become a performance bottleneck. NASD has benefits of scaling aggregate bandwidth by spreading partial functions over a number of disk processors.

Object-level storage abstraction, introduced by NASD, is recognized to be the third alternative in addition to block-level storage abstraction typically seen in SAN storage environments and file-level storage abstraction in NAS storage environments. Content-Addressable Storage is another example which offers object-level storage accesses and is now widely deployed in enterprise systems.

The idea of NASD has been standardized as ANSI T10 SCSI OSD, which specifies an extension of the SCSI protocol for clients and devices to exchange objects and their related information. Thus, OSD is sometimes seen as a promising infrastructure of intelligent storage devices in which more intelligence will be incorporated.

Cross-references

- ▶ Active Disks
- ▶ Intelligent Storage Systems

Recommended Reading

1. ANSI. Information Technology - SCSI Object-Based Storage Device Commands (OSD). Standard ANSI/INCITS 400–2004. 2004.
2. Gibson G.A. and Van Meter R. Network attached storage architecture. Commun. ACM, 43(11):37–45, 2000.
3. Gibson G.A., Nagle D.F., Amiri K., Chang F.W., Feinberg E.M., Gobioff H., Chen Lee, Ozceri B., Riedel E., Rochberg D., and Zelenka J. File server scaling with network-attached secure disks. In Proc. 1997 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Comp. Syst., 1997, pp. 272–284.

Network Attached Storage

Kazuo Goda
The University of Tokyo, Tokyo, Japan

Synonyms

NAS

Definition

Network attached storage is a storage device which is connected to a network and provides file access services. Network attached storage is often abbreviated to NAS. Although the term NAS refers originally to such a storage device, when the term is used in the context of explaining and comparing storage network architectures it sometimes refers to a storage network architecture in which NAS devices are mainly implemented.

Key Points

A NAS device is basically comprised of disk drives which store files and controllers which export access services to the files. A file sever which runs network file system (NFS) and/or common internet file system (CIFS) to export file sharing services is a type of NAS implementation, but recent NAS products are sometimes implemented using dedicated hardware and software to increase reliability and performance. A diskless NAS device which contains only controllers is sometimes referred to as a NAS gateway or a NAS head. A NAS gateway/head has two types of network ports: one is connected to disk storage devices over a SAN and the other is to NAS clients over IP networks. The clients are thus provided with access services towards files stored in the storage devices. That is, a NAS gateway/head can be seen as a service bridge between SAN and NAS systems.

Cross-references

- ▶ Direct Attached Storage
- ▶ Storage Area Network
- ▶ Storage Network Architectures

Recommended Reading

1. Storage Network Industry Association. The Dictionary of Storage Networking Terminology. Available at: http://www.snia.org/.
2. Troppens U., Erkens R., and Müller W. Storage Networks Explained. Wiley, New York, 2004.

Network Data Model

Jean-Luc Hainaut
University of Namur, Namur, Belgium

Synonyms

CODASYL data model; DBTG data model

Definition

A database management system complies with the *network data model* when the data it manages are organized as data records connected through binary relationships. Data processing is based on navigational primitives according to which records are accessed and

updated one at a time, as opposed to the set orientation of the relational query languages. Its most popular variant is the CODASYL DBTG data model that was first defined in the 1971 report from the CODASYL group, and that has been implemented into several major DBMSs. They were widely used in the seventies and eighties, but most of them are still active at the present time.

Historical Background

In 1962, C. Bachman of General Electric, New-York, started the development of a data management system according to which data records were interconnected via a network of relationships that could be navigated through [2]. Called Integrated Data Store (IDS), this disk-based system quickly became popular to support the storage, the management and the exploitation of corporate data.

IDS was the main basis of the work of the CODASYL Data Base Task Group (DBTG) that published its first major report in 1971 [3,6], followed by a revision in 1973 [3,9]. This report described a general architecture for DBMSs, where the respective roles of the operating system, the DBMS and application programs were clearly identified. It also provided a precise specification of languages for data structure definition (Data Description Language or Schema DDL), for data extraction and update (Data Manipulation Language or DML) and for defining interfaces for application programs through language-dependent views of data (Sub-schema DDL).

The 1978 report [7,9] clarified the model. In particular physical specifications such as indexing structures and storage were removed from the DDL and collected into the Data Storage Description Language (DSDL), devoted to the physical schema description. In 1985, the X3H2 ANSI Database Standard committee issued standards for network database management systems, called NDL and based on the 1978 CODASYL report. However, due to the increasing dominance of the relational model these proposals have never been implemented nor updated afterwards.

Some of the most important implementations were Bull IDS/II (an upgrade of IDS), NCR DBS, Siemens UDS-1 and UDS-2, Digital DBMS-11, DBMS-10 and DBMS-20 (now distributed by Oracle Corp.), Data General DG/DBMS, Philips Phollas, Prime DBMS, Univac DMS 90 and DMS 1100 and Culliname IDMS, a machine-independent rewriting of IDS (now distributed by Computer Associates). Other DBMSs have been developed, that follow more or less strictly the CODADYL specifications. Examples include Norsk-Data SYBAS, Burroughs DMS-2, CDC IMF, NCR IDM-9000, Cincom TOTAL and its clone HP IMAGE (which were said to define the *shallow* data model), and MDBS and Raima DbVista that both first appeared on MS-DOS PCs.

In the seventies, IBM IMS was the main competitor of CODASYL systems [11]. From the early eighties, they both had to face the increasing influence of relational DBMSs such as Oracle (from 1979) and IBM SQL/DS (from 1982 [8]). Nowadays, most CODASYL DBMSs provide an SQL interface, sometimes through an ODBC API. Though the use of CODASYL DBMSs is slowly decreasing, many large corporate databases are still managed by network DBMSs. This state of affairs will most probably last for the next decade. Network databases, as well as hierarchical databases, are most often qualified *legacy*, inasmuch as they are often expected to be replaced, sooner or later, by modern database engines.

Foundations

The presentation of the network model is based on the specifications published in the 1971 and 1973 reports, with which most CODASYL DBMSs comply.

The Languages

The data structures and the contents of a database can be created, updated and processed by means of four languages, namely the *Schema DDL* and *Sub-schema DDL*, through which the global schema and the subschemas of the database are declared, the *Data Storage Description Language* or *DSDL* (often named *DMCL*) that allows physical structures to be defined and tuned, and the *DML* through which application programs access and update the contents of the database.

Gross Architecture of a CODASYL DBMS

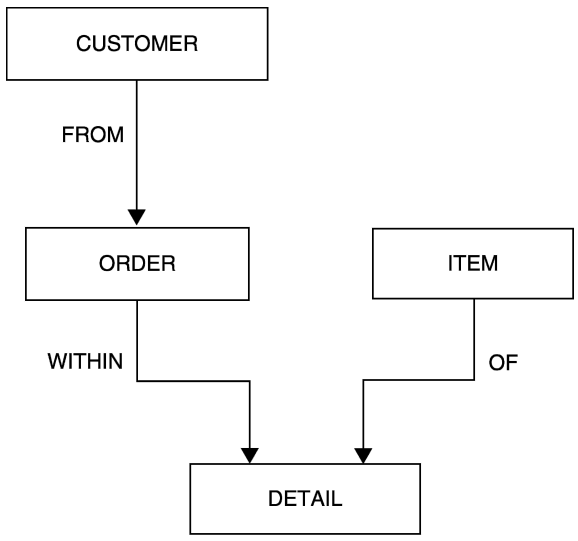
The CODASYL reports define the interactions between client application programs and the database. The resulting architecture actually laid down the principles of modern DBMSs. The DBMS includes (at least) three components, namely the DDL compiler, the DML compiler and the database control system (DBCS, or simply system). The DDL compiler translates the data description code into internal tables that are stored in the database, so that they can be exploited by the DML compiler at program compile time and by the DBCS at program run time. Either the DML compiler

is integrated into the host language compiler (typically COBOL) or it acts as a precompiler. It parses applications programs and replaces DML statements with calls to DBMS procedures. The DBCS receives orders from the application programs and executes them. Each program includes a *user working area* (UWA) in which data to and from the database are stored. The UWA also comprises registers that inform the program on the status of the last operations, and in particular references to the last records accessed/updated in each record type, each area, each set type and globally for the current process. These references, called *currency indicators*, represent static predefined cursors that form the basis for the navigational facilities across the data.

The Data Structures

The pictorial representation of a schema, or *Data Structure Diagram* [1], looks like a graph, where the nodes are the database *record types* and the edges are *set types*, that is, binary 1:N relationship types between record types, conventionally directed from the *one* side to the *many* side. Both record types and set types are named (Fig. 1). In this popular representation, record fields as well as various characteristics of the data structures are ignored.

Records and Record Types A *record* is the data unit exchanged between the database and the application program. A program reads one record at a time from the database and stores one record at a time in the database. Records are classified into *record types* that define their common structure and default behavior. The intended goal of a record type is to represent a real world entity type. A database key, which is a database-wide system-controlled identifier, is associated with each record, acting as an object-id.



Network Data Model. Figure 1. Diagram representation of the schema of a sample database.

Each database includes the SYSTEM record type, with one occurrence only, that can be used to define access paths across user record types through SYSTEM-owned singular set types.

The schema specifies, via the record type *location mode*, how a record is stored and how it is retrieved when storing a dependent record. This feature is both very powerful and, when used at its full power, fairly complex. Two main variants are proposed:

- location mode *calc using field-list*: the record is stored according to a hashing technique (or later through B-tree techniques) applied to the record key, composed of one or several fields of the record type (*field-list*); at run time, the default way to access a record will be through this record key;
- location mode *via set type S*: the record is physically stored as close as possible to the current record of set type *S*; later on, the default way to access a record will be through an occurrence of S identified by its *set selection* mode.

Record Fields A record type is composed of *fields*, the occurrences of which are *values*. Not surprisingly (CODASYL was also responsible for the COBOL specifications), their structure closely follow the record declaration of COBOL. The DDL offers the following field structures:

- *data item*: elementary piece of data of a certain type (arithmetic, string, implementor defined);
- *vector*: array of values of the same type; its size can be fixed or variable;
- *repeating group*: a somewhat misleading name for a possibly repeating aggregate of fields of any kind.

The fields of a record type can be atomic or compound, single-valued or multi-valued, mandatory or optional (through the null value); these three dimensions allow complex, multi-level field structures to be defined.

Sets and Set Types Basically, a CODASYL *set* is a list of records made up of a head record (the *owner* of the set) followed by zero or more other records (the *members* of the set). A *set type* S is a schema construct defined by

its name and comprising one owner record type and one or more member record type(s). Considering set type S with owner type A and member type B, any A record is the owner of one and only one occurrence of S and no B record can be a member of more than one occurrence of S. In other words, a set type materializes a 1:N relationship type. The owner and the members of a set type are distinct. This limitation has been dropped in the 1978 specifications, but has been kept in most implementations (exceptions: SYBAS and MDBS). Cyclic structures are allowed provided they include at least two record types. It must be noted that a set type can include more than one member record type.

The member records of S can be ordered (first, last, sorted, application-defined). This characteristic is static and cannot be changed at run-time as in SQL. The insertion of a member record in an occurrence of S can be performed at creation time (automatic insertion mode) or later by the application program (manual insertion mode). Once a record is a member of an occurrence of S, its status is governed by the retention mode; it can be removed at will (optional), it cannot be changed (fixed) or it can be moved from an occurrence to another but cannot be removed (mandatory).

The *set [occurrence] selection* of S defines the default way an occurrence of S is determined in certain DML operations such as storing records with automatic insertion mode.

Areas An area is a named logical repository of records of one or several types. The records of a definite type can be distributed in more than one area. The intended goal is to offer a way to partition the set of the database records according to real world dimensions, such as geographic, organizational or temporal. However, since areas are mapped to physical devices, they are sometimes used to partition the data physically, e.g., across disk drives.

Schema and Sub-schemas The data structures of each database are described by a schema expressed in the DDL. Though DDL is host language independent, its syntax is reminiscent of COBOL. Views are defined by sub-schemas. Basically, a subschema is a host language dependent description of a subset of the data structures of a schema. Some slight variations are allowed, but they are less powerful than relational database capabilities. Figure 2 shows a fragment of the schema declaring the data structures of Fig. 1.

Data Manipulation

The DML allows application programs to ask the DBCS data retrieval and update services. The program accesses the data through a sub-schema that identifies the schema objects the instances of which can be retrieved and updated as well as their properties, such as the data type of each field. Exchange between the host language and the DBCS is performed via the UWA, a


```
schema name is ORDER-MANAGEMENT.

area name is DOMESTIC.
area name is FOREIGN.

record name is CUSTOMER within DOMESTIC, FOREIGN;
location mode is calc using CUST-NO duplicates not allowed.
2 CUST-NO type is character 10.
2 CUST-NAME type is character 45.
2 CUST-ADDRESS.
  4 STREET character 32.
  4 CITY character 20.
  4 PHONE type is character 15 occurs 3 times.

record name is ORDER within DOMESTIC;
location mode is via FROM set;
duplicates not allowed for ORD-NO.
  2 ORD-NO type is decimal 12.
  2 ORD-DATE type is date.

set name is FROM;
owner is CUSTOMER;
order is sorted duplicates not allowed;
member is ORDER mandatory automatic key is ascending ORD-NO;
set selection is through current of FROM.
```

Network Data Model. Figure 2. Fragment of the DDL code defining the schema of Fig. 1.

shared set of variables included in each running program. This set includes the currency indicators, the process status (e.g., the error indicators) and record variables in which the data to and from the database are temporarily stored. Many DML statements use the currency indicators as implicit arguments. Such is the case for set traversal and for record storing. Based on the currency indicators, on the *location mode* of record types and on the *set selection* option of set types, sophisticated positioning policies can be defined, leading to tight application code.

Data Retrieval The primary aim of the *find* statement is to retrieve a definite record on the basis of its position in a specified collection and to make it the current of all the *communities* which it belongs to, that is, its database, its area, its record type and each of its set types. For instance, if an *ORDER* record is successfully retrieved, it becomes the current of the database for the running program (the *current of run unit*), the current of the *DOMESTIC* area, the current of the *ORDER* record type and the current of the *FROM* and *WITHIN* set types. The variants of the *find* statement allow the program to scan the records of an area, of a record type and of the members and the owner of a set. They also provide selective access among the members of a set.

The *get* statement transfers field values from a current record in the UWA, from which they can then be processed by the program.

Data Update

A record *r* is inserted in the database as follows: first, field values of *r* are stored in the UWA, then the current of each set in which *r* will be inserted is retrieved and finally a *store* instruction is issued. The *delete* instruction applies to the current record. For this operation, the DBCS enforces a *cascade* policy: if the record to be deleted is the owner of sets whose members have a mandatory or fixed retention mode, those members are deleted as well. The *modify* statement transfers in the current of a record type the new values that have been stored in the UWA. Insertion and removal of the current of a record type is performed by *insert* and *remove* instructions. Transferring a mandatory member from a set to another cannot be carried out by merely removing then inserting the record. A special case of the *modify* statement makes such a transfer possible. Later specifications as well as some implementations propose a specific statement for this operation.

Figure 3 shows, in an arbitrary procedural pseudocode, a fragment that processes the orders of customer *C400* and another fragment that creates an *ORDER* record for the same customer.

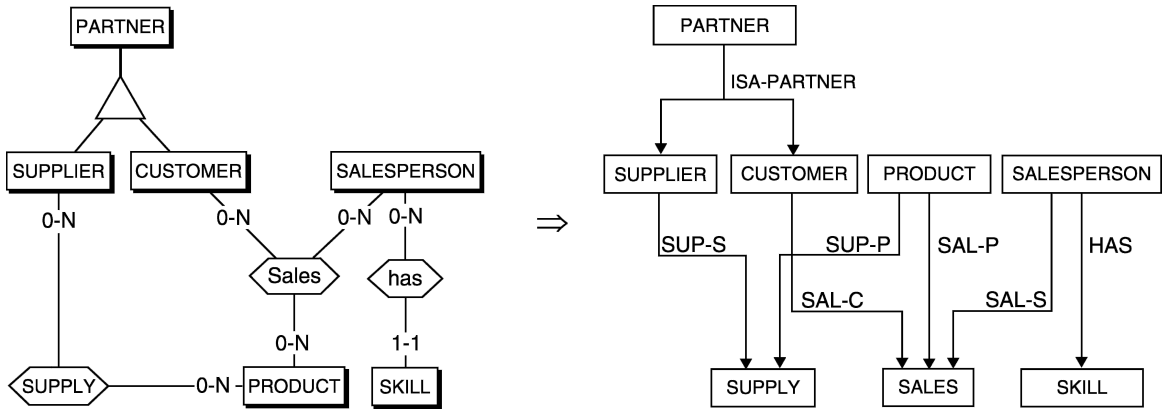
Entity-relationship to Network Mapping

Among the many DBMS data models that have been proposed since the late sixties, the network model is probably the closest to the Entity-Relationship model [5]. As a consequence, network database schemas tend to be more readable than those expressed in any other DBMS data model, at least for simple schemas. Each entity type is represented by a record type, each attribute by a field and each simple relationship type by a set type. Considering modern conceptual formalisms, the network model suffers from several deficiencies,

```
CUSTOMER.CUST-NO := "C400"                    /* store search key value in the UWA
find CUSTOMER record                          /* find CUSTOMER record "C400"
find first ORDER record of FROM set           /* find first ORDER record owned by this CUSTOMER record
while ERROR-COUNT = 0
    get ORDER                                 /* get item values of current ORDER record
    <process item values of current ORDER>
    find next ORDER record of FROM set        /* find first ORDER record
end-while

CUSTOMER.CUST-NO := "C400"
find CUSTOMER record                          /* make CUSTOMER record "C400" the current of FROM
if ERROR-COUNT = 0 then
    ORDER.ORD-NO := 30183                     /* store value of ORD-NO in the UWA
    ORDER.CUST-DATE := "2008/08/19"           /* store value of ORD-DATE in the UWA
    store ORDER                               /* store ORDER record and insert it in the current FROM set
end-if
```

Network Data Model. Figure 3. Two examples of data manipulation code.



Network Data Model. Figure 4. Partial translation of a representative Entity-relationship schema (left) into a network schema (right).

notably the lack of generalization-specialization (is-a) hierarchies and the fact that relationship types are limited to the 1:N category. Translating an Entity-relationship schema into the network model requires the transformation of these missing constructs into standard structures.

Is-a hierarchies. Three popular transformations can be applied to express this construct in standard data management systems, namely one record type per entity type, one record type per supertype and one record type per subtype. Representing each entity type by a distinct record type and forming a set type *S* with each super-type (as owner of *S*) and all its direct subtypes (as members of *S*) is an appropriate implementation of the first variant.

1:1 relationship type. This category is a special case of 1:N and can be expressed by a mere set type, together with dynamic restriction on the number of members in each set. However, merging both record types when one of them depends on the other one (e.g., as an automatic, mandatory member) is also a common option.

Complex relationship type. In most implementations, n-ary and N:N relationship types as well as those with attributes must be reduced to constructs based on 1:N relationship types only through standard transformations. A complex relationship type *R* is represented by a relationship record type *RT* and by as many set types as *R* has roles. The attributes of *R* are translated into fields of *RT*. Cyclic relationship types, if necessary, will be translated in the same way.

Figure 4 illustrates some of these principles.

Discussion

The network model offers a simple view of data that is close to semantic networks, a quality that accounts for much if its past success. The specifications published in the 1971 and 1973 reports exhibited a confusion between abstraction levels that it shared with most proposals of the seventies and that was clarified in later recommendations, notably the 1978 report and X3H2 NDL. In particular, the DDL includes aspects that pertain to logical, physical and procedural layers.

Though they were not implemented in most commercial DBMSs, the CODASYL recommendations included advanced features that are now usual in database technologies such as database procedures, derived fields, check and some kind of triggers.

Key Applications

CODASYL DBMSs have been widely used to manage large corporate databases submitted to both batch and OLTP (On-line Transaction Processing) applications. Compared with hierarchical and relational DBMS, their simple and intuitive though powerful model and languages made them very popular for the development of large and complex applications. However, their intrinsic lack of flexibility in rapidly evolving contexts and the absence of user-oriented interface made them less attractive for decisional applications, such as data warehouses.

Cross-references

► Hierarchical Data Model
► Relational Model

► Database Management System
► Entity-Relationship Model

Recommended Reading

1. Bachman C. Data structure diagrams. ACM SIGMIS Database, 1(2):4–9, 1969.
2. Bachman C. The programmer as navigator. Commun. ACM, 16(11):635–658, 1973.
3. DBTG C. CODASYL data base task group, April 1971 report, ACM, New York, 1971.
4. DDLC C. CODASYL data description language committee, CODASYL DDL Journal of Development (June 1973), NBS Handbook 113 (Jan. 1974), 1973.
5. Elmasri R. and Navathe S. Fundamentals of Database Systems (3rd edn.). Addison-Wesley, 2000. (The appendix on the network data model has been removed from later editions but is now available on the authors' site.)
6. Engels R.W. An analysis of the April 1971 DBTG report. In Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, 1971, pp. 69–91.
7. Jones J.L. Report on the CODASYL data description language committee. Inf. Syst., 3(4):247–320, 1978.
8. Michaels A., Mittman B., and Carlson C.A. Comparison of relational and CODASYL approaches to data-base management. ACM Comput. Surv., 8(1):125–151, 1976.
9. Olle W. The CODASYL Approach to Data Base Management, Wiley, New York, NY, 1978.
10. Taylor R. and Frank R. CODASYL data-base management systems. ACM Comput. Surv., 8(1):67–103, 1976.
11. Tsichritzis D. and Lochovsky F. Data Base Management Systems, Academic Press, New York, NY, 1977.

Network Database

► Graph Database

Network Topology

► Visualizing Network Data

Neural Networks

Pang-Ning Tan
Michigan State University, East Lansing, MI, USA

Synonyms

Connectionist model
Parallel distributed processing

Definition

An artificial neural network (ANN) is an abstract computational model designed to solve a variety of supervised and unsupervised learning tasks. While the discussion in this chapter focuses only on supervised classification, readers who are interested in unsupervised learning using ANN may refer to the literature on vector quantization [6] and self organizing maps [11]. An ANN consists of an assembly of simple processing units called neurons connected by a set of weighted edges (or synapses), as shown in Fig. 1. The neurons are often configured into a feed-forward multi-layered topology, with outputs from one layer being fed into the next layer. The first layer, which is known as the input layer, encodes the attributes of the input data, while the last layer, known as the output layer, encodes the neural network's output. Hidden layers are the intermediary layers of neurons between the input and output layers. A feed-forward ANN without any hidden layer is called a perceptron [16]. Another common ANN architecture is the recurrent network, which allows a neuron to feed its output back into the inputs of other preceding neurons in the network. Such a network topology is useful for modeling temporal and sequential relationships in dynamical systems.

Historical Background

The design of an ANN was inspired by the desire to emulate how a human brain works. Interest in this field began to emerge following the seminal work of McCulloch and Pitts [13], who attempted to understand how complex patterns can be modeled in the brain using a large number of inter-connected neurons. They presented a simplified model of a neuron and showed how a collection of these neurons could be used to represent logical propositions. Nevertheless, they did not provide an algorithm to estimate the weights of the network.

A major step forward in the study of ANN occurred when Hebb [7] formulated a postulate relating the cerebral activities of the brain to the synaptic connections between neurons. Hebb theorized that the process of learning takes place when a pair of neurons is activated repeatedly, thus making their synaptic connection stronger. By strengthening the connection, this enables the network to recognize the appropriate response when the same stimulus is re-applied. This idea forms the basis for what is now known as Hebbian learning.