



*International
Virtual
Observatory
Alliance*

Astronomical Data Query Language

Version 2.1

IVOA Working Draft 2017-10-23

Working group

Data Access Layer Working Group

This version

<http://www.ivoa.net/documents/ADQL/20171023>

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

Version Control

Revision 4554, 2017-10-24 22:44:46 -0300 (Tue, 24 Oct 2017)

<https://volute.g-vo.org/svn/trunk/projects/dal/ADQL/ADQL.tex>

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	Language structure	7
2.1	Characters, Keywords, Identifiers and Literals	8
2.1.1	Characters	8
2.1.2	Keywords and Identifiers	9
2.1.3	SQL reserved keywords	9
2.1.4	ADQL reserved keywords	10
2.1.5	ADQL deprecated keywords	10
2.1.6	Identifiers	10
2.1.7	Escape syntax	10
2.1.8	Case sensitivity	10
2.1.9	Literals	11
2.2	Query syntax	13
2.2.1	Subqueries	14
2.2.2	Joins	14
2.2.3	Search condition	14
2.3	Mathematical and Trigonometrical Functions	16
3	Type System	18
3.1	Logical types	18
3.1.1	BOOLEAN	18
3.2	Numeric types	19
3.2.1	Numeric primitives	19
3.2.2	INTERVAL	19
3.3	Date and time	20
3.3.1	TIMESTAMP	20
3.4	Character types	20
3.4.1	Character primitives	20

3.4.2	CLOB	21
3.5	Binary types	21
3.5.1	Binary primitives	21
3.5.2	BLOB	21
3.6	Geometric types	22
3.6.1	POINT	22
3.6.2	CIRCLE	22
3.6.3	POLYGON	23
4	Optional components	24
4.1	Service capabilities	24
4.2	Geometrical Functions	24
4.2.1	Overview	24
4.2.2	Data Type Functions	25
4.2.3	Coordsys	26
4.2.4	Predicate Functions	26
4.2.5	Utility Functions	26
4.2.6	AREA	29
4.2.7	BOX	29
4.2.8	CENTROID	30
4.2.9	CIRCLE	31
4.2.10	CONTAINS	32
4.2.11	COORD1	33
4.2.12	COORD2	34
4.2.13	COORDSYS	34
4.2.14	DISTANCE	35
4.2.15	INTERSECTS	36
4.2.16	POINT	38
4.2.17	POLYGON	39
4.3	User Defined Functions	40
4.3.1	Overview	40
4.3.2	Metadata	41
4.4	String functions and operators	42
4.4.1	LOWER	42
4.4.2	ILIKE	42
4.5	Set operators	43
4.5.1	UNION	43
4.5.2	EXCEPT	44
4.5.3	INTERSECT	44
4.6	Common table expressions	45
4.6.1	WITH	45

4.7	Type operations	46
4.7.1	CAST	46
4.8	Unit operations	47
4.8.1	IN_UNIT	47
4.9	Bitwise operators	47
4.9.1	Bit AND	48
4.9.2	Bit OR	48
4.9.3	Bit XOR	48
4.9.4	Bit NOT	49
4.10	Cardinality	49
4.10.1	OFFSET	49
A	BNF Grammar	50
B	Language feature support	66
C	Changes from Previous Versions	67

Acknowledgements

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 a 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 Backus Naur Form (BNF) based language definition. 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 keywords indicating requirement levels can be found in the References section.

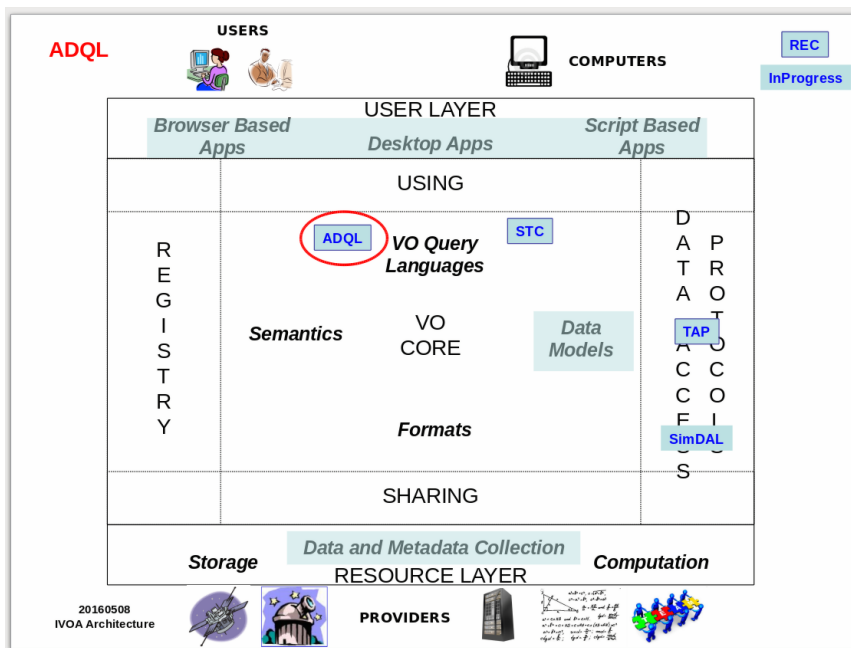


Figure 1: Architecture diagram for this document

1.1 Role within the VO Architecture

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

2 Language structure

This section describes the ADQL language structure. 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 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 insensitive 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 unless it is isolated using the ADQL escape syntax defined in section 2.1.7 [Escape syntax](#).

The ADQL specification extends 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).

Although the following lists are all in UPPERCASE, the matching of keywords is case insensitive.

2.1.3 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

2.1.4 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

2.1.5 ADQL deprecated keywords

REGION

2.1.6 Identifiers

Identifiers MUST begin with a letter {aA-zZ}, subsequent characters MAY be letters, underscores or digits {0-9} as follows:

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

2.1.7 Escape syntax

To address reserved keyword and special character conflicts the ADQL language provides a way to escape a non-compliant identifiers by using the double quote character " as a delimiter.

For example, to use the reserved word `size` as a column name it must be isolated using double quotes.

- `size` – Invalid column name
- `"size"` – Valid column name

2.1.8 Case sensitivity

In addition to isolating keyword conflicts and and special characters, the double quote escape syntax also denotes case sensitivity.

Without double quotes, the following identifiers are all equivalent:

```
alpha == Alpha == ALPHA
```

When escaped using double quotes, the same set of identifiers are not equivalent:

```
"alpha" != "Alpha" != "ALPHA"
```

2.1.9 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 more detailed description of the select statement can be found in Appendix A: BNF Grammar at the <query_specification> construct. A simplified syntax for the **SELECT** statement follows, 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 apply to a set of tables specified in the **FROM** clause. As a result of this query, a subset of the tables 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 both **TOP** and **OFFSET** are used together then **OFFSET** is applied first followed 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, unless asterisk is specified. The selection list **MAY** include any numeric, string or geometry value expression.

2.2.1 Subqueries

Table subqueries MAY be used by predicates such as `IN` and `EXISTS` in the `WHERE` clause of a query:

```
SELECT
    alpha_source.id
FROM
    alpha_source
WHERE
    alpha_sourceid >=5
AND
    alpha_sourceid IN
    (
        SELECT id FROM alpha_source WHERE id < 10
    )
```

Table subqueries MAY be used for declaring derived tables in the `FROM` clause of a query:

```
SELECT
    alpha_source.id
FROM
    alpha_source,
    (
        SELECT alpha_source.id FROM alpha_source WHERE id < 10
    ) AS subsample
WHERE
    alpha_source.id >=5
AND
    alpha_source.id = subsample.id
```

2.2.2 Joins

ADQL supports `INNER` and `OUTER` (`LEFT`, `RIGHT` and `FULL`) joins. If no type is specified, the default is `INNER`. All of these can be `NATURAL` or not.

2.2.3 Search condition

The search condition can be part of several other clauses: `JOIN`, `HAVING` and, `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 include the 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 x/y.
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 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, service 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 the `TAP_SCHEMA` tables defined in the TAP specification and the `/tables` webservice response defined in the VOSI specification.

Services SHOULD, if at all possible, try to express their column metadata in these terms even if the underlying database employs different types. Services SHOULD also use the following mappings when interfacing to user data, either by serializing result sets into VOTables or by ingesting user-provided VOTables into ADQL-visible tables.

3.1 Logical types

3.1.1 BOOLEAN

The `BOOLEAN` data type maps to the corresponding `boolean` data type is defined in the DALI specification. The serialization format for `boolean` is defined in the VOTable specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
BOOLEAN	boolean	1	-

Table 3: ADQL type mapping for BOOLEAN

The literal values `1` and `TRUE` are equivalent, and the values `0` and `FALSE` are equivalent:

```
foo = 1
foo = TRUE
```

```
bar = 0
bar = FALSE
```

The literal values `TRUE` and `FALSE` are not case sensitive:

```
foo = true
foo = True
foo = TRUE
```

```
bar = 0
bar = false
bar = False
bar = FALSE
```

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 a BOOLEAN value is undefined.

3.2 Numeric types

3.2.1 Numeric primitives

The numeric data types, BIT, SMALLINT, INTEGER, BIGINT, REAL and DOUBLE map to the corresponding data types defined in the VOTable specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
BIT	bit	-	-
SMALLINT	short	-	-
INTEGER	int	-	-
BIGINT	long	-	-
REAL	float	-	-
DOUBLE	double	-	-

Table 4: ADQL type mapping for numeric values

Where possible ADQL numeric values SHOULD be implemented using database types that correspond to the VOTable serialization types, e.g. SMALLINT should map to a 16 bit integer, INTEGER should map to a 32 bit integer, etc.

3.2.2 INTERVAL

The DALI specification defines INTERVAL as a pair of integer or floating-point numeric values which are serialized as an array of numbers.

TBD - The details of how INTERVAL values behave in ADQL are not yet defined.

ADQL	VOTable		
type	datatype	arraysize	xtype
INTERVAL	short, int, float, double	2	interval

Table 5: ADQL type mapping for INTERVAL

3.3 Date and time

Where possible, date and time values SHOULD be implemented as described in the DALI specification.

3.3.1 TIMESTAMP

The `TIMESTAMP` datatype maps to the corresponding type defined in the DALI specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
<code>TIMESTAMP</code>	<code>char</code>	<code>n, n*, *</code>	<code>timestamp</code>

Table 6: ADQL type mapping for `TIMESTAMP`

`TIMESTAMP` literals should be created using the `TIMESTAMP()` constructor, using the syntax defined in the DALI specification:

```
YYYY-MM-DD['T'hh:mm:ss[.SSS] ['Z']]
```

The basic comparison operators, `=`, `<`, `>`, `<=`, `>=`, `<>` and `BETWEEN` can all be applied to `TIMESTAMP` values:

```
SELECT
  ..
WHERE
  obstime > TIMESTAMP('2015-01-01')
OR
  obstime
    BETWEEN
      TIMESTAMP('2014-01-01')
    AND
      TIMESTAMP('2014-01-02')
```

Within the database, the details of how `TIMESTAMP` values are implemented is platform dependent. The primary requirement is that the results of the comparison operators on `TIMESTAMP` values are consistent with respect to chronological time.

3.4 Character types

3.4.1 Character primitives

The `CHAR` and `VARCHAR` data types map to the `char` or `unicodeChar` type defined in the VOTable specification.

The choice of whether `CHAR` and `VARCHAR` map to `char` or `unicodeChar` is implementation dependent and may depend on the data content.

ADQL	VOTable		
type	datatype	arraysize	xtype
CHAR(n)	char, unicodeChar	n	-
VARCHAR(n)	char, unicodeChar	n*	-

Table 7: ADQL type mapping for character strings

3.4.2 CLOB

To support strings which are generated by the server when a query is executed ADQL includes the Character Large Object (CLOB) data type, which behaves as an opaque immutable string of characters.

None of the ADQL operators apply to CLOB values. However, specific database implementations MAY provide user defined functions that operate on some CLOB values.

CLOB values are serialized as arrays of characters.

ADQL	VOTable		
type	datatype	arraysize	xtype
CLOB	char, unicodeChar	n, n*, *	-

Table 8: ADQL type mapping for CLOB

The details of how CLOB values are handled within a database is implementation dependent.

3.5 Binary types

3.5.1 Binary primitives

The BINARY and VARBINARY data types map to the `unsignedByte` type defined in the VOTable specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
BINARY(n)	unsignedByte	n	-
VARBINARY(n)	unsignedByte	n*	-

Table 9: ADQL type mapping for character strings

3.5.2 BLOB

To support large blocks of binary data such as images, ADQL includes the Binary Large Object (BLOB) data type, which behaves as an opaque immutable array of bytes.

None of the ADQL operators apply to BLOB values. However, specific database implementations MAY provide user defined functions that operate on some BLOB values.

BLOB values are serialized as arrays of `unsignedByte` defined in the VOTable specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
BLOB	unsignedByte	n, n*, *	-

Table 10: ADQL type mapping for BLOB

The details of how BLOB values are handled within a database is implementation dependent.

3.6 Geometric types

ADQL provides support for the POINT, CIRCLE and POLYGON geometric types defined in the DALI specification.

3.6.1 POINT

The POINT datatype maps to the corresponding type defined in the DALI specification.

POINT values are serialized as arrays of floating point numbers using the `point` xtype defined in the DALI specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
POINT	float, double	2	point

Table 11: ADQL type mapping for POINT

POINT values can be created using the `POINT()` constructor defined in section 4.2.16. For example:

```
POINT(
  12.3,
  45.6
)
```

3.6.2 CIRCLE

The CIRCLE datatype maps to the corresponding type defined in the DALI specification.

CIRCLE values are serialized as arrays of floating point numbers using the `circle` xtype defined in the DALI specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
CIRCLE	float, double	3	circle

Table 12: ADQL type mapping for CIRCLE

CIRCLE values can be created using the CIRCLE() constructor defined in section 4.2.9. For example:

```
CIRCLE(  
    12.3,  
    45.6,  
    0.5  
)
```

3.6.3 POLYGON

The POLYGON datatype maps to the corresponding type defined in the DALI specification.

POLYGON values are serialized as arrays of floating point numbers using the polygon xtype defined in the DALI specification.

ADQL	VOTable		
type	datatype	arraysize	xtype
POLYGON	float, double	n, *, n*	polygon

Table 13: ADQL type mapping for POLYGON

POLYGON values can be created using the POLYGON() constructor defined in section 4.2.17. For example:

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

describes a triangle, whose vertices are (10.0, -10.5), (20.0, 20.5) and (30.0, 30.5) degrees.

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 feature, then a client SHOULD assume that the service does NOT support that feature, and SHOULD NOT make use of that feature in any ADQL queries that it sends.

4.1 Service capabilities

The TAPRegExt specification defines an XML schema that a service SHOULD use to declare which optional features it supports.

In general, each group of language 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 language features defined in this document using the TAPRegExt specification XML schema.

For full details on the XML schema and how it can be used, please refer to the TAPRegExt specification.

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

If the arguments for a geometric function represent spherical coordinates then the values SHOULD be limited to [0, 360] and [-90, 90], and the units MUST be in degrees (square degrees for area).

If the arguments for a geometric function represent cartesian coordinates then there are no inherent limits to the range of values, but coordinate vectors MUST be normalized.

Details of the mechanism for reporting the out of range arguments are implementation dependent.

4.2.2 Data Type Functions

The following functions provide constructors for each of the geometry data types. The semantics of these data types are based on the corresponding concepts from the STC data model.

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>
| <user_defined_function>
```

and `<value_expression_primary>` enables the use of geometric functions and column references.

4.2.3 Coordsys

For historical reasons, the geometry constructors (BOX, CIRCLE, POINT and POLYGON) all accept an optional string value as the first argument. This was originally intended to carry information on a reference system or other coordinate system metadata. As of this version of the specification 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.

4.2.4 Predicate Functions

Functions CONTAINS and INTERSECTS each accept two geometry data types and return a numeric value of 1 or 0 according to whether the relevant verb (e.g.: "contains") is satisfied against the two input geometries; 1 if the condition is met and 0 if it is not.

Each of these functions can be used as a WHERE clause predicate by comparing the numeric result with zero or one. For example:

```
SELECT
  *
FROM
  table
WHERE
  1 = CONTAINS(
    POINT(...),
    CIRCLE(...)
  )
```

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.

4.2.5 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 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 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.

4.2.6 AREA

Language feature :

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

name: AREA

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

For example, an expression to calculate the area of a POLYGON could be written as follows:

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

The AREA of a single POINT is zero.

The geometry argument may be a literal value, as above, or it may be a column reference, function or expression that returns a geometric type.

4.2.7 BOX

Language feature :

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

name: BOX

The BOX 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. It is specified by a center position and size (in both axes) 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 center position and both the width and height values, where:

- the center position is given by a pair of numeric coordinates in degrees, or a POINT object.
- the the width and height numeric are given by numeric expressions in degrees.

For example, a BOX of ten degrees centered on a position (25.4, -20.0) in degrees could be written as follows:

```
BOX(
  25.4,
  -20.0,
  10.0,
  10.0
)
```

Alternatively, the center position could be expressed as a POINT:

```
BOX(
  POINT(
    25.4,
    -20.0
  ),
  10.0,
  10.0
)
```

The position argument may be a literal value, as above, or it may be a column reference, function or expression that returns a geometric type.

For historical reasons, the BOX function accepts an optional string value as the first argument. As of this version of the specification this parameter has been marked as deprecated. Future versions of this specification may remove this parameter (See 4.2.13).

4.2.8 CENTROID

Language feature :

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

name: CENTROID

The CENTROID function computes the centroid of a given geometry and returns a POINT.

For example, an expression to calculate the centroid of a POLYGON could be written as follows :

```
CENTROID(
  POLYGON(
    10.0,
```

```

        -10.5,
        20.0,
        20.5,
        30.0,
        30.5
    )
)

```

The CENTROID of a single POINT is the POINT.

The geometry argument may be a literal value, as above, or it may be a column reference, function or expression that returns a geometric type.

4.2.9 CIRCLE

Language feature :

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

name: CIRCLE

The CIRCLE function expresses a circular region on the sky (a cone in space) and corresponds semantically to the STC Circle. The function arguments specify the center position and the radius, where:

- the center position is given by a pair of numeric coordinates in degrees, or a single POINT value.
- and the radius is a numeric value expression in degrees.

For example, a CIRCLE of ten degrees radius centered on position (25.4, -20.0) in degrees could be written as follows:

```

CIRCLE(
    25.4,
    -20.0,
    10.0
)

```

Alternatively, the center position could be expressed as a POINT:

```

CIRCLE(
    POINT(
        25.4,
        -20.0,
    ),
    10.0
)

```

The position argument may be a literal value, as above, or it may be a column reference, function or expression that returns a geometric type.

For historical reasons, the CIRCLE function accepts an optional string value as the first argument. As of this version of the specification this parameter has been marked as deprecated. Future versions of this specification may remove this parameter (See 4.2.13).

4.2.10 CONTAINS

Language feature :

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

name: CONTAINS

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

For example, an expression to determine whether the point (25.0, -19.5) degrees is within a circle of ten degrees radius centered on position (25.4, -20.0) degrees, could be written as follows:

```
CONTAINS(  
  POINT(  
    25.0,  
    -19.5  
  ),  
  CIRCLE(  
    25.4,  
    -20.0,  
    10.0  
  )  
)
```

The CONTAINS function is not symmetric in the meaning of the arguments.

The CONTAINS function returns the numeric value 1 if the first argument is in, or on, the boundary of the second argument and the numeric value 0 if it is not.

When used as a predicate in the WHERE clause of a query, the numeric return value must be compared to the numeric values 0 or 1 to form a SQL predicate:

```
WHERE  
  1 = CONTAINS(  
    POINT(  
      25.0,  
      -19.5
```



```

    ),
    CIRCLE(
        25.4,
        -20.0,
        10.0
    )
)

```

for "does contain" and

```

WHERE
    0 = CONTAINS(
        POINT(
            25.0,
            -19.5
        ),
        CIRCLE(
            25.4,
            -20.0,
            10.0
        )
    )

```

for "does not contain".

The geometric arguments for CONTAINS may be a literal values, as above, or they may be column references, functions or expressions that return a geometric type.

If the arguments are expressed in different coordinate systems, the CONTAINS function is responsible for converting one, or both, of the arguments into a different coordinate system. If it cannot perform the required conversion then it SHOULD throw an error. Details of the mechanism for reporting the error condition are implementation dependent.

4.2.11 COORD1

Language feature :

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

name: COORD1

The COORD1 function extracts the first coordinate value, in degrees, of a given POINT (See 4.2.16) 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
      )
    )
```

which would return a numeric value of 25.0 degrees.

The COORD1 function may be applied to any expression that returns a geometric POINT value. For example:

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

where *t.point* is a column reference for a POINT value stored in the database.

4.2.12 COORD2

Language feature :

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

The COORD2 function extracts the second coordinate value, in degrees, of a given POINT (See 4.2.16) or column reference.

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

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

which would return a numeric value of -19.5 degrees.

The COORD2 function may be applied to any expression that returns a geometric POINT value. For example:

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

where *t.point* is a column reference for a POINT value stored in the database.

4.2.13 COORDSYS

Language feature :

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

Note - as of this version of the specification the COORDSYS function has been marked as deprecated. This function may be removed in future versions of this specification. Details of the coordinate system for a database column are available as part of the service metadata, available via the TAP_SCHEMA tables defined in the TAP specification and the /tables webservice response defined in the VOSI specification.

The COORDSYS function returns the formal name of the coordinate system for a given geometry as a string.

The following example would return the coordinate system of a POINT literal:

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

which would return a string value representing the coordinate system used to create the POINT.

The COORDSYS function may be applied to any expression that returns a geometric data type. For example:

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

where *t.circle* is a column reference for a geometric value stored in the database.

4.2.14 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 POINT values, and a second 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, an expression calculating the distance between two points of coordinates (25,-19.5) and (25.4,-20) could 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 function may be applied to any expression that returns a geometric POINT value. For example, the distance between points stored in the database could be calculated as follows:

```

DISTANCE(
  t.p1,
  t.p2
)

```

where *t.p1* and *t.p2* refer to POINT value columns in the database.

If the geometric arguments are expressed in different coordinate systems, the function is responsible for converting one, or both arguments into a different coordinate system. If it cannot perform the required conversion then it SHOULD throw an error. Details of the mechanism for reporting the error condition are implementation dependent.

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

4.2.15 INTERSECTS

Language feature :

type: [ivo://ivoa.net/std/TAPRegExt#features-adql-geo](http://ivoa.net/std/TAPRegExt#features-adql-geo)

name: INTERSECTS

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

For example, an expression to determine whether a circle of one degree radius centered on position (25.4, -20.0) degrees overlaps with a box of ten degrees centered on position (20.0, -15.0) degrees, could be written as follows:

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

where the INTERSECTS function returns the numeric value 1 if the two arguments overlap and 0 if they do not.

When used as a predicate in the WHERE clause of a query, the numeric return value must be compared to the numeric values 0 or 1 to form a SQL predicate:

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

for "does intersect" and

```
WHERE  
  0 = INTERSECTS(  
    CIRCLE(  
      25.4,  
      -20.0,  
      1  
    ),  
    BOX(  
      20.0,  
      -15.0,  
      10,  
      10  
    )  
  )
```

```
        20.0,  
        -15.0,  
        10,  
        10  
    )  
)
```

for "does not intersect".

The geometric arguments for INTERSECTS may be a literal values, as above, or they may be column references, functions or expressions that return a geometric type.

The arguments to INTERSECTS SHOULD be geometric expressions evaluating to either BOX, CIRCLE or POLYGON. 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.

If the geometric arguments are expressed in different coordinate systems, the function is responsible for converting one, or both arguments into a different coordinate system. If it cannot perform the required conversion then it SHOULD throw an error. Details of the mechanism for reporting the error condition are implementation dependent.

4.2.16 POINT

Language feature :

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

name: POINT

This POINT function expresses a single location on the sky, and corresponds semantically to an STC SpatialCoord. The arguments specify the position, where:

- the two numeric coordinates match the units and ranges defined in Section 4.2.1.

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 for POINT may be a literal values, as above, or they may be column references, functions or expressions that return a numeric types:

```
POINT(  
    t.ra,  
    t.dec  
)
```

where *t.ra* and *t.dec* are numeric columns in the database.

For historical reasons, the POINT function accepts an optional string value as the first argument. As of this version of the specification this parameter has been marked as deprecated. Future versions of this specification may remove this parameter (See 4.2.13).

4.2.17 POLYGON

Language feature :

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

name: POLYGON

The POLYGON function expresses a region on the sky with boundaries denoted by great circles passing through specified coordinates. It corresponds semantically to the STC Polygon. The arguments specify the coordinate system and three or more sets of 2-D coordinates, where:

- the coordinates are given as pairs of numeric values, with units and legal ranges as defined in Section 4.2.1.

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.

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.

The coordinates for POLYGON may be a literal values, as above, or they may be column references, functions or expressions that return a numeric values:

```
POLYGON(  
    t.ra,  
    t.dec + 5,  
    t.ra - 5,  
    t.dec - 5,  
    t.ra - 5,  
    t.dec + 5,  
)
```

where *t.ra* and *t.dec* are numeric columns in the database.

For historical reasons, the POLYGON function accepts an optional string value as the first argument. As of this version of the specification this parameter has been marked as deprecated. Future versions of this specification may remove this parameter (See 4.2.13).

4.3 User Defined Functions

4.3.1 Overview

ADQL also provides a place holder 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>  
    [  
        <value_expression>  
        [  
            {  
                <comma> <value_expression>  
            }...  
        ]  
    ]  
    <right_paren>
```

In order to avoid name conflicts, user defined function names SHOULD include a prefix which indicates the name of the institute or project which created the function.

For example, the names of `align` and `convert` functions developed by the Wide Field Astronomy Unit (WFAU) could be prefixed as follows:

```
wfau_align()  
wfau_convert()
```


This enables users to distinguish between function with similar names developed by a different service provider, e.g. the German Astrophysical Virtual Observatory (GAVO):

```
gavo_align()
gavo_convert()
```

The `ivo` prefix is reserved for functions that have been defined in an IVOA specification. For example the RegTAP specification defines the following functions:

```
ivo_nocasematch()
ivo_hasword()
ivo_hashlist_has()
ivo_string_agg()
```

4.3.2 Metadata

The URI for identifying the language feature for a user defined function is defined as part of the TAPRegExt specification.

```
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 specification 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 the 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 the 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 operator combines the results of two queries, accepting rows from both the first and second set of results, removing duplicate rows unless UNION ALL is used.

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 columns in the operands MUST have the same data types.

In addition, the following criteria SHOULD be met:

- The 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 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, despite the difference in units.

$$2\text{m} = 2\text{km}$$

4.5.2 EXCEPT

Language feature :

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

name: EXCEPT

The EXCEPT operator combines the results of two queries, accepting rows that are in the first set of results but are not in the second, removing duplicate rows unless EXCEPT ALL is used.

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 columns in the operands MUST have the same data types.

In addition, the following criteria SHOULD be met:

- The 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 removing duplicates is based purely on the column value 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, despite the difference in units.

$$2\text{m} = 2\text{km}$$

4.5.3 INTERSECT

Language feature :

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

name: INTERSECT

The INTERSECT operator combines the results of two queries, accepting rows that are in the first set of results but are not in the second, removing duplicate rows unless INTERSECT ALL is used.

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 columns in the operands MUST have the same data types.

In addition, the following criteria SHOULD be met:

- The 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 removing duplicates is based purely on the column value 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, despite the difference in units.

`2m = 2km`

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
```

The WITH operator creates a temporary named result set that can be referred to elsewhere in the main query.

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

For example, the following query with a nested subquery:

```
SELECT
    ra,
    dec
FROM
    (
    SELECT
        *
    FROM
        alpha_source
    WHERE
        id % 10 = 0
    )
```

```

WHERE
    ra > 10
AND
    ra < 20

```

can be refactored as a named WITH query and a simpler main query:

```

WITH alpha_subset AS
(
    SELECT
        *
    FROM
        alpha_source
    WHERE
        id % 10 = 0
)

SELECT
    ra,
    dec
FROM
    alpha_subset
WHERE
    ra > 10
AND
    ra < 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.

The rounding mechanism used when converting from floating point values (REAL or DOUBLE) to integer values (SHORTINT, INTEGER or BIGINT) is implementation dependent.

When converting a numeric value to a data type that is too small to represent the value, this SHOULD be treated as an error. Details of the mechanism for reporting the error condition are 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.

The system SHOULD report an error in response to the following conditions:

- If the second argument is not a valid unit description.
- If the system is not able to convert the value into the requested units.

Details of the mechanism for reporting the error condition are implementation dependent.

4.9 Bitwise operators

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

- not \sim 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 bitwise AND operation on two integer operands.

$$x \& y$$

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 bitwise 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 bitwise XOR (exclusive or) operation on two integer operands.

$$x \wedge y$$

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 bitwise NOT operation on an integer operand.

~ x

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:

- OFFSET

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 rows are dropped by the OFFSET clause.

If the total number of rows 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 beginning 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>
    | <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>

<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

In the TAPRegExt specification 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 supports 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

- Changes from WD-ADQL-2.1-20160502
 - Removed hard coded version number
(svn version r4544, 23 Oct 2017)
 - Restored coordsys param for now
(svn version r4543, 23 Oct 2017)
 - Updates to CAST and IN_UNIT
(svn version r4539, 18 Oct 2017)
 - Updates to UNION, EXCEPT, INTERSECT and WITH
(svn version r4538, 18 Oct 2017)
 - Updates to user defined functions
(svn version r4537, 18 Oct 2017)
 - Updates to DISTANCE, POINT and POLYGON
(svn version r4536, 18 Oct 2017)
 - Proof reading typos and readability fixes
(svn version r4527, 16 Oct 2017)
 - Updates to COORDSYS
(svn version r4522, 16 Oct 2017)
 - Removed (commented) text describing coordsys argument
(svn version r4521, 16 Oct 2017)
 - Updates to COORD1, COORD2 and COORDSYS
(svn version r4520, 16 Oct 2017)
 - Updates to AREA, BOX, CENTROID, CIRCLE and CONTAINS
(svn version r4519, 13 Oct 2017)
 - Removed old section about 'Geometry in the SELECT clause'
(svn version r4481, 10 Oct 2017)
 - Updated text for AREA and BOX
(svn version r4480, 10 Oct 2017)
 - Updated text for AREA
(svn version r4469, 09 Oct 2017)
 - Cleaned up text describing ranges for coordinates
(svn version r4467, 09 Oct 2017)
 - Removed REGION
(svn version r4466, 09 Oct 2017)
 - Removed reference to ADQL from section titles in the ADQL document
(svn version r4465, 09 Oct 2017)

- Expanded the data types to add sub-section for each xtype
(svn version r4353, 19 Sep 2017)
- Removed restriction on nested JOINS
(svn version r4283, 12 Sep 2017)
- Added subversion properties
(svn version r4282, 12 Sep 2017)
- Updated types and xtypes to match DALI
(svn version r4281, 12 Sep 2017)
- Clarify text for SELECT and subqueries
(svn version r4256, 12 Sep 2017)
- Improved wording for keywords and identifiers
(svn version r4242, 11 Sep 2017)
- Fixed section references
(svn version r3637, 18 Oct 2016)
- Fixed typo in definition of MOD
(svn version 3456)
- Fixed section references
(svn version 3637)
- Improved wording for keywords and identifiers
(svn version 4242)
- Changes from ADQL-20150601
 - 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 and Harrison et al. \(2013\)](#)
 - 2.1.1. The Separator Nonterminal
 - 2.1.2. Type System
 - 2.1.4. Empty Coordinate Systems

- 2.1.5. Explanation of optional features
 - 2.2.2. No Type-based Decay of INTERSECTS
 - 2.2.3. Generalized User Defined Functions
 - 2.2.4. Case-Insensitive String Comparisons
 - 2.2.5. Set Operators
 - 2.2.6. Boolean Type
 - 2.2.7. Casting to Unit
 - 2.2.10. Bitwise operators
 - 2.2.10. Hexadecimal literals
 - 2.2.11. CAST operator
 - 2.NN 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.
<http://www.ivoa.net/documents/Notes/IVOAArchitecture>
- Bradner, S. (1997), ‘Key words for use in RFCs to indicate requirement levels’, RFC 2119.
<http://www.ietf.org/rfc/rfc2119.txt>
- Demleitner, M., Harrison, P. and Taylor, M. (2013), ‘TAP Implementation Notes, Version 1.0’, IVOA Note.
- The Unicode Consortium (2012), ‘The Unicode standard, version 6.1 core specification’.
<http://www.unicode.org/versions/Unicode6.1.0>