



AnyDAC

Architecture Guide

Version 1.4.0

1	Introduction.....	4
2	How is AnyDAC structured?	5
2.1	Layers and Packages	5
2.2	Naming Conventions	5
2.3	Stan Layer	6
2.4	Phys layer.....	7
2.5	DApt Layer	8
2.6	DatS Layer	8
2.7	GUIx Layer	9
2.8	Comp Layer	10
3	Stan Layer.....	11
3.1	Interface Factory.....	12
3.2	Resource pooling	13
3.3	Error handling.....	15
3.4	Expression evaluation.....	17
3.5	Persistent definition list.....	19
3.6	Persistent connection definition list.....	22
3.7	Async operation execution	24
3.8	Parameters and macros	25
3.9	Options	26
3.10	Debug monitoring.....	28
4	DatS Layer	32
4.1	DatS Manager	34
4.2	DatS Table.....	34
4.2.1	DatS Column	34
4.2.2	DatS Row	36
4.2.3	DatS constraint	39
4.2.4	DatS View.....	40
4.2.4.1	Mechanism	42
4.2.4.2	Aggregate.....	44

5	Phys Layer.....	45
5.1	Phys Driver	46
5.2	Phys Layer manager.....	48
5.3	Driver definition file	49
5.4	Phys connection	51
5.5	Phys command.....	53
5.5.1	Macro processing	56
5.5.1.1	Substitution variables	56
5.5.1.2	Escape sequences	57
5.5.2	Special character processing	58
5.5.3	Asynchronous execution	59
5.5.4	Batch command execution	60
5.5.5	Stored procedure execution	61
6	DApt Layer.....	63
6.1	Structure Mapping	64
6.2	Posting updates to DB	66
6.2.1	DatS Table Adapter	66
6.2.2	DatS Manager Adapter	67
6.2.3	Concurrency Control	68
6.2.4	Row Refreshing	69
6.2.5	Commands Handling	71
6.2.6	Error Handling.....	71
	Appendix 1. TADDataType.....	72
	Appendix 2. Macro data types	74
	Appendix 3. Top level interfaces and GUID's	75
	Appendix 4. Connection definition parameters	76
	Appendix 5. Macro functions.....	78

1 Introduction

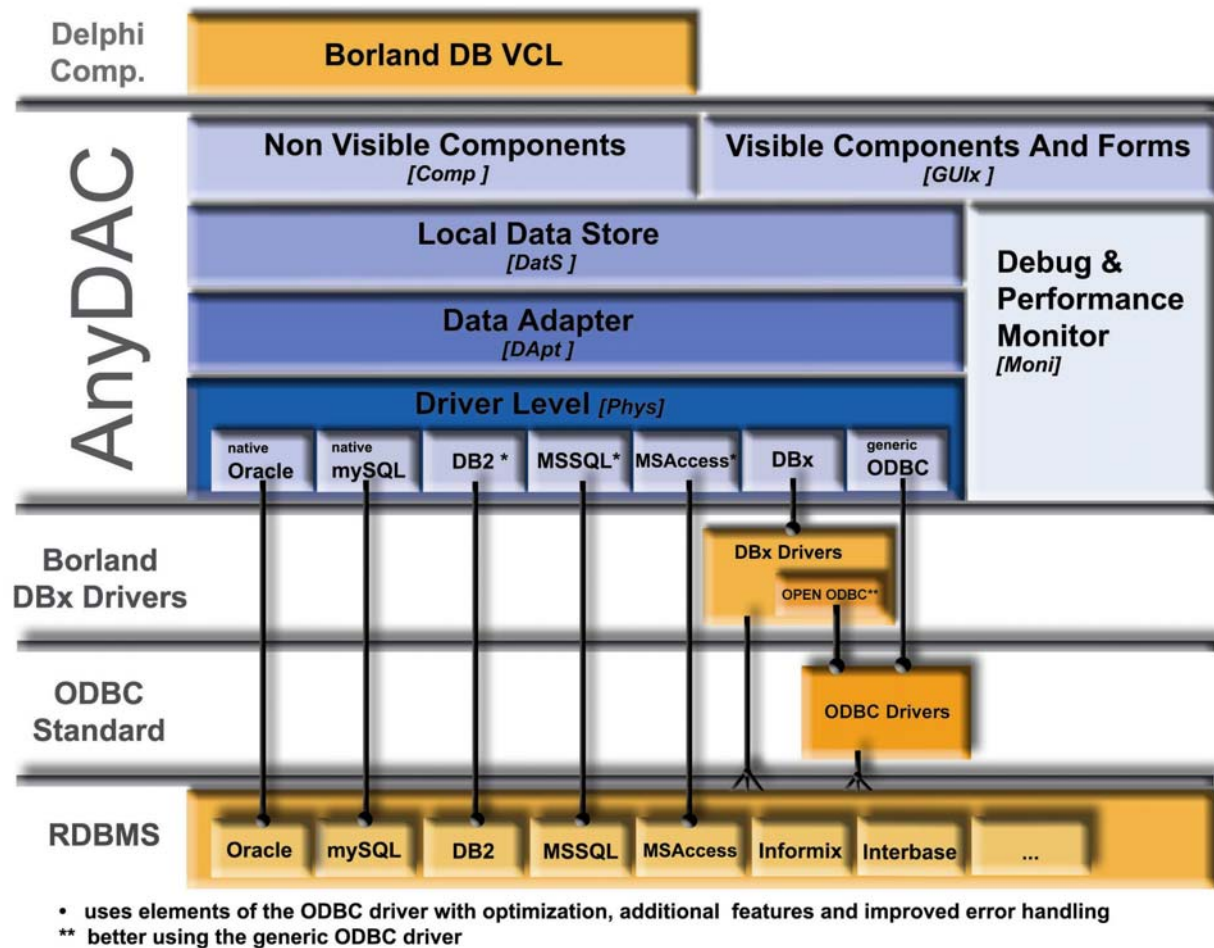


Figure 1 AnyDAC Overview

AnyDAC Architecture provides information about layers and packages composing AnyDAC and their interaction. This document is a road map for AnyDAC users and allows orienting in AnyDAC software. The guide also explains design decisions made by us to achieve AnyDAC. To meet the quality requirements to use AnyDAC in mission critical database application the emphasis was put on four main factors:

- Stability
- Performance
- Flexibility
- Extendibility

In this document you will not find guidelines on how to use the components of AnyDAC. For Information about how to write efficient AnyDAC applications please read the **AnyDAC Programmer's Guide**.

2 How is AnyDAC structured?

2.1 Layers and Packages

The AnyDAC framework was designed using a layered approach. Each layer defines a set of interfaces through which a clean communication between components of *different layers* is possible. All interfaces of each layer belong to *the Coml package*. Packages were used as a way to structure AnyDAC Delphi code. An AnyDAC layer can consist of 1 or more packages, whereas a package belongs to maximum one layer.

The interfaces from the Coml package build a communication bus. This allows designing an AnyDAC application in independent sub-systems. The different sub-systems can then interact as a unit through the common bus.

Layers

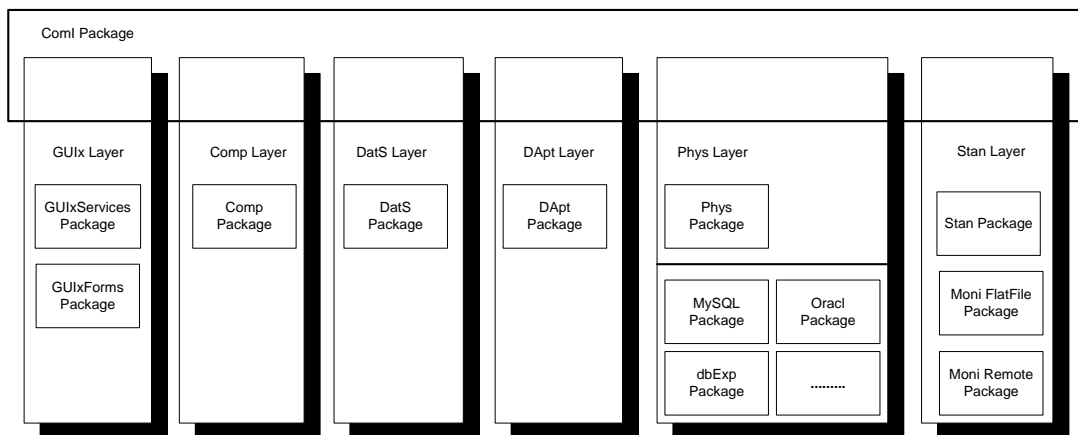


Figure 2 AnyDAC Layers Overview

For each AnyDAC layer there exists a package with the same name. The package contains common layer code. An AnyDAC layer can contain additional so called “alternative” packages. This is to extend the AnyDAC framework with other “alternative” implementations of its layer interfaces. The alternative packages are optional, because are one of possible implementations. For example, the **Phys** layer has two alternative packages (there are much more drivers! <g>), Oracle and MySQL. Each of them is an alternative implementation of the **Phys** layer interfaces. The Oracle package implements data access to the ORACLE RDBMS and the MySQL package to the MySQL database.

2.2 Naming Conventions

Short Name	Long Name
Stan Layer	Standard classes, interfaces, routines, constants, etc

Phys Layer	Physical Access Layer
DApt Layer	Data Adapter
DatS Layer	Data Store (Local)
Comp Layer	non visible Components
GUIx Layer	visible Components and Forms = GUI
ComI	Common Interfaces

The name of an AnyDAC layer or package is 4 characters long.

The unit names in the AnyDAC framework are built of a prefix + layer name + unit role + unit sub role.

So, the unit names as **daAD<layer name><unit role>[<unit sub role>]** will best reflect their containment. For example, a unit defining a layer interfaces would be called **daADPhysIntf**.

2.3 Stan Layer

Layer Name	Stan
Packages	Stan, MoniIndy, MoniFlatFile, MoniCustom
Uses Packages	ComI
Interface Unit	daADStanConst daADStanError daADStanFactory daADStanIntf daADStanOption daADStanParam daADStanResStrs daADStanTracer daADStanUtil

The **Stan** layer is a set of common classes, routines and constants. The layer implements the following main facilities:

- Persistent definition list
- Persistent connection definition list
- Error handling

- Expression parsing and execution
- Resource pooling
- Asynchronous operation execution
- Macros and parameters
- AnyDAC options handling
- Software-debugging capabilities

The **MoniXXX** packages provide AnyDAC software-debugging capabilities. More precisely, they implements debug monitor interfaces, which allow monitoring the interaction between AnyDAC application and the DBMS. In a specific application environment these interfaces could be implemented using some sort of message transport. Today AnyDAC has 3 implementations – one for interactive monitoring, one for generating a trace file and another for custom tracing.

Interactive monitoring implementation is based on the Indy TCP/IP components. And is in **MoniIndy** package. This package assumes the existence of some external monitoring application to which the AnyDAC monitor client will communicate. `$(AnyDAC)\Tool\Monitor\ADMonitor.dpr` is standard one, supplied with AnyDAC.

Trace file implementation is in **MoniFlatFile** package. And custom implementation is in **MoniCustom**.

2.4 Phys layer

Layer Name:	Phys
Packages:	Phys DbExp, ODBC, Oracl, MySQL, MSAcc, MSSQL, DB2, ASA
Uses Packages:	ComI, Stan, DatS
Interface Units:	daADPhysIntf

The **Phys** layer defines interfaces for physical data access. It implements them in a separate packages as drivers, whereas each driver package belongs to the **Phys** layer and implements the required interfaces using appropriate DBMS API. The Phys Manager singleton object, which is the entry point, controls the layer. The main **Phys** layer interfaces are:

- **IADPhysDriver** Represents the AnyDAC driver API.
- **IADPhysConnection** Controls the physical database connection and transactions.
- **IADPhysCommand** Executes RDBMS commands and fetches the result data.

- **IADPhysMetaInfoCommand** Retrieves RDBMS object's Meta information.

2.5 DApt Layer

Layer name	DApt
Packages	DApt
Uses packages	Coml, Stan, DatS
Interface unit	daADDAptIntf daADDAptColumn

The **DApt** layer allows automation and fine-tuning of *read operation* with complex result sets (master-details, nested, ADT, etc) and allows *posting back updates* to the database system. The **DApt** layer is the **Phys** layer's superstructure and cannot be used on its own. However, it is convenient to use the **Phys** layer without the **DApt** layer. The main **DApt** layer classes and interfaces are:

- **TADDAptColumnMapping**. Maps individual column of a result set to a column of the local data storage.
- **TADDAptColumnMappings**. Maps SELECT list of a result set to a column list of the local data storage.
- **IADDAptTableAdapter**. Specifies, how local data storage table should be filled from IADPhysCommand and how changes are posted back.
- **IADDAptTableAdapters**. It is a list of table adapters.
- **IADDAptSchemaAdapter**. Specifies, how the local data storage should be filled and, how changes should be posted back.

2.6 DatS Layer

Layer name	DatS
Packages	DatS
Uses packages	Stan, Coml
Interface unit	daADDatSManager

DatS layer does not have a dedicated interface unit like other AnyDAC layers. That is due to the current implementation and is subject to change in future.

DatS layer is a Local Data Storage implementation, which is analogue to the ADO.Net's DataSet and its related objects (DataTable, DataRow, DataView). See Ado.Net documentation for details. It is an in-memory data engine, which may be filled with data and structures the data in different ways:

- Directly from AnyDAC application code
- Using the methods of the AnyDAC **Phys** layer: IADPhysCommand.Define / IADPhysCommand.Fetch
- Using the interface of the AnyDAC **Dapt** layer

2.7 GUIx Layer

Layer name	GUIx
Packages	GUIx GUIxForms GUIxConsole
Uses packages	Stan Coml Comp
Interface unit	daADGUIxIntf

The **GUIx** layer provides a way to interact with the user from an AnyDAC application. Depending on the environment or the domain this can be specific for the application. Hence, **GUIx** layer interfaces are implemented by one of the alternative packages:

- **GUIxForms**. The package implements Delphi TForm based dialogs, which are interacting with end user through Windows GUI.
- **GUIxConsole**. The package is useful for Win32 Service or similar application. This package does not use any GUI API, but provides a silent application mode without desktop interaction.

The main interfaces of the **GUIx** layer are:

- **IADGUIxLoginDialog**. This Dialog asks the user for his login credential. It may be connected to the **Phys** layer via **IADPhysConnection.LoginDialog** or to the **Comp** layer via **TADConnection.LoginDialog**.
- **IADGUIxErrorDialog**. AnyDAC exception handling can be hooked to this Dialog. It shows errors messages with extended information for AnyDAC exceptions.
- **IADGUIxAsyncExecuteDialog**. This Dialog shows progress of asynchronous operation execution and also allows the user to cancel it.

- **IADGUILxWaitCursor**. This component is changing the mouse cursor into an hourglass cursor. It should be used at the beginning of long running operations.
- **TADGUILxFormsQBldrDialog**. SQL Queries can be built with this graphical tool.

2.8 Comp Layer

Layer name	Comp
Packages	Comp
Uses packages	Stan ComI DatS Phys DApt
Interface unit	-

The **Comp** layer represents the AnyDAC public interfaces as Delphi non-visual components, similar to other Delphi data access components.

3 Stan Layer

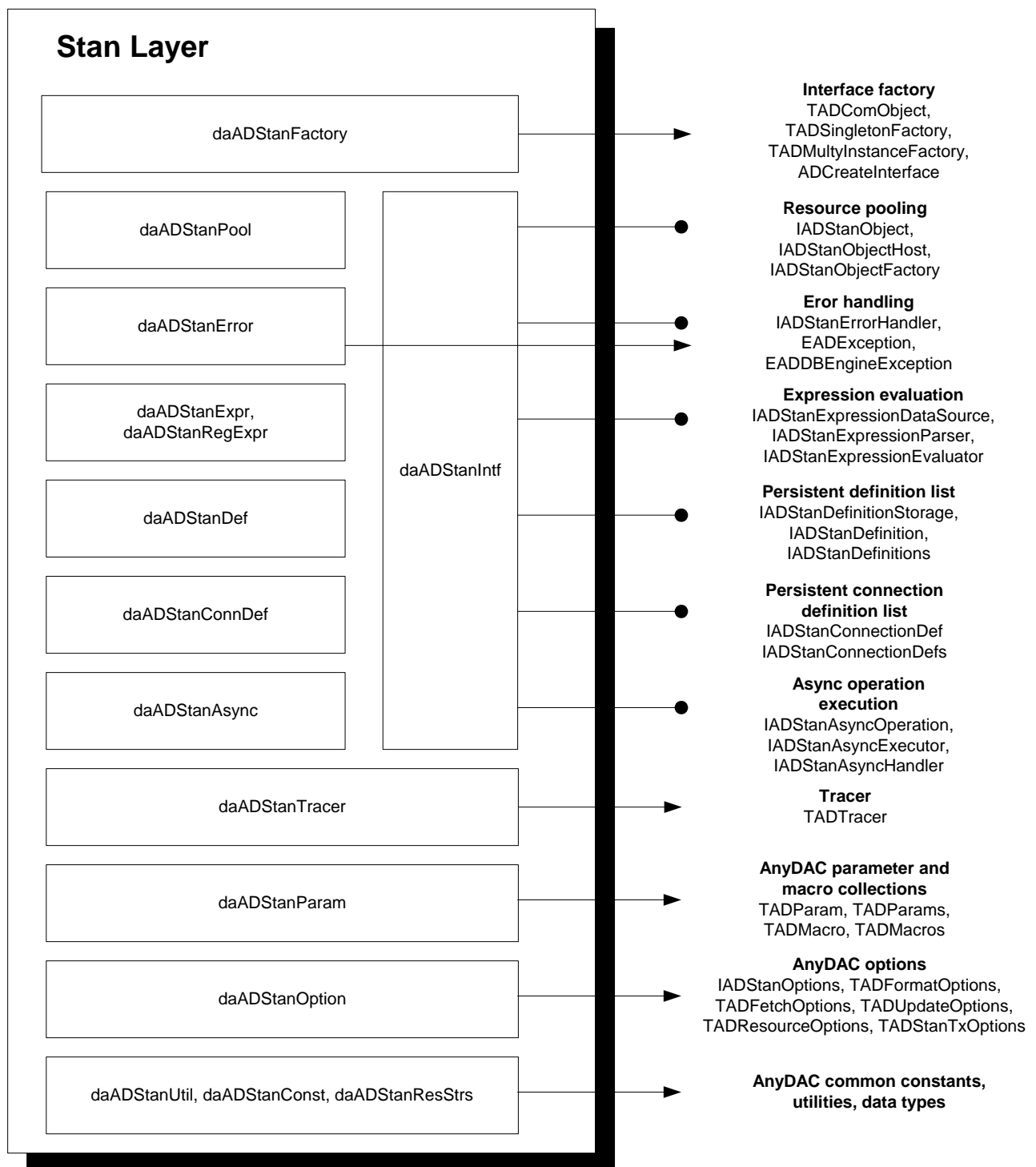


Figure 3 Stan Layer Overview

The **Stan** layer is a set of common algorithms, routines, data types and constants. Algorithms are exposed as COM interfaces as well as Delphi classes and routines. Classes were used instead of interfaces to achieve compatibility with legacy component libraries. All layer COM interfaces are declared in the **daADStanIntf** unit.

3.1 Interface Factory

Interface unit	daADStanFactory
Standard implementation unit	daADStanFactory

Public AnyDAC class instances may be created directly. And, to create interface instances, AnyDAC uses factory-based approach.

Some AnyDAC interfaces may be instantiated directly (*top-level*), but some just through another interfaces. Object class implementing top-level interfaces, must be inherited from **TADComObject** base class, which itself is inherited from **TComObject** Delphi class. For example, definition class:

```
TADDefinitions = class (TADComObject, IADStanDefinitions)
.....
end;
```

Then interface-implementing class must be registered with one of standard AnyDAC factories. All factory classes are inherited from **TComObjectFactory** Delphi class. There are two factories:

- **TADMultyInstanceFactory**. This factory will create new implementing object instance each time, when user asks for interface instance. The object will count references to it interfaces. And after count will be equal to zero object instance will be destroyed.
- **TADSingletonFactory**. This factory will create single implementing object instance at first request for interface instance. All subsequent requests will share the same object instance. The object will be destroyed at AnyDAC application shutdown.

The units containing classes, implementing AnyDAC top-level interfaces, will have, for example, following factory registration code:

```
initialization
  TADMultyInstanceFactory.Create(TADFileDefinitionStorage, IADStanDefinitionStorage);
  TADMultyInstanceFactory.Create(TADDefinitionStandalone, IADStanDefinition);
  TADMultyInstanceFactory.Create(TADDefinitions, IADStanDefinitions);
  TADMultyInstanceFactory.Create(TADConnectionDefStandalone, IADStanConnectionDef);
  TADMultyInstanceFactory.Create(TADConnectionDefs, IADStanConnectionDefs);
end.
```

And to have top-level interface implementation in your application, you should include specific implementation unit into your application uses clause. Without that, you can get error, saying “Object factory for class <interface GUID> is missing”. In this case refer to the Appendix 3. Top level interfaces and GUID’s. To create top-level interface instance you should use **ADCreateInterface** function. For example, following code creates an instance of **IADStanConnectionDefs** and returns it:

```
function TADPhysManager.GetConnectionDefs: IADStanConnectionDefs;
begin
    if FConnectionDefs = nil then
        ADCreateInterface(IADStanConnectionDefs, FConnectionDefs);
    Result := FConnectionDefs;
end;
```

3.2 Resource pooling

Interface unit	daADStanIntf
Standard implementation unit	daADStanPool

AnyDAC uses resource pooling to share limited resource instances across clients. AnyDAC pool collects and uses statistic about resource usage. This allows creating additional resource instances, if there are too few. And backward, if there will be too many not used resource instances, AnyDAC will destroy extra instances. This is useful for connection pooling, but may be used for other tasks, including own application needs.

In AnyDAC resource-pooling behavior is defined by these interfaces:

```
{ ----- }
{ Resource pool interfaces }
{ ----- }

IADStanObject = interface (IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2001}']
    procedure BeforeReuse;
    procedure AfterReuse;
    .....
end;

IADStanObjectHost = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2002}']
    procedure CreateObject(out AObject: IADStanObject);
    .....
end;

IADStanObjectFactory = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2003}']
    procedure Open(const AHost: IADStanObjectHost; const ADef: IADStanDefinition);
```

```
procedure Close;
procedure Acquire(out AObject: IADStanObject);
procedure Release(const AObject: IADStanObject);
end;
```

IADStanObject is an interface, which has to implement resource object. Pooler will call **BeforeReuse** method just before a resource instance will be returned to application. And **AfterReuse** right after the application has returned resource instance backward to pool. The pool implementation keeps single reference to each resource instance. So, object implementing **IADStanObject**, may automatically return itself to pool:

```
// The code is simplified for goal of this manual
function TADPhysConnection._Release: Integer;
var
  oPoolItem: IADStanObject;
begin
  Result := InterlockedDecrement(FRefCount);
  // If connection is from pool, then check for last reference
  // which is kept by pool itself. And if so, then return object
  // to pool.
  if (Result = 1) and (FPool <> nil) then begin
    Supports(Self, IADStanObject, oPoolItem);
    FPool.Release(oPoolItem);
    InterlockedDecrement(FRefCount);
    Pointer(oPoolItem) := nil;
  end
  else if Result = 0 then
    Destroy;
end;
```

IADStanObjectHost is an interface, which is responsible for creating resource instances. It self is not a resource pool, but a resource instance factory.

IADStanObjectFactory is an top-level interface, which provides the required methods for managing a resource pool. The method **Open** activates the resource pool, therefore it should be called first. Then, it is possible to check resource items out and in. The second parameter is a definition, where pool will look for parameters. See Appendix 4. Connection definition parameters for parameters list.

The **Acquire** method is called to check out the instance, for example:

```
procedure TADPhysConnectionHost.CreateConnection(out AConn: IADPhysConnection);
var
  oObj: IADStanObject;
begin
  if FConnectionDef.Pooled then begin
    FPool.Acquire(oObj);
    Supports(oObj, IADPhysConnection, AConn);
```

```
end
else
    InternalCreateConnection(AConn);
end;
```

Calling **Release** checks in the instance. In the example above, instance returns himself to the pool automatically. That means, an **IADPhysConnection** instance will return itself to pool automatically, so programmer should not call pooler **Release** method. Actually, in this case pooler is not accessible to programmer.

3.3 Error handling

Interface unit	daADStanIntf, daADStanError, daADStanConst, daADStanResStrs
Standard implementation unit	daADStanError

A correct error handling is very valuable for stable and flexible database application. AnyDAC has its own error-handling infrastructure. It consists of:

- Exception base classes (**daADStanError** unit);
- Exception error codes (**daADStanConst** unit);
- Exception error messages (**daADStanResStrs** unit);
- Exception handling interface (**daADStanIntf**).

All AnyDAC exception classes are derived from the base class **EADException**:

```
EADException = class(EDatabaseError)
public
    constructor Create(AADCode: Integer; const AMessage: String); overload;
    procedure Duplicate(var AValue: EADException); virtual;
    property ADCode: Integer read FADCode;
end;
```

Here, **ADCode** contains an AnyDAC specific error code. All error codes are represented by the **er_AD_XXX** constants in the unit **daADStanConst**.

All RDBMS initiated exceptions are of class **EADDBEngineException**, which is inherited from **EADException**. This class provides detailed information returned by database system. In general, RDBMS errors are a sequence of sub errors, or few pairs of *reason-consequence items*. However, we

more often see that they only return a single error item. The sequence of errors is exposed as a list of **TADDBError** instances, each of them can describe one error item. The following declaration shows details:

```
TADDBError = class (TObject)
public
  constructor Create(ALevel, AErrorCode: Integer; const AMessage,
    AObjName: String; AKind: TADCommandExceptionKind; ACmdOffset: Integer); overload;
  property ErrorCode: Integer read FErrorCode write FErrorCode;
  property Kind: TADCommandExceptionKind read FKind write FKind;
  property Level: Integer read FLevel write FLevel;
  property Message: String read FMessage write FMessage;
  property ObjName: String read FObjName write FObjName;
  property CommandTextOffset: Integer read FCommandTextOffset write FCommandTextOffset;
  property RowIndex: Integer read FRowIndex write FRowIndex;
end;

EADDBEngineException = class(EADException)
public
  constructor Create(AADCode: Integer; const AMessage: String); overload;
  destructor Destroy; override;
  procedure Dublicate(var AValue: EADException); override;
  procedure Append(AItem: TADDBError);
  procedure Prepend(AItem: TADDBError);
  property ErrorCount: Integer read GetErrorCount;
  property Errors[Index: Integer]: TADDBError read GetErrors; default;
  property Kind: TADCommandExceptionKind read GetKind;
  property MonitorAdapterIntf: IADMoniDebugAdapter read FMonitorAdapterIntf;
end;
```

Here **EADDBEngineError.Errors** is a collection of error items. The property **Kind** is the RDBMS independent error code and **ErrorCode** is the RDBMS native error code. The property **ObjName** identifies erroneous object, that may be violated constraint name or name of object failed to create or so on. For SQL syntax errors, property **CommandTextOffset** will contain SQL command text offset. And the **RowIndex** property is an index in parameter array for batch command execution.

Not all database systems offer the same range of information about errors occurred. Therefore, some of the properties of **EADDBEngineException** or **TADDBError** may have no values assigned.

For example, following code shows handling of *the primary key violation*:

```
try
  MyQuery.ExecSQL;
except
  on E: E EADDBEngineException do
    If E.Kind = ekUKViolated then
      ShowMessage('Unique key ' + E.Errors[0].ObjName + ' is violated');
end;
```


For objects to handle their own exceptions in consistent way, AnyDAC defines the **IADStanErrorHandler** interface. Many AnyDAC objects such as the connection or command classes implement this interface. The following declaration shows more details:

```
IADStanErrorHandler = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2007}']
  procedure HandleException(const AInitiator: IADStanObject; var AException: Exception);
end;
```

The method **HandleException** of implementing this interface object will be called for every AnyDAC exception raised by methods of this object. The parameter **AException** will receive exception object. Actually, not all object methods, but most valuable, was programmed to call this error handler. For example, method **IADPhysCommand.Execute** has that, but **IADPhysCommand.GetParams** has not this functionality. **IDAPhysCommand** has property **ErrorHandler**, which allows to set custom error handler.

3.4 Expression evaluation

Interface unit	daADStanIntf
Standard implementation unit	daADStanExpr daADStanRegExpr

Many features in AnyDAC are implemented using *expression parser* and *evaluator*. Examples are filtering capabilities, checking constraints, aggregate functionality, etc. The expression evaluator is based on the Oracle expression syntax and additionally supports:

- MIDAS set of expression functions;
- ODBC set of escape functions;
- A very limited kind of natural expressions (experimental);
- RegExp regular expressions.

The following declaration shows details:

```
IADStanExpressionDataSource = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2008}']
  // public
  property VarIndex[const AName: String]: Integer read GetVarIndex;
  property VarType[AIndex: Integer]: TADDataType read GetVarType;
  property VarData[AIndex: Integer]: Variant read GetVarData write SetVarData;
  property Position: Pointer read GetPosition write SetPosition;
```

```
property RowNum: Integer read GetRowNum;
end;

IADStanExpressionParser = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2009}']
  // public
  function Prepare(const ADataSource: IADStanExpressionDS;
    const AExpression: String; AOptions: TADExpressionOptions;
    AParserOptions: TADParserOptions; const AFixedVarName: String): IADStanExpression;
  property DataSource: IADStanExpressionDS read GetDataSource;
end;

IADStanExpressionEvaluator = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2010}']
  function Evaluate: Variant;
  property DataSource: IADStanExpressionDS read GetDataSource;
end;
```

The high-level interface - **IADStanExpressionParser** - allows parsing expressions. If the process completes successfully, it returns an instance of **IADStanExpressionEvaluator**. It represents the input parsed and optimized for expression evaluation.

Following example shows how to parse and evaluate expression:

```
var
  oParser: IADStanExpressionParser;
  oEval: IADStanExpressionEvaluator;
begin
  ADCreateInterface(IADStanExpressionParser, oParser);
  oEval := oParser.Prepare(nil, 'to_char(10+20)', [ekNoCase], [poDefaultExpr], '');
  oEval.Evaluate;
end;
```

The expression engine is independent of a data source and may be integrated with anyone. The interface **IADStanExpressionDataSource** represents an abstract data source. From the expression engine point of view, data source is a matrix. Each column in the matrix is a named variable, which has an index, data type and value (properties: **VarIndex**, **VarType**, **VarData**). One of rows in the matrix may be current and is represented by position (**Position** property). For example, Delphi's **TDataSet** may be modeled that **TFields** are matrix columns and **TDataSet** rows are rows of the matrix.

For example, method **TADDataSet.CreateExpression** creates evaluator for custom expression and setups it to use field values from current row:

```
var
  oEval: IADStanExpressionEvaluator;
begin
  oEval := MyQuery1.CreateExpression('Amount * Price + Taxes');
```

```
while not MyQuery1.Eof do begin
    // move evaluator to current record
    oEval.DataSource.Position := MyQuery1.GetRow;
    // evaluate expression for current record
    oEval.Evaluate;
    MyQuery1.Next;
end;
```

3.5 Persistent definition list

Interface unit	daADStanIntf
Standard implementation unit	daADStanDef

In AnyDAC, a *definition* is a named set of parameters. Each parameter is a name-value pair. The value type may be any one of following:

- String
- Integer
- Boolean (is encoded by True/False values)
- YesNo (similar to Boolean, but is encoded by Yes/No values)

A parent-child relation may relate any two definitions. In this case parameters of a *child definition* will override cognominal parameters of *parent definition*. For example:

Parent definition parameters	Child definition parameters	Resulting parameters set
P1=QWE	-	P1=QWE
P2=123	P2=456	P2=456
-	P3=True	P3=True
P4=ASD	P4=<empty>	P4=<empty>

Definitions are collected together into a list, which may be stored to external storage. There may be definitions of the following styles:

- Internal. It is not a member of the definition list and cannot be stored in external storage. Most time is used as an unnamed temporary parameter set. To create internal definition use method **IADStanDefinitions.AddInternal**.

- Private. It is a member of definition list and may be founded in the list by name. But it cannot be stored in external storage. Most time is used as a named temporary parameter set. It is default style.
- Persistent. It is like a private, but is stored in external storage. By default, AnyDAC creates private definitions. To make it persistent, mark it so calling the method **MarkPersistent**.

Private and Persistent definition names have to be unique (case insensitive) in the list. The following declaration shows more details:

```
IADStanDefinitionStorage = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2012}']
  // public
  function CreateIniFile: TCustomIniFile;
  // R/O
  function ActualFileName: String;
  // R/W
  property FileName: String read GetFileName write SetFileName;
  property GlobalFileName: String read GetGlobalFileName write SetGlobalFileName;
  property DefaultFileName: String read GetDefaultFileName write SetDefaultFileName;
end;

IADStanDefinition = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2013}']
  // public
  procedure Apply;
  procedure Clear;
  procedure Cancel;
  procedure Delete;
  procedure MarkPersistent;
  procedure OverrideBy(const ADefinition: IADStanDefinition; AAll: Boolean);
  function ParseString(AStr: String; AKeywords: TStrings; AFmt: TADParseFmtSettings): String;
overload;
  function BuildString(AKeywords: TStrings; AFmt: TADParseFmtSettings): String; overload;
  function HasValue(const AName: String): Boolean; overload;
  function OwnValue(const AName: String): Boolean;
  property State: TADDefinitionState read GetState;
  property Style: TADDefinitionStyle read GetStyle;
  property AsString[const AName: String]: String read GetAsString write SetAsString;
  property AsBoolean[const AName: String]: LongBool read GetAsBoolean write SetAsBoolean;
  property AsYesNo[const AName: String]: LongBool read GetAsBoolean write SetAsYesNo;
  property AsInteger[const AName: String]: LongInt read GetAsInteger write SetAsInteger;
  property ParentDefinition: IADStanDefinition read GetParentDefinition write
SetParentDefinition;
  // published
  property Params: TStrings read GetParams write SetParams;
  property Name: String read GetName write SetName;
  property MonitorBy: String read GetMonitorBy write SetMonitorBy;
  property OnChanging: TNotifyEvent read GetOnChanging write SetOnChanging;
  property OnChanged: TNotifyEvent read GetOnChanged write SetOnChanged;
end;
```

```
IADStanDefinitions = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2014}']
    function Add: IADStanDefinition;
    function AddInternal: IADStanDefinition;
    function FindDefinition(const AName: String): IADStanDefinition;
    function DefinitionByName(const AName: String): IADStanDefinition;
    procedure Cancel;
    procedure Save(AIfModified: Boolean = True);
    function Load: Boolean;
    procedure Clear;
    property Count: Integer read GetCount;
    property Items[AIndex: Integer]: IADStanDefinition read GetItems; default;
    property Loaded: Boolean read GetLoaded;
    // published
    property AutoLoad: Boolean read GetAutoLoad write SetAutoLoad;
    property Storage: IADStanDefinitionStorage read GetStorage;
    property BeforeLoad: TNotifyEvent read GetBeforeLoad write SetBeforeLoad;
    property AfterLoad: TNotifyEvent read GetAfterLoad write SetAfterLoad;
end;
```

The top-level interface **IADStanDefinitionStorage** represents the external definition storage. There are two kinds of storage implemented, a file based storage and a registry based storage.

A registry-based storage will look for data in keys:

- If **FileName** is specified, at **HKLM\[FileName]**
- If **FileName** is empty, at **HKLM\Software\da-soft\AnyDAC\[DefaultFileName]**

A file-based storage will look for data in paths:

- If **FileName** is specified with directory, **FileName** is used;
- If **FileName** is specified without directory, <executable module directory>**FileName** is used;
- else at <executable module directory>**DefaultFileName**, if **DefaultFileName** specified and it exists;
- else at **GlobalFileName**.

The top-level interface **IADStanDefinition** represents a definition. Its property **Name** contains the definition name, which is required for persistent and private definitions. The **ParentDefinition** property points to the parent definition.

To start editing just modify definition data. Properties **AsString**, **AsBoolean** and **AsInteger** allow to get/set parameter values.

After editing, deleting or adding a persistent definition, updates are not stored to external storage automatically, but are cached in memory. Programmer should call the **Apply** method of the definition object to apply changes to a single object, or call the **Apply** method of the definition list to apply changes to all definitions. Before a definition is stored to external storage, updates may be canceled calling the **Cancel** method of definition or definition list, to cancel all updates in definition list.

Following code shows how to add new definition:

```
var
  oDefs: IADStanDefinitions;
.....
ADCreateInterface(IADStanDefinitions, oDefs);
oDefs.Storage.FileName := 'Employees.ini';
with oDefs.Add do begin
  AsString['Name'] := 'John';
  AsInteger['Age'] := 30;
  AsBoolean['IsManager'] := True;
  MarkPersistent;
  Apply;
end;
```

The top-level interface **IADStanDefinitions** represent a list of definitions. The properties **Items** and **Count** are used to traverse through the list.

If the switch **AutoLoad** is False, then method **Load** has to be called explicitly to load the definitions from the storage. Additionally it is still possible to add further definitions by calling method's **Add** / **AddInternal**. But it is not possible first adding definitions on fly and then loading definitions from the storage. If the switch **AutoLoad** is True, then the first use of an definition (e.g. **FindDefinition**, **Add**, **Count**, etc) triggers the method **Load**. Loading definitions from storage more than once is not allowed (call of **Load** if property **Loaded** is True gives an exception). To clear and initialize list use method **Clear**.

BeforeLoad event is fired right before loading definitions. It may be used, for example, to set the definition file name. **AfterLoad** event is fired after loading definitions. It may be used, for example, to add additional definitions on fly.

3.6 Persistent connection definition list

Interface unit	daADStanIntf
Standard implementation unit	daADStanConnDef

A *persistent connection definition list* is a sub kind of *persistent definition list*. Connection definition list is used to store named sets of connection properties such as hostname, database name, username, etc. It is similar to the old BDE's aliases or DSNs in ODBC. See the following declaration for details:

```
IADStanConnectionDef = interface(IADStanDefinition)
    ['{3E9B315B-F456-4175-A864-B2573C4A2015}']
    // public
    procedure WriteOptions(AFormatOptions: TObject; AUpdateOptions: TObject;
        AFetchOptions: TObject; AResourceOptions: TObject);
    procedure ReadOptions(AFormatOptions: TObject; AUpdateOptions: TObject;
        AFetchOptions: TObject; AResourceOptions: TObject);
    property UserName: String read GetUserName write SetUserName;
    property Password: String read GetPassword write SetPassword;
    property NewPassword: String read GetNewPassword write SetNewPassword;
    property Database: String read GetDatabase write SetDatabase;
    property ExpandedDatabase: String read GetExpandedDatabase;
    property Pooled: Boolean read GetPooled write SetPooled;
    property DriverID: String read GetDriverID write SetDriverID;
    property MonitorBy: String read GetMonitorBy write SetMonitorBy;
end;

IADStanConnectionDefs = interface(IADStanDefinitions)
    ['{3E9B315B-F456-4175-A864-B2573C4A2016}']
    // public
    function AddConnectionDef: IADStanConnectionDef;
    function FindConnectionDef(const AName: String): IADStanConnectionDef;
    function ConnectionDefByName(const AName: String): IADStanConnectionDef;
    property Items[AIndex: Integer]: IADStanConnectionDef read GetConnDefs; default;
end;
```

In addition to defined methods and properties in **IADStanDefinition**, the top-level interface **IADStanConnectionDef** defines common parameters for most kinds of connections. Also, **IADStanConnectionDefs.DefaultFileName** default value is 'ADConnectionDefs.ini'. The **GlobalFileName** property value is loaded from registry key HKCU\Software\da-soft\AnyDAC\ConnectionDefFile, if it exists, otherwise from HKLM\...

The parameter **DriverID** is mandatory. Its value is an identifier for one of AnyDAC's **Phys** layer drivers. If the property **Pooled** is set to True, the **Phys** layer manager will associate a connection pool with this connection definition. Definition cannot change as long as at least one instance of **IADPhysConnection** is associated with this definition.

If the **NewPassword** property is specified, the user password in the database system is set to the new one as soon as the connection to the RDBMS has been established. Please note, that not all AnyDAC **Phys** layer drivers support this feature.

As long as no connection of a specific connection definition is open it is possible to change / remove a connection definition. Otherwise it will be locked until will exist connections. Following code demonstrates the creation on fly of new private connection definition.

```
DEManager.ConnectionDefs.AutoLoad := False;
with DEManager.ConnectionDefs.AddConnectionDef do
begin
  Name:= 'MyDefNew';
  DriverID:= 'MSAcc';
  Database:= '${DEHOME}\DB\Data\DEDemo.mdb';
  AsBoolean['ReadOnly'] := True;
end;
```

3.7 Async operation execution

Interface unit	daADStanIntf
Standard implementation unit	daADStanAsync

AnyDAC supports synchronous and asynchronous execution of operations with additional option.

Following execution modes are support:

- Blocking mode: The operation will be performed in the calling thread. And it is blocked until the operation has finished. If it is the main application thread, the user interface is blocked too.
- Non-blocking mode: The operation will be performed in the background thread. A running thread is blocked until the operation has finished. But the user interface is not blocked and can still respond to keyboard and mouse events.
- Cancel dialog: The operation will be performed in the background thread. A running thread is blocked until the operation has finished. Although the user interface is blocked, a window with the execution time and a *cancel button* is shown. This allows the user to force an operation to terminate immediately.
- Asynchronous Mode: The operation will be performed in the background thread. A running thread is not blocked and continues to execute, not waiting for operation to be finished.

Following declaration shows details:

```
IADStanAsyncHandler = interface(IIInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2025}']
  procedure HandleFinished(const AInitiator: IADStanObject; AState: TADStanAsyncState);
end;

IADStanAsyncOperation = interface(IIInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2022}']
```



```

    procedure Execute;
    procedure AbortJob;
    function AbortSupported: Boolean;
end;

IADStanAsyncExecutor = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2023}']
    // public
    procedure Setup(const AOperation: IADStanAsyncOperation;
        const AMode: TADStanAsyncMode; const ATimeout: LongWord;
        const AHandler: IADStanAsyncHandler);
    procedure Run;
    procedure AbortJob;
    // R/O
    property State: TADStanAsyncState read GetState;
    property Mode: TADStanAsyncMode read GetMode;
    property Timeout: LongWord read GetTimeout;
    property Operation: IADStanAsyncOperation read GetOperation;
    property Handler: IADStanAsyncHandler read GetHandler;
end;

```

The **IADStanAsyncOperation** interface represents the operation to be performed. The method **Execute** performs action. If operation may be aborted, then **AbortSupported** method returns True. Calling **AbortJob** method will cancel operation, if it supported.

The top-level interface **IADStanAsyncExecutor** represents the asynchronous execution engine. The method **Setup** initializes engine and **Run** actually performs operation. After operation is finished and if **Handler** is assigned, the method **Handler.HandleFinished** will be called. Also, if **Timeout** is not \$FFFFFFFF and mode is Blocking, then operation will be performed in background thread.

The **IADPhysCommand** interface implementation uses **Options.ResourceOptions.AsyncXXXX** properties to perform **Execute** / **Open** / **Fetch** operations in one of execution modes. If execution mode is Cancel dialog, then you can use **TADGuiFormsAsyncExecuteDialog** component. It links standard dialog to application.

3.8 Parameters and macros

Interface unit	daADStanParam
Standard implementation unit	daADStanParam

The AnyDAC command parameters definition is a superset of standard Delphi **TParam** and **TParams** classes. Parameters do not have COM interfaces and are represented as the classes **TADParam** and **TADParams**. The main difference to the old Delphi parameters is the support for an array of values for batch operations, Oracle PL/SQL tables and Unicode strings.

Macros are a unique feature of AnyDAC. They are similar to parameters – both are named variables in the body of a RDBMS command, but parameter values are transmitted to RDBMS and AnyDAC expands macros into RDBMS command text. So, RDBMS does not see macros. Hence, macros can be used where parameters cannot, for example to parameterize table names in SQL FROM clauses. The classes **TADMacro** and **TADMacros** represent a macro and a list of macros.

3.9 Options

Interface unit	daADStanOption
Standard implementation unit	daADStanOption

A large set of options makes AnyDAC a flexible database framework. Options are organized in five groups:

- **TADFetchOptions** - Fetch options control, how the **Phys** layer command will fetch data from the RDBMS. For example, it is possible to fetch all records at once, or fetch records on demand.
- **TADFormatOptions** - Format options control, how RDBMS data types will be mapped to the data types available on the client and vice-versa. For example, a programmer may setup a mapping for Oracle NUMBER (38) onto **dtBCD** or onto **dtInt64**.
- **TADUpdateOptions** - Update options control, how AnyDAC will post updates to RDBMS. For example, during an update AnyDAC can update all fields in a table or only the changed ones.
- **TADResourceOptions** - Resource options control, how system resources are used. For example, a AnyDAC **Phys** layer command can be performed asynchronously or blocked.
- **TADTxOptions** - Transaction options control, how transactions are performed. For example, perform them in **ReadCommitted** mode.

Although AnyDAC introduces a lot of options, setting up each command makes programming complex and error prone. AnyDAC solves this issue by introducing the *parent-child option values inheritance model*. Option values are propagated from parent to child (top-down). If a lower level has no option value assigned explicitly, the value will be taken from the higher level. At each level, options are represented by the **IADStanOptions** interface, which collects the first four options groups in the single entity. The following declaration shows the details:

```
IADStanOptions = interface (IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2018}']
    // public
    property FetchOptions: TADFetchOptions read GetFetchOptions;
    property FormatOptions: TADFormatOptions read GetFormatOptions;
    property UpdateOptions: TADUpdateOptions read GetUpdateOptions;
```

```
property ResourceOptions: TADResourceOptions read GetResourceOptions;  
property ParentOptions: IADStanOptions read GetParentOptions;  
end;
```

Here, **ParentOptions** property points to the higher level (if it exists), or it is set to NIL.

The AnyDAC **Phys** layer has Manager, Connection and Command entities. From the options' point of view, they are levels. A Manager is at the top level, a Connection is at the intermediate and a Command is at the bottom level. So, by setting any particular Manager option, all Commands will inherit its value.

This is true as long as the programmer has not explicitly assigned a value to the Command option.

Examples:

- The mapping of data types from the RDBMS types to the client types defined in **FormatOptions** is inherited by all Commands from their Connection.

```
with oConnection.Options.FormatOptions do begin  
  OwnMapRules := True;  
  MapRules.Clear;  
  with MapRules.Add do begin  
    PrecMax := 19;  
    PrecMin := 4;  
    SourceDataType := dtFmtBCD;  
    TargetDataType := dtCurrency;  
  end;  
end;
```

- A Data Warehouse application may setup high-speed fetching mode, using **FetchOptions** of the Manager level. So, all connection and all their commands will inherit these options.

```
with ADPhysManager.Options.FetchOptions do begin  
  Items := [];  
  Cache := [];  
  RowsetSize := 200;  
end;
```

- The OLTP application may set optimistic locking mode in **UpdateOptions** for specific data adapters.

3.10 Debug monitoring

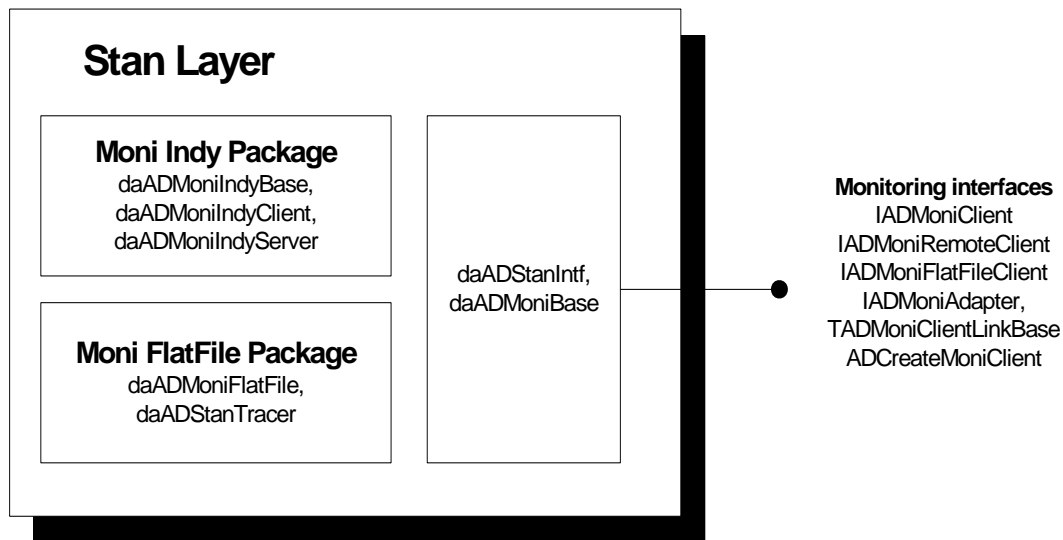


Figure 4 Moni Packages Overview

Interface unit	daADStanIntf
Standard implementation unit	daADMoniIndyClient, daADMoniFlatFile, daADMoniCustom

AnyDAC defines and implements *debug monitor client* interface, which allow monitoring the interaction of AnyDAC application and DBMS. The monitor client is build into the application. It packs the trace of application execution into the messages and sends them to *the server part*. This requires some sort of a message transport implementation. Today AnyDAC has 3 implementations:

- *Flat File*. The implementation is inside of **MoniFlatFile** package. The monitor client output the trace of application execution into the flat ANSI text file. So, the file may be inspected by Windows Notepad application (server part <g>).
- *Remote monitoring*. The implementation is inside of **MoniIndy** package. The monitor client sends binary formatted messages to the server part using Indy TCP/IP components. The server part is the AnyDAC Monitor application, located in \$(AnyDAC)\Tool\Monitor directory.
- *Custom monitoring*. The implementation is inside of **MoniCustom** package. The monitor client fires **OnOutput** event and it is up to the programmer how the output will be formed.

The remote monitoring (additionally to flat file one) allowing inspecting the current state of **Phys**, **DApt** and **Comp** layer objects inside of AnyDAC application. Also AnyDAC Monitor is able to monitor few

applications running as on the same machine, as on any other machine which has TCP/IP link with machine running Monitor. Following picture illustrates this:

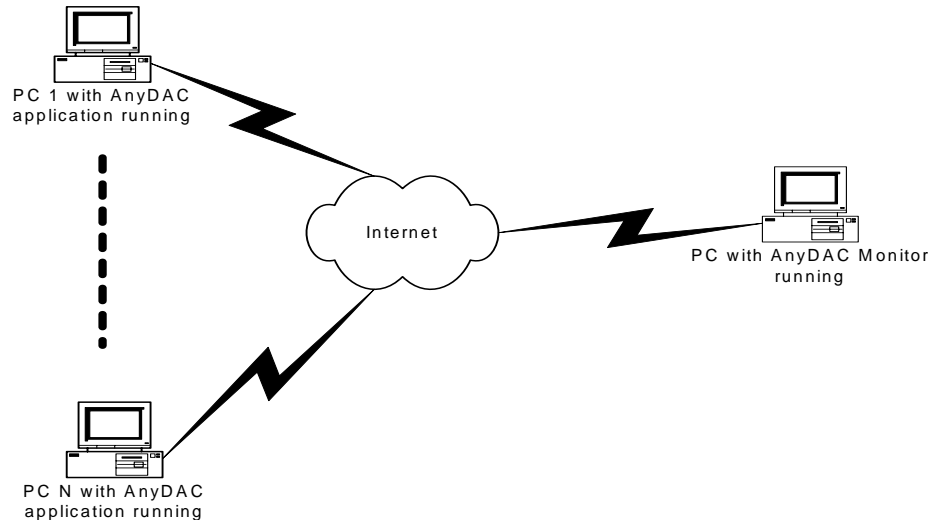


Figure 5 Remote Monitoring network

Generic monitor client is represented by **IADMoniClient** interface. Following shows it details:

```
IADMoniClient = interface (IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2005}']
    procedure SetupFromDefinition(const AParams: IADStanDefinition);
    procedure ResetFailure;
    procedure Notify(AKind: TADDebugEventKind; AStep: TADDebugEventStep;
        ASender: TObject; const AMsg: String; AArgs: array of const);
    function RegisterAdapter(const AAdapter: IADMoniAdapter): LongWord;
    procedure UnregisterAdapter(const AAdapter: IADMoniAdapter);
    procedure AdapterChanged(const AAdapter: IADMoniAdapter);
    property Tracing: Boolean read GetTracing write SetTracing;
    property Name: TComponentName read GetName write SetName;
    property EventKinds: TADDebugEventKinds read GetEventKinds write SetEventKinds;
end;
```

Property **Name** allows setting a client name, which will appear as client identifier on server. Property **Tracing** enables or disables client output. Method **Notify** outputs tracing message. Method **RegisterAdapter** registers adapter with client and returns adapter handle. **UnregisterAdapter** method finishes registration. Method **AdapterChanged** notifies monitor client about change of adapter state. The client will send adapter state to the server and server will display updated information.

Then, each monitor client kind has own top-level COM interface. Following shows it details:

```
IADMoniRemoteClient = interface (IADMoniClient)
    ['{3E9B315B-F456-4175-A864-B2573C4A2026}']
    property Host: String read GetHost write SetHost;
    property Port: Integer read GetPort write SetPort;
```

```
property Timeout: Integer read GetTimeout write SetTimeout;
end;

IADMoniFlatFileClient = interface (IADMoniClient)
    ['{3E9B315B-F456-4175-A864-B2573C4A2027}']
    property FileName: String read GetFileName write SetFileName;
    property FileAppend: Boolean read GetFileAppend write SetFileAppend;
end;
```

AnyDAC may create only one instance of each monitor client kind. So, all RDBMS connections in the same application will share monitoring clients. By default, monitoring clients are not linked into AnyDAC application. Drop one of (or all) following components to one of your forms or data modules to link monitor client implementation to application:

- **TADMoniIndyClientLink** – for remote monitoring.
- **TADMoniFlatFileClientLink** – for flat file monitoring.
- **TADMoniCustomClientLink** – for custom monitoring.

Few link components of the same class will share the same monitor client. Following example sends messages to the trace:

```
var
    oConnDef: IADStanDefinition;
    oRemMoni: IADMoniRemoteClient;
    oMoni: IADMoniClient;
    i: Integer;
    iFactorial: Integer;
.....
ADCreateInterface(IADMoniRemoteClient, oRemMoni);
oMoni := oRemMoni as IADMoniClient;
i := 1;
iFactorial := 1;
oMoni.Notify(ekVendor, esStart, nil, 'Starting', ['i', i, 'iFactorial', iFactorial]);
while i < N do begin
    Inc(i);
    iFactorial := iFactorial * i;
    oMoni.Notify(ekVendor, esProgress, nil, 'Calculating', ['i', i, 'iFactorial', iFactorial]);
end;
oMoni.Notify(ekVendor, esEnd, nil, 'Finishing', ['i', i, 'iFactorial', iFactorial]);
```

The remote monitoring uses *monitor adapters* to inspect object state. For that object must implement monitor adapter interface and register itself with monitor client. Adapter interface publishes properties and represents an item of AnyDAC application hierarchical structure, identifying itself by unique path in this hierarchy. AnyDAC Monitor reconstructs and displays AnyDAC application structure and its item properties. Following picture illustrates this:

Objects	Trace																					
<div><div><div></div><div>dmlMainComp</div></div><div><div></div><div>dbMain</div></div></div> <div><div><div></div><div>frmAggregates</div></div><div><div></div><div>qryAggregates</div></div></div>	<table><tr><th>No</th><th>Time</th><th>Text</th></tr><tr><td>0</td><td>20:22:57:421</td><td>***** Client [H: 127.</td></tr><tr><td>1</td><td>20:22:57:655</td><td>. Aggregates 7c4</td></tr><tr><td>2</td><td>20:22:57:717</td><td>. Aggregates 7c4</td></tr><tr><td>3</td><td>20:22:57:717</td><td>>> Create [Connecti</td></tr><tr><td>4</td><td>20:22:57:717</td><td><< Create [Connecti</td></tr><tr><td>5</td><td>20:22:57:717</td><td>. Annrenates 7c4</td></tr></table>	No	Time	Text	0	20:22:57:421	***** Client [H: 127.	1	20:22:57:655	. Aggregates 7c4	2	20:22:57:717	. Aggregates 7c4	3	20:22:57:717	>> Create [Connecti	4	20:22:57:717	<< Create [Connecti	5	20:22:57:717	. Annrenates 7c4
No	Time	Text																				
0	20:22:57:421	***** Client [H: 127.																				
1	20:22:57:655	. Aggregates 7c4																				
2	20:22:57:717	. Aggregates 7c4																				
3	20:22:57:717	>> Create [Connecti																				
4	20:22:57:717	<< Create [Connecti																				
5	20:22:57:717	. Annrenates 7c4																				

Monitor adapter is represented by **IADMoniAdapter** interface. Following shows details:

```
IADMoniAdapter = interface (IInterface)
    procedure GetItem(AIndex: Integer; var AName: String; var AValue: Variant;
        var AKind: TADDebugMonitorAdapterItemKind);
    property Handle: LongWord read GetHandle;
    property ItemCount: Integer read GetItemCount;
end;
```

Property **Handle** must return handle received from **RegisterAdapter** call. **ItemCount** property returns a number of properties published by adapter. And method **GetItem** actually returns property information. The adapted object also must implement **IADStanObject** interface. The client determines adapter place in the application hierarchy using it **Name** and **Parent** properties. It is adapter responsibility to notify monitor client about its state change, so client will send updated info to the server. For that adapter should call method **IADMoniClient.AdapterChanged**. For example:

```
procedure TADPhysCommand.Prepare(const ACommandText: String = '');
begin
    .....
    if FConnection.Tracing then
        FConnection.Monitor.AdapterChanged(Self);
end;
```

4 DatS Layer

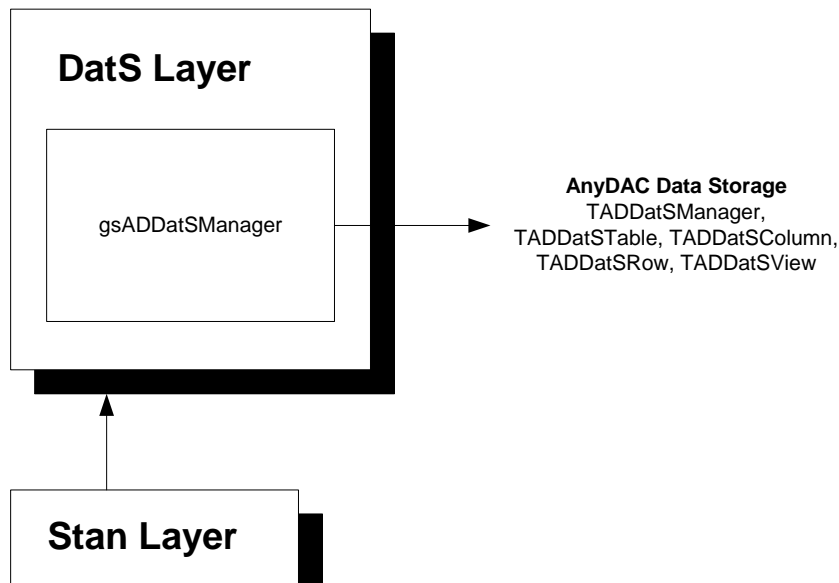


Figure 6 DatS Layer Overview

DatS layer consists of single package called DatS. All interfaces of this layer are exposed as Delphi classes. The COM is not used here (with small exceptions), due to complexity and performance. The **DatS** layer makes use of the resources from the **Stan** layer:

- Expression evaluation engine (**IADStanExpressionDS**, **IADStanExpressionParser**, **IADStanExpression** interfaces)
- Standard routines, functions, constants, types.

DatS layer is actually *In-Memory Data Storage* with relational database engine limited capabilities. DatS is independent on any specific RDBMS and database structure. Moreover it may be used in the same time with many data sources. Following diagram shows simplified object model of **DatS** layer:

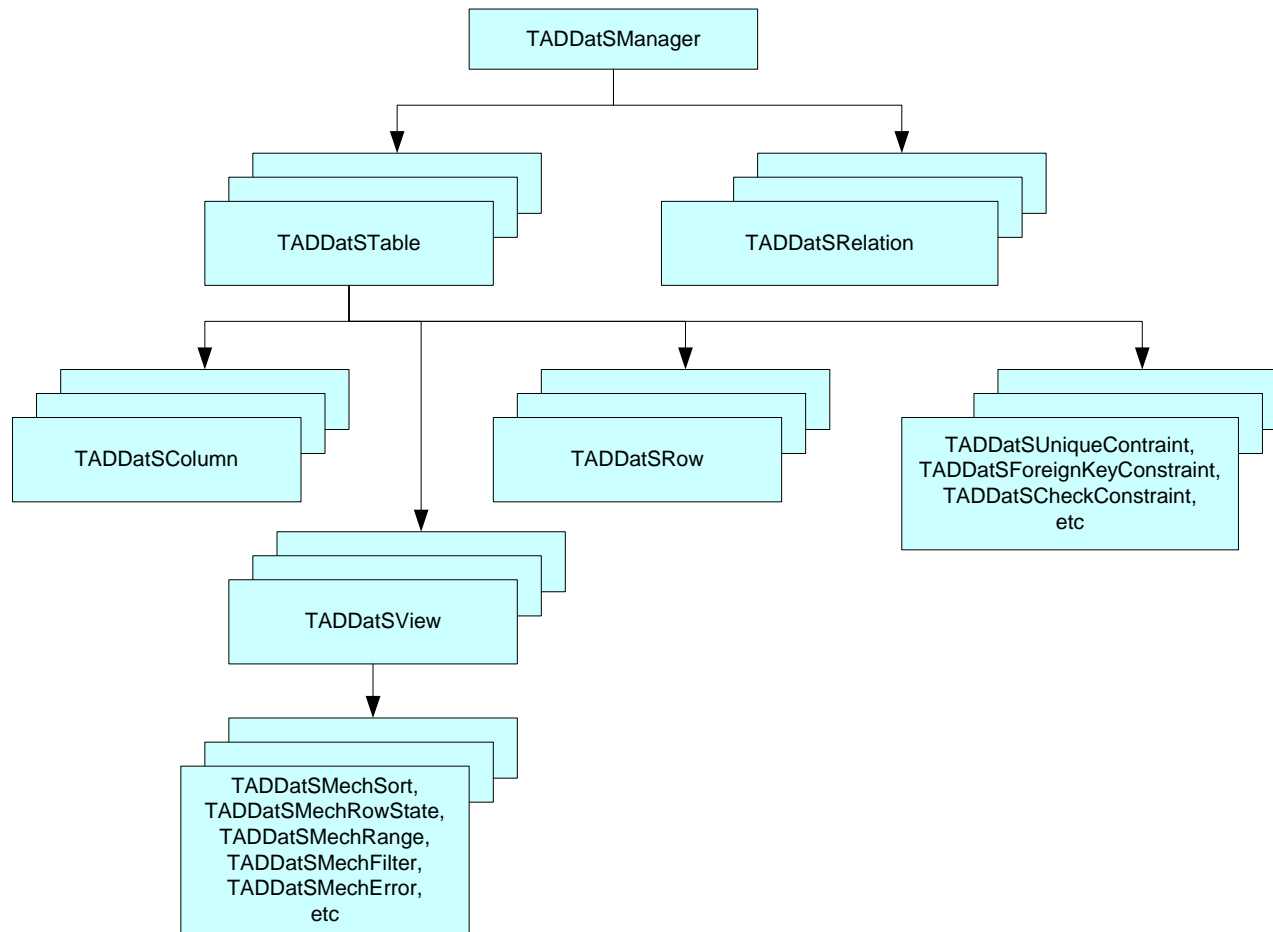


Figure 7 DatS Class Diagram

AnyDAC **DatS** layer by architecture is similar to the ADO.NET®. Most information sources about Ado.Net may be successfully used for **DatS** layer. The table below lists the point in which the two products differ.

Feature	Ado.Net	DatS layer
Destruction of objects.	Implicit. It is responsibility of garbage collector.	Explicit. Programmer is responsible for destruction of “root” objects (like a manager, table). “Root” object will destroy “sub” objects (like tables in manager, columns in table, etc) automatically.
Class names.	DataXXXX	TADDatSXXXX . DatS follows AnyDAC naming conventions.
	DataSet	TADDatSManager
Data types.	Not limited to strict set.	Defined by TADDDataType enumeration type. This is because of differences between Delphi RTL and .NET CLR.
Collections.	Support of ICollection and .Net foreach keyword.	DatS supports: property ItemI[AIndex: Integer]: <object>; property ItemS[AIndex: String]: <object>; property Count: Integer;
Internals.	Own implementation.	Own implementation.

4.1 DatS Manager

The *root object* of DatS is a **TADDatSManager**. DatS manager is a set of tables and relationships between tables. Tables itself contain a data. Relationships allow navigating through these data, using hierarchical relations. The programmer is responsible for defining relationships between tables. The table structure may be defined by the programmer or by the **Phys** layer command.

4.2 DatS Table

The **TADDatSTable** class is *the central object* in the **DatS** layer. A data table itself is a collection of columns, rows, constraints and views. To create new table, just create it using it constructor.

4.2.1 DatS Column

A column collection defines the structure of table rows. Columns represent meta information and do not store data itself. Data is stored in rows only. To create rows, the programmer should add at least one column to

the columns collection of a table. The class **TADDatSColumn** represents a column. The following declaration shows its details:

```
TADDatSColumn = class (TADDatSBindedObject)
public
    // rw
    property AllowDBNull: Boolean read GetAllowDBNull write SetAllowDBNull
        default True;
    property Attributes: TADDataAttributes read FAttributes write
        SetAttributes default [caAllowNull];
    property AutoIncrement: Boolean read FAutoIncrement write SetAutoIncrement
        default False;
    property AutoIncrementSeed: Integer read FAutoIncrementSeed write
        SetAutoIncrementSeed default 1;
    property AutoIncrementStep: Integer read FAutoIncrementStep write
        SetAutoIncrementStep default 1;
    property Caption: String read GetCaption write SetCaption;
    property DataType: TADDataType read FDataType write SetDataType
        default dtUnknown;
    property Expression: String read FExpression write SetExpression;
    property Options: TADDataOptions read FOptions write SetOptions
        default [coAllowNull, coInUpdate, coInWhere];
    property Precision: Integer read FPrecision write SetPrecision default 0;
    property ReadOnly: Boolean read GetReadOnly write SetReadOnly default False;
    property Scale: Integer read FScale write SetScale default 0;
    property Size: LongWord read FSize write SetSize default 50;
    property SourceDataType: TADDataType read FSourceDataType write FSourceDataType;
    property SourcePrecision: Integer read FSourcePrecision write FSourcePrecision;
    property SourceScale: Integer read FSourceScale write FSourceScale;
    property SourceSize: LongWord read FSourceSize write FSourceSize;
    property SourceDataTypeName: String read FSourceDataTypeName
        write FSourceDataTypeName;
    property Unique: Boolean read GetUnique write SetUnique default False;
end;
```

Main properties of **TADDatSColumn** are:

- **Name** – name of column (inherited from base class).
- **DataType** – type of column data. For details, see Appendix 1. **TADDataType**. AnyDAC does not allow extending the set of data types like ADO.NET (with some exception).
- **Size** – maximum size of string data types, as ANSI strings, UNICODE strings and byte strings. Size is specified in data type units, not in bytes.
- **Precision** and **Scale** – precision and scale of real numeric data types such as double, currency, BCD and FmtBCD.

If DatS table has rows and programmer will change **DataType** or **Size** properties, exception will be raised. Property **Attribute** contains a set of column objective characteristics. And property **Options** contains a set of characteristics assigned to column by user. For example, **caAllowNull** excluded from **Attribute** of binded

column means, column in DB table requires value to be specified. By default, including / excluding **caAllowNull** in / from **Attribute**, will include / exclude **coAllowNull** in / from **Options**. But, only **coAllowNull** controls column value optionality. The similar is about **caReadOnly** and **coReadOnly**.

The programmer may use the properties **AutoIncrement**, **AutoIncrementSeed**, **AutoIncrementStep** to control automatic column data generation. If **AutoIncrement** is set to True, the data table generates new unique column value when a new row is added to the table. For example, following code will add column incrementing with -1 step, starting from 10:

```
var
  oTab: TADDatSTable;
.....
with oTab.Columns.Add('ID', dtInt32) do begin
  AutoIncrement := True;
  AutoIncrementSeed := 10;
  AutoIncrementStep := -1;
end;
```

The property **Expression** defines expressions to calculate column value, based on the values of other columns in the same row. If the **caCalculated** attribute is included into **Attributes**, column value is calculated only by **Expression** and the programmer cannot change its value. If the **caCalculated** attribute not included, **Expression** value is only the default value of the column. So, the value will be assigned each time a new row is created and the programmer may change the column value. Following example will create calculated column:

```
var
  oTab: TADDatSTable;
.....
with oTab.Columns.Add('Total', dtCurrency) do
  Expression := 'ItemPrice * ItemAmount';
```

The set of properties **SourceXXXX** maps column to data source column from which data will be fetched into table. The property **SourceID** is inherited from **TADDatSBindedObject**. It *binds* column to data source field with the specified ID (sequential number). Worth to mention, that setting **Expression** for bind column (**SourceID** > 0) will automatically make that default value expression, otherwise calculated column.

4.2.2 DatS Row

The *data row collection* stores table data. Each data row has a layout as defined by the *table column collection*. Rows may be added either programmatically or using **Phys** layer objects (fetched from data source). **TADDatSRow** class represents data rows. The following declaration shows details:

```
TADDatSRow = class (TADDatSObject)
public
  procedure AcceptChanges;
  procedure AssignDefaults;
  procedure BeginEdit;
```

```

procedure CancelEdit;
procedure Clear(ASetColsToDefaults: Boolean);
procedure ClearErrors;
procedure Delete(ANotDestroy: Boolean = False);
procedure EndEdit;
function GetChildRows(const AChildRelationName: String): TADDatSView; overload;
function GetChildRows(AChildTable: TADDatSTable): TADDatSView; overload;
function GetChildRows(AChildRelation: TADDatSRelation): TADDatSView; overload;
function GetParentRows(const AParentRelationName: String): TADDatSView; overload;
function GetParentRows(AParentTable: TADDatSTable): TADDatSView; overload;
function GetParentRows(AParentRelation: TADDatSRelation): TADDatSView; overload;
function GetData(const AColumnName: String;
    AVersion: TADDatSRowVersion = rvDefault): Variant; overload;
function GetData(AColumn: Integer;
    AVersion: TADDatSRowVersion = rvDefault): Variant; overload;
function GetData(AColumn: TADDatSColumn;
    AVersion: TADDatSRowVersion = rvDefault): Variant; overload;
function GetData(AColumn: Integer; AVersion: TADDatSRowVersion; var
    ABuff: Pointer; ABuffLen: LongWord; var ADataLen: LongWord; AByVal:
    Boolean): Boolean; overload;
function HasVersion(AVersion: TADDatSRowVersion): Boolean;
procedure RejectChanges;
procedure SetData(AColumn: Integer; const AValue: Variant); overload;
procedure SetData(AColumn: TADDatSColumn; const AValue: Variant); overload;
procedure SetData(AColumn: Integer; ABuff: Pointer; ADataLen: LongWord); overload;
procedure SetValues(const AValues: array of Variant);
property ParentRow: TADDatSRow read GetParentRow write SetParentRow;
property NestedRow[AColumn: Integer]: TADDatSRow read GetNestedRow write SetNestedRow;
property NestedRows[AColumn: Integer]: TADDatSNestedRowList read GetNestedRows;
property Fetched[AColumn: Integer]: Boolean read GetFetched write SetFetched;
property HasErrors: Boolean read GetHasErrors;
property RowError: EADException read GetRowError write SetRowErrorPrc;
property RowState: TADDatSRowState read FRowState;
property RowPriorState: TADDatSRowState read FRowPriorState;
property ValueI[AColumn: Integer; AVersion: TADDatSRowVersion]: Variant read GetDataI;
property ValueO[AColumn: TADDatSColumn; AVersion: TADDatSRowVersion]: Variant read GetDataO;
default;
end;

```

The property **RowState** indicates current row state. The row has the following live cycle:

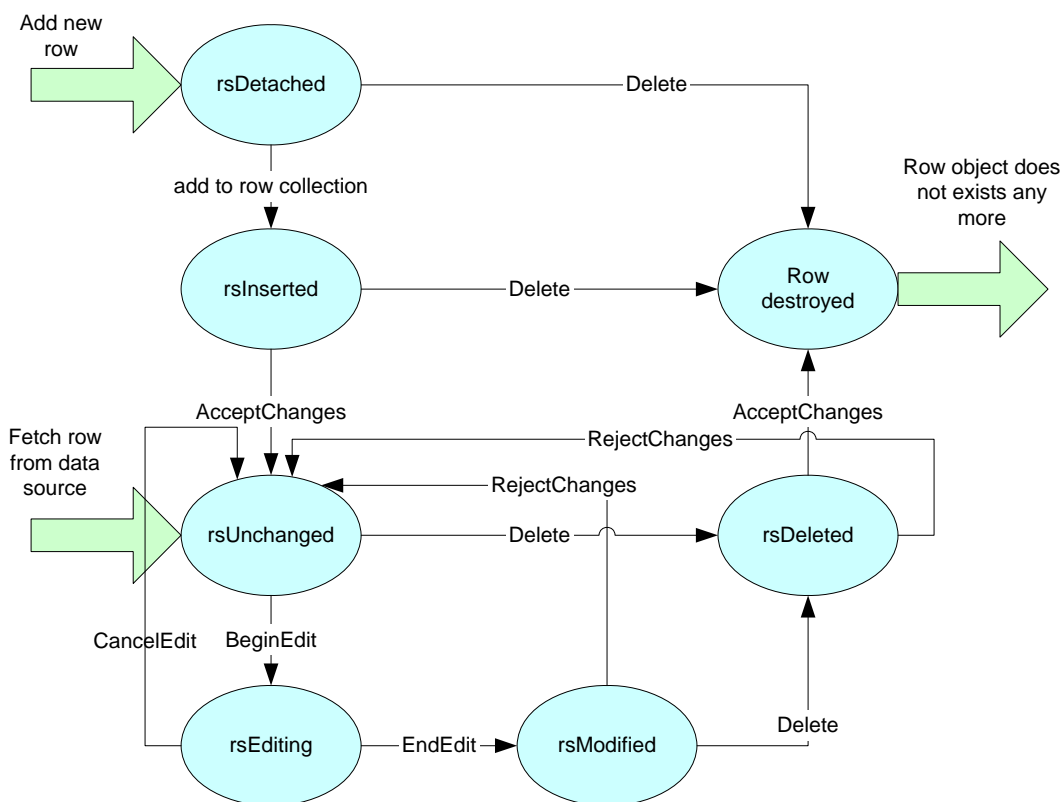


Figure 8 Life Cycle of DatS Row

AnyDAC distinct two main kind of rows, those fetched from data source (**rsUnchanged**) and those added programmatically (**rsDetached** / **rsInserted**).

The method **TADatSTable.NewRow** creates new row and marks it as **rsDetached**. The row is associated with the table, but is not added to the table row list. Method **TADatSTable.Rows.Add** will add row to the table row list and will mark row as **rsInserted**. This row may be posted to database. Following code will create new row at set it column values:

```

oRow := oTab.NewRow;
// at moment row is detached
oRow.SetData(0, 1000);
oRow.SetData(1, 'Delphi');
oRow.SetData(2, Today());
oTab.Rows.Add(oRow);
// at moment row is inserted
  
```

BeginEdit places row into **rsEditing** state, allowing modifications to row. **BeginEdit** cannot be called for detached or marked for deletion row. **EndEdit** call places fetched rows into **rsModified** state. **CancelEdit** call undoes all changes made since last **BeginEdit** call. Before data can be edited, fetched column values will be stored as *row original version*. Therefore, two versions will be accessible:

- Original – as it was fetched.

- Current – as it is after editing.

Method **Delete** will mark row as **rsDeleted** if it was fetched and, optionally, was modified. If the row was just inserted into the row list, then row will be deleted from the list and destroyed.

The method **AcceptChanges** transfers a row to the **rsUnchanged** state. After that, the row is in the same state as a row just fetched. The method **RejectChanges** undoes all changes made to row after it was fetched, or since the method **AcceptChanges** was called.

The overloaded methods **GetData** read a column value. The overloaded methods **SetData** modify a column value. Column value modification is possible, if the column's **ReadOnly** property value is False and row is in state "detached" or "editing".

When the **Phys** layer posts back row changes to the data source, the RDBMS engine may raise an error, for example a data integrity violation error. AnyDAC will store this error in the row. The property **RowError** refers to it.

4.2.3 DatS constraint

A constraint is a data integrity rule used in data tables. Each constraint acts on single row at every moment. It is defined on a set of columns belonging to a data table. If a constraint rule is violated, the **DatS** layer will raise an exception. The following declaration shows details of a constraint base class:

```
TADDatSConstraintBase = class (TADDatSNamedObject)
public
  procedure Check(ARow: TADDatSRow; AProposedState: TADDatSRowState;
    ATime: TADDatSCheckTime);
  procedure CheckAll; virtual;
  property ActualEnforce: Boolean read GetActualEnforce;
  property ConstraintList: TADDatSConstraintList read GetConstraintList;
  property Enforce: Boolean read FEnforce write SetEnforce default True;
  property Rely: Boolean read FRely write FRely default True;
  property CheckTime: TADDatSCheckTime read FCheckTime write FCheckTime default ctAtEditEnd;
  property Message: String read FMessage write FMessage;
end;
```

The property **CheckTime** defines when a constraint will be checked:

- **ctAtEditEnd** - at the end of an edit operation (**EndEdit** / **Delete** / Add a row to rows collection).
- **ctAtColumnChange** – each time when any of columns validated by constraint is modified.

Only if the property **Enforce** is True, constraint will be checked. When the programmer activates constraint checking and **Rely** is False, then all table rows will be checked. If any row violates the constraint, then **Enforce** will be turned back to False. If **Rely** is True, then constraint will be enforced without checking all data table rows.

The property **Message** defines the message of the constraint violation exception.

AnyDAC implements three kinds of constraints. Each of them is implemented as a subclass of

TADDatSConstraintBase:

- **TADDatSUniqueConstraint** – unique / primary key constraint.
- **TADDatSForeignKeyConstraint** – foreign key constraint. It supports cascading operations.
- **TADDatSCheckConstraint** – check constraint.

Also **TDatSTable.Constraints** has few methods to fast add constraints. For example, to add Primary Key constraint, you can code:

```
var
  oPK: TADDatSUniqueConstraint;
  oCustTab: TDatSTable;
.....
oPK := TADDatSUniqueConstraint.Create;
with oPK do begin
  Name := 'CustomersPK';
  ColumnNames := 'CustNo';
  IsPrimaryKey := True;
end;
oCustTab.Constraints.Add(oPK);
```

4.2.4 DatS View

Data view represents a subset of table rows. A view does not change a row layout, nor does it add new columns or hide others. A view strictly operates on rows of a single *source object*. It may be either table, either another view of the same table. A view may have aggregate values, which will be calculated from *visible* rows in a view. The following declaration shows details:

```
TADDatSView = class (TADDatSNamedObject)
  constructor Create; overload; override;
  constructor Create(ATable: TADDatSTable; const AFilter: String = '');
  const ASort: String = ''; AStates: TADDatSRowStates = []; overload;
  constructor Create(ATable: TADDatSTable; const ABaseName: String;
    ACreator: TADDatSViewCreator; ACountRef: Boolean = True); overload;
  procedure Clear;
  procedure Rebuild;
  function Find(const AValues: array of Variant; AOptions: TADLocateOptions = []): Integer;
  overload;
  function Find(ARow: TADDatSRow; AOptions: TADLocateOptions = [];
    ARowVersion: TADDatSRowVersion = rvDefault): Integer; overload;
  function Search(AKeyRow: TADDatSRow; AKeyColumnList,
    AKeyColumnList2: TADDatSColumnSublist; AKeyColumnCount: Integer;
    AOptions: TADLocateOptions; var AIndex: Integer; var AFound: Boolean;
    ARowVersion: TADDatSRowVersion = rvDefault): Integer;
  function IndexOf(AKeyRow: TADDatSRow; ARowVersion: TADDatSRowVersion = rvDefault): Integer;
  overload;
  function Locate(var ARowIndex: Integer; AGoForward: Boolean = True;
    ARestart: Boolean = False): Boolean;
  function GetGroupState(ARecordIndex, AGroupingLevel: Integer): TADDatSGroupPositions;
```



```

procedure DeleteAll;
// ro
property ActualActive: Boolean read GetActualActive;
property Actual: Boolean read GetActual;
property SortingMechanism: IADDatSMechSort read FSortingMechanism;
property GroupingLevel: Integer read GetGroupingLevel;
property Mechanisms: TADDatSViewMechList read FMechanisms;
property Rows: TADDatSRowListBase read FRows;
property Aggregates: TADDatSAggregateList read FAggregates;
// rw
property Active: Boolean read FActive write SetActive;
property SourceView: TADDatSView read FSourceView write SetSourceView;
property RowFilter: String read GetRowFilter write SetRowFilter;
property RowStateFilter: TADDatSRowStates read GetRowStateFilter
    write SetRowStateFilter default [];
property Sort: String read GetSort write SetSort;
end;

```

After a view is created, all source object rows are visible. The property **SourceView** allows setting another view as a source object. If it is **nil**, then table is a source object. So, in general, few views may be linked into the line. Views will be updated, starting from one having the table as a source object. For example:

```

var
    OTab: TADDatSTable;
    oView1, oView2, oView3: TADDatSView;
.....
// create views
oView1 := TADDatSView.Create;
OTab.Views.Add(oView1);
oView2 := TADDatSView.Create;
OTab.Views.Add(oView2);
oView3 := TADDatSView.Create;
OTab.Views.Add(oView3);

// link views
oView2.SourceView := oView1;
oView3.SourceView := oView2;

// activate views
oView1.Active := True;
oView2.Active := True;
oView3.Active := True;

```

There oView1 has oTab table as source object. For oView2 source object is oView1 and for oView3 source object is oView2. So, first will be updated oView1, then oView2, then oView3.

When the programmer edits rows, the set of visible rows will be automatically updated. They will be accessible through the **Rows** property. Setting property **Active** to False, view will be not automatically updated and **Rows** will remain unchanged (snapshot). If a visible rows set is still actual, then property **Actual** returns True.

4.2.4.1 Mechanism

To filter and sort rows, the programmer should create *mechanisms* and add them to the view's *mechanism collection* (property **Mechanisms**). The **DatS** layer implements few kind of data mechanisms. Each of these mechanisms differs in the rows filtering and sorting abilities. The following declaration shows the details of a mechanisms base class:

```
TADDatSMechBase = class (TADDatSObject)
public
    function GetRowsRange(var ARowList: TADDatSRowListBase; var ABeginInd,
        AEndInd: Integer): Boolean; virtual;
    function AcceptRow(ARow: TADDatSRow; AVersion: TADDatSRowVersion):
        Boolean; virtual;
    // rw
    property Active: Boolean read FActive write SetActive default False;
    property Locator: Boolean read FLocator write FLocator default False;
end;
```

The property **Active** activates a mechanism, so it will participate in filtering and sorting rows. The property **Locator** works together with the **Locate** method of the view. If it is True, the mechanism does not filter rows, but will be used by the **Locate** method to find rows meeting the filtering conditions.

The **DatS** layer defines the following mechanisms:

- **TADDatSMechSort** – sorts rows.
- **TADDatSMechRowState** – filters rows based on row state.
- **TADDatSMechRange** – filters rows, sorted before by sorting mechanism. Filtering is based on range values (top, bottom) of sorted columns.
- **TADDatSMechFilter** – filters rows using boolean expression
- **TADDatSMechError** – filters rows with errors
- **TADDatSMechDetails** – filters rows using data relations. For a master, detail rows are filtered.
- **TADDatSMechMaster** – filters rows using data relations. For detail row, master row is filtered.

For example, following code shows, how to create view listing customers from state 'AZ' and sorting them by name:

```
var
    oView: TADDatSView;
    oFlt: TADDatSMechFilter;
    oSort: TADDatSMechSort;
.....
    // create view
    oView := TADDatSView.Create;
    oView.Active := True;
    // create filtering mechanism
```

```

oFlt := TADDatSMechFilter.Create;
oFlt.Expression := 'State = 'AZ''';
oFlt.Active := True;
oView.Mechanisms.Add(oFlt);
// create sorting mechanism
oSort := TADDatSMechSort.Create;
oSort.Columns := 'Name';
oSort.Active := True;
oView.Mechanisms.Add(oSort);
// adding view to view list, right there will be builded
// visible row list
oCustTab.Views.Add(oView);

```

A view has *short cut properties*, which allow adding mechanisms to a view quickly:

- **RowFilter** – creates, defines and adds **TADDatSMechFilter** with specified expression. If **TADDatSMechFilter** instance already exists across mechanisms, then setting this property will modify existing one.
- **RowStateFilter** – creates, defines and adds **TADDatSMechRowState** with specified row states. If **TADDatSMechRowState** instance already exists across mechanisms, then setting this property will modify existing one.
- **Sort** – creates, defines and adds **TADDatSMechSort** with specified list of columns to sort. If **TADDatSMechSort** instance already exists across mechanisms, then setting this property will modify existing one.

Previous example may be rewritten in short form:

```

var
  oView: TADDatSView;
.....
// create view
oView := TADDatSView.Create;
oView.RowFilter := 'State = 'AZ''';
oView.Sort := 'Name';
oView.Active := True;
oCustTab.Views.Add(oView);

```

or even:

```

oView := TADDatSView.Create(oCustTab, 'State = 'AZ''', 'Name');

```

But there exist much more other short cut methods.

After adding mechanism to the view **Mechanisms** list and setting property **Active** to True, mechanism will be activated and participate in view. Similar, after adding view to the table **Views** list and setting property **Active** to True, view will be activated and rows are filtered and sorted.

4.2.4.2 Aggregate

To define aggregates, the programmer must use the **Aggregates** collection property. The aggregate describes a calculation that summarizes the data in a group or in all of the data view rows. The following declaration shows the details:

```
TADDatSAggregate = class (TADDatSNamedObject)
public
  procedure Recalc;
  procedure Update;
  // ro
  property ActualActive: Boolean read GetActualActive;
  property State: TADDatSAggregateStates read FState;
  property Value[ARowIndex: Integer]: Variant read GetValue;
  // rw
  property Expression: String read FExpression write SetExpression;
  property GroupingLevel: Integer read FGroupingLevel write SetGroupingLevel default 0;
  property Active: Boolean read FActive write SetActive default False;
end;
```

The property **Expression** defines a calculation expression.

5 Phys Layer

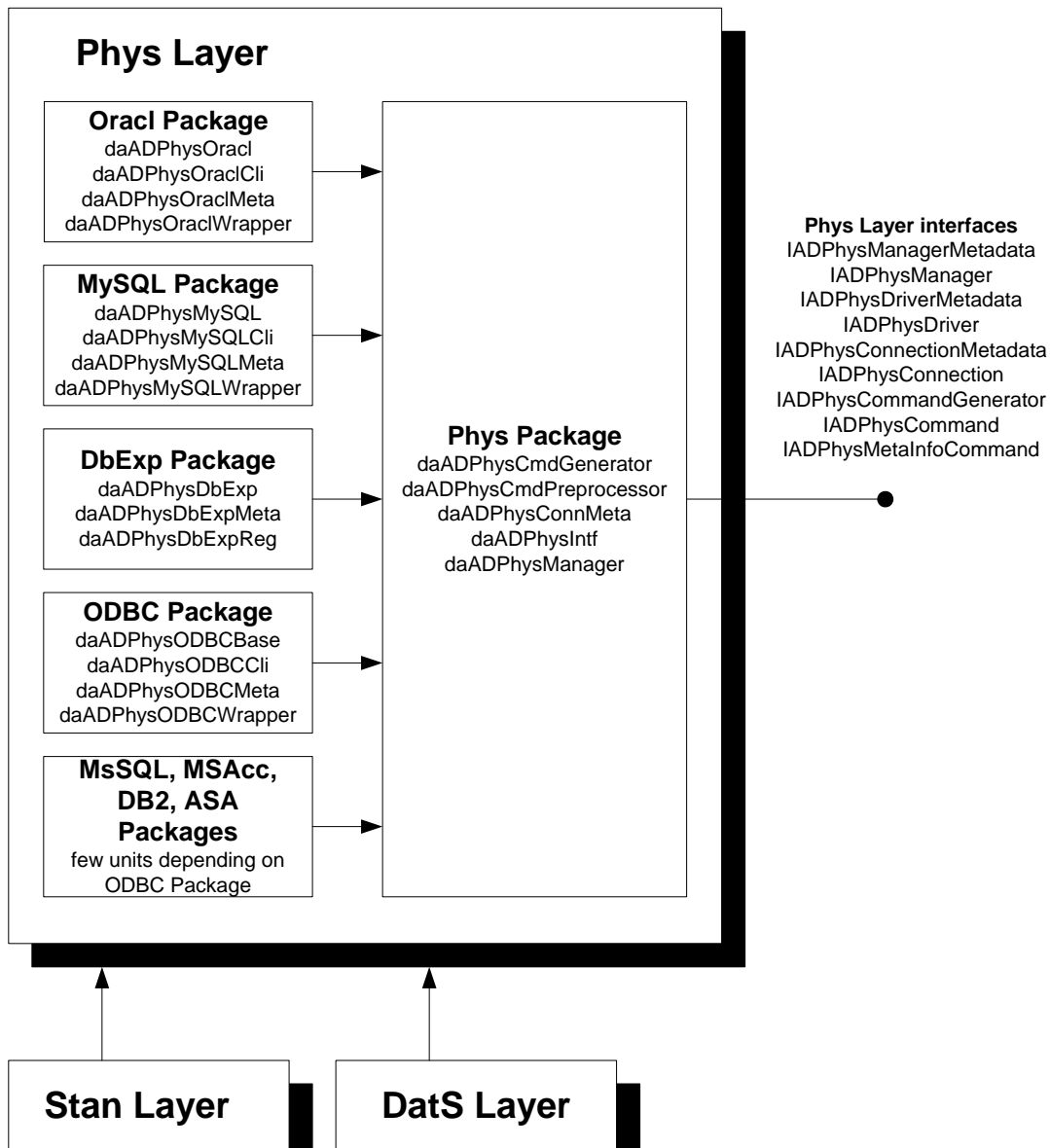


Figure 9 Phys Layer Overview

The **Phys** layer consists of a set of packages, the Phys package and few packages specific to each RDBMS / DAC. The **Phys** layer uses from **Stan** layer at most all interfaces and facilities. The relationship between **Phys** and **DatS** layers could be understood as follows:

The **Phys** layer can define structures and fetches data into **DatS** objects. **DatS** can also be used on its own. From DAC user's point of view, the **Phys** layer consists of:

- RDBMS independent drivers – DbExp, ODBC packages;
- RDBMS specific drivers – OracI, MySQL, MsAcc, MSSQL (2000), MSSQL Native Client (2005), DB2, ASA packages;

- Driver manager – Phys package.

All interfaces of **Phys** layer are exposed as COM interfaces and are defined in the **daADPhysIntf** unit. Other units, such as **daADPhysManager** from the Phys package, contain base (abstract) implementation of RDBMS dependent interfaces – **IADPhysConnection**, for example. They also contain concrete implementation for RDBMS independent interfaces – **IADPhysManager**, for example.

5.1 Phys Driver

The Phys Driver is the central building block of the **Phys** layer. It is a package implementing a set of interfaces:

- **IADPhysDriverMetadata**: Provides access to driver metadata, including supported connection definition parameters, driver description and version, etc.
- **IADPhysDriver**: Controls driver, enumerates established connection through this driver, creates **IADPhysDriverMetadata** interface, etc.
- **IADPhysConnectionMetadata**: Access to RDBMS meta data, including object lists, options supported by RDBMS, characteristics of SQL dialect, etc.
- **IADPhysConnection**: Controls the physical connection to RDBMS, handles transactions, creates physical commands, etc.
- **IADPhysCommandGenerator**: Generates data update and others command texts.
- **IADPhysCommand**: Executes RDBMS commands, fetches data, etc.
- **IADPhysMetaInfoCommand**: Fetches a list of objects from the RDBMS.

A **DriverID** identifies an AnyDAC driver, which is also the driver's package name. A driver package consists of the following units:

- **daADPhys[DriverID]Cli**: Contains RDBMS Call Level Interface declarations.
- **daADPhys[DriverID]Wrapper**: Contains a set of classes, wrapping RDBMS CLI.
- **daADPhys[DriverID]Meta**: Contains RDBMS specific meta data retrieval code.
- **daADPhys[DriverID]**: Contains driver main code.

The DbExp package does not have CLI and Wrapper units, because Borland provides CLI (DBXpress) unit, and Wrapper is not required due to the high level of interface itself. MsAcc, MSSQL, DB2 and ASA packages have only Meta and driver units, because data access to these RDBMS is implemented using generic ODBC package.

A driver itself may be linked with an AnyDAC application either:

- Statically – including driver main unit into **uses** clause of your program units.

- Dynamically – driver package is a Delphi package (BPL) and will be loaded by **Phys** layer manager when it is required.

The **Phys** layer manager controls the driver live cycle. In general, the programmer should not change the driver live cycle. Below is a driver state diagram:

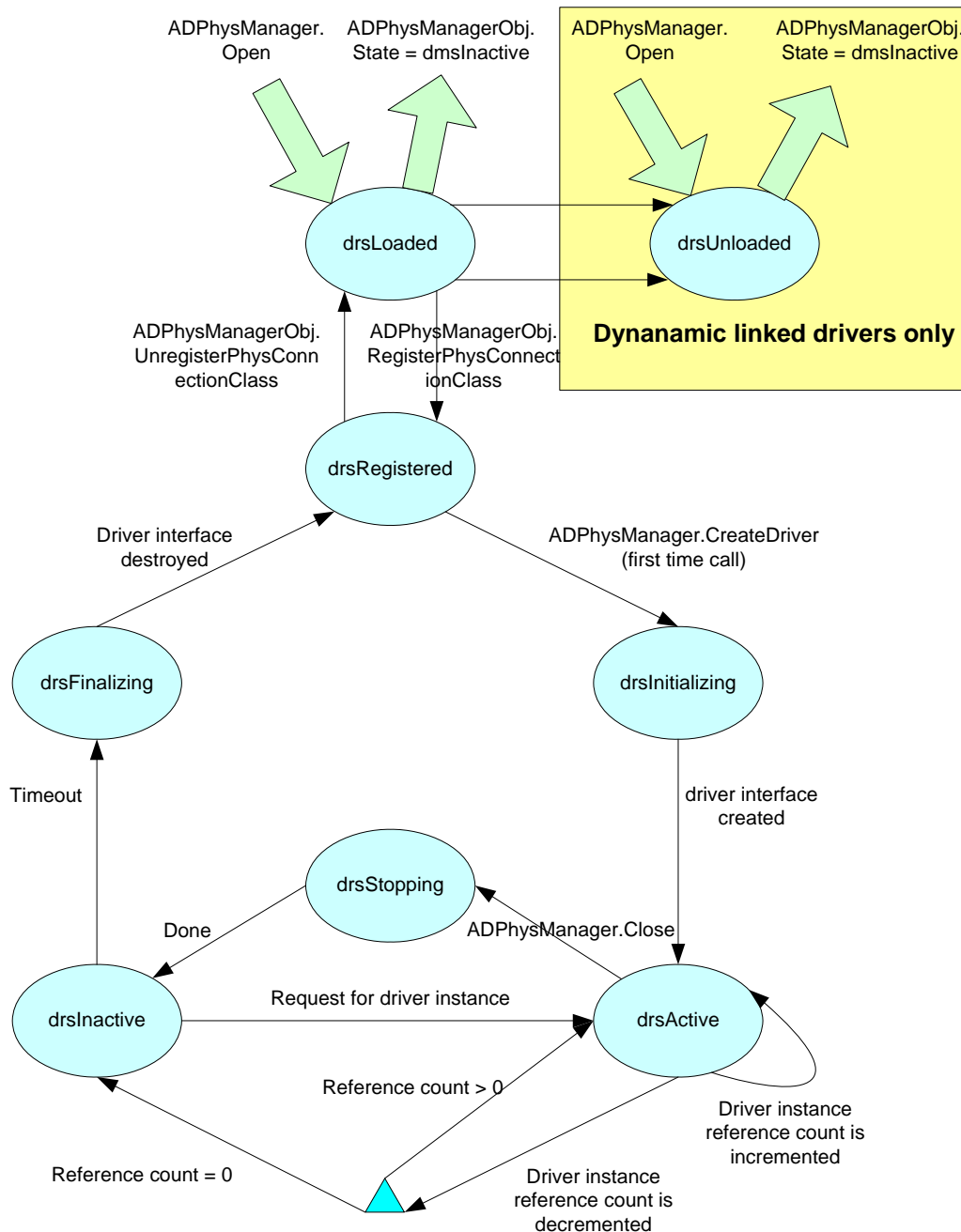


Figure 10 Phys Driver STD

Each driver's main unit contains following code:

```

initialization
  ADPhysManager();
  
```

```
ADPhysManagerObj.RegisterPhysConnectionClass(TADPhysOracleConnection);
end.
```

Here the driver package links the physical connection implementation class to the **Phys** layer manager object. It is only visible from the driver packages. To all other packages, only the manager interface is accessible. After drivers have registered a connection classes, the manager object may be activated to bring the driver in a state “*registered*”. When the first instance of any of driver interfaces is requested, the **Phys** layer manager will initialize the driver and create an interface implementation object. It is then in an “*active*” state. The driver will change to “*inactive*” as soon as all references to the interfaces are released. Then, the driver manager will timeout and, if no further interfaces are requested, the manager destroys the driver object. This will unload RDBMS client library. Then changes back to a driver state “*registered*”.

Some aspects that make AnyDAC applications safe and the programmer’s job easy are:

- A single instance of driver implementation object handles all physical connections established through the driver. All driver implementations are thread safe.
- A manager optimizes creation / destruction of the driver implementation object. The manager is also responsible for loading / unloading the dynamically linked drivers.
- The process is controlled by the manager and is isolated from driver interfaces consumers.

5.2 Phys Layer manager

The **Phys** layer manager primary goal is to control the live cycle of AnyDAC drivers. The manager creates a *root* driver interfaces – **IADPhysManagerMetadata**, **IADPhysDriver** and **IADPhysConnection**. The manager itself is represented by the top-level interface **IADPhysManager**. The **daADPhysIntf** unit has a global function **ADPhysManager**, returning a reference to the instance of **IADPhysManager** interface. The following declaration shows details:

```
IADPhysManager = interface(IInterface)
  ['{3E9B315B-F456-4175-A864-B2573C4A2101}']
  // public
  procedure CreateConnection(const AConDef: IADStanConnectionDef;
    out AConn: IADPhysConnection; AIntfRequired: Boolean = True); overload;
  procedure CreateConnection(const AConDefName: String;
    out AConn: IADPhysConnection; AIntfRequired: Boolean = True); overload;
  procedure CreateDriver(const ADriverID: String;
    out ADrv: IADPhysDriver; AIntfRequired: Boolean = True);
  procedure CreateMetadata(out AMeta: IADPhysManagerMetadata);
  procedure Open;
  procedure Close(AWait: Boolean = False);
  property DriverDefs: IADStanDefinitions read GetDriverDefs;
  property ConnectionDefs: IADStanConnectionDefs read GetConnectionDefs;
  property Options: IADStanOptions read GetOptions;
  property State: TADPhysManagerState read GetState;
end;
```

Property **State** returns manager current state. Following picture shows manager state diagram:

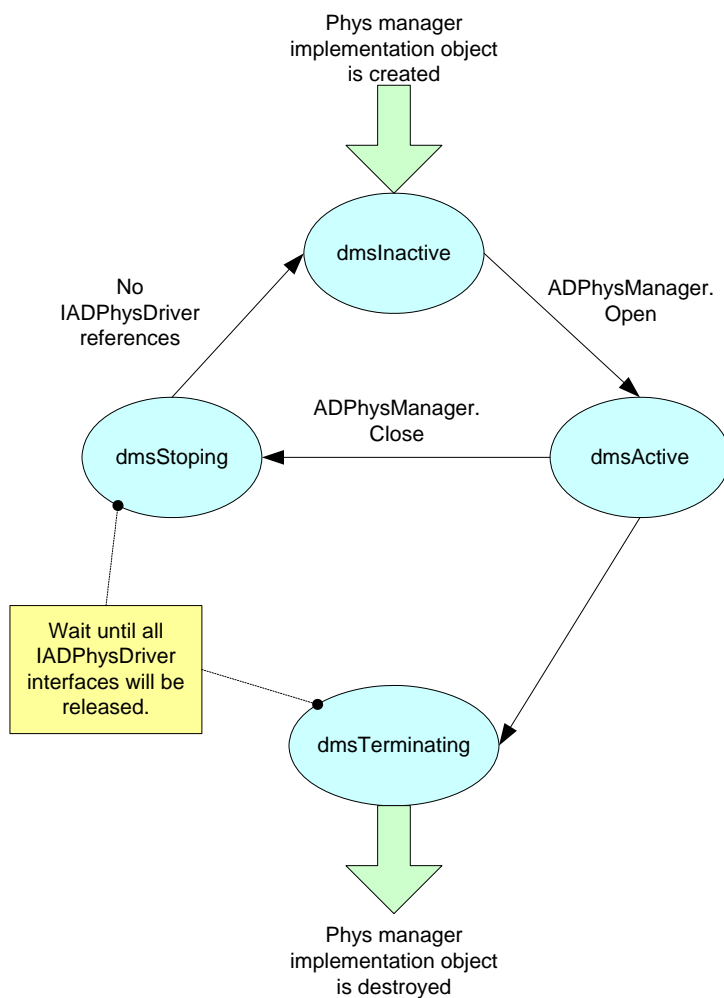


Figure 11 Phys Manager STD

Before the manager can be used and, hence, any other **Phys** layer interface, the programmer should call its **Open** method. This causes the manager to open driver definition file – **DriverDefs** and create driver control structures (not driver implementation objects). After that point, the drivers are in **drsRegistered** state.

When the **Phys** layer manager's reference counter is equal to zero, it will terminate and destroy the manager's implementation object. The method **Close** is for the programmer to close it manually. If the manager is receiving a **Close** request, it waits until all connection and layer interfaces are released and will then close. To use it again, the **Open** method has to be called again.

While **Phys** manager is open and references to connections exists, it will protect **ConnectionDefs** connection definition on which **IADPhysConnection** was created. To change the filename of a driver definition file or a connection definition file, the manager must be inactive. Worth to mention, **Phys** manager is the top level in options hierarchy. Changes to **Options** will be inherited by **IADPhysConnection** and, consequently, by **IADPhysCommand**.

5.3 Driver definition file

Driver definition file consists from sections, one per driver, in following format:

```
[<DriverID>]
BaseDriverID = <DriverID>
Package<tool ID> = <path to BPL file>
VendorHome = <vendor specific home identifier>
VendorLib = <path to RDBMS client library>
<driver specific optional parameters>
```

This structure allows registering multiple logical drivers for the same physical AnyDAC driver, if they have different optional initialization parameters. For example, in case of MySQL, driver should load MySQL client library (LIBMYSQL.DLL) of appropriate version to MySQL server.

The **DriverID** is the ID of the logical driver. **BaseDriverID** is the ID of the physical driver and is optional if physical and logical drivers are the same. Tool ID is an abbreviation of the Borland tool, used to build AnyDAC applications. Supported are Delphi 5 – **D5**, Delphi 6 – **D6**, Delphi 7 - **D7**, Delphi 2005 – **D9**, **BCB5** – C++Builder 5, **BCB6** – C++Builder 6.

The parameter **Package<tool ID>** has meaning only for dynamically loading drivers. It points to a Delphi package library containing the driver. If package library file has default name (for example, for MySQL - daADPhysMySQLD7.bpl) and it is in PATH, then this parameter may be omitted.

Parameters **VendorHome** and **VendorLib** choose one of the installed RDBMS client libraries and they values are specific to RDBMS. For example, in case of Oracle, **VendorHome** value is a name of one of Oracle homes on the computer. AnyDAC knows standard client library name for all supported DBMS. So, use **VendorLib** just if the name is not standard or library is not in PATH.

Optional parameters are specific for driver.

Returning to the example with MySQL, the following driver definition file is possible:

```
[MySQL]
PackageD6=daADPhysMySQLPackD6.bpl
PackageD7=daADPhysMySQLPackD7.bpl

[MySQL327]
BaseDriverID=MySQL
VendorLib=c:\LIBMYSQL327.DLL

[MySQL410]
BaseDriverID=MySQL
VendorLib =c:\LIBMYSQL410.DLL
```

Actually, a driver definition file is not required in the following cases:

- Drivers are linked statically to AnyDAC application.
- For each physical driver will be one logical driver.
- No specific driver parameters must be supplied.

Phys manager uses **IADStanDefinitions** interface to work with driver definition file. The **DefaultFileName** is 'ADDDrivers.ini'. The **GlobalFileName** property value is loaded from registry key HKCU\Software\da-soft\AnyDAC\DriverFile, if it exists, otherwise from HKLM\....

5.4 Phys connection

Phys connection is represented by the **IADPhysConnection** interface. The Phys connection is responsible for maintaining the connection to the RDBMS, creating commands and for transactional control. The following declaration shows details:

```
IADPhysConnection = interface (IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2105}']
    procedure CreateMetadata(out AConnMeta: IADPhysConnectionMetadata);
    procedure CreateCommandGenerator(out AGen: IADPhysCommandGenerator;
        const ACommand: IADPhysCommand = nil);
    procedure CreateCommand(out ACmd: IADPhysCommand);
    procedure CreateMetaInfoCommand(out AMetaCmd: IADPhysMetaInfoCommand);
    procedure Open;
    procedure Close;
    procedure TxBegin;
    procedure TxCommit;
    procedure TxRollback;
    // R/O
    property Driver: IADPhysDriver read GetDriver;
    property State: TADPhysConnectionState read GetState;
    property TxIsActive: Boolean read GetTxIsActive;
    property ConnectionDef: IADStanConnectionDef read GetConnectionDef;
    property CommandCount: Integer read GetCommandCount;
    property Commands[AIndex: Integer]: IADPhysCommand read GetCommands;
    property Messages: EADDBEngineException read GetMessages;
    // R/W
    property Options: IADStanOptions read GetOptions write SetOptions;
    property TxOptions: TADStanTxOptions read GetTxOptions write SetTxOptions;
    property ErrorHandler: IADStanErrorHandler read GetErrorHandler write SetErrorHandler;
    property Login: IADGUIxLoginDialog read GetLogin write SetLogin;
    property LoginPrompt: Boolean read GetLoginPrompt write SetLoginPrompt;
end;
```

To create a connection interface, programmer should call one of Phys manager **CreateConnection** methods. One method gets a connection definition interface as parameter, another the connection string and internally builds the connection definition from this string. **IADPhysConnection** remains associated with Phys driver and connection definition until end of live. The Phys manager protects the connection definitions from being changed, if they have associated connection.

Open establishes a connection with the RDBMS using parameters from the connection definition **ConnectionDef**. If the **Login** property is initialised by login dialog interface, it will be called from the **Open** method. Calling **Close** closes the connection to the RDBMS. It is called automatically, when there are no

more references to connection interface and the connection object is destroyed. Following is an example of **IADPhysConnection** creation:

```
// Open manager
ADPhysManager.ConnectionDefs.Storage.FileName := '${ADHOME}\DB\ADConnectionDefs.ini';
ADPhysManager.Open;
// Create connection using existing connection definition
ADPhysManager.CreateConnection('Access_Demo', oConnIntf);
oConnIntf.Open;
// will automatically close connection and destroy it
oConnIntf := nil;
// Create connection using connection string
ADPhysManager.CreateConnection('DriverID=MSAcc;Database=${ADHOME}\DB\Data\ADDemo.mdb', oConnIntf);
oConnIntf.Open;
```

The method **TxBegin** starts a new transaction, **TxCommit** – commits a transaction and **TxRollback** – rolls back the current transaction. AnyDAC supports nested transactions. Depending on the RDBMS, it is supported either by RDBMS or emulated by AnyDAC using save points. AnyDAC does not support multiple parallel transactions, like Borland Interbase does. For example, in case of Oracle we can write code:

```
// start TX
oConnIntf.TxBegin;
try
    .....
    // emulating start of nested TX -> set savepoint
    oConnIntf.TxBegin;
    try
        .....
        // emulating commit of nested TX -> nothing todo
        oConnIntf.TxCommit;
    except
        // emulating rollback of nested TX -> rollback to savepoint
        oConnIntf.TxRollback;
        raise;
    end;
    // commit TX
    oConnIntf.TxCommit;
except
    // rollback TX
    oConnIntf.TxRollback;
    raise;
end;
```

CreateCommand creates a Phys command and associates it with the Phys connection. Property **Commands** lists commands associated with this connection. Property **ErrorHandler** allows to assign an exception handler, which will intercept exceptions raised from Phys connection implementation methods.

5.5 Phys command

Phys command is represented by the **IADPhysCommand** COM interface. It is responsible for executing RDBMS commands, exchanging parameter values and fetching row sets from RDBMS. The following declaration shows details:

```
IADPhysCommand = interface (IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2107}']
    // public
    procedure AbortJob(AWait: Boolean = False);
    procedure Close;
    procedure Disconnect;
    function Define(ADatSManager: TADDatSManager; const ATabMaps: IADPhysTableMappings = nil;
        AMetaInfoMergeMode: TADPhysMetaInfoMergeMode = mmReset): TADDatSTable; overload;
    function Define(ATable: TADDatSTable; const ATabMaps: IADPhysTableMappings = nil;
        AMetaInfoMergeMode: TADPhysMetaInfoMergeMode = mmReset): TADDatSTable; overload;
    procedure Execute(ATimes: Integer = 0; AOffset: Integer = 0);
    function Fetch(ATable: TADDatSTable; AAll: Boolean = True): Integer; overload;
    procedure Fetch(ADatSManager: TADDatSManager; const ATabMaps: IADPhysTableMappings = nil;
        AMetaInfoMergeMode: TADPhysMetaInfoMergeMode = mmReset); overload;
    procedure Open;
    procedure Prepare(const ACommandText: String = '');
    procedure Unprepare;
    // R/O
    property Connection: IADPhysConnection read GetConnection;
    property State: TADPhysCommandState read GetState;
    property RowsAffected: Integer read GetRowsAffected;
    property ErrorAction: TADErrorAction read GetErrorAction;
    // R/W
    property Options: IADStanOptions read GetOptions write SetOptions;
    property SchemaName: String read GetSchemaName write SetSchemaName;
    property CatalogName: String read GetCatalogName write SetCatalogName;
    property BaseObjectName: String read GetBaseObjectName write SetBaseObjectName;
    property CommandKind: TADPhysCommandKind read GetCommandKind write SetCommandKind;
    property CommandText: String read GetCommandText write SetCommandText;
    property Params: TADParams read GetParams;
    property Macros: TADMacros read GetMacros;
    property ParamBindMode: TADPhysParamBindMode read GetParamBindMode write SetParamBindMode;
    property Overload: Word read GetOverload write SetOverload;
    property NextRecordSet: Boolean read GetNextRecordSet write SetNextRecordSet;
    property SourceObjectName: String read GetSourceObjectName write SetSourceObjectName;
    property ErrorHandler: IADStanErrorHandler read GetErrorHandler write SetErrorHandler;
    property AsyncHandler: IADStanAsyncHandler read GetAsyncHandler write SetAsyncHandler;
    property MappingHandler: IADPhysMappingHandler read GetMappingHandler write SetMappingHandler;
end;
```

The following diagram shows calls flow for **IADPhysCommand**. Then it will be discussed.

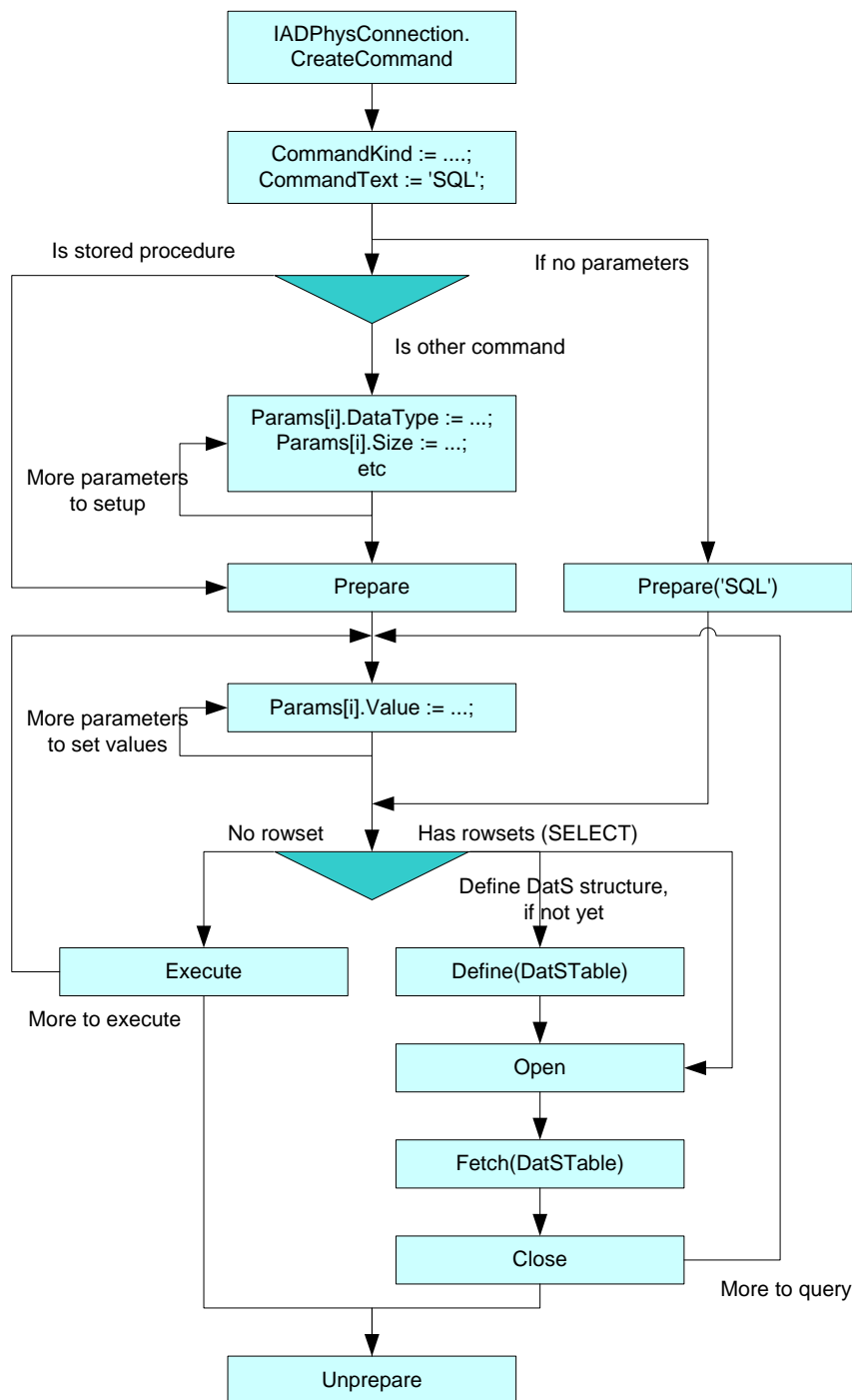


Figure 12 Phys Command Call Flow Diagram

The programmer should set **CommandText** and **CommandKind** properties to define command. Explicit setting of command kind is only required for stored procedures and for non-SQL data sources.

Command text may have parameters and macros. If the command is a stored procedure, the command cannot contain macros and the parameter list is automatically filled after **Prepare** has been called. For all

other command kinds, parameter list and macros list will be accessible after command text is assigned to command. Macro features are explained in the next chapter.

The **Prepare** method, internally performs macro expansion, submits command text to RDBMS and binds the client side parameter and row set buffers to the RDBMS command control structures. Therefore, parameter types and macro values cannot be changed after they have been bound to the control structures.

Following is an example of defining and preparing command:

```
oConnIntf.CreateCommand(oCommIntf);
with oCommIntf do begin
  CommandText := 'select * from !tab where ID = :ID';
  // Setting up the macro 'TAB'
  with Macros [0] do begin
    Value := 'Employees';
    DataType := mdIdentifier;
  end;
  // Setting up the param 'ID'
  with Params [0] do begin
    Value := 1000;
    DataType := ftInteger;
  end;
  Prepare;
```

The programmer may assign parameter values and choose a method for command execution. Different methods should be used whether the command returns a row set or not. The classical command returning a row set is the SQL SELECT command. If the command returns a row set, the methods **Define**, **Open**, **Fetch** and **Close** should be used. If the command does not return a row set, we use the method **Execute**.

The method **Define** defines the structure of **TADDatSTable** (columns, primary key, nested tables). Calling **Define** can be omitted, if the **DatS** table structure is already defined. It is worth to mention, before defining the structure AnyDAC will call the table method **Reset**. And it will destroy all table sub objects, like columns, views and constraints.

Calling the **Open** method actually executes the command and initiates retrieval of a row set. AnyDAC supports different modes for fetching data, such as *fetch-all*, *fetch-incrementally*, and a few others. That is controlled by **Options.FetchOptions**. Fetch-all will fetch all rows from a row set at first **Fetch** call. If the programmer uses the incremental mode, the **Fetch** method call fetches the number of rows specified in **Options.FetchOptions.RowsetSize** and returns actual number of rows fetched. AnyDAC will automatically call the **Close** method after all rows are fetched or the programmer can call it explicitly.

Following code proceeds previous example:

```
Define(oTab);
Open;
Fetch(oTab, True);
end;

PrintRows(oTab, Console.Lines);
```

If the format options **Options.FormatOptions** are set, the command will automatically translate the data format from server to client and vice-versa. Setting **Options.ResourceOptions**, the programmer controls server and client resources usage. Setting the option **AsyncCmdMode** affects **Open**, **Execute** and **Fetch** methods and controls the asynchronous execution mode. With the **AsyncCmdTimeout** option the programmer can set a time period after which a command will stop execution and AnyDAC will raise an exception.

The command's **AbortJob** method cancels a method execution. Calling **AbortJob** has no effect on drivers, which do not support that feature. If a command is not needed anymore, **Unprepare** can be called to release the resources used on the database server. Also, if **Options.ResourceOptions.Disconnectable** is True and application has too many prepared commands, AnyDAC will automatically unprepare least used commands.

The property **ErrorHandler** allows assigning an exception handler, which will intercept exceptions raised from methods of the Phys command implementation. And the property **AsyncHandler** allows assigning a handler, which will be called after asynchronous command execution is finished.

5.5.1 Macro processing

Setting **CommandText** will fill the **Macros** collection property. At command **Prepare** call, the AnyDAC *macro processor* transforms command text into a form understood by the RDBMS. This means, the macros are not visible to the RDBMS. Setting the property **Options.ResourceOptions.MacroCreate** turns macro processing on or off.

AnyDAC supports two kinds of macro instructions:

- Substitution variables: They allow substitution to put parameters in command text. This is to extend the use of parameters. For example, to parameterise a table name in FROM clauses or column names in SELECT clauses substitution variables can be used but parameters are of no use.
- Escape sequences: They allow writing RDBMS independent SQL commands.

5.5.1.1 Substitution variables

Substitution variables starts with a '!' or '&' symbol and is followed by a macro variable name. For example:

```
SELECT * FROM &SomeTab
```

The symbols have the following meaning:

- '!' - "string" substitution mode. Macro value will be substituted "as is", directly into the command text without any transformations.
- '&' – "SQL" substitution mode. Macro value will be substituted depending on the macro data type, using target RDBMS syntax rules.

5.5.1.2 Escape sequences

AnyDAC has 5 kinds of escape sequences:

- Allowing constant substitution.
- Allowing identifier substitution.
- Conditional substitution.
- LIKE operator escape sequence.
- Scalar functions.

Constant substitution escape sequences allow writing constants in command text, independent on RDBMS syntax and regional settings. Following describes escape sequences expansion to RDBMS syntax:

{e <number>}	Number constant of numeric format. For example: {e 123.7} -> 123,7 on MSAccess
{d <date>}	Date constant. Here <date> must be specified in "yyyy-mm-dd" format. For example: {d 2004-08-30} -> TO_DATE('2004-08-30', 'yyyy-mm-dd') on Oracle
{t <time>}	Time constant. Here <time> must be specified in "hh24:mi:ss" format. For example: {t 14:30:00} -> CONVERT(DATETIME, '14:30:00', 114) on MSSQL
{dt <date & time>}	Date and time constant. Here <date & time> must be in format as above.
{l <boolean>}	Boolean constant. Here <boolean> is False or True. If RDBMS supports Boolean data type, then sequence expands to that type constant, otherwise to numeric values 0 or 1.

Identifier substitution escape sequence allows abstracting from RDBMS specific identifier quoting rules. The syntax is:

{id <identifier name>}	Expands to RDBMS specific quoted identifier syntax. For example: {id Order Details} -> "Order Details" on Oracle.
-------------------------------------	---

Conditional substitution escape sequence allows substitute text into command, depending on either RDBMS application is connected to, either on macro variable value. Escape sequence syntax is:

{if (X1, Y1, ..., XN, YN, YN, 1) 1}	Here Xi is either:
--	--------------------

YN+1) }	<ul style="list-style-type: none"> AnyDAC RDBMS kind identifier. So, if application is connected to this RDBMS, then Yi text will be substituted into command. Macro variable. If it value is not empty, then Yi text will be substituted into command. <p>If neither of conditions is meted and YN+1 text is specified, then it will be substituted into command. For example: {if (OracI, TO_CHAR, MSSQL, CONVERT)} -> TO_CHAR on Oracle. {if (&v1, Me, &v2, You, We)} -> You if &v1 has empty value and &v2 nonempty one.</p>
---------	--

The escape functions syntax and set follows close to the rules of ODBC escape functions. In [Appendix 2](#) AnyDAC escape functions are listed and it is explained how to use them. Escape functions have the following syntax:

```
{fn <function name>(<arguments>)}
```

For example:

```
SELECT * FROM MyTab WHERE Year = {fn YEAR({fn NOW()})}
```

In this example, the RDBMS searches for records where column named 'Year' equals to current year. Depending on the target RDBMS, this command is translated into the correct SQL dialect for the RDBMS. If the target database system is Oracle, the resulting command text would be:

```
SELECT * FROM MyTab WHERE Year = TO_NUMBER(TO_CHAR(SYSDATE, 'YYYY'))
```

5.5.2 Special character processing

To transmit special characters - '!', '&', ':', '{' or '}' – to DBMS, you will need:

- Either precede this character by '^' character. For example, '^:'.
- Either double special character. For example, '{{'.

Otherwise special characters will be treated as macro command, excluding following cases:

- PL/SQL assignment operator is detected by AnyDAC and will not be treated as parameter marker.
- TSQL label is detected by AnyDAC and will not be treated as parameter marker.

5.5.3 Asynchronous execution

Phys command supports asynchronous execution of methods: **Open**, **Execute** and **Fetch**. The chapter “Async operation execution” describes asynchronous execution modes. The property **Options.ResourceOptions.AsyncCmdMode** controls execution mode of listed methods. And property **AsyncCmdTimeout** defines maximum allowed time to perform method. If execution will take more time, then AnyDAC will raise exception and method execution will be aborted. Also programmer may cancel execution of call using command's **AbortJob** method.

If driver does not supports command cancellation, then execution will be continued after **AbortJob** method is called in following cases:

- **Open**, **Execute** methods of MySQL driver.
- **Open**, **Execute** methods of MSAccess driver.

After asynchronous execution of command is finished, AnyDAC calls the method **HandleFinished** of **IADPhysCommand.AsyncHandler** handler, if it is assigned. Parameter **AState** receives an asynchronous method completion status:

- **asFinished** - if execution finished successfully.
- **asAborted** – if execution was aborted.
- **asFailed** – if execution failed due to exception was raised.
- **asExpired** – if execution was timed out.

The implementation of **IADStanAsyncHandler** interface:

- in case of **IADPhysCommand** is the programmer responsibility.
- in case of **TADCustomCommand** component is already done by AnyDAC developers. The events **AfterOpen**, **AfterExecute**, **AfterFetch** are fired after corresponding action is completed.

For example, following code will run command in asynchronous mode:

```
type
  TMyAsyncHandler = class(TInterfacedObject, IADStanAsyncHandler)
    procedure HandleFinished(const AInitiator: IADStanObject;
      AState: TADStanAsyncState);
  end;

procedure TMyAsyncHandler.HandleFinished(const AInitiator: IADStanObject;
  AState: TADStanAsyncState);
const
  StateNames: array[TADStanAsyncState] of String = ('asInactive', 'asExecuting',
    'asFinished', 'asFailed', 'asAborted', 'asExpired');
begin
  writeln('    The HandleFinished is called - ' + StateNames[AState]);
end;
```

```

var
  oCmd: IADPhysCommand;
.....
oCmd.CommandText := 'update SalesHistory set Price = Price + Tax, Tax = 0';
oCmd.Options.ResourceOptions.AsyncCmdMode := amAsync;
oCmd.Options.ResourceOptions.AsyncCmdTimeout := 10000;
oCmd.AsyncHandler := TMyAsyncHandler.Create;
oCmd.Prepare;
oCmd.Execute;
// 1) here we will not wait for command to be finished
// 2) if command will execute more than 10 sec, it will be canceled
// 3) in any case TMyAsyncHandler.HandleFinished will be called

```

5.5.4 Batch command execution

In general, the idea of *batch command execution* is to submit a single RDBMS command with an array of parameters. All parameter values are arrays of the same size. And we can ask the RDBMS to execute a command once for each array item. This technique reduces the amount of communication between RDBMS and client enormously and speeds up the execution. Following picture shows that:

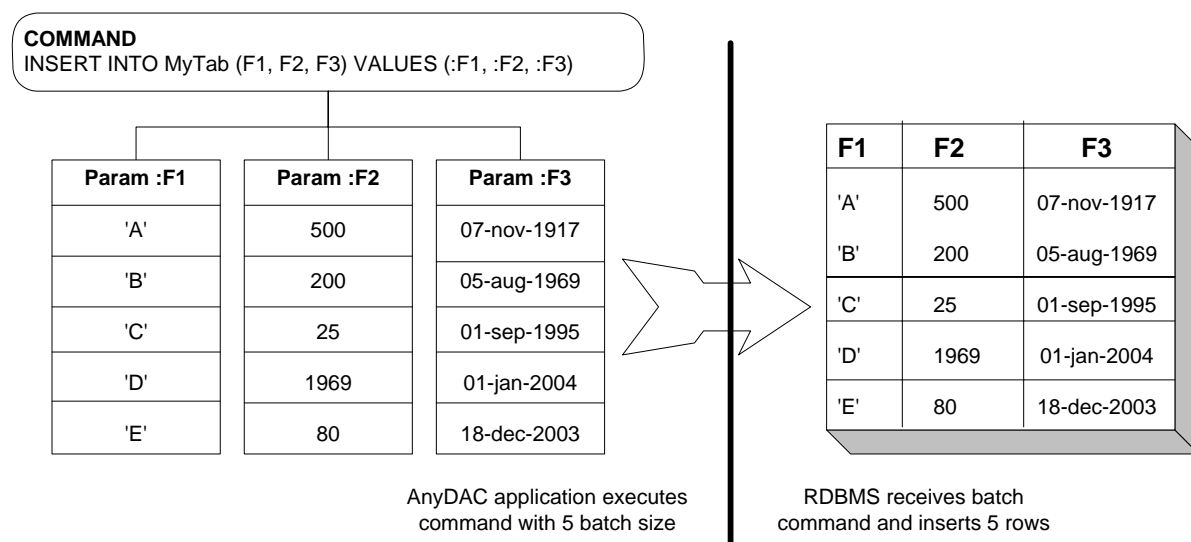


Figure 13 Batch Execution Mode

AnyDAC supports batch command execution mode using the native RDBMS capabilities, if the driver supports batch mode, or emulating batch execution, if the driver does not support it. At the moment, the Oracle, MSSQL and MySQL drivers support native batch execution mode.

Programmer should use the following **IADPhysCommand** method to do batch processing:

```

procedure Execute(ATimes: Integer = 0; AOffset: Integer = 0);

```

Here, **ATimes** defines the size of a batch job. **AOffset** is index of first item in the batch job. If an error occurs, such as a violation of constraints, then the error in the job will be handled as follows:

- If **ErrorHandler** is not assigned, then **RowsAffected** property will be equal to amount of rows successfully processed before the row failed and **ErrorAction** property value will be **eaFail**.
- If **ErrorHandler** is assigned, then **ErrorHandler.HandleException** will receive a special exception of class **EADPhysArrayExecuteError**. It has a property **Action**, which allows returning the kind of action. For example, it may be **eaRetry** – repeat execution from failed row.

Following code shows example, how to execute batch command:

```
var
  oCmd: IADPhysCommand;
.....
with oCmd do begin
  CommandText := 'insert into Customers (ID, Name) values (: ID, :Name)';
  // Set up parameters
  Params[0].DataType := ftInteger;
  Params[1].DataType := ftString;
  Params[1].Size := 40;
  // Set up parameters' array size
  Params.ArraySize := 10000;
  // Prepare command
  Prepare;
  // Set parameter values
  for i := 0 to 10000 - 1 do begin
    Params[0].AsIntegers[i] := i;
    Params[1].AsStrings[i] := 'Somebody ' + IntToStr(i);
  end;
  // Execute batch
  Execute(10000, 0);
end;
```

It is important to properly setup parameters, including setting property **Size** for string parameters. In case of Oracle, for example, AnyDAC will allocate ~1400 bytes for each string parameter value, if this parameter property **Size** is not assigned. So, for 10,000 of values will be allocated 14Mb buffer and due to some Oracle issues, **Execute** call will hang up.

5.5.5 Stored procedure execution

The important difference between executing stored procedures and other commands is the automatic setup of parameters and its types. AnyDAC handle all on its own. It does not support manual definition of parameters. In case of a stored procedure, the command text could look like this:

[Catalogue name.][Schema name.][Package name.][Stored procedure name]

Also, each part of a name can be assigned to the corresponding properties:

- Catalogue name to **CatalogName**.
- Schema name to **SchemaName**.

- Package name to **BaseObjectName**.
- Stored procedure name to **CommandText**.

CommandKind must be one of following:

- **skStoredProc**. If set, AnyDAC will automatically detect if the procedure returns a row set or not and will set **CommandKind** to either **skStoredProcNoCrs** or **skStoredProcWithCrs**.
- **skStoredProcNoCrs**. Procedure does not return row sets. Use **Execute** method.
- **skStoredProcWithCrs**. Procedure returns row sets. Use **Open** method.

6 DApt Layer

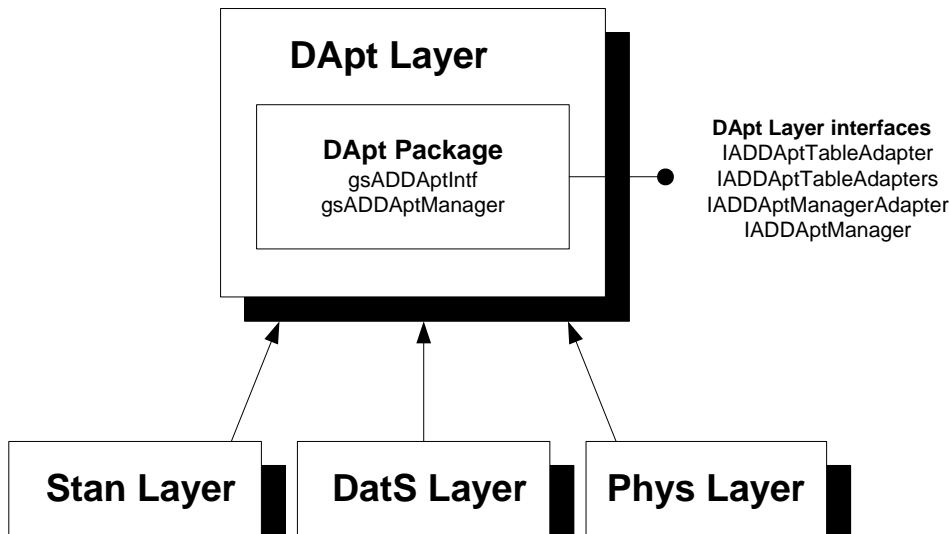


Figure 14 DApt Layer Overview

DApt layer consists from a single DApt package. **DApt** layer uses following interfaces and facilities from the **Stan** layer:

- Error handling - **IADStanErrorHandler** interface and **EADException** class.
- Option handling – **IADStanOptions** interface and option classes.
- Parameters – **TADParams**, **TADParam** classes.
- Monitor facilities – **IADMoniClient** interface.

And from **GUIx** layer is used AnyDAC wait cursor facility – **IADGUIxWaitCursor** interface. Relation of **DApt**, **Phys** and **DatS** layers better will be described as – **DApt** layer is integrator of **Phys** and **DatS** layers. **DApt** layer uses from **Phys** and **DatS** layers almost all interfaces and facilities.

All interfaces of layer are exposed as COM interfaces and are defined in **daADDaptIntf** unit.

daADDaptManager unit contains implementation.

Main functions of **DApt** layer are:

- Mapping of **Phys** layer result set structure into **DatS** layer structures.
- Automatic generation or usage of existing **Phys** layer commands to post updates from **DatS** layer objects to RDBMS.
- Reconciliation of errors after posting updates.

All that functionality is optional and application programmer may implement that. But **DApt** layer brings new level of flexibility and solves standard tasks of application programmer. Following picture shows object model of **DApt** layer:

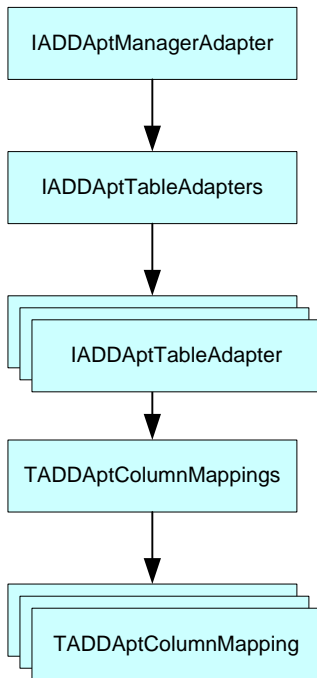


Figure 15 DApt Layer Object Model

6.1 Structure Mapping

One application may need to work as with different RDBMS's as with different versions of DB schema. For example, in MySQL DB we have table:

`Customers`
ID
Name
Address

In Oracle DB we have table:

CUSTOMER
CustID
Name
Addr

To achieve structure mapping of **Phys** layer result sets into **DatS** layer structures, AnyDAC software should be able:

- Map **TADDatSTable** structure (column list) to physical table structure in DB.
- Map **TADDatSTable** name to physical (source) table name in DB.
- To verify / alter existing **TADDatSTable** structure according to physical table structure in DB.

TADDaptColumnMappings represents a collection of **TADDaptColumnMapping** objects.

TADDaptColumnMapping class maps SELECT list item of **IADPhysCommand** or physical DB column to single **TADDatSColumn**. Following declaration shows details:

```
TADDaptColumnMapping = class(TCollectionItem)
public
    property DatSColumn: TADDatSColumn read GetDatSColumn;
published
    property SourceColumnName: String read FSourceColumnName write SetSourceColumnName;
    property SourceColumnID: Integer read FSourceColumnID write SetSourceColumnID default -1;
    property UpdateColumnName: String read GetUpdateColumnName write FUpdateColumnName;
    property DatSColumnName: String read GetDatSColumnName write FDatSColumnName;
end;
```

SourceColumnName is a name of physical column in SELECT list item. **UpdateColumnName** is a name of physical column in DB table, which will receive updates from **DatS** column. If **UpdateColumnName** property value is empty, then **SourceColumnName** will be used. **DatSColumnName** is a name of **DatS** column. If **DatSColumnName** property is empty, then **SourceColumnName** will be used. **DatSColumn** property returns column object named **DatSColumnName**.

Interface **IADDaptTableAdapters** represents a collection of **IADDaptTableAdapter** interfaces. It maps **Phys** command result set or physical DB table to single **TADDatSTable**. Following declaration shows details (just mapping related properties):

```
IADDaptTableAdapter = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2304}']
    // public
    .....
    property SourceRecordSetID: Integer read GetSourceRecordSetID write SetSourceRecordSetID;
    property SourceRecordSetName: String read GetSourceRecordSetName write SetSourceRecordSetName;
    property UpdateTableName: String read GetUpdateTableName write SetUpdateTableName;
    property DatSTableName: String read GetDatSTableName write SetDatSTableName;
    property DatSTable: TADDatSTable read GetDatSTable write SetDatSTable;
    property ColumnMappings: TADDaptColumnMappings read GetColumnMappings;
    .....
end;
```

SourceRecordSetName is a name of **Phys** layer command result set. In place of **SourceRecordSetName** programmer can specify **SourceRecordSetID**. **UpdateTableName** is a name of physical DB table, which will receive updates from **DatS** table. If **UpdateTableName** property value is empty, then **SourceRecordSetName** will be used. **DatSTableName** is a name of **DatS** table. If **DatSTableName**

property is empty, then **SourceRecordSetName** will be used. **DatSTable** property returns table object named **DatSTableName**. **ColumnMappings** property returns column-mapping collection.

Following example shows how to map table from MySQL DB to the same structure as it is in Oracle DB:

```
// SrcName: Customers -> DatSName: Customer
oAdapt := oSchAdapt.TableAdapters.Add('Customers', 'Customer');
// SrcName: ID -> DatSName: CustID
oAdapt.ColumnMappings.Add('ID', 'CustID');
// SrcName: Name -> DatSName: Name
oAdapt.ColumnMappings.Add('Name');
// SrcName: Address -> DatSName: Addr
oAdapt.ColumnMappings.Add('Address', 'Addr');
```

6.2 Posting updates to DB

IADDaptTableAdapter, **IADDaptTableAdapters** and **IADDaptSchemaAdapter** interfaces are responsible for this functionality. It is possible to create as standalone **DatS** table adapter as **DatS** manager schema adapter.

6.2.1 DatS Table Adapter

IADDaptTableAdapter is responsible for posting updates from **DatS** row to DB. Table adapter handles rows from single **DatS** table, pointed by **DatSTable** property. To each kind of **DatS** row update, adapter links specific **Phys** layer command. Row update kind is determined using **TADDatSRow.RowState** property value:

- **rsInserted** – row is inserted. By default, will be performed INSERT SQL command.
- **rsDeleted** – row is deleted. By default, will be performed DELETE SQL command.
- **rsModified** – row is modified. By default, will be performed UPDATE SQL command.
- **rsUnchanged** – row is unchanged. Nothing to do.

To post updates following row versions will be used:

- **rvOriginal** – original row version, as row was fetched from DB or as it is after **AcceptChanges** or **RejectChanges** last call. Adapter use this version to determine which row in DB to update or delete.
- **rvCurrent** – current row version, after row was modified. Adapter use this version to determine new column values for inserted or modified rows.

If specific command is not assigned, then adapter will generate it automatically. For some kinds of updates adapter will perform few DB physical commands. For example, in case of modified row and pessimistic locking mode, following sequence of commands will be performed:

- Lock row. In case of MSSQL it will be **SELECT ... WITH (ROWLOCK,UPDLOCK) ...** SQL command.

- Update row – SQL **UPDATE** command.

Other example - in case of appended row with auto incremental fields following sequence of commands will be performed:

- Append row – SQL **INSERT** command.
- Reread auto incremental field value - **SELECT @@IDENTITY** command.

Following declaration shows details:

```
IADDAptTableAdapter = interface(IIInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2304}']
    // public
    .....
    function Update(AMaxErrors: Integer = -1): Integer; overload;
    function Reconcile: Boolean;
    procedure Reset;

    procedure Update(ARow: TADDatSRow; var AAction: TADErrorAction;
        AUpdRowOptions: TADPhysUpdateRowOptions = [];
        AForceRequest: TADPhysRequest = arFromRow); overload;
    procedure Lock(ARow: TADDatSRow; var AAction: TADErrorAction;
        AUpdRowOptions: TADPhysUpdateRowOptions = []);
    procedure UnLock(ARow: TADDatSRow; var AAction: TADErrorAction;
        AUpdRowOptions: TADPhysUpdateRowOptions = []);

    property SelectCommand: IADPhysCommand read GetSelectCommand write SetSelectCommand;
    property InsertCommand: IADPhysCommand read GetInsertCommand write SetInsertCommand;
    property UpdateCommand: IADPhysCommand read GetUpdateCommand write SetUpdateCommand;
    property DeleteCommand: IADPhysCommand read GetDeleteCommand write SetDeleteCommand;
    property LockCommand: IADPhysCommand read GetLockCommand write SetLockCommand;
    property UnLockCommand: IADPhysCommand read GetUnLockCommand write SetUnLockCommand;
    property FetchRowCommand: IADPhysCommand read GetFetchRowCommand write SetFetchRowCommand;

    property Options: IADStanOptions read GetOptions;
    property ErrorHandler: IADStanErrorHandler read GetErrorHandler write SetErrorHandler;
    property UpdateHandler: IADDAptUpdateHandler read GetUpdateHandler
        write SetUpdateHandler;
end;
```

Properties **XXXXCommand** references to commands to use to post updates to DB (Insert, Update, Delete, Lock, UnLock), refresh single row (FetchRow) and fetch all row set (Select). **Update (AMaxErrors)** method post updates from **DatS** table updates journal (**TADDatSTable.Updates**) to DB. **Reconcile** method allows reconciling errors after posting updates. Other methods with first **ARow** parameter operate on single specified row.

6.2.2 DatS Manager Adapter

Collection of table adapters – **IADDAptTableAdapters** – helps to match rows from **DatS** tables to table adapters. So, row changes from **TADDatSManager** updates journal may be posted to DB in order rows was

changed in application. **IADDaptSchemaAdapter** interface controls this process, delegating specific row handling to appropriate table adapter. Following declaration shows details:

```
IADDaptSchemaAdapter = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2306}']
    function Update(AMaxErrors: Integer = -1): Integer;
    function Reconcile: Boolean;

    property DatSManager: TADDatSManager read GetDatSManager write SetDatSManager;
    property TableAdapters: IADDaptTableAdapters read GetTableAdapters;

    property ErrorHandler: IADStanErrorHandler read GetErrorHandler write SetErrorHandler;
    property UpdateHandler: IADDaptUpdateHandler read GetUpdateHandler
        write SetUpdateHandler;
end;
```

Schema adapter will handle **DatS** manager, pointed by **DatSManager** property. **TableAdapters** property is a collection of **DatS** table adapters. Each of them will post updates belonging to associated **DatS** table to DB. Method **Update** post updates from **DatS** manager updates journal (**TADDatSManager.Updates**) to DB. Method **Reconcile** allows reconciling errors after posting updates.

6.2.3 Concurrency Control

DApt layer uses DB row locking for DB concurrency access control. AnyDAC implements 2 kinds of row locking:

- **Optimistic.** Optimistic row locking schema does not explicitly lock row, but expecting only single user will update row at a time. At posting updates, AnyDAC verifies row was not changed after fetching, using original column values.
- **Pessimistic.** Pessimistic row locking schema explicitly lock row, when it is required. This mode depends on RDBMS locking implementation. For example, MS Access does not support explicit locking.

UpdateOptions has properties controlling locking mode:

- **LockMode.** Defines kind of row locking. **ImPessimistic** – pessimistic mode, **ImOptimistic** – optimistic mode, **ImRely** – no AnyDAC locking activity at all.
- **LockPoint.** Defines a moment when to lock a row. **IpImmediate** – immediately before editing row, **IpDeferred** – deferred until row changes will be posted to DB.
- **LockWait.** If True, then AnyDAC will wait until row lock acquired by other user is released.
- **UpdateMode.** This property controls optimistic locking mode. It is similar to standard Delphi **UpdateMode** property.

Explicitly locked rows will be unlocked at end of transaction where row was locked.

6.2.4 Row Refreshing

Some RDBMS may modify rows after posting changes to DB. Classic examples of that are:

- Auto incremental fields. IDENTITY columns in MSSQL, AUTO_INCREMENT columns in MySQL and COUNTER columns in MSAccess. When new record is inserted into DB, RDBMS will automatically assign unique auto incremented value to column.
- DEFAULT field option. When new record is inserted into DB and field with DEFAULT option is not in INSERT command, then RDBMS will automatically assign specified in DEFAULT option value to column.
- Triggers. When record change is posted to DB, trigger defined for updating table may modify row.

In these cases **DApt** layer can fetch column values modified by RDBMS to client. Following properties controls that:

- **TADatSColumn.Attributes**. Property is a set of enumerated values, which may contain following attributes: **caAutoInc** – column is auto incremental, **caROWID** – column is RDBMS row identifier, **caDefault** – column has default value.
- **TADatSColumn.Options**. Property is a set of enumerated values, which may contain following options: **coAfterInsChanged** – column value is changed by RDBMS after row is inserted, **coAfterUpdChanged** – column value is changed by RDBMS after row is updated.
- **TADUpdateOptions.AutoRefresh**. If property value is True (default value), then **DApt** layer will automatically refresh changed by RDBMS columns after posting updates.

Attributes are column physical attributes (how it is defined in DB), which DB driver will setup automatically in most cases. Programmer may change options, although AnyDAC automatically sets column options, basing on their attributes.

Using AnyDAC, following classic task may be solved without algorithm coding, but just setting up appropriate objects properties. Let say we have 2 MSSQL DB tables:

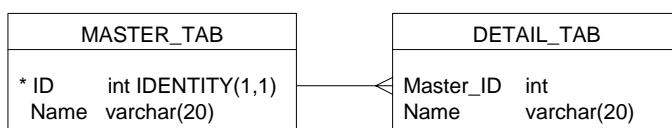


Figure 16 Tables related through FK

Now we want to fill these tables on client in cached updates mode, then post updates (insert new records) to DB. Main requirement is to preserve relation of detail records with they master records. Following example performs that:

```
var
oSchAdapt: IADDaptSchemaAdapter;
oMasterAdapt, oDetailAdapt: IADDaptTableAdapter;
oMasterRow: TADatSRow;
```

```

begin
    // 1. create master table adapter
    oMasterAdapt := oSchAdapt.TableAdapters.Add('master_tab');
    oMasterAdapt.SelectCommand := oConn.CreateCommand;
    oMasterAdapt.SelectCommand.Prepare('select * from master_tab');
    oMasterAdapt.Define;

    // 2. Mark ID column autoincrementing with negative step, so it
    // new generated value will be always distinguishing from fetched
    // from DB one.
    with oMasterAdapt.DatSTable.Columns.ItemsS['ID'] do begin
        AutoIncrement := True;
        AutoIncrementSeed := 0;
        AutoIncrementStep := -1;
    end;
    oMasterAdapt.Fetch(True);

    // 3. create detail table adapter
    oDetailAdapt := oSchAdapt.TableAdapters.Add('detail_tab');
    oDetailAdapt.SelectCommand := oConn.CreateCommand;
    oDetailAdapt.SelectCommand.Prepare('select * from detail_tab');
    oDetailAdapt.Fetch(True);

    // 4. create primary key for Master table
    with oSchAdapt.DatSManager.Tables.ItemsS['master_tab'] do
        Constraints.AddUK('master_pk', 'ID', True);

    // 5. create foreign key Detail -> Master with cascading update
    with oSchAdapt.DatSManager.Tables.ItemsS['detail_tab'] do begin
        with Constraints.AddFK('fk_detail_master', 'master_tab', 'ID', 'Master_ID') do begin
            UpdateRule := crCascade;
            DeleteRule := crCascade;
            AcceptRejectRule := arCascade;
        end;
    end;

    // 5. fill DatS tables
    oMasterRow := oMasterAdapt.DatSTable.Rows.Add([Unassigned, 'RecMaster']);
    oDetailAdapt.DatSTable.Rows.Add([Unassigned, oMasterRow.GetData('ID'), 'RecDetail']);

    // 6. post all updates in single consistent batch. Here new (assigned by DB)
    // Master table ID will be fetched, assigned to DatS row, and Detail table
    // will be cascade updated. Then update of Detail row will be posted to DB.
    oSchAdapt.Update;

```

There at **Update** call following will happen:

- Inserting new master row into MASTER_TAB DB table.
- Fetching new ID column value and set it to processing master table row.
- Firing cascading updates of detail rows Master_ID column.

- Inserting new detail row into DETAIL_TAB DB table. Here Master_ID column will have value as it was inserted into MASTER_TAB table.

6.2.5 Commands Handling

As was mentioned above, each physical command performed by adapter to post updates from **DatS** row to DB may be overridden by application programmer. For that one should create command and assign it to one of **IADDAptTableAdapter.XXXXCommand** properties. Command should have parameter names of special form:

- **:NEW_<column name>** or **<column name>**. Input parameter will receive as value current version of column value (**rvCurrent**). Output parameter value will be assigned after command execution to column.
- **:OLD_<column name>**. Input parameter will receive as value original version of column value (**rvOriginal**).

If command is not assigned explicitly, **DApt** layer will generate it automatically. It will be optimised for target RDBMS command language dialect. Table adapter will cache generated commands, if

TADUpdateOptions.CacheUpdateCommands property value is True. Otherwise, after usage, command interface will be discarded. If adapter need to regenerate command, cached command interface will be discarded too and command will be regenerated.

If it is desirable to completely override standard adapter algorithms, then programmer should use update handler. It may be assigned to **IADDAptTableAdapter.UpdateHandler** property. Which is of **IADDAptUpdateHandler** interface type. Following declaration shows details:

```
IADDAptUpdateHandler = interface(IInterface)
    ['{3E9B315B-F456-4175-A864-B2573C4A2302}']
    procedure ReconcileRow(ARow: TADDatSRow; var Action: TADDAptReconcileAction);
    procedure UpdateRow(ARow: TADDatSRow; ARequest: TADPhysUpdateRequest;
        AUpdOptions: TADPhysUpdateOptions; var AAction: TADErrorAction);
end;
```

It has **UpdateRow** method, which will be called by **IADDAptTableAdapter.Update** method.

6.2.6 Error Handling

At updates posting RDBMS errors may happen, for example, uniqueness violation or other constraints violation. AnyDAC represents each error as an exception, which will be associated with appropriate **DatS** row, using **TADDatSRow.Error** property. Updates will be posted to DB using **Update(AMaxErrors)** method until **AMaxErrors** errors will happen. After that errors may be reconciled using **Reconcile** method, it requires **IADDAptTableAdapter.UpdateHandler** property to be assigned. Reconciliation will call **ReconcileRow** method.

Appendix 1. TADDataType

Data type	Description	Range
dtBoolean	Boolean data type with values – True and False.	True, False
dtSByte	Signed 1-byte integer	-128...127
dtInt16	Signed 2-byte integer	-32768...32767
dtInt32	Signed 4-byte integer	-2147483648...2147483647
dtInt64	Signed 8-byte integer	-2 ⁶³ ...2 ⁶³ -1
dtByte	Unsigned 1-byte integer	0...255
dtUInt16	Unsigned 2-byte integer	0...65535
dtUInt32	Unsigned 4-byte integer	0...4294967295
dtUInt64	Unsigned 8-byte integer	0...2 ⁶⁴ -1
dtDouble	8-byte native IA-32 floating point value. Corresponds to Delphi Double data type	5.0 x 10 ⁻³²⁴ ... 1.7 x 10 ³⁰⁸
dtCurrency	8-byte fixed point value. Corresponds to Delphi Currency data type	-922337203685477.5808 ... 922337203685477.5807
dtBCD	Binary Coded Decimal. Corresponds to Delphi TBcd data and ftBcd field types.	-10 ⁶⁴ +1 ... 10 ⁶⁴ -1
dtFmtBCD	Binary Coded Decimal. Corresponds to Delphi TBcd data and ftFmtBCD field types.	-10 ⁶⁴ +1 ... 10 ⁶⁴ -1
dtDateTime	Encoded as TDateTimeRec data and time. Consist from 2 Longint values. Low order – date and high order – time.	See dtTime and dtDate.
dtTime	Encoded as Longint time. Equals to number of milliseconds since midnight.	-2147483648...2147483647
dtDate	Encoded as Longint date. Equals to number of days since beginning of current era.	-2147483648...2147483647
dtDateTimeStamp	Date and time stamp. Corresponds to Delphi TSQLTimeStamp data and ftTimeStamp field types.	See TSQLTimeStamp for details.
dtAnsiString	Ansi character string.	~2 ³¹ characters
dtWideString	Unicode character (16 bit per char) string.	~2 ³⁰ characters
dtByteString	Byte string.	~2 ³¹ bytes
DtBlob	Long byte string (Blob).	~2 ³² bytes
dtMemo	Long Ansi character string (Memo).	~2 ³² characters
dtWideMemo	Long Unicode character string (Memo).	~2 ³¹ characters

dtHBlob	As dtBlob, but corresponds to Oracle BLOB data type and other handle-based BLOB data types.	~2 ³² bytes
dtHMemo	As dtMemo, but corresponds to Oracle CLOB data type and other handle-based ANSI text BLOB data types.	~2 ³² characters
dtWideHMemo	As dtWideMemo, but corresponds to Oracle NCLOB data type and other handle-based Unicode text BLOB data types.	~2 ³¹ characters
dtRowSetRef	Nested data set. Value is an “invariant” data type, represented by list of nested rows. Corresponds to Delphi ftDataSet field type.	--
dtCursorRef	Nested data set. Value is an “invariant” data type, represented by list of nested rows. Corresponds to Delphi ftCursor field type.	--
dtRowRef	Nested row. Value is an “invariant” data type, represented by single nested row. Corresponds to Delphi ftADT field type.	--
dtArrayRef	Array of values. Value is an “invariant” data type, represented by list of nested rows. Corresponds to Delphi ftArray field type.	--
dtParentRowRef	AnyDAC internal data type.	--
dtGUID	GUID data structure.	See TGUID for details.
dtObject	Reference to IADDataStoredObject interface instance.	IADDataStoredObject interface.

Appendix 2. Macro data types

Data type	Meaning	Macro value	Substitution
mdString	Literal constant	QWE	'QWE'
mdIdentifier	Quoted identifier	QWE	[QWE]
mdInteger	Integer value	123	123
mdBoolean	Boolean value	1	True
mdFloat	Floating point numeric value	1.23	1,234
mdDate	Date value	12-02-03	DateValue('12-02-03')
mdTime	Time value	09:00	DateValue('09:00')
mdDateTime	Date and time value	12-02-03 09:00	DateValue('12-02-03 09:00')
mdRaw	Any sequence of symbols, "as is"	QWE !@# 123	QWE !@# 123

Appendix 3. Top level interfaces and GUID's

GUID	Interface	AnyDAC standard implementation unit
3E9B315B-F456-4175-A864-B2573C4A2003	IADStanObjectFactory	daADStanPool
3E9B315B-F456-4175-A864-B2573C4A2009	IADStanExpressionParser	daADStanExpr
3E9B315B-F456-4175-A864-B2573C4A2012	IADStanDefinitionStorage	daADStanDef
3E9B315B-F456-4175-A864-B2573C4A2013	IADStanDefinition	daADStanDef
3E9B315B-F456-4175-A864-B2573C4A2014	IADStanDefinitions	daADStanDef
3E9B315B-F456-4175-A864-B2573C4A2015	IADStanConnectionDef	daADStanDef
3E9B315B-F456-4175-A864-B2573C4A2016	IADStanConnectionDefs	daADStanDef
3E9B315B-F456-4175-A864-B2573C4A2023	IADStanAsyncExecutor	daADStanAsync
3E9B315B-F456-4175-A864-B2573C4A2026	IADMoniRemoteClient	daADMoniIndyClient
3E9B315B-F456-4175-A864-B2573C4A2027	IADMoniFlatFileClient	daADMoniFlatFile
3E9B315B-F456-4175-A864-B2573C4A2030	IADMoniCustomClient	daADMoniCustom
3E9B315B-F456-4175-A864-B2573C4A2101	IADPhysManager	daADPhysManager
3E9B315B-F456-4175-A864-B2573C4A2200	IADGUIxLoginDialog	daADGUIxFormsfLogin
3E9B315B-F456-4175-A864-B2573C4A2201	IADGUIxWaitCursor	daADGUIxFormsfWait, daADGUIxConsoleWait
3E9B315B-F456-4175-A864-B2573C4A2202	IADGUIxAsyncExecuteDialog	daADGUIxFormsfAsync
3E9B315B-F456-4175-A864-B2573C4A2203	IADGUIxErrorDialog	daADGUIxFormsfError
3E9B315B-F456-4175-A864-B2573C4A2204	IADGUIxDefaultLoginDialog	daADGUIxFormsfLogin
3E9B315B-F456-4175-A864-B2573C4A2304	IADDaptTableAdapter	daADDaptManager
3E9B315B-F456-4175-A864-B2573C4A2306	IADDaptSchemaAdapter	daADDaptManager

Appendix 4. Connection definition parameters

[AnyDACSettings] section

This connection definition file section is common for all definitions in the same file. At most, there are parameters controlling debug monitor.

Parameter	Description	Default value
MonitorInDelphiIDE	The monitor client, running at design time in Delphi IDE, will not produce output if False is specified.	False
MonitorCategories	The monitor client will output messages only of specified categories. The value is a bit mask, where each bit corresponds to TADDebugEventKind enumeration item.	\$FFFF
MonitorByIndy_Host	The remote monitor client will connect to remote monitor server running on specified host.	127.0.0.1
MonitorByIndy_Port	The remote monitor client will connect to remote monitor server listening on specified port.	8050
MonitorByIndy_Timeout	The remote monitor client will try to connect to remote monitor server specified time.	1000
MonitorByFlatFile_FileName	The file monitor client will produce output into specified file.	AnyDAC \$(RAND).TRC
MonitorByFlatFile_Append	The file monitor client will append output into file or rewrite a file.	False

Connection definition section. General.

Parameter	Description	Default value
DriverID	Specifies AnyDAC driver identifier. May be one of the following values: <ul style="list-style-type: none">• DBX – dbExpress generic source• ODBC – ODBC generic source• MSSQL – Microsoft SQL Server (pre 2000)• MSSQL2005 – Microsoft SQL Server Native Client (2005 and later)	<none>

	<ul style="list-style-type: none"> • MSAcc – Microsoft Access • DB2 – IBM DB2 UDB • MySQL – MySQL • Ora – Oracle • ASA – Sybase ASA 	
MonitorBy	<p>Specifies monitoring output mode. May be one of the following values:</p> <ul style="list-style-type: none"> • Indy – interactive monitoring using Indy transport • FlatFile – output to the flat file • Custom – custom output 	<none>

Appendix 5. Macro functions.

CHARACTER

Function	Description
ASCII (<i>string_exp</i>)	Returns the ASCII code value of the leftmost character of <i>string_exp</i> as an integer.
BIT_LENGTH (<i>string_exp</i>)	Returns the length in bits of the string expression.
CHAR (<i>code</i>)	Returns the character that has the ASCII code value specified by <i>code</i> . The value of <i>code</i> should be between 0 and 255; otherwise, the return value is data source–dependent.
CHAR_LENGTH (<i>string_exp</i>)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHARACTER_LENGTH function.)
CHARACTER_LENGTH (<i>string_exp</i>)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHAR_LENGTH function.)
CONCAT (<i>string_exp1</i> , <i>string_exp2</i>)	Returns a character string that is the result of concatenating <i>string_exp2</i> to <i>string_exp1</i> . The resulting string is DBMS-dependent. For example, if the column represented by <i>string_exp1</i> contained a NULL value, DB2 would return NULL but SQL Server would return the non-NULL string.
DIFFERENCE (<i>string_exp1</i> , <i>string_exp2</i>)	Returns an integer value that indicates the difference between the values returned by the SOUNDEX function for <i>string_exp1</i> and <i>string_exp2</i> .
INSERT (<i>string_exp1</i> , <i>start</i> , <i>length</i> , <i>string_exp2</i>)	Returns a character string where <i>length</i> characters have been deleted from <i>string_exp1</i> , beginning at <i>start</i> , and where <i>string_exp2</i> has been inserted into <i>string_exp</i> , beginning at <i>start</i> .
LCASE (<i>string_exp</i>) (ODBC 1.0)	Returns a string equal to that in <i>string_exp</i> , with all uppercase characters converted to lowercase.

LEFT (<i>string_exp</i> , <i>count</i>)	Returns the leftmost <i>count</i> characters of <i>string_exp</i> .
LENGTH (<i>string_exp</i>)	Returns the number of characters in <i>string_exp</i> , excluding trailing blanks.
LOCATE (<i>string_exp1</i> , <i>string_exp2</i> [, <i>start</i>])	Returns the starting position of the first occurrence of <i>string_exp1</i> within <i>string_exp2</i> . The search for the first occurrence of <i>string_exp1</i> begins with the first character position in <i>string_exp2</i> unless the optional argument, <i>start</i> , is specified. If <i>start</i> is specified, the search begins with the character position indicated by the value of <i>start</i> . The first character position in <i>string_exp2</i> is indicated by the value 1. If <i>string_exp1</i> is not found within <i>string_exp2</i> , the value 0 is returned.
LTRIM (<i>string_exp</i>)	Returns the characters of <i>string_exp</i> , with leading blanks removed.
OCTET_LENGTH (<i>string_exp</i>)	Returns the length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8.
POSITION (<i>character_exp</i> , <i>character_exp</i>)	Returns the position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precision and a scale of 0.
REPEAT (<i>string_exp</i> , <i>count</i>)	Returns a character string composed of <i>string_exp</i> repeated <i>count</i> times.
REPLACE (<i>string_exp1</i> , <i>string_exp2</i> , <i>string_exp3</i>)	Search <i>string_exp1</i> for occurrences of <i>string_exp2</i> , and replace with <i>string_exp3</i> .
RIGHT (<i>string_exp</i> , <i>count</i>)	Returns the rightmost <i>count</i> characters of <i>string_exp</i> .
RTRIM (<i>string_exp</i>)	Returns the characters of <i>string_exp</i> with trailing blanks removed.
SOUNDEX (<i>string_exp</i>)	Returns a data source–dependent character string representing the sound of the words in <i>string_exp</i> . For example, SQL Server returns a 4-digit SOUNDEX code; Oracle returns a phonetic representation of each word.
SPACE (<i>count</i>)	Returns a character string consisting of <i>count</i> spaces.
SUBSTRING (<i>string_exp</i> , <i>start</i> , <i>length</i>)	Returns a character string that is derived from <i>string_exp</i> , beginning at the character position specified by <i>start</i> for <i>length</i> characters.

UCASE (<i>string_exp</i>)	Returns a string equal to that in <i>string_exp</i> , with all lowercase characters converted to uppercase.
------------------------------------	---

NUMERIC

Function	Description
ABS (<i>numeric_exp</i>)	Returns the absolute value of <i>numeric_exp</i> .
ACOS (<i>float_exp</i>)	Returns the arccosine of <i>float_exp</i> as an angle, expressed in radians.
ASIN (<i>float_exp</i>)	Returns the arcsine of <i>float_exp</i> as an angle, expressed in radians.
ATAN (<i>float_exp</i>)	Returns the arctangent of <i>float_exp</i> as an angle, expressed in radians.
ATAN2 (<i>float_exp1</i> , <i>float_exp2</i>)	Returns the arctangent of the x and y coordinates, specified by <i>float_exp1</i> and <i>float_exp2</i> , respectively, as an angle, expressed in radians.
CEILING (<i>numeric_exp</i>)	Returns the smallest integer greater than or equal to <i>numeric_exp</i> . The return value is of the same data type as the input parameter.
COS (<i>float_exp</i>)	Returns the cosine of <i>float_exp</i> , where <i>float_exp</i> is an angle expressed in radians.
COT (<i>float_exp</i>)	Returns the cotangent of <i>float_exp</i> , where <i>float_exp</i> is an angle expressed in radians.
DEGREES (<i>numeric_exp</i>)	Returns the number of degrees converted from <i>numeric_exp</i> radians.
EXP (<i>float_exp</i>)	Returns the exponential value of <i>float_exp</i> .
FLOOR (<i>numeric_exp</i>)	Returns the largest integer less than or equal to <i>numeric_exp</i> . The return value is of the same data type as the input parameter.
LOG (<i>float_exp</i>)	Returns the natural logarithm of <i>float_exp</i> .
LOG10 (<i>float_exp</i>)	Returns the base 10 logarithm of <i>float_exp</i> .
MOD (<i>integer_exp1</i> , <i>integer_exp2</i>)	Returns the remainder (modulus) of <i>integer_exp1</i> divided by <i>integer_exp2</i> .
PI ()	Returns the constant value of pi as a floating-point value.
POWER (<i>numeric_exp</i> , <i>integer_exp</i>)	Returns the value of <i>numeric_exp</i> to the power of <i>integer_exp</i> .
RADIANS (<i>numeric_exp</i>)	Returns the number of radians converted from <i>numeric_exp</i> degrees.
RAND ([<i>integer_exp</i>])	Returns a random floating-point value using <i>integer_exp</i> as the optional seed value.
ROUND (<i>numeric_exp</i> , <i>integer_exp</i>)	Returns <i>numeric_exp</i> rounded to <i>integer_exp</i> places right of the decimal point. If <i>integer_exp</i> is negative, <i>numeric_exp</i> is rounded to <i>integer_exp</i> places to the left of the decimal point.

SIGN (<i>numeric_exp</i>)	Returns an indicator of the sign of <i>numeric_exp</i> . If <i>numeric_exp</i> is less than zero, -1 is returned. If <i>numeric_exp</i> equals zero, 0 is returned. If <i>numeric_exp</i> is greater than zero, 1 is returned.
SIN (<i>float_exp</i>)	Returns the sine of <i>float_exp</i> , where <i>float_exp</i> is an angle expressed in radians.
SQRT (<i>float_exp</i>)	Returns the square root of <i>float_exp</i> .
TAN (<i>float_exp</i>)	Returns the tangent of <i>float_exp</i> , where <i>float_exp</i> is an angle expressed in radians.
TRUNCATE (<i>numeric_exp</i> , <i>integer_exp</i>)	Returns <i>numeric_exp</i> truncated to <i>integer_exp</i> places right of the decimal point. If <i>integer_exp</i> is negative, <i>numeric_exp</i> is truncated to $ integer_exp $ places to the left of the decimal point.

TIME, DATE

Function	Description
CURRENT_DATE()	Returns the current date.
CURRENT_TIME[(<i>time-precision</i>)]	Returns the current local time. The <i>time-precision</i> argument determines the seconds precision of the returned value.
CURRENT_TIMESTAMP [(<i>timestamp-precision</i>)]	Returns the current local date and local time as a timestamp value. The <i>timestamp-precision</i> argument determines the seconds precision of the returned timestamp.
CURDATE()	Returns the current date.
CURTIME()	Returns the current local time.
DAYNAME(<i>date_exp</i>)	Returns a character string containing the data source–specific name of the day (for example, Sunday through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of <i>date_exp</i> .
DAYOFMONTH(<i>date_exp</i>)	Returns the day of the month based on the month field in <i>date_exp</i> as an integer value in the range of 1–31.
DAYOFWEEK(<i>date_exp</i>)	Returns the day of the week based on the week field in <i>date_exp</i> as an integer value in the range of 1–7, where 1 represents Sunday.
DAYOFYEAR(<i>date_exp</i>)	Returns the day of the year based on the year field in <i>date_exp</i> as an integer value in the range of 1–366.
EXTRACT(<i>extract-field</i>, <i>extract-source</i>)	<p>Returns the <i>extract-field</i> portion of the <i>extract-source</i>. The <i>extract-source</i> argument is a datetime or interval expression. The <i>extract-field</i> argument can be one of the following keywords:</p> <p>YEAR MONTH DAY HOUR MINUTE SECOND</p> <p>The precision of the returned value is implementation-defined. The scale is 0 unless SECOND is specified, in which case the scale is not less than the fractional seconds precision of the <i>extract-source</i> field.</p>
HOUR(<i>time_exp</i>)	Returns the hour based on the hour field in <i>time_exp</i> as an integer value in the range of 0–23.
MINUTE(<i>time_exp</i>)	Returns the minute based on the minute field in <i>time_exp</i> as an integer value in the range of 0–59.

MONTH (<i>date_exp</i>)	Returns the month based on the month field in <i>date_exp</i> as an integer value in the range of 1–12.
MONTHNAME (<i>date_exp</i>)	Returns a character string containing the data source–specific name of the month (for example, January through December or Jan. through Dec. for a data source that uses English, or Januar through Dezember for a data source that uses German) for the month portion of <i>date_exp</i> .
NOW ()	Returns current date and time as a timestamp value.
QUARTER (<i>date_exp</i>)	Returns the quarter in <i>date_exp</i> as an integer value in the range of 1–4, where 1 represents January 1 through March 31.
SECOND (<i>time_exp</i>)	Returns the second based on the second field in <i>time_exp</i> as an integer value in the range of 0–59.
TIMESTAMPADD (<i>interval</i> , <i>integer_exp</i> , <i>timestamp_exp</i>)	<p>Returns the timestamp calculated by adding <i>integer_exp</i> intervals of type <i>interval</i> to <i>timestamp_exp</i>. Valid values of <i>interval</i> are the following keywords:</p> <p>FRAC_SECOND SECOND MINUTE HOUR DAY WEEK MONTH QUARTER YEAR</p> <p>where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and his or her one-year anniversary date:</p> <pre>SELECT NAME, {fn TIMESTAMPADD('YEAR', 1, HIRE_DATE)} FROM EMPLOYEES</pre> <p>If <i>timestamp_exp</i> is a time value and <i>interval</i> specifies days, weeks, months, quarters, or years, the date portion of <i>timestamp_exp</i> is set to the current date before calculating the resulting timestamp.</p> <p>If <i>timestamp_exp</i> is a date value and <i>interval</i> specifies fractional seconds, seconds, minutes, or hours, the time portion of <i>timestamp_exp</i> is set to 0 before calculating the resulting timestamp.</p>

TIMESTAMPDIFF(*interval*, *timestamp_exp1*, *timestamp_exp2*) Returns the integer number of intervals of type *interval* by which *timestamp_exp2* is greater than *timestamp_exp1*. Valid values of *interval* are the following keywords:

FRAC_SECOND
SECOND
MINUTE
HOUR
DAY
WEEK
MONTH
QUARTER
YEAR

where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and the number of years he or she has been employed:

```
SELECT NAME, {fn TIMESTAMPDIFF('YEAR', {fn CURDATE()},  
HIRE_DATE)} FROM EMPLOYEES
```

If either timestamp expression is a time value and *interval* specifies days, weeks, months, quarters, or years, the date portion of that timestamp is set to the current date before calculating the difference between the timestamps.

If either timestamp expression is a date value and *interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of that timestamp is set to 0 before calculating the difference between the timestamps.

WEEK(*date_exp*)

Returns the week of the year based on the week field in *date_exp* as an integer value in the range of 1–53.

YEAR(*date_exp*)

Returns the year based on the year field in *date_exp* as an integer value. The range is data source–dependent.

SYSTEM

Function	Description
DATABASE()	Returns the name of the database corresponding to the connection.
IFNULL(<i>exp</i>, <i>value</i>)	If <i>exp</i> is null, <i>value</i> is returned. If <i>exp</i> is not null, <i>exp</i> is returned. The possible data type or types of <i>value</i> must be compatible with the data type of <i>exp</i> .
USER()	Returns the user name in the DBMS. This can be different than the login name.

CONVERT

The format of the **CONVERT** function is:

CONVERT(*value_exp*, *data_type*)

The function returns the value specified by *value_exp* converted to the specified *data_type*, where *data_type* is one of the following keywords:

BIGINT	REAL
BINARY	SMALLINT
BIT	DATE
CHAR	TIME
DECIMAL	TIMESTAMP
DOUBLE	TINYINT
FLOAT	VARBINARY
GUID	VARCHAR
INTEGER	WCHAR
LONGVARBINARY	WLONGVARCHAR
LONGVARCHAR	WVARCHAR
NUMERIC	

The syntax for the explicit data type conversion function does not support specification of conversion format. The argument *value_exp* can be a column name, the result of another scalar function, or a numeric or string literal. For example:

```
{ fn CONVERT( { fn CURDATE() }, CHAR ) }
```

converts the output of the CURDATE scalar function to a character string. Because AnyDAC does not mandate a data type for return values from scalar functions (because the functions are often data source-specific), applications should use the CONVERT scalar function whenever possible to force data type conversion. The following two examples illustrate the use of the **CONVERT** function. These examples assume the existence of a table called EMPLOYEES, with an EMPNO column of type SQL_SMALLINT and an EMPNAME column of type SQL_CHAR. If an application specifies the following SQL statement:

```
SELECT EMPNO FROM EMPLOYEES WHERE {fn CONVERT(EMPNO,CHAR)} LIKE '1%'
```

- A driver for ORACLE translates the SQL statement to:

```
SELECT EMPNO FROM EMPLOYEES WHERE to_char(EMPNO) LIKE '1%'
```

- A driver for SQL Server translates the SQL statement to:

```
SELECT EMPNO FROM EMPLOYEES WHERE convert(char,EMPNO) LIKE '1%'
```

If an application specifies the following SQL statement:

```
SELECT {fn ABS(EMPNO)}, {fn CONVERT(EMPNAME,SMALLINT)}  
FROM EMPLOYEES WHERE EMPNO <> 0
```

- A driver for ORACLE translates the SQL statement to:

```
SELECT abs(EMPNO), to_number(EMPNAME) FROM EMPLOYEES WHERE EMPNO <> 0
```

- A driver for SQL Server translates the SQL statement to:

```
SELECT abs(EMPNO), convert(smallint, EMPNAME) FROM EMPLOYEES  
WHERE EMPNO <> 0
```