Long-Term Database Support for EXPRESS Data

Axel Herbst*
IBM Heidelberg Scientific Center
Vangerowstr. 18
D-69115 Heidelberg

Abstract

The data modeling language STEP/EXPRESS enjoys a spreading popularity beyond engineering. Its structural object orientation, for instance, is appropriate for scientific and statistical database (SSDB) modeling as well. As our database prototype illustrates, EXPRESS can also be used for data definition. Following the STEP approach, data manipulation is done via SDAI (Standard Data Access Interface).

Besides increasing awareness of evolving STEP technology, this paper elaborates on a common issue of both engineering databases and SSDBs: the long-term retention of vast amounts of data. The storage of massive data over decades imposes serious problems on any DBMS. Application-oriented database archiving helps solve them. We indicate how to exploit and enhance SDAI to make it an archive interface.

1 Steps beyond STEP

STEP (Standard for the Exchange of Product Model Data, ISO 10303) is much more than yet another data exchange standard [19]. STEP is a unique and ambitious effort to overcome fundamental problems connected with proprietary data management in engineering. The development and use of the formal modeling language EXPRESS has been a novelty in this application domain. Various aspects of a technical product (e.g. the geometry of a cardan-shaft or the assembly of a car) are modeled in EXPRESS. STEP concerns itself with data from the early design process to the maintenance phase of a product. Data instances that conform to EXPRESS schemas can either be exchanged by STEP files or can be shared in STEP databases. It is evident that the emphasis will shift to data sharing in the near future. We are focusing on the corresponding database issues.

*also: University of Kaiserslautern, Computer Science Dept.

This paper, however, does not strictly deal with STEP. We neither refer to predefined product data models nor consider specific STEP semantics for information sharing. Instead, we simply borrow those parts from STEP which are independent of the originally-intended application domain and which are of interest for scientific and statistical database (SSDB) management [7, 21]. We are encouraged to do so by at least two facts. On the one hand, the first parts of STEP such as EXPRESS have finally become separate worldwide standards so that STEP compliant data management tools are advancing from prototypes to products [30]. On the other hand, applications beyond engineering are arising which make use of STEP methods and software. Examples include the modeling of genomic data and their manipulation in a database as part of a biodata information system [25].

Many domain-specific databases share a major problem: Data need to be stored over long periods of time, causing an enormous growth of the database. For instance, continually-acquired statistical micro (e.g. census) data enlarge the database, whereas deletions occur only rarely. Data collected by a geologist need to be kept unchanged to allow varying extractions and investigations. In engineering, having old product data accessible has both legal and economic reasons. Since we stress the data's reusability, we prefer a database approach to a file system one. The long retention times call for inexpensive storage media to be handled by the DBMS. Tertiary storage is appropriate when the DBMS knows which data are concerned. The DBMS can be informed about this if it provides an application-oriented archiving functionality.

The remainder of this paper is organized as follows. Section 2 concentrates on data definition with EXPRESS, whereas the corresponding data manipulation is introduced in section 3. Our EXPRESS database prototype is described in section 4. The long-term issue with focus on database archiving is the objective of section 5 followed by archive interface considerations in section 6. Section 7 draws conclusions.

2 Data modeling with EXPRESS

This section presents EXPRESS basics and selected features we find relevant to SSDBs. To familiarize also those who are language novices with EXPRESS's object flavor, relationship handling, and use of procedural elements, we give several examples.[1] We introduce EXPRESS not only as a conceptual modeling language but also as a data definition language (**DDL**).

2.1 Structural object orientation

Objects of the real (or hypothetical) world are represented by instances of strongly-typed *entities*. An entity describes objects with common properties. Properties are modeled by *attributes* and *constraints*:

```
ENTITY A;
 a1:          REAL(5);
 a2: OPTIONAL REAL;
 a3:          INTEGER;
 b:           ARRAY[1:100] OF LIST OF BAG OF NUMBER;
 c:           myType;
UNIQUE
 a3;
WHERE
 IF EXISTS(a2) THEN a1*a2 < 0.0; END_IF;
END_ENTITY;
```

example 1

The entity A has the attributes of selected data types:

- simple data type (for a1-a3; *NUMBER* combining reals and integers is also simple)
- collection data type called *aggregates* (for b)
- named data type (for c; *myType* denotes another entity or user-defined data type)

One of the fundamental principles of EXPRESS addressing object identity demands that entity instances be distinguished by some (non-visible) associative identifier. Deep and shallow equality of objects can be tested. In example 1, we additionally request the uniqueness of values for a3. Compound keys are also possible. This sort of optional value-based identification is heavily used in engineering applications and is appropriate for scientific and statistical data, too (e.g. index of experiments, time of measurements).

Another property of objects specified by example 1 is the following: if a2 is provided, either a1 or a2 must be negative. From a DDL point of view, valid A-instances do not violate the integrity constraint stated in the *WHERE* clause (see section 2.3).

[1] The official ISO document is [13]. A first text book on EXPRESS has been published only recently [24].

Inheritance is expressed by the *SUBTYPE* clause which lists all inheriting supertypes. Ambiguities due to polymorphism must be syntactically resolved.

```
ENTITY B
SUBTYPE OF (A);
 x:          BINARY;
 SELF\A.a1: REAL(10);
 SELF\A.a2: REAL;
WHERE
 a2 < 0.0;
END_ENTITY;
```

example 2

In example 2, entity B is a specialization of entity A. We added attribute x and redefined a1 and a2. Redefinition in EXPRESS is always refinement: Whereas a higher precision is required for a1, a2 is no longer allowed to be optional. Moreover, B may only contain instances with positive values for a1. This is because the original constraint on A still holds for B. Hence, an "is a" relationship is modeled by entity subtyping. Both attributes and constraints are inherited.

Actually, specifying sub- and supertypes is more subtle. For example 3, imagine that statisticians wanted to record information about SSDB conference participants. Suppose they insisted on treating male and female participants separately. Furthermore, database experts and domain scientists should be distinguished. The latter need not be an affront to people who claim to be both if EXPRESSed like this:

```
ENTITY participant
SUPERTYPE OF (ONEOF(male,female)
              AND  (DBexpert ANDOR domainScientist));
END_ENTITY;

ENTITY DBexpert
SUBTYPE OF (participant);
 DDLsOfInterest: SET OF STRING;
END_ENTITY;
```

example 3

The resulting subtype/supertype graph establishes a so-called *complex entity type*: the representation of a single real-world object is spread over more than one (simple) entity type. This idea must not be confused with the more common concept of a *complex object* formed by means of non-inheritance relationships.

2.2 Non-inheritance relationships

As outlined particularly in [5], non-inheritance relationships such as "is part of", "is owned by" or "is associated with" are essential for scientific databases. In EXPRESS, these relationships are modeled by attributes of entities.

```
ENTITY X;               ENTITY Y;
 refToY: Y;              INVERSE refToX: X FOR refToY;
END_ENTITY;             END_ENTITY;
```

example 4

In example 4, a one-to-one relationship is established between X and Y. It is up to the user to interpret whether every X "contains a" Y, "is associated with" it or holds some other semantics. EXPRESS does not require **inverse** attributes explicitly. If we had omitted *refToX* in entity Y, we would have established a bi-directional relationship anyway. Traversal from Y to X would not have been possible, though. Moreover, the cardinality of the reverse direction of the relationship (X $\leftarrow$ Y) would not have been constrained.

```
ENTITY experiment;
 consistsOf: OPTIONAL SET[1:50] OF run;
 INVERSE performedBy: SET[1:?] OF person FOR performs;
END_ENTITY;

ENTITY run;
 INVERSE which: experiment FOR consistsOf;
END_ENTITY;

ENTITY person;
 performs: OPTIONAL SET OF experiment;
END_ENTITY;
```

example 5

Example 5 is the EXPRESS version of a self-explanatory part of the scenario presented in [5]. We additionally constrained an *experiment* to be *run* at most 50 times. However, the example must lack the definition of behavior such as deletion semantics.

2.3 Use of functions and procedures

EXPRESS provides built-in functions as well as procedures and supports user-defined ones. However, there are only two intentions for their use:

- *exploitation in derived attributes*
  The values of these attributes are calculated when accessed (see example 6).
- *definition of sophisticated integrity constraints*

The *WHERE* clause of an entity declaration states an integrity constraint which must be satisfied by every single instance of that entity. In example 1, we used the built-in function *EXISTS()* to determine if the attribute a2 contained a value.

Besides the *local WHERE constraints*, there are *global WHERE rules*. They, too, must not be violated by a data set that claims to conform to the EXPRESS specification. For a better understanding, recall example 3. Imagine we wanted to make sure that at least as many database experts interested in EXPRESS attend the conference as determined by a user-defined function *MinNumber()*. The following rule needs to be added:

```
RULE EXPRESS_fans FOR (DBexpert);
WHERE
 MinNumber()                -- declared somewhere else
 <=  SIZEOF(QUERY(x <* DBexpert |
                 x.DDLsOfInterest='EXPRESS'))
END_RULE;
```

This rule applies to all instances of the entity *DBexpert*.[2] The *QUERY* expression returns those instances $x \in DBexpert$ which satisfy the predicate after the symbol |. The function *MinNumber()* can be constructed using common procedural language statements (e.g. *IF, CASE, REPEAT*) according to the EXPRESS grammar. Some arithmetical functions such as *ABS(), SQRT()*, or *LOG()* are built-in. Nevertheless, typical statistical functions which aggregate data must either be supplied by the modeler or be pushed into the standardization of *EXPRESS 2* [29].

Rules, user-defined functions, procedures, and data types are actually not "free-standing" as in the previous examples. They are constituents of an EXPRESS *schema*. Instead of going into further detail, we compile and extend some of the EXPRESS fragments already listed:

```
SCHEMA SSDBM;
 TYPE continent = ENUMERATION OF
      (Africa, America, Asia, Australia, Europe);
 END_TYPE;

 ENTITY participant
 SUPERTYPE OF (ONEOF(male,female)
               AND  (DBexpert ANDOR domainScientist));
  name:       STRING;
  origin:     continent;
  whichYears: SET OF INTEGER;
 DERIVE
  regular:    BOOLEAN := SIZEOF(whichYears) = 7;
 END_ENTITY;

 (* add male, female, DBexpert, and domainScientist *)

 RULE EXPRESS_fans FOR (DBexpert);
 (* see above *)
 END_RULE;

 FUNCTION MinNumber(): INTEGER;
  RETURN (* something smart *);
 END_FUNCTION;
END_SCHEMA;
```

example 6

[2] In general, a global rule can affect several entities.

Obviously, EXPRESS is both a **standardized** and **powerful data modeling** language. Its data structuring mechanisms make it suited for a wide range of applications even though EXPRESS lacks behavior modeling. If the system classification as introduced in [4] is applied to languages, EXPRESS is **structurally** object-oriented.

It is possible to consider an EXPRESS schema a conceptual schema for a **database**. However, this approach raises some problems, especially because of integrity constraints (scope, time of evaluation and enforcement, optimization). Things are even more difficult if EXPRESS has to be mapped to the native DDL of a conventional DBMS. In this case, preserving the EXPRESS semantics is quite difficult; just think of the poor data abstraction concepts of SQL. The full-fledged implementation of global rules, derived attributes, and the different inheritance relationships are challenges for object-oriented systems as well.

Data defined in EXPRESS need to be manipulated by application programs at the same level of abstraction. This is the objective of another part of STEP, namely SDAI (Standard Data Access Interface [14]).

3 Accessing and manipulating EXPRESS data via SDAI

3.1 Motivation for SDAI

Only recently has SDAI been discussed as a database **DML**. Initially, the still prevalent technique of exchanging files pushed the development of an **API** (application programming interface): The STEP file import and export, together with the checking and processing of the exchanged data, is done by so-called STEP processors typically attached to CAD systems. To reduce the implementation effort of these processors, the origin and encoding of the STEP data should be hidden; SDAI is supposed to relieve the application programmer from distinguishing whether the STEP data are accessed from main memory, from a file system, or even from a database.

3.2 SDAI operations

The abstract, logical data store for EXPRESS data is called *SDAI repository*.[3] Again, we keep a database implementation in mind, i.e. a DBMS that "understands" SDAI requests. Which operations on such a repository must be supported, then?

At first, we need to know that SDAI groups entity instances in so-called *SDAI models*. Instances of a model belong to a particular EXPRESS schema. Since a repository can host many schemas, the particular schema must be identified when a model is created. In the following example, we put a single instance in a new model based on the schema from example 6:

```
schemaName = 'SSDBM'
modelName  = 'participants_of_7SSDBM'
sampleModel = CREATE_MODEL
              (sampleRepository, modelName, schemaName)
sampleEntity = GET_ENTITY_DEFINITION
               (sampleModel, 'participant')

sampleInstance = CREATE_ENTITY_INSTANCE
                 (sampleEntity, sampleModel)
PUT_ATTRIBUTE (sampleInstance, 'name', 'Axel Herbst')
PUT_ATTRIBUTE (sampleInstance, 'origin', Europe)

sampleAggr = CREATE_AGGREGATE_INSTANCE
             (sampleInstance, 'whichYears')
ok = ADD (sampleAggr, 1994)
```

example 7

Actually, the SDAI operations are specified independently of a programming language. Language bindings are specified additionally (see section 4.2). As indicated by example 7, there are operations on models (e.g. *create, delete, rename*), entity instances (e.g. *copy, get_type, find_model, get_attribute*), and aggregates (e.g. *add, remove, indexed_get*). Nested aggregates such as in example 1 are supported, too. *Iterators* (cursors) allow for the access to single members. They are also used to scan all instances of an entity because there is the concept of an *entity extent*.

An SDAI implementation does not automatically enforce the EXPRESS integrity constraints. Rather, there are separate operations to validate uniqueness, rules, or even the presence of values for non-optional attributes. It is left to the application program to call them. This approach seems pragmatic but dangerous. Note that the procedural part of EXPRESS is Turing-complete although EXPRESS is not considered a programming language. Even the consequence of the failure of a validation operation (constraint checking) is undefined. We avoided speaking of **database transactions** deliberately but there is the term *transaction repository* in SDAI. Opening a *transaction repository*, executing some other SDAI operations and committing them may violate data consistency unless consistency (C) is defined differently from validity of integrity constraints. Atomicity (A) of SDAI transac-

[3] This term is subject to change late in 1994, and storage system related aspects of SDAI repositories will also be addressed. Throughout this paper, we refer to the document [14].

tions is not guaranteed, either [16]; neither is isolation (I), because concurrent access of multiple applications is outside the scope of [14]. In short, durability (D) is the only ACID property [11] of an SDAI transaction conceptually supported so far.

3.3 Metadata management

Unlike the common understanding of metadata in the SSDB community [8], SDAI metadata have nothing to do with application domain specifics. They are also "data about data" but from the database management point of view, which is equally accepted. We are introducing this issue for two reasons. The first is to show that SDAI is by far more than an API; the second is the metadata's importance for long-term database support.

Quite often, application programs are written and compiled at a time when not all data and data formats to be processed are known. Hence, flexibility is required because either the type information is not yet known or it is no longer present. Interactive users may ad-hoc query the DBMS catalog to familiarize themselves with the content of the database. As SDAI aims at encapsulating the underlying DBMS, a data dictionary is needed that keeps the EXPRESS application schema information. SDAI defines the structure of such a data dictionary by a special schema, which is also written in EXPRESS. We will present only pieces from the *SDAI data dictionary schema*:

```
ENTITY entity_definition SUBTYPE OF (named_type);
 supertypes       : LIST OF UNIQUE entity_definition;
 attributes       : LIST OF UNIQUE attribute;
 uniqueness_rules : SET OF uniqueness_rule;
 where_rules      : SET OF where_rule;
INVERSE
 parent_schema    : schema_definition FOR entities;
END_ENTITY;

ENTITY schema_definition;
 name             : STRING;
 entities         : SET OF entity_definition;
 types            : SET OF defined_type;
 global_rules     : SET OF global_rule;
UNIQUE
 UR1: name;
END_ENTITY;
```

The application schema in example 6 could be one instance of the entity *schema_definition*. Together with the referenced *entity_definition* instances (*participant*, *male* etc.), it forms an SDAI model of its own. A program can consult this model by applying the same SDAI read operations as for application instances. Thus, it can find out which entities are included in the application schema, can examine their attributes, and so on. All in all, the "real data" can be **interpreted by means of the metadata**; at least from the SDAI user's, i.e. application program's, standpoint.

Concepts such as SDAI repository, model, and entity extent are defined in another schema, namely the *SDAI session schema*. Again, these metadata are formalized by means of EXPRESS and meant to be managed by an SDAI implementation. An application program can therefore figure out which repositories are available and how to get to a single entity instance.

Altogether, SDAI needs fixing and extension (e.g. transaction concept, descriptive query language) to serve as a database interface. The main idea of SDAI, that is defining a basic and **EXPRESS adequate access** facility to a data store, regardless of its implementation, must be highly appreciated anyway. It is the well-tried database concept of **data independence**, which is acknowledged and adjusted to STEP and beyond by the SDAI designers.

4 An EXPRESS database prototype

This section applies the previously-selected STEP aspects to an implementation of an EXPRESS database. We will briefly describe a prototype scenario which we intend to exploit in future work on long-term archiving (see sections 5 and 6). Other EXPRESS-based tools covering research prototypes and first commercial products are listed in [30]. All implementations differ in their actual support of EXPRESS features, SDAI operations, and SDAI metadata. Because SDAI itself is still changing, especially the few persistent data management tools currently available have a preliminary character. However, we invite the SSDB community to evaluate and influence the upcoming systems according to their needs.

4.1 Architectural overview

Given that an **EXPRESS database** is a database which stores data defined by arbitrary EXPRESS schemas and provides full SDAI functionality, a "from scratch implementation" requires a lot of effort. Instead, we implemented and extended parts of the proposed SDAI C++ binding as a software layer on top of a commercial object-oriented DBMS. Our prototype is distinctive through the comprehensive SDAI metadata support in combination with both the early and late C++ binding option exploiting ObjectStore.

As sketched in figure 1, four components are involved in creating an EXPRESS database and running SDAI applications. Details can be studied in [20].

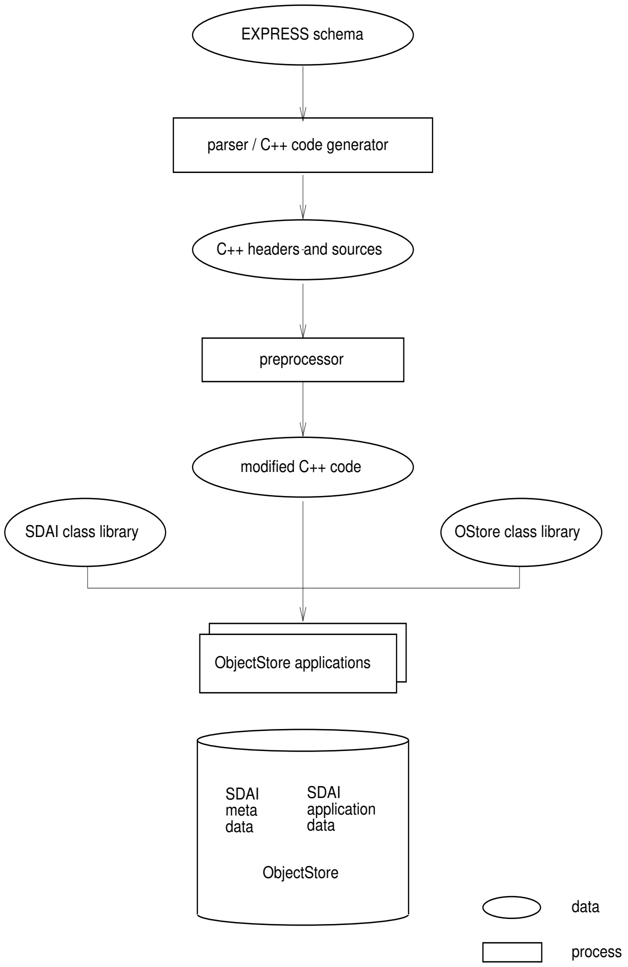

figure 1

1. *EXPRESS Parser*
   From [30], we have chosen the publicly available parser developed by the National Institute of Standards and Technology (NIST). This parser is coupled with a C++ class generator ("fedex+").

2. *Preprocessor*
   We developed a preprocessor to transform some of the generated C++ source code in order to better conform to the SDAI conventions. The preprocessor also assures that the entity instances will be persistently created at run-time, i.e. in the database and not just in main memory.

3. *SDAI class library*
   These C++ classes are independent of the EXPRESS application schema. They implement the SDAI dictionary and SDAI session schema. Example 8 shows parts of our class definition for an SDAI session which takes advantage of ObjectStore collections:

```
class SdaiSession : public SdaiSessionInstance {
  friend SdaiRepo;
protected:
  os_List<SdaiRepo*>* _known_servers;
         /* _servers denote SDAI repositories */
  os_Set<SdaiRepo*>   _active_servers;
  os_Set<SdaiModel*>  _active_models;
  SdaiRepo*           _dictionary_rep;

private:
  void AddActiveServer(SdaiRepoH&);
}

example 8
```

4. *ObjectStore*
   We decided to use this object-oriented DBMS because we found the following technical features well-suited for the C++ binding implementation:
   - seamless integration with C++
   - persistence orthogonal to type system
   - built-in support for inverse attributes
   - transaction management
     (We expect the introduction of ObjectStore-like transactions into SDAI soon)
   - queries on collections

4.2 Usage aspects of the C++ binding

There are also C and FORTRAN bindings for SDAI specified, but we follow the argumentation e.g. in [3] promoting an implementation based on C++ and an object-oriented DBMS. The C++ binding captures plenty of the EXPRESS semantics [22]. For example, the most prevalent form of EXPRESS inheritance maps smoothly to the C++ data model. Furthermore, operations on strongly-typed aggregates can be type-checked at compile-time. The compiler can also be enlisted to check (sub)object references.

According to [14], a **late binding** is an SDAI language binding which is independent of the application schema being accessed. Only for C++, an **early binding** is additionally defined. There are naming conventions e.g. for the mapping of EXPRESS entities and attributes to C++ classes and attribute-specific access functions. Thus the C++ user can choose between dealing with lots of efficient methods (early binding) or writing more generic but slower applications (late binding). In fact, we experienced a significant performance loss when the SDAI dictionary and session schema were being populated and maintained. However, this SDAI metadata support is a prerequisite for future data interpretation. Our further studies on long-term database support will make use of our prototype's metadata management.

5 Database support for long-term data retention

5.1 Problems, approaches, and related work

In general, ensuring the durability of changes to data caused by committed transactions becomes a challenging DBMS task when the database grows over very long periods of time. On the one hand, there is the **increasing amount** of data that requires reliable and economical storage over years and possibly decades. On the other hand, the data need to **remain usable** in the future. Both these physical and logical aspects are gaining more and more attention in product data management and, apparently, in scientific and statistical data management as well [8].

For example, the manufacturer's liability laws force CAD users to **archive** their designs so that drawings are still readable after 10 years and more. Also, spare part supply relies on old data. Finally, the reuse of data can speed up new product development. The storage of CAD models in native file formats, however, is inappropriate for long-term archiving because of the strong system, release, and version dependencies. (There are cases where even release *n* invalidates the native file format used in release *n-2* of the same CAD system!) File archiving in neutral formats (e.g. IGES, STEP) is not entirely satisfying, either [12]. The more data are stored in product databases, the more they are expected to be retained by a DBMS even in the long run and without loss of information. The latter requires that metadata (in the sense of section 3.3) be kept accessible as well. From an EXPRESS point of view, multiple application schemas need to be stored in the database to **interpret** the data instances. Furthermore, the language EXPRESS itself will experience amendments one has to care about.

Besides these logical issues, there are problems at the physical **storage layer** of a DBMS. First, the reliability of disks is limited. Disk mirroring and RAID (Redundant Array of Inexpensive Disks) compensate this to a certain extent. Of course, these methods cannot prolong the lifetime of a single disk. Secondly, disks remain expensive despite the falling prices and (slowly) increasing capacity. To illustrate a magnitude, Sequoia 2000 scientists who are studying Earth's global change would like to store about 100 TB of data [28]! Even with compression techniques, a pure disk-solution is not reasonable. Database compression [23] is an approach orthogonal to our focus.

Regarding disks, two extreme positions can be found in the literature [9]: The *disks-forever view* contradicts the *disks-are-dead view*. The latter predicts that electronic memory will be followed immediately by robotic storage in the **storage hierarchy**, which is traditionally modeled as a pyramid with little, expensive but fast storage at the pinnacle and large-capacity, cheap but high-latency storage at the base. In any case, **tertiary storage**, i.e. storage "below" secondary (magnetic disk) one, is of growing importance. Examples include optical disk jukeboxes and automated magnetic tape libraries. New media such as optical tapes will continue to turn up over time.

Nowadays, tertiary storage can hardly be met in DBMSs. Note that we think of tertiary devices as "first class citizens" [1], i.e. storage **under full control of a DBMS**, and not just for backup purposes. One reason for neglecting it is that DBMSs insist on random block access, which performs poorly on sequential devices such as tapes. The significant latency of tertiary storage necessitates larger I/O granulates. Diverse cost functions for query optimization need to be integrated in the DBMS. Managing WORM media requires completely different algorithms for (simulated) updating, clustering, and indexing. Nevertheless, the need for long-term storage of massive data and attractive cost per megabyte push the research on tertiary storage support also for DBMSs [1].

We see two basic questions here: How to **integrate** tertiary storage media into the DBMS and how to let the user **take advantage** of them. We favor that the first question be answered by DBMS extensibility as studied in STARBURST [10] or especially in the **device-extensible** version of POSTGRES [18]. For the latter, a new software layer between the data manager and the storage devices has been introduced. This so-called *storage manager switch* allows for the uniform incorporation of tertiary storage devices. Even though POSTGRES uses a special no-overwrite technique[4], the write-once characteristic of the WORM optical disk jukebox is not completely hidden and is subject to future work. Our second question (how to make the most of the available storage pool) takes the location of data into account. Obviously, not all data should be put on tertiary storage. There are basically three approaches to influence the data **placement** with trade-offs in efficiency and transparency:

1. *static placement*
   The user's data definition statements or data creation requests determine where the data shall be put. Symbolic parameters directing the location (e.g. "video_tape", "magneto-optical_disk") allow for a minimum of physical data independence.

[4] which is actually "seldom-overwrite" as explained in [18]

2. *automatic, system-controlled migration of data*
   This approach is a generalization of the proposal to archive file versions transparently on optical disks [15]. The shifting of data can depend on several criteria (usually age, frequency of access etc.) and can be done asynchronously (e.g. the "vacuum cleaner" for invalid records in POSTGRES [26]). Descriptive migration policies may also control the movement of data in a hierarchical (tertiary) storage management system.

3. *explicit data transfer operations issued by an application or by the database administrator (DBA)*
   This is our preference. We will finally introduce a specialization of this approach.

5.2 Database archiving

5.2.1 Basic characteristics

In many cases, database users[5] know when data are going to be accessed rarely or may not be updated anymore; they know which data can afford longer access times for the sake of storage economy. However, a user might not be willing to select various tertiary storage options. It should be left to the DBA (or the DBMS itself) to further optimize the location of the "less interesting" data. Apart from that, these data must typically be hidden until their reuse is explicitly desired. In short, we propose that a DBMS provides an **application-oriented archiving service**.

This kind of long-term support is motivated by but not restricted to CAD archiving as illustrated above. We found the idea of integrating such a user-controlled archiving component into a DBMS first in connection with the engineering database prototype PRODAT [6]. We agree with [6] in the following characteristics of advanced database archiving:

- *user consciousness*
  The user is offered archive operations. It is his or her deliberate decision to invoke them and to face the consequences (see below).

- *level of abstraction*
  The data to be archived are described in terms of the data model; different (but not arbitrary) data granularity should be supported. For example, applications based on a relational DBMS might choose to archive tuples or tables.

- *data movement*
  To conserve space, data are taken from limited expensive (disk) storage and are shifted to economical (tertiary) storage which is, additionally, more durable. At the same time, data can be compressed and can change representation (altered logical access paths or physical data stream layout) to adjust to the new device behavior [27].

Contrary to [6], we do **not** envision a **uniform view** of active and archived data as far as **visibility** is concerned. In PRODAT, one has navigational access to all database objects. Data referenced at run-time are transparently restored from the (on-line) archive. If there were any EXPRESS-like global integrity constraints in the data model of PRODAT, their scope would comprise all archived and active data together.

Rather, we see applications in which the archived data are explicitly excluded from the current work database (e.g. a finished design, a deferred analysis of satellite data). A less tight integration of active and archive data management also allows greater flexibility in handling metadata (schema) evolution. For a start, and roughly speaking, we propose to **move data** and to **copy metadata** during archiving. This is reasonable because metadata such as application schemas are less bulky and likely to be reused immediately.

Our intended **logical separation** of archived from active data does not prevent the user from applying the **same access operations** including queries to either kind of data. Performance, of course, will differ. We do not want to force the user to restore data from the archive. The redirecting of data from tertiary to secondary storage is only recommended when extensive reuse is going to happen or updates-in-place are requested but not supported by the archive device. Archive updates are controversially discussed. On the one hand, updates contradict most people's philosophy of an archive. On the other hand, even conventionally archived technical drawings undergo corrections provided that the changes are documented.

5.2.2 Traditional database archiving

It should be obvious now that our perception of application-oriented archiving differs fundamentally from the archiving **commonly available** in DBMSs. Usually, an *archive copy utility* is used to dump pages of a database for recovery purposes. Even novel methods for incremental and full archiving presented in [17] are based on this backup semantics. Besides, *log archiving* means that the database log files are kept in addition – preferably off-line in *electronic vaults* [9].

[5] Our (DBMS) perception of a user is simplified. Depending on the context, the DBA might be included. Usually, there is another (end-) user.

An archive command on a higher abstraction level, namely for the relational model, was suggested by [2]. When the DBA issues this command, all tuples of a relation shall be moved to some archive. With respect to our motivated long-term support, there are a few conceptual differences, e.g.: The identification of associated active and archive tables by name implies that the **structure** of the original table **must not change**; **over-writing** tables in either direction (i.e. during archiving or during the explicitly required "reactivating") is not very convenient.

However, some native DBMS features may help implement application-oriented archiving. They need to be studied in particular if the user's data model is a layer on top of the proprietary one as in the case of EXPRESS databases accessed via SDAI.

6 SDAI as archive interface

As shown in section 3, SDAI needs enhancements to serve as a database interface. Further extensions are necessary to make SDAI an archive interface.[6]

Before the SDAI user invokes a new *ARCHIVE(...)* operation, data have to be **selected**. Such an SDAI operation *SELECT(...)* must ensure that a valid data set is chosen. Details are beyond the scope of this paper. We restrict ourselves to refining the aspect of different **data granularities**. One can specify units of data with respect to the SDAI metadata definitions.

1. *According to the dictionary schema:*
   A schema and a (single or complex) entity are meaningful EXPRESS units for archiving. Selecting a set of entities cannot be done arbitrarily. Global rules, for example, might cover certain entities. Non-inheritance relationships could also stipulate that some entities be archived together.

2. *According to the session schema:*
   Instances making up an entity extent, an SDAI model, or a complete repository could be destined for archiving.

After selection, upon the *ARCHIVE* operation, the data are (physically) moved to tertiary storage. We suggest that this location (logically) establish another **SDAI repository**. Thus, SDAI operations do not touch the archived data anymore. Returning to the fragment of our session class from example 8, it is a straight-forward solution to let an archive be one of the *_known_servers*. Opening and accessing it just requires operations the SDAI user is familiar with.

As soon as tertiary storage enjoys full support by the underlying DBMS, the implementation of the member function *AddActiveServer* for usual repositories is basically the same as for archive ones. Hence, this approach fits well into the SDAI framework. It causes problems when archiving is extensively used. The management overhead to handle the *_known_servers* will increase; the identification of the repositories can no longer be burdened on the user.

Several times in this paper we have stressed that those metadata which are necessary for future interpretation of the application data should be archived as well. The structure of an **application schema** is represented by instances of the SDAI dictionary schema (see section 3.3). If the application schema evolves over time, the original schema can at least be "frozen" in the archive. (In our prototype, we exploit ObjectStore's versioning concept [20]). Although new schema versions are approved only rarely in STEP due to ISO regulations, keeping metadata alone seems to be too basic a support in general. This is one of the items for our future work.

What about changes to the **language EXPRESS** itself? If an EXPRESS schema were to include behavior specification, this would affect the definition of the SDAI dictionary schema. A further level of abstraction that holds a schema of which the SDAI **dictionary schema is an instance** would reflect these updates if such a support were introduced in SDAI. A comprehensive meta model for EXPRESS is currently being developed [16, 29].

7 Conclusion

We have identified some problems of and approaches to long-term data retention in a DBMS. Access to massive data over decades is not only an issue for engineering databases. Parts of STEP, especially EXPRESS and SDAI, are worth being studied by the data modelers and database designers of various application domains including SSDBs. Primarily, these standards address the logical aspects of long-term support such as data independence and data interpretability through metadata. Nevertheless, the upcoming EXPRESS databases hardly satisfy the physical long-term requirements such as the inexpensive storage of data on tertiary devices. We will continue working on application-oriented database archiving while taking both kinds of problems into account.

[6] Note that this is not yet the end-user's archiving interface. For instance, if a CAD system has a local data management component, archiving a design implies data transfer from the CAD system to the STEP database [12].

Acknowledgements

I would like to thank K. Küspert (IBM) and W. Käfer (Daimler–Benz) for their valuable comments on the draft of this paper. Furthermore I especially appreciated the collaboration with A. Primbs.

References

[1] M. J. Carey, L. M. Haas, and M. Livny. Tapes hold data, too: Challenges of tuples on tertiary store. In *ACM SIGMOD*, pages 413–417, Washington, 1993.

[2] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison–Wesley, Massachusetts, 1990.

[3] J. B. Cushing, D. Maier, M. Rao, D. M. DeVaney, and D. Feller. Object-oriented database support for computational chemistry. In *6th Int. Working Conf. on Scientific and Statistical Database Management*, pages 58–76, Ascona, 1992.

[4] K. R. Dittrich. Preface. In *Advances in Object-Oriented Database Systems*, volume 334 of *LNCS*, Berlin Heidelberg New York, 1988. Springer.

[5] B. K. Ehlmann, G. A. Riccardi, and L. C. Dennis. Representing non-inheritance relationships in an object-oriented, scientific database. In *6th Int. Working Conf. on Scientific and Statistical Database Management*, pages 99–109, Ascona, 1992.

[6] J. Encarnacao and P. C. Lockemann. *Engineering Databases*. Springer, Berlin Heidelberg New York, 1990.

[7] J. C. French, A. K. Jones, and J. L. Pfaltz. Summary of the final report of the NSF workshop on scientific database management. *SIGMOD Record*, 19(4):33–40, Dec. 1990.

[8] H. Gilgen and D. Steiger. The BSRN database – metadata management as a prerequisite for the radiation data in the BSRN. In *6th Int. Working Conf. on Scientific and Statistical Database Management*, pages 307–326, Ascona, 1992.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1993.

[10] L. Haas, W. Chang, G. Lohman, et al. Starburst mid-flight: As the dust clears. In *IEEE Trans. on Knowledge and Data Engineering*, Mar. 1990.

[11] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[12] A. Herbst and B. Malle. Perspectives for archiving CAD data in a STEP environment. In *CAD'94*, Paderborn, Mar. 1994. Hanser. in German.

[13] ISO. *10303-11: Product Data Representation and Exchange - Part 11: EXPRESS Language Reference Manual.* as IS to be released in 1994.

[14] ISO. *10303-22: Product Data Representation and Exchange - Part 22: Standard Data Access Interface.* editorially revised CD (N350), May 1994.

[15] R. H. Katz and T. J. Lehmann. Database support for versions and alternatives of large design files. *IEEE Trans. on Software Engineering*, 10:191–200, Mar. 1984.

[16] F. Leymann. Towards the STEP neutral repository. In *CALS Europe'93 Conf.*, Berlin, 1993.

[17] C. Mohan and I. Narang. An efficient and flexible method for archiving a data base. In *ACM SIGMOD*, Washington, 1993.

[18] M. A. Olson. Extending the Postgres database system to manage tertiary storage. Master's thesis, Univ. of California, Berkeley, 1992.

[19] J. Owen. *STEP - An Introduction*. Information Geometers, Winchester, 1993.

[20] A. Primbs. STEP/EXPRESS data management using an object-oriented database system. Diploma thesis, IBM Heidelberg Scientific Center / Univ. of Mannheim, Dec. 1993. in German.

[21] M. Rafanelli. Statistical and scientific database management systems. *Encyclopedia of Computer Science and Technology*, 23(8), 1990.

[22] T. Rando and L. McCabe. Object–oriented approaches to the implementation of EXPRESS–based systems. In *3rd Int. EXPRESS Users Group Conf.*, Berlin, 1993.

[23] M. A. Roth and S. J. VanHorn. Database compression. *SIGMOD Record*, 22(3), Sept. 1993.

[24] D. A. Schenck and P. R. Wilson. *Information Modeling: The EXPRESS Way*. Oxford University Press, 1994.

[25] F. Schoenefeld and C. Bohm. Using EXPRESS database technology for accessing NCBI genomic data. In *3rd Int. EXPRESS Users Group Conf.*, Berlin, 1993.

[26] M. Stonebraker. The design of the Postgres storage system. In *13th VLDB*, pages 289–300, Brighton, 1987.

[27] M. Stonebraker. Managing persistent objects in a multi-level store. In *ACM SIGMOD*, pages 2–11, Denver, 1991.

[28] M. Stonebraker, J. Frew, and J. Dozier. The Sequoia 2000 architecture and implementation strategy. Technical Report Sequoia 2000 93/23, Univ. of California, Berkeley, 1993.

[29] B. G. Wenzel. The future of EXPRESS. In *CAD'94*, Paderborn, Mar. 1994. Hanser. in German.

[30] P. R. Wilson. EXPRESS tools and services. Rensselaer Polytechnic Institute, Troy, Dec. 1993.