International

Virtual

Observatory

Alliance

IVOA Astronomical Data Query Language Version 1.05

IVOA Working Draft 29 May 2006

This version:

1.05: http://www.ivoa.net/Documents/WD/ADQL/ADQL－20060701.doc

Latest version:

http://www.ivoa.net/Documents/latest/ADQL.html

Previous versions:

none

Working Group:

http://www.ivoa.net/twiki/bin/view/IVOA/IvoaVOQL

Editors:

Yuji Shirasaki, Maria A. Nieto-Santisteban, Masatoshi Ohishi, William O'Mullane, and Alexander Szalay

Authors:

IVOA VOQL Working group

Abstract

This document describes the Astronomical Data Query Language (ADQL) and its two representations as String (ADQL/s) and XML (ADQL/x). ADQL has been developed based on SQL. This document describes the subset of the SQL grammar supported by

ADQL. Special extensions to SQL have been defined in order to support astronomy specific operations such as a geometric data type and its functions.

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".

Acknowledgments

This working draft has been developed based on discussions at various IVOA meetings and continuing emails on the mailing list. The editors express their appreciation for many valuable contributions by Naoki Yasuda, Clive Page, Bob Mann, Martin Hill, and many others.

Contents

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, and the SkyNode Interface protocol to access catalogs. Different VO services have different needs in terms of query complexity. For example, SIAP and SSAP might be satisfied using a single table query. However, SkyNodes usually include more than one catalog table which makes necessary richer language expressivity. ADQL has been designed in a layered hierarchy so services implement and register the complexity level that meets their needs. In this way, clients know what type of queries a VO service will accept.

ADQL is based on the Structured Query Language (SQL). The VO has a number of tabular data sets and many of them are stored in Relational Databases (RDBs), making SQL a convenient access language. ADQL focuses on a subset of the SELECT statement, adding a few extensions to define specific astronomy operations like a geometric data type and its function.

SkyNode services (often denoted as nodes) are an example of VO data services accepting queries in ADQL. The mechanism of passing a query to a node is described in the SkyNode Interface specification [□] developed by the IVOA VOQL WG as well. SkyNodes are defined and implemented as XML Web services. To access some SkyNode implementations you can visit OpenSkyQuery.net. The Open SkyQuery portal is an example of how astronomers can use ADQL to query a federation of astronomical databases which have been published as SkyNodes.

Astronomical Data Query Language (ADQL)

ADQL is based on a subset of SQL, which has been extended to support queries specific to astronomy. ADQL has two representations:

ADQL/s : A string form based on the SELECT statement of the SQL standard [□] that conforms to the ADQL grammar (see appendix). Some non-standard SQL extensions have been added to support astronomy queries.

ADQL/x : An XML document conforming to the ADQL schema [□]. The XML document is the mechanism used to pass a query to VO services as for example the SkyNode Web service interface.

ADQL/s and **ADQL/x** are translatable to each other without loss of information, so this document is mainly devoted to describe syntax of the ADQL/s and the way of mapping from ADQL/s to ADQL/x is described briefly.

ADQL/s grammar is described in an extended BNF. 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 should be ignored otherwise stated.

ADQL/s grammar consists of **CORE** grammar and **EXTENSIONS** to it. The CORE grammar is defined aiming for interoperability among all the data services, so it provides just minimum functionality (selection and projection in the relational database term) so that a service that conforms to the grammar is easily set up. The EXTENSIONS is defined to enable the enhancement of service functionality. All VO services accepting ADQL queries **MUST** conform to the **CORE** specification.

ADQL-s Core Syntax

- Syntax of core ADQL is as follow:

```
[ <comment> ]
SELECT [ TOP <number> ]
( [ <table_alias> . ] "*" | count "(" "*" ")"
| <column_list> )
FROM <table_name> [ AS ] <table_alias>
[ WHERE  <condition_core> ]
[ <comment> ]
```

- `SELECT` statement defines a query to a specified table. As a result of this query, a subset of the table is returned. The order of the rows MAY be arbitrary. The order of columns to return SHOULD be the same as the order as specified in the `<column_list>` or the order defined on the original table if `"*"` is specified.

- `TOP <number>` construct is used to specify the maximum number of rows to return. Any arbitrary rows MAY return.

- Selected data are either column values or the number of selected rows. An expression like `a+b` is not supported in the core syntax, which is supported as an extension.

- `<column_list>` is a list of columns to return, which is specified in a standard SQL form, that is a list of comma separated column references. A column name MAY be aliased, and MAY be qualified by a table alias name. Note that the table name SHOULD NOT be used to qualify the column name. The column reference is expressed as:

```
[<table_alias>.] <column_name> [[ AS ] <alias_name>]
```

- "*" represents all the columns, and MAY be qualified by a table alias name.

- `Count(*)` is an aggregate function which returns the number of selected rows.

- Exactly one table SHALL be specified in the FROM clause. A table is specified by a table name followed by an alias name. The table alias name MUST be supplied.

- Selection condition `<condition_core>` is specified by a regional condition and/or a non-regional condition. When both of the regional and non-regional conditions are specified they SHALL be connected by "AND" logical operator.

- Non-regional condition is one of the following SQL boolean value expression:

```
- <B> OR <B>
- <B> AND <B>
- NOT <B>
- <E> <comparison_op> <E>
- <E> [ NOT ] BETWEEBN <L> and <L>
- <E> [ NOT ] LIKE <pattern>
- <E> [ NOT ] IN "(" <L> [, <L> ]* ")"
- <E> IS [ NOT ] NULL
- <boolean_value_function>
- "(" <boolean_value_expression> ")"
```

  where `<B>` is a boolean value expression, `<E>` is any type of value expression, and `<L>` is a literal value. `<comparison_op>` supported in the Core syntax is basic comparison operators listed in table 2. Wild cards that are used for expressing a string pattern of a LIKE predicate are "_" and "%". "_" matches a single arbitral character and "%" matches arbitrary number (>=0) of characters.

- Regional condition SHOULD be supported for a table that has a set of columns representing a position in a two dimensional space. Those columns SHOULD have

metadata related to their coordinate frame.

- Allowed region shapes for a regional condition are BOX and CIRCLE. The region is expressed by a region shape type, a coordinate frame, an optional unit of coordinates and region sizes, center coordinates, and region sizes. Two region sizes measured along the two coordinate directions are specified in the case of BOX region, while a radius is specified in the case of CIRCLE region. The unit of the box sizes or radius is the same as the one of the center coordinate. The syntax of the regional condition is:

```
REGION('BOX <frame> [<unit>] <c1> <c2>
                                <size1> <size2>')

REGION('CIRCLE <frame> [<unit>] <c1> <c2>
                                        <radius>')
```

- <frame> is a frame name defined in the STC specification. A table that supports regional search SHALL accept at least one of the frame names, which SHALL be provided through a metadata query. A list of all the supported frame names SHOULD also be provided through a metadata query. Several examples of the frame expression are:

```
<frame> = FK4 [<epoch>] | FK5 [<epoch>]
        | ECLIPTIC [<epoch>] | ICRS | GALACTIC_II
| …

<epoch> = J2000 | B1950 | …
```

- An aggregate function supported in core ADQL is count(*).
- A function support is not mandatory in core ADQL specification, however RECOMMENDED to support basic functions list in table 5.
- Comments SHALL be supported using the **/* … */** syntax to delimit comments. Comments are only supported before and after the main query.

ADQL-s extension syntax

- Syntax of the extended ADQL is:

```
[ <comment> ]
SELECT [ ALL | DISTINCT ]
    [ OFFSET <unsigned_integer> ]
    [ TOP <unsigned_integer> ]
    <selection_list_ext>
    [ INTO <store_reference> ]
    FROM <table_list>
    [ WHERE <search_condition_ext> ]
    [ GROUP BY <group_item_list> ]
    [ HAVING <search_condition_ext> ]
    [ ORDER BY <order_list> ]

    [ <comment> ]
```

- ADQL `SELECT` statement defines a query to a derived table specified in the `FROM` clause. As a result of this query, a subset of the table is returned. The order of the rows MAY be arbitrary unless `ORDER BY` clause is specified. The order of columns to return SHOULD be the same as the order as specified in the `<column_list>` or the order defined on the original table if "*" is specified.
- SQL standard of **`ALL`** and **`DISTINCT`** construct is defined as an extension.
- **`OFFSET n`** construct is defined as an extension to skip the first n-records. It is RECOMMENDED to use the `OFFSET` keyword along with the `ORDER BY` keyword, since it is meaningless to use this if the order of rows is not specified.
- **`TOP n`** construct is used to return first n-rows from the offset position specified by a `OFFSET` keyword. The combination of `TOP`, `OFFSET` and `ORDERBY BY` can be used to retrieve a result by splitting it to smaller peaces. It is recommended to order the record by primary keys, since most of the database management system generates an index for the primary keys as a default, and gives a better response.
- Selection list MAY include any value expression, such as `a+b`, `a-b`, `a*b`, `a/b`, `+a`, `-a`, `a*(b+c)`, where `a`, `b` and `c` represent a column, function or other valid value expressions.
- **INTO** construct is defined as an extension to specify the VOSpace location where the result is stored. The exact syntax of the VOSpace location is defined in a separate specification.

```
SELECT g.* INTO VOS:/JHU/gal FROM galaxy g
    WHERE g.redshift > 3.5
```

- Multiple tables separated by commas MAY be specified at a FROM clause.
- SQL standard of table join construct is defined as an extension. The following join types are supported:
  - **CROSS JOIN**
  - **INNER JOIN**
  - **OUTER LEFT, RIGHT, FULL JOIN**
  - **NATURAL JOIN**
  - **USING JOIN**
- In addition to the CORE search condition, following SQL standard predicates are defined as an extension
  -
  - **EXIST**
  - **ALL**

  - **SOME**

- SQL standard **GROUP BY** clause is defined as an extension.
- SQL standard **HAVING** clause is defined as an extension.
- SQL standard **ORDER BY** clause is defined as an extension.
- **#UPLOAD** keyword MAY be used at a FROM clause to represents votables. Using this syntax, table join between internal tables and external votables can be described. A votable name, which is an attribute of a TABLE element, may be followed to distinguish the multiple tables in votables. A syntax to refer to the votable and it example are:

```
UPLOAD [ <votable_name> ] [ AS ] <alias>
FROM galaxy g, UPLOAD name1 vot1,
      UPLOAD name2 vot2
```

- Subquery MAY be used at a FROM clause.
- Table name qualified by a service identifier MAY be supported to specify a table that belongs to another SkyNode service. A short name of the service MAY be specified, however note that it does not guaranty the uniqueness in the VO.

```
[( <service_identifier> | <short_name> ) : ]
<table_name>
```

  e.g. `ivo://jvo/sxds:tableName`

  e.g. `sxds:tableName`

- XPath expression in selection list and selection criteria MAY be supported. Square brackets ([,]) and standard operators such as parent are NOT supported. An example of a valid query of this form would be:

```
SELECT /Resource/Contact/Name
     FROM Resource
     WHERE /Resource/Type LIKE 'catalog'
```

- Supported extensions SHOULD be provided through a metadata query using extension IDs that are listed in table 1.

Extension ID	Description of the extension
DST	**ALL** or **DISTINCT** keyword.
OFF	**OFFSET** keyword.
EXP	Expressions (the four fundamental rules of arithmetic, unary operation by + and -, and closed expression) in a selection list.

INT	**INTO** keyword.
TML	Comma separated multiple tables in a FROM clause.
TJN	Table joins: **CORSS JOIN**, **INNER JOIN**, **OUTER LEFT**, **RIGHT**, **FULL JOIN**, **NATURAL JOIN** and **USING JOIN**.
TID	A table name qualified by a service identifier.
VOT	**#UPLOAD** keyword to specify VOTables.
TSQ	A derived table with a sub-query in a FROM clause
EXI	**EXISTS**, **ALL**, **SOME** predicates in a WHERE clause.
GBY	**GROUP BY** clause.
OBY	**ORDER BY** clause.
HVN	**HAVING** clause.
FUN	BASIC functions.
CON	Concatenation operator "|" for character and character array data type.
DOP	Data/Time operator
NAR	Array of numeric data types
ITV	Time interval data type
GEO	Geometrical data type
AGR	All the aggregate functions

Table 1: ADQL syntax extensions

Keyword, Identifier and delimited identifier

ADQL/s is constituted of a reserved and un-reserved keyword, identifier, delimited identifier, and literal. A reserved keyword has a special meaning in ADQL and cannot be used as an identifier. A un-reserved keyword has a special meaning in specific contexts and can be used as an identifier in the other contexts. An identifier and a delimited identifier are used to express a table name, column name, service specific function and data type name, and alias name. A literal is used to express a constant value of each data type.

- Reserved keywords MUST NOT be used as an identifier.

- A keyword and an identifier SHALL begin with a letter {a-z}. Subsequent characters SHALL be letters, underscores '_' or digits {0-9}.
- A keyword and an identifier are case insensitive.
- Reserved keywords are:

  ```
  SELECT, ALL, DISTINCT, TOP, OFFSET, INTO, FROM, WHERE,
  GROUP, BY, HAVING, ORDER, AS, UPLOAD, CROSS, JOIN,
  NATURAL, INNER, OUTER, LEFT, RIGHT, FULL, ON, USING,
  IN, OVERLAPS, COVERS, TRUE, FALSE, BETWEEN, LIKE, IN,
  ASC, DESC, NOT, AND, OR, SHORT, INT, LONG, FLOAT,
  DOUBLE, CHAR, DATE, TIME, TIMESTAMP, BOOLEAN, CHAR,
  TIME_INTERVAL, POSITION_2D, REGION_2D.
  ```

- Un-reserved keywords are:

  ```
  COUNT, MIN, MAX, AVG, SUM, ACOS, ASIN, ATAN, ATAN2,
  COS, COT, SIN, TAN, ABS, CEILING, DEGREES, EXP, FLOOR,
  LOG, LOG10, MODE, PI, POWER, RADIANS, SQRT, RAND,
  ROUND, TRUNCATE, GC_DISTANCE, POSITION, CIRCLE, BOX,
  JOIN_CHI2, JOIN_DISTANCE, INFO_SPECS, INFO_TABLES,
  INFO_COLUMNS, INFO_FRAMES, INFO_FUNCTIONS.
  ```

- Identifier that includes a non-permitted character, that is case-sensitive or that matches the ADQL keywords SHALL be delimited by delimiters. Double quotations are used a delimiter. Some examples are shown below:

  ```
  SELECT "select" FROM table t
  SELECT "O/Fe" FROM table t
  SELECT * FROM "2mass" t
  ```

- The way of writing a delimiter within a delimited identifier is to repeat two adjacent delimiters. E.g. "abc""def" is a literal expression of abc"def.
- Use of a delimited identifier is not encouraged.

Data type

ADQL defines five numeric data types (short, int, long, float, double), one character data type (char), four date and time data types (date, time, timestamp, time_interval), one boolean data type (boolenan), two geometric data type (position_2d, region_2d), and array of numeric and character data types. A service specific data type is also allowed to be used. The list of the defined data types are shown in Table 2. The time interval data type is defined as an "interval" data type in the standard SQL, however time_interval is used in ADQL to distinguish from spatial interval. The geometric data types are not part of the standard SQL, however these are introduced to express the spatial search condition in more flexible way than "Region" function.

- A numeric, character, boolean and array of character data types SHALL be supported.

- A time_interval, geometric and service specific data type MAY be used.
- Literal expressions of numeric data types are:

```
<digits>
<digits> . [<digits>] [ e [ + | - ] <digits>]
[<digits>] . <digits> [ e [ + | - ] <digits>]
<digits> e [ + | - ] <digits>
```

- A literal expression of boolean is either ADQL keyword TRUE or FALSE.
- A literal expression of character and character array data types are a character or a string delimited by single quotations.
- Literal expressions of the other data types are described by a type name followed by a string expression delimited by single quotations. The data type name MAY be omitted if there is no ambiguity as to the type that the value must be in the context.

```
[ <type_name> ] ‘ <data_string> ’
```

- String expression data/time data types are:

```
<date_string> =
     <four_digits> - <two_digits> - <two_digits>

<time_string> =
     <two_digits> : <two_digits> : <two_digits>
     [ . <digits> ]

<timestamp_string> =
     <date_string> T <time_string> | <Julian_day>

<time_interval_string> =

     <digit> <unit> [ <digit> <unit> ]*
```

  where <unit> is either second, minute, hour, day, week, month, or year. Some examples:

```
date ‘2006-06-20’
time ‘23:40:24.56’
timestamp ‘2006-06-20T23:40:24.56’
timestamp ‘2453907.486111111’
time_interval ‘30 days 15 hours’
```

- The timezone SHOULD be the GMT.
- The operations allowed for the date/time data types are summarized in table 3.
- String expression of geometric data types are:

```
<position_string> = POSITION <frame> <pos2>
<region_string> = CIRCLE <frame> <pos2> <radius>
```

```
| BOX <frame> <pos2> <size2>
```

- The unit of the coordinates, region radius and sizes are always a “degree”. Some examples:

```
POSITION_2D ‘POSITION FK5 120.3 +20.0’
REGION_2D ‘CIRCLE ICRS 120.3 +20.0 1.0’
REGION_2D ‘BOX GALACTIC_II 30.0 45.3 1.0 1.0’
```

- The operations allowed among the geometric data types are:

```
<position> IN <region>
<region> IN <region>
<region> OVERLAPS <region>
<region> COVERS <region>
```

  Some examples are:

```
SELECT * FROM catalog t WHERE Position(ra,dec) IN
  Circle(‘FK5’,20,30,1.0)
```

```
SELECT t1.*, t2.*
   FROM image t1, catalog t2
   WHERE t1.region IN
   Cirlce(‘FK5’,t2.ra,t2.dec,1.0)
```

- Service specific data type MAY be defined and used. The literal expression of the service specific data type SHALL be described by a data type name followed by a string expression delimited by single quotations as follows:

```
<type_name> ‘ <value> ’.
```

- Every column SHALL be assigned one of the Core data types, the extension data types or the service specific data types. The basic binary and unary operators shown in the table SHALL be supported.

Name	Description	Basic binary operators (upper = arithmetic / logical operator, lower = comparison operator)	Basic unary operators	Basic Predicate	Optional operators / predicates	Core and Extension ID
short	signed two-byte integer	+ - * / = <> <= < >= >	+ -	BETWEEN IN		Core
int	signed four-byte integer	+ - * / = <> <= < >= >	+ -	BETWEEN IN		Core
long	signed eight-	+ - * /	+ -	BETWEEN		Core

	byte integer	= <> <= < >= >		IN		
float	Single precision floating-point number	+ - * / = <> <= < >= >	+ -	BETWEEN IN		Core
double	Double precision floating-point number	+ - * / = <> <= < >= >	+ -	BETWEEN IN		Core
char	one-byte character	 = <> <= < >= >		LIKE IN	|	Core,CON (concatenation operator support)
date	calendar date	 = <> <= < >= >	+(*)	BETWEEN IN	+, -(*)	Core,DOP (date/time operator support)
time	time of day	 = <> <= < >= >	+(*)	BETWEEN IN	+, -(*)	Core,DOP
timestamp	date and time	 = <> <= < >= >		BETWEEN IN	+, -(*)	Core,DOP
boolean	Logical boolean	AND OR =	NOT			Core
char*, char[n]	array of character	 = <> <= < >= >		LIKE IN	|	Core,CON
short[n], int[n], long[n], float[n], double[n]	array of number					NAR (array of numeric data type support)
interval	time interval				+ - * /(*)	ITV (interval data type support)
position_2D	position on a 2D plain				IN OVERLAPS COVERS	GEO (astronomical coordinate data types and operators support)
region_2D	region on a 2D plane				IN OVERLAPS COVERS	GEO

Table 2: Data types of ADQL. (*) Refer table 3 for actual allowed operations.

A \ B	date	time	timestamp	time_interval	int	double
date	-	+		+ -	+ -	+ -
time	+	-		+ -		
timestamp			-	+ -		
time_interval	+	+	+	+ -	*	* /
int				*		
double				*		

Table 3: Matrix of allowed data/time operations. Allowed operators for A <op> B are shown. The result of operation follows the SQL standard.

Aggregate Function

ADQL defines six aggregate functions. The functionality of the aggregate functions is the same as the standard SQL. Count(*) is a mandatory function and SHOULD be supported. The others are optional function and MAY be supported. If the optional aggregate function is supported, extension ID of "AGR" SHOULD be provided as ADQL extension metadata.

Function	Argument type	Return type	description	Core or Extension ID
Count(*)		long		Core
Count([ALL | DISTINCT] expression)	any	long		AGR
Min([ALL | DISTICT] expression)	Numeric char char[n] date/time	Same as the argument type		AGR
Max([ALL | DISTINCT] expression)	Numeric char char[n] date/time	Same as the argument type		AGR
Sum([ALL | DISTINCT] expression)	Numeric	Long for argument of integer type, double for floating type		AGR
Avg([ALL | DISTINCT] expression)	number	double		AGR

Table 4: ADQL aggregate functions.

Function

ADQL defines functions listed in table 5.

- Support of the basic functions is not mandatory, however they are RECOMMEDED to be supported.
- Support of the advanced functions is not mandatory, and MAY be supported. Extension IDs are not assigned to those functions, instead the supported functions SHOULD be provided as function metadata.
- **`Position`** function takes one optional frame parameter and two double type coordinates parameters, and returns a position_2d type value corresponding to the specified position.

  ```
  Position"(" [ '<frame>' ] <coord1>, <coord2>")"
  ```

  where, `<coord1>` is a double value expression for first coordinate of a position in two dimensional space, and `<coord2>` is for its second coordinate.

  If coordinates parameters are specified by columns for which frame metadata is defined, the frame parameter MAY be omitted. Otherwise the frame parameter SHOULD not be omitted. The frame parameter SHOULD be ignored if the coordinate parameters are assigned frame metadata. The two coordinate parameters SHOULD have a common frame metadata. As a shorthand, the function name "Position" MAY be omitted, which reduces the complexity to write a region using a position function. The following examples shows valid and invalid usages:

  `Position(ra, dec)` (valid),
  `Position('FK5 J2000', 20.0 +10.0)` (valid),
  `Position('ICRS', ra, dec)` (valid、but 'ICRS' is ignored),
  `Position(dec, ra)` (invalid),
  `Position(ra, mag)` (invalid),
  `Position('ICRS', mag, mag)` (valid, but result might be meaningless)

  where `ra`, `dec`, `mag` represent columns of right ascension, declination and magnitude of brightness, respectively, and frame metadata of "FK5 J2000" is assigned to the ra and dec column.

- **`GC_distance`** function calculates a great circle distance between two positions on a spherical plane. The two positions are specified in the following forms:

  ```
  GC_distance"(" ['<frame>'], <coord1>, <coord2>,
   ['<frame>'], <coord1>, <coord2> ")"
  | GC_dinstance"(" <position>, <position> ")"
  ```

  In the first form, the first three parameters including the optional frame parameter specify the first position and the next three parameters specify

another position. The frame parameter MAY be omitted or ignored under the same condition with the Position function. The requirement for the coordinate parameters (same frame) are also the same as those for the Position function.

In the second form, the two positions are specified by parameters of position_2d data type.

- **Circle** and **Box** functions return a value of region_2d data type that corresponds to the specified region of circle and box shapes, respectively.

```
Circle "(" <position>, <radius> ")"
Circle "(" ['<frame>',] <coord1>, <coord2>,
       <radius> ")"
Box ( <position>, <size1>, <size2> )
Box ( [<frame>,] <coord1>, <size1>, <size2> )
```

- **Join_chi2** function is used to join multiple tables based on the angular distances of objects' positions and on the errors of the coordinates. The chi square is calculated for a combination of objects of each table, and if it is less than a specified value the function returns true. Otherwise it returns false.

```
Join_chi2 "(" <tables>, <sigma> ")"
<tables> = '<table>, [!]<table>, [,
[!]<table> ]*'
```

Where <sigma> is the maximum value of chi-square to select records from the cross joined tables. The exact algorithm is described in Appendix.

- **Join_distance** function is used to join multiple tables based on only the angular distance between the two specified positions. If the two specified position is neare than a specified distance, it returns true. Otherwise it returns false.

```
Join_distance "(" [<frame>], <coord1>, <coord2>,
  [<frame>],<coord1>, <coord2>, <max_distance>
")"

Join_distance "(" <position>, <position>,
  <max_distance> ")"
```

where <max_distance> is a maximum angular distance in degree to select records.

Name	Return type	Comment	Extension ID
acos(x)	double	Basic function. Inverse cosine.	BFN (Basic function support)
asin(x)	double	Basic function. Inverse sine.	BFN
atan(x)	double	Basic function. Inverse tangent.	BFN
atan2(x,y)	double	Basic function. Inverse tangent of x/y.	BFN
cos(x)	double	Basic function. Cosine.	BFN

cot(x)	double	Basic function. Cotangent.	BFN
sin(x)	double	Basic function. Sine.	BFN
tan(x)	double	Basic function. Tangent.	BFN
abs(x)	double	Basic function. Absolute value.	BFN
ceiling(x)	double	Basic function. Smallest integer not less than argument.	BFN
degrees(x)	double	Basic function. Radians to degrees.	BFN
exp(x)	double	Basic function. Exponetial.	BFN
floor(x)	double	Basic function. Larget integer not greater than argument.	BFN
log(x)	double	Basic function. Natural logarithm.	BFN
log10(x)	double	Basic function. Base 10 logarithm.	BFN
mod(x, y)	double	Basic function. Remainder of y/x.	BFN
pi()	double	Basic function. Pi constant.	BFN
power(x, y)	double	Basic function. X raised to the power of Y.	BFN
radians(x)	double	Basic function. Degree to radians.	BFN
sqrt(x)	double	Basic function. Square root.	BFN
rand()	double	Basic function. Random value between 0.0 and 1.0.	BFN
round(x, n)	double	Basic function. Round to nearest integer.	BFN
truncate(x, n)	double	Basic function. Truncate to n decimal places.	BFN
GC_Distance(p1, p2) or GC_Distance(frame1, x1, y1, frame2, x2, y2)	double	Advanced function. See text.	Not assigned.
Position(x1, x2) or Position(frame, x1, x2)	position_2d	ADVANCED. See text.	Not assigned.
Circle(x1, x2, z) or Circle(p, z)	region_2d	ADVANCED. See text.	Not assigned.
Box(x1, y1, z1, z2) or Box(p, s1, z2)	region_2d	ADVANCED. See text.	Not assigned.
join_chi2(s, x)	boolean	ADVANCED. See text.	Not assigned.
join_distance(x1, y1, x2, y2, z) or join_distance(p1, p2, z)	boolean	ADVANCED. See text	Not assigned.

Table 5: ADQL Functions. Where x*, y* and z* represents double, n integer, p* position_2d and s array of character.

Metadata Query

Metadata about grammar specifications, tables, columns, functions, supported frames and so on SHALL be able to be queried by an ADQL core syntax, which means metadata tables for them exits. The name of the metadata tables are prefixed by "`INFO_`". Metadata of all the table including metadata table as well as data tables SHALL be registered in a `INFO_TABLES` table. For an example, following query returns metadata of a table named "tableName".

```
SELECT *
    FROM INFO_TABLES
    WHERE table_name = 'tableName'
```

This document defines mandatory metadata tables ant contents of them. Any service specific metadata MAY be added to any of the mandatory metadata tables and/or service specific metadata table.

- Metadata table "`INFO_SPECS`" SHALL have the following columns:
  - **adql_version**: (char*) Supported ADQL version number.
  - **extention_id**: (char[3]) Supported ADQL extension Ids
- Metadata table "`INFO_TABLES`" SHALL have the following columns:
  - **table_name** : (char*) name of a table.
  - **description** : (char*) description of the table.
  - **max_records** : (long) maximum number of returned records if known, otherwise –1.
  - **row_count** : (long) number of records if known, otherwise -1.
  - **rank** : (int) relative importance of the table if know, otherwise –1. Put a larger value for a more important table
  - **ucd** : (char*) ucd of the table.
  - **class** : (char*) class name of the table. The following class is defined as a canonical name: "general", "objects", "image", "spectrum".
  - **pos_coord1**: Name of a column that is referred to as a first coordinate in a region search.
  - **pos_coord2**: Name of a column that is referred to as a second coordinate in a region search.
  - **last_modified**: (timestamp) last modified time of contents of a table.
- Metadata table "`INFO_COLUMNS`" SHALL have the following columns:
  - **column_name**: (char*) name of a column. [REQ]
  - **table_name**: (char*) name of a table to which the column belong to. [REQ]

  - **description**: (char*) description of the column.
  - **data_type**: (char*) data type of the column. [REQ]
  - **unit**: (char*) unit of the column value.
  - **arraysize**: (char*) dimension of the column value. Positive integer or "*". [REQ]
  - **precision**: (char*) precision of the column value.
  - **ucd**: (char*) UCD of the column.
  - **utype**: (char*) UTYPE of the column.
  - **ordinal_position**: (char*) position number (>=0) of the column in the table. [REQ]
  - **primary_key**: (bolean) true if a column is primary key**.** Multiple primary keys are allowed in a table.
  - **rank**: (int) relative importance of the table if known, otherwise –1. Put a larger value for a more important column.

- Metadata table "`INFO_FUNCTIONS`" SHALL have the following columns:
  - **function_name**: name of a function.
  - **description**: description of the function.
  - **return**: return data of the function.
  - **arguments**: List of arguments data types.
- Metadata table "`INFO_FRAMES`" SHALL have the following columns:
  - **table_name**: name of a table.
  - **frame**: supported frames for the tables in regional query.

Version information

ADQL/x documents SHALL contain a version identifier for the version of ADQL. This will start as 1.0. The version number is a dot separated string of numbers. The version number is included in the document solely so the receiving node may decide if it wishes to deal with the document or to return an exception. This is assumed to only come into use at some later stage when there may be a major version change causing some possible incompatibility between versions.

ADQL example

An **ADQL/s** might be as follows:

```
SELECT a.objid, a.ra, a.dec
```

```
       FROM Photoprimary a
       WHERE Region('CIRCLE FK5 181.3 -0.76 6.5')
```

This would be represented in **ADQL/x** as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<Select xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.ivoa.net/xml/ADQL/v1.?">
  <SelectionList>
    <Item xsi:type="columnReferenceType" Table="a" Name="objid" />
    <Item xsi:type="columnReferenceType" Table="a" Name="ra" />
    <Item xsi:type="columnReferenceType" Table="a" Name="dec" />
  </SelectionList>
  <From>
    <Table xsi:type="tableType" Name="Photoprimary" Alias="a" />
  </From>
  <Where>
    <Condition xsi:type="regionSearchType">
      <Region xmlns:stc="http://www.ivoa.net/xml/STC/stc-v1.30.xsd"
         xsi:type="stc:STCRegion"
         xlink:href="ivo://STClib/CoordSys#UTC-FK5-TOPO"
         id="UTC-FK5-TOPO">
         <stc:Circle coord_system_id="UTC-FK5-TOPO">
            <stc:Center unit=”deg”>
                 <stc:C1>181.3<stc:C1><std:C2>-0.76<stc:C2>
            </stc:Center>
            <stc:Radius>6.5</stc:Radius>
         <stc:Circle>
      </Region>
    </Condition>
  </Where>
</Select>
```

ADQL XSD

The XML schema for ADQL is found at http://www.ivoa.net/xml/ADQL/ADQL-v1.?.xsd.

Changes from previous versions

- None. This is the first release.

References

- IVOA SkyNode Interface 1.? http://www.ivoa.net/Documents/latest/SNI.html
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ADQL XML schema. http://www.ivoa.net/xml/ADQL/ADQL-v1.?.xsd

- Space-Time Coordinates for the Virtual Observatory Version 1.30
  http://www.ivoa.net/xml/STC/STCregion/v1.30

Appendix A ADQL Grammar

A- 1 BNF for Core Query Syntax

```
    <select_core> = [ <comment> ]
    SELECT [ TOP <unsigned_integer> ]
    ( [ <table_alias> . ] "*" | count "(" "*" ")"
    | <column_list> )
     FROM <table_name> [ AS ] <table_alias>
     [ WHERE <condition_core> ] [ <comment> ]

 <comment> = / "*" <string> "*" /

 <column_list> = <column> [, <column> ]*
<column> = [ <table_alias> "."] <column_name>
     [ [ AS ] <alias> ]

 <condition_core> =
     <region_function> [ AND <sql_condition> ]
     | <sql_condition> [ AND <region_function> ]
     [ AND <sql_condition> ] ]
 <region_function> = REGION "(" ' <region_string> ' ")"
<region_string> = <box_string> | <circle_string>

 <box_string> = BOX <frame> [ <unit> ] <coord1> <coord2>
     <size2> <size2>

 <circle_string> = CIRCLE <frame> [ <unit> ] <coord1>
                <coord2> <radius>

 <sql_condition> = <boolean_value>

 <boolean_value> = <boolean_term>
     | <boolean_value> OR <boolean_term>

 <boolean_term> = <boolean_factor>
     | <boolean_term> AND <bololean_factor>

 <boolean_factor> = [ NOT ] <boolean_primary>

 <boolean_primary> = <comparison_predicate>
     | <null_predicate> | <between_predicate>
     | <in_predicate> | <like_predicate>
     | <boolean_value_function>
```

```
      | "(" <boolean_value> ")"
<comparison_predicate> = <value_expression>
      <comparison_operator> <value_expression>

   <comparison_operator> =
      "=" | ">" | "<" | "<" ">" | ">" "=" | "<" "="

   <between_predicate> = <value_expression>
[ NOT ] BETWEEN <literal> AND <literal>
<in_predicate> = <value_expression> [ NOT ] IN
      "(" <literal> [, <literal> ]* ")"
<like_predicate> = <value_expression>
       -  [ NOT ] LIKE <string_pattern>
<value_expression> = <value_term>
      | <value_expression> ( + | - ) <numeric_value_term>
<value_term> = <value_factor>
      | <value_term> ( "*" | / ) <value_factor>

   <value_factor> = [ + | - ] <value_primary>

   <value_primary> = <parenthesised_value_expression>
      | <unparenthesised_value_primary>

   <parenthesized_value_expression> =
      "(" <value_expression> ")"

   <unparenthesised_value_primary> =
<column_reference> | <literal> | <function>
<aggregate_function> = count(*)
```

A- 2 BNF for Extended ADQL/s Syntax

```
   <select_e> = [ <comment> ]
      SELECT [ ALL | DISTINCT ]
      [ OFFSET <unsigned_integer> ]
      [ TOP <unsigned_integer> ]
      <selection_list_e>
      [ INTO <store_reference> ]
      FROM <table_list>
      [ WHERE <search_condition_e> ]
      [ GROUP BY <group_item_list> ]
      [ HAVING <search_condition_e> ]
      [ ORDER BY <order_list> ] [ <comment> ]

   <selection_list_e> = [ <table_alias> . ] "*"
      | <aliased_select_item_list>
```

```
   <aliased_select_item_list> =
       aliased_select_item ( , <aliased_select_item> )*

   <aliased_select_item> =
       <value_expression> [ [ AS ] <alias> ]
       | <xpath_expression> [ [ AS ] <alias> ]

   <xpatch_exprssion> = / relative_element_path
       [ / @ attribute_name ]

   <relative_element_path> =
       <element_name> [ / <element_name> ]*

<aliased_table_list = <aliased_table>
                  [, <aliased_table> ]*
<aliased_table> = <alased_table_primary> |
       <aliased_derived_table>
   <aliased_table_primary> = <db_table> | <votable>

   <db_table> = [ ( <service_identifier> |
       <short_name> ) : ] <table_name> [ AS ] <alias>

   <votable> = UPLOAD [ <table_name> ] [ AS ] <alias>

<aliased_derived_table> = ( <cross_join>
       | <conditional_join> | <natural_join>
       | "(" <select_e> ")" ) [ AS ] <alias>

   <cross_join> = <aliased_table> CROSS JOIN
       <aliased_table_primary>
 <conditional_join> = <aliased_table> [ <join_type> ] JOIN
       <aliased_table_primary> ( ON <comparison_predicate>
       | USING "(" <column_name_list> ")" )
       <natural_join> = <aliased_table> NATURAL
[ <join_type> ]
       JOIN <aliased_table_primary>

   <join_type> = INNER | ( LEFT | RIGHT | FULL ) [ OUTER ]
   <column_name_list> = <column_name> [ , <column_name> ]*

       <search_condition_e> =
       <region_function> [ AND <sql_condition_e> ]
       | <sql_condition_e> [ AND <region_function> ]
       [ AND <sql_condition> ] ]

   <sql_condition_e> = <boolean_value_e>
```

```
   <boolean_value_e> = <boolean_term_e>
       | <boolean_value_e> OR <boolean_term_e>

   <boolean_term_e> = <boolean_factor_e>
       | <boolean_term_e> AND <bololean_factor_e>

   <boolean_factor_e> = [ NOT ] <boolean_primary_e>

   <boolean_primary_e> = <comparison_predicate_e>
       | <between_predicate> | <in_predicate>
       | <like_predicate> | <boolean_value_function>
       | <exist_predicate> | <some_predicate>
       | <all_predicate>

   <comparison_predicate_e> = <value_expression>
       <comparison_operator_e> <value_expression>

   <comparison_operator_e> = <comparison_operator>
       | "|" | IN | OVERLAPS | COVERS

   <group_item_list> =
       <value_expression> [ , <value_expression> ]*

                   <order_list> = <order_item> [ ,
<order_item> ]*

   <order_item> = <value_expression> [ ASC | DESC ]
```

Appendix B Algorithm of join_chi2 function

An example for the cross-matching algorithm is a probabilistic calculation that minimizes the chi-square parameter as defined by:

$$\chi^2 = \frac{1}{2}\sum_n \alpha_n \left[(x - x_n)^2 + (y - y_n)^2 + (z - z_n)^2\right] - \frac{1}{2}\lambda\left[x^2 + y^2 + z^2 - 1\right]$$

where x, y, z are the Cartesian coordinates corresponding to the best estimate of ra and dec, **a** is a weighting parameter calculated from the astrometric precision of the survey, and λ is the Langrange multiplier in the minimization to ensure that the (**x,y,z**) is a unit vector.

We compute four cumulative quantities at each cross-identification step – these are

$$a = \sum \frac{1}{\sigma_i^2}, \quad a_x = \sum \frac{x_i}{\sigma_i^2}, a_y = \sum \frac{y_i}{\sigma_i^2}, a_z = \sum \frac{z_i}{\sigma_i^2}.$$

The best position is given by the direction of (a_x, a_y, a_z). The log-likelihood at that point is given by

$$\chi^2 = a - \sqrt{a_x^2 + a_y^2 + a_z^2}$$

- This is divided by the number of surves, and compared to the tolerance. If a Tuple's log-likelihood exceeds this threshold, it return false. Otherwise true.