows from the final result set."[3]

[2] https://en.wikipedia.org/wiki/Set_operations_%28SQL%29#EXCEPT_operator
[3] https://en.wikipedia.org/wiki/Set_operations_%28SQL%29#INTERSECT_operator

For an INTERSECT operation to be valid in ADQL, the following criteria MUST be met:

- The two queries MUST result in the same number of columns
- The corresponding columns in the operands MUST have the same data types
- The corresponding columns in the operands SHOULD have the same metadata, e.g. units, UCD etc.
- The metadata for the results MUST be generated from the left-hand operand

Note that the comparison used for identifying matching rows and for removing duplicates is based purely on the column value only and does not take into account the units.

This means that row with a numeric value of `2` and units of `m` and a row with a numeric value of `2` and units of `km` will be considered equal.

4.6 Common table expressions

An ADQL service implementation MAY include support for the following optional support for common table expressions :

- `WITH`

4.6.1 WITH

Language feature :

```
type: ivo://ivoa.net/std/TAPRegExt#features-adql-common-table
name: WITH
```

"A Common Table Expression, or CTE, is a temporary named result set, derived from a simple query and defined within the execution scope of a SELECT statement."[4]

Using a common table expression can make queries easier to understand by factoring subqueries out of the main SQL statement.

Example

```
WITH alpha_subset AS
    (
    SELECT
        *
```

[4] https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL#Common_table_expression

```
    FROM
        alpha_source
    WHERE
        id % 10 = 0
    )
SELECT
    *
FROM
    alpha_subset
WHERE
    ra BETWEEN 10 AND 20
```

4.7 Type operations

An ADQL service implementation MAY include support for the following optional type conversion functions:

- `CAST()`

4.7.1 CAST

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-type`
`name: CAST`

The `CAST()` function returns the value of the first argument converted to the data type specified by the second argument.

The ADQL `CAST()` function does not replicate the full functionality and range of types supported by common RDBMS implementations of CAST.

The ADQL `CAST()` function only supports type conversion between the standard numeric data types. The `CAST()` function does not support casting to or from the character, binary, datetime or geometric data types.

Type	Numeric	Character	Binary	Datetime	Geometric
Numeric	Y	N	N	N	N
Character	N	N	N	N	N
Binary	N	N	N	N	N
Datetime	N	N	N	N	N
Geometric	N	N	N	N	N

Table 4: CAST type groups

The ADQL `CAST()` function supports type conversion between the numeric data types.

When converting from floating point value (REAL or DOUBLE) to an integer value (SHORTINT, INTEGER or BIGINT) the rounding mechanism used is implementation dependent.

Type	SHORTINT	INTEGER	BIGINT	REAL	DOUBLE
SHORTINT	-	Y	Y	Y	Y
INTEGER	Y	-	Y	Y	Y
BIGINT	Y	Y	-	Y	Y
REAL	Y	Y	Y	-	Y
DOUBLE	Y	Y	Y	Y	-

Table 5: CAST numeric types

When converting a numeric value to a data type that is too small to represent the value, this SHOULD be treated as an error. However, the mechanism for reporiting the overflow condition is implementation dependent.

4.8 Unit operations

An ADQL service implementation MAY include support for the following optional unit conversion functions:

- `IN_UNIT()`

4.8.1 IN_UNIT

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-unit`
`name: IN_UNIT`

The `IN_UNIT()` function returns the value of the first argument transformed into the units defined by the second argument.

The second argument MUST be a string literal containing a valid unit description using the formatting defined in the VOUnits specification (Demleitner et al., 2014).

- If the second argument is not a valid unit description, then the query is rejected as erroneous.
- If the translator does not know how to convert the value into the requested units, then the query is rejected as erroneous.

4.9 Bitwise operators

An ADQL service implementation MAY include support for the following optional bitwise operators:

- not `~ x`

- and `x & y`
- or `x | y`
- xor `x ^ y`

4.9.1 Bit AND

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-bitwise`
`name: BIT_AND`

The ampersand, `&`, operator performs a bit wise AND operation on two integer operands.

```
x & y
```

The the bitwise AND operation is only valid for integer numeric values, SMALLINT, INTEGER or BIGINT. If the operands are not integer values, then the result of the bitwise AND operation is undefined.

4.9.2 Bit OR

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-bitwise`
`name: BIT_OR`

The vertical bar, `|`, operator performs a bit wise OR operation on two integer operands.

```
x | y
```

The bitwise OR operation is only valid for integer numeric values, SMALLINT, INTEGER or BIGINT. If the operands are not integer values, then the result of the bitwise OR operation is undefined.

4.9.3 Bit XOR

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-bitwise`
`name: BIT_XOR`

The circumflex, `^`, operator performs a bit wise XOR (exclusive or) operation on two integer operands.

```
x ^ y
```

The the bitwise XOR operation is only valid for integer numeric values, SMALLINT, INTEGER or BIGINT. If the operands are not integer values, then the result of the bitwise XOR operation is undefined.

4.9.4 Bit NOT

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-bitwise`
`name: BIT_NOT`

The tilde, `~`, operator performs a bit wise NOT operation on an integer operand.

```
~ x
```

The the bitwise NOT operation is only valid for integer numeric values, SMALLINT, INTEGER or BIGINT. If the operand is not an integer value, then the result of the bitwise NOT operation is undefined.

4.10 Cardinality

An ADQL service implementation MAY include support for the following optional clauses to modify the cardinality of query results.

4.10.1 OFFSET

Language feature :
`type: ivo://ivoa.net/std/TAPRegExt#features-adql-offset`
`name: OFFSET`

An ADQL service implementation MAY include support for the OFFSET clause which limits the number of rows returned by removing a specified number of rows from the beginning of the result set.

If a query contains both an ORDER BY clause and an OFFSET clause, then the ORDER BY is applied before the specified number of of rows are dropped by the OFFSET clause.

If the total number of rows is less than is less than the value specified by the OFFSET clause, then the result set is empty.

If a query contains both an OFFSET clause and a TOP clause, then the OFFSET clause is applied first, dropping the specified number of rows from the begining of the result set before the TOP clause is applied to limit the number of rows returned.

A BNF Grammar

An easier to navigate version of the BNF grammar can be found at
http://www.ivoa.net/internal/IVOA/IvoaVOQL/adql-bnf-v2.0.html

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

<ADQL_reserved_word> ::=
    ABS
  | ACOS
  | AREA
  | ASIN
  | ATAN
  | ATAN2
  | BIT_AND
  | BIT_NOT
  | BIT_OR
  | BIT_XOR
  | BOX
  | CEILING
  | CENTROID
  | CIRCLE
  | CONTAINS
  | COORD1
  | COORD2
  | COORDSYS
  | COS
  | DEGREES
  | DISTANCE
  | EXP
  | FLOOR
  | ILIKE
  | INTERSECTS
  | IN_UNIT
  | LOG
  | LOG10
  | MOD
  | PI
  | POINT
  | POLYGON
  | POWER
  | RADIANS
  | REGION
  | RAND
  | ROUND
```

```
  | SIN
  | 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 | ILIKE | 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>

<bitwise_expression> ::=
    <bitwise_not> <numeric_value_expression>
    | <numeric_value_expression> <bitwise_and> <numeric_value_expression>
    | <numeric_value_expression> <bitwise_or>  <numeric_value_expression>
    | <numeric_value_expression> <bitwise_xor> <numeric_value_expression>

<bitwise_and> ::= <ampersand>
<bitwise_not> ::= <tilde>
<bitwise_or>  ::= <vertical_bar>
<bitwise_xor> ::= <circumflex>

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

<boolean_function> ::=

<boolean_literal> ::= True | False

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

<boolean_term> ::=
```

```
    <boolean_factor>
  | <boolean_term> AND <boolean_factor>

<boolean_value_expression> ::=
    <boolean_literal>
  | <boolean_function>
  | <user_defined_function>

<box> ::=
    BOX <left_paren>
        <coord_sys>
        <comma> <coordinates>
        <comma> <numeric_value_expression>
        <comma> <numeric_value_expression>
    <right_paren>

<catalog_name> ::= <identifier>

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

<character_factor> ::= <character_primary>

<character_primary> ::=
    <value_expression_primary>
  | <string_value_function>

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

<character_string_literal> ::=
    <quote> [ <character_representation>... ] <quote>

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

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

<circumflex> ::= ^

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

<coord1> ::= COORD1 <left_paren> <coord_value> <right_paren>

<coord2> ::= COORD2 <left_paren> <coord_value> <right_paren>

<coord_sys> ::= <string_value_expression>

<coord_value> ::= <point> | <column_reference>

<coordinate1> ::= <numeric_value_expression>

<coordinate2> ::= <numeric_value_expression>

<coordinates> ::= <coordinate1> <comma> <coordinate2>
```

```
<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>
        <coord_value> <comma>
        <coord_value>
        <right_paren>
  | DISTANCE <left_paren>
        <numeric_value_expression> <comma>
        <numeric_value_expression> <comma>
        <numeric_value_expression> <comma>
        <numeric_value_expression>
        <right_paren>

<double_period> ::= ..

<double_quote> ::= "

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

```
<equals_operator> ::= =

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

<exists_predicate> ::= EXISTS <table_subquery>

<exponent> ::= <signed_integer>

<extract_coordsys> ::=
    COORDSYS <left_paren>
        <geometry_value_expression>
    <right_paren>

<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_value_expression> ::=
    <value_expression_primary > | <geometry_value_function>

<geometry_value_function> ::=
    <box>
  | <centroid>
  | <circle>
  | <point>
  | <polygon>
  | <region>
  | <user_defined_function>

<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>

<hex_digit> ::= <digit> | a | b | c | d | e | f | A | B | C | D | E | F

<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 > ::=
    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>

<left_bracket> ::= [

<left_paren> ::= (

<less_than_operator> ::= <

<less_than_or_equals_operator> ::= <=

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

```
<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>
  | LOG10 <left_paren> <numeric_value_expression> <right_paren>
  | MOD <left_paren>
        <numeric_value_expression> <comma> <numeric_value_expression>
    <right_paren>
  | PI <left_paren><right_paren>
  | POWER <left_paren>
        <numeric_value_expression> <comma> <numeric_value_expression>
    <right_paren>
  | RADIANS <left_paren> <numeric_value_expression> <right_paren>
  | RAND <left_paren> [ <unsigned_decimal> ] <right_paren>
  | ROUND <left_paren>
        <numeric_value_expression> [ <comma> <signed_integer>]
    <right_paren>
  | SQRT <left_paren> <numeric_value_expression> <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> ::=

<non_predicate_geometry_function> ::=
    <area>
  | <coord1>
  | <coord2>
  | <distance>

<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> ::= !=

<non_join_query_expression> ::=
    <non_join_query_term>
    | <query_expression> UNION [ ALL ] <query_term>
    | <query_expression> EXCEPT [ ALL ] <query_term>

<non_join_query_primary> ::=
    <query_specification>
    | <left_paren> <non_join_query_expression> <right_paren>

<non_join_query_term> ::=
    <non_join_query_primary>
    | <query_term> INTERSECT [ ALL ] <query_expression>

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

<numeric_geometry_function> ::=
    <predicate_geometry_function> | <non_predicate_geometry_function>

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

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

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

<offset_clause> ::= OFFSET <unsigned_decimal>

<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>

<predicate_geometry_function> ::= <contains> | <intersects>

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

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

<query_expression> ::=
    <non_join_query_expression>
    | <joined_table>

<query_term> ::=
    <non_join_query_term>
    | <joined_table>
```

```
<query_name> ::= <identifier>

<query_specification> ::=
    WITH <with_query> [, ...]
    <select_query>

<question_mark> ::= ?

<quote> ::= ’

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

<radius> ::= <numeric_value_expression>

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

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

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

<semicolon> ::= ;

<set_function_specification> ::=
    COUNT <left_paren> <asterisk> <right_paren>
```

```
  | <general_set_function>

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

<set_limit> ::= TOP <unsigned_decimal>

<set_quantifier> ::= DISTINCT | ALL

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

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

<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_decimal>

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

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

<space> ::=

<string_geometry_function> ::= <extract_coordsys>

<string_value_expression> ::= <character_value_expression>

<string_value_function> ::=
    <string_geometry_function> | <user_defined_function>

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

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

```
    [ <offset_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>

<tilde> ::= ~

<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_decimal> ::= <digit>...

<unsigned_hexadecimal> ::= 0x<hex_digit>...

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

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

```
<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>
  | <boolean_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>

<with_query> :=
    <query_name>
    [ (<column_name> [,...]) ] AS (<query_specification>)
```

B Language feature support

Within the TAPRegExt (Demleitner et al., 2012) XML schema, each group of features is described by a `languageFeatures` element, which has a `type` URI that identifies the group, and contains a `form` element for each individual feature from the group that the service supports.

For example, the following XML fragment describes a service that sup-

ports the `POINT` and `CIRCLE` functions from the set of geometrical functions identified by the URI `ivo://ivoa.net/std/TAPRegExt#features-adql-geo`.

```
<languageFeatures
    type="ivo://ivoa.net/std/TAPRegExt#features-adql-geo"
    >
    <feature>
        <form>POINT</form>
    </feature>
    <feature>
        <form>CIRCLE</form>
    </feature>
</languageFeatures>
```

C Changes from Previous Versions

C.1 Changes from ADQL-2.0

- Typo fixes to the ADQL grammar.
- Changes from Ortiz et al. (2015)
  - Added boolean type (svn version 3364).
  - Removed bitwise functions and updated the operators (svn version 3365).
  - Changed 'hierarchical queries' to 'common table expressions' (svn version 3366).
  - Added OFFSET clause (svn version 3367).
  - Added four parameter DISTANCE (svn version 3370).
  - Added hexadecimal literals (svn version 3374).
- Changes from Demleitner et al. (2013)
  - 2.1.1. (done) The Separator Nonterminal
    - Imported changes from Molinaro (2014)
  - 2.1.2. (done) Type System
  - 2.1.4. (done) Empty Coordinate Systems
  - 2.1.5. (done) Explanation of optional features
  - 2.2.2. (done) No Type-based Decay of INTERSECTS
  - 2.2.3. (done) Generalized User Defined Functions
  - 2.2.4. (done) Case-Insensitive String Comparisons
  - 2.2.5. (done) Set Operators

  - 2.2.6. (TODO) Boolean Type
  - 2.2.7. (done) Casting to Unit
  - 2.2.10. (done) Bitwise operators
  - 2.2.10. (TODO) Hexadecimal literals
  - 2.2.11. (done) CAST operator
  - 2.NN (done) WITH
- Created [Optional components] section.
- Moved [Geometrical Functions] into [Optional components].
- Added [Language feature] information.

References

Arviset, C., Gaudet, S. and the IVOA Technical Coordination Group (2010), 'IVOA architecture', IVOA Note.
URL: *http://www.ivoa.net/documents/Notes/IVOAArchitecture*

Bradner, S. (1997), 'Key words for use in RFCs to indicate requirement levels', RFC 2119.
URL: *http://www.ietf.org/rfc/rfc2119.txt*

Demleitner, M., Derriere, S., Gray, N., Louys, M. and Ochsenbein, F. (2014), 'Units in the VO, version 1.0', IVOA Recommendation.
URL: *http://www.ivoa.net/documents/VOUnits/index.html*

Demleitner, M., Dowler, P., Plante, R., Rixon, G. and Taylor, M. (2012), 'TAPRegExt: a VOResource schema extension for describing TAP services, version 1.0', IVOA Recommendation.
URL: *http://www.ivoa.net/documents/TAPRegExt*

Demleitner, M., Harrison, P. and Taylor, M. (2013), 'TAP Implementation Notes, Version 1.0', IVOA Note.

Dowler, P., Rixon, G. and Tody, D. (2010), 'Table access protocol version 1.0', IVOA Recommendation.
URL: *http://www.ivoa.net/documents/TAP*

Molinaro, M. (2014), 'Adql 2.0 erratum 1: remove nonterminal separator grammar token', IVOA Note.

Ortiz, I., Lusted, J., Dowler, P., Szalay, A., Shirasaki, Y., Nieto-Santisteba, M. A., Ohishi, M., O'Mullane, W., Osuna, P., the VOQL-TEG and the VOQL Working Group (2015), 'IVOA astronomical data query language,

version 2.1 (wd), 2015-06-01', IVOA Working draft.
URL: *http://wiki.ivoa.net/internal/IVOA/ADQL/WD-ADQL-2.1-20150601.pdf*

Plante, R., Stébé, A., Benson, K., Dowler, P., Graham, M., Greene, G., Harrison, P., Lemson, G., Linde, T. and Rixon, G. (2010), 'VODataService: a VOResource schema extension for describing collections and services version 1.1', IVOA Recommendation.
URL: *http://www.ivoa.net/documents/VODataService/*

The Unicode Consortium (2012), 'The Unicode standard, version 6.1 core specification'.
URL: *http://www.unicode.org/versions/Unicode6.1.0*