



International

Virtual

Observatory

Alliance

IVOA Astronomical Data Query Language Version 1.01

IVOA Working Draft 24 June 2005

This version:

1.01: <http://www.ivoa.net/Documents/WD/ADQL/ADQL-20050624.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:

Masatoshi Ohishi and Alex 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 SQL92. This document describes the SQL92 grammar, as well as the

restrictions and extensions to SQL92 necessary to support astronomy specific operands such as Region and XMATCH. The XML schema representing the grammar is placed in the IVOA XML schema area.

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 e-mails on the mailing list. The editors express their appreciation for many valuable contributions Wil O'Mullane, Naoki Yasuda, Yuji Shirasaki, Clive Page, Bob Mann, Martin Hill, and many others.

Contents

Abstract	1
Status of this document.....	2
Acknowledgments	2
1 Introduction	3
2 Astronomical Data Query Language (ADQL)	3
2.1 Restrictions on SQL92	4
2.2 Extensions to SQL92.....	5
2.3 Version information	7
2.4 Regions	7
3 ADQL example	8
4 ADQL XSD	9
5 Changes from previous versions.....	9
6 References.....	9
Appendix ADQL Grammar.....	10

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 data services (SkyNodes). IVOA has developed several standardized protocols to access astronomical databases, e.g., SIAP for image data and SSAP for spectral data. Current ADQL (ADQL 1.0) is designed to access astronomical catalog data only through the SkyNode Interfaces. The work to integrate SIAP, SSAP and ADQL is under progress toward the future version of ADQL.

ADQL 1.0 is based on Structured Query Language (SQL), especially on SQL92. The VO has a number of tabular data sets and many of them are stored in relational databases (RDBs), making SQL a convenient access means. .

The mechanism of passing a query to a node is described in the SkyNode Interface document **Error! Reference source not found.** that is also developed in the VOQL WG of the IVOA. It should be noted that the SkyNode Interface is also related to Data Access Layer WG of the IVOA.

To access some current implementations of SkyNodes, visit , e.g., at OpenSkyQuery.net. The OpenSkyQuery portal is an example of how astronomers can use ADQL to query a federation of astronomical databases which have been published as SkyNodes.

2 Astronomical Data Query Language (ADQL)

ADQL is based on a subset of SQL which has been extended to support queries which are specific to astronomy.

ADQL has two forms or representations:

- **ADQL/s** : A String form based on SQL92 **Error! Reference source not found.** and conforming to the ADQL grammar in Appendix to this document. Some non standard SQL extensions have been added to support distributed astronomical queries; and
- **ADQL/x** : An XML document conforming to the ADQL schema (XSD) included in Section 4. The XML document is the mechanism used to pass a query to the SkyNode Web service Interface.

ADQL/s and **ADQL/x** are translatable to each other without loss of information.

Services for translation of SQL (ADQL/s) to XML (ADQL/x) and back may be found at <http://openskyquery.net/ADQLTranslator>. Please note that this site corresponds to ADQL version 0.7.4, but it will be updated soon after the current specification is released.

Since ADQL is similar in semantics to SQL, the requirements below list differences or special considerations only.

2.1 Restrictions on SQL92

The formal notation for syntax of computing languages is often expressed in the “Backus Naur Form” BNF¹. BNF is used by popular tools such as LEX and YACC² for producing parsers for a given syntax. Appendix to this document provides the YACC type grammar for ADQL/s.

The BNF exactly defines the form of SQL92 which is ADQL/s. In essence this is any valid SQL statement. However ADQL has restrictions described below.

2.1.1 Built-in Functions

In ADQL built-in functions which are defined on the server system may be called. These would include, e.g., a function to provide great circle distance, converter such as from sexagesimal to decimal, and unit converters. The SkyNodeInterface specification also defines a method by which all functions available on the server may be discovered. If a user knows that certain functions exist in the target system (SkyNode, etc.), the user may use such functions in ADQL. An example of a function would be (in ADQL/s):

```
Select HEALPIXID(a.ra, a.dec), a.ra, a.dec from photobjall a
```

A concise set of common built-in functions that represent the necessary astronomical functionality, together with their standard function names, will be defined in later versions of the ADQL specification.

2.1.2 INTO clause

INSERT is supported for future interoperability with VOSpace. The VOSpace specification is under development within the Grid and Web Services WG of the IVOA. In SQL we may use ‘SELECT INTO’ to create a new table or ‘INSERT INTO’ to add data into an existing table. In ADQL this will probably be a VOSpace endpoint wherein the file/table will be created or appended to. How that is specified is not part of ADQL. ADQL simply supports syntax to allow to specification of a destination, e.g.:

```
Select g.* into VOS:/JHU/gal from galaxy g where g.redshift > 3.5
```

2.1.3 Comments

Comments will only be supported using the /* */ syntax to delimit comments. Comments are only supported before or after the main query – they may not be interspersed with the actual query.

¹ <http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html#Johnson75>

² <http://epaperpress.com/lexandyacc/>

2.2 Extensions to SQL92

This specification adds requirements on top of SQL92. ADQL SHALL support the extension described below.

These extensions to SQL92 are given with examples in **ADQL/s**, but of course **ADQL/x** can express any string from **ADQL/s**.

2.2.1 Aliases

All table names in ADQL MUST have an alias. Aliasing tables is a part of standard SQL, but we are enforcing this in **ADQL/s**.

This means queries in **ADQL/s** must take the form


```
Select * from table t
```

This makes substitution of table names much easier as it must be done in only one place to change the alias.


2.2.2 Regions

ADQL adds a keyword **REGION** to be used in the **WHERE** clause to specify search constraints. The **REGION** specification is supported as defined by the IVOA Data Model WG [3]. See subsection 2.4 for its detailed specification. The default coordinate system and units have been specified to simplify ADQL and the SkyNode implementation.

2.2.3 Mathematical Functions

C [4] mathematical functions shall be allowed in ADQL as follows:

Trigonometric functions:

acos(x), **asin(x)**, **atan(x)**, **atan2(x, y)** where x and y are  numeric, and
cos(x), **cot(x)**, **sin(x)**, **tan(x)** where x is expressed in radians

Math functions:

abs(x), **ceiling(x)**, **degrees(x)**, **exp(x)**, **floor(x)**, **log(x)**, **log10(x)**, **mod(x, y)**,
pi(), **power(x, y)**, **radians(x)**, **sqrt(x)**, **rand()**, **round(x, n)**, **truncate(x, n)**
where x and y are numeric and n is an integer.

2.2.4 XMATCH

XMATCH implies cross-match between two or more astronomical catalogues. The semantic meaning of XMATCH is defined more precisely in the SkyNode Interface specification. This document only specifies the syntax. The XMATCH keyword appears in the WHERE clause and looks like a function. The XMATCH has three parameters; first two parameters are table names to be cross-matched, the third parameter is the sigma value for the chi-square match, and the final (fourth) parameter, N, represents the cross-matching mode (in this example, it is assumed that N=1 corresponds to the chi-square match).

Here is an example in ADQL/s:

```
SELECT o.objId, o.ra, o.r, o.type, t.objId
FROM SDSS:PhotoPrimary o,
     TWOMASS:PhotoPrimary t
WHERE XMATCH(o,t,3.5,1)
      AND Region('Circle J2000 181.3 -0.76 6.5')
      AND o.type=3
```



2.2.5 Archive Qualification

ADQL allows for an archive to be specified in from of the table name as shown in the query above. An archive's SHORTNAME is prepended to the table name with the ':' separator. This implies a distributed query and would probably only be sent to a portal.

2.2.6 XPATH for Columns

To support XQuery as well as SQL, and since some of our data formats are described as XSD, it will be possible to express selections and selection criteria as a simple XPath. 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'
```

2.2.7 Returning subset of records – TOP

ADQL supports the TOP syntax to return only the first N records from a query, e.g.,

```
Select top 10 g.* from galaxy g
```



The semantics of this may vary on different database management systems. In ADQL the assumption is that TOP returns the first N records satisfying the criteria specified in the query.

2.2.8 Units

ADQL allows units for all constant values specified in the query. These are **optional**. ADQL does not specify what the units mean, and it simply allows for them syntactically specified, e.g:

```
Select g.* from galaxy g where g.gmag > 100 Jansky
```

2.2.9 Table Names with special chars

ADQL supports the use of '[']' to enclose literal names which may otherwise cause parse errors. For example if a table name starts with a number the parser could not deal with this but the following is valid:

```
Select a.* from [2df] a
```

This is also true for table names with spaces in or tables whose names are reserved words. Many database systems also support this syntax.

2.3 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. We should strive for backward compatibility i.e. only adding new features not deprecating the old.

Sample applications and tutorials for development and deployment of ADQL services are available at <http://skyservice.pha.jhu.edu/develop/vo/adql/>.

2.4 Regions

ADQL/s SHALL support the `Region` keyword. This will be followed by a single quoted string specifying a region in a simple manner similar to the current SDSS coverage specification in [5]. This would look something like:

```
Region('CIRCLE J2000 19.5 -36.7 0.02')
```

This is a one way operation. If an **ADQL/s** string is converted to **ADQL/x** this `Region` string will be converted to XML. If the resulting **ADQL/x** is converted back to **ADQL/s** the `Region` should remain as inlined XML using the `RegionXML` keyword.

Astronomical Data Query Language

There may be a comment section added to the region.xsd. In this comment section the original string should be kept. The comment section will be used for display purposes in certain areas, and should contain a summary description (in English) of the region. Other constructs mentioned in [5] are RECT, POLY, and CHULL are also supported.

As implied above it is possible to inline a region specification as in **ADQL/s** using the RegionXML keyword, e.g., (not a valid region specification)

```
RegionXML ('<circle><coordsys>ICRS</coordsys><ra>19.5</ra><dec>-36.7</dec><radius>0.02</radius></circle>')
```

It is also possible to refer to a region specification as a URL in **ADQL/s** using the RegionURL keyword, e.g.

```
RegionURL ('http://aserver.edu/aregion.xml')
```

3 ADQL example

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

```
SELECT a.objid, a.ra, a.dec
FROM SDSSDR2:Photoprimary a
WHERE Region('CIRCLE J2000 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/v1.0">
  <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="archiveTableType" Archive="SDSSDR2"
Name="Photoprimary" Alias="a" />
  </From>
  <Where>
    <Condition xsi:type="regionSearchType">
```

Astronomical Data Query Language

```
<Region xmlns:q1="http://www.ivoa.net/xml/STC/STCregion/v1.10"
xsi:type="q1:circleType" unit="deg">
  <q1:Center>181.3 -0.76</q1:Center>
  <q1:Radius>6.5</q1:Radius>
</Region>
</Condition>
</Where>
</Select>
```

4 ADQL XSD

The XML schema for ADQL is found at <http://www.ivoa.net/xml/ADQL/ADQL-v1.0.xsd>.

5 Changes from previous versions

- None. This is the first release.

6 References

- [1] IVOA SkyNode Interface – get latest one from <http://www.ivoa.net/Documents/latest/SNI.html>
- [2] Database Language SQL92 <http://www.contrib.andrew.cmu.edu/%7Eshadow/sql/sql1992.txt>
- [3] Space-Time Coordinates for the Virtual Observatory Version 1.10 <http://www.ivoa.net/xml/STC/STCregion/v1.10>
- [4] Java Database Connectivity Specification 3.0; download from <http://java.sun.com/products/jdbc/index.jsp>
- [5] SQLServer2000 HTM Interface specification; Alex Szalay, George Fekete, Jim Gray; July 2003 ; http://skyservice.pha.jhu.edu/develop/vo/adql/html/2_0.doc



Appendix ADQL Grammar

Below is the ANTLR grammar used to produce the parser in C#.

```

options {
    language = "CSharp";
    namespace="net.ivoa.adql";
}

// PARSE
*****
*****

class SqlParser extends Parser;
options {
    k = 2;
}
{
    public ADQLBuilder adqlb = new ADQLBuilder();
    public void init(){
        adqlb.sp = this;
    }
}
// starting rule
sql : selectStatement (SEMICOLON)? EOF;
selectStatement : queryExpression (computeClause)? (forClause)?
(optionClause)?;
queryExpression : subQueryExpression (unionOperator
subQueryExpression)* (orderByClause)?;
subQueryExpression : querySpecification | LPAREN queryExpression
RPAREN;
querySpecification : {init();adqlb.sql();}
    selectClause (fromClause)? (whereClause)? (groupByClause)?
(havingClause)? ;
selectClause :SELECT (all_distinct)? (restrictClause)? selectList;
restrictClause : {adqlb.AddSelectTop();}
    TOP Integer (PERCENT)? (WITH TIES)?;
all_distinct: ALL | DISTINCT;
whereClause : {adqlb.where();}
    WHERE searchCondition;
orderByClause : ORDER BY expression (orderType)?
{adqlb.AddOrderBy();} (COMMA expression (orderType)?
{adqlb.AddOrderBy();})*;

```

Astronomical Data Query Language

```
orderType: {adqlb.StackOrderType();} (ASC | DESC);
groupByClause : GROUP BY (ALL)? expression (COMMA expression)*
(WITH (CUBE | ROLLUP) )? {adqlb.AddGroupBy();};
havingClause : HAVING predicate {adqlb.AddHaving();};
optionClause : OPTION LPAREN queryHint (COMMA queryHint)* RPAREN;
queryHint :
    (HASH | ORDER) GROUP
    | (CONCAT | HASH | MERGE) UNION
    | (LOOP | MERGE | HASH) JOIN
    | FAST Integer
    | FORCE ORDER
    | MAXDOP Integer
    | ROBUST PLAN
    | KEEP PLAN
    | KEEPFIXED PLAN
    | EXPAND VIEWS;
forClause :
FOR (
    BROWSE
    | XML (RAW | AUTO | EXPLICIT) (COMMA XMLDATA)? (COMMA
ELEMENTS)? (COMMA BINARY BASE64)
    );
computeClause :
    COMPUTE
    // only allowed functions are: AVG, COUNT, MAX, MIN, STDEV,
STDEV, VAR, VARP, SUM
    identifier LPAREN expression RPAREN
    (COMMA identifier LPAREN expression RPAREN)*
    (BY expression (COMMA expression)* )?;
searchCondition :
    subSearchCondition ( {adqlb.setLastSearchType();} (AND | OR)
subSearchCondition)*;
subSearchCondition :
(NOT)? (
    (LPAREN searchCondition RPAREN) => LPAREN searchCondition
RPAREN
    | predicate | xMatch | region
    ) {adqlb.AddWhereComparison();};
predicate :
(
    expression
```

Astronomical Data Query Language

```
(
  // expression comparisonOperator expression
  comparisonOperator (
    expression
  | (ALL | SOME | ANY) LPAREN selectStatement RPAREN
  )
  | IS (NOT)? NULL
  | (NOT)? (
    LIKE expression (ESCAPE expression)? // only single char
  | BETWEEN expression AND expression
  | IN LPAREN (
      (selectStatement) => selectStatement
    | expression (COMMA expression)*
    ) RPAREN
  )
  | CONTAINS LPAREN (dbObject | STAR) COMMA (stringLiteral |
Variable) RPAREN
  | FREETEXT LPAREN (dbObject | STAR) COMMA (stringLiteral |
Variable) RPAREN
  )
  | EXISTS LPAREN selectStatement RPAREN
);
region: REGION LPAREN regionClause RPAREN {adqlb.StackRegion()};
regionClause: QuotedIdentifier;
xMatch: XMATCH LPAREN xAlias (COMMA xAlias)* (
  (COMMA xSigma RPAREN ) |
  (RPAREN LESSTHAN xSigma)
){adqlb.StackXMatch()};
xSigma: number;
selectList : selectItem ( COMMA selectItem )*;
selectItem :
(STAR // "*", "*" is a valid select list
| (
  // starts with: "alias = column_name"
  (alias2) => (
    (alias2 dbObject COMMA) => alias2 column
    | (alias2 dbObject (binaryOperator | LPAREN)) =>
alias2 expression
    | (alias2 column) => alias2 column
    | (alias2 expression) => alias2 expression
  )
  // all table columns: "table.*"
```

Astronomical Data Query Language

```
| (tableColumns) => tableColumns
| (explicitFunction) => explicitFunction
| (function) => function
// some shortcuts:
| (dbObject (alias1)? COMMA) => column (alias1)?

| (dbObject (binaryOperator | LPAREN) ) => expression (alias1)?

| (column) => column (alias1)?
| (expression) => expression (alias1)?
)
){adqlb.AddSelectItem();};
fromClause : {adqlb.from();}FROM tableSource (COMMA tableSource)*;
tableSource : subTableSource (joinedTable)*;
subTableSource :
(
  LPAREN (
    (joinedTables) => joinedTables RPAREN
    | (queryExpression) => queryExpression RPAREN alias1 //
"derived table", mandatory alias
  )
  | (function) => function (alias1)?
  | (archiveTable)? dbObject (alias1)? ( (WITH)? LPAREN tableHint
(COMMA tableHint)* RPAREN )?
  | Variable (alias1)?
  | (CONTAINSTABLE | FREETEXTTABLE) LPAREN
dbObject COMMA (dbObject | STAR) COMMA (stringLiteral | Variable)
(COMMA Integer)?
  RPAREN (alias1)?
  | COLON COLON function (alias1)? // built-in function
){adqlb.AddFromItem();};
joinedTable :
CROSS JOIN subTableSource
// "joinHint JOIN" is invalid join expression
| ( (INNER | (LEFT | RIGHT | FULL) (OUTER)? ) (joinHint)? )? JOIN
tableSource ON searchCondition;
joinedTables : subTableSource (joinedTable)+;
joinHint : LOOP | HASH | MERGE | REMOTE;
tableHint :
INDEX (
  LPAREN (identifier | Integer) ( COMMA (identifier | Integer) ) *
  RPAREN
```

Astronomical Data Query Language

```
    | ASSIGNEQUAL identifier    // old index hint syntax
    )
| FASTFIRSTROW
| HOLDLOCK
| NOLOCK
| PAGLOCK
| READCOMMITTED
| READPAST
| READUNCOMMITTED
| REPEATABLEREAD
| ROWLOCK
| SERIALIZABLE
| TABLELOCK
| TABLELOCKX
| UPDLOCK
| XLOCK
;
collate : COLLATE identifier
;
alias1 : // alias name can also be single-quoted literal (but not
for table names)
(AS)?
{adqlb.StackAlias();}
(
  identifier
  | stringLiteral
  | keywordAsIdentifier
);
alias2 :
(
  identifier
  | stringLiteral
  | keywordAsIdentifier
)
ASSIGNEQUAL;
xAlias: {adqlb.StackXAlias();} (NOT alias1 | QUESTIONMARK alias1 |
alias1);
tableColumns : dbObject DOT_STAR;

column :
(PLUS)* // "++column_name" is valid and updatable column name
```

Astronomical Data Query Language

```
(
  dbObject
  // for expression like "(column)" SQL Server returns updatable
  column
  | LPAREN column RPAREN
)
(collate)? // it is not well documented but COLLATE can be used
almost anywhere ...
;

expression :
  subExpression (binaryOperator subExpression )*
  {adqlb.CloseExpression();};
subExpression :
  (unaryOperator)?
  (
    constant
    | Variable
    | (explicitFunction) => explicitFunction
    | (function) => function
    | LPAREN {adqlb.OpenExpression();} (
      (selectStatement) => selectStatement // select statement
      returning a single value
      | expression2
      ) RPAREN
    | dbObject // column
    | parameterlessFunction
    | caseFunction
    | castFunction
  )
  (collate)?; // it is not well documented but COLLATE can be used
almost everywhere ...

expression2: expression;
function : // LEFT and RIGHT keywords are also function names
(dbObject | LEFT | RIGHT ) LPAREN (
  expression (COMMA expression)*
  | STAR // aggregate functions like
Count(), Checksum() accept "*" as a parameter
  | (ALL | DISTINCT) (STAR | expression) //
aggregate function
  | Variable ASSIGNEQUAL expression (COMMA
Variable ASSIGNEQUAL expression)*
```

Astronomical Data Query Language

```
        )?

RPAREN;
starArg: STAR {adqlb.StackStar();};
//don't use these directly
aggregateFunction:
(AVG|MAX|MIN|SUM|COUNT) {adqlb.StackAggregateOp();} LPAREN
(starArg|expression) RPAREN;
trigFunction:
(SIN|COS|TAN|COT|ASIN|ACOS|ATAN|ATAN2) {adqlb.StackTrigOp();} LPAREN
expression RPAREN;
mathFunction:
(CEILING|DEGREES|EXP|FLOOR|LOG|PI|POWER|RADIANS|SQRT|SQUARE|LOG10|R
AND|ROUND|TRUNCATE) {adqlb.StackMathOp();}
    LPAREN expression RPAREN ;
//use this one
explicitFunction: (aggregateFunction | trigFunction | mathFunction)
{adqlb.CloseFunction();};

archiveTable : {adqlb.StackArchiveTable();}
    identifier COLON;
caseFunction :
CASE (
    expression (WHEN expression THEN expression)+
    | (WHEN searchCondition THEN expression)+ // boolean
expression
    )
(ELSE expression)? END
;

castFunction : CAST LPAREN expression AS identifier (LPAREN Integer
(COMMA Integer)? RPAREN)? RPAREN;

dbObject : {adqlb.StackDBObject();}
(identifier | IDENTITYCOL | ROWGUIDCOL | keywordAsIdentifier)
(
    DOT (identifier | IDENTITYCOL | ROWGUIDCOL |
keywordAsIdentifier)
    | (DOT DOT) => DOT DOT (identifier | IDENTITYCOL | ROWGUIDCOL
| keywordAsIdentifier)
    )*;
parameterlessFunction : // any others ?
CURRENT_TIMESTAMP
| CURRENT_USER
| SESSION_USER
| SYSTEM_USER
```

Astronomical Data Query Language

```
;
systemVariable :
F_CONNECTIONS
| F_CPU_BUSY
| F_CURSOR_ROWS
| F_DATEFIRST
| F_DBTS
| F_ERROR
| F_FETCH_STATUS
| F_IDENTITY
| F_IDLE
| F_IO_BUSY
| F_LANGID
| F_LANGUAGE
| F_LOCK_TIMEOUT
| F_MAX_CONNECTIONS
| F_MAX_PRECISION
| F_NESTLEVEL
| F_OPTIONS
| F_PACK_RECEIVED
| F_PACK_SENT
| F_PACKET_ERRORS
| F_PROCID
| F_REMSERVER
| F_ROWCOUNT
| F_SERVERNAME
| F_SERVICENAME
| F_SPID
| F_TEXTSIZE
| F_TIMETICKS
| F_TOTAL_ERRORS
| F_TOTAL_READ
| F_TOTAL_WRITE
| F_TRANCOUNT
| F_VERSION
;
keywordAsIdentifier :
(
  AUTO
  | BASE64
```

Astronomical Data Query Language

```
| BINARY  
| CAST  
| CONCAT  
| CUBE  
| ELEMENTS  
| EXP  
| EXPAND  
| EXPLICIT  
| FAST  
| FASTFIRSTROW  
| FORCE  
| HASH  
| KEEP  
| KEEPFIXED  
| LOOP  
| MAXDOP  
| MIN  
| MAX  
| MERGE  
| NOLOCK  
| PAGLOCK  
| RAW  
| READCOMMITTED  
| READPAST  
| READUNCOMMITTED  
| REMOTE  
| REPEATABLEREAD  
| ROBUST  
| ROLLUP  
| ROWLOCK  
| SERIALIZABLE  
| TABLOCK  
| TABLOCKX  
| TIES  
| UPDLOCK  
| VIEWS  
| XLOCK  
| XML  
| XMLDATA  
)
```

Astronomical Data Query Language

```
;
stringLiteral : UnicodeStringLiteral | ASCIIStringLiteral;
identifier: NonQuotedIdentifier | QuotedIdentifier;
constant :
    {adqlb.StackIntAtom();} Integer |
    {adqlb.StackRealAtom();} Real |
    NULL |
    {adqlb.StackStringAtom();} stringLiteral |
    HexLiteral |
    Currency |
    ODBCDateTime |
    systemVariable;
unaryOperator : PLUS | MINUS | TILDE
;
binaryOperator : arithmeticOperator | bitwiseOperator;
arithmeticOperator : {adqlb.StackArithmeticOperator();} (PLUS |
MINUS | STAR | DIVIDE | MOD);
bitwiseOperator : AMPERSAND | TILDE | BITWISEOR | BITWISEXOR
;
comparisonOperator : {adqlb.StackComparisonOperator();}
(
    ASSIGNEQUAL | NOTEQUAL1 | NOTEQUAL2 | LESSTHANOREQUALTO1 |
LESSTHANOREQUALTO2
    | LESSTHAN | GREATERTHANOREQUALTO1 | GREATERTHANOREQUALTO2 |
GREATERTHAN
);
logicalOperator : ALL | AND | ANY | BETWEEN | EXISTS | IN | LIKE |
NOT | OR | SOME
;
unionOperator : UNION (ALL)?
;
number: (SIGN)? Number;

// LEXER
*****
*****

class SqlLexer extends Lexer;

options {
    testLiterals = false;
    k = 2;
    caseSensitive = false;
```

Astronomical Data Query Language

```
    caseSensitiveLiterals = false;
    charVocabulary='\u0000'..\uFFFE';
}

tokens {
    ADD = "add" ;
    ALL = "all" ;
    ALTER = "alter" ;
    AND = "and" ;
    ANY = "any" ;
    AS = "as" ;
    ASC = "asc" ;
    AUTHORIZATION = "authorization" ;
    AUTO = "auto" ;
    BACKUP = "backup" ;
    BASE64 = "base64" ;
    BEGIN = "begin" ;
    BETWEEN = "between" ;
    BINARY = "binary" ;
    BREAK = "break" ;
    BROWSE = "browse" ;
    BULK = "bulk" ;
    BY = "by" ;
    CASCADE = "cascade" ;
    CASE = "case" ;
    CAST = "cast" ;
    CHECK = "check" ;
    CHECKPOINT = "checkpoint" ;
    CLOSE = "close" ;
    CLUSTERED = "clustered" ;
    // COALESCE = "coalesce" ;
    COLLATE = "collate" ;
    COLUMN = "column" ;
    COMMIT = "commit" ;
    COMPUTE = "compute" ;
    CONCAT = "concat" ;
    CONSTRAINT = "constraint" ;
    CONTAINS = "contains" ;
    CONTAINSTABLE = "containstable" ;
    CONTINUE = "continue" ;
```

Astronomical Data Query Language

```
// CONVERT = "convert" ;
CREATE = "create" ;
CROSS = "cross" ;
CUBE = "cube" ;
CURRENT = "current" ;
CURRENT_DATE = "current_date" ;
CURRENT_TIME = "current_time" ;
CURRENT_TIMESTAMP = "current_timestamp" ;
CURRENT_USER = "current_user" ;
CURSOR = "cursor" ;
DATABASE = "database" ;
DBCC = "dbcc" ;
DEALLOCATE = "deallocate" ;
DECLARE = "declare" ;
DEFAULT = "default" ;
DELETE = "delete" ;
DENY = "deny" ;
DESC = "desc" ;
DISK = "disk" ;
DISTINCT = "distinct" ;
DISTRIBUTED = "distributed" ;
DOUBLE = "double" ;
DROP = "drop" ;
// DUMMY = "dummy" ;
DUMP = "dump" ;
ELEMENTS = "elements" ;
ELSE = "else" ;
END = "end" ;
ERRLVL = "errlvl" ;
ESCAPE = "escape" ;
EXCEPT = "except" ;
EXEC = "exec" ;
EXECUTE = "execute" ;
EXISTS = "exists" ;
EXIT = "exit" ;
EXPAND = "expand" ;
EXPLICIT = "explicit" ;
FAST = "fast" ;
FASTFIRSTROW = "fastfirstrow" ;
FETCH = "fetch" ;
```

Astronomical Data Query Language

```
FILE = "file" ;
FILLFACTOR = "fillfactor" ;
FOR = "for" ;
FORCE = "force" ;
FOREIGN = "foreign" ;
FREETEXT = "freetext" ;
FREETEXTTABLE = "freetexttable" ;
FROM = "from" ;
FULL = "full" ;
FUNCTION = "function" ;
GOTO = "goto" ;
GRANT = "grant" ;
GROUP = "group" ;
HASH = "hash" ;
HAVING = "having" ;
HOLDLOCK = "holdlock" ;
IDENTITY = "identity" ;
IDENTITY_INSERT = "identity_insert" ;
IDENTITYCOL = "identitycol" ;
IF = "if" ;
IN = "in" ;
INDEX = "index" ;
INNER = "inner" ;
INSERT = "insert" ;
INTERSECT = "intersect" ;
INTO = "into" ;
IS = "is" ;
JOIN = "join" ;
KEEP = "keep" ;
KEEPFIXED = "keepfixed" ;
KEY = "key" ;
KILL = "kill" ;
LEFT = "left" ;
LIKE = "like" ;
LINENO = "lineno" ;
LOAD = "load" ;
LOOP = "loop" ;
MAXDOP = "maxdop" ;
MERGE = "merge" ;
NATIONAL = "national" ;
```

Astronomical Data Query Language

```
NOCHECK = "nocheck" ;
NOLOCK = "nolock" ;
NONCLUSTERED = "nonclustered" ;
NOT = "not" ;
NULL = "null" ;
// NULLIF = "nullif" ;
OF = "of" ;
OFF = "off" ;
OFFSETS = "offsets" ;
ON = "on" ;
OPEN = "open" ;
OPENDATASOURCE = "opendatasource" ;
OPENQUERY = "openquery" ;
OPENROWSET = "openrowset" ;
OPENXML = "openxml" ;
OPTION = "option" ;
OR = "or" ;
ORDER = "order" ;
OUTER = "outer" ;
OVER = "over" ;
PAGLOCK = "paglock" ;
PERCENT = "percent" ;
PLAN = "plan" ;
PRECISION = "precision" ;
PRIMARY = "primary" ;
PRINT = "print" ;
PROC = "proc" ;
PROCEDURE = "procedure" ;
PUBLIC = "public" ;
RAISERROR = "raiserror" ;
RAW = "raw" ;
READ = "read" ;
READCOMMITTED = "readcommitted" ;
READPAST = "readpast" ;
READTEXT = "readtext" ;
READUNCOMMITTED = "readuncommitted" ;
RECONFIGURE = "reconfigure" ;
REFERENCES = "references" ;
REGION = "region";
REMOTE = "remote" ;
```

Astronomical Data Query Language

```
REPEATABLE_READ = "repeatableread" ;
REPLICATION = "replication" ;
RESTORE = "restore" ;
RESTRICT = "restrict" ;
RETURN = "return" ;
REVOKE = "revoke" ;
RIGHT = "right" ;
ROBUST = "robust" ;
ROLLBACK = "rollback" ;
ROLLUP = "rollup" ;
ROWCOUNT = "rowcount" ;
ROWGUIDCOL = "rowguidcol" ;
ROWLOCK = "rowlock" ;
RULE = "rule" ;
SAVE = "save" ;
SCHEMA = "schema" ;
SELECT = "select" ;
SERIALIZABLE = "serializable" ;
SESSION_USER = "session_user" ;
SET = "set" ;
SETUSER = "setuser" ;
SHUTDOWN = "shutdown" ;
SOME = "some" ;
STATISTICS = "statistics" ;
SYSTEM_USER = "system_user" ;
TABLE = "table" ;
TABLOCK = "tablock" ;
TABLOCKX = "tablockx" ;
TEXTSIZE = "textsize" ;
THEN = "then" ;
TIES = "ties" ;
TO = "to" ;
TOP = "top" ;
TRAN = "tran" ;
TRANSACTION = "transaction" ;
TRIGGER = "trigger" ;
TSEQUAL = "tsequal" ;
UNION = "union" ;
UNIQUE = "unique" ;
UPDATE = "update" ;
```

Astronomical Data Query Language

```
UPDATETEXT = "updatetext" ;
UPDLOCK = "updlock" ;
USE = "use" ;
USER = "user" ;
VALUES = "values" ;
VARYING = "varying" ;
VIEW = "view" ;
VIEWS = "views" ;
WAITFOR = "waitfor" ;
WHEN = "when" ;
WHERE = "where" ;
WHILE = "while" ;
WITH = "with" ;
WRITETEXT = "writetext" ;
XLOCK = "xlock" ;
XMATCH = "xmatch";
XML = "xml" ;
XMLDATA = "xmldata" ;

// system variables
F_CONNECTIONS = "@@connections" ;
F_CPU_BUSY = "@@cpu_busy" ;
F_CURSOR_ROWS = "@@cursor_rows" ;
F_DATEFIRST = "@@datefirst" ;
F_DBTS = "@@dbts" ;
F_ERROR = "@@error" ;
F_FETCH_STATUS = "@@fetch_status" ;
F_IDENTITY = "@@identity" ;
F_IDLE = "@@idle" ;
F_IO_BUSY = "@@io_busy" ;
F_LANGID = "@@langid" ;
F_LANGUAGE = "@@language" ;
F_LOCK_TIMEOUT = "@@lock_timeout" ;
F_MAX_CONNECTIONS = "@@max_connections" ;
F_MAX_PRECISION = "@@max_precision" ;
F_NESTLEVEL = "@@nestlevel" ;
F_OPTIONS = "@@options" ;
F_PACK_RECEIVED = "@@pack_received" ;
F_PACK_SENT = "@@pack_sent" ;
F_PACKET_ERRORS = "@@packet_errors" ;
```

Astronomical Data Query Language

```
F_PROCID = "@@procid" ;
F_REMSERVER = "@@remserver" ;
F_ROWCOUNT = "@@rowcount" ;
F_SERVERNAME = "@@servername" ;
F_SERVICENAME = "@@servicename" ;
F_SPID = "@@spid" ;
F_TEXTSIZE = "@@textsize" ;
F_TIMETICKS = "@@timeticks" ;
F_TOTAL_ERRORS = "@@total_errors" ;
F_TOTAL_READ = "@@total_read" ;
F_TOTAL_WRITE = "@@total_write" ;
F_TRANCOUNT = "@@trancount" ;
F_VERSION = "@@version" ;
```

```
//math
CEILING = "ceiling";
DEGREES = "degrees";
EXP = "exp";
FLOOR = "floor";
LOG = "log";
PI = "pi";
POWER = "power";
RADIANS = "radians";
SQRT = "sqrt";
SQUARE = "square";
LOG10 = "log10";
RAND = "rand";
ROUND = "round";
TRUNCATE = "truncate";
```

```
//trig
SIN = "sin";
COS = "cos";
TAN = "tan";
COT = "cot";
ASIN = "asin";
ACOS = "acos";
ATAN = "atan";
ATAN2 = "atan2";
```

Astronomical Data Query Language

```
//aggregate
MAX = "max";
MIN = "min";
SUM = "sum";
AVG = "avg";
COUNT = "count";
}

// Operators

protected DOT:; // generated as a part of Number rule
COLON : ':' ;
COMMA : ',' ;
SEMICOLON : ';' ;

LPAREN : '(' ;
RPAREN : ')' ;
//LSQUARE : '[' ;
//RSQUARE : ']' ;

ASSIGNEQUAL : '=' ;
NOTEQUAL1 : "<>" ;
NOTEQUAL2 : "!=" ;
LESSTHANOREQUALTO1 : "<=" ;
LESSTHANOREQUALTO2 : "!>" ;
LESSTHAN : "<" ;
GREATERTHANOREQUALTO1 : ">=" ;
GREATERTHANOREQUALTO2 : "!<" ;
GREATERTHAN : ">" ;

DIVIDE : '/' ;
PLUS : '+' ;
MINUS : '-' ;
STAR : '*' ;
MOD : '%' ;

AMPERSAND : '&' ;
TILDE : '~' ;
BITWISEOR : '|' ;
BITWISEXOR : '^' ;
```

Astronomical Data Query Language

```
DOT_STAR : ".*" ;

NOT : '!';
QUESTIONMARK : '?';

Whitespace
  : ( ' ' | '\t' | '\n' | '\r' )
  { _ttype = Token.SKIP; }
  ;

// COMMENTS
SingleLineComment
  : "--" ( ~('\r' | '\n') )*
  { _ttype = Token.SKIP; }
  ;

MultiLineComment
  : "/*" (~'')* '*' ('*' | ( ~('' | '/') (~'')* '*' ) )* '/'
  { _ttype = Token.SKIP; }
  ;

// LITERALS

protected Letter : 'a'..'z' | '_' | '#' | '@' | '\u0080'..'ufffe';
protected Digit : '0'..'9';
protected Integer ;;
protected Real ;;
protected Exponent : 'e' (Sign)? (Digit)+ ;
protected Sign : (PLUS | MINUS);
Number :
  ( (Digit)+ ('.' | 'e') ) => (Digit)+ ( '.' (Digit)*
  (Exponent)? | Exponent) { _ttype = Real; }
  | '.' { _ttype = DOT; } ( (Digit)+ (Exponent)? { _ttype = Real;
  } )?
  | (Digit)+ { _ttype = Integer; }
  | "0x" ('a'..'f' | Digit)* { _ttype = HexLiteral; } // "0x" is
  valid hex literal
  ;

protected Currency : // generated as a part of NonQuotedIdentifier
  rule
  ('$' | '\u00a3'..'u00a5' | '\u09f2'..'u09f3' | '\u0e3f' |
  '\u20a0'..'u20a4' | '\u20a6'..'u20ab')
```

Astronomical Data Query Language

```
    ((Digit)+ ('.' (Digit)* )? | '.' (Digit)+
;
ODBCDateTime :
    '{' (Whitespace)? ("ts" | 't' | 'd') (Whitespace)?
    ('n')? '\'' (~'\'' )* '\'' ( '\'' (~'\'' )* '\'' )* (Whitespace)?
    '}'
;

NonQuotedIdentifier options { testLiterals = true; } :
(
    (Currency) => Currency { _ttype = Currency; }
    | ('a'..'z' | '_' | '#' | '\u0080'..'ufffe') (Letter | Digit)* //
    first char other than '@'
);

QuotedIdentifier
:
(
    '[' (~']')* ']' (']' (~']')* ']')*
    | '"' (~'"')* '"' ('"' (~'"')* '"')*
)
;

Variable
    // test for literals in case of a function begining with '@'
    (eg. "@@ERROR")
    options { testLiterals = true; }
    : '@' (Letter | Digit)+
    ;

ASCIIStringLiteral
:
'\'' (~'\'' )* '\'' ( '\'' (~'\'' )* '\'' )*
;

UnicodeStringLiteral
:
'n' '\'' (~'\'' )* '\'' ( '\'' (~'\'' )* '\'' )*
;

// Numeric Constants
```

Astronomical Data Query Language

protected

HexLiteral // generated as a part of Number rule

: // "0x" ('0'..'9' | 'a'..'f')*

;
