# Scientific Data Manager (SciDM)

Design, Features and Architecture

# Table of Contents

# Overview

**Scientific Data Manager** (**SciDM**) is a DBMS designed for applications that require:
operating quickly with heavy data sets over the network -
on standard inexpensive hardware, maintenance-free -
through a light and easy-to-use programming interface -
using multiple platforms and various programming languages.

SciDM provides:

- **Unmatched performance**:
  retrieving 1,000,000 data objects in 0.9 seconds
  updating 1,000,000 data objects in 2.1 seconds
  creating 1,000,000 data objects in 37 seconds
  deleting 1,000,000 data objects in 21.5 seconds
  processing 1,000,000 select queries in 5.2 seconds
  handling 10,000 client sessions simultaneously -
  over the network[1], running on standard hardware[2], on a database preloaded with 540 Gigabytes of data[3].

- **Strong reliability:** All features were tested in the real-world applications that intensively operate with huge volumes of data. These applications include genomic data management, literature databases, document management, and bug tracking.

- **Truly unlimited storage:** limited by hardware only. The largest instance of SciDM-managed database we worked with held 12.5 Terabytes. The storage is theoretically limited to $2^{48}$ objects; each object can contain up to 2 billion data attributes; each data attribute can contain up to $2^{63}$ bytes.

- **Reach data management capabilities** at an unprecedented speed. It uses an object–relational model for data representation and provides methods for storing, retrieving, and deleting data objects, for sequential and indexed access and for dynamic data structure discovery. Along with traditional indexing by the entire attribute contents, SciDM provides 'context'-style indices by individual words in the stored texts.

- **Structured framework for data processing:** formalized in terms of particular application fields. It makes SciDM highly suitable for research and prototyping, and also simplifies the development by removing traditional data translation layer between the application and DBMS and bringing structured data directly into the processing modules.

- **Interoperability over various platforms**, operating systems and programming languages through the use of CORBA as a network transport. CORBA delivers structured data in constructs natural for the target programming language, creating a friendly and **light programming model**.

- **Security model with object–level protection**. The access rights are controlled individually for every object.

---

[1] TCP/IP over Gigabit Ethernet
[2] AMD Athlon 7750 Dual-Core CPU (2.7 GHz), 4Gb DDR2 RAM, 5x1Tb SATA2 Hard drives (RAID5)
[3] The particular table used for benchmarking already contained 20,000,000 records.

- Among other features of SciDM there are **flexible object structure**, allowing dynamic addition of new attributes to existing objects; **integrated set management**, allowing both persistence and server-side operations for objects sets; **data integrity support** through locking and object subordination.

SciDM server can be viewed as middleware between CORBA-interface, reflecting the structure of the client's objects, and data storage back-end. Current production version of SciDM is based on Sequiter's CodeBase, combined with a proprietary storage manager (EMBEDB). The EMBEDB have been specifically designed to overcome traditional volume limitations, and provides a highly efficient storage of practically unlimited amounts of data in a maintenance-free mode. A reference implementation of SciDM that uses ORACLE as data storage back-end is also available.

# Motivation

SciDM project was started as an attempt to build a data-handling layer for a suite of DNA sequence analysis applications. Commercial DBMS systems failed because they could not fulfill two major requirements: 1) data had to be loaded in a finite time and 2) massive amounts of data had to be quickly retrievable. The number of DNA sequences in the public GenBank database is roughly 150 million. One of the simplest but typical tasks is comparing of a novel sequence to the already known ones. *It involves retrieving many (or all) of the records, passing them to a comparison program, and storing back the produced output. Widely used DNA comparison algorithms are capable of processing the entire GenBank in the range of second to minutes. Retrieving such amount of data from a conventional DBMS would take 2 to 4 orders of magnitude longer on conventional hardware.*

Such slowness is, indeed, reasonable. The primary market for the commercial DBMSes is financial or material databases. Even if the database serves the banking system with a million of customers, and every customer makes ten transactions per day (which is unlikely), it comes to an average of 10*1,000,000 / 24*60*60=116 transactions per second. We wanted 10,000,000 records to be retrieved in the time lesser then 5 minutes, which requires a speed of 10,000,000 / 5*60 = 33,333 transactions per second – 286 times more.

Thus, commercial DBMSes are not designed for and typically not used for applications requiring very high data exchange rates. Benchmark demonstrations of commercial DBMSes employ extreme hardware and require substantial support of DBMS administrators and tuners. In normal operating conditions they do not demonstrate their posted speed figures.

There are a few major bottlenecks that impose limitations on the DBMS operating speed. First is the network transfer. TCP stack on Linux, Windows or Mac has a limited throughput of about 1000 – 1500 data packets per second. If data objects (or records) are transferred one by one, there is no chance to transfer more than a half of this number (half because for each record sent to a client the server needs a confirmation of receiving). Some commercial DBMSes offer special operations for transferring many objects per transaction. For example, ORACLE has array methods available through Oracle Call Interface. These methods, however, are far from industrial standard: they require extreme programming efforts for use; besides, and are poorly documented and fragile.

Other bottlenecks are related to transaction safety and journaling. Every piece of data in commercial DBMSes is written to a database at least three times: into a data store, into a rollback log, and into a journal (transaction log) after the transaction's completion. This multiplies the number of disk accesses, increases the sparseness of the data and inflates the storage volume. Journaling provides the ability to roll back to a state in which the database was an arbitrary time

ago. This ability is rarely needed for non-financial applications. The transaction/rollback control provides data integrity in case of hardware failure. An alternative method for supporting integrity is the use of external tools to find and fix desynchronized pieces of data. Hardware failures happen rear enough to afford spending time on running such tool after a failure instead of continuous slowing the data access by (at least) the factor of two.

Along with the speed, the needs of biological data analysis posted a whole set of requirements not satisfied by traditional DBMSes. A major one is that the data management system had to serve the research needs, allowing for fast prototyping. It was desirable to access the data as if it was not persistent somewhere on the server, but available locally within a processing program. Among other requirements were object-level security, bindings to both production (C/C++, JAVA) and prototyping (Perl, Python) languages, availability on multiple platforms (Linux, Solaris, Win2K, Mac), and operating over a LAN and Internet.

It also has to be pointed that high volumes of data create heavy problems with commercial databases. Their handling requires careful planning and continuous effort of database administrator. For the research environment, it is merely impossible to do such planning, as there is no possibility to estimate the volume of particular tables upfront. A system that supports research tasks has to work in maintenance-free mode with relatively heavy data load, assuming the continuous disk space is available, and should not degrade in performance as it accumulates more data.

There are solutions for data management other then commercial DBMSes. Careful consideration shows, however, that none of them are acceptable for the proposed task. PostgreSQL possessed all limitations of commercial SQL packages. The non-SQL systems, leaded by BerkleyDB, do not offer the required integrity and can be viewed only as a set of blocks out of which the data management system could be built. The BerkleyDB in particular also shows heavy performance problems when operating simultaneously on multiple indices. The majority of commercial and non-commercial ‘embedded’ databases is based on Dbase and carries unacceptable limitations on the amount of stored data. Most of them also do not provide the desired high-level features like remote access or security.

Data management requirements for generic data are relatively standard. Any commercial or academic institution operating with large amounts of data would share similar needs. This applies to biotechnology in particular because over the past decade it vastly outgrew the abilities of traditional DBMSes. This also applies to economy, medicine, meteorology, geography, astronomy, oceanography and many other fields where the amount of accumulated data is huge, and typical analysis tasks require retrieval (or storage) of its substantial portions.

We have formulated a set of requirements for a Data Management System suitable for academic and commercial research applications. Among them we prioritized structured data storage, versatile indexed access, high operating speed, unlimited data volumes, multi-user network access, availability over multiple platforms, natural representation of the data in various programming languages, light programming model, security at the level of individual objects and maintenance–free operation on inexpensive hardware. Having the list of requirements, we designed and implemented the SciDM.

# Design Principles

## *High Performance*

Speed of operation has been considered the first priority principle. Wherever other features could be reasonably compromised for speed, the faster solution was chosen. If the faster implementation required a heavier design, the faster implementation was chosen. If the faster implementation required more resources from the executing system, like more memory, the faster implementation was chosen, as long as the increase in speed was higher than the increase in the system cost. If the faster implementation required more storage space, the faster implementation was chosen.

One of solutions favoring speed over flexibility is the use of compiled data access methods instead of dynamic ones. The complied access methods allow for compile-time optimization and thus highly increase the speed. However, this makes dynamic allocation of new types a much heavier task. In the current implementation of SciDM, adding a new type involves changing the schema, rebuilding the data manager executable, building the schema update tool, stopping the server, running the schema update tool and starting the updated server instead of the old one. All existing data is preserved, and a new type is added. In the future this procedure is going to be replaced by dynamic loading of complied schema component into the running server along with automatic creation of persistent structures on the disk. This is quite a burden, but the access methods are executed much more often than the schema changes occur. The accumulated time saved by the invocation of compiled access methods vastly covers the time overhead for the schema updates.

## *Unlimited Storage*

The volume of data that needs to be stored is hard to estimate upfront. Any ‘reasonable’ limit seems to become outdated very quickly. The limitations imposed by the computer technology also tend to weaken very quickly and should also be put out of consideration.

The requirement of unlimited storage has two aspects. First is related to the nature of object identity. Every object in the system should have an identity inherent to the object (not derived from a particular relation between the object and the system, such as physical address). Such identity is represented as identifier – a data structure operable by the computer system. The unlimited number of objects in the system requires an identifier of an unspecified (or variable) size. This imposes a limitation on performance, since operations on the object IDs are the most frequent in DMS, and a variable length ID takes more time to process then a fixed length one. For the time being, we assumed that 64-bit object identity is enough for all imaginable practical applications. We reserved, however, a possibility to increase the ID size to 128 (or any arbitrary number) bits. So far 64-bit ID was more than enough for all applications of SciDM.

The second aspect is related to the underlying operating environment. Many operating systems have limitations on the file or volume size; besides, the hardware has certain capacity limits. Coping with this restrictions means that the low-level data storage should be designed in a way that allows spanning the data over multiple files and multiple storage volumes. On some systems it requires also caching limited resources such as file handles.

## Extensibility

The extensibility can be viewed from several different perspectives. First is operational extensibility: the user should be able to add new types of data to the existing database. This is trivial and available in most commercial DBMS packages as basic functionality. In SciDM, however, the schema update is less trivial because it involves generating, compiling and loading the access methods code into the server. It must be noted, however, that schema design, especially in dynamic research areas, always contains certain flows and deficiencies. Restructuring the database is not always affordable when such a deficiency is discovered. To suppress negative effect of these situations, all objects in SciDM are supplied with a set of methods allowing creation and manipulation with any number of custom attributes. These attributes, named dynamic attributes, are associated with a particular object only, and do not affect the schema structure. When the schema lacks certain features, they still can be added and used through attaching dynamic attributes to the objects of interest.

The dynamic attributes provide an implementation buffer between the demands of the client application designer and the database schema. The client application may use the custom attributes as it would use the static ones. When a set of dynamically implemented attributes stabilizes through R&D cycle, the changes may be promoted into the static database schema. SciDM provides services for collecting statistics on the dynamic attributes, which may facilitate such promotion.

Dynamic attributes are also particularly useful when only a small subset of objects of a type s associated with additional information; they are a good alternative to reflecting these peculiarities in the database schema. In this case dynamic attributes may be used as add-on annotations to some of the objects. This inspired naming the interface for manipulating dynamic attributes '*Annotatable*'.

Another aspect of extensibility is the extensibility of server functionality. Certain operations involving large volumes of data are often common and trivial enough to be performed on the server, without transferring the data to the clients. Good examples of such operations are manipulations with sets of object IDs. To avoid unnecessary data transfers, these operations are implemented as a server component. The server component is a subsystem with a defined interface capable of accessing and manipulating the data directly on the server. Server component exports its own methods through CORBA. In the basic SciDM, the *IdSetManager*, the *RightsManager*, the *LockManager* and the Type system itself are implemented as server components. Any number of other components may be added. Currently adding a component requires rebuilding the server executable. In the future this will be replaced by dynamic loading of the compiled component code.

## High Accessibility

Different R&D communities use diverse hardware and diverse operating systems. Every field, if not every laboratory, has certain preferences in programming languages and tools. Even within the same institution different people often use different platforms. The data management system should make the data available through all these variations. Among few technologies that allow such interoperability CORBA is the most standard, widely accepted and most developed. Many CORBA providers over past years achieved both stabile operation and good performance. CORBA operates through all more-or-less standard platforms and operating systems, and have bindings to the majority of programming languages. These bindings are represented in constructs natural to the language and do not require learning or creating a translation layer around data access

methods. CORBA does not require any prerequisites on the system it operates besides a network protocol running.

These features made CORBA the only choice for using as a transport mechanism between SciDM clients and the server. We have used three different ORBs in the implementations of server and four in the implementation of clients. One of them, namely TAO, was able to inter-operate only with clients using the same ORB; the other two, namely omniORB and MICO, have shown excellent inter-operability between themselves, and with ORBIX and FNORB. Current production version of SciDM server is built using the omniORB.

## *Light Programming Model*

A typical production cycle, involving collection of requirements, design, prototyping, evaluation, production development, testing, quality control and support, usually makes little sense in the research community. There, a realistic cycle could include only two stages: prototyping and evaluation. These steps typically cannot be delegated to some trained person who knows how to program a particular data retrieval system: they have to be done on a daily basis by a researcher who formulates the questions. This means that to be efficient, the researcher (or developer) should have the data under their fingertips, in a form natural for them and not for the data storage system; and operations on the data should be expressed in a language natural for the data. CORBA serves the purpose of bringing remote objects into a familiar operating environment, but does not pose any limitation on the nature of the data: the data representation model is to be chosen while designing the DMS.

We found an object-relational model of data representation reasonable to satisfy the demands of both research and production. The details of object-relational representation are outlined below, in the Features section. Within SciDM client applications, the data objects are represented as natural constructs of the client language: objects in C++ or Java, structures in C, instances in Python, and blessed objects in Perl. The user deals not with some special data manipulation constructs like SQL queries or datasets, but directly with objects of his/her interest, possessing meaningful features as declared in the data schema.

## *Low Resource Consumption*

The last important consideration is the cost of the hardware system on which the SciDM server is running. For the academic communities and in many cases for commercial ones as well, this is a very important problem. The goal of our design was to avoid any special requirements for the hardware: the system should perform well on a standard single-processor PC computer. We considered that making SciDM efficient on a standard computer is much more important then making it scaleable and able to make use of multi-processor architecture. Also we considered important that SciDM could serve data in a single-computer setup, so it has to be able to stay as low as possible on the resource (memory and CPU) consumption to allow other applications (including client ones) to run efficiently.

Another important consideration relates to performance. As we already mentioned, a typical task we are addressing involves an access to a large set of objects at a time. Allocating and instantiating object representations for every object of such a set on the server would consume a lot of CPU time and memory. Such consumption is unnecessary, because even if the client processes the entire set of objects, they do not have to co-exist in the server's memory – the client accesses them one by one, so only the actually accessed object(s) needs to be present at any given moment.

To take advantage of this and to avoid instantiation of every accessed object on the server, the client accesses the data through special *Accessor* objects. The Accessor can be viewed as a window that shows one data object at a time and makes all methods associated with that object available. The Accessor can be switched (tuned) to any object of a given type by invoking the '*access*' method with the object ID as a parameter. So when browsing through a set of objects, the client uses a single Accessor and tunes it sequentially to all objects in the set. If the client must retain some object available, it may obtain an additional Accessor and keep it pointing to the object of interest, while another Accessor goes through the set. There is no limit on the number of Accessors; however, instantiating too many of them degrades the performance in the same way as instantiating of many objects would.

The Accessor mechanism is used universally through SciDM. Actually, all data manipulations are done through Accessors, including retrieval of metadata and handling of the type contents (see below).

# Features

## *Object – Relational Model*

Data entities are represented as *Objects*. Each object possesses an *Identifier* (*ID*), which is system-wide, unique, and never gets reused. Each object has a set of *Methods* it is capable to perform. The objects are classified into *Types* (similar to RDBMS relations), so that objects with the same set of methods belong to the same type. A set of Type layouts comprise the database *Schema*.

Typical methods are attribute access methods. They have a standard syntax of *get(AttributeName)* and *set(AttributeName)*. Each *Attribute* is a key–value pair, where "key" is an attribute's name (character string) and "value" belongs to any *DATATYPE* (described below).

The Types themselves are objects of a special type, called "Type". An object of the type 'Type' serves as container for all objects belonging to it. The Type object has methods for creating, destroying and enumerating objects as well as methods for retrieving metadata (lists of attributes, etc.). Each type has a *Type ID* and a *Type Name* (string). The type 'Type' contains itself as one of objects.

As mentioned above, all objects on the server are accessible through *Accessors*. An *Accessor* is a flyweight adapter to an actual object. The *accessor* can be tuned to any particular object of a given type. The *accessors* are typed: an *accessor* for objects of a certain type cannot access objects of another type. The Type objects serve as factories for accessors.

## *Extensible Objects*

Along with standard attributes defined by type, each object can possess any number of custom (dynamic) attributes. The custom attributes are accessible through dynamic attribute access methods: *get[AttrType](AttrName)* and *set[AttrType](AttrName, Value)*. For convenience of operations, methods for enumerating, testing, and deleting attributes are also available. The dynamic attributes may contain only data of certain kinds (see below).

## *Fast Data Storage and Retrieval through use of:*

### Generated Code

Benefits and drawbacks of compiled data access methods versus dynamic ones have been discussed earlier. C++ code for data access methods is generated for a particular schema by employing C@™ (cat) generative technology and then compiled by a C++ compiler into an executable. The code optimization performed by the C++ compiler allows for a faster operation in comparison with dynamic access methods.

### Bulk Methods

Typical data management tasks on large data sets are retrieval and storage of attributes of vast numbers of objects. Sending them over a network one by one hits the packet transfer rate limit. To pack many data transfer events into fewer network transactions, the *bulk methods* are introduced. The *Bulk* is a data structure that contains arrays of selected attributes for a selected set of objects. The accessor's *getBulk* and *setBulk* methods allow storage and retrieval bulks of data at once.

## *Fine-Grain Indexing by Entire Attribute Content and by Words in Text Attributes*

Two methods of indexing are provided via SciDM. First is conventional index by the attribute contents, allowing retrieval of sets of objects whose attributes fully or partially match a query. Any attribute of a type may be indexed by this method. Another is context index, providing for retrieval of sets of objects in which a given (or any) text attribute contains a word completely or partially matching the query. Text fields are broken into words according to Standard English grammar. (In the future, more flexible tokenizers may be introduced.) The latter indexing method is valuable for many applications that use datasets with elements of natural texts.

## *Custom Data Types of Unlimited Complexity*

Objects maintained by SciDM contain attributes. The attributes are named elements of data of particular structure. To describe the structure of the attribute content, SciDM data definition system is used. It defines a few atomic *DATATYPE*s and rules of their combination into the complex ones.

Enforcing the DATATYPE system on the attribute content serves the goal of keeping user-defined data structure through the storage/retrieval cycle. When the attribute content is obtained from the server, it retains its structure in the form of CORBA representation of data elements, structures and arrays (possibly nested). ORB automatically maps CORBA data structures to constructs natural for the programming language being used. These constructs can immediately be used in a way natural for a programming environment.

Along with translating structural representation of data into constructs natural for a programming language, storage of structured attributes allows to introduce methods for meaningful operations on the attribute content on the server. This specifically applies to situations when an attribute stores arrays. Such arrays may potentially be very long, and retrieving or saving the entire content could be undesirable. Server-side knowledge on the structure allows implementation of

operations on array slices, such as retrieval, replacement, insertion or deletion of a slice. This is not done in the current version of SciDM, but is planned for the future versions.

### *Integrated Support for Object Sets*

Analyzing and structuring large amounts of data always involves operations on object sets. The ability to select a set of objects, make it persistent, retrieve saved sets, compare sets or perform Boolean operations on them is critical for many data processing tasks. Because of very common nature of such operations, SciDM provides methods for object set manipulation and persistence on the server. For every user-defined type, the SciDM schema compiler creates a hierarchy of *Set types*. An object of a set type of the first level is capable of holding a collection of IDs of objects of an underlying type. An object of a set type of the second level can hold a collection of IDs of the above-described 'set' objects. This chain continues up to a certain *Set Nesting* maximum (64 in the current design).

The Set type is a legitimate type with methods for creating, destroying and enumerating the set objects. The Set object is a legitimate object that contains methods for manipulating the set contents (adding, removing and retrieving contained IDs) and also can contain any number of dynamic attributes. The Set types are named after their primary type with the addition of nesting level in curly braces (thus, type 'set of (Employee)' would have the name "Employee{1}"; 'set of (sets of (Employee))' would have the name "Employee{2}", and so on).

Along with persistent sets, SciDM supports methods for manipulating with temporary sets of IDs – through the *IdSetManager* component. This component serves the need of manipulating on the object ID sets right on the server without transferring them to clients. It is particularly useful when combining results of multiple search requests and also when the actual IDs are not needed on the client (only the content is needed). The IdSetManager provides methods for creating and destroying temporary ID sets and for performing Boolean operations on them. It is integrated with types, persistent sets and with bulk operations in a way so that the IDs retrieved or used by these operations can be stored in (or taken from) the ID sets instead of transferring to and from the client.

# Architecture

### *Life Cycle*

The life cycle of a particular incarnation of SciDM server includes the following steps:

- Schema Definition
- Building of the Server
- Starting the Server
- Working with Server

Once defined, the schema does not freeze forever. When a change is needed, appropriate changes should be made to the schema, and the server should be built again. After that, the old server should be stopped, the data update utility should run to modify the data according to the new schema, and the new server should be started on the updated data. For simple schema modifications, which include adding and deleing an entire type, the data update utility is getting

built automatically together with the new server. Modification of the existing types is not directly supported (though possible).

## Data Definition, Server Build and Startup

The *Schema definition file* is a python module. It contains definitions for both DATATYPEs and Types. Please consult the implementation manual for detailed instructions on how to write the schema definition.

The schema is compiled by the schema compiler based on the C@™ generative technology, and transforms into two types of files: IDL scripts and C++ sources with server implementation. The build process compiles the IDL and C++ code and builds the SciDM server executable. Upon startup, the server executable searches the current directory for a dataset structured as defined in its schema. If it does not find one, it creates the dataset *de novo*. It also creates a file with the server's IOR that is later used by the clients to establish the CORBA connection.

## Data Use: Root, Session and Component Subsystems

After the server has started, it is ready to accept the client connections. The clients connect to the server by connecting to an object referred by IOR. This is a root object, whose only capability is to authenticate the users and issue the *Session* objects to them. The *Session* object serves as a factory for all other objects the user may request. The set of these other objects depends on the Session and gets destroyed when the Session terminates.

The Session provides factory methods for creating instances of the component subsystems. Currently there are four such subsystems: '*Type'*, '*IdSetMgr'*, '*LockMgr'* and '*RightsMgr'*. The *Type* subsystem provides basic data access and reflection functionality for persistent data. The *IdSetMgr* provides methods for server-side operations on sets of object IDs. The *LockMgr* is the interface for user-level control on sets of locked objects. The *RightsMgr* is the interface for manipulation with object protection settings. The instance of a subsystem is obtained from the Session by invoking the Session's '*open'* method with the subsystem's name as a parameter. The Session may open many instances of '*Type*' simultaneously; for the rest of the subsystems, subsequent calls to '*open*' would return the same instance of the component.

Along with the '*open'* method, the session also exports series of methods for the lifecycle controlling. These are '*setTimeOut'*, '*setPingCallback'* and '*pong'*. The '*setTimeOut'* sets a time interval after which the server considers the session idle and closes it. The '*setPingCallback*' registers with the server the address of the remote *PingCallback* interface. This interface contains a single method named ping. The server uses this interface for discovering dead clients: if no client shows no activity for a timeout period, the server calls the ping method of registered callback and expects to get the pong called within the timeout. If this does not happen, the client considered being dead and the session gets closed.

## Objects and Types

The Type component subsystem provides methods for managing persistent data storage, including data access and manipulation methods, as well as reflection.

### Identifiers

SciDM generates an ID for an object at the time of the object's creation. The object ID uniquely identifies every persistent object within the instance of SciDM. The object ID is never reused: after the object is destroyed, the ID is forgotten forever. The object ID is an opaque piece of data, and in the current implementation of SciDM it is represented as a 64-bit integer. In the future versions, representation of OID may change.

### Accessors

The client accesses persistent data objects through the *Accessor* interface. The accessors are typed, which means a certain accessor may be used to access objects of a certain type (thus possessing the same set of methods). An object returned by a Session's *open* method called with parameter 'Type' is an accessor to an object of type 'Type'. Accessors to other objects are created through invoking the *openAccessor* method of the corresponding type.

Accessors are capable of the following:

- **accessing an object** by invoking the '*access*' method with ObjectID as parameter. The *access* method notifies the client about a failure by throwing an exception. The access may fail because an object for a given ID does not exist, the user does not have the access rights to the object, or the object ID is of an incorrect type (does not match the accessor's type). Repeated access calls with the same ObjectID revalidate the accessor. This operation may fail if the object was deleted, or if the access rights have been revoked since the last call.
- **checking validity** of an accessed object (method '*valid*'). An accessor is invalid right after the creation by *openAccessor* invocation, or if an object gets deleted while the accessor points to it, or if the user's access rights for an accessed object get revoked.
- **retrieving an ID** of a currently accessed object. This is done through the '*current*' method, which returns OID. This method may throw an exception if called on an invalid accessor.
- **creating a copy** of an accessor, pointing to the same object. This is done by calling the '*clone*' method. After the cloning, the accessors operate independently.
- **destroying itself** by invocation of the '*close*' method.

### Object Structure: Static and Dynamic Attributes. Annotatables

An interface for an object of any particular type inherits from the *Annotatable* interface, which, in turn, inherits from the *Accessor* interface. The *Annotatable* interface contains methods for manipulating dynamic attributes. Every object can contain any number of dynamic attributes. Dynamic attributes are distinguishable by their names. The values of dynamic attributes could be of the following types: *OID*, *LongLong*, *Float*, *DateTime*, *Char8* and *Octet8*. The former three are primitive types: *Object ID*, a double 64-bit integer and a double-precision floating point respectively. The latter three are complex types: a structure representing date and time, a 8-character string and an 8-byte array.

Each dynamic attribute, along with the name and the value, is associated with an *attribute number*. The attributes can be accessed either by their names or by their numbers. The *Annotatable* interface contains a method for retrieving and storing the attributes, enumerating the attributes, checking the attribute presence, obtaining the attribute DATATYPE and removing the attributes.

The most primitive Type, implicitly defined in any instance of the SciDM server, is the type *Dictionary*. An object of the type *Dictionary* contains no static attributes, so the dictionary is a pure incarnation of the *Annotatable*. Another implicit type is *Text*. This type contains a single static

attribute – text, along with the access methods *getText* and *setText*. Data stored in the *Text* attribute is a character array of an undefined length (not a null-terminated string).

The *Annotatable* interface contains distinct methods for enumerating dynamic and static attributes. The DATATYPE retrieval methods work on both dynamic and static attributes.

Along with these methods for typed access to the attributes, the *Annotatatable* defined methods for *stringified* (untyped) *access* – '*getAsString'* and '*setFromString'*. These methods also work on both static and dynamic attributes.

### Types as Object Managers. *Type* Methods

Object accessors allow manipulation on the object content. Methods for operating with objects themselves, such as object creation, destruction, and enumeration, are available through the *Type* interface. The *Type* interface also provides methods for accessor creation, navigation over the *Type* system, and reflection of the *Type's* object attributes, both static and dynamic. Methods of the *Type* interface can be subdivided into the following groups:

**Object Manipulation:**

**object creation methods**: '*create*', '*createRange*', and '*createIdSet*'. The '*create*' method just creates a new object and returns its ID. The '*createRange*' is a bulk version of 'create' – it creates a number of objects at a time and returns a range (an array) of IDs to the client. The '*createIdSet*', like '*createRange*', creates a number of objects, but sends the result to another destination – instead of passing an array of the new IDs to the client, it stores them in the *IdSetManager* as a transient ID set, and only passes a handle for this set to the client. The latter method is particularly useful when inserting ranges of new data into the database: the client does not have to know the IDs of newly created objects; however, they are needed for data update operations that come after creation. The *IdSetManager* keeps these IDs until the client sends in the actual bulk of data for the newly created objects.

**object destruction methods**: *destroy*, *destroyRange*, *destroyIdSet*. The first method deletes an object which ID has been passed as a parameter, from the persistent storage. The second deletes all objects designated by the passed array of IDs. The third deletes all objects designated by the IDs stored in the transient IDSet; the ID for this IDSet is passed to this method as a parameter.

**object enumeration methods**: '*rewind*', '*getNext'*, '*getNextN*' and '*getNextIntoIdSet*'. Enumeration is performed in the style of unidirectional iteration. This is a state-full operation: the *Type* object remembers the last object obtained during the iteration and is capable of returning the next one. The '*rewind'* method positions an internal pointer at the 'first' object in the type. After the rewinding, subsequent calls to '*getNext'* are guaranteed to return the entire content of the type. The '*getNext'* returns an ID of the next object and advances the internal pointer by one position; the '*getNextN'* returns a number (passed as a parameter) of IDs as a range, and advances the internal pointer by this number. The '*getNextNIntoIdSet'* stores a number of IDs in the transient set kept by the *IdSetManager*. It also advances the pointer and returns the number of objects retrieved. When the type exhausts, the '*getNext*' signals by returning a special value of *INVALID_ID*. The getNextN returns an empty array of IDs, and the '*getNextNIntoIdSet*' returns zero as the number of retrieved objects.

**container methods:** '*getSize*' and '*contains*'. The '*getSize*' method returns the number of the objects of a Type defined in the server's Schema. The '*contains*' method checks whether the passed ID corresponds to a valid and alive object of this Type.

**Reflection:** A set of reflection methods allows obtaining a list of static attributes for a type, properties for each of the attributes, and information on the indices available for the type. In

addition to that, the *Type* reflection allows obtaining aggregate information on the dynamic attributes used by objects of this type, including their full list and the numbers of objects using them. The reflection will be discussed later in more details.

**Context Search:** the context search methods allow the selection of objects that contain the query words in their *text* attribute. Two flavors of the context search are available: those passing the resulting ID set to client (*contextSearch*), and those storing the results in a transient set managed by the *IdSetMgr* (*contextSearchIdSet*). In addition to that, the method is available for counting the objects matching the search request (*contextCount*).

**Typesystem Navigation:** two methods are available for testing the position of the current type in the Set types hierarchy: '*getBaseType'*, returning the base (non-set) type, and '*getNestLevel'*, returning the set nesting level of the current type. For the base type itself, the '*getBaseType'* would return its own ID (the same as would be returned by '*current'*), and '*getNestLevel'* would return zero.

Another group of methods allows obtaining the type for an arbitrary object ID, obtaining the type ID for a set of objects of a given type, and obtaining the type IDs for the elements of a set of a given type.

**Datatype Reflection:** this group of methods allows obtaining the list of all *Datatypes* defined in the system and discovering the structure of each individual *datatype*.

### Types as Objects

The *Types* are available to the client through the Type accessors, and thus are representing the objects of the type '*Type'*. Each type has a unique Type ID and carries the same set of methods as any other type. The Type accessors possess all properties of any accessor: they can be tuned to any Type object by using the Type ID, they can return the ID of a currently accessed type, etc. In addition to the access-by-ID method, the Type accessor presents one more: '*accessByName'*. Each type has a unique name, defined in the database schema. The '*accessByName'* method tunes a Type accessor into a named type. The name of the current type may be obtained by calling '*getName'* method. (please note that there is no corresponding set method: a type cannot be renamed). Calling '*accessByName'* ('Type') positions the accessor at the Type object '*Type'*, which is a container of all types in the system, including itself.

The creation and destruction methods are not operational for the Type object '*Type'*, because SciDM does not support dynamic type creation and destruction. The rest of the methods in the *type* interface, however, are operational, including the object enumeration ones. This allows discovery of the list of available types by the client.

### Implicit Set Types

With a definition of the object type in the database schema, the hierarchy of the set types is getting defined implicitly. Set types are named after their basic type, with the addition of the set nesting level in the curly braces. The type "set of Text" would have a name '*Text{1}*' (nest level 1). The type "set of sets of Text" would have a name '*Text{2}'.* The maximal nesting level currently supported by the SciDM is 64.

While enumerating objects of the type '*Type'*, only non-empty set types appear. Empty Set types are not listed to avoid unnecessary crowding. Such 'hidden' empty Set types are accessible through invocation of the '*accessByName'* method on a Type accessor: *typeAcc.accessByName ("BaseType{N}")*

An object of the Set type is a collection of object IDs. The sets are homogenous, so that only objects of the same type can constitute a set. The type for the objects contained in a set is named *set element type.* A Set object provides methods for manipulating on the set contents. These methods include: checking the set size; adding object IDs to the set; retrieving the set content; checking for the presence of an ID in the set; removing elements from the set. The method for obtaining the type of a set element is also available in the set interface. Set content operations are available in by-element and aggregate forms with the use of both client-side ID arrays and server side persistent ID sets.

Because the set object is a normal object whose interface inherits *Annotatable*, it can be supplied with any set of dynamic attributes.

## *Attributes*

The object signature, which is determined by the schema, consists of the methods. It is useful, however, to present the object contents in terms of attributes. An attribute is a named data entity accessible through *set* and *get* methods. The Database schema is formed in terms of the attributes rather than methods. Each attribute name is associated with a unique attribute ID. The attribute ID binds the attribute name with a particular DATATYPE of the attribute's value. Attributes of the same name within a type always share the same DATATYPE for their values.

### Static Attributes

Each type defines a list of static attributes, which appear in every object of the type. They are accessible through attribute-specific static methods. The static attributes can be read-only, then only '*get[AttributeName]*' method would appear in the object interface. It could be write-only, then only '*set[AttributeName]*' method appears, or read-write, in which case both set and get methods would be present. The access rules for the attributes (WRONLY, RDONLY or RDWR) are defined by the schema.

The list of static attributes is available in both *Type* and *Object* interfaces through the use of '*getStaticAttributes*' method.

### Dynamic Attributes

Along with the type-specific static attributes, any object can contain an arbitrary list of dynamic attributes. Methods for manipulating the dynamic attributes are inherited from the *Annotatable* interface. These methods, in contrary to the static attribute access ones, are dynamic and not bound to a particular attribute name. Instead, the attribute name (or attribute ID) is passed to them as a parameter.

The dynamic attributes could contain data only of a certain DATATYPEs. They include: ObjectID, LongLong Integer, Float, DateTime, Char8 and Octet8 DATATYPEs. Because CORBA does not allow polymorphic methods, for every DATATYPE there are specific access methods. They include:

| | |
|---|---|
| get-by-name methods: | *getObject, getInteger, getFloat, getDatTime, getChar8, getOctet8;* |
| set-by-name methods: | *setObject, setInteger, setFloat, setDatTime, setChar8, setOctet8*; |
| get_by_id methods: | *getObjectById, getIntegerById, getFloatById, getDatTimeById, getChar8ById, getOctet8ById*; |

set_by_id methods: *setObjectById, setIntegerById, setFloatById, setDatTimeById, setChar8ById, setOctet8ById*

Along with the accessing, the dynamic attributes may de destroyed by using '*removeAttribute*' or '*removeAllAttributes*' methods; checked for presence by using '*hasAttribute*' method; listed by using '*getDynamicAttributes'* method. The DATATYPE for the attribute value may be obtained by using '*getAttributeDatatypeByName'* and '*getAttributeDatatype'* methods.

### Custom Methods

The database schema contains a construct for declaring custom methods not tied to any particular attribute. There is, however, no standard way to define a method itself – it must be coded explicitly in C++, using the internal server data exchange and object access conventions. The body of the method is placed in a special location and is later picked up by the server build process. Currently there is no standard API for writing method extensions; however, it may be defined in the future. The custom methods are particularly useful when several attributes are interdependent and have to be kept synchronous.

## *DATATYPE Definition*

The DATATYPE definition system offered by SciDM allows the construction of custom DATATYPEs out of a number of primitive ones.

### Primitive Types

The atomic DATATYPEs used in SciDM are:

- character;

- octet (8-bit value);

- short (16-bit), long (32-bit) and longlong (64-bit) integer numbers;

- real (double precision floating point) number;

- Object identifier

Along with an atomic DATATYPE, the *Enumerations* can be defined and used in a complex type construction as well as directly as an attribute DATATYPE.

### Constructors for Complex Types

The rules for generating a new DATATYPE out of an existing one include:

- **aliasing** (renaming). Any DATATYPE may be given a new name.

- **array construction**. Any DATATYPE may be used as an element of an array; the size of the array may be fixed or variable.

- **structure construction**. The existing types may be put in a 'structure', which is a list of the named values. The size of the structure is arbitrary; the names of the structure elements must be unique within the structure.

A DATATYPE of any complexity can be constructed by recurrent application of the above rules. Many of the widely used DATATYPEs are the complex ones: for instance, text is an array of characters of an undefined size.

## Bulk Operations

The bulk operations allow storage and retrieval of mass amounts of data in a single transaction. The *Bulk* is defined as a data structure containing arrays of selected attributes for a selected set of objects. It can be viewed as a matrix, with the columns corresponding to the attributes and rows - to the objects. The *bulk* may contain all attributes for a set of objects, some attributes, or even a single attribute. It may also contain attributes for a single object, many objects, or even all objects of a type. The size of the *bulk* in practice is limited by the network and the operating system. A practical size that does not seem to cause trouble on conventional systems is up to a few megabytes.

There are two major bulk operations: '*setBulk'* and '*getBulk*'. Both exist in two flavors: those passing object IDs between the server and the client, and those using the *IdSetManager* for the ID storage. The latter ones are named '*getBulkIdSet'* and '*setBulkIdSet*', respectively. The bulk access methods are members of the *Object Accessor* interface (despite the fact that they are not tied to a particular object).

The list of attributes passed or retrieved in the bulk operation is controlled through the mask parameter. The mask is an array of bytes where each byte corresponds to an attribute; setting it to 0 or 1 disables or enables the presence of the attribute in the bulk.

The *Set* accessor also provides bulk operations. The set's *bulk* is an array of arrays of the object IDs. The Set's '*getBulk*' and '*getBulkIdSet'* operations extract contents of the sets identified by the passed IDs. The Set's '*setBulk'* and '*setBulkIdSet*' fill the passed sets with data.

## Text-Mode Operations

In addition to the typed attribute access, the generic methods are provided for manipulating attributes in the form of ACSII strings. The string-style access is slower because the server does the conversion to and from ASCII representation. Using this style reduces the development effort for applications that do not require extreme operation speed, like data browsers or other UI tools. These methods are especially useful for generic tools intended for the use on various unspecific data layouts.

The *Annotatable* interface presents two such methods: '*getAsString'* and *'setFromString*'. An attribute name is passed to these methods as a parameter. They are operational on both dynamic and static attributes. To encode complex attribute structures or to specify the type of the argument where it is needed, a special syntax of tagged brace expressions are used.

Currently there are no string-style select methods, but they may appear in the future versions.

## Indexed Access:

The indexed access to the data is delivered through two types of *select* methods: select by attribute content, and context selection. The indices for each type are declared in the database schema.

### Attribute Indices and *selectBy* Methods

Any fixed-length attribute or their combination can be used for the content indexing. The content index allows selection of the objects where a given set of attributes fully or partially matches the query. The server *build* process generates appropriate selection methods for every declared index. These methods are available in the object *Accessor* interface. The result of the invocation of a selection method is a set of IDs of objects matching the query. This set may be empty if no matching objects exist.

Each index declared in the database schema consists of the index name and the list of attributes by which the index is build. Selection methods are named after the index names. The arguments for the selection methods are lists of values for the indexed attributes. The syntax is defined to distinguish full and partial matches.

The list of object IDs obtained as a result of the indexed search may be either passed to the client, or retained on the server within the transient ID set, managed by *IdSetManager*. For each index, two *select* methods are generated for these two different destinations*:* *'selectBy[IndexName]'* and '*selectIdSetBy[IndexName]*'.

The indices are maintained through all operations with the server, and are updated for every change in the data. This provides the user with the convenience of continuous data consistency for a slight performance price.

### Context Indices and *contextSearch*/*contextCount* Methods

The context index is available for the text fields. It allows selection of the objects that contain in the *given* attribute the words fully or partially matching the query. The context search methods are available through the *Type* Accessor (unlike the content selection methods, available through the *Object* Accessor). The fields that are contextually indexed are declared in the database schema. The context index can be maintained only on the attributes of the DATATYPE 'text'.

There are two methods for the context search: '*contextSearch*' and '*contextSearchIdSet*'. The former passes a list of IDs of the objects that match the search criteria to the client; the second one stores them in the transient ID set on the server. Particular attributes on which the selection should be performed are passed to these methods as parameters.

In addition to the context search methods, the '*contextCount'* provides a way to obtain the number of objects matching a specific context query without retrieving their IDs.

For every operation that involves a contextually indexed attribute, the index gets updated. This operation may be costly, especially when there is a need for storing relatively long text fragments. To allow the user flexibility to update or not the context index for a particular attribute, the context indexing control methods are available through the *Session* interface. These methods allow disabling or enabling the context indexing for the attributes of particular types. The context settings are session-wide.

## *Reflection*

The name 'Reflection' refers to a set of methods available in the various interfaces that allow SciDM client to discover the structure of data stored on a particular SciDM server. The reflection is useful for applications not tied to some particular schema, like generic data browsers or editors. The reflection methods of SciDM allow obtaining lists of types, lists of objects for every type, lists of attributes and indices for every type, lists of static and dynamic attributes for every

object, as well as properties of each individual attribute including its read/write state and the DATATYPE. It also allows obtaining the list of DATATYPE and the structure of every DATATYPE. The reflection methods are spread over various interfaces of SciDM. All of them were discussed in the above sections. Here is a brief summary:

| **To obtain** | **Use interface** | **Use methods** |
|---|---|---|
| List of types | Type Accessor (tuned to type 'Type') | *rewind / getNext(N)* |
| List of objects for a type | Type Accessor | *rewind / getNext(N)* |
| List of type's static attributes | Type Accessor | *getStaticAttributes; getAttributeName* |
| List of dynamic attributes used by a type's objects | Type Accessor | *getDynamciAttributes* |
| Attribute properties for type X | Type Accessor | *getAttributeInfo* |
| Attribute datatype for type X | Type Accessor | *getAttributeDatatype* |
| Indices for type X | Type Accessor | *getIndices, getIndexName* |
| Index structure | Type Accessor | *getIndexFields, getIndexInfo* |
| Dynamic attribute list for an object | Object Accessor | *getDynamicAttributes, getAttributeName* |
| Static attribute list for an object | Object Accessor | *getStaticAttributes, getAttributeName* |
| Attribute datatype for an object | Object Accessor | *getAttributeDatatype, getAttributeDatatypeByName* |
| All datatypes in the server instance | Type Accessor | *getAllDatatypes* |
| Datatype structure | Type Accessor | *unwrapDatatype* |

## Data Integrity

SciDM is designed in the assumption that rare events of hardware or other failures are not a sufficient condition for a continuous transaction integrity support. There are, however, two practical cases where the integrity should be considered.

First is the case of simultaneous modification of the same data by two or more clients. This may confuse the client applications, thus leading to a loss of data correctness. The assumption that particular pieces of data do not change between two sequential write events needs an enforcement. Such enforcement is provided by the *lock* mechanism.

Second is the case of object subordination. The situation when the life cycle of some object is tightly bound to the life cycle of a master object is considerably frequent. This especially applies if a certain set of objects is tightly bound to a master one. To handle such situations, many DBMSes use nested tables or similar constructs. SciDM offers the use of subordinate persistent sets.

### Locking

The Lock Manager (*LockMgr)* component subsystem provides a mechanism for locking sets of objects. While locked, an object can be modified only by the session that imposed the lock. Other sessions may access the object or read its content according to their rights, but cannot alter or destroy it. An alteration attempt results in the *AccessViolation* exception. To avoid the deadlocks, only one object set may be locked by the session at a time. Other objects may be locked only after the unlock operation is invoked. The unlock operation releases all objects in the currently locked set. Locking and unlocking is performed through the use of *LockMgr's* '*lock*' and '*unlock*' methods.

An attempt to lock a set of objects containing already locked ones would result in the *AccessViolation* exception. The *LockMgr* also offers the alternate lock method called '*lockSync*', which waits until all requested objects become available for locking or until the expiration of the timeout.

The LockManager imposes a lock on a heterogeneous set of Object IDs. This heterogeneous set must be stored under the session's *IdSetManager*. A special method of *IdSetManager*, '*createHetero*', is used for creation of heterogeneous ID sets (which are homogenous in all other cases).

### Subordinate Objects

The subordinate object is the object whose life cycle is bound to the life cycle of a master object. The subordinate objects are created automatically when the master objects are created and destroyed when the master objects are destroyed. The subordinate objects may not be destroyed separately. An attempt to call 'destroy' method on the ID of a subordinate object results in the *AccessViolation* exception.

Subordinate objects are declared in the database schema by raising the "SUBORDINATE" flag for an attribute referring to an object of a certain type. If this flag is raised, the following series of actions are triggered when a new object is created: the object of the type specified as the attribute's DATATYPE is created and marked as subordinate; this object's ID is getting filled into the attribute's value. This object exists while the master object is alive. The subordinate attributes are immutable (read-only) – there are no set methods for them.

There is no limit on the subordination nesting; recursive subordination, however, is not allowed for the obvious reasons.

The subordinate nature of an attribute is reflected in the value returned by '*getAttributeInfo'*, which lists all schema flags associated with the attribute.

## *Protection and Access Rights*

SciDM utilizes a concept of "user" to control the access to the data. The system may have any number of users. For every object in the system, the access rights can be assigned for each user individually. When an object is created, the rights for this object are assigned according to the special set of rights possessed by its creator, called the creation domain.

### Right Types

The following types of rights are supported in SciDM:

**creation right** – allows the creation of the objects, applicable to the types;

**read right** – allows reading the object attribute values; when applied to the types, allows enumeration of the type contents;

**write right** – allows changing the content of the object attributes;

**delete right** – allows destruction of the objects;

**seek right** – allows enumeration of the object attributes or type attributes

**change right –** allows changing the rights on the object (analogous to full control)

There are also the two special rights, called **ImpersonateUser** and **InstantiateComponent.** They allow establishing sessions without a password (immediately after the user creation) and to instantiate the component subsystem respectively.

For every possible pair (user, object) in the system the right of each type is either granted or not. An array of rights containing an entry for every right type is called the *rights mask*. Each element of the rights mask can hold one of the three values: granted, revoked or inherited from the parent user.

### Users

Each user in the SciDM system is represented by an object of the type 'User'. When the SciDM server starts for the first time, it automatically creates the user 'sys'. The user 'sys' is created with all possible access rights on all objects at the server startup. It also has a full set of rights on all objects it creates.

The users are created through the 'User' type as usual objects. The users form the tree-like hierarchy that affects the object right assessment. Each user has a list of 'children' and a single 'parent'. The access rights on every object are inherited from the parents to the children. For a child object, some or all of the parent's rights may be overwritten. Such overwriting affects all the children of the user and their children as well, up to the leave nodes of the user tree.

### RightsManager

The *RightsManager* component provides methods for reading, checking and changing the rights of the groups of users on the groups of objects. It has the following methods:

***get***: reads the range of bit masks representing the rights assignments for the given user and the given set of objects. The granted right (whether inherited or explicit) is represented by bit 1; the revoked - by 0 in the appropriate position of the mask.

***set***: assigns certain rights for a set of users and a set of objects. The values for the rights are passed in the rights mask; the set of right types to assign are passed in the other mask called transparency mask. Only the right types for which the transparency mask has the bits set to 1 are assigned; the rest retain their old values.

***reset***: resets the right types marked in the passed transparency mask to the 'inherited from parent' values.

***check***: checks whether the rights passed in the mask are granted on the given set of objects for the current user.

## *Server-Side Operations on Object Sets*

The *IdSetManager* subsystem of SciDM provides support for server-side operations on sets of object IDs (*IdSets*). The data retained in the *IdSetManager* is not persistent and exists only within the context of the current session. The use of *IdSetManager* allows for avoiding unnecessary transfer of arrays of object IDs between the server and the client when they are used only in the server–side operations. It also releases the client from the responsibility of performing the Boolean operations on object ID sets.

Each IdSet is a set in the STL sense – it contains each ID only once. The uniqueness of the IDs within the set is guaranteed by the *IdSetManager* and can not be violated by any operation.

The *IdSetManager* can handle multiple ID sets at once. Each IdSet has a unique identifier that is generated at the time of the IdSet creation.

### Methods

The *IdSetManager* interface contains methods for creation and destruction of the IdSets, including one for creating special heterogeneous IdSets used in the locking and rights assessment; content access and modification methods, including ones for individual elements, for fractions (slices) and for the entire content of the IdSet, and methods for performing Boolean operations AND, OR, XOR and SUB on the IdSets. All these methods in the *IdSetManager* interface accept at least one parameter - the IdSet identifier. The Boolean operations accept two SetId operands. All Boolean operations place the result in a new IdSet, leaving the operands intact.

### Mass Operations Integration

The *IdSetManager* is tightly integrated with other components of SciDM. Wherever an operation involves an array of object IDs, two versions of the method are provided: one using a client-side array of IDs and one using a server-side IdSet. The *RightsManager* and *LockManager* use the IdSets only as a source for object ID lists. All methods using the IdSets are described above. Here we provide just a brief list of them:

- enumeration, creation, destruction of objects and context search methods of the Type Accessor
- content retrieval and manipulation methods and bulk methods of the Set Accessor
- bulk methods of the Object Accessors
- locking methods of Lock Manager
- rights assessment methods of Rights Manager