ELSEVIER

Information and Software Technology 41 (1999) 969–978

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Thirty years (and more) of databases

M. Jackson

School of Computing and Information Technology, University of Wolverhampton, Lichfield St., Wolverhampton WV1 1EL, UK

Abstract

This paper outlines the historical development of data management systems in order to identify the key issues for successful systems. It identifies the need for data independence and the embedding of structural and behavioural semantics in the database as key issues in the development of modern systems. Hierarchical, Network, Relational, Object-oriented and Object-relational data management systems are reviewed. A short summary of related research is given. The paper concludes with some speculation on the future directions that database technology might take. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Data management systems; Database; Data engineering

1. Introduction

We all know that there is a discipline which we call *software engineering*, it has to be the case for there are a sufficient number of textbooks available with the phrase appearing prominently in the title. Many worthy academic institutions have chairs of software engineering and there are numerous international conferences, workshops, symposiums and the like dedicated to the exploration of sub-areas of software engineering. It is not as clear that there exists a similar discipline called *data engineering*. It is true that there are a small number of conferences dedicated to this topic as well as an IEEE Technical Committee on Data Engineering, however, the term continues to be misunderstood by anyone not involved in data engineering and by software engineers in particular. I often have to explain the meaning of the title of my own chair (Professor of Data Engineering) to individuals with considerable experience in the computer industry.

The disciplines of software engineering and data engineering are similar but have different emphases and historical roots. The starting point for a software engineer is a task that must be carried out on a computer. A data engineer most often begins with a task that exists already either as a paper-based system or in some computerised form and seeks to engineer a better solution. A software engineer says that a program is made up of data and algorithms and generally means transient (main memory) data. A data engineer says that applications are constructed to run on top of data and means permanent (secondary storage) data. A software engineer designs systems, a data engineer constructs a basis upon which systems may be built.

This said, the similarity between data engineering and software engineering is probably greater than the difference. Both disciplines attempt to encourage principles and practices that enable developers to speedily construct systems which match their specifications and can be demonstrated to function correctly. The two disciplines are about *engineering* solutions to similar problems. Both disciplines attempt to extract essential semantics from a real world situation and preserve them in an application. Software engineering tends to seek for ways of encoding these semantics in code whilst data engineering embeds them in metadata.

The flagship product and the most obvious achievement of data engineering is the database management system. Such systems now form part of a billion dollar industry and knowledge of the most commonly used database language SQL is a skill quoted in almost as many job adverts as the leading programming languages. This paper surveys the history of the development of database management systems (DBMS) with regard to the engineering principles that succeeding generations of such systems have embodied.

2. The origins of data storage systems

The invention of magnetic storage media such as magnetic tape and magnetic disks enabled the permanent storage of large quantities of data in a manner that made them amenable to computer processing. The term ‘large’ is not used in absolute sense it is simply an indication that storing punched card or paper tape representations of data

E-mail address: m.s.jackson@wlv.ac.uk (M. Jackson)

was never a realistic option for many potential data processing applications. A number of business-related uses of computers came into being as a direct result of this development. Typically these relied on 'batch' operations. Stored records were kept on master files. Over a period of time a set of transactions or operations were collected and at an appropriate time run against a master file. The master file and the transaction file were sorted in the same order on some key. At regular intervals the transactions were applied to the master file and a new updated master file was produced. At the same time a report indicating the success or the failure of each transaction was generated.

This scenario contained a number of inadequacies. Firstly this type of system made no attempt to describe the data it held. The only assistance a programmer could hope to receive from underlying software was that the operating system could find the file. Once the file had been located on the disk it was the programmer's task to handle the file as a contiguous piece of permanent storage. There was no indication given as to whether the bytes read were representing single characters, character strings or numbers. It was the programmer's task to add the semantics of the application to the stream of data retrieved from the disk. Eventually programming language support was provided to make this task easier, however, such support was limited to aiding an individual programmer and not everyone (including non-programmers) who needed to access the data. It was possible for two programmers to describe the same data in two different ways and hence apply different semantics to it. What semantics were made available in a programming language were limited to a simple description of the way in which the data might be displayed and did not describe the operations and constraints that were appropriate to it.

Secondly, it was quickly recognised that this pattern of processing was repeated time and time again. The central logic of each program was identical, all that altered was the details of the input and output operations. Despite this, each program was handcrafted each time. This did not improve productivity nor did it ensure that a solution known to be correct was applied consistently.

Thirdly, the idea of a file of data in isolation did not correspond with the way data was known to behave in application areas. A computer file corresponded to what one might expect from a manual filing cabinet. It was a bringing together of a number of fixed format pieces of information called records. Records consisted of a number of fields that held individual pieces of information. The records in a file were normally sorted in some order to allow speedy processing and retrieval. It was known, however, that many applications relied on an ability to retrieve records based on their relationships to records in other files. More than this, the validity of entries in some records depended on entries found in other files. Implementing systems that embodied these semantics was possible but involved the construction of quite complex programs that were difficult to maintain.

Fourthly, these file-based systems did not support the type of processing that businesses were beginning to expect. The promise of information technology has always been the speedy delivery of flexible solutions to business problems. The rate of change in the business environment has grown remorselessly as the twentieth century has progressed. Companies now expect to introduce new products and new working practices on a regular basis. They expect their information systems to cope with these changes. First generation information systems were not able to do this. Changes required long term planning and often risked introducing faults into systems that were known to work correctly. In addition, these systems separated the users of the data from the data itself requiring them to use technical staff as intermediaries as there was no way of submitting ad hoc queries.

3. Hierarchical systems

The first problem that was addressed by database technology was that of mirroring the relationships that exist in the real world in the structure of the data. This was impractical where data was stored on paper tape or punched cards. With the introduction of addressable magnetic disks, however, it is possible to embed pointers into records. These pointers represented relationships between data. Manipulation of such pointers is a standard programming task but one that requires an approach that leads to consistency in the data. Software to aid the programmer in producing consistent record references formed the basis of the first database management software. In the late 1960s, however, magnetic tape was still a major medium for data storage. Tape does not have the addressing flexibility of the magnetic disk and therefore a data model that supported sequential access was necessary for this type of storage. This requirement led to the development of the hierarchical model of data implemented in IBM's database product: IMS [1]. Any hierarchy of records can be represented as a sequence and such a sequence can be stored on magnetic tape. The first major data model came into being purely out of consideration for the underlying physical storage it had to work on.

The original use intended for IMS was "bill of materials processing" and the data model chosen was ideal for this purpose. This type of application deals with facts such as "Part A is constructed from Parts B and C, Part B is constructed from Parts D, E and F". This is a natural hierarchy (tree) and is easily mapped to the IMS data model. More complex scenarios required extensions to the original model so that data whose relationships could not be represented by a single tree could efficiently stored as a collection of trees.

IMS did not capture the semantics of the data it stored beyond being able to represent relationships between records. Individual fields were not identified by the database management system; a record was defined simply as a number of bytes into which data could be placed. As a

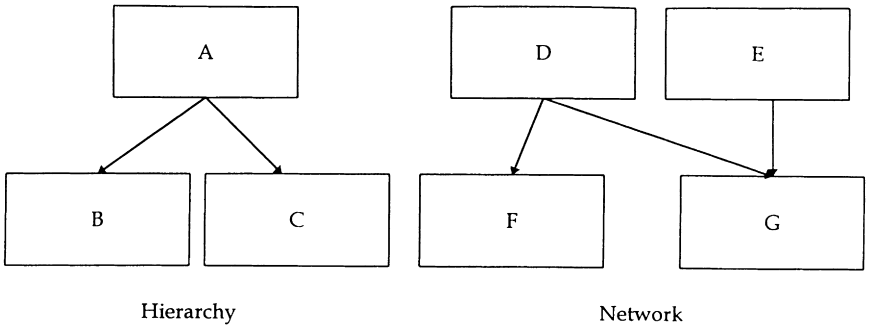

Fig. 1. Comparison of hierarchy and Network.

consequence it was unable to support ad hoc queries. The processing semantics were entirely embedded within the programs written for applications and it was necessary to write programs in order to access the database.

4. The network model

The concept of a network model of data and the relational model of data (see later) were developed at roughly the same time. The network model was, however, more quickly embodied in commercial database management systems than the relational model. Moreover, many of the features that appeared in systems based on the network model influenced the research and development of commercial relational systems. This paper will therefore deal with the network model first and then consider the relational model later.

The network model removes a limitation in the relationships that can be represented in the hierarchical model. In a pure hierarchical model of data it is only possible to represent one-to-many relationships (one record of type A is related to many records of type B). This comes about because in a hierarchy a node may only have a single parent. In a network nodes may be related to more than one other node.

Fig. 1 shows logical diagrams of relationships in both the hierarchical and the network models. In the hierarchy, a record of type A may be related to many records of type B and many records of type C. This is all that is permitted. In the network diagram a record of type D may be related to many records of type F and also to many records of type G. This could be represented in the hierarchical model. However, the diagram also indicates that a record of type E may be related to many records of type G. These additional relationships would not be permitted in a pure hierarchical model. Given these two one-to-many relationships it is possible to construct many-to-many relationships between records of type D and records of type E (i.e. a record of type D may be related to many records of type E and vice versa).

There were a number of commercial systems that adopted this model of data (e.g. TOTAL [2]), however; systems based on the recommendations of the CODASYL committee are discussed most widely in the literature [3]. Not only did the CODASYL report popularise the network model it also introduced a number of other important innovations to database management systems. These are in themselves worthy of discussion.

Ostensibly the CODASYL effort was an attempt to preserve the supremacy of the COBOL language. In the 1960s COBOL had become the predominant language for data processing and CODASYL was the body that had responsibility for developing COBOL standards. By the end of the decade it was clear that database management systems were destined to become a major player in the data processing arena and it was considered appropriate to examine ways in which database technology could be integrated with the COBOL language. CODASYL set up a working party known as the data base task group (DBTG). The outcome of this exercise was a report that specified a number of languages to define and manipulate a database based on the network model.

The main innovation of the DBTG report was the separation of the various concerns of data management. Three languages were defined. Schema data definition language (DDL) allowed a database designer to completely define a database without reference to the applications that might run against it. The syntax of this was similar to that used in the COBOL data definition section. There was also a subschema DDL which allowed a subset of the total database to be made visible to a user (in the context of CODASYL a user was a programmer). Subschema DDL allowed minor redefinitions of the structures defined in the schema. Importantly, in both the schema and the subschema both the length and the type of individual fields of a record could be defined. In addition, it was possible to define constraints on relationships. For example, it was possible to stipulate that if a record of a given type was added to the database that it must be related to a record of another type. The same mechanism defined what happened to a record when a related record was deleted. A data manipulation language (DML) allowed a programmer to navigate the relationships in the database

without having to be aware that the records were connected via address pointers.

CODASYL also introduced a limited form of physical data independence. There was no requirement that records should be stored as they were described in the schema. The only guarantee was that the system would deliver the data to the applications program as though it was stored in the way the schema indicated. Instances of relationships in CODASYL were often drawn as though they were implemented via rings of records connected by pointers. While this was a possible means of physically implementing relationships, a pointer array was an alternative physical storage mechanism. The programmer could navigate the database using the same DML commands, regardless of the underlying storage mechanism chosen by the database designer. This was not the case in IMS where the type of storage structure chosen limited the operations a programmer could perform. Despite this limited form of data independence, CODASYL schemas mixed together physical and logical concerns. The schema not only described what was to be stored but also contained definitions that governed the way the data could be accessed. This meant that the database designers had to guess what applications might use the data and devise appropriate access paths to support them. Such an approach is not suited to an environment in which new application areas are constantly emerging.

Data storage and retrieval in a CODASYL database was limited to predetermined operations invoked through programming languages which would support an embedded DML. Access to the database was typically performed on a record at a time basis with the programmer testing each record retrieved to see whether or not it belonged in the result set. As a consequence much of the semantics of the data remained embedded in application programs and different sets of semantics could be applied by different applications.

The writers of the CODASYL report also defined syntax for two key aspects of database management systems: concurrency (the ability of the database to safely support simultaneous users of the system) and security. The concurrency mechanism provided a facility whereby portions of the database (known initially as areas) could be locked to prevent simultaneous access where this was necessary. The syntax for security allowed a password to be associated with virtually every object (even as far as individual fields) described in the schema.

The multiple schema approach adopted by CODASYL undoubtedly influenced the conclusions of the ANSI/SPARC study group who met to discuss which areas of database technology were amenable to standardisation. The outcome was a three-schema approach [4], which continues to influence database system architects. The three schemas were the internal, the external and the conceptual. The internal schema describes the physical storage of data. The external schema describes a user view of data (there may be many of these in a given system). The conceptual schema provides a community view of the database. The mappings between the schemas are responsible for making applications independent from the storage of the data.

5. The relational model

Although the relational model was propounded at roughly the same time the CODASYL report was published, the two types of system did not develop at the same rate. At the time it appeared that the relational model was a revolutionary step whereas the CODASYL report was an evolutionary step. Many modern writers in hindsight view the development of the relational model as an obvious step forward from systems constructed out of files with fixed length records, but at the time it emerged it was not regarded in this light.

The relational model devised by Codd [5] at IBM arose from the consideration of a number of concerns. These were:

- the need to increase data independence in database management systems;
- the need for a mathematical approach to data storage and retrieval;
- the need to support ad hoc query processing.

Data independence has already featured in this paper; however, it is probably worthwhile at this point briefly defining the term. It is the purpose of any database management system to provide data independence, that is, to hide certain necessary storage and retrieval operations from the applications programmer. For example, in IMS and CODASYL although the data structures implemented by both of these systems relied on the use of pointers between records, the programmers using them did not manipulate the pointers directly. The purpose of a database management system is to make life easier for the user and this is achieved by hiding the complexities of the actual storage of the data from the application software. In a database system where true data independence exists it is possible to restructure the physical storage of data without invalidating any of the existing applications. Network and hierarchical systems suffer from a type of data dependence known as access path dependence. Access to IMS data, for example, requires a programmer to enter the database via a record at the top of the hierarchy. If a programmer does not know which of the top level records to select then the whole database must be searched. CODASYL designers identify entry points to the database via key hashing mechanisms or special relationships. Both the IMS and the CODASYL operated on a record at a time basis and allowed a programmer to build applications that relied on the fact that records would be retrieved in a given order. There are many potential ways in which records can be ordered but only one can be in use for a given record set. This effectively means that the

database favours some access paths (or queries) and makes others either impossible or hopelessly inefficient. The database is biased in favour of certain applications. The database designer must attempt to anticipate all the possible applications that may be run against the database and build appropriate access paths at design time. If, subsequently, applications emerge which are not supported by the preplanned access paths then the database must be redesigned and rebuilt. This is almost always a lengthy process. The relational model attempted to improve on this situation by having no predefined access paths. The user of a relational system is able to extract any results that follow from the content of the raw data and not just those permitted by the database system. In order to facilitate this no aspect of a relational system depends on the order in which the data is stored.

The need for a mathematical basis for database management can be viewed, in retrospect, in the light of the recognition of the need for formal methods in software engineering. The mechanisms for retrieving data from a database prior to the emergence of relational systems were similar to those familiar in every programming environment. A specification was drawn up, this was transformed into a design and the design was used to construct a program. This sequence, as everyone knows, is full of opportunities for introducing errors. It takes considerable skill to write a program that performs correctly against a database and considerably more skill to demonstrate that the program fulfils the requirements of the original specification. With the relational model, however, it is possible to submit a query expressed in predicate calculus and have the system automatically retrieve the appropriate data. This is equivalent to having an executable specification language. The problem of ensuring that the query matches the real world requirement remains but the problem of transforming the query into executable code disappears. The importance to the relational model of a mathematical basis goes beyond a mechanism for expressing queries. It has often been underemphasised. When the model was first propounded it was tempting to regard relational theory as being something which gave academic respectability to a database system which stored fixed length records in files which supported multiple indexes. Subsequently, the mathematical underpinning has often been neglected for marketing purposes. Potential purchasers of systems based on the relational model are often wary of mathematical thinking or unwilling to trust their businesses to something which relies on seemingly abstract theory. In practice, users of relational database management systems do not normally need to be aware of the underlying mathematics but DBMS developers certainly do. The commercial success of relational systems (the majority of database management software currently in use is based on the relational model) has been built on advances that depend on relational theory. For example, early critics of relational systems were quick to point out that a naive implementation of the model would almost certainly perform poorly when compared to IMS or a CODASYL implementation. It was necessary, therefore, to devise mechanisms for extracting the best possible performance from relational databases. One obvious strategy was to make sure that the version of a query that executes is the most efficient version of that query. Some form of query optimisation is required for this. Query optimisation is not possible in IMS and CODASYL systems. A query in these systems is as good as the programmer who wrote it. There isn't a way of taking a computer program and reliably transforming it to an equivalent but more efficient version. In a relational database, however, a query is a mathematical expression and may be transformed to another expression that is completely equivalent to the first. If this property is combined with a set of heuristics for determining which queries are likely to run in shorter time then a query optimisation mechanism can be implemented.

Codd anticipated in his early papers on the relational model that the expectations of database users would change. The era that witnessed the emergence of the relational model also saw the introduction of timesharing facilities and interactive dialogues with computers. Codd believed that users would no longer be satisfied with obtaining output from a database on a daily basis following the off-line execution of a query program, instead they would expect to interactively submit a query and immediately receive an answer. Provision of such a facility relies, to a certain extent, on data independence and relational theory. Ad hoc queries are not ad hoc if they are limited to the access paths built by the database designer. Similarly, this feature cannot be supplied unless there is a mechanism to take any high level query, check whether it is valid against the database and transform into something which may be executed. The facility to support ad hoc querying in relational systems depends upon the fact that the relational model defines a small number of relational algebra operations and that it has been demonstrated that any predicate calculus expression can be mapped to a sequence of these operations. A further feature is, however, required to support ad hoc queries. If a user submits a query requesting the output of some given columns in a named table, the database software must have a mechanism to check whether the table and the columns exist. The database schema must also support ad hoc queries. Such a feature was not available in IMS or CODASYL systems, here programs were compiled against the schema and references to data were turned into absolute addresses in the object code of programs. A natural way of implementing this requirement is to make the database self describing. A relational database therefore contains relations that contain descriptions of all the relations stored in the database. This is effectively a collection of database semantics available to any database application. One type of application which can make good use of this is the application generator (sometimes know as a fourth generation language). Application generators offer a quick and convenient way of building systems based on an already defined

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.