ASAM CAT ODS

Open Data Services 5.3.1

Part 9 of 14

Physical Storage

Version 1.3.1

Date: 2015-01-01

Base Standard

Association for **S**tandardisation of
Automation and **M**easuring Systems

Status of Document

Date:	2015-01-01
Author:	ASAM e.V.
Version:	1.3.1
Status:	Release

Table of Contents

FOREWORD

This document contains chapter 9 of the ASAM ODS base standard, version 5.3.1.

Chapter 9 of the ASAM ODS base standard specifies the Physical Storage of ASAM ODS and includes some examples. It provides details on the general and application-model-specific tables within the relational database system used to store ASAM ODS information. This includes a compact and a searchable alternative to store mass data. Furthermore it specifies how to outsource mass data into external binary files in mixed mode operation.

It is intended for implementers of ASAM ODS and shall be used as a technical reference for setting up a database system, for storing ASAM ODS information into such physical storage, and for correctly interpreting the content of the physical storage of ASAM ODS.

This chapter is part of a series of chapters specifying the ASAM ODS base standard, and must not be used as a stand-alone specification. The technical reference of the ASAM ODS base standard is built by the complete set of chapters as listed below:

- Chapter 1: Introduction
- Chapter 2: Relation to Other Standards
- Chapter 3: Architecture
- Chapter 4: Base Model (32)
- Chapter 5: OO-API (5.3.1)
- Chapter 6: RPC-API (3.2.1)
- Chapter 7: ATF/XML (1.3.1)
- Chapter 8: ATF/CLA (1.4.1)
- Chapter 9: Physical Storage (1.3.1)
- Chapter 10: MIME Types and External References (1.2.1)
- Chapter 11: Terms and Definitions
- Chapter 12: Symbols and Abbreviations
- Chapter 13: Bibliography
- Chapter 14: Appendices

9 Physical Storage

9.1 Introduction

9.1.1 General

This chapter describes how measurement data and results are (physically) stored in a relational database to comply with the ASAM ODS standard.

For the physical storage of ASAM ODS information several tables are used in addition to those that hold the result data.

These tables are named SVCxxx, where xxx describes the kind of table.

The tables are:

• **SVCENT**	for storage of application elements (metamodel)
• **SVCATTR**	for storage of application attributes (metamodel)
• **SVCREF**	for storage of references (metamodel)
• **SVCVAL**	for storage of values (submatrix resp. local column)
• **SVCINST**	for storage of instance attributes
• **SVCENUM**	for storage of enumerations
• **SVCACLI**	for storage of security data for protection of instances
• **SVCACLA**	for storage of security data for protection of elements and attributes
• **SVCTPLI**	for storage of ACL templates
• **SVCVAL_SPS**	for storage of values (submatrix resp. local column) as single points

The following sections describe those tables in detail. The scripts in those sections are specified for SQL-92 (except section 9.2.4 and 9.2.10 where Oracle database systems version 7.3 and higher are assumed). They should serve as examples for other database languages and systems.

A subsequent section describes how the values of application attributes are stored, especially if they cannot be placed immediately into the corresponding tables.

A final section explains the storage mechanism in the so-called "mixed-mode", where mass data are stored externally while most application model information is kept in the relational database.

9.1.2 Data Types Used

ASAM has specified standardized ASAM data types which have been published end of 2002. These data types show standardized names, always beginning with A_. Those data types will be used by ASAM ODS consistently in all new specifications in the future.

The Physical Storage based on relational databases has been developed and specified several years ago. At that time no standardized data types have been available ASAM-wide. Therefore ASAM ODS decided to define data types that cover the needs of ASAM ODS. These data types always start with either T_ (in case they are single valued/structured types) or S_ (in case they are sequences resp. arrays of single valued/structured types).

The Physical Storage described in this chapter has been implemented in ODS servers and installed in a quite large number of companies. Replacing the T_ types by the A_ types would require to modify and exchange those installed servers and the underlying database designs, and also to modify the clients accessing the servers.

That is why ASAM ODS has decided to not introduce the standardized ASAM data types (A_) in the Physical Storage description.

Instead, chapter 3 of the ASAM ODS specification describes the relationship between the T_ types and the A_ types. One should note that the enumeration of the data types and their enumeration names are identical to those specified in the ASAM data type specification, and that the data types themselves are in most cases binary compatible. So a one-to-one mapping is easily possible, if needed.

9.1.3 STRING TYPE DATA

Starting with ASAM ODS version 5.3.0 the physical storage supports UTF-8 for string data encoding. One should note that for this purpose some table definitions of the meta data tables (SVC...) have been changed compared to earlier versions; the string column data types now specify the number of characters (not bytes).

New ASAM ODS installations should use the table definitions as given in the subsequent sections of this chapter, even if no UTF-8-specific characters are going to be used or an old ODS server shall be used. Also the tables holding the instances should specify the number of characters (not bytes) for string type attributes.

Legacy ASAM ODS data (i.e. data in data bases with previous table definitions) may cause problems if used with a new ODS server. When converting an existing database to UTF8 character set any strings within the SVCVAL blobs will not be converted automatically.

9.2 Description of SVC Tables

9.2.1 SVCENT

This table holds the descriptions of the different application elements.

The SQL-92 syntax for creating this table is as follows:

```
create table svcent
(
  AID      integer NOT NULL,       /* Identifier                             */
  ANAME    char(30 char) NOT NULL, /* Unique(!) name of application element */
  BID      integer NOT NULL,       /* ID of base element, only internal)     */
  DBTNAME  char(30 char) NOT NULL, /* Name of the table                      */
  SECURITY integer                 /* Security mode (bit masked)             */
);
```

The column AID is an identifier for an application element; it must be unique within the complete ASAM ODS physical storage and is usually given by the ASAM ODS server creating the application model. Oracle database servers use an Oracle sequence to determine the next free ID, the name of the sequence (calculation algorithm) is DBTNAME+ID_SEQ.

The column ANAME is the name of the application element as it has been specified by the creator of the application model. Within an application model the application element names must be unique. This specification restricts the length of application element names and database table names to a maximum of 30 characters. Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS.

The column BID is the unique identifier of the base element of which this application element has been derived. BIDs are standardized by ASAM ODS as shown in the following table:

Table 1 - Base element IDs (BID)

BID	Base Element Name
0	AoAny
47	AoAttributeMap
1	AoEnvironment
40	AoExternalComponent
48	AoFile
39	AoLocalColumn
43	AoLog
3	AoMeasurement
4	AoMeasurementQuantity
46	AoNameMap
44	AoParameter
45	AoParameterSet

BID	Base Element Name
15	AoPhysicalDimension
11	AoQuantity
12	AoQuantityGroup
38	AoSubmatrix
2	AoSubTest
36	AoTest
37	AoTestDevice
23	AoTestEquipment
24	AoTestEquipmentPart
25	AoTestSequence
26	AoTestSequencePart
13	AoUnit
14	AoUnitGroup
21	AoUnitUnderTest
22	AoUnitUnderTestPart
34	AoUser
35	AoUserGroup

The column DBTNAME specifies the name of the table which contains all instances of that application element. This table name will only be known to the ASAM ODS server; it will not be provided to a client through any interface. The length of the complete table name must not exceed the maximum allowed length of table names within the database.

For array tables, the naming convention is "<Table name>_ARRAY". This means e.g. for Oracle databases that the length of <Table name> must not exceed 24 characters in that case, which is the maximum table name length (30 characters) minus "_ARRAY" (6 characters).

The column SECURITY holds the information on the security level for the application element. Possible security levels are 'element security', 'instance security' and 'attribute security' or combinations of them. Each of them is enabled/disabled via a bit code (0 = disabled, 1 = enabled).

Table 2 - Bit coding of ACL types

Bit #	Decimal	Configuration of:
0	1	Element security: ACL-entries on application element (= access rights are set commonly for all instances of the application element)
1	2	Instance security: ACL-entries on the application element's instances (= access rights are set individually for each instance)
2	4	Attribute security: ACL-entries on the application element's attributes (= access rights are set individually for each attribute)

The information stored in the column SECURITY corresponds to the entity 'SecurityLevel' which is defined and described in chapter 4 (Base Model).

More Information on the security concept of ASAM ODS is described in chapter 3.

9.2.1.1 EXAMPLE FOR A SVCENT TABLE

To clarify the specification of SVCENT, the following example is given.

EXAMPLE: FOR A SVCENT TABLE

The following table declares three application elements and the names of the tables in which their corresponding instances are stored.

AID	ANAME	BID	DBTNAME	SECURITY
1	`TestObject`	`21 (AoUnitUnderTest)`	`HEADER`	`0`
2	`TestRun`	`36 (AoTest)`	`STEP_NAMES`	`0`
3	`Capture`	`3  (AoMeasurement)`	`TEST_STEP`	`0`

9.2.2 SVCATTR

This table holds the descriptions of the application attributes and application relations of all application elements.

The SQL-92 syntax for creating this table in is as follows:

```
create table svcattr
(
  AID      integer NOT NULL,       /* Identifier of the element               */
  ATTRNR   integer,                /* Attribute number for the sequence       */
  AANAME   char(30 char) NOT NULL, /* Unique(!) name of AE attribute          */
  BANAME   char(30 char),          /* Name of base attribute                  */
  FAID     integer,                /* Reference to foreign application element */
  FUNIT    integer,                /* Reference to unit, if always the same    */
  ADTYPE   integer,                /* Data type of attribute, (eg. float, long)*/
  AFLEN    integer,                /* Max. length of strings or streams       */
  DBCNAME  char(30 char) NOT NULL, /* Name of column in the respective table   */
  ACLREF   integer,                /* attribute for ACL-Template reference     */
  INVNAME  char(30 char),          /* Inverse name of AE attribute            */
  FLAG     integer,                /* Flags of the attribute.                 */
  ENUMNAME char(30 char)           /* Name of the enumeration in case ADTYPE is
                                      DT_ENUM, otherwise unused               */
);
```

The column AID specifies the unique identifier of the application element to which the application attribute or relation belongs. It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT.

The column ATTRNR contains the number of the application attribute/relation and is usually given by the ASAM ODS server creating the application model. This number must be unique among all application attributes and relations of the same application element. It should start with 1 and be incremented continuously. When using the RPC-API a value of 0 must be avoided.

The maximum number of application attributes may be restricted by the underlying database. The column AANAME holds the name of the application attribute/relation as it has been specified by the creator of the application model. However, n:m relations are not listed in this table but are described in the table SVCREF. Within an application element the names of application attributes/relations must be unique. This specification restricts the length of application attribute/relation names to a maximum of 30 characters. Also some other strings are restricted in length. Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS.

The column BANAME contains the name of the base attribute/relation in case the application attribute/relation is derived from a base attribute/relation (i.e. is a based application attribute); otherwise this database field is empty. No two application attributes of the same application element may reference the same base attribute; however in case of relations to abstract superclasses, more than one application relation may reference the same base relation.

The column FAID is relevant for application relations and for the inheritance concept.

In case of an application relation it contains the identifier of the application element which is the target of the application relation. It is a reference to one of the entries (the one whose AID equals this value of FAID) in the table SVCENT.

In case of the inheritance concept, the FAID column is used to specify for a subclass application element which of the other application elements is its superclass application element. For this purpose, the entry in the FAID column of the subclass's attribute derived from the base attribute 'id' is used; it will contain the AID of the corresponding superclass application element. An example is given in section 9.3.8.

The column FUNIT may hold a reference to the identifier of an instance of an application element that is derived from the base element AoUnit.

The column ADTYPE specifies the enumeration value of the data type of the application attribute. In case of an application relation, this column contains the data type of the reference identifiers. These may be DT_LONG and DT_LONGLONG for 1:n and DS_LONG or DS_LONGLONG for the inverse. A list of ASAM ODS data types and corresponding enumeration values is given below in this section. From these, the data types corresponding to DT_UNKNOWN, DT_ID, and DS_ID may not be used with one exception: the attribute derived from the base attribute 'values' of a local column may have a variety of data types, depending on the setting at the corresponding measurement quantity; thus for this attribute ADTYPE will be set to DT_UNKNOWN (0).

The column AFLEN holds the maximum length (in characters) of application attributes that have string, date, or stream data types (In all other cases this column may be empty). The maximum value that may be set here depends on the database capabilities to store strings. Boundless strings are not supported by ASAM ODS.

The column DBCNAME specifies the name of the column which finally holds the application attribute/relation values. All instances of an application element are stored together in one instances table whose name is given in SVCENT>DBTNAME. The value of an application attribute/relation (with a few exceptions) will be stored in a column of that table, the column's name being specified by DBCNAME. The values for DBCNAME must be unique within one application element.

Exceptions to this rule are:

- Attributes derived from the base attributes 'values', 'flags', and 'generation-parameters' of AoLocalColumn will not be stored in a column of the instances table but in the separate table SVCVAL or SVCVAL_SPS (see corresponding sections) or in external files for mixed mode server operation (see section 9.4). In this case the corresponding fields of DBCNAME contain the string "NULL".
- Relations of type 1:n will only be stored in the instances table of one of the two related application elements: the one that relates to only one partner instance. In this case the corresponding field of DBCNAME at the other application element (inverse relation) contains the string "NULL".
- Relations of type n:m will not be stored in a column of the instances table but in a separate table specified in SVCREF (see section 9.2.3). (They are not specified in SVCATTR at all.)
- Application attributes of sequence types (DS_...) or of other multiple-element types are stored in a separate table, further described in section 9.3.

The column name DBCNAME will only be known to the ASAM ODS server; it will not be provided to a client through any interface. This specification restricts the length of the column names to a maximum of 30 characters.

The column ACLREF is used in case the application element has declared 'instance security' (by the SECURITY entry in the SVCENT table). It may only be set for table entries that specify an application relation of 1:n type (and that are not inverse relations) and contains a flag which indicates whether (=1) or not (=0) the application relation must be resolved to find an ACL-template. It will be used by the server to determine what ACL-entries to attach to a newly to be created instance in case no specific ACL-entry is provided for the instance.

The information stored in the column ACLREF corresponds to the entity 'InitialRightsAt' which is defined and described in chapter 4 (Base Model).

More information on the ASAM ODS security concept can be found in chapter 3.

The column INVNAME is only relevant for application relations and holds the name of the inverse relation. Within the ODS models (base as well as application models) all relations also have inverse ones, thus an inverse name will be available. The inverse name equals the application relation name (AANAME) of the related application element; it may differ from the name in the base model. This column will remain empty for application attributes.

The column FLAG contains three flags that control the content of the application attribute/relation. They are explained in detail below in this section.

The column ENUMNAME holds the name of the enumeration in case the data type ADTYPE of the application attribute is DT_ENUM (=30) or DS_ENUM(=31). This specification restricts the length of enumeration names to a maximum of 30 characters.

An enumeration is a bundle of fixed name value pairs, it is used to make a data item readable for humans by using the string representation. Computers will use the value representation because of a more compact storage and faster compare operations.

The individual items of all enumerations are stored in SVCENUM; items belonging to the same enumeration are identified by having the same ENUMNAME (see also section 9.2.6).

9.2.2.1 THE FLAGS OF APPLICATION ATTRIBUTES

In an ASAM ODS model (base model as well as application models) the application attributes may have flags like AUTOGENERATE, UNIQUE or OBLIGATORY. These flags will be stored as bit-pattern in the column FLAG.

The following table shows the bits used to generate the values in this column and their respective meaning:

Table 3 - Bit coding of application attribute flags

Bit	Meaning
0	(UNIQUE) This flag tells the server that the values of this attribute have to be unique. 1 means the values must be unique. The server can secure such behavior by database constraints.
1	(OBLIGATORY) This flag tells the server that the values of this attribute are obligatory. 1 means the values must be set. The server can secure such behavior by database constraints. This is the logically inverse information to OPTIONAL in the ASAM ODS base model.
2	(AUTOGENERATE) This flag tells the server that the values of this attribute have to be generated by the server, e.g. the Id. 1 means the attribute values will automatically be generated by the server. For each column specified as AUTOGENERATE, a trigger needs to be defined manually within the database or the server and an appropriate algorithm must be specified to generate the values. The behavior is not under control of any ASAM ODS client.

9.2.2.2 THE DATA TYPES ENCODING IN THE COLUMN ADTYPE

The column ADTYPE specifies the data type with which values of this application attribute will be coded in the database. Instead of specifying the data type as a string, it is given as a number (representing its data type enumeration value). The relationship between the enumeration and the data type itself is shown in the following table, where T_xxx are basic data types, S_xxx are sequences of the basic data types, and DT_xxx (resp. DS_xxx) are the names of the data type enumerations.

Table 4 - ASAM ODS data types

Name of Data Type Enumeration		Name of Data Type in OO-API	Description
DT_UNKNOWN	(=0)		Unknown data type.
DT_STRING	(=1)	T_STRING	String.
DT_SHORT	(=2)	T_SHORT	Short value (16 bit).
DT_FLOAT	(=3)	T_FLOAT	Float value (32 bit).
DT_BOOLEAN	(=4)	T_BOOLEAN	Boolean value.
DT_BYTE	(=5)	T_BYTE	Byte value (8 bit).
DT_LONG	(=6)	T_LONG	Long value (32 bit).
DT_DOUBLE	(=7)	T_DOUBLE	Double precision float value (64 bit).
DT_LONGLONG	(=8)	T_LONGLONG	LongLong value (64 bit).
DT_ID	(=9)	T_ID	LongLong value (64 bit). Not used. DT_LONGLONG is used instead.
DT_DATE	(=10)	T_DATE	Date.
DT_BYTESTR	(=11)	T_BYTESTR	Bytestream.
DT_BLOB	(=12)	T_BLOB	Blob.
DT_COMPLEX	(=13)	T_COMPLEX	Complex value (32 bit each part).
DT_DCOMPLEX	(=14)	T_DCOMPLEX	Complex value (64 bit each part).
DS_STRING	(=15)	S_STRING	String sequence.
DS_SHORT	(=16)	S_SHORT	Short sequence.
DS_FLOAT	(=17)	S_FLOAT	Float sequence.
DS_BOOLEAN	(=18)	S_BOOLEAN	Boolean sequence.
DS_BYTE	(=19)	S_BYTE	Byte sequence.
DS_LONG	(=20)	S_LONG	Long sequence.
DS_DOUBLE	(=21)	S_DOUBLE	Double sequence.
DS_LONGLONG	(=22)	S_LONGLONG	Longlong sequence.
DS_COMPLEX	(=23)	S_COMPLEX	Complex sequence.
DS_DCOMPLEX	(=24)	S_DCOMPLEX	Double precision complex sequence.
DS_ID	(=25)	S_ID	LongLong sequence. Not used. DS_LONGLONG is used instead.
DS_DATE	(=26)	S_DATE	Date sequence.
DS_BYTESTR	(=27)	S_BYTESTR	Bytestream sequence.
DT_EXTERNALREFERENCE	(=28)	T_EXTERNALREFERENCE	External reference.
DS_EXTERNALREFERENCE	(=29)	S_EXTERNALREFERENCE	Sequence of external reference.
DT_ENUM	(=30)	T_LONG	Enumeration.
DS_ENUM	(=31)	S_LONG	Sequence of enumerations.

This definition of data types is compliant with the one for ASAM ODS 4.1. The data type is also delivered at the RPC-API. The value of the data type is identical to the value of the attribute 'datatype‘ in 'AoMeasurementQuantity' and 'default_datatype‘ in 'AoQuantity‘.

9.2.2.3 Example for a SVCATTR Table

To clarify the specification of SVCATTR, the following example is given.

EXAMPLE: FOR A SVCATTR TABLE

The following table declares three application attributes for the application element with AID=1 and three application attributes for the application element with AID=3. The values of the id-attributes HeadId and TestId are generated by the server and must be unique, so the FLAG values of the attributes HeadId and TestId are set to 5 (0101).

AID	ATTRNR	AANAME	BANAME	FAID	FUNIT	ADTYPE	AFLEN	DBCNAME	ACLREF	INVNAME	FLAG	ENUM NAME
1	1	HeadId	id			6		HEAD_ INDEX			5	
1	2	SerialNo	name			1	32	PART_ID				
1	3	Head_rel		3		6		TEST_ INDEX		TestObj_rel		
3	1	TestId	id			6		TEST_ INDEX			5	
3	2	TestObj_rel		1		6		HEAD_ INDEX		Head_rel		
3	3	State				1	32	TEST_ STATUS				

9.2.3 SVCREF

This table holds the descriptions of the n:m relations between application elements

The SQL-92 syntax for creating this table is as follows:

```
create table svcref
(
AID1      integer  NOT NULL,      /* ID of the first application element       */
AID2      integer  NOT NULL,      /* ID of the second application element      */
REFNAME   char(80 char) NOT NULL, /* Name of the application relation          */
DBTNAME   char(30 char) NOT NULL, /* Name of the table containing the relations*/
INVNAME   char(80 char) NOT NULL, /* Inverse name of AE relation               */
BANAME    char(30 char),          /* base relation name of AE relation         */
INVBANAME char(30 char)           /* inverse base relation name.               */
);
```

The column AID1 contains the identifier of the first application element (the start-point of the application relation). It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT.

The column AID2 contains the identifier of the second application element (the end-point of the application relation). It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT.

The column REFNAME contains the name of the application relation as it has been specified by the creator of the application model. This specification restricts the length of the relation names to a maximum of 80 characters. Also some other strings are restricted in length. Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS.

The names of application relations between the same two application elements must be unique. Additionally the name of an application relation may not equal the name of an application attribute/relation specified in SVCATTR of the application element referenced by AID1.

The column DBTNAME specifies the name of the table in which the relations between instances are stored finally; this table name must be unique within the table SVCREF and must not equal any other table name within the ASAM ODS physical storage. This specification restricts the length of the table names to a maximum of 30 characters. The structure of that table is explained below in this section. The table name will only be known to the ASAM ODS server; it will not be provided to a client through any interface.

The column INVNAME contains the name of the inverse relation. The inverse name equals the application relation name (REFNAME) of the inverse n:m relation in this table (which may differ from the relation name in the base model). This specification restricts the length of the inverse relation names to a maximum of 80 characters.

The column BANAME contains the name of the base relation in case the application relation is derived from a base relation (i.e. is a based application relation); otherwise the database field is empty.

The column INVBANAME contains the name of the inverse base relation in case the application relation is derived from a base relation (i.e. is a based application relation); otherwise the database field is empty.

The combination of AID1, AID2, and REFNAME must be unique within this table. It is allowed to have more than one n:m relation between the same two application elements as long as they have different names.

Each n:m relation is described by only one entry in the table SVCREF.

9.2.3.1 THE TABLE CONTAINING N:M RELATIONS BETWEEN INSTANCES

The table containing the n:m relations (and whose name is given in DBTNAME) has three columns:

- the name of the first column must be identical to the DBCNAME of the corresponding entry in the SVCATTR table for the base attribute "id" of the element given by the application element ID in column AID1 of table SVCREF.
- the name of the second column must be identical to the DBCNAME of the corresponding entry in the SVCATTR table for the base attribute "id" of the element given by the application element ID in column AID2 of table SVCREF.
- the name of the third column must be REFNAME.

EXAMPLE: FOR A TABLE FINALLY CONTAINING THE RELATIONS

Assuming that the tables SVCENT, SVCATTR, and SVCREF contain the entries given below, the table which finally contains the relations between instances will have the name "UNITTOGROUPS" and look like shown below.

SVCENT:

AID	ANAME	BID	DBTNAME	SECURITY
3	Unit	...	...	0
4	UnitGroup	...	...	0
...	...	...	...	0

SVCATTR:

AID	ATTRNR	AANAME	BANAME	FAID	FUNIT	ADTYPE	AFLEN	DBCNAME	ACLREF	INVNAME	FLAG	ENUM NAME
3	1	Unitid	id			6		UID			5	
3	2	Unit	name			1	32	UNIT				
4	1	UnitGroupId	id			6		UGID			5	
4	2	UnitGroup	name			1	32	UNITGROUP				
...	...	...	...	...	...	...	...	...	...	...	...	...

SVCREF:

AID1	AID2	REFNAME	DBTNAME	INVNAME	BANAME	INVBANAME
3	4	Unit_to_Groups	UNITTOGROUPS	...	...	...
...	...	...	...	...	...	...

UNITTOGROUPS:

UID	UGID	REFNAME
ID of instance of Unit	ID of instance of UnitGroup	Unit_to_Groups
ID of instance of Unit	ID of instance of UnitGroup	Unit_to_Groups
...	...	...

9.2.4 SVCVAL

This table holds the measured data and the value flags of the data.

The syntax for creating this table in an Oracle database server is as follows:

```
create table svcval
(
  MEQID       integer NOT NULL,     /* ID of measurement quantity */
  PMATNUM     integer NOT NULL,     /* ID/number of submatrix     */
  SEGNUM      integer NOT NULL,     /* Number of segment          */
  VALINDEP    integer NOT NULL,     /* Independent flag           */
  VALEXIMP    integer NOT NULL,     /* Implicit/explicit flag     */
  VALBLOBLEN  integer NOT NULL,     /* Length of the blob         */
  VALBLOB     long raw NOT NULL     /* Data of the column         */
);
```

Each row in this table stores a set of values; they are finally put in VALBLOB.

The column MEQID specifies the id of that measurement quantity whose values are stored in the row of SVCVAL. It is a reference to an instance of an application element derived from the base element AoMeasurementQuantity.

The column PMATNUM holds the identifier of the submatrix to whom the values belong. It is a reference to an instance of an application element derived from the base element AoSubmatrix.

The column SEGNUM contains the segment number. Due to limitations of some databases, VALBLOB is restricted to a maximum size of 10000 bytes. If the set of values to be stored exceeds this size, they are segmented and stored in separate VALBLOBs. In that case SEGNUM will start with 1 and be incremented for each subsequent segment. Segmentation may be applied to values even if the VALBLOB size limit is not reached.

The column VALINDEP holds the independent flag. Its value must be identical to the value of the attribute derived from the base attribute "independent" of the corresponding local column.
It is defined as:

1 =	this local column is independent within its submatrix (e.g. set values, scales, time axis)
0 =	this local column is dependent within its submatrix

The column VALEXIMP holds a flag that indicates whether the values are explicitly stored or implicitly given by a generation algorithm. Its value must correspond to the value of the attribute derived from the base attribute "sequence_representation" of the corresponding local column.
It is defined as:

1 =	this local column is implicitly defined
0 =	this local column is explicitly defined

More information on implicit local columns are given below in this section.

The column VALBLOBLEN specifies the number of values stored in VALBLOB. The data type of the values is given in the attribute derived from the base attribute "datatype" of the corresponding measurement quantity.

The column VALBLOB finally contains the values to be stored, together with optional flags. The size of VALBLOB depends on the data type of the values, on VALBLOBLEN, and on the existence of flags. VALBLOB is a blob (binary large object) with the following structure:

Table 5 - Structure of VALBLOB

VALUES	GAP	FLAGS
	(conditional)	(conditional)

In VALBLOB only values and flags are stored.

The endian order of VALUES and FLAGS is identical to the endian order of the server.

In Oracle databases a LENGTH field must be inserted before the VALUES field. LENGTH is a four byte field which gives the real number of bytes in VALBLOB (whereas VALBLOBLEN gives only the number of data items); the LENGTH element itself is not counted. The maximum length of a segment is always limited to 10000; LENGTH$\leq$10000.

This structure of VALBLOB is used for values of any data type.

In case of simple numerical values (DT_BYTE, DT_SHORT, DT_LONG, DT_LONGLONG, DT_FLOAT, DT_DOUBLE) the size of each such value is well known (1, 2, 4, 8, 4, 8 bytes respectively), and the number of values that fit into one segment can be easily calculated in advance; it depends on whether or not flags need to be stored as well. The values x_1, x_2, .., x_N are stored in their binary representation of the ODS server, one after the other, starting with x_1. Note that DT_BOOLEAN values are stored using one byte per value, representing TRUE by a byte value $\neq$0 and FALSE by a byte value 0.

EXAMPLE:

VALBLOBLEN = 2500
Data type = DT_FLOAT (4 byte)
→ LENGTH = 2500 * 4 = 10.000

This is also the maximum number of float values that fit in one segment of SVCVAL.

In case of structured numerical values (DT_COMPLEX, DT_DCOMPLEX) each value consists of two simple numericals representing the real and the imaginary part of the complex value. Thus the total length required to store the values x_1, x_2, .., x_N can again be calculated in advance (2*N*4 bytes for DT_COMPLEX, 2*N*8 for DT_DCOMPLEX). The values are stored in their binary representation of the ODS server, one after the other, starting with x_1, and within each value starting with the real part followed by the imaginary part. A value is always kept within one VALBLOB; if not both of its components (real, imaginary) fit into the remaining space of a VALBLOB the whole value will be put into the next VALBLOB.

Simple string values (DT_STRING, DT_DATE) are stored as sequence with the delimiter "`\0`" (a binary 0x00 byte). Example: "`Hello\0Peter\0Test\0`" gives LENGTH=17, VALBLOBLEN=3.

Binary stream values (DT_BYTESTR, DT_BLOB) are stored with some length information embedded.
A bytestream value (DT_BYTESTR) is stored with a preceding 4-byte length information followed by the stream of bytes. The length information is interpreted as an unsigned integer value and stored in the endian order of the ODS server; it gives the number of bytes in the following stream of bytes. A sequence x_1, x_2, .., x_N of such values are stored one after the other, starting with x_1.
A blob value (DT_BLOB) consists of the (textual) blob header and the blob byte stream. Within VALUES of VALBLOB a blob is stored starting with the header string, followed by "`\0`" (a binary 0x00 byte), followed by a 4-byte length information, followed by the blob byte stream. The length information is interpreted as an unsigned integer value and stored in the endian order of the ODS server; it gives the number of bytes in the following stream of bytes. A sequence x_1, x_2, .., x_N of such values are stored one after the other, starting with x_1. Note that there is no additional separator between the end of one blob value x_i and the start of the header of the next blob value x_{i+1}. A value is always kept within one VALBLOB; if not both of its components (header, byte stream) fit into the remaining space of a VALBLOB the whole value will be put into the next VALBLOB.

If LENGTH is encountered to be larger than the number of bytes required for the values, flags are defined for the local column and are stored in VALBLOB. The number of bytes available for flags is the difference between LENGTH and the number of bytes required for the values. One flag is represented by two bytes (one short integer value). The flag is used to store status information on individual data values; each status information is coded into one bit of the 16-bit flag. The meaning of the individual bits is given in the following table. Typically all bits are set (thus providing a value of 0x000F).

Table 6 - Bit coding of value flags

Abbreviation	Value	Description
AO_VF_VALID	0x0001	Value is valid.
AO_VF_VISIBLE	0x0002	The value has to be visualized.
AO_VF_UNMODIFIED	0x0004	Value is not modified.
AO_VF_DEFINED	0x0008	Value is defined. If the value in a value matrix is not available this bit is not set.

The number of bytes required for the flags is twice the value of VALBLOBLEN. If LENGTH is encountered to be larger than the number of bytes required for the values plus the flags, there is a GAP between VALUES and FLAGS. The sequence of the values and the flags is left aligned within the element VALUES resp. FLAGS. The content of the GAP is not relevant.

EXAMPLE: 2.500 FLOAT VALUES WITH FLAGS

First blob:
1666 values + 1666 flags = 6664 bytes for values + 3332 bytes for flags.
If the LENGTH is set to 10.000 bytes, the GAP will be 4 bytes.
If the LENGTH is set to the actual length of 9996 bytes, there will be no GAP.

Second blob:
834 values + 834 flags (2500-1666).
LENGTH = 5004 bytes: results in no gap.
If LENGTH is from 5006 to 10.000 bytes: there will be a gap.

An ASAM ODS server must always be able to read blobs with GAPs, but it is up to the server manufacturer whether the server will use the option of writing blobs with GAPs.

The combination of MEQID, PMATNUM, and SEGNUM must be unique within this table.

Chapter 3 explains the relationship between local columns, submatrices, and the measurement matrix. Important characteristics of a submatrix are that it contains no gaps and that the nth values in each of its local columns relate to each other. Thus measurement quantities can keep their values in the same submatrix only if they have the same number of values and relate to each other (e.g. are sampled at the same time with the same rate).

The name of a local column must be identical with the name of the corresponding measurement quantity. No two local columns within one submatrix may refer to the same measurement quantity. Therefore the combination of MEQID and PMATNUM is indirectly also a unique reference to a corresponding local column. Each row in SVCVAL thus relates to exactly one local column.

A local column on the other hand may be represented by more than one row of SVCVAL. This is typically the case if the values of the local column are segmented into several segments (e.g. due to limitations of blob size).

This may also be the case if the same measurement quantity is measured in different rates. Its values will then be divided between different submatrices, each of them having a local column that relates to the same measurement quantity; the final value storage uses a different row of SVCVAL for each rate, identified by the combination of MEQID (the measurement quantity) and PMATNUM (the submatrix).

9.2.4.1 EXPLICIT AND IMPLICIT LOCAL COLUMNS AND GENERATION ALGORITHMS

The implicit column flag VALEXIMP specifies the interpretation of the values stored in VALBLOB.

Usually values are stored explicitly (e.g. each sample is stored as a value). This is the default case and indicated by VALEXIMP=0. In that case the element VALUES of VALBLOB contains explicitly all values (of one segment) of that measurement quantity:

VALUES = $\{x_1, x_2, x_3, \dots x_n, \dots x_N\}$

The attribute derived from the base attribute 'sequence_representation' of the corresponding local column must correspond to this information and thus be set to 'explicit' (=0).

For some kind of data (e.g. timestamps on time driven sampling) the value for each row may be generated; in that case, indicated by VALEXIMP=1, only the generation algorithm needs to be specified, which may save huge amounts of storage space.

This case is subdivided into two subcases:

a) pure implicit value generation:

In this subcase the VALUES element of VALBLOB contains only a set of generation parameters required for the generation algorithm:

VALUES = $\{p_1, p_2, p_3, \ldots p_m, \ldots, p_M\}$

The number M of required parameters depends on the generation algorithm. The data type of the parameters $p_1 \ldots p_M$ is specified by the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity.

The following table lists the variants which currently are provided for pure implicit value generation. Which of the variants is selected is specified by the attribute derived from the base attribute 'sequence_representation' of the corresponding local column; it is also specified in the table. The final values x_n are calculated according to the formula given in the table; n starts with 1 and ends with the value of the attribute derived from the base attribute 'number_of_rows' of the corresponding submatrix application element. VALBLOBLEN in this case must be M (the number of parameters).

Table 7 - Generation algorithms for pure implicit value generation

sequence_representation	M	Generation algorithm
implicit_constant (=1)	1	$x_n = p_1$
implicit_linear (=2)	2	$x_n = p_1 + (n - 1) * p_2$
implicit_saw (=3)	3	$x_n = p_1 + ((n - 1) \bmod (p_3 - p_1)/p_2) * p_2$

Only measurement quantities with numerical data types corresponding to DT_SHORT, DT_FLOAT, DT_BYTE, DT_LONG, DT_DOUBLE, DT_LONGLONG are allowed for this type of implicit storage. The calculation must always be done using the data type defined for the measurement quantity. However, the expression $(p_3 - p_1)/p_2$ must be truncated to integer to start each saw curve cycle at p_1.

No flags may be stored in VALBLOB in this case.

b) raw data preprocessing:

In this subcase, N raw data values are stored together with M generation parameters. The VALUES element is structured as follows:

VALUES = $\{p_1, p_2, p_3, \ldots p_m, \ldots p_M, r_1, r_2, r_3, \ldots r_n, \ldots r_N\}$

The number N of raw data values equals the number of resulting values x_n which itself equals the value of the attribute derived from the base attribute 'number_of_rows' of the corresponding submatrix application element. The number M of generation parameter depends on the generation algorithm. The data type of the generation parameters corresponds to DT_DOUBLE (using eight bytes per generation parameter). The data type of the raw data r_n is specified by the attribute derived from the base attribute 'raw_datatype' of the corresponding local column.

In case all generation parameters and raw data fit into one segment, they are consecutively placed into the VALUES part of the VALBLOB as indicated above.

In case more than one segment is required to store generation parameters and raw data, the first segment contains all generation parameters p_1, p_2, ... p_M, followed by the first N1 raw data. The second segment contains the next set of raw data and so on, so that the following structure will be set up:

Table 8 - Structure of VALBLOBs for raw data preprocessing

segment (SEGNUM)	generation parameters and raw data (VALBLOB)		number of raw data (VALBLOBLEN)
1	p_1, p_2, ... p_M	r_1, r_2, ... r_{N1}	N1
2	r_{N1+1}, r_{N1+2}, ... r_{N2}		N2-N1
3	r_{N2+1}, r_{N2+2}, ... r_{N3}		N3-N2
...	...		
k+1	r_{Nk+1}, r_{N1+2}, ... r_N		N-Nk

Each of the segments are stored in a separate row of SVCVAL. The corresponding entries for VALBLOBLEN are the individual numbers of raw data values; they sum up to N. Generation parameters are not counted in VALBLOBLEN.

The following table lists the variants which currently are provided for raw data preprocessing. Which of the variants is selected is specified by the attribute derived from the base attribute 'sequence_representation' of the corresponding local column; it is also specified in the table. The final values x_n are calculated according to the formula given in the table; n starts with 1 and ends with the value of the attribute derived from the base attribute 'number_of_rows'.

Table 9 - Generation algorithms for raw data preprocessing

sequence_representation	M	Generation algorithm
raw_linear (=4)	2	$x_n = p_1 + p_2*r_n$
raw_polynomial (=5)	p_1+2	$x_n = p_2 + p_3*r_n + p_4*r_n^2 + p_{2+p1}*r_n^{p1}$
raw_linear_calibrated (=10)	3	$x_n = (p_1 + p_2*r_n)*p_3$

In case of raw_polynomial, p_1 gives the order of the polynomial (and thus also indirectly the number of generation parameters). The order of the polynomial is restricted by the maximum size of a segment of the physical storage.

Only measurement quantities with numerical data types corresponding to DT_SHORT, DT_FLOAT, DT_BYTE, DT_LONG, DT_DOUBLE, DT_LONGLONG, DT_COMPLEX, DT_DCOMPLEX are allowed for this type of implicit storage. The resulting values x_n are provided by the server with a data type specified by the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity.

Note that generation parameters are values of type DT_DOUBLE.
In case the raw data are of an integer type (DT_BYTE, DT_SHORT, DT_LONG or DT_LONGLONG) the calculations according to the generation algorithm must be performed in highest possible floating point resolution, no matter what the final data type of the measurement quantity is; a typecast to the final data type should only be done as last processing step.
The same applies for raw data of any floating point type.
In case raw data r_n are of a complex type (DT_COMPLEX or DT_DCOMPLEX), having a real part RE(r_n) and an imaginary part IM(r_n), the calculations according to the generation algorithm must be applied mathematically correct. Taking into account that generation parameters are real valued, this includes:

- multiplying a real value p_2 with a complex value r_n results in a complex value whose real part is p_2*RE(r_n) and whose imaginary part is p_2*IM(r_n).
- adding a real value p_1 to a complex value r_n results in a complex value whose real part is p_1+RE(r_n) and whose imaginary part still is IM(r_n).
- raising a complex value r_n to the power of m (thus building r_n^m) is best performed on the absolute value ABS(r_n) and the angle ANG(r_n) of r_n: the absolute value of the result is $(\text{ABS}(r_n))^m$, and the angle of the result is (m*ANG(r_n)); though in most cases applications will keep the angle within a range of [$-\pi$, π] or [0, 2π], ASAM ODS servers will not do that; a modulo operation on client side might be needed.

9.2.5 SVCINST

This table holds the instance attributes. Instance attributes are attributes given to an instance that are not specified in the application model. They cannot be placed into the tables holding the instances because there is no table column prepared for them. Instead, all instance attributes of instances of any application element are stored together in this single table SVCINST.

The SQL-92 syntax for creating this table is as follows:

```
create table svcinst
(
  AID       integer NOT NULL,         /* Application element id         */
  IID       integer NOT NULL,         /* Application instance id        */
  NAME      char(30 char) NOT NULL,   /* Instance attribute name        */
  AODT      integer,                  /* ASAM ODS data type             */
  UNITID    integer,                  /* Unit-ID                        */
  NUMVAL    number(38,10),            /* Numeric with maximum precision */
  TXTVAL    varchar2(255 char)        /* String value                   */
);
```

The table SVCINST looks like this:

Logical:	Appl. Element ID	Appl. Instance ID	Inst. Attr, Name	Data type	Unit ID	Numeric Value	Textual Value
Database-column:	AID	IID	NAME	AODT	UNITID	NUMVAL	TXTVAL

The column AID contains the identifier of the application element to which the instance belongs that receives the instance attribute. It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT.

The column IID contains the identifier of the instance itself, receiving the instance attribute. It is a reference to one of the instances contained in the table of instances of the corresponding application element.

The column NAME holds the name of the instance attribute. This specification restricts the length of names of instance attributes to a maximum of 30 characters. Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS.

The column AODT contains the data type enumeration value that gives the data type of the instance attribute value. It is specified according to the ASAM ODS 'datatype'-enumeration given in section 9.2.2. No sequence data types and no complex data types are supported for instance attributes.

The column UNITID stores the identifier of the unit of the instance attribute; it thus refers to an instance of an application element derived from the base element AoUnit.

The column NUMVAL contains the value of the instance attribute in case it is of a numeric data type. It will be stored with the maximum precision data type available in the database system, regardless what data type is specified in AODT. AODT thus does not specify the data type for storing the instance attribute in the physical storage but specifies the data type with which this instance attribute must be returned to a requesting client. Supported numerical data types are those corresponding to DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_DOUBLE, DT_LONGLONG (enumeration values 2,...,8).

The column TXTVAL contains the value of the instance attribute in case it is of a textual data type. Supported textual data types are DT_STRING, DT_DATE (enumeration values 1 and 10). This specification restricts the length of the textual instance attribute values to a maximum of 255 characters.

The combination of AID, IID and NAME must be unique within this table.

9.2.6 SVCENUM

This table holds the enumeration data.

The SQL-92 syntax for creating this table is as follows:

```
create table svcenum
(
  ENUMID      integer            NOT NULL,     /* Id of the enumeration.   */
  ENUMNAME    char(30 char)      NOT NULL,     /* Name of the enumeration. */
  ITEM        integer            NOT NULL,     /* Value of the item.       */
  ITEMNAME    char(128 char)     NOT NULL      /* Name of the item.        */
);
COMMENT ON TABLE svcenum IS 'ASAM ODS enumeration definitions';
```

The column ENUMID holds the identifier of the enumeration, which must be unique for each enumeration. If an enumeration consists of more than one item, the same ENUMID value will appear in several rows of the table SVCENUM.

The column ENUMNAME contains the name of the enumeration, which must be unique for each enumeration. If an enumeration consists of more than one item, the same ENUMNAME value will appear in several rows of the table SVCENUM. This specification restricts the length of enumeration names to a maximum of 30 characters. Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS.

The column ITEM holds the value of the enumeration item. This allows to specify enumerations quite flexibly, though it is good practice to start an enumeration with an item number of 0 and continuously increment the item values without any gaps. Values for ITEM must be unique within one enumeration.

The column ITEMNAME holds the human-readable string value of the enumeration item. This specification restricts enumeration item names to a maximum of 128 characters. Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS. Values for ITEMNAME should be unique within one enumeration.

9.2.6.1 Definition of ASAM ODS Enumeration

The ASAM ODS base model defines several enumerations, such as datatype_enum or seq_rep_enum. These enumerations are available in the physical storage of an ASAM ODS server. When specifying an application model, additional application-specific enumerations may be defined.

An enumeration is a bundle of fixed name value pairs; it is used to make a data item readable for humans by using the string representation. The computers will use the value representation because of the more compact storage and the faster compare-operations.

Up to now in ASAM ODS the enumerations were represented by a T_LONG value, except in ATF, where they are stored as T_STRING.

The following example shows the enumeration seq_rep_enum given in the base model.

EXAMPLE: FOR THE DEFINITION OF AN ENUMERATION IN SVCENUM

ENUMID	ENUMNAME	ITEM	ITEMNAME
1013	seq_rep_enum	0	EXPLICIT
1013	seq_rep_enum	1	IMPLICIT_CONSTANT
1013	seq_rep_enum	2	IMPLICIT_LINEAR
1013	seq_rep_enum	3	IMPLICIT_SAW
1013	seq_rep_enum	4	RAW_LINEAR
1013	seq_rep_enum	5	RAW_POLYNOMIAL
1013	seq_rep_enum	6	FORMULA
1013	seq_rep_enum	7	EXTERNAL_COMPONENT
1013	seq_rep_enum	8	RAW_LINEAR_EXTERNAL
1013	seq_rep_enum	9	RAW_POLYNOMIAL_EXTERNAL
1013	seq_rep_enum	10	RAW_LINEAR_CALIBRATED
1013	seq_rep_enum	11	RAW_LINEAR_CALIBRATED_EXTERNAL

In the base model the names of the items are lower case, programmers normally use upper case names for the enumeration items, but it is up to the creator of the model what to use. It is important that ITEMNAME is case sensitive.

9.2.6.2 WORKING WITH THE PHYSICAL STORAGE OF EARLIER VERSIONS

The definition of the physical storage for relational databases is not downward compatible. Servers of previous versions will find values DT_ENUM or DS_ENUM in the column ADTYPE of the table SVCATTR and do not know how to handle these. So once the physical storage is upgraded to the ASAM ODS Version 5.0 or higher using enumerations, the server must be exchanged.

When servers capable to handle enumerations work on previous versions of the physical storage, they will not find the table SVCENUM. It is recommended for a server to check the existence of the table before this table will be used. If the table SVCENUM does not exist and API-methods accessing the enumerations are called, the exception AO_NOT_IMPLEMENTED must be thrown. The servers must work properly with regard to the remaining functionality.

When servers capable to handle enumerations work on previous versions of the physical storage, they will not find the column ENUMNAME in the table SVCATTR. It is recommended that a server compliant to this definition should work properly even if the column ENUMNAME does not exist in the table SVCATTR. Any access to enumerations through API methods must be refused and the exception AO_NOT_IMPLEMENTED must be thrown.

9.2.7 SVCACLI

This table holds the security data for the instance protection (for more details on the ASAM ODS security concept and access rights see chapter 3).

The SQL-92 syntax for creating this table is as follows:
```
create table svcacli
(
  USERGROUPID    integer NOT NULL,   /* Reference to a UserGroup instance */
  AID            integer NOT NULL,   /* Application element id            */
  IID            integer NOT NULL,   /* Application instance id           */
  RIGHTS         integer NOT NULL    /* Rights value (bit masked)         */
);
```

This table contains the access control list (ACL)-entries for individual instances. Each entry (table row) stored in this table corresponds to the entity 'ACLI' which is defined and described in chapter 4 (Base Model). One should note however that the application element is not specified by its name (as defined in chapter 4) but by its (much shorter) application element ID.

Access will be granted to a user group only, not to individual users. Access is specified in means of permissions, not restrictions. Each access permission is specified in one row of the table.

The column USERGROUPID contains the identifier of the user group for which the access permission is specified. It refers to an instance of an application element derived from the base element AoUserGroup.

The column AID contains the identifier of the application element to which the instance belongs. It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT.

The column IID contains the identifier of the instance that is subject to this access permission. It is a reference to one of the instances in the instances table of the corresponding application element.

The column RIGHTS holds the five basic rights Read, Update, Insert, Delete, and Grant as a bit masked value. If the respective bit is set (=1), it means the respective right is granted. The following table shows the available rights:

Table 10 - Bit coding of access rights

Bit 4	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Grant	Delete	Insert	Update	Read

The table SVCACLI looks like this:

Logical:	Group ID	Appl. Element ID	Instance ID	Rights
Database-column:	USERGROUPID	AID	IID	RIGHTS

9.2.8 SVCACLA

This table holds the security data for the attribute and element security (for more details on the ASAM ODS security concept and access rights see chapter 3).

The SQL-92 syntax for creating this table is as follows:

```
create table svcacla
(
  USERGROUPID  integer NOT NULL,      /* Reference to a UserGroup instance     */
  AID          integer NOT NULL,      /* Application element id                */
  AANAME       char(30 char),         /* Application attribute name            */
  TYPE         char(5 char) NOT NULL, /* Selector ("AA" or "AE") for rights on
                                         attribute or element                  */
  RIGHTS       integer NOT NULL       /* Rights value (bit masked)             */
);
```

This table contains the access control list (ACL)-entries for application elements and application attributes. Each entry (table row) stored in the this table corresponds to the entity 'ACLA' which is defined and described in chapter 4 (Base Model). One should note however that the application element is not specified by its name (as defined in chapter 4) but by its (much shorter) application element ID. Also a TYPE column has been introduced, so that the decision whether the rights for 'element security' or 'attribute security' are specified in a particular table row does not base on whether or not the AANAME field is empty.

Access will be granted to a user group only, not to individual users. Access is specified in means of permissions, not restrictions. Each access permission is specified in one row of the table.

Instead of 'char(..)' the more flexible type 'varchar2(..)' may be used, when an Oracle database server is used. ASAM ODS servers must support both types for compatibility with earlier versions of ASAM ODS.

The column USERGROUPID contains the identifier of the user group for which the access permission is specified. It refers to an instance of an application element derived from the base element AoUserGroup.

The column AID contains the identifier of the application element that is subject to this access permission or which owns the attribute that is subject to this access permission. It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT.

The column AANAME contains the name of the application attribute that is subject to this access permission. It is a reference to one of the entries in the table SVCATTR (the one for which AID and AANAME match). In case the permission specifies access control to an application element as a whole (indicated by TYPE="AE"), AANAME remains empty.

The column TYPE holds the selector for the type of security mechanism. In case of element security, TYPE is set to "AE"; in case of attribute security, TYPE is set to "AA".

In case of "AE", AANAME is irrelevant, and any non-empty value of AANAME shall be ignored by a server.

In case of "AA", AANAME must not be empty and must match the name of an application attribute of the application element specified by AID.

The column RIGHTS holds the five basic rights Read, Update, Insert, Delete, and Grant as a bit masked value. If the respective bit is set (=1), it means the respective right is granted. The following table shows the available rights:

Table 11 - Bit coding of access rights

Bit 4	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Grant	Delete	Insert	Update	Read

The table SVCACLA looks like this:

Logical:	Group ID	Appl. Element ID	Attr. Name	Type	Rights
Database-column:	USERGROUPID	AID	AANAME	TYPE	RIGHTS

9.2.9 SVCTPLI

This table holds the ACL-templates for security (for more details on the ASAM ODS security concept see chapter 3). ACL-templates are of importance only in the case of 'instance security'.

The SQL-92 syntax for creating this table is as follows:

```
create table svctpli
(
  USERGROUPID    integer NOT NULL,   /* Reference to a UserGroup instance */
  AID            integer NOT NULL,   /* Application element id            */
  IID            integer NOT NULL,   /* Application instance id           */
  REFAID         integer NOT NULL,   /* Referencing application element   */
  RIGHTS         integer NOT NULL    /* Rights value (bit masked)         */
);
```

This table contains templates for access control list (ACL)-entries. Each entry (table row) stored in this table corresponds to the entity 'ACLTemplate' which is defined and described in chapter 4 (Base Model). One should note however that the application element and the referencing application element are not specified by their names (as defined in chapter 4) but by their (much shorter) application element IDs.

Access will be granted to a user group only, not to individual users. Access is specified in means of permissions, not restrictions. Each access permission is specified in one row of the table.

One should note that the access rights specified by an ACL-template do not apply for the access to the application element or instance to which this ACL-template is attached. Instead, they will be used for ACL-entries of newly to be created instances (usually of other application elements). The mechanism of using ACL-templates is provided to allow an ASAM ODS server to automatically create a set of ACL-entries for new instances without the need that each instance provides its own ACL-entries. It is described in detail in the context of ASAM ODS Security in chapter 3.

The column USERGROUPID contains the identifier of the user group for which the access permission is specified. It refers to an instance of an application element derived from the base element AoUserGroup.

The column AID contains the identifier of an application element to which this ACL-template is attached. It is a reference to one of the entries (the one with the same identifier AID) in the table SVCENT. Depending on the value of IID, the ACL-template is either attached to the application element itself (IID=0) or to one of its instances (IID>0).

The column IID contains the identifier of the instance to which the ACL-template is attached. It is a reference to one of the entries in the instances table of the corresponding application element. In case the template is attached to an application element, IID must contain the value 0.

The column REFAID holds the application element ID of the referencing instance. If new instances of the application element RefAID are created that do not specify an ACL-entry of their own, the server may find the corresponding ACL-template at AID-IID and copy USERGROUPID and RIGHTS from the ACL-template into an ACL-entry for that instance. The column ACLREF in the table SVCATTR specifies, which application relation(s) must be resolved to find the ACL-templates that will be used while creating a new instance. An exact description of this procedure is given in chapter 3.

The column RIGHTS holds the five basic rights Read, Update, Insert, Delete, and Grant as a bit masked value. If the respective bit is set (=1), it means the respective right is granted.

The table SVCTPLI looks like this:

Logical:	Group ID	Appl. Element ID	Instance ID	Ref. Appl-ID	Rights
Database-column:	USERGROUPID	AID	IID	REFAID	RIGHTS

9.2.10 SVCVAL_SPS

Values and flags of a local column are usually stored in a BLOB within the table SVCVAL. Though this is a very compact and storage-saving method there are some performance issues involved when single values of a local column need to be selected and analyzed.

In those cases the SPS (single point storage) version of local column storage may be used. The advantage is that the measurement values have not to be packed into a BLOB and no segmentation of the measurement values is needed. Therefore the performance does not get worse with the size and number of segments.

Furthermore SVCVAL_SPS supports some data types that were not available with the SVCVAL storage option.

Contrary to the conventional SVCVAL table, the SVCVAL_SPS table has a reference directly to the local column and the corresponding submatrix.

Note that this section is only applicable for local columns where the values of the attribute derived from the base attribute 'sequence_representation' is of the subset {explicit, implicit_constant, implicit_linear, implicit_saw, raw_linear, raw_polynomial, raw_linear_external, raw_polynomial_external, raw_linear_calibrated, raw_linear_calibrated_external}, as in case of "external_component" no information on the values and flags is stored within the database at all. The three ".._external" cases of the subset use the SVCVAL_SPS table only for storing the generation parameters.

Note further that an ODS server will either use SVCVAL or SVCVAL_SPS; this is decided at initialization time of the server. It is not possible to mix both value representations within one ODS server.

In an Oracle database the syntax for creating the SVCVAL_SPS table is as follows:

```
create table svcval_sps
(
  SUBMATRIXID    NUMBER(20) NOT NULL, /* The id of the submatrix               */
  LOCALCOLUMNID  NUMBER(20) NOT NULL, /* The id of the local column            */
  POINT          NUMBER(20) NOT NULL, /* Number of the measurement point       */
  ORD            NUMBER(20) NOT NULL, /* Ordinal number                        */
  FLAG           NUMBER(5),           /* Flag value                            */
  NUMBER_VALUE   NUMBER,              /* Number value, for all numbers         */
  STRING_VALUE   VARCHAR2(4000 char), /* String value, for strings and dates   */
  BLOB_VALUE     BLOB                 /* BLOB value, for BLOBs and byte arrays */
);
```

Each row in this table is used to store

- one value of a measured point (in case it is given as a simple data type), or
- a part of one value of a measured point (in case it cannot be given as a simple value), or
- a generation parameter or one part of a set of generation parameters

The column SUBMATRIXID specifies the id of the submatrix to which the value belongs. It is a reference to an instance of an application element derived from the base element AoSubmatrix.

The column LOCALCOLUMNID specifies the id of the local column to which the value belongs. It is a reference to an instance of an application element derived from the base element AoLocalColumn.

The column POINT specifies the measured point whose value is provided in that row. It ranges from 1 to N (given N measured points in the corresponding submatrix). A value of '-1' may occur, in which case this row does not contain a specific measured value but some part of the generation parameters (see below).

The column ORD is used to enumerate the rows in case more than one row is needed to specify the value of a measured point (e.g. array types) or a set of generation parameters. It thus uniquely identifies each part of a multipart value. It is set to 0 if not needed.

The column FLAG contains the flag of the value that is given in its row. It is NULL if no flag is given for the value, or if the row contains a subsequent part of a multipart value, or if a parameter or a part of a parameter set is stored in that row.

The column NUMBER_VALUE is used to store a numerical value.

The column STRING_VALUE is used to store a string value.

The column BLOB_VALUE is used to store a bytestream or BLOB value. Note that in this case also the column STRING_VALUE will contain a value; it specifies the header of the BLOB.

The usage of SVCVAL_SPS depends strongly on the type of representation (given by the value of the attribute derived from the base attribute 'sequence_representation' of the local column) and on the data type of the values (given by the value of the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity).

The following paragraphs provide details on the supported cases.

9.2.10.1 Explicit Local Columns

This case is identified by a sequence representation 'explicit'.
All values are explicitly given and may be stored in SVCVAL_SPS. The way they are put into this table depends on their data type.

a) Simple numerical values

Supported data types: DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_LONGLONG, DT_DOUBLE.

Simple numerical values are stored within the NUMBER_VALUE column. The storage precision is the same for all such values. The conversion of the precision to/from the API data types is done during reading and writing of the value. Note that DT_LONGLONG is a structure of two long values ('high' and 'low'), when accessing such values through the API. They will be combined by the server into one 64-bit integer before storage and separated into two long values after retrieval.

ORD will be 0, STRING_VALUE and BLOB_VALUE will contain NULL. One row of the table is used to store one value, and its structure is:

Table 12 - SVCVAL_SPS for simple numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	<value>	NULL	NULL

In this and all subsequent structure tables <sID> represents the identifier of the corresponding submatrix, <cID> represents the identifier of the local column the value belongs to, <point> represents the measured point (from 1 .. N), <flag> represents the flag associated with the value (see table 6 in section 9.2.4), and <value> represents the value to be stored.

b) Structured numerical values

Supported data types: DT_COMPLEX, DT_DCOMPLEX.

Structured numerical values are also stored within the NUMBER_VALUE column. However, as their representation in the API data types is given by a structure, specific measures need to be taken to put them into SVCVAL_SPS.

DT_COMPLEX and DT_DCOMPLEX each are a structure of two components of floating point type ('r', 'i'; real and imaginary part). Each component will be stored in a row of the table. The storage precision is the same for both data types; the conversion of the precision to/from the API data types is done during reading and writing of the value. STRING_VALUE and BLOB_VALUE will contain NULL. Two rows of the table are used to store one value, ORD will allow to distinguish between these two rows, and the structure is (with<r-value> being the real and <i-value> being the imaginary part of the value):

Table 13 - SVCVAL_SPS for structured numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	<r-value>	NULL	NULL
<sID>	<cID>	<point>	1	NULL	<i-value>	NULL	NULL

c) Simple string values

Supported data types: DT_DATE, DT_STRING.

Simple string values are stored within the STRING_VALUE column.

ORD will be 0, NUMBER_VALUE and BLOB_VALUE will contain NULL. One row of the table is used to store one value, and its structure is:

Table 14 - SVCVAL_SPS for simple string values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	NULL	<value>	NULL

d) Binary stream values

Supported data types: DT_BYTESTR, DT_BLOB.

DT_BYTESTR values are stored within the BLOB_VALUE column. A preceding 4-byte length information is followed by the stream of bytes. The length information is interpreted as an unsigned integer value and stored in the endian order of the ODS server; it gives the number of bytes in the following stream of bytes. Thus the length of a byte stream is implicitly restricted to (2^{32}-1) bytes, and the maximum length of BLOB_VALUE is (2^{32}+3) bytes.
ORD will be 0, NUMBER_VALUE and STRING_VALUE will contain NULL. One row of the table is used to store one value, and its structure is:

Table 15 - SVCVAL_SPS for bytestream values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	NULL	NULL	<value>

DT_BLOB values are composed of a header and the BLOB byte stream itself. The header is stored within the STRING_VALUE column while the BLOB byte stream is stored in the BLOB_VALUE column. The content of BLOB_VALUE will start with a 4-byte length information, followed by the stream of bytes. The length information is interpreted as an unsigned integer value and stored in the endian order of the ODS server; it gives the number of bytes in the following stream of bytes. Thus the length of the byte stream part of a blob is implicitly restricted to (2^{32}-1) bytes, and the maximum length of BLOB_VALUE is (2^{32}+3) bytes.
ORD will be 0, NUMBER_VALUE will contain NULL. One row of the table is used to store one value, and its structure is:

Table 16 - SVCVAL_SPS for blob values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	NULL	<header>	<value>

e) Array of simple numerical values

Supported data types: DS_SHORT, DS_FLOAT, DS_BOOLEAN, DS_BYTE, DS_LONG, DS_LONGLONG, DS_DOUBLE.

Each array element will be stored in the NUMBER_VALUE column, using one row of the table. Thus an array of M elements will use M rows, and ORD will be used to identify the array element that is contained in a row (the values for ORD thus range between 0 and M-1).

STRING_VALUE and BLOB_VALUE will contain NULL, and the structure is (with <value[m]> being the value of the mth array element; m=0..M-1):

Table 17 - SVCVAL_SPS for an array of simple numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	<value[0]>	NULL	NULL
<sID>	<cID>	<point>	1	NULL	<value[1]>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	<point>	M-1	NULL	<value[M-1]>	NULL	NULL

<sID>, <cID>, and <point> entries are the same in all rows. The <flag> entry associated with the whole array value, is only specified at the first array element.

As the values are stored in the column NUMBER_VALUE independent of their data types, the conversion of the precision to/from the API data types must be done by the server during reading and writing of the value. Note further that each array element of DS_LONGLONG is a structure of two long values ('high' and 'low'), when accessing such values through the API. They will be combined by the server into one 64-bit integer each before storage and separated into two long values after retrieval.

f) Array of structured numerical values

Supported data types: DS_COMPLEX, DS_DCOMPLEX

Arrays of structured numerical values are also stored within the NUMBER_VALUE column. However, as one single value consists of several components, more than one row is needed for each value. Thus (given M array elements) storing them requires a multiple of M rows in the table.

DS_COMPLEX and DS_DCOMPLEX each are an array of values that are composed of two parts each ('r', 'i'; real and imaginary part). Each component will be stored in a row of the table. The storage precision is the same for both data types; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Thus an array of M elements will use 2*M rows, and ORD will be used to identify the component as well as the array element that is contained in a row (the values for ORD thus range between 0 and 2*M-1). STRING_VALUE and BLOB_VALUE will contain NULL, and the structure is (with <r[m]> being the real part and <i[m]> being the imaginary part of the m^th array element; m=0..M-1):

Table 18 - SVCVAL_SPS for an array of structured numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	<r[0]>	NULL	NULL
<sID>	<cID>	<point>	1	NULL	<i[0]>	NULL	NULL
<sID>	<cID>	<point>	2	NULL	<r[1]>	NULL	NULL
<sID>	<cID>	<point>	3	NULL	<i[1]>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	<point>	2M-1	NULL	<i[M-1]>	NULL	NULL

<sID>, <cID>, and <point> entries are the same in all rows. The <flag> entry associated with the whole array of complex values, is only specified at the real part of the first array element.

g) Array of simple string values

Supported data types: DS_DATE, DS_STRING.

Each array element will be stored in the STRING_VALUE column, using one row of the table. Thus an array of M elements will use M rows, and ORD will be used to identify the array element that is contained in a row (the values for ORD thus range between 0 and M-1).

NUMBER_VALUE and BLOB_VALUE will contain NULL, and the structure is (with <value[m]> being the value of the m^th array element; m=0..M-1):

Table 19 - SVCVAL_SPS for an array of simple string values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	<point>	0	<flag>	NULL	<value[0]>	NULL
<sID>	<cID>	<point>	1	NULL	NULL	<value[1]>	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	<point>	M-1	NULL	NULL	<value[M-1]>	NULL

9.2.10.2 Implicit Local Columns and Generation Algorithms

This case is identified by a sequence representation from the subset {implicit_constant, implicit_linear, implicit_saw, raw_linear, raw_polynomial, raw_linear_calibrated}.

The values are either generated 'on the fly' or only raw values are given, and the final values must be calculated from them. In both cases there are generation parameters involved which also need to be stored in SVCVAL_SPS. The way they are put into this table depends on the sequence representation and the data type of the raw values.

In any case only numerical values can be created for a local column; supported data types are DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_LONGLONG, DT_DOUBLE, DT_COMPLEX, DT_DCOMPLEX.

a) implicit representation with simple numerical data types

If the sequence representation is 'implicit_constant', 'implicit_linear', or 'implicit_saw' only the generation parameters need to be stored. The number and meaning of the generation parameters depends on the type of sequence representation and is independent of the number of values generated for the local column.

This paragraph explains the SVCVAL_SPS in case the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity is of type DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_LONGLONG, or DT_DOUBLE.

For 'implicit_*constant'* all values x_n of the local column are constant and only one generation parameter p_1 needs to be stored, representing the constant value. It is stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n will have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

POINT will be -1, ORD will be 1, STRING_VALUE and BLOB_VALUE will contain NULL. One row of the table is used to store the complete specification of the local column, and its structure is:

Table 20 - SVCVAL_SPS for implicit_constant values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL

For *'implicit_linear'* all values x_n of the local column are on a straight line given by $x_n=p_1+(n-1)*p_2$ and only two generation parameters p_1 and p_2 need to be stored. They are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n will have; the conversion of the precision to/from the API data types is done during reading and writing of the value.
POINT will be -1, ORD will be used to distinguish between those two parameters, and STRING_VALUE and BLOB_VALUE will contain NULL. Two rows of the table are used to store the complete specification of the local column, and the structure is:

Table 21 - SVCVAL_SPS for implicit_linear values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL

<sID> and <cID> entries are the same in both rows.

For *'implicit_saw'* all values x_n of the local column build a sawtooth-curve according to $x_n=p_1+((n-1)*\mathrm{mod}(p_3-p_1)/p_2)*p_2$ and three generation parameters p_1, p_2, and p_3 need to be stored. They are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n will have; the conversion of the precision to/from the API data types is done during reading and writing of the value.
POINT will be -1, ORD will be used to distinguish between those three parameters, and STRING_VALUE and BLOB_VALUE will contain NULL. Three rows of the table are used to store the complete specification of the local column, and the structure is:

Table 22 - SVCVAL_SPS for implicit_saw values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
<sID>	<cID>	-1	3	NULL	<p_3>	NULL	NULL

<sID> and <cID> entries are the same in all three rows.

b) raw data preprocessing with simple numerical data types

If the sequence representation is 'raw_linear', 'raw_polynomial', or 'raw_linear_calibrated' the raw data and the generation parameters need to be stored. The number and meaning of the generation parameters depends on the type of raw data preprocessing and is independent of the number of values generated for the local column. The number of raw data matches the number of values generated for the local column.

This paragraph explains the SVCVAL_SPS in case the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity is of type DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_LONGLONG, or DT_DOUBLE; this results in each generation parameter and each raw value being a simple numerical value.

For *'raw_linear'* all values x_n of the local column are linearly mapped from the raw values r_n according to $x_n=p_1+p_2*r_n$ and only two generation parameters p_1 and p_2 need to be stored besides the raw values. The parameters and the raw values are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n and r_n have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters.

Each raw value is also stored in one row of the table. ORD is set to 0, POINT will be used to identify the measured point (ranging from 1..N), STRING_VALUE and BLOB_VALUE will contain NULL. Thus the structure is:

Table 23 - SVCVAL_SPS for raw_linear simple numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
<sID>	<cID>	n	0	<flag>	<r_n>	NULL	NULL

<sID> and <cID> entries are the same in all rows.

For *'raw_polynomial'* all values x_n of the local column are mapped from the raw values r_n through a k-order polynomial according to $x_n = p_2 + p_3*r_n + p_4*r_n^2 + p_{2+k}*r_n^k$ with $k=p_1$, and k+2 generation parameters p_1 .. p_{2+k} need to be stored besides the raw values. The parameters and the raw values are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n and r_n have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters.

Each raw value is also stored in one row of the table. ORD is set to 0, POINT will be used to identify the measured point (ranging from 1..N), STRING_VALUE and BLOB_VALUE will contain NULL. Thus the structure is:

Table 24 - SVCVAL_SPS for raw_polynomial simple numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	-1	k+2	NULL	<p_{k+2}>	NULL	NULL
<sID>	<cID>	n	0	<flag>	<r_n>	NULL	NULL

<sID> and <cID> entries are the same in all rows.

For *'raw_linear_calibrated'* all values x_n of the local column are mapped from the raw values r_n through a specific linear procedure according to $x_n = (p_1 + p_2*r_n)*p_3$, and only three generation parameters p_1 .. p_3 need to be stored besides the raw values. The parameters and the raw values are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n and r_n have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters.

Each raw value is also stored in one row of the table. ORD is set to 0, POINT will be used to identify the measured point (ranging from 1..N), STRING_VALUE and BLOB_VALUE will contain NULL. Thus the structure is:

Table 25 - SVCVAL_SPS for raw_linear_calibrated simple numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
<sID>	<cID>	-1	3	NULL	<p_3>	NULL	NULL
<sID>	<cID>	n	0	<flag>	<r_n>	NULL	NULL

<sID> and <cID> entries are the same in all rows.

b) raw data preprocessing with structured numerical data types

If the sequence representation is 'raw_linear', 'raw_polynomial', or 'raw_linear_calibrated' and the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity is DT_COMPLEX or DT_DCOMPLEX, each of the raw values is a structured numerical value, composed of two floating point numbers ('r', 'i'; real and imaginary part). Therefore two rows of the table are required to store each of the raw values. The way such local columns are stored in this table depends on the sequence representation.

For *'raw_linear'* all values x_n of the local column are linearly mapped from the raw values r_n according to $x_n=p_1+p_2*r_n$ and only two generation parameters p_1 and p_2 need to be stored besides the raw values. The parameters and the raw values are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n and r_n have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters.

Each raw value is stored in two rows of the table. ORD is used to distinguish between real and imaginary part of the raw values (set to 0 for the real part and to 1 for the imaginary part), POINT will be used to identify the measured point (ranging from 1..N), and STRING_VALUE and BLOB_VALUE will contain NULL. Thus the structure is (with <r[r_n]> being the real part and <i[r_n]> being the imaginary part of the n^{th} raw value r_n; n=0..N-1):

Table 26 - SVCVAL_SPS for raw_linear structured numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
<sID>	<cID>	1	0	<flag>	<r[r_0]>	NULL	NULL
<sID>	<cID>	1	1	NULL	<i[r_0]>	NULL	NULL
...	...	...	...	...	...	...	...

<sID>	<cID>	N	0	<flag>	<r[r_{N-1}]>	NULL	NULL
<sID>	<cID>	N	1	NULL	<i[r_{N-1}]>	NULL	NULL

<sID> and <cID> entries are the same in all rows. The <flag> entry associated with each complex raw value, is only specified at its real part.

For *'raw_polynomial'* all values x_n of the local column are mapped from the raw values r_n through a k-order polynomial according to $x_n = p_2 + p_3 * r_n + p4 * r_n^2 + p_{2+k} * r_n^k$ with $k = p_1$, and k+2 generation parameters p_1 .. p_{2+k} need to be stored besides the raw values. The parameters and the raw values are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n and r_n have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters.

Each raw value is stored in two rows of the table. ORD is used to distinguish between real and imaginary part of the raw values (set to 0 for the real part and to 1 for the imaginary part), POINT will be used to identify the measured point (ranging from 1..N), and STRING_VALUE and BLOB_VALUE will contain NULL. Thus the structure is (with <r[r_n]> being the real part and <i[r_n]> being the imaginary part of the nth raw value r_n; n=0..N-1):

Table 27 - SVCVAL_SPS for raw_polynomial structured numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	-1	k+2	NULL	<p_{k+2}>	NULL	NULL
<sID>	<cID>	1	0	<flag>	<r[r_0]>	NULL	NULL
<sID>	<cID>	1	1	NULL	<i[r_0]>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	N	0	<flag>	<r[r_{N-1}]>	NULL	NULL
<sID>	<cID>	N	1	NULL	<i[r_{N-1}]>	NULL	NULL

<sID> and <cID> entries are the same in all rows. The <flag> entry associated with each complex raw value, is only specified at its real part.

For *'raw_linear_calibrated'* all values x_n of the local column are mapped from the raw values r_n through a specific linear procedure according to $x_n = (p_1 + p_2 * r_n) * p_3$, and only three generation parameters p_1 .. p_3 need to be stored besides the raw values. The parameters and the raw values are stored within the NUMBER_VALUE column. The storage precision is independent of the data type that the values x_n and r_n have; the conversion of the precision to/from the API data types is done during reading and writing of the value.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters.

Each raw value is stored in two rows of the table. ORD is used to distinguish between real and imaginary part of the raw values (set to 0 for the real part and to 1 for the imaginary part), POINT will be used to identify the measured point (ranging from 1..N), and STRING_VALUE and BLOB_VALUE will contain NULL. Thus the structure is (with <r[r_n]> being the real part and <i[r_n]> being the imaginary part of the n^{th} raw value r_n; n=0..N-1):

Table 28 - SVCVAL_SPS for raw_linear_calibrated structured numerical values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
<sID>	<cID>	-1	3	NULL	<p_3>	NULL	NULL
<sID>	<cID>	1	0	<flag>	<r[r_0]>	NULL	NULL
<sID>	<cID>	1	1	NULL	<i[r_0]>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	N	0	<flag>	<r[r_{N-1}]>	NULL	NULL
<sID>	<cID>	N	1	NULL	<i[r_{N-1}]>	NULL	NULL

<sID> and <cID> entries are the same in all rows. The <flag> entry associated with each complex raw value, is only specified at its real part.

9.2.10.3 External Local Columns and Generation Parameters

This case is identified by a sequence representation from the subset {raw_linear_external, raw_polynomial_external, raw_linear_calibrated_external}. The raw values are stored in an external component file, and only the generation parameters need to be stored in the SVCVAL_SPS table. The number of raw values matches the number of values generated for the local column. Generation parameters are always of type DT_DOUBLE and thus are applied in the same way for all cases where the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity is of type DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_LONGLONG, DT_DOUBLE, DT_COMPLEX, or DT_DCOMPLEX. The number and meaning of the generation parameters depends on the sequence representation type and is independent of the number of values generated for the local column.

For *'raw_linear_external'* all values x_n of the local column are linearly mapped from the external raw values r_n according to $x_n=p_1+p_2*r_n$ and only two generation parameters p_1 and p_2 need to be stored in SVCVAL_SPS. The generation parameters are stored within the NUMBER_VALUE column.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters. FLAG, STRING_VALUE, and BLOB_VALUE will contain NULL. Thus the structure is:

Table 29 - SVCVAL_SPS for raw_linear_external values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL

<sID> and <cID> entries are the same in both rows.

For *'raw_polynomial_external'* all values x_n of the local column are mapped from the external raw values r_n through a k-order polynomial according to $x_n = p_2 + p_3{*}r_n + p_4{*}r_n^2 + p_{2+k}{*}r_n^k$ with $k=p_1$, and k+2 generation parameters p_1 .. p_{2+k} need to be stored in SVCVAL_SPS. The generation parameters are stored within the NUMBER_VALUE column.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters. FLAG, STRING_VALUE, and BLOB_VALUE will contain NULL. Thus the structure is:

Table 30 - SVCVAL_SPS for raw_polynomial_external values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
...	...	...	...	...	...	...	...
<sID>	<cID>	-1	k+2	NULL	<p_{k+2}>	NULL	NULL

<sID> and <cID> entries are the same in all rows.

For 'raw_linear_calibrated_external' all values x_n of the local column are mapped from the external raw values r_n through a specific linear procedure according to $x_n = (p_1 + p_2{*}r_n){*}p_3$, and only three generation parameters p_1 .. p_3 need to be stored in SVCVAL_SPS. The generation parameters are stored within the NUMBER_VALUE column.

Each parameter is stored in one row of the table; POINT is -1 in this case and ORD is used to identify the parameters. FLAG, STRING_VALUE, and BLOB_VALUE will contain NULL. Thus the structure is:

Table 31 - SVCVAL_SPS for raw_linear_calibrated_external values

SUBMATRIXID	LOCALCOLUMNID	POINT	ORD	FLAG	NUMBER_VALUE	STRING_VALUE	BLOB_VALUE
<sID>	<cID>	-1	1	NULL	<p_1>	NULL	NULL
<sID>	<cID>	-1	2	NULL	<p_2>	NULL	NULL
<sID>	<cID>	-1	3	NULL	<p_3>	NULL	NULL

<sID> and <cID> entries are the same in all rows.

9.3 Storage of the Attribute Values

9.3.1 The Data Types of the Database Columns

Attributes are given an ASAM ODS data type each. Their values will be stored within the physical storage in tables of the underlying database system. As data types of existing database systems do not match the ASAM ODS data types exactly, a mapping is required.

Currently the ASAM ODS data types are mapped to SQL92 data types (as far as a corresponding data type is available) according to the table below. Also given in that table are the data types currently used in Oracle and Microsoft SQL Server based ODS server installations. These database types are to be used when creating the columns for the attribute values in new ASAM ODS installations. Already existing ASAM ODS installations may keep the types as they were created.

Table 32 - Mapping of ASAM ODS data types to database systems' data types

ODS data type	SQL92	currently used in Oracle	currently used in MS SQL server
T_FLOAT T_COMPLEX.r and .i S_FLOAT S_COMPLEX.r and .i	REAL	NUMBER	REAL
T_DOUBLE T_DCOMPLEX.r and .i S_DOUBLE S_DCOMPLEX.r and .i	DOUBLE PRECISION	NUMBER	FLOAT
T_BOOLEAN S_BOOLEAN	BIT	NUMBER(1)	SMALLINT
T_BYTE S_BYTE	SMALLINT	NUMBER(3)[1]	SMALLINT
T_SHORT S_SHORT	SMALLINT	NUMBER(5)[1]	SMALLINT
T_ENUM T_LONG S_LONG	INTEGER	NUMBER(10,0)	INT
T_LONGLONG S_LONGLONG		NUMBER(20,0)	BIGINT
T_STRING T_EXTERNALREFERENCE .description, .mimeType, and .location S_STRING S_EXTERNALREFERENCE .description, .mimeType, and .location	VARCHAR	VARCHAR2(AFLEN+1 char)	NVARCHAR(AFLEN+1)
T_DATE S_DATE	TIMESTAMP	VARCHAR2(n char) or DATE or TIMESTAMP(3)[2]	DATETIME
T_BYTESTR S_BYTESTR		LONGRAW or BLOB or CLOB	IMAGE

T_BLOB.header	VARCHAR	VARCHAR2(AFLEN+1 char)	NVARCHAR(aflen+1)
T_BLOB.bytestream		LONGRAW or BLOB or CLOB	IMAGE

Notes to the table:

- AFLEN represents the value specified for the attribute in the SVCATTR table.
- [1]: other precisions are possible.
- [2]: customer-specific, depending on intended maximum length.

9.3.2 Storage of Single Values

Most attributes are single value attributes with data type enumeration values in the range of 1...8. These values are stored directly in the column given by DBCNAME in the table SVCATTR. The ASAM ODS data type must correspond with the data type of the database engine.

The storage of attributes with the data type enumeration DT_DATE (=10) is described in section 9.3.5 below.

Other data types will not be stored as single values (see chapter 3).

9.3.3 Storage of ByteStream Values

The values of the attributes with the data type enumeration DT_BYTESTR (=11) are stored in a column of the corresponding entity table with a binary raw data type (e.g. 'LONGRAW' for Oracle database servers). The length of the bytestream corresponds with the length of the 'LONGRAW' field. Oracle database servers are restricted to one column of 'LONGRAW' per table; so only one application attribute with data type enumeration DT_BYTESTR or DT_BLOB is allowed for each application element.

9.3.4 Storage of Blob Values

The values of the attributes with the data type enumeration DT_BLOB (=12) are stored in two columns of the table.

The header of the blob is stored in a column whose name is given by DBCNAME of the table SVCATTR. The database data type of this column is a string data type (e.g. 'VARCHAR2' for Oracle database servers).

The body of the blob attribute is given in a separate column, with a fixed name, called 'Blob'. This column has a binary raw data type (e.g. 'LONGRAW' for Oracle database servers).

If a Blob value shall be undefined, both columns must be set to NULL.

This specification restricts the number of application attributes with blob data type to a maximum of one per application element.

Moreover in case of Oracle database servers, a maximum of one application attribute per application element may be assigned a data type enumeration of DT_BLOB or DT_BYTESTR.

9.3.5 Storage of Date and Time Values

The standard way for storing application attribute values with the data type enumeration DT_DATE in ASAM ODS is using a string. The syntax of a T_DATE value is described in section 3.5.2.

An unlimited resolution is available with this storage format; however with regard to the storage space required this is not optimal. Therefore a server may internally use date and time types that are available in the database used (e.g. the Oracle 'DATE' and 'TIMESTAMP' data types are preferably used in Oracle databases). As this impacts the data type of a column in the corresponding application element's table, and as changing the data type of a column that already contains values without losing them is not trivial, the decision on the data type for date and time storage should be done when specifying the application model.

Note: Whatever data type is used for internal storage must be hidden from the user. The ASAM ODS server must provide date and time information as a date value represented on a string as specified above and will always receive such string values by a client application. This requires that an ASAM ODS server must be able to handle all database specific date and time types as well as the ASAM ODS string representation and must be able to convert between the ASAM ODS string representation and any database specific type (how to convert is implicitly given by the data type of the column that contains an application attribute of type DT_DATE; this data type may have been set manually by an administrator).

9.3.6 STORAGE OF ARRAY VALUES AND OBJECT VALUES FOR ATTRIBUTES

In ASAM ODS attribute values may be an array of values or objects (e.g. DS_BYTE, DT_EXTERNALREFERENCE), both providing more than one element per attribute value.

The values of application attributes of such kind are stored in a separate table. Thus, in case an application element contains single-element application attributes as well as multiple-element application attributes, the instances are stored in two tables.

- One of the tables is named according to the name specified by DBTNAME in SVCENT; it contains a column for each single-element application attribute which then hold the values for each of the instances.
- The other table is named according to the name specified by DBTNAME in SVCENT, extended by "_ARRAY" (**Attention**: this further restricts the maximum length of table names for all application elements containing multiple valued application attributes!); it contains the values of all multiple-element application attributes.

While the first table does not show any specifics, the second table is special in its structure. It has

- one column for the instance id (called IID and being of the same type as the ID-column in the table given by DBTNAME)
- one column for the ordinal number of the array- or object element (called ORD, always starting with 0, being a non-negative integer number) and
- one column for each multiple-element application attribute, holding the values of that attribute. The name of this column is given by DBCNAME in the table SVCATTR. The data type of the values in this column corresponds with the data type given by ADTYPE in the table SVCATTR.

The values of multiple-element application attributes of one instance are stored in several consecutive rows of this table. The number of rows N an instance requires is determined by the maximum of the numbers of array/object elements of all of its multiple-element attributes. For all of these rows the IID is constant while ORD counts up, starting from 0 and ending at N-1. It could happen that even a huge number of table cells remain empty.

EXAMPLE: TABLE: TEST_ARRAY

Given an application element "T1" with DBTNAME="Test" and with two multiple-element application attributes ATT1 and ATT2. ATT1 specifies ADTYPE=DS_DOUBLE (=21) and DBCNAME="DOUBLE_SEQ". ATT2 specifies ADTYPE=DS_LONG (=20) and DBCNAME="LONG_SEQ".

Given two instances of this application element, whose sequences differ in length. The first instance with IID=1 has a sequence of 4 double values and a sequence of 2 long values. The second instance with IID=2 has a sequence of 2 double values and a sequence of 4 long values.

In this case the table with the two multiple-element application attributes is named "Test_ARRAY" and has the following structure and content:

```
IID    ORD   DOUBLE_SEQ   LONG_SEQ
1      0     1.0          2
1      1     2.0          4
1      2     3.0
```

```
1      3     4.0
2      0     1.0          2
2      1     2.0          4
2      2                  6
2      3                  8
```

The attribute values of objects (e.g. with the data type enumeration DT_COMPLEX, DT_DCOMPLEX, DT_EXTERNALREFERENCE) have a predefined number of elements with predefined meaning; their ordinal number for each element is specified in the following table:

Table 33 - Object type application attributes

Type	Field	Ordinal Number	Field data type
T_EXTERNALREFERENCE	description	0	T_STRING
T_EXTERNALREFERENCE	mimeType	1	T_STRING
T_EXTERNALREFERENCE	location	2	T_STRING
T_COMPLEX	r	0	T_FLOAT
T_COMPLEX	i	1	T_FLOAT
T_DCOMPLEX	r	0	T_DOUBLE
T_DCOMPLEX	i	1	T_DOUBLE

Fields which are not used in the object (e.g. mimeType) will be stored with a NULL-value in the value cell; the corresponding IID and ORD cells will show the correct values.
In case object sequences are to be stored the ordinal number continuously increments until the last field of the last value in the sequence.

9.3.6.1 ACCESS THROUGH THE RPC-API

The ASAM ODS RPC-API is not able to handle attributes of array or object type.

In case an ASAM ODS server with RPC-API shall be used, a modification of the application model (and possibly of already existing values in the physical storage) is required, so that there are no multiple-element application attributes.

As soon as an application model has been created that contains application elements with multiple-element application attributes, it cannot be accessed by the RPC-API anymore.

9.3.7 STORAGE OF ENUMERATION TYPE ATTRIBUTES

In case the data type ADTYPE of an application attribute is DT_ENUM (=30) or DS_ENUM (=31), one or more values of data type T_LONG are written to the physical storage.

For an application to present the human-readable string instead of the enumeration value, it must look for the name of the enumeration (which is given in SVCATTR in the column ENUMNAME of the application attribute) and then find the string representation ITEMNAME corresponding to the value of the application attribute (which is done by fetching from table SVCENUM the ITEMNAME in the row where ENUMNAME equals the resolved enumeration name and ITEM equals the value given).

Appropriate methods are provided to an application by the ASAM ODS APIs. However the translation from the numerical application attribute values into their string representation is the responsibility of the application, not of the ASAM ODS server. The same is true for a writing application: It must write numerical values into the server, while usually a user will select one of the enumeration items by their string representations. Translating the string representation into a number is up to the application in the client machine.

For performance reasons it will be wise for the client to cache the enumeration definition, otherwise a lot of client/server network traffic will be produced. If this attribute will be used as sorting criteria, also the value (of type T_LONG) will be used for sorting, not the string representation.

ASAM ODS servers allow the client to create or modify the application model using the OO-API; it also provides methods to modify the name or particular name value pairs of the enumeration. These methods finally modify the application model and may only be used by superusers (nobody else is allowed to modify the application model).

9.3.8 The Inheritance Concept and the Physical Storage

The inheritance concept is mainly provided by ASAM ODS to give further structure to the information stored and to save storage space especially in the physical storage. How this is achieved will be explained in this section.

9.3.8.1 Storage of Application Attributes and Relations

Both, the superclass application element and the subclass application element are specified in SVCENT and their application attributes are specified in SVCATTR. The common application attributes are specified in SVCATTR with the AID of the superclass. The AID of a subclass is only used in SVCATTR when specifying the subclass's specific application attributes. There is one exception though: each subclass additionally specifies the application attribute derived from the base attribute 'id'. This application attribute therefore is specified several times in SVCATTR: once for the superclass and once for each subclass. Though the physical storage in that case would allow to give different application attribute names to it, this should strictly be avoided and is only accepted by ASAM ODS because of existing legacy data.

Based application relations (which may be of relation type FATHER_CHILD or INFO) are distinguished from extended application relations (which may be of relation type INFO) in that they have an entry in the BANAME column of SVCATTR. The relation between superclass and subclass (as an INHERITANCE relation type) is identified simply by the fact that the FAID column of the subclass's application attribute derived from the base attribute 'id' is not empty but contains the AID of the related superclass application element. In case of INHERITANCE type relations the entry in INVNAME must remain empty in the ‘id’ entry for superclass and subclass.

A separate entity table to store the instances is created for each of these application elements.

The entity table for the superclass application element contains columns for all application attributes and application relations that are common to both elements.

The entity table for the subclass application element contains columns only for those application attributes and relations that are not yet contained in the superclass application element plus a column for the attribute derived from the base attribute 'id'. (Please note that ASAM ODS requires to place based application relations into the superclass application element. Thus a subclass application element may only specify extended application relations.)

All attribute and relation values of an instance of the superclass application element will be stored in the superclass entity table.

The attribute and relation values of an instance of the subclass application element will be split between the superclass and the subclass entity tables. All values of common attributes and relations are stored in the superclass entity table while all values of specific subclass attributes and relations (plus the 'id'-value as a key to link the entity tables) are stored in the subclass entity table. It is allowed to add array-type attributes to subclass elements. They are stored in the corresponding array tables (.._ARRAY) of the subclass element.

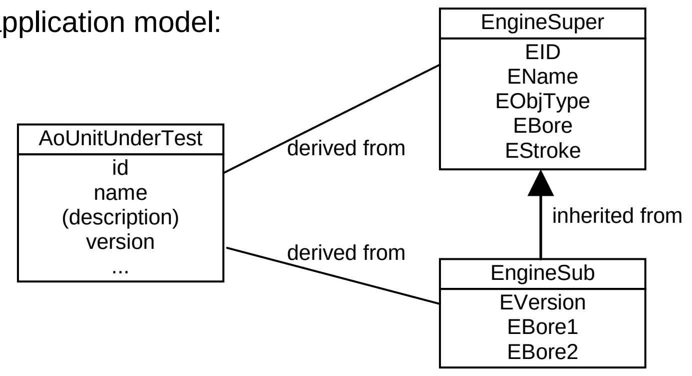
For each inherited application element that represents a subclass, a database view of the tables of the superclass and subclass must be available. These two tables are joined via the value of the application attribute derived from the base attribute 'id', and the instance type, i.e. the common part of an instance has the same 'id'-value in the superclass as the specific part in the inherited application element.

The name of the table implementing the subclass is stored in SVCENT as DBTNAME. The name of the view is derived from the table name, preceded by the prefix "SCV_“ (subclass-view; please note that this naming differs from the service table names in the physical storage, starting with "SVC...").

For inherited application elements no Oracle sequence for an automatic calculation of the next free identifier value is specified in the database; the 'id'-value of the superclass is used instead.

EXAMPLE: PHYSICAL STORAGE WITH INHERITANCE

Given the following application model:



An instance of EngineSuper (with EName="SuperEng", EBore=80.4, EStroke=85.6) and an instance of EngineSub (with EName="SubEng", EBore=80.5, EStroke=85.3, EVersion="2.4", EBore1=81.5, EBore2=79.5) are stored. The identifier (EID) is specified implicitly by the server; also will be the table names and column names.

SVCENT contains the following entries (among others):

AID	ANAME	BID	DBTNAME	SECURITY
...	...	...	...	...
12	EngineSuper	21 (AoUnitUnderTest)	TabEngineSuper	0
13	EngineSub	21 (AoUnitUnderTest)	TabEngineSub	0

SVCATTR contains the following entries (among others):

AID	ATTRNR	AANAME	BANAME	FAID	FUNIT	ADTYPE	AFLEN	DBCNAME	ACLREF	INVNAME	FLAG	ENUM NAME
...	...	...	...			...		...			...	
12	1	EID	id			8		ColEID			5	
12	2	EName	name			1	32	ColEName				
12	3	EObjType	objecttype			8		ColEObjType				
12	4	EBore				7		ColEBore				

12	5	EStroke				7		ColEStroke				
13	1	EID	id	12		8		ColEID			5	
13	2	EVersion	version			1	32	ColEVersion				
13	3	EBore1				7		ColEBore1				
13	4	EBore2				7		ColEBore2				

Note that in the subclass application element the name of the application attribute derived from the base attribute 'id' has been set identical to the corresponding name in the superclass application element.

The table **'TabEngineSuper'** contains all values of the common attributes of all instances of superclass and subclass application elements:

ColEID	ColEName	ColEObjType	ColEBore	ColEStroke	...
...	...	...	...	...	...
27	SuperEng	12	80.4	85.6	...
31	SubEng	13	80.5	85.3	...

The table **'TabEngineSub'** contains the values of the additional attributes of instances of subclass application elements plus the EID value:

ColEID	ColEVersion	ColEBore1	ColEBore2	...
...	...	...	...	...
31	2.4	81.5	79.5	...

The EID value is unique for all instances of the superclass and subclass application elements. With its values being available in both tables a server can build the complete set of attributes of a specific instance of a subclass application element.

To access the sub class instances, the view SCV_TabEngineSub must be created by the following SQL statement:

```
create or replace view SCV_TabEngineSub as
select
   Sup.ColEID,
   Sup.ColEName,
   Sup.ColEObjType,
   Sup.ColEBore,
   Sup.ColEStroke,
   Sub.ColEVersion,
   Sub.ColEBore1,
   Sub.ColEBore2
from TabEngineSuper Sup, TabEngineSub Sub
where Sup.ColEID = Sub.ColEID
and Sup.ColEObjType = 13;
```

This view contains all attributes of the superclass and subclass, but only one ID attribute.

9.3.8.2 Storage of Instance Attributes

Instance attributes are stored with the AID of the corresponding element. Thus if an instance attribute belongs to an instance of the superclass application element, the AID-column in the SVCINST table holds the AID of the superclass application element; if an instance attribute belongs to an instance of the subclass application element, the AID-column in the SVCINST table holds the AID of the subclass application element.

9.4 The Mixed-Mode Storage

In many cases in the automotive testing environment a huge number of measurement data must be stored. For this purpose ASAM ODS provides the base element AoLocalColumn. The RDB based physical storage requires to store them as blob in the table SVCVAL. This may rapidly increase database size and searching for specific data often is extremely time-consuming due to the database restrictions (e.g. Oracle has to process blobs within a database).

For this purpose ASAM ODS has specified an alternative to the pure RDB based storage, allowing to hold mass data in separate files, but placing the description of those external data within the database.

The concept of storing mass data in separate binary files is extended by the concept of segmentation. Segmentation is the procedure to split mass data of one local column into several binary files; this breaks huge binary files into several smaller ones and thereby improves performance, concurrency, and file-handling.

This approach will be explained in this section. It is called the Mixed-Mode storage, a server implementing it is called a Mixed-Mode server, and some remarks on a Mixed-Mode server are given later in this section.

The data stored in separate files are restricted to mass data. Only the attributes derived from the base attributes "values" and "flags" of AoLocalColumn may be put into those files. Therefore only local columns must know about the way values are stored externally in files. For this purpose each local column may refer to one or more instance of an application element of type AoExternalComponent. See also chapter 4 for a description of the ASAM ODS base model.

One external file may contain data of several measurement quantities thus belonging to different local columns. To distinguish them from each other, their position within the file must be specified exactly and must show some regular pattern. The structure of the files depend on the data types that are to be stored within. This section describes the structure for those data types where Mixed-Mode storage is currently available.

It should be noted that the value of the attribute derived from the base attribute 'sequence_representation' finally determines the contents of the external component file as well as the VALBLOB (consisting of the VALUES and the FLAGS elements) in the table SVCVAL. The following table gives an overview of the possible combinations, specifying the content of the VALUES element of VALBLOB, the FLAGS element of VALBLOB and the external component file (if any).

Table 34 - External component files and sequence representation

sequence_representation	**VALUES**	**FLAGS**	**external component files**
explicit	the final values	the flags (if any)	none
implicit_constant implicit_linear implicit_saw	the generation parameters	no flags allowed	none
raw_linear raw_polynomial raw_linear_calibrated	the generation parameters followed by the raw values	the flags (if any)	none
external_component	empty	empty	the final values the flags (if any)
raw_linear_external raw_polynomial_external raw_linear_calibrated_external	the generation parameters	empty	the raw values the flags (if any)

The first three cases are described in section 9.2.4 as no external component file is involved.

In case 'sequence_representation' is given as *'external_component'*, the final values are put in the external component file.

The data type of these values (with which they are presented through the APIs to a user) is determined by the value of the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity; it may correspond to any of DT_STRING, DT_SHORT, DT_FLOAT, DT_BOOLEAN, DT_BYTE, DT_LONG, DT_DOUBLE, DT_LONGLONG, DT_DATE, DT_BYTESTR, DT_BLOB, DT_COMPLEX, DT_DCOMPLEX.

The representation in the external component file may however differ from these data types; it is given by the value of the attribute derived from the base attribute 'value_type' of the corresponding external component.

A server thus must read the raw values from the external component files using 'value_type' and translate them into 'datatype' before presenting them to a user.

In case 'sequence_representation' is either *'raw_linear_external'*, *'raw_polynomial_external'*, or *'raw_linear_calibrated_external'*, the raw values are put in the external component file, and the final values must be computed from them using the generation parameters given in the VALUES element of VALBLOB of the corresponding local column.

The data type of the values of this local column (with which they are presented through the APIs to a user) is determined by the value of the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity; it may correspond to any of DT_SHORT, DT_FLOAT, DT_BYTE, DT_LONG, DT_DOUBLE, DT_LONGLONG, DT_COMPLEX, DT_DCOMPLEX (note that only numerical data types are allowed).

The data type of the raw data is given by the value of the attribute derived from the base attribute 'raw_datatype' of the corresponding local column.

The representation in the external component file may however differ from these data types; it is given by the value of the attribute derived from the base attribute 'value_type' of the corresponding external component (where also only numeric data types will be used).

A server thus must read the raw values from the external component files using 'value_type' and translate them into the corresponding 'raw_datatype'.

External component files may contain values as well as flags. Values and flags may be placed in the same or in different external component files. External component files may contain values of different data types thereby combining data of different local columns. Though not all combinations are allowed, a wide variety of combinations are possible. How these external component files are structured, is described in this section.

9.4.1 Storage of Values of Blob Type

One or more binary files may contain blob values of one local column.

In case the values are blobs, one binary file contains exactly one blob (without the blob header, which is stored in the attribute derived from the base attribute 'description' of the external component). The length information of the blob's byte sequence is not stored in the binary file. Note that this is different from the storage of blobs in SVCVAL and SVCVAL_SPS, where the first 4 bytes of the binary content represent the number of following bytes, which then build up the binary part of the blob.

If more than one blob belongs to the local column, each blob is stored in a separate binary file, and is described by a separate instance of an application element of type AoExternalComponent using some of its attributes:

- The attributes derived from the base attributes 'start_offset', 'block_size', 'valuesperblock', and 'value_offset' are irrelevant and may be omitted.
- The attribute derived from the base attribute 'value_type' specifies the data type used for storing the values. For blob type values it is the enumeration item "dt_blob".
- The attribute derived from the base attribute 'filename_url' or the instance referenced by the relation derived from 'ao_values_file' provides the name of the file containing the blob.
- The attribute derived from the base attribute 'component_length' specifies the length of the blob (number of bytes). It is at the same time the length of the binary file, as no further information may be contained in that file.
- The attribute derived from the base attribute 'ordinal_number' specifies the sequence of blobs within the local column if the local column has more than one external component. The external component with 'ordinal_number'=1 contains the first blob, the external component with 'ordinal_number'=2 contains the second blob, and so on, until finally the external component with highest 'ordinal_number' contains the last blob for this local column.

With this information given, the values of local columns of blob types can easily be retrieved:

- Each external component can read the blob value, as it knows the name of the binary file (from 'filename_url' or 'ao_values_file'), that the file contains exactly one blob (from 'value_type'), and the size of the blob (from 'component_length').
- A local column can subsequently collect the blobs from all related external components, as it knows the correct sequence of external components (from 'ordinal_number').

9.4.2 Storage of Values of String Type

One or more binary files may contain string values of one or more local columns.

Binary files containing string values may not contain values of other data types.

Each binary file consists of an optional header part followed by a set of blocks.

In case the values are of string type, (other than with blob type data) more than one value may be stored in one binary file, although all values contained in a file that are of one local column must be placed in the same block of that file next to each other.

One binary file may contain values of more than one local column. On the other hand the values of one local column may be spread over more than one binary file, if appropriate.

Thus an overall structure of the external binary files for storing string type values in a Mixed-Mode environment can be represented by the following figure:

f_1					f_2	..	f_N				
h	b_1	b_2	..	b_{J1}	..	..	h	b_1	b_2	..	b_{JN}

Figure 1 - Mixed-Mode storage: file structure for string type values

with:

- f_1, f_2, .., f_n, .., f_N are the external binary files containing string values of one or more local columns.
- h is the header part of a binary file.
- b_1, b_2, .., b_j, .., b_{Jn} are the blocks within a binary file f_n (n=1..N).
  The number Jn of blocks within a binary file f_n may differ from the number Jm of blocks within a different binary file f_m.
- A block contains one or more values of one local column.
  No two different blocks within a file may contain values of the same local column.
- The values within a block either are contained in bytesets of fixed length (if a value is given for 'ao_bit_count') or are variable-length strings.
  If they are variable-length strings they are always NUL-terminated. A 0x00 (NUL) character separates consecutive values within a block from each other without a gap between them.
  If they are contained in bytesets of fixed length the value of 'ao_bit_count' (which must be a multiple of 8) specifies the length of each byteset; they always start at byte boundaries (the value of 'ao_bit_offset' must be 0 or undefined). Bytesets within a block follow each other without a gap. The bytes of a byteset as read from the file will contain one string each. They may or may not contain a 0x00 byte as string delimiter. If such delimiter is missing the string occupies the complete byteset, if it is present it determines the end of the string, which then is shorter than the byteset.
- Each character of a string is stored as an 8-bit number for values of value_type 'dt_string' or 'dt_string_flags_beo', and as 1..4 byte UTF-8 code for values of value_type 'dt_string_utf8' or 'dt_string_utf8_flags_beo'.

EXAMPLE: IN CASE NO VALUE IS GIVEN FOR 'AO_BIT_COUNT'

2 strings with (ASCII-)contents "ABCDEFG" and "XYZ" are represented by the following hexadecimal sequence
41 42 43 44 45 46 47 00 58 59 5A 00
The total length is 12, thus component_length=12.

The contents of each binary file with regard to one local column is described by an instance of an application element of type AoExternalComponent using some of its attributes:

- The attributes derived from the base attributes 'block_size' and 'value_offset' are irrelevant and may be omitted.
- The attribute derived from the base attribute 'value_type' specifies the data type used for storing the values. For string type values it is the enumeration item "dt_string", "dt_string_utf8", "dt_string_flags_beo", or "dt_string_utf8_flags_beo" (the latter two types are only introduced to distinguish the endian order of the corresponding flags (see below); the byte sequence within the string does not depend on the enumeration item specified here).
- The attribute derived from the base attribute 'filename_url' or the instance referenced by the relation derived from 'ao_values_file' provides the name of the file containing the string values.
- The attribute derived from the base attribute 'component_length' specifies the total length (number of bytes) of all string values contained in the file that belong to the corresponding local column. The NUL-termination characters as well as any unused bytes in bytesets also contribute to this value, so that in case no value is given for 'ao_bit_count' 'component_length' equals the sum of the number of bytes of all contained strings plus the number of contained strings, while in case a value is given for 'ao_bit_count' no such relationship can be given.
- The attribute derived from the base attribute 'start_offset' (number of bytes) specifies the byte position of the start of the first string value for the corresponding local column within the file. This is the length of the header part plus (possibly) the accumulated length of all preceding blocks in the file.
- The attribute derived from the base attribute 'ordinal_number' specifies the sequence of external components within the local column if the local column has more than one external component. The external component with 'ordinal_number'=1 contains the first set of string values, the external component with 'ordinal_number'=2 contains the second set of string values, and so on, until finally the external component with highest 'ordinal_number' contains the last set of string values of this local column.
- The attribute derived from the base attribute 'valuesperblock' may specify the number of string values contained in the file, which belong to the corresponding local column. Note that the value of this attribute may be undefined. If available this may improve server performance; otherwise a server might need to read the whole block and count the number of NUL characters to determine the number of string values in that block.

With this information given, the values of local columns of string types can easily be retrieved:

- Each external component can collect the string values related to its local column, as it knows the name of the binary file (from 'filename_url' or from 'ao_values_file'), that the file contains string values and how they are encoded (from 'value_type'), the start byte position of the string values belonging to the corresponding local column (from 'start_offset'), the total number of bytes it must analyze (from 'component_length'), a potential bit-shift required in case bytesets are the containers of the strings (from 'ao_bit_offset') and that strings are either separated by a NUL-character or stored in bytesets of fixed length (given by 'ao_bit_count'). The 'component_length' always includes the NUL-characters at the end of each string and any unused bytes at the end of bytesets. The number of strings in a component can be retrieved by either determining the number of bytesets or counting the NUL-characters within the component (or directly from 'valuesperblock', if available).

- A local column can subsequently collect the lists of string values from all related external components, as it knows the correct sequence of external components (from 'ordinal_number').

9.4.3 STORAGE OF VALUES OF BYTESTREAM TYPE

One or more binary files may contain bytestream values of one or more local columns.

Each binary file consists of an optional header part followed by a set of blocks.

In case the values are of bytestream type, (other than with blob type data) more than one value may be stored in one binary file, although all values contained in a file that are of one local column must be placed in the same block of that file next to each other.

One binary file may contain values of more than one local column. On the other hand the values of one local column may be spread over more than one binary file, if appropriate.

Thus an overall structure of the external binary files for storing bytestream type values in a Mixed-Mode environment can be represented by the following figure:

f_1					f_2	..	f_N				
h	b_1	b_2	..	b_{J1}	..	..	h	b_1	b_2	..	b_{JN}

Figure 2 - Mixed-Mode storage: file structure for bytestream type values

with:

- f_1, f_2, .., f_n, .., f_N are the external binary files containing bytestream values of one or more local columns.
- h is the header part of a binary file.
- b_1, b_2, .., b_j, .., b_{Jn} are the blocks within a binary file f_n (n=1..N).
  The number Jn of blocks within a binary file f_n may differ from the number Jm of blocks within a different binary file f_m.
- A block contains one or more values of one local column.
  No two different blocks within a file may contain values of the same local column.
- The values within a block are preceded by a four-byte length information which is interpreted as an unsigned integer value; its endian order depends on the value given by the attribute derived from the base attribute 'value_type'.

EXAMPLE:

In case of"dt_bytestr_beo", 2 bytestreams with (ASCII-)contents "ABCDEFG" and "XYZ" are represented by the following hexadecimal sequence
00 00 00 07 41 42 43 44 45 46 47 00 00 00 03 58 59 5A
The total length is 18, thus component_length=18.

The contents of each binary file with regard to one local column is described by an instance of an application element of type AoExternalComponent using some of its attributes:

- The attributes derived from the base attributes 'block_size' and 'value_offset' are irrelevant and may be omitted.
- The attribute derived from the base attribute 'value_type' specifies the data type used for storing the values. For bytestream type values it is the enumeration item "dt_bytestr_leo" or "dt_bytestr_beo" (these two types are only introduced to distinguish the endian order of the corresponding flags (see below) and of the

length information of the bytestream; the byte sequence within the bytestream does not depend on the enumeration item specified).
Note that 'value_type' may also show a value of "dt_bytestr" in legacy data. Such value is deprecated as its specific handling in ATF files may lead to some confusion. For legacy data with a "dt_bytestr" value the four-byte length information in external component files must be interpreted in **big endian order**.

- The attribute derived from the base attribute 'filename_url' or the instance referenced by the relation derived from 'ao_values_file' provides the name of the file containing the bytestream values.
- The attribute derived from the base attribute 'component_length' specifies the total length (number of bytes) of all bytestream values contained in the file that belong to the corresponding local column. The four-byte length information of each bytestream also contributes to this value, so that component_length equals the sum of the number of bytes of all contained bytestreams plus four times the number of contained bytestreams.
- The attribute derived from the base attribute 'start_offset' (number of bytes) specifies the byte position of the start of the first bytestream value (precisely: the start of its length information) for the corresponding local column within the file. This is the length of the header part plus (possibly) the accumulated length of all preceding blocks in the file.
- The attribute derived from the base attribute 'ordinal_number' specifies the sequence of external components within the local column if the local column has more than one external component. The external component with 'ordinal_number'=1 contains the first set of bytestream values, the external component with 'ordinal_number'=2 contains the second set of bytestream values, and so on, until finally the external component with highest 'ordinal_number' contains the last set of bytestream values of this local column.
- The attribute derived from the base attribute 'valuesperblock' may specify the number of bytestream values contained in the file, which belong to the corresponding local column. Note that the value of this attribute may be undefined. If available this is redundant information and may improve server performance; otherwise a server must read the whole block to determine the number of bytestream values in that block.

With this information given, the values of local columns of bytestream types can easily be retrieved:

- Each external component can collect the bytestream values related to its local column, as it knows the name of the binary file (from 'filename_url' or from 'ao_values_file'), that the file contains bytestream values (from 'value_type'), the start byte position of the bytestream values belonging to the corresponding local column (from 'start_offset'), the total number of bytes it must analyze (from 'component_length'), and that bytestreams are preceded by a four-byte length information at the beginning of each bytestream. The number of bytestreams in a component can be retrieved by reading the whole block and counting the bytestreams (or directly from 'valuesperblock', if available).
- A local column can subsequently collect the lists of bytestream values from all related external components, as it knows the correct sequence of external components (from 'ordinal_number').

9.4.4 STORAGE OF VALUES OF NUMERICAL TYPES

A set of binary files may contain integer or floating point values of a set of local columns.

Each binary file consists of an optional header part followed by a set of blocks.

Each block may contain values of different local columns. A local column may have more than one value in the same block, although all values of the same local column that are contained in the same block must be consecutively placed next to each other, thereby building a sub-block. The structure of each block within a binary file must be the same; this means it must contain the same number of sub-blocks and the position of a sub-block within each block must be the same if it relates to the same local column.

Thus an overall structure of the external binary files for storing numerical values in a Mixed-Mode environment can be represented by the following figure:

f_1														f_2	..	f_N													
h	b_1				b_2				..	b_{J1}				..	..	h	b_1				b_2				..	b_{JN}			
h	s_1	s_2	..	s_{K1}	s_1	s_2	..	s_{K1}	..	s_1	s_2	..	s_{K1}	..	..	h	s_1	s_2	..	s_{KN}	s_1	s_2	..	s_{KN}	..	s_1	s_2	..	s_{KN}

Figure 3 - Mixed-Mode storage: file structure for numerical values

with:

- f_1, f_2, .., f_n, .., f_N are the external binary files containing numerical values of different local columns.
- h is the header part of a binary file.
- b_1, b_2, .., b_j, .., b_{Jn} are the blocks within a binary file f_n (n=1..N).
  The number Jn of blocks within a binary file f_n may differ from the number Jm of blocks within a different binary file f_m.
- s_1, s_2, .., s_k, .. s_{Kn} are the sub-blocks within one block of a binary file f_n (n=1..N).
  The number of sub-blocks within each block of the same binary file must be the same.
  The number Kn of sub-blocks within a block of one binary file f_n may differ from the number Km of sub-blocks within a block of another binary file f_m.
- A sub-block contains one or more values of one local column.
  Within the same binary file f_n each sub-block s_k within any block b_j must contain the same number of values of the same local column as the sub-block s_k at the same position in another block b_l (with j, l $\in$ [1,Jn]).
  No two different sub-blocks within a block may contain values of the same local column.

All values contained in sub-blocks s_k within one file belong to the same local column; they contribute to the local column in exactly the sequence as they appear in the file.

If values of a local column are spread over more than one binary file, the attribute derived from the base attribute 'ordinal_number' of AoExternalComponent specifies how the final sequence of values of the local column is created from the individual value sequences of each file: it starts with the values from the file referenced by the component with 'ordinal_number'=1 and subsequently continues with rising 'ordinal_number'.

AoExternalComponent provides several attributes that may be used in its corresponding application elements to exactly specify where values of a local column are positioned in the binary file:

- Each local column knows about all external components that contain its values. The sequence in which external components are used to access the values, is given by the attribute derived from the base attribute 'ordinal_number' (if the local column has more than one external component). The external component with 'ordinal_number'=1 contains the first set of values, the external component with 'ordinal_number'=2 contains the second set of values, and so on, until finally the external component with highest 'ordinal_number' contains the last set of values of this local column.
- Each external component knows about exactly one binary file, given by the attribute derived from the base attribute 'filename_url' or by the instance referenced by the relation derived from 'ao_values_file'. The sequence of values of the external component is contained within this file.
- Each external component knows about the length of the header part in that file, given by the attribute derived from the base attribute 'start_offset' (number of bytes). It must skip that number of bytes from the beginning of the file.
- Each external component knows about the position of the sub-block (containing the relevant values) within each block, given by the attribute derived from the base attribute 'value_offset'. Again it is specified as a number of bytes; the external component must skip that number of bytes from the beginning of each block.
- Each external component knows about the data type of the values to be accessed, given by the attribute derived from the base attribute 'value_type'. It also knows about the way values are physically stored, which includes information about the number of bits/bytes used for storing that data type in the binary file, about the endian order if more than 8 bits are used to store a value, and about a potential shift of one or more bits that occurred to the values; such information is given based on the attributes derived from 'ao_bit_count' and 'ao_bit_offset' (see section 4.4.5 and below).
  Note that 'value_type' specifies the physical representation of one value in the binary file and is not necessarily identical to the value representation within the ASAM ODS server and across the API. The latter is specified by the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity.
- Each external component knows about the number of values within each sub-block, given by the attribute derived from the base attribute 'valuesperblock'. It thus can extract exactly that number of values (i.e. the sub-block) from the current block.
- Each external component knows about the size of one block, given by the attribute derived from the base attribute 'block_size' (number of bytes). It therefore can move ahead to the beginning of the same sub-block within the next block by adding 'block_size' bytes to the start pointer of the current sub-block. It can then extract another set of values (another sub-block).
- Each external component knows about the total number of values of its corresponding local column segment expected to be in that file, given by the attribute derived from the base attribute 'component_length'. Thus it knows when to move on to the next sub-block and when to stop.
- The local column may collect the value lists from all its external components and build up its final sequence of values, based upon the information in the attribute derived from the base attribute 'ordinal_number'.

The following paragraphs describe the physical representation of values of numerical data types on the external binary file. They differ depending on the value type (which is given by the attribute derived from the base attribute 'value_type' of the external component). Note that there is a difference between data types used in component files (the value type, found in typespec_enum) and ASAM ODS internal data types (found in datatype_enum). An ODS server will convert between file data types and internal data types. See also section 4.4.5 for related information.

9.4.4.1 VALUES OF VALUE TYPE DT_BOOLEAN OR DT_BOOLEAN_FLAGS_BEO

In a binary file each boolean shall be represented by one bit (0=false, 1=true). These bits are filled into a sequence of bytes. If the number of bits is not sufficient to fill a byte completely, then the rest of that byte shall remain unused. The byte is filled from left to right, i.e. beginning with the most significant bit. There is no difference in the bit sequence between dt_boolean and dt_boolean_flags_beo; these two types are only introduced to distinguish the endian order of the corresponding flags (see below).

In the byte sequence (by_1, by_2, by_3,...) the n-th boolean (n=1...Nbi) can be located using the following integer formula. Assuming the bit positions in a byte are numbered (again from left to right) 7,6,5,4,3,2,1,0 then the n-th boolean is in the Iby-th byte at bit position Ibi:

    Iby = (n+7) / 8
    Ibi = Iby*8 - n

The number of used bytes (Nby) can be calculated from the number of booleans (Nbi) by integer arithmetic

    Nby = 1 + (Nbi-1) / 8

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of bits used (i.e. the number of given booleans) and thus equals Nbi.

9.4.4.2 VALUES OF VALUE TYPE DT_BYTE OR DT_BYTE_FLAGS_BEO

This unsigned data type is represented simply by 8 bits. Thus the value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of byte values which are stored in the external file and belong to that component.

There is no difference in the byte sequence between dt_byte and dt_byte_flags_beo; these two types are only introduced to distinguish the endian order of the corresponding flags (see below).

9.4.4.3 VALUES OF VALUE TYPE DT_SBYTE OR DT_SBYTE_FLAGS_BEO

This signed data type is represented simply by 8 bits. Thus the value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of byte values which are stored in the external file and belong to that component.

There is no difference in the byte sequence between dt_sbyte and dt_sbyte_flags_beo; these two types are only introduced to distinguish the endian order of the corresponding flags (see below).

9.4.4.4 VALUES OF VALUE TYPE DT_SHORT OR DT_SHORT_BEO

dt_short and dt_short_beo represent 16-bit signed integer data, stored using 16 bits in the external binary file.

- value type=dt_short: When reading/writing bytes from/to the file i/o stream, the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=dt_short_beo: When reading/writing bytes from/to the file i/o stream, the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.5 VALUES OF VALUE TYPE DT_USHORT OR DT_USHORT_BEO

dt_ushort and dt_ushort_beo represent 16-bit unsigned integer data, stored using 16 bits in the external binary file.

- value type=dt_ushort: When reading/writing bytes from/to the file i/o stream, the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=dt_ushort_beo: When reading/writing bytes from/to the file i/o stream, the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.6 VALUES OF VALUE TYPE DT_LONG OR DT_LONG_BEO

dt_long and dt_long_beo represent 32-bit signed integer data, stored using 32 bits in the external binary file.

- value type=dt_long: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type= dt_long_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.7 VALUES OF VALUE TYPE DT_ULONG OR DT_ULONG_BEO

dt_ulong and dt_ulong_beo represent 32-bit unsigned integer data, stored using 32 bits in the external binary file.

- value type=dt_ulong: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=dt_ulong_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.8 VALUES OF VALUE TYPE DT_LONGLONG OR DT_LONGLONG_BEO

dt_longlong and dt_longlong_beo represent 64-bit signed integer data, stored using 64 bits in the external binary file.

- value type=dt_longlong: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=dt_longlong_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.9 VALUES OF VALUE TYPE DT_BIT_INT OR DT_BIT_INT_BEO

dt_bit_int and dt_bit_int_beo represent signed integer data, stored using a variable number of bits in the external binary file. The number of bits used for storage is given by the attribute derived from the base attribute 'ao_bit_count'. The values may be stored at non-byte boundaries in which case the attribute derived from the base attribute 'ao_bit_offset' provides the information on the number of bit-shifts needed to finally set the value right-aligned into some bigger-sized integer variable. See section 4.4.5 for more information on accessing values of these types.

- value type=dt_bit_int: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type= dt_bit_int_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.10 VALUES OF VALUE TYPE DT_BIT_UINT OR DT_BIT_UINT_BEO

dt_bit_uint and dt_bit_uint_beo represent unsigned integer data, stored using a variable number of bits in the external binary file. The number of bits used for storage is given by the attribute derived from the base attribute 'ao_bit_count'. The values may be stored at non-byte boundaries in which case the attribute derived from the base attribute 'ao_bit_offset' provides the information on the number of bit-shifts needed to finally set the value right-aligned into some bigger-sized integer variable. See section 4.4.5 for more information on accessing values of these types.

- value type=dt_bit_uint: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type= dt_bit_uint_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of integer values which are stored in the external file and belong to that component.

9.4.4.11 VALUES OF VALUE TYPE IEEEFLOAT4 OR IEEEFLOAT4_BEO

ieeefloat4 and ieeefloat4_beo represent 4 byte floating point data, stored using 32 bits in the external binary file.

- value type=ieeefloat4: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=ieeefloat4_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The location and number of bits of sign, exponent and mantissa is defined according to the IEEE single format specified in „IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE; New York)“.

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of floating point values which are stored in the external file and belong to that component.

Complex single precision values:
In case the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity is DT_COMPLEX (which means values of this quantity are complex numbers), the attribute derived from the base attribute 'value_type' of the corresponding external component must be either ieeefloat4 or ieeefloat4_beo. The physical representation of values in the external binary file must always show pairs of two consecutive 4-byte floating point values, the first one being interpreted as real part, the second one as imaginary part of the complex value. The total number of floating point numbers contained in the file is twice the number of (complex) data values. Even in this case the attribute derived from the base attribute 'component_length' contains the number of 4-byte floating point values. As each data value is represented by two consecutive 4-byte floating point values, the number of data values is half of the value of the attribute derived from the base attribute 'component_length' and this attribute may only contain non-negative even integer values.

9.4.4.12 VALUES OF VALUE TYPE IEEEFLOAT8 OR IEEEFLOAT8_BEO

ieeefloat8 and ieeefloat8_beo represent 8 byte floating point data, stored using 64 bits in the external binary file.

- value type=ieeefloat8: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=ieeefloat8_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The location and number of bits of sign, exponent and mantissa is defined according to the IEEE double format specified in „IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE; New York)".

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of floating point values which are stored in the external file and belong to that component.

Complex double precision values:
In case the attribute derived from the base attribute 'datatype' of the corresponding measurement quantity is DT_DCOMPLEX (which means values of this quantity are complex numbers), the attribute derived from the base attribute 'value_type' of the corresponding external component must be either ieeefloat8 or ieeefloat8_beo. The physical representation of values in the external binary file must always show pairs of two consecutive 8-byte floating point values, the first one being interpreted as real part, the second one as imaginary part of the complex value. The total number of floating point numbers contained in the file is twice the number of (complex) data values. Even in this case the attribute derived from the base attribute 'component_length' contains the number of 8-byte floating point values. As each data value is represented by two consecutive 8-byte floating point values, the number of data values is half of the value of the attribute derived from the base attribute 'component_length' and this attribute may only contain non-negative even integer values.

9.4.4.13 VALUES OF VALUE TYPE DT_BIT_IEEEFLOAT OR DT_BIT_IEEEFLOAT_BEO

dt_bit_ieeefloat and dt_bit_ieeefloat_beo represent floating point data, stored using 32 or 64 bits in the external binary file. They are used if the floating point values do not start on byte boundaries within the file. The size of the floating point number is given by the attribute derived from the base attribute 'ao_bit_count'. The attribute derived from the base attribute 'ao_bit_offset' provides the information on the number of bit-shifts needed to finally set the value right-aligned. See section 4.4.5 for more information on accessing values of these types.

- value type=dt_bit_ieeefloat: When reading/writing bytes from/to the file i/o stream the least significant byte comes first (lower address), the most significant byte comes last (higher address).
- value type=dt_bit_ieeefloat_beo: When reading/writing bytes from/to the file i/o stream the most significant byte comes first, the least significant byte comes last; this order is also known as "Big Endian Order".

The location and number of bits of sign, exponent and mantissa is defined according to the IEEE double format specified in „IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE; New York)“.

The value of the attribute derived from the base attribute 'component_length' of the corresponding external component denotes the number of floating point values which are stored in the external file and belong to that component.

9.4.5 STORAGE OF FLAGS

A binary file may contain value flags for one or more measurement quantities resp. local columns. Flags are placed in a binary file instead of the FLAGS element of VALBLOB if the value of the attribute derived from the base attribute 'sequence_representation' is external_component, raw_linear_external, raw_polynomial_external, raw_linear_calibrated_external. In those cases, the value of the attribute derived from the base attribute 'flags_filename_url' or the instance referenced by the relation derived from the base relation 'ao_flags_file' provide determines the file that contains them. If that attribute is empty, no flags are available at all. The file containing the flags may be (but need not be) the same as the one containing the values.

Each binary file consists of an optional header part followed by a set of blocks. All flags of one local column segment must be placed in the same block of that file next to each other. One binary file may contain flags of more than one local column.

Thus an overall structure of the external binary files for storing flags in a Mixed-Mode environment can be represented by the following figure:

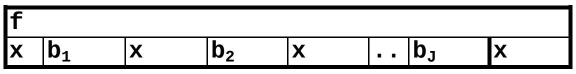


Figure 4 - Mixed-Mode storage: file structure for flags

with:

- f is the external binary file containing flags of one or more local columns.
- The blocks named as ‘x’ are optional; if they exist, they contain information that does not contribute to the flags (don’t care).
- b_1, b_2, .., b_J are the blocks within the binary file containing flags information.
- A block b_j contains all flags of one local column segment.
- The number of flags must be identical to the number of values of the corresponding local column segment.

Each flag is stored as a 16-bit number (i.e. DT_SHORT). Thus the number of bytes of a block is twice the number of values of the corresponding local column. If the attribute derived from the base attribute 'value_type' of that external component indicates big endian order (by the extension _beo), the flags are also stored in big endian order so that when reading/writing bytes from/to the file i/o stream the most significant byte comes first. Otherwise the least significant byte comes first (little endian order).
Each flag is a combination of status information for a data value; each status information is coded into one bit of the 16-bit flag. The meaning of the individual bits is given in the following table. Typically all bits are set (thus providing a value of 0x000F). The meaning of these flags is further described in the base model at AoLocalColumn.

Table 35 - Bit coding of value flags

Abbreviation	Value	Description
AO_VF_VALID	0x0001	Value is valid.
AO_VF_VISIBLE	0x0002	The value has to be visualized.
AO_VF_UNMODIFIED	0x0004	Value is not modified.
AO_VF_DEFINED	0x0008	Value is defined. If the value in a value matrix is not available this bit is not set.

9.4.6 EXAMPLES

This section provides some examples for Mixed-Mode storage of numerical values.

EXAMPLE 1: ONE CHANNEL PLUS TIME INFORMATION

Given one channel that provides ten values of one measurement quantity MQ at a specific sample rate. The relative time (with respect to the measurement begin) is also provided as a separate measurement quantity Time. MQ data are stored in an external file as four-byte float values (ieeefloat4); the time information is also stored in that file however as 2-byte integer values (dt_short; e.g. multiples of seconds), preceding the MQ values. Both data sets are stored as a block each. The first 6 bytes are file header information (h).

Logical representation:

Time	MQ
t0	v0
t1	v1
...	...
t9	v9

Physical representation on file:

h	t0	t1	...	t9	v0	v1	...	v9

Attributes of external component::

related to	Time	MQ
component_length	10	10
value_type	dt_short	ieeefloat4
start_offset	6	26
block_size	20	40
valuesperblock	10	10
value_offset	0	0

Example 2: Three Channels Interleaved

Given three channels that provide three values each for three different measurement quantities at the same sample rate. The values are stored in an external file as four-byte integer values (dt_long). The time is not stored in the external file in this example.

Logical representation:

Time	**MQ1**	**MQ2**	**MQ3**
`t1`	`m1v1`	`m2v1`	`m3v1`
`t2`	`m1v2`	`m2v2`	`m3v2`
`t3`	`m1v3`	`m2v3`	`m3v3`

Physical representation on file:

`m1v1`	`m2v1`	`m3v1`	`m1v2`	`m2v2`	`m3v2`	`m1v3`	`m2v3`	`m3v3`

Attributes of external component::

related to	**MQ1**	**MQ2**	**MQ3**
`component_length`	`3`	`3`	`3`
`value_type`	`dt_long`	`dt_long`	`dt_long`
`start_offset`	`0`	`0`	`0`
`block_size`	`12`	`12`	`12`
`valuesperblock`	`1`	`1`	`1`
`value_offset`	`0`	`4`	`8`

EXAMPLE 3: THREE CHANNELS PARTLY INTERLEAVED

Given three channels that provide six values each for three different measurement quantities at the same sample rate. The values are stored in an external file as eight-byte float values (ieeefloat8) such that always two consecutive values of one channel are placed in a sub-block next to each other followed by two consecutive values of another channel. The time is not stored in the external file in this example.

Logical representation:

Time	MQ1	MQ2	MQ3
t1	m1v1	m2v1	m3v1
t2	m1v2	m2v2	m3v2
t3	m1v3	m2v3	m3v3
t4	m1v4	m2v4	m3v4
t5	m1v5	m2v5	m3v5
t6	m1v6	m2v6	m3v6

Physical representation on file:

m1v1	m1v2	m2v1	m2v2	m3v1	m3v2	m1v3	m1v4	m2v3	>>
>>	m2v4	m3v3	m3v4	m1v5	m1v6	m2v5	m2v6	m3v5	m3v6

Attributes of external component::

related to	MQ1	MQ2	MQ3
component_length	6	6	6
value_type	ieeefloat8	ieeefloat8	ieeefloat8
start_offset	0	0	0
block_size	48	48	48
valuesperblock	2	2	2
value_offset	0	16	32

The next example may not be implemented in a Mixed-Mode storage, though the samples of each channel have been taken at equidistant time intervals:

EXAMPLE 4: HOMOGENEOUS MEASUREMENT; NOT SUPPORTED

Given three channels measured with different sampling rates:

- m1: double precision floating-point numbers with 10 Hertz sampling
- m2: short integer numbers with 20 Hertz sampling
- m3: long integer numbers with 30 Hertz sampling

Logical representation:

Time	MQ1	MQ2	MQ3
t1	m1v1	m2v1	m3v1
t2			
t3			m3v2
t4		m2v2	
t5			m3v3
t6			
t7	m1v2	m2v3	m3v4
t8			
t9			m3v5
t10		m2v4	
t11			m3v6
t12			
t13	m1v3	m2v5	m3v7
...			

If the values are written to the file as they come from the measuring instrument, one would find a file structure as follows:

m1v1	m2v1	m3v1	m3v2	m2v2	m3v3	m1v2	m2v3	m3v4	>>
>>	m3v5	m2v4	m3v6	m1v3	m2v5	m3v7	...	...	...

Such a set of data cannot be kept in one external binary file under the Mixed-Mode storage concept.

Note however, that the component files of ATF may hold such file structures, as is given in example 5 in section 8.7.22.

9.4.7 Relationship to ATF Components

The Mixed-Mode storage concept is closely linked to the ATF component element (see chapters 7 and 8). As in ATF, the binary file will hold the mass data, while the descriptive data are stored in the RDB (resp. in the ATF file).

However there are some differences:

- The Mixed-Mode storage specification is less flexible than the ATF specification with regard to the positioning of values of the same local column within one block. Example 4 of section 9.4.6 may not be used for a Mixed-Mode storage though it may describe a valid ATF component file. This is due to the fact that AoExternalComponent may only specify one value for the attribute derived from the base attribute 'value_offset' while ATF may have a set of those offsets.
- The Mixed-Mode storage specification is more flexible than the ATF specification in that it allows to segment the values and distribute them over several physical files. Each of them is then specified as one external component, and a local column may refer to more than one, distinguished by the value of the attribute derived from the base attribute 'ordinal_number'. The ATF specification requires that all values of a local column are placed within the same component file.
- According to ATF/CLA, flags will not be stored in binary component files; the Mixed-Mode storage requires to store flags in binary files in case the values are stored in binary files; ATF/XML allows to store flags either in the ATF/XML file or in an external component file.

9.4.7.1 Mapping of AoExternalComponent Attributes to ATF/CLA Elements

The following table shows a comparison between ATF/XML and ATF/CLA elements and the Mixed-Mode storage specification.

Table 36 - Mapping of AoExternalComponent attributes to ATF/CLA elements

ATF/XML element in <component> element	ATF/CLA element in COMPONENT structure	Attribute of AoExternalComponent
<identifier>	identifier	filename_url
<datatype>	type specification [1)]	value_type
<length>	length specification	component_length
<description>	DESCRIPTION	Description
<inioffset>	INIOFFSET	start_offset
<blocksize>	BLOCKSIZE	block_size
<valoffsets>	VALOFFSETS	value_offset (note: only one allowed)
<valperblock>	VALPERBLOCK	Valuesperblock
N/A (only one binary file allowed per local column)	N/A (only one binary file allowed per local column)	ordinal_number

[1)] Note that not all values of 'typespec_enum' are allowed in ATF/CLA and that "dt_bytestr" may show a different endian order in the length field, as ATF/CLA assumes that the length field is always big endian order. The length value may give a hint on whether big or little endian order is used.

9.4.8 THE MIXED-MODE SERVER

The following figure shows the model of a Mixed-Mode Server:

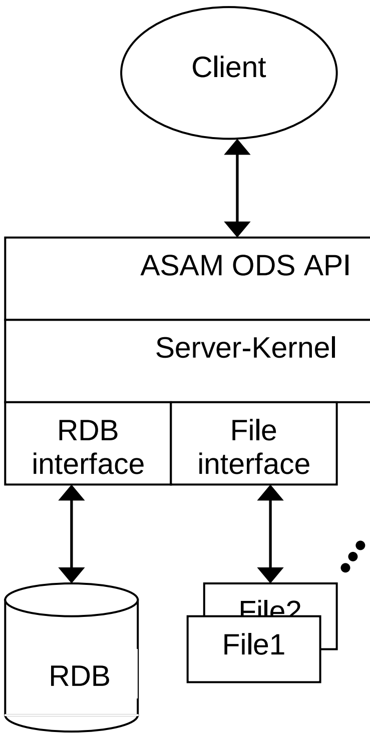


Figure 5 - Mixed-Mode server architecture

A Mixed-Mode Server must fulfill the following requirements:

- Access to the database (RDB) for all ASAM ODS elements
- Storage of mass data within the database or in separate files outside the database; the behavior may be configured at the server. For this purpose a session-specific context variable may be set by a client via the APIs, specifying whether newly written mass data shall remain in the database or be placed in external files.
- Transparency to API, so that a client does not need to care how mass data are stored, when retrieving them.
- Asynchronous access: the server must synchronize the asynchronous access to the files. Access of different clients using the API must be coordinated by the server. While access to the same information by different clients is normally synchronized by the database management system, it is the responsibility of the ASAM ODS server in mixed mode operation. Access to the files not using the API cannot be synchronized by the ASAM ODS server.

9.4.8.1 Considerations for Mixed-Mode Server Implementations

- The server must ensure the consistency of the data after any write operation on the external-component file. E.g. if the client sends a delete request on some values of one local column and not on the remaining local columns inside one partial matrix, the server must detect this inconsistency and must be able to do a "rollback" (keep a file copy or detect it before writing, etc.).
- The server must lock the file during modification (Shared lock – no write access on the involved files, only read access). The server must hold a private working copy of the file and place an exclusive lock on the involved files during the "commit"-phase.
- The server must support some error recovery on start-up after a system crash.

9.4.8.2 Parameter for Defining the File Size

The context variable EXT_COMP_SEGSIZE defines the maximum file size of server-created external component files in bytes. When the client changes the value of this parameter, all following write operations must use the new definition. The default value of this parameter should be set at server-start-up-time to an appropriate value by using a server-side configuration mechanism (e.g. start-up-file, or start-up-parameters).

9.4.8.3 Mode-Switching of Mixed-Mode Server

To enforce the writing of an external file or the storage within the database of the server, the session-specific context variable ***write_mode*** with the possible values: ***database (default) / file*** is used.

Index

None.

Figure Directory

Table Directory