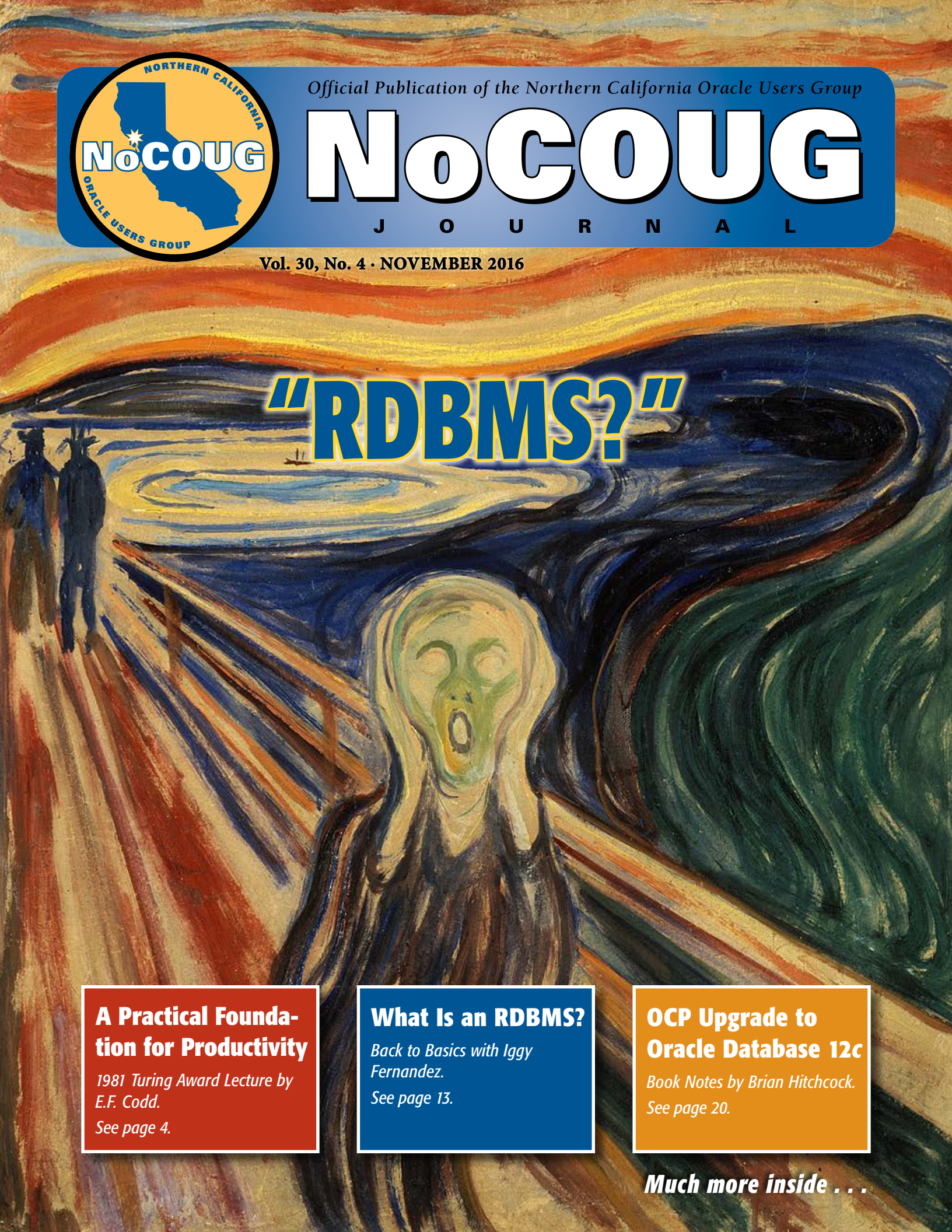
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 30, No. 4 · NOVEMBER 2016



A Practical Foundation for Productivity

1981 Turing Award Lecture by E.F. Codd.

See page 4.

What Is an RDBMS?

Back to Basics with Iggy Fernandez.

See page 13.

OCP Upgrade to Oracle Database 12c

Book Notes by Brian Hitchcock.

See page 20.

Much more inside . . .

It's time for

ZeroIMPACT

Oracle database replication at half the cost

SharePlex

your golden alternative

Visit the Dell Software booth and experience how you can:

- Dramatically **reduce** downtime by up to 99%.
- Eliminate fire drills and **migrate** at your speed.
- Minimize risk and **prevent** data loss.
- **Validate** migration success.

DELL Software

The Little User Group That Could

It requires a vast amount of work to organize a full-day user group conference and publish a printed journal every quarter. No sooner has a conference ended and a journal been mailed than it is time to start work on the next conference and the next journal. But the awesome NoCOUG volunteers have pulled it off—quarter after quarter—for 30 long years. The upcoming NoCOUG conference on Thursday, November 17 at PayPal Town Hall in San Jose will be our 120th quarterly conference! The conference is sponsored by Google Cloud Platform, and the theme is "A Bright and Cloudy Future for All Your Databases."

Special thanks to book reviewer Brian Hitchcock, copyeditor Karen Mead, and graphics duo Kenneth Lockerbie and Richard Repas for their help with the *NoCOUG Journal*. This is our 120th issue. Enjoy! ▲

—*NoCOUG Journal* Editor

Table of Contents

Publication Notices and Submission Format

The *NoCOUG Journal* is published four times a year by the Northern California Oracle Users Group (NoCOUG) approximately two weeks prior to the quarterly educational conferences.

Please send your questions, feedback, and submissions to the *NoCOUG Journal* editor at **journal@nocoug.org**.

The submission deadline for each issue is eight weeks prior to the quarterly conference. Article submissions should be made in Microsoft Word format via email.

Relational Database: A Practical Foundation for Productivity

by E.F. Codd

Editor's note: *The originator of the relational model for databases, Dr. Edgar F. Codd, delivered the following lecture on November 9, 1981 at ACM '81 where he was presented with the ACM Turing Award for his fundamental contributions to the theory and practice of database management systems. The lecture was published in Communications of the ACM, February 1982, Volume 25, Number 2.*

It is well known that the growth in demands from end users for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. There are two complementary approaches to attacking this problem (and both approaches are needed): one is to put end users into direct touch with the information stored in computers; the other is to increase the productivity of data processing professionals in the development of application programs. It is less well known that a single technology, relational database management, provides a practical foundation for both approaches. It is explained why this is so. While developing this productivity theme, it is noted that the time has come to draw a very sharp line between relational and non-relational database systems, so that the label "relational" will not be used in misleading ways. The key to drawing this line is something called a "relational processing capability."

1. Introduction

It is generally admitted that there is a productivity crisis in the development of "running code" for commercial and industrial applications. The growth in end user demands for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. In the late sixties and early seventies many people in the computing field hoped that the introduction of database management systems (commonly abbreviated DBMS) would markedly increase the productivity of application programmers by removing many of their problems in handling input and output files. DBMS (along with data dictionaries) appear to have been highly successful as instruments of data control, and they did remove many of the file handling details from the concern of application programmers. Why then have they failed as productivity boosters?

There are three principal reasons:

(1) These systems burdened application programmers with numerous concepts that were irrelevant to their data retrieval and manipulation tasks, forcing them to think and code at a needlessly low level of structural detail (the "owner-member set" of CODASYL DBTG is an outstanding example[1]);

(2) No commands were provided for processing multiple records at a time—in other words, DBMS did not support *set processing* and, as a result, programmers were forced to think and code in terms of iterative loops that were often unnecessary (here we use the word "set" in its traditional mathematical sense, not the linked structure sense of CODASYL DBTG);

(3) The needs of end users for direct interaction with databases, particularly interaction of an unanticipated nature, were inadequately recognized—a query capability was assumed to be something one could add on to a DBMS at some later time.

Looking back at the database management systems of the late sixties, we may readily observe that there was no sharp distinction between the programmer's (logical) view of the data and the (physical) representation of data in storage. Even though what was called the logical level usually provided protection from placement expressed in terms of storage addresses and byte offsets, many storage-oriented concepts were an integral part of this level. The adverse impact on development productivity of requiring programmers to navigate along access paths to reach the target data (in some cases having to deal directly with the layout of data in storage and in others having to follow pointer chains) was enormous. In addition, it was not possible to make slight changes in the layout in storage without simultaneously having to revise all programs that relied on the previous structure. The introduction of an index might have a similar effect. As a result, far too much manpower was being invested in continual (and avoidable) maintenance of application programs.

[1] The crux of the problem with the CODASYL DBTG owner-member set is that it combines into one construct three orthogonal concepts: one-to-many relationship, existence dependency, and a user-visible linked structure to be traversed by application programs. It is the last of these three concepts that places a heavy and unnecessary navigation burden on application programmers. It also presents an insurmountable obstacle for end users.

Another consequence was that installation of these systems was often agonizingly slow, due to the large amount of time spent in learning about the systems and in planning the organization of the data at both logical and physical levels, prior to database activation. The aim of this preplanning was to "get it right once and for all" so as to avoid the need for subsequent changes in the data description that, in turn, would force coding changes in application programs. Such an objective was, of course, a mirage, even if sound principles for database design had been known at the time (and, of course, they were not).

To show how relational database management systems avoid the three pitfalls cited above, we shall first review the motivation of the relational model and discuss some of its features. We shall then classify systems that are based upon that model. As we proceed, we shall stress application programmer productivity, even though the benefits for end users are just as great, because much has already been said and demonstrated regarding the value of relational database to end users (see [23] and the papers cited therein).

2. Motivation

The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management (including database design, data retrieval, and data manipulation). We call this the *data independence objective*.

A second objective was to make the model structurally simple, so that all kinds of users and programmers could have a common understanding of the data, and could therefore communicate with one another about the database. We call this the *communicability objective*.

A third objective was to introduce high level language concepts (but not specific syntax) to enable users to express operations upon large chunks of information at a time. This entailed providing a foundation for set-oriented processing (i.e., the ability to express in a single statement the processing of multiple sets of records at a time). We call this the *set-processing objective*.

There were other objectives, such as providing a sound theoretical foundation for database organization and management, but these objectives are less relevant to our present productivity theme.

3. The Relational Model

To satisfy these three objectives, it was necessary to discard all those data structuring concepts (e.g., repeating groups, linked structures) that were not familiar to end users and to take a fresh look at the addressing of data.

Positional concepts have always played a significant role in computer addressing, beginning with plugboard addressing, then absolute numeric addressing, relative numeric addressing, and symbolic addressing with arithmetic properties (e.g., the symbolic address $A + 3$ in assembler language; the address $X(I + 1, J - 2)$ of an element in a Fortran, Algol, or PL/I array named X). In the relational model we replace positional addressing by totally associative addressing. Every datum in a relational database can be uniquely addressed by means of the relation name, primary key value, and attribute name. Associative addressing of this form enables users (yes, and even programmers also!) to leave it to the system to (1) determine the details of placement of a new piece of information that is being inserted into a database and (2) select appropriate access paths when retrieving data.

All information in a relational database is represented by values in tables (even table names appear as character strings in at least one table). Addressing data by value, rather than by position, boosts the productivity of programmers as well as end users (positions of items in sequences are usually subject to change and are not easy for a person to keep track of, especially if the sequences contain many items). Moreover, the fact that programmers and end users all address data in the same way goes a long way to meeting the communicability objective.

"The adverse impact on development productivity of requiring programmers to navigate along access paths to reach the target data (in some cases having to deal directly with the layout of data in storage and in others having to follow pointer chains) was enormous."

The *n*-ary relation was chosen as the single aggregate structure for the relational model, because with appropriate operators and an appropriate conceptual representation (the table) it satisfies all three of the cited objectives. Note that an *n*-ary relation is a mathematical set, in which the ordering of rows is immaterial.

Sometimes the following questions arise: Why call it the relational model? Why not call it the tabular model? There are two reasons: (1) At the time the relational model was introduced, many people in data processing felt that a relation (or relationship) among two or more objects must be represented by a linked data structure (so the name was selected to counter this misconception); (2) Tables are at a lower level of abstraction than relations, since they give the impression that positional (array-type) addressing is applicable (which is not true of *n*-ary relations), and they fail to show that the information content of a table is independent of row order. Nevertheless, even with these minor flaws, tables are the most important conceptual representation of relations, because they are universally understood.

Incidentally, if a data model is to be considered as a serious alternative for the relational model, it too should have a clearly defined conceptual representation for database instances. Such a representation facilitates thinking about the effects of whatever operations are under consideration. It is a requirement for programmer and end-user productivity. Such a representation is rarely, if ever, discussed in data models that use concepts such as entities and relationships, or in functional data models. Such models frequently do not have any operators either! Nevertheless, they may be useful for certain kinds of data type analysis encountered in the process of establishing a new database, especially in the very early stages of determining a preliminary informal organization. This leads to the question: What is a data model?

A data model is, of course, not just a data structure, as many people seem to think. It is natural that the principal data models are named after their principal structures, but that is not the whole story. A data model [9] is a combination of at least three components:

(1) A collection of data structure types (the database building blocks);

(2) A collection of operators or rules of inference, which can be applied to any valid instances of the data types listed in (1), to retrieve, derive, or modify data from any parts of those structures in any combinations desired;

(3) A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both—these rules are general in the sense that they apply to any database using this model (incidentally, they may sometimes be expressed as insert-update-delete rules).

The relational model is a data model in this sense, and was the first such to be defined. We do not propose to give a detailed definition of the relational model here—the original definition appeared in [7], and an improved one in Secs. 2 and 3 of [8]. Its *structural part* consists of domains, relations of assorted degrees (with tables as their principal conceptual representation), attributes, tuples, candidate keys, and primary keys. Under the principal representation, attributes become columns of tables and tuples become rows, but there is no notion of one column succeeding another or of one row succeeding another as far as the database tables are concerned. In other words, the left to right order of columns and the top to bottom order of rows in those tables are arbitrary and irrelevant.

The *manipulative part* of the relational model consists of the algebraic operators (select, project, join, etc.) which transform relations into relations (and hence tables into tables).

The *integrity part* consists of two integrity rules: entity integrity and referential integrity (see [8, 11] for recent developments in this latter area). In any particular application of a data model it may be necessary to impose further (database-specific) integrity constraints, and thereby define a smaller set of consistent database states or changes of state.

In the development of the relational model, there has always been a strong coupling between the structural, manipulative, and integrity aspects. If the structures are defined alone and separately, their behavioral properties are not pinned down, infinitely many possibilities present themselves, and endless speculation results. It is therefore no surprise that attempts such as those of CODASYL and ANSI to develop data structure definition language (DDL) and data manipulation language (DML) in separate committees have yielded many misunderstandings and incompatibilities.

4. The Relational Processing Capability

The relational model calls not only for relational structures (which can be thought of as tables), but also for a particular kind of set processing called *relational processing*. Relational processing entails treating whole relations as operands. Its primary purpose is loop-avoidance, an absolute requirement for end users to be productive at all, and a clear productivity booster for application programmers.

The SELECT operator (also called RESTRICT) of the relational algebra takes *one* relation (table) as operand and produces a new relation (table) consisting of selected tuples (rows) of the first. The PROJECT operator also transforms *one* relation (table) into a new one, this time however consisting of selected attributes (columns) of the first. The EQUI-JOIN operator takes *two* relations (tables) as operands and produces a third consisting of rows of the first concatenated with rows of the second, but only where specified columns in the first and specified columns in the second have matching values. If redundancy in columns is removed, the operator is called NATURAL JOIN. In what follows, we use the term "join" to refer to either the equi-join or the natural join.

The relational algebra, which includes these and other operators, is intended as a yardstick of power. It is *not* intended to be a standard language, to which all relational systems should adhere. The set-processing objective of the relational model is intended to be met by means of a data sublanguage[2] having at least the power of the relational algebra *without making use of iteration or recursion statements*.

Much of the derivability power of the relational algebra is obtained from the SELECT, PROJECT, and JOIN operators alone, provided the JOIN is not subject to any implementation restrictions having to do with predefinition of supporting physical access paths. A system has an *unrestricted join capability* if it allows joins to be taken wherein *any* pair of attributes may be matched, providing only that they are defined on the same domain or data type (for our present purpose, it does not matter whether the domain is syntactic or semantic and it does not matter whether the data type is weak or strong, but see [10] for circumstances in which it does matter).

Occasionally, one finds systems in which join is supported only if the attributes to be matched have the same name or are supported by a certain type of predeclared access path. Such restrictions significantly impair the power of the system to derive relations from the base relations. These restrictions consequently reduce the system's capability to handle unanticipated queries by end users and reduce the chances for application programmers to avoid coding iterative loops.

Thus, we say that a data sublanguage *L* has a *relational processing capability* if the transformations specified by the SELECT, PROJECT, and unrestricted JOIN operators of the relational algebra can be specified in *L* without resorting to commands for iteration or recursion. For a database management system to be called *relational* it must support:

(1) Tables without user-visible navigation links between them;

(2) A data sublanguage with at least this (minimal) relational processing capability.

One consequence of this is that a DBMS that does *not* support relational processing should be considered *non-relational*. Such a system might be more appropriately called *tabular*, providing that it supports tables without user-visible navigation links between tables. This term should replace the term "semi-relational" used in [8], because there is a large difference in implementation complexity between tabular systems, in which the programmer does his own navigation, and relational systems, in which the system does the navigation for him, i.e., the system provides *automatic navigation*.

The definition of relational DBMS given above intentionally permits a lot of latitude in the services provided. For example, it

[2] A data sublanguage is a specialized language for database management, supporting at least data definition, data retrieval, insertion, update, and deletion. It need not be computationally complete, and usually is not. In the context of application programming, it is intended to be used in conjunction with one or more programming languages.

is not required that the full relational algebra be supported, and there is no requirement in regard to support of the two integrity rules of the relational model (entity integrity and referential integrity). Full support by a relational system of these latter two parts of the model justifies calling that system *fully relational* [8]. Although we know of no systems that qualify as fully relational today, some are quite close to qualifying, and no doubt will soon do so.

In Fig. 1 we illustrate the distinction between the various kinds of relational and tabular systems. For each class the extent of shading in the **S** box is intended to show the degree of fidelity of members of that class to the structural requirements of the relational model. A similar remark applies to the **M** box with respect to the manipulative requirements, and to the **I** box with respect to the integrity requirements.

m denotes the minimal relational processing capability. **c** denotes relational completeness (a capability corresponding to a two-valued first order predicate logic without nulls). When the manipulation box **M** is fully shaded, this denotes a capability corresponding to the full relational algebra defined in [8] (a three-valued predicate logic with a single kind of null). The question mark in the integrity box for each class except the fully relational is an indication of the present inadequate support for integrity in relational systems. Stronger support for domains and primary keys is needed [10], as well as the kind of facility discussed in [14].

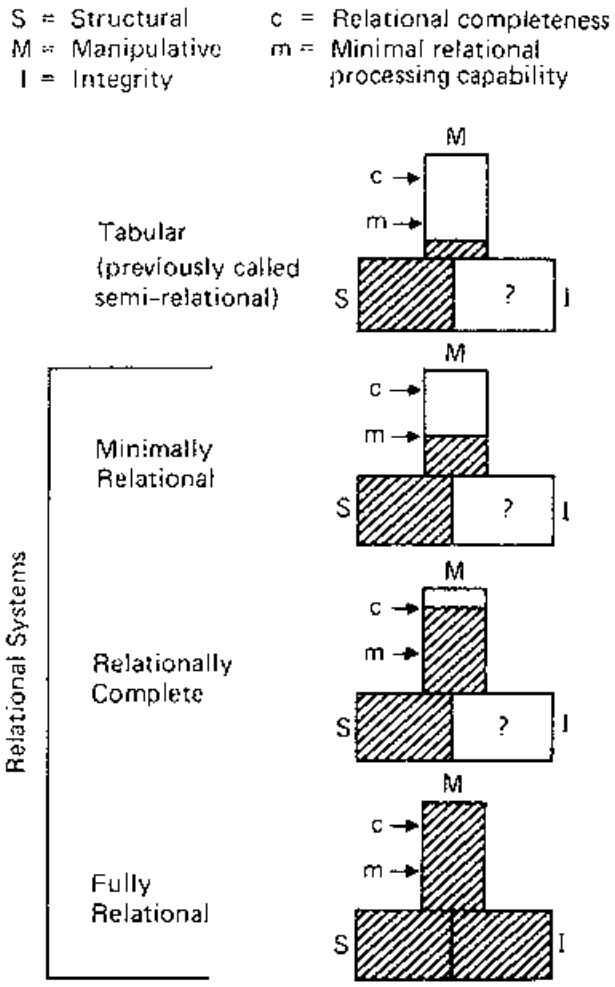


Fig. 1. Classification of DBMS

Note that a relational DBMS may package its relational processing capability in any convenient way. For example, in the INGRES system of Relational Technology, Inc., the RETRIEVE statement of QUEL [29] embodies all three operators (select, project, join) in one statement, in such a way that one can obtain the same effect as any one of the operators or any combination of them.

In the definition of the relational model there are several prohibitions. To cite two examples: user-visible navigation links between tables are ruled out, and database information must not be represented (or hidden) in the ordering of tuples within base relations. Our experience is that DBMS designers who have implemented non-relational systems do not readily understand and accept these prohibitions. By contrast, users enthusiastically understand and accept the enhanced ease of learning and ease of use resulting from these prohibitions.

Incidentally, the Relational Task Group of the American National Standards Institute has recently issued a report [4] on the feasibility of developing a standard for relational database systems. This report contains an enlightening analysis of the features of a dozen relational systems, and its authors clearly understand the relational model.

5. The Uniform Relational Property

In order to have wide applicability most relational DBMS have a data sublanguage which can be interfaced with one or more of the commonly used programming languages (e.g., Cobol, Fortran, PL/I, APL). We shall refer to these latter languages as *host languages*. A relational DBMS usually supports at least one end-user oriented data sublanguage—sometimes several, because the needs of these users may vary. Some prefer string languages such as QUEL or SQL [5], while others prefer the screen-oriented two-dimensional data sublanguage of Query-by-Example [33].

Now, some relational systems (e.g., System R [6], INGRES [29]) support a data sublanguage that is usable in two modes: (1) interactively at a terminal and (2) embedded in an application program written in a host language. There are strong arguments for such a *double-mode* data sublanguage:

(1) With such a language application programmers can separately debug at a terminal the database statements they wish to incorporate in their application programs—people who have used SQL to develop application programs claim that the double-mode feature significantly enhances their productivity;

(2) Such a language significantly enhances communication among programmers, analysts, end users, database administration staff; etc.;

(3) Frivolous distinctions between the languages used in these two modes place an unnecessary learning and memory burden on those users who have to work in both modes.

The importance of this feature in productivity suggests that relational DBMS be classified according to whether they possess this feature or not. Accordingly, we call those relational DBMS that support a double-mode sublanguage *uniform relational*. Thus, a uniform relational DBMS supports relational processing at both an end-user interface and at an application programming interface *using a data sublanguage common to both interfaces.*

The natural term for all other relational DBMS is *non-uniform relational*. An example of a non-uniform relational DBMS is the TANDEM ENCOMPASS [19]. With this system, when retrieving data interactively at a terminal, one uses the relational data sublanguage ENFORM (a language with relational processing capability). When writing a program to retrieve or manipulate data, one uses an extended version of Cobol (a language that does not possess the relational processing capability). Common to both levels of use are the structures: tables without user-visible navigation links between them.

"In the definition of the relational model there are several prohibitions. To cite two examples: user-visible navigation links between tables are ruled out, and database information must not be represented (or hidden) in the ordering of tuples within base relations."

A question that immediately arises is this: how can a data sublanguage with relational processing capability be interfaced with a language such as Cobol or PL/I that can handle data one record at a time only (i.e., that is incapable of treating a set of records as a single operand)? To solve this problem we must separate the following two actions from one another: (1) definition of the relation to be derived; (2) presentation of the derived relation to the host language program.

One solution (adopted in the Peterlee Relational Test Vehicle [31]) is to cast a derived relation in the form of a file that can be read record-by-record by means of host language statements. In this case delivery of records is delegated to the file system used by the pertinent host language.

Another solution (adopted by System R) is to keep the delivery of records under the control of data sublanguage statements and, hence, under the control of the relational DBMS optimizer. A query statement Q of SQL (the data sublanguage of System R) may be embedded in a host language program, using the following kind of phrase (for expository reasons, the syntax is not exactly that of SQL)

DECLARE C CURSOR FOR Q

where C stands for any name chosen by the programmer. Such a statement associates a *cursor* named C with the defining expression Q. Tuples from the derived relation defined by Q are presented to the program one at a time by means of the named cursor. Each time a FETCH per this cursor is executed, the system delivers another tuple from the derived relation. The order of delivery is system-determined unless the SQL statement Q defining the derived relation contains an ORDER BY clause.

It is important to note that in advancing a cursor over a derived relation the programmer is *not* engaging in navigation to some target data. The derived relation is itself the target data! It is the DBMS that determines whether the derived relation should be materialized *en bloc* prior to the cursor-controlled scan or materialized piecemeal during the scan. In either case, it is the system (not the programmer) that selects the access paths by which the derived data is to be generated. This takes a significant burden off the programmer's shoulders, thereby increasing his productivity.

6. Skepticism About Relational Systems

There has been no shortage of skepticism concerning the practicality of the relational approach to database management. Much of this skepticism stems from a lack of understanding, some from a fear of the numerous theoretical investigations that are based on the relational model [1, 2, 15, 16, 24]. Instead of welcoming a theoretical foundation as providing soundness, the attitude seems to be: if it's theoretical, it cannot be practical. The absence of a theoretical foundation for almost all nonrelational DBMS is the prime cause of their *ungepotchket* quality. (This is a Yiddish word, one of whose meanings is patched up.)

On the other hand, it seems reasonable to pose the following two questions:

(1) Can a relational system provide the range of services that we have grown to expect from other DBMS?

(2) If (1) is answered affirmatively, can such a system perform as well as non-relational DBMS?[3]

We took at each of these in turn.

6.1 Range of Services

A full-scale DBMS provides the following capabilities:

- data storage, retrieval, and update;
- a user-accessible catalog for data description;
- transaction support to ensure that all or none of a sequence of database changes are reflected in the pertinent databases (see [17] for an up-to-date summary of transaction technology);
- recovery services in case of failure (system, media, or program);
- concurrency control services to ensure that concurrent transactions behave the same way as if run in some sequential order;
- authorization services to ensure that all access to and manipulation of data be in accordance with specified constraints on users and programs [18];
- integration with support for data communication;
- integrity services to ensure that database states and changes of state conform to specified rules.

Certain relational prototypes developed in the early seventies felt far short of providing all these services (possibly for good reasons). Now, however, several relational systems are available as software products and provide all these services with the exception of the last. Present versions of these products are admittedly weak in the provision of integrity services, but this is rapidly being remedied [10].

Some relational DBMS actually provide more complete data services than the non-relational systems. Three examples follow.

As a first example, relational DBMS support the extraction of all meaningful relations from a database, whereas non-relational

[3] One should bear in mind that the non-relational ones always employ comparatively low level data sublanguages for application programming.

systems support extraction only where there exist statically predefined access paths.

As a second example of the additional services provided by some relational systems, consider views. A *view* is a virtual relation (table) defined by means of an expression or sequence of commands. Although not directly supported by actual data, a view appears to a user as if it were an additional base table kept up-to-date and in a state of integrity with the other base tables. Views are useful for permitting application programs and users at terminals to interact with constant view structures, even when the base tables themselves are undergoing structural changes at the *logical* level (providing that the pertinent views are still definable from the new base tables). They are also useful in restricting the scope of access of programs and users. Non-relational systems either do not support views at all or else support much more primitive counterparts, such as the CODASYL subschema.

As a third example, some systems (e.g., SQL/DS [28] and its prototype predecessor System R) permit a variety of changes to be made to the logical and physical organization of the data dynamically—while transactions are in progress. These changes rarely require application programs to be recoded. Thus, there is less of a program maintenance burden, leaving programmers to be more productive doing development rather than maintenance. This capability is made possible in SQL/DS by the fact that the system has complete control over access path selection.

In non-relational systems such changes would normally require all other database activities including transactions in progress to be brought to a halt. The database then remains out of action until the organizational changes are completed and any necessary recompiling done.

6.2 Performance

Naturally, people would hesitate to use relational systems if these systems were sluggish in performance. All too often, erroneous conclusions are drawn about the performance of relational systems by comparing the time it might take for one of these systems to execute a complex transaction with the time a non-relational system might take to execute an extremely simple transaction. To arrive at a fair performance comparison, one must compare these systems on the same tasks or applications. We shall present arguments to show why relational systems should be able to compete successfully with non-relational systems.

Good performance is determined by two factors: (1) the system must support performance-oriented physical data structures; (2) high-level language requests for data must be compiled into lower-level code sequences at least as good as the average application programmer can produce by hand.

The first step in the argument is that a program written in a Cobol-level language can be made to perform efficiently on large databases containing production data structured in tabular form with no user-visible navigation links between them. This step in the argument is supported by the following information [19]: as of August 1981, Tandem Computer Corp. had manufactured and installed 760 systems; of these, over 700 were making use of the Tandem ENCOMPASS relational database management system to support databases containing production data. Tandem has committed its own manufacturing database to the care of ENCOMPASS. ENCOMPASS does not support links between the database tables, either user-visible (navigation) links or user-invisible (access method) links.

"Instead of welcoming a theoretical foundation as providing soundness, the attitude seems to be: if it's theoretical, it cannot be practical. The absence of a theoretical foundation for almost all nonrelational DBMS is the prime cause of their 'ungepotchket' (patched up) quality."

In the second step of the argument, suppose we take the application programs in the above-cited installations and replace the database retrieval and manipulation statements by statements in a database sublanguage with a relational processing capability (e.g., SQL). Clearly, to obtain good performance with such a high level language, it is essential that it be compiled into object code (instead of being interpreted), and it is essential that that object code be efficient.

Compilation is used in System R and its product version SQL/DS. In 1976 Raymond Lorie developed an ingenious pre- and post-compiling scheme for coping with dynamic changes in access paths [21]. It also copes with early (and hence efficient) authorization and integrity checking (the latter, however, is not yet implemented). This scheme calls for compiling in a rather special way the SQL statements embedded in a host language pro-

gram. This compilation step transforms the SQL statements into appropriate CALLs within the source program together with access modules containing object code. These modules are then stored in the database for later use at runtime. The code in these access modules is generated by the system so as to optimize the sequencing of the major operations and the selection of access paths to provide runtime efficiency. After this precompilation step, the application program is compiled by a regular compiler for the pertinent host language. If at any subsequent time one or more of the access paths is removed and an attempt is made to run the program, enough source information has been retained in the access module to enable the system to re-compile a new access module that exploits the now existing access paths *without requiring a re-compilation of the application program*.

"All too often, erroneous conclusions are drawn about the performance of relational systems by comparing the time it might take for one of these systems to execute a complex transaction with the time a non-relational system might take to execute an extremely simple transaction."

Incidentally, the same data sublanguage compiler is used on ad hoc queries submitted interactively from a terminal and also on queries that are dynamically generated during the execution of a program (e.g., from parameters submitted interactively). Immediately after compilation, such queries are executed and, with the exception of the simplest of queries, the performance is better than that of an interpreter.

The generation of access modules (whether at the initial compiling or re-compiling stage) entails a quite sophisticated optimization scheme [27], which makes use of system-maintained statistics that would not normally be within the programmer's knowledge. Thus, only on the simplest of all transactions would it be possible for an average application programmer to compete with this optimizer in generation of efficient code. Any attempts to compete are bound to reduce the programmer's productivity. Thus, the price paid for extra compile-time overhead would seem to be well worth paying.

Assuming non-linked tabular structures in both cases, we can expect SQL/DS to generate code comparable with average hand-written code in many simple cases, and superior in many complex cases. Many commercial transactions are extremely simple. For example, one may need to look up a record for a particular railroad wagon to find out where it is or find the balance in someone's savings account. If suitably fast access paths are supported (e.g., hashing), there is no reason why a high-level language such as SQL, QUEL, or QBE should result in less efficient runtime code for these simple transactions than a lower level language, even though such transactions make little use of the optimizing capability of the high-level data sublanguage compiler.

7. Future Directions

If we are to use relational database as a foundation for productivity, we need to know what sort of developments may lie ahead for relational systems.

Let us deal with near-term developments first. In some relational systems stronger support is needed for domains and primary keys per suggestions in [10]. As already noted, all relational systems need upgrading with regard to automatic adherence to integrity constraints. Existing constraints on updating join-type views need to be relaxed (where theoretically possible), and progress is being made on this problem [20]. Support for outer joins is needed.

Marked improvements are being made in optimizing technology, so we may reasonably expect further improvements in performance. In certain products, such as the ICL CAFS [22] and the Britton-Lee IDM500 [13], special hardware support has been implemented. Special hardware may help performance in certain types of applications. However, in the majority of applications dealing with formatted databases, software-implemented relational systems can compete in performance with software-implemented non-relational systems.

At present, most relational systems do not provide any special support for engineering and scientific databases. Such support, including interfacing with Fortran, is clearly needed and can be expected.

Catalogs in relational systems already consist of additional relations that can be interrogated just like the rest of the database using the same query language. A natural development that can and should be swiftly put in place is the expansion of these catalogs into full-fledged active dictionaries to provide additional on-line data control.

Finally, in the near term, we may expect database design aids suited for use with relational systems both at the logical and physical levels. In the longer term we may expect support for relational databases distributed over a communications network [25, 30, 32] and managed in such a way that application programs and interactive users can manipulate the data (1) as if all of it were stored at the local node—*location transparency*—and (2) as if no data were replicated anywhere—*replication transparency*. All three of the projects cited above are based on the relational model. One important reason for this is that relational databases offer great decomposition flexibility when planning how a database is to be distributed over a network of computer systems, and great recomposition power for dynamic combination of decentralized information. By contrast, CODASYL DBTG databases are very difficult to decompose and recompose due to the entanglement of the owner-member navigation links. This property makes the CODASYL approach extremely difficult to adapt to a distributed database environment and may well prove to be its downfall. A second reason for use of the relational model is that it offers concise high level data sublanguages for transmitting requests for data from node to node.

The ongoing work in extending the relational model to capture in a formal way more meaning of the data can be expected to lead to the incorporation of this meaning in the database catalog in order to factor it out of application programs and make these programs even more concise and simple. Here, we are, of course, talking about meaning that is represented in such a way that the system can understand it and act upon it.

Improved theories are being developed for handling missing data and inapplicable data (see for example [3]). This work should yield improved treatment of null values.

As it stands today, relational database is best suited to data with a rather regular or homogeneous structure. Can we retain the advantages of the relational approach while handling heterogeneous data also? Such data may include images, text, and miscellaneous facts. An affirmative answer is expected, and some research is in progress on this subject, but more is needed.

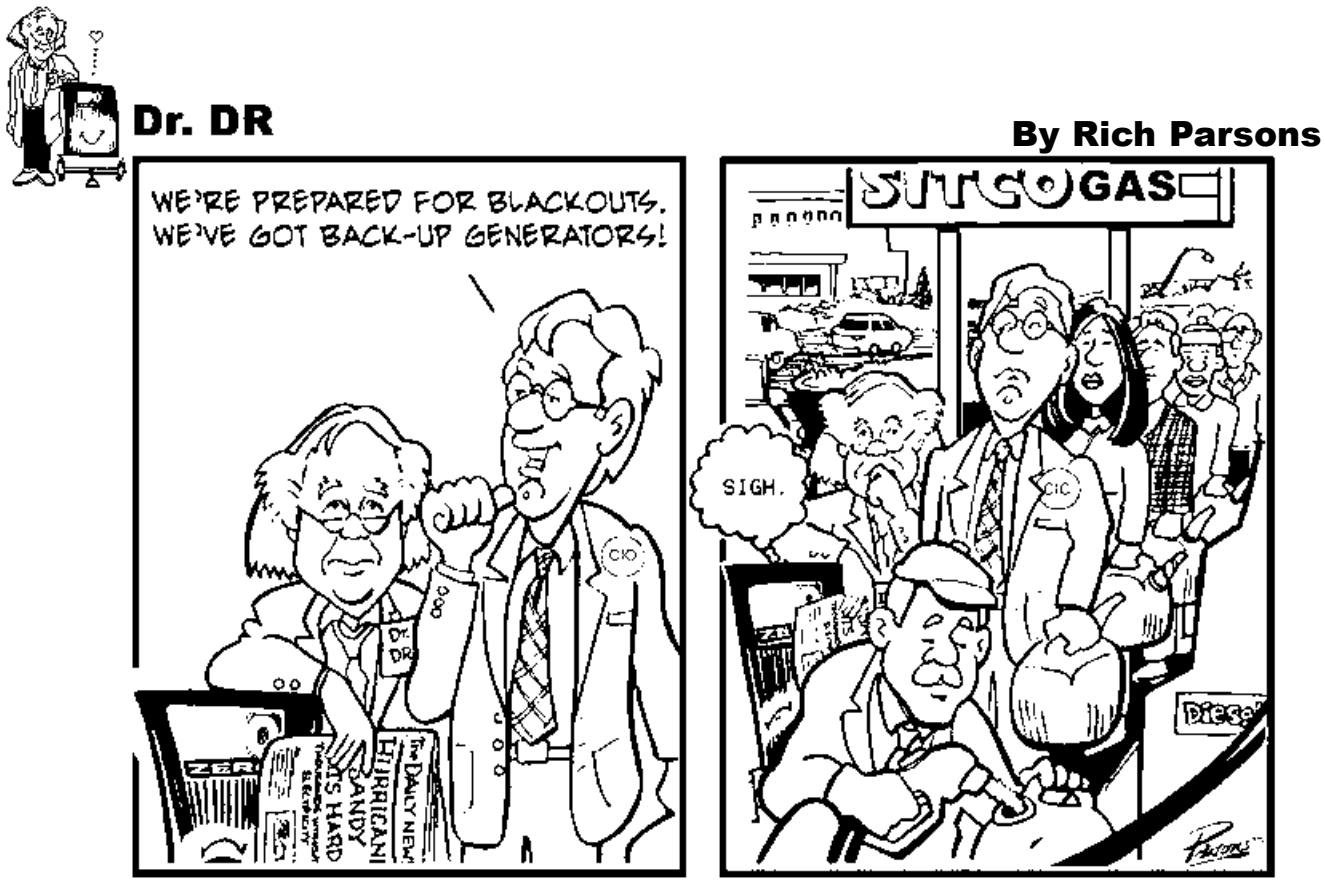
Considerable research is needed to achieve a rapprochement between database languages and programming languages. Pascal/R [26] is a good example of work in this direction. Ongoing investigations focus on the incorporation of abstract data types into database languages on the one hand [12] and relational processing into programming languages on the other.

8. Conclusions

We have presented a series of arguments to support the claim that relational database technology offers dramatic improvements in productivity both for end users and for application programmers. The arguments center on the data independence, structural simplicity, and relational processing defined in the relational model and implemented in relational database management systems. All three of these features simplify the task of developing application programs and the formulation of queries and updates to be submitted from a terminal. In addition, the first feature tends to keep programs viable in the face of organizational and descriptive changes in the database and therefore reduces the effort that is normally diverted into the maintenance of programs.

"Can we retain the advantages of the relational approach while handling heterogeneous data also? Such data may include images, text, and miscellaneous facts. An affirmative answer is expected, and some research is in progress on this subject, but more is needed."

Why, then, does the title of this paper suggest that relational database provides only a foundation for improved productivity and not the total solution? The reason is simple: relational database deals only with the shared data component of application programs and end-user interactions. There are numerous complementary technologies that may help with other components or aspects, for example, programming languages that support relational processing and improved checking of data types, improved editors that understand more of the language being used, etc. We use the term "foundation," because interaction with shared data (whether by program or via terminal) represents the core of so much data processing activity. The practicality of the relational approach has been proven by the test and production installations that are already in operation. Accordingly, with relational systems we can now look forward to the productivity boost that we all hoped DBMS would provide in the first place.



Dr. DR is brought to you by Axxana.

Acknowledgements

I would like to express my indebtedness to the System R development team at IBM Research, San Jose for developing a full-scale, uniform relational prototype that entailed numerous language and system innovations; to the development team at the IBM Laboratory, Endicott, N.Y. for the professional way in which they converted System R into product form; to the various teams at universities, hardware manufacturers, software firms, and user installations, who designed and implemented working relational systems; to the QBE team at IBM Yorktown Heights, N.Y.; to the PRTV team at the IBM Scientific Centre in England; and to the numerous contributors to database theory who have used the relational model as a cornerstone. A special acknowledgement is due to the very few colleagues who saw something worth supporting in the early stages, particularly, Chris Date and Sharon Weinberg. Finally, it was Sharon Weinberg who suggested the theme of this paper. ▲

References

1. Beeri, C., Bernstein, P., Goodman, N. A sophisticate's introduction to database normalization theory. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept. 1978.
2. Bernstein, P.A., Goodman, N., Lai, M-Y. Laying phantoms to rest. Report TR-03-81, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., 1981.
3. Biskup, J.A. A formal approach to null values in database relations. *Proc. Workshop on Formal Bases for Data Bases*, Toulouse, France, Dec 1979; published in [16] (see below) pp 299–342.
4. Brodie, M. and Schmidt, J. (Eds), Report of the ANSI Relational Task Group., (to be published ACM SIGMOD Record).
5. Chamberlin, D.D., et al. SEQUEL2: A unified approach to data definition, manipulation, and control. *IBM J. Res. & Dev., 20*, 6, (Nov. 1976) 560–565.
6. Chamberlin, D.D., et al. A history and evaluation of system R. *Comm. ACM, 24*, 10, (Oct. 1981) 632–646.
7. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM, 13*, 6, (June 1970) 377–387.
8. Codd, E.F. Extending the database relational model to capture more meaning. *ACM TODS, 4*, 4, (Dec. 1979) 397–434.
9. Codd, E.F. Data models in database management. *ACM SIGMOD Record, 11*, 2, (Feb. 1981) 112–114.
10. Codd, E.F. The capabilities of relational database management systems. *Proc. Convencio Information Llatina*, Barcelona, Spain, June 9–12, 1981, pp 13–26; also available as Report 3132, IBM Research Lab., San Jose, Calif.
11. Date, C.J. Referential integrity. *Proc. Very Large Data Bases*, Cannes, France, September 9–11, 1981, pp 2–12.
12. Ehrig, H., and Weber, H. Algebraic specification schemes for data base systems. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept 13–15, 1978, 427–440.
13. Epstein, R., and Hawthorne, P. Design decisions for tile intelligent database machine. *Proc. NCC 1980, AFIPS, Vol. 49. May 1980*, pp 237–241.
14. Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for database integrity. *Proc. Very Large Data Bases*, Framingham, Mass., Sept. 1975, pp 48–68.
15. Fagin, R. Horn clauses and database dependencies. *Proc. 1980 ACM SIGACT Symp. on Theory of Computing*, Los Angeles, CA, pp 123–134.
16. Gallaire, H., Minker, J., and Nicolas, J.M. *Advances in Data Base Theory*. Vol 1, Plenum Press, New York, 1981.
17. Gray, J. The transaction concept: virtues and limitations, *Proc. Very Large Data Bases*, Cannes, France, September 9–11, 1981, pp 144–154.
18. Griffiths, P.G., and Wade, B.W. An authorization mechanism for a relational database system. *ACM TODS, 1*, 3, (Sept 1976) 242–255.
19. Held, G. ENCOMPASS: A relational data manager. Data Base/81, Western Institute of Computer Science, Univ. of Santa Clara, Santa Clara, Calif., August 24–28, 1981.
20. Keller, A.M. Updates to relational databases through views involving joins. Report RJ3282, IBM Research Laboratory, San Jose, Calif., October 27, 198l.
21. Lorie, R.A, and Nilsson, J.F. An access specification language for a relational data base system. *IBM J. Res. & Dev., 23*, 3, (May 1979) 286–298.
22. Maller, V.A.J. The content addressable file store—CAFS. *ICL Technical J., 1*, 3, (Nov. 1979) 265–279.
23. Reisner, P. Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys, 13*, 1, (March 1981) 13–31.
24. Rissanen, J. Theory of relations for databases—A tutorial survey. *Proc. Symp. on Mathematical Foundations of Computer Science*, Zakopane, Poland, September 1978, Lecture Notes in Computer Science, No. 64, Springer Verlag, New York, 1978.
25. Rothnie, J.B., Jr. et al. Introduction to a system for distributed databases (SDD-1). *ACM TODS, 5*, 1, (March 1980) 1–17.
26. Schmidt, J.W. Some high level language constructs for data of type relation. *ACM TODS, 2*, 3, (Sept 1977) 247–261.
27. Selinger, P.G., et al. Access path selection in a relational database system. *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, Boston, MA, May 1979, pp 23–34.
28. ---, SQL/Data system for VSE: A relational data system for application development. IBM Corp. Data Processing Division, White Plains, N.Y., G320-6590, Feb 1981.
29. Stonebraker, M.R., et al. The design and implementation of INGRES, *ACM TODS, 1*, 3, (Sept. 1976) 189–222.
30. Stonebraker, M.R., and Neuhold, E.J. A distributed data base version of INGRES. *Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence-Berkeley Lab., Berkeley, Calif., May 1977, pp 19–36.
31. Todd., S.J.P. The Peterlee relational test vehicle—A system overview. *IBM Systems J., 15*, 4, 1976, 285–308.
32. Williams, R. et al. R*: An overview of the architecture. Report RJ3325, IBM Research Laboratory, San Jose, Calif., October 27, 1981.
33. Zloof, M.M. Query by example. Proc. NCC, AFIPS Vol 44, May 1975, pp 431–438.

What Is a Relational Database Management System

by Iggy Fernandez

I'd like to spend a few minutes at the outset considering what the word *database* signifies. I'll begin by saying that databases can contain data that is confidential and must be protected from prying eyes. Only authorized users should be able to access the data, their privileges must be suitably restricted, and their actions must be logged. Even if the data in the databases is for public consumption, you still may need to restrict who can update the data, who can delete from it, and who can add to it. Competent security management is, therefore, part of the database administrator's job.

Databases can be critical to an organization's ability to function properly. Organizations such as banks and e-commerce websites require their databases to be available around the clock. Competent *availability management* is thus an important part of the database administrator's job. In the event of a disaster such as a flood or fire, the databases may have to be relocated to an alternative location using backups. Competent *continuity management* is therefore another important element of the database administrator's job. The database also needs competent *change management* to protect a database from unauthorized or badly tested changes, *incident management* to detect problems and restore service quickly, *problem management* to provide permanent fixes for known issues, *configuration management* to document infrastructure components and their dependencies, and *release management* to bring discipline to the never-ending task of applying patches and upgrades to software and hardware.

I'll also observe that databases can be very big. The first database I worked with, for the semiconductor manufacturing giant Intel, was less than 100 MB in size and had only a few dozen data tables. Today, databases used by enterprise application suites like PeopleSoft, Siebel, and Oracle Applications are tens or hundreds of gigabytes in size and might have 10,000 tables or more. One reason databases are now so large is that advancements in magnetic disk storage technology have made it feasible to efficiently store and retrieve large quantities of nontextual data such as pictures and sound. Databases can grow rapidly, and you need to plan for growth. In addition, database applications may consume huge amounts of computing resources. Capacity management is thus another important element of the database administrator's job; the database needs a capacity plan that accommodates both continuous data growth and increasing needs for computing resources.

What Is a *Relational* Database?

Relational database theory was laid out by Codd in 1970 in a paper titled "A Relational Model for Data for Large Shared Data Banks." His theory was meant as an alternative to the "programmer as navigator" paradigm that was prevalent in his day.

In pre-relational databases, records were chained together by pointers, as illustrated in the following figures. Each chain has an owner and zero or more members. For example, all the employee records in a department could be chained to the corresponding department record in the departments table. In such a scheme, each employee record points to the next and previous records in the chain as well as to the department record. To list all the employees in a department, you would first navigate to the unique department record (typically using the direct-access technique known as *hashing*) and then follow the chain of employee records.

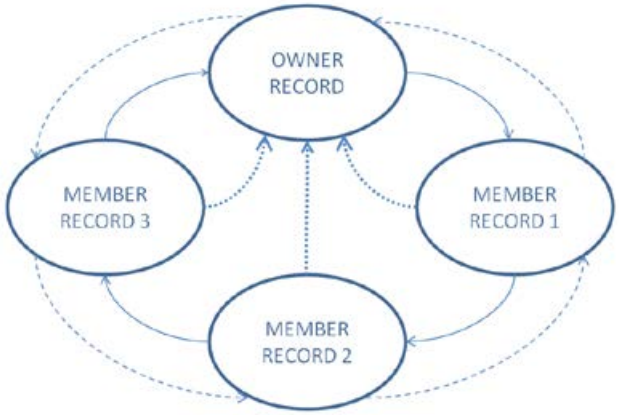


Figure 1.

This scheme was invented by Charles Bachman, who received the ACM Turing Award in 1973 for his achievement. In his Turing Award lecture, titled "The Programmer as Navigator," Bachman enumerated seven ways in which you can navigate through such a database:

1. Records can be retrieved sequentially.
2. A specific record can be retrieved using its physical address if it's available.
3. A specific record can be retrieved using a unique key. Either a unique index or hash addressing makes this possible.
4. Multiple records can be retrieved using a non-unique key. A non-unique index is necessary.
5. Starting from an owner record, all the records in a chain can be retrieved.
6. Starting from any member record in a chain, the prior or next record in the chain can be retrieved.

7. Starting at any member record in a chain, the owner of the chain can be retrieved.

Bachman noted, "Each of these access methods is interesting in itself, and all are very useful. However, it is the synergistic usage of the entire collection which gives the programmer great and expanded powers to come and go within a large database while accessing only those records of interest in responding to inquiries and updating the database in anticipation of future inquiries."

An example of a pre-relational database technology is so-called *network database technology*, one of the best examples of which was DEC/DBMS, created by Digital Equipment Corporation for the VAX/VMS and OpenVMS platforms—which still survives today as Oracle/DBMS. Yes, it's strange but it's true—Oracle, the maker of the world's dominant relational database technology, also sells a pre-relational database technology. According to Oracle, Oracle/DBMS is a very powerful, reliable, and sophisticated database technology that has continued relevance and that Oracle is committed to supporting.

In Bachman's scheme, you need to know the access paths defined in the database. In 1979, Codd made the startling statement that programmers *need not* and *should not* have to be concerned about the access paths defined in the database. The opening words of the first paper on the relational model were, "Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)" ("A Relational Model of Data for Large Shared Data Banks").

Productivity and ease of use were the stated goals of the relational model. In "Normalized Data Base Structure: A Brief Tutorial" (1971), Codd said,

> What is less understandable is the trend toward more and more complexity in the data structures with which application programmers and terminal users directly interact. Surely, in the choice of logical data structures that a system is to support, there is one consideration of absolutely paramount importance—and that is the convenience of the majority of users. . . . To make formatted data bases readily accessible to users (especially casual users) who have little or no training in programming we must provide the simplest possible data structures and almost natural language. . . . What could be a simpler, more universally needed, and more universally understood data structure than a table?

As IBM researcher Donald Chamberlin recalled later (*The 1995 SQL Reunion: People, Projects, and Politics*):

> [Codd] gave a seminar and a lot of us went to listen to him. This was as I say a revelation for me because Codd had a bunch of queries that were fairly complicated queries and since I'd been studying CODASYL, I could imagine how those queries would have been represented in CODASYL by programs that were five pages long that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. These would be queries like, "Find the employees who earn more than their managers." He just whacked them out and you could sort of read them, and they weren't complicated at all, and I said, "Wow." This was kind of a conversion experience for me, that I understood what the relational thing was about after that.

Donald Chamberlin and fellow IBM researcher Raymond Boyce went on to create the first relational query language based on Codd's proposals and described it in a short paper titled "SEQUEL: A Structured English Query Language" (1974). The acronym SEQUEL was later shortened to SQL because SEQUEL was a trademarked name.

Codd emphasized the productivity benefits of the relational model in his acceptance speech for the 1981 Turing Award:

> It is well known that the growth in demands from end users for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. There are two complementary approaches to attacking this problem (and both approaches are needed): one is to put end users into direct touch with the information stored in computers; the other is to increase the productivity of data processing professionals in the development of application programs. It is less well known that a single technology, relational database management, provides a practical foundation to both approaches.

In fact, the ubiquitous data-access language SQL was originally intended for the use of non-programmers. As explained by the creators of SQL in their 1974 paper, there is "a large class of users who, while they are not computer specialists, would be willing to learn to interact with a computer in a reasonably high-level, non-procedural query language. Examples of such users are *accountants, engineers, architects, and urban planners* [emphasis added]. It's for this class of users that SEQUEL is intended."

Secret Sauce

Codd's secret sauce was *relational algebra*, a collection of operations that could be used to combine tables. Just as you can combine numbers using the operations of addition, subtraction, multiplication, and division, you can combine tables using operations like *selection*, *projection*, *union*, *difference*, and *join* (more precisely, *Cartesian join*), listed in the following table.

Why did Codd name this *relational* algebra? Codd based his theory on rigorous mathematical principles and used the esoteric mathematical term *relation* to denote what is loosely referred to as a table.

Leonardo da Vinci said, "Those who are in love with practice without knowledge are like the sailor who gets into a ship without rudder or compass and who never can be certain [where] he is going. Practice must always be founded on sound theory." To competently administer a relational database management system like Oracle, we must know what makes a "relational" database relational and what a database "management" system manages.

Operator	Operator
Selection	Form another table by extracting a subset of the rows of a table of interest using some criteria. This can be expressed in SQL as follows (the "*" character is a wildcard that matches all columns in the table that is being operated on): select * from [table] where [criteria]
Projection	Form another table by extracting a subset of the columns of a table of interest. Any duplicate rows that are formed as a result of the projection operation are eliminated: select [column list] from [table]
Union	Form another table by selecting all rows from two tables of interest. If the first table has 10 rows and the second table has 20 rows, then the resulting table will have at most 30 rows, because duplicates are eliminated from the result: select * from [first table] union select * from [second table]
Difference	Form another table by extracting from one table of interest only those rows that don't occur in a second table: select * from [first table] minus select * from [second table]
Join	select * from [first table], [second table]

Worked Example

Let's use the five operations defined in the above table to answer this question: "Which employees have worked in all accounting positions—that is, those for which the job_id starts with the characters AC?" The current job of each employee is stored in the job_id column of the employees table. Any previous jobs are held in the job_history table. The list of job titles is held in the jobs table.

Here is the description of the employees table:

```
Name                 Null?    Type
-------------------- -------- -------------
EMPLOYEE_ID          NOT NULL NUMBER(6)
FIRST_NAME                    VARCHAR2(20)
LAST_NAME            NOT NULL VARCHAR2(25)
EMAIL                NOT NULL VARCHAR2(25)
PHONE_NUMBER                  VARCHAR2(20)
HIRE_DATE            NOT NULL DATE
JOB_ID               NOT NULL VARCHAR2(10)
SALARY                        NUMBER(8,2)
COMMISSION_PCT                NUMBER(2,2)
MANAGER_ID                    NUMBER(6)
DEPARTMENT_ID                 NUMBER(4)
```

Here is the description of the job_history table:

```
Name                 Null?    Type
-------------------- -------- -------------
EMPLOYEE_ID          NOT NULL NUMBER(6)
START_DATE           NOT NULL DATE
END_DATE             NOT NULL DATE
JOB_ID               NOT NULL VARCHAR2(10)
DEPARTMENT_ID                 NUMBER(4)
```

Here is the description of the jobs table:

```
Name                 Null?    Type
-------------------- -------- -------------
JOB_ID               NOT NULL VARCHAR2(10)
JOB_TITLE            NOT NULL VARCHAR2(35)
MIN_SALARY                    NUMBER(6)
MAX_SALARY                    NUMBER(6)
```

Given these three tables, you can execute the following steps to answer the business question that has been posed. Each step builds on the previous step and a SQL query slowly takes shape. You can follow along in the SQL worksheet in SQL Developer.

1. This step uses only the employee_id column from the employees table. To do this, you need the projection operation. The employees table contains 107 rows, so this command also produces 107 rows. Only the first five rows are shown here:

```
select employee_id
from employees

       100
       101
       102
       103
       104
```

Note that certain formatting aspects of SQL statements, such as lowercase, uppercase, white space, and line feeds, are immaterial except in the case of string literals—that is, strings of characters enclosed within quotation marks.

2. This step uses only the job_id column from the jobs table. To obtain this, you need the projection operation again. The jobs table contains 19 rows, so this command also produces 19 rows. Only the first five rows are shown here:

```
select job_id
from jobs

AC_ACCOUNT
AC_MGR
AD_ASST
AD_PRES
AD_VP
```

3. Remember that the result of any relational operation is always another table. You need a subset of rows from the table created by the projection operation used in step 2. To obtain this, you need the selection operation, as shown in the following SQL command and its results. "*" is a wildcard that matches all the columns of a table. "%" is a wildcard that matches any combination of characters. Note that the "table" that is operated on is actually the SQL command from step 2. The result contains only two rows:

```
select *
from (select job_id from jobs)
where job_id like 'AC%'

AC_ACCOUNT
AC_MGR
```

You can streamline this SQL command as follows. This version expresses both the projection from step 2 and the selection from step 3 using a unified syntax. Read it carefully, and make sure you understand it:

```
select job_id from jobs
where job_id like 'AC%'
```

4. You need the job title of every employee; that is, you need the job_id column from the employees table. The employees table has 107 rows, so the resulting table also has 107 rows; five of them are shown here:

```
select employee_id, job_id
from employees

       100 AD_PRES
       101 AD_VP
       102 AD_VP
       103 IT_PROG
       104 IT_PROG
```

5. Next, you need the employee_id and job_id columns from the job_history table. The jobs table contains 19 rows, so this command also produces 19 rows. Only the first five rows are shown here:

```
select employee_id, job_id
from job_history

       101 AC_ACCOUNT
       200 AC_ACCOUNT
       101 AC_MGR
       200 AD_ASST
       102 IT_PROG
```

6. Remember that the current job of each employee is stored in the job_id column of the employees table. Any previous jobs are held in the job_history table. The complete job history of any employee is therefore the union of the tables created in step 4 and step 5:

```
select employee_id, job_id
from employees
union
select employee_id, job_id
from job_history
```

7. You need to join the tables created in step 1 and step 3. The resulting table contains all possible pairings of the 107 emp_id values in the employees table with the two job_id values of interest. There are 214 such pairings, a few of which are shown next:

```
select *
from
  (select employee_id from employees),
  (select job_id from jobs where job_id like 'AC%')

       100 AC_ACCOUNT
       101 AC_ACCOUNT
       102 AC_ACCOUNT
       103 AC_ACCOUNT
       104 AC_ACCOUNT
```

You can streamline this SQL command as follows. This version expresses the projections from step 1 and step 2, the selection from step 3, as well as the join in the current step using a unified syntax. Read it carefully and make sure you understand it. This pattern of combining multiple projection, join, and selection operations is the most important SQL pattern, so understanding it is vital. Note that you prefix the table names to the column names. Such prefixes are required whenever there are ambiguities. In this case, there is a job_id column in the employees table in addition to the jobs table:

```
select employees.employee_id, jobs.job_id
from employees, jobs
where jobs.job_id like 'AC%'
```

8. From the table created in step 7, you need to subtract the rows in the table created in step 6. To do this, you need the difference operation—the appropriate SQL keyword being minus. The resulting table contains those pairings of employee_id and job_id that are not found in the job_history table. Here is the SQL command you need. The resulting table contains exactly 211 rows, a few of which are shown:

```
select employees.employee_id, jobs.job_id
from employees, jobs
where jobs.job_id like 'AC%'
minus
select employee_id, job_id
from job_history

       100 AC_ACCOUNT
       100 AC_MGR
       102 AC_ACCOUNT
       102 AC_MGR
       103 AC_ACCOUNT
```

9. Thus far, you've obtained pairings of employee_id and job_id that are not found in the employee's job history—that is, the table constructed in step 6. Any employee who participates in such a pairing is not an employee of interest; that is, any employee who participates in such a pairing isn't an employee who has worked in all positions for which the job_id starts with the characters AC. The first column of this table therefore contains the employees in which you're not interested. You need another projection operation:

```
select employee_id from
(
  select employees.employee_id, jobs.job_id
  from employees, jobs
  where jobs.job_id like 'AC%'
  minus
  select employee_id, job_id
  from job_history
)

       100
       100
       102
       102
       103
```

10. You've identified the employees who don't satisfy your criteria. All you have to do is to eliminate them from the table created in step 1! Exactly one employee satisfies your criteria:

```
select employee_id
from employees
minus
select employee_id from
(
  select employees.employee_id, jobs.job_id
```

```
from employees, jobs
where jobs.job_id like 'AC%'
minus
(
  select employee_id, job_id
  from job_history
  union
  select employee_id, job_id
  from job_history
)
)

       101
```

You had to string together 10 operations—5 projection operations, 1 selection operation, 1 union operation, 1 join operation and 2 difference operations—to produce the final answer:

The Definition at Last

We're now ready to define the term "relational database":

> A relational database is a database in which: The data is perceived by the user as tables (and nothing but tables) and the operators available to the user for (for example) retrieval are operators that derive "new" tables from "old" ones.—Chris Date, *An Introduction to Database Systems,* 8th ed. (Addison-Wesley, 2003)

What Is a Database Management System?

Database management systems such as Oracle are the interface between users and databases. Database management systems differ in the range of features they provide, but all of them offer certain core features, such as transaction management, data integrity, and security. And, of course, they offer the ability to create databases and to define their structure, as well as to store, retrieve, update, and delete the data in the databases.

Transaction Management

A *transaction* is a unit of work that may involve several small steps, all of which are necessary in order not to compromise the integrity of the database. For example, a logical operation such as inserting a row into a table may involve several physical operations, such as index updates, trigger operations, and recursive operations. A transaction may also involve multiple logical operations. For example, transferring money from one bank account to another may require that two separate rows be updated. A DBMS needs to ensure that transactions are atomic, consistent, isolated, and durable.

The Atomicity Property of Transactions

It's always possible for a transaction to fail at any intermediate step. For example, users may lose their connection to the database, or the database may run out of space and may not be able to accommodate new data that a user is trying to store. If a failure occurs, the database management system performs automatic rollback of the work that has been performed so far. Transactions are therefore atomic or indivisible from a logical perspective. The end of a transaction is indicated by an explicit instruction such as COMMIT.

The Consistency Property of Transactions

Transactions also have the consistency property. That is, they don't compromise the integrity of the database. However, it's easy to see that a database may be temporarily inconsistent during the operation of the transaction. In the previous example, the database is in an inconsistent state when money has been subtracted from the balance in the first account but has not yet been added to the balance in the second account.

The Isolation Property of Transactions

Transactions also have the isolation property; that is, concurrently occurring transactions must not interact in ways that produce incorrect results. A database management system must be capable of ensuring that the results produced by concurrently executing transactions are *serializable*: the outcome must be the same as if the transactions were executed in serial fashion instead of concurrently.

For example, suppose that one transaction is withdrawing money from a bank customer's checking account, and another transaction is simultaneously withdrawing money from the same customer's savings account. Let's assume that negative balances are permitted as long as the sum of the balances in the two accounts isn't negative. Suppose that the operation of the two transactions proceeds in such a way that each transaction determines the balances in both accounts before either of them has had an opportunity to update either balance. Unless the database management system does something to prevent it, this can potentially result in a negative sum.

The Durability Property of Transactions

Transactions also have the durability property. This means that once all the steps in a transaction have been successfully completed and the user notified, the results must be considered permanent even if there is a subsequent computer failure, such as a damaged disk.

Data Integrity

Data loses its value if it can't be trusted to be correct. A database management system provides the ability to define and enforce what are called *integrity constraints*. These are rules you define, with which data in the database must comply. For example, you can require that employees not be hired without being given a salary or an hourly pay rate.

The database management system rejects any attempt to violate the integrity constraints when inserting, updating, or deleting data records, and typically displays an appropriate error code and message. In fact, the very first Oracle error code, ORA-00001, relates to attempts to violate an integrity constraint. It's possible to enforce arbitrary constraints using trigger operations; these can include checks that are as complex as necessary, but the more common types of constraints are check constraints, uniqueness constraints, and referential constraints. These work as follows:

- *Check constraints*: Check constraints are usually simple checks on the value of a data item. For example, a price quote must not be less than $0.00.
- *Uniqueness constraints*: A uniqueness constraint requires that some part of a record be unique. For example, two employees may not have the same employee number. A unique part of a record is called a *candidate key*, and one of the candidate keys is designated as the primary key. Intuitively, you expect every record to have at least one candidate key; otherwise, you would have no way of

specifying which records you needed. Note that the candidate key can consist of a single item from the data record, a combination of items, or even all the items.

- *Referential constraints*: Consider an employee database in which all payments to employees are recorded in a table called SALARY. The employee number in a salary record must obviously correspond to the employee number in some employee record; this is an example of a referential constraint.

Data Security

A database management system gives the owners of the data a lot of control over it—they can delegate limited rights to others if they choose to. It also gives the database administrator the ability to restrict and monitor the actions of users. For example, the database administrator can disable the password of an employee who leaves the company, to prevent them from gaining access to the database. Relational database management systems use techniques such as *views* (virtual tables defined in terms of other tables) and query modification to give individual users access to just those portions of data they're authorized to use.

Oracle also offers extensive query-modification capabilities under the heading Virtual Private Database (VPD). Additional query restrictions can be silently and transparently appended to the query by *policy functions* associated with the tables in the query. For example, the policy functions may generate additional query restrictions that allow employees to retrieve information only about themselves and their direct reports between the hours of 9 a.m. and 5 p.m. on weekdays only.

What Makes a Relational Database Management System Relational?

Having already discussed the meaning of both *relational database* and *database management system*, it may appear that the subject is settled. But the natural implications of the relational model are so numerous and profound that critics contend that, even today, a "truly relational" database management system doesn't exist. Codd listed more than 300 separate requirements that a database management system must meet in order to fulfill his vision properly, and I have time for just one of them: physical data independence. Here is the relevant quote from Codd's book:

> RP-1 Physical Data Independence: The DBMS permits a suitably authorized user to make changes in storage representation, in access method, or in both—for example, for performance reasons. Application programs and terminal activities remain logically unimpaired whenever any such changes are made.—E. F. Codd, The Relational Model for Database Management: Version 2 (Addison Wesley, 1990)

What Codd meant was that you and I shouldn't have to worry about implementation details such as the storage structures used to store data.

Getting Started with Oracle Database

Oracle provides a convenient *virtual machine* (VM) containing a complete and ready-to-use installation of Oracle Database 12*c* on Linux. All you need to do is to download and install the Oracle VirtualBox virtualization software and then import a ready-to-use VM. The instructions for doing so are at www.oracle.com/technetwork/community/developer-vm. Pick the Database App Development VM option, and follow the download and installation instructions. The instructions are short and couldn't be simpler (I hope you don't want me to regurgitate them here), because you don't need to install and configure Oracle Database; rather, you import a prebuilt VM into Oracle VirtualBox. The only difficulty you may experience is that the prebuilt VM is almost 5 GB in size, so you need a reliable and fast Internet connection.

If you follow the instructions and fire up the VM, you see the screen in Figure 2:

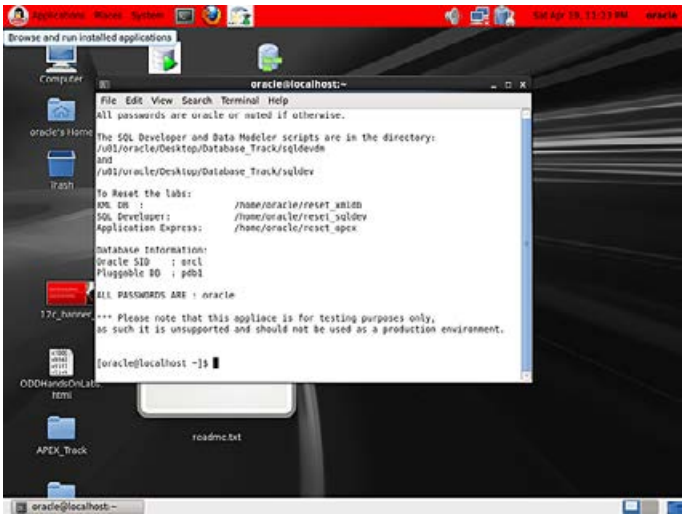


Figure 2.

Minimize the terminal window that's hogging the screen, and click the SQL Developer icon in the top row of icons. SQL Developer is a GUI tool provided by Oracle for database administration. Figure 3 shows what you will see when SQL Developer starts up.

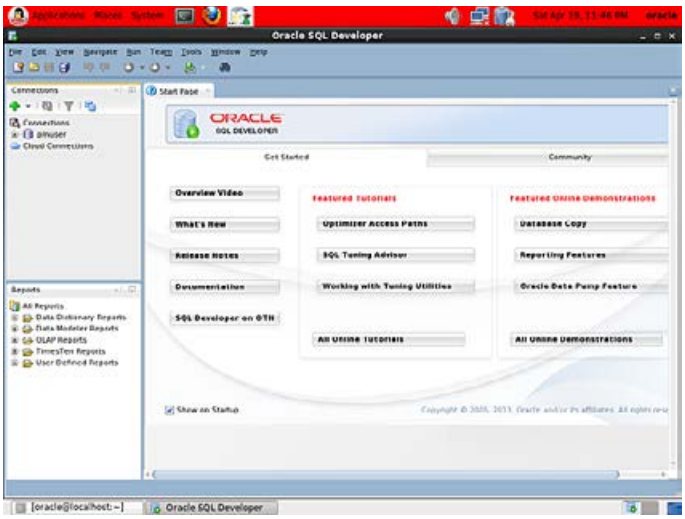


Figure 3.

Click Connections, and create a new connection. Give the connection the name "hr" (Human Resources) or any other name you like. Use the following settings:

- Username "hr"
- Password "oracle"

- Connection Type “Basic”
- Role “Default”
- Hostname “localhost”
- Service Name “pdb1” if using the 12.1.0.1 version of the virtual machine; “orcl” if using the 12.1.0.2 version

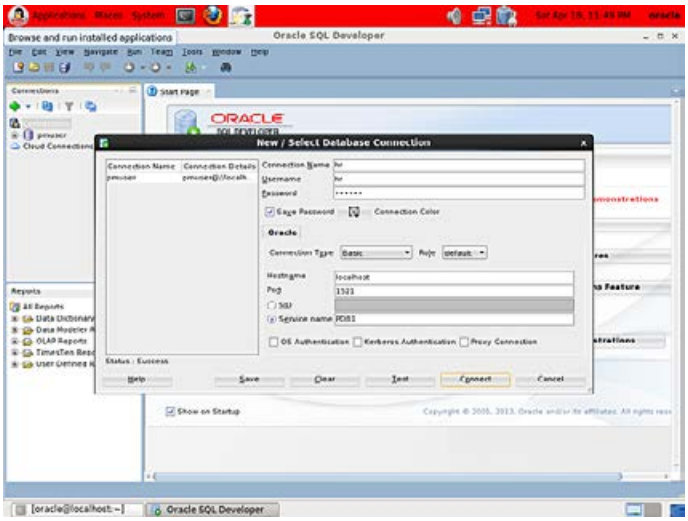


Figure 4.

Then click Connect. Figure 5 shows what you’ll see.

Expand the Tables item in the navigation pane on the left. Six tables are shown; click the EMPLOYEES table. The data in the EMPLOYEES table is listed in a full-screen editor, as shown in Figure 6. If you like, you can make changes to the data and then either save your changes (commit) or discard them (roll back) using the Commit Changes and Rollback Changes buttons or the F11 and F12 keys. ▲

Adapted from Beginning Oracle Database 12*c* Administration: From Novice to Professional, *Apress, 2015 by Iggy Fernandez.*

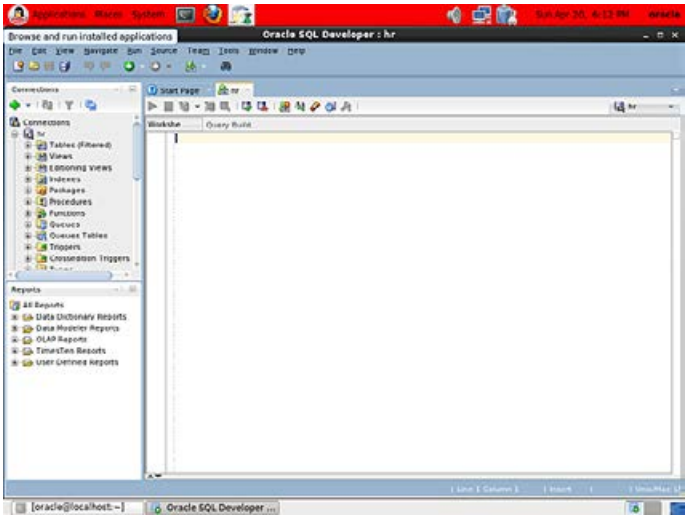


Figure 5.

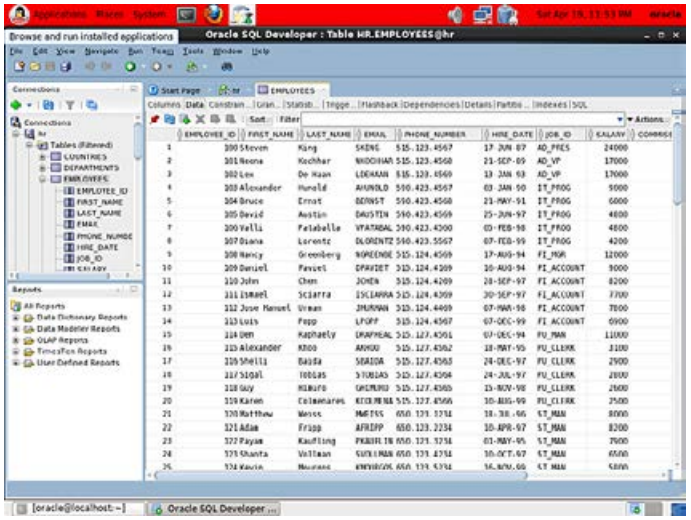


Figure 6.

NoCOUG Quarterly Picnic at Mission Peak Regional Preserve.

OCP Upgrade to Oracle Database 12*c* Exam Guide (Exam 1Z0-060)

Book Notes by Brian Hitchcock

Details

Author: Sam R. Alapati
ISBN-13: 978-0071819978
Pages: 528
Date of Publication: June 3, 2014
Edition: 2
List Price: $70.00
Publisher: Oracle Press

Summary

This book is a good review of all the new features that you may be asked about on the OCP exam. Since I have not yet taken the exam, I can't comment from experience as to how much this book helped me.

I'm not sure how much this would help if you weren't already familiar with the new features of 12*c* in general. I'm not saying that you couldn't learn what you need for the exam from this book alone, but I do think it is less likely.

This book assumes that you know a lot of the material, and it reviews only the most important aspects to prepare you for the exam. While there is a lot of detail about many of the new features, there isn't a comparable discussion of why these features are useful for your applications or your business.

I was surprised at how many errors I found. I know from personal experience that you can't write anything without some errors, but even a casual reading of the text would have revealed many errors, which tells me no one involved with this book bothered to do so.

Introduction

The introduction for this book is more relevant than most. It starts by discussing how the composition of this exam is different from other OCP database exams. In the past, the OCP DBA Upgrade exam used to have some questions about older versions of the database to test your overall knowledge of the DBA job—but only a few. You could easily pass the overall exam and not get any of these general DBA questions correct. Now, one-half of the exam covers general Oracle DBA topics, and this exam prep guide does not address this half; it only covers the new features of 12*c*. You are on your own to find material for the other half of the exam. In the Appendix there is a link to a practice exam that does include questions about both 12*c* new features and general DBA topics. I think the author should have made this clearer: you can't just ignore the general DBA questions and pass the exam; you have to pass both parts: the 12*c* new features part and the general DBA part.

The author does point this out, but I would have discussed this issue in more detail. I don't think most people preparing for the exam really understand that this entire book only prepares them for one-half of the OCP exam.

There are sections that discuss exam structure, number of questions, passing percentages, and so on. I can add, again from personal experience, that you must read each exam question very carefully. While I am sure the exam preparers would disagree with my viewpoint, I believe the questions are designed to trick you and not simply to test your knowledge!

Very specific advice is given, including that we should look out for questions that ask for multiple correct choices. You can't assume there is only one correct answer. This is what makes the certification exams difficult. You probably know most of the correct answers, but knowing all of them can be tricky. Knowing which answers are correct can also be problematic because the questions may make some subtle assumptions. If you don't completely get the context of the question, you may think one of the offered answers is wrong—when in fact, in the very specific context of the question, it's correct.

Another section covers how to prepare for the exam with specific advice about questions on syntax and new features. We are also advised to practice all the relevant commands shown in the Oracle manuals. I can't imagine having the time for this, and I've never done so for all of the previous OCP exams that I've taken and passed.

There is a table listing the OCP Official Objectives, the Certification Objectives, and the relevant chapters and page numbers in the book. Note that since I'm reading the book online in Safari, the page numbers don't help much. This also shows that simply dumping the text of a book onto the internet leaves something to be desired. To really make it work, someone needs to review the content after it has been moved online to make sure all of it still makes sense.

Since half of the exam will be on core DBA skills, a listing of those skills is shown, broken up into core administration, performance management, storage, and security. Again, note that none of these core DBA skills are discussed in this book.

The author suggests, and I recommend as well, that you use the two-minute drill and self-test sections at the end of each chapter to help you prepare for the exam. This will give you a feel for what the exam will be like. For previous certification

exams, I have copied these sections and printed them out for review and to create a practice exam. There are other practice exams available, but you might as well use the one that comes with this book along with any other resources you have access to.

Chapter 1: Enterprise Manager and Database Monitoring

Each chapter has sections for each of the main certification objectives.

This chapter starts with a section titled Use Oracle Enterprise Manager Database Express, OUI, and DBCA. There is coverage of Oracle Enterprise Manager Database Express, which replaces EM Database Control, monitors only one database, and is very lightweight. It uses less memory and CPU, but it also does a lot less compared to Database Control. For example, you can't start or stop the database using EM Database Express. Oracle, of course, recommends that you use Oracle Cloud Control to manage all of the components of your enterprise. Configuring EM Database Express can be done manually or when you create a database. The EM Database Express home page is shown, along with the steps to connect to this utility. Granting access to other users is explained as well as how to set up multiple ports so that EM Database Express can be used to monitor multiple databases on the same server. The main pages within this new version of EM are discussed, followed by the new features of the Database Configuration Assistant and SQL*Developer.

Real-Time Database Operation Monitoring was new in 11*g*, and the new features for 12*c* are reviewed. This starts by defining a database operation and classifying it as simple or composite. Next is a discussion of how real-time monitoring is useful, how each operation is identified, and how to enable the monitoring. Examples of the SQL needed are shown. A specific example of a simple monitoring operation is covered as well as monitoring this operation from EM Database Express and from specific database tables. You can also generate reports on database operations.

Both Emergency Monitoring and Real-Time ADDM are new in 12*c*. Emergency Monitoring allows you to see what is happening inside a database that is hanging or has become nonresponsive. This allows you to make a simple analysis of the situation before deciding to restart the database. You connect in a diagnostic mode, which works even when a normal connection does not by accessing the SGA directly. How to set this up is explained. Real-Time ADDM, on the other hand, is used to look for the root cause of a database that hangs and automatically examines the last ten minutes of the recent SGA activity to look for performance problems that are happening in real time.

11*g* has the AWR Compare Periods Report, which allows comparison of two sets of snapshots. 12*c* has the Compare Period ADDM report, which looks at the cause of the differences between two sets of snapshots. The process used by this feature and how to generate the report are shown. To make the report more useful, you need to assess how much the workload was the same between sets of snapshots. When you generate the report, the degree of SQL commonality will be reported. Oracle tells us that the report is accurate for levels of SQL commonality above 80%.

12*c* provides EM page improvements to better display the ASH data that was available in 11*g*. The ADR (Automatic Diagnostic Repository) provides a consistent format and location for all Oracle products, and for 12*c* we have the new log subdirectory that has two further subdirectories, one for DDL and one for debug. When configured, any DDL changes will be logged in a dedicated file, instead of in the alert log as was done in 11*g*. The debug log records events that don't generate an incident package and don't appear in the alert log but should be saved for later analysis. The new features related to the network include SQL*Net compression and the ability to adjust the session data unit (SDU), both of which can help when your network bandwidth is limited.

Chapter 2: Multitenant Databases—Architecture and Configuration

It isn't surprising that this section tells us that the multitenant database will reduce costs, speed up provisioning and upgrades, simplify management, and much more. Of more significance is the section that explains that 12*c* can be configured as a multitenant database, a single-tenant database, or a non-multitenant database. The last option simply means your 12*c* database has the same setup as any 11*g* database.

Here we cover the basics of the multitenant database. It has a root container database (CDB), which has a seed database inside of it by default. As you create your own application databases, they become the tenants in the multitenant root container database and are called pluggable databases (PDBs). Note that the root, seed, and PDBs are all called containers, and the root database is a container database, containing the other containers. Yes, I find this confusing at times. The use of a seed database as a template for creating other PDBs is explained. Each PDB is pretty much the same as an old-style non-multitenant database, i.e., like a database in previous versions of Oracle. Details of which tablespaces are in each of the databases is covered as well as which database structures are shared (redo log files, for example) and which are not (tablespaces for application data, for example) among all the PDBs. Local and common users and roles are introduced as well as how all the Oracle-supplied shared objects are linked to all the PDBs from the CDB.

Here we learn that creating a multitenant database requires a specific Oracle license if you want to create multiple PDBs in the CDB. The steps to create CDBs and PDBs are shown, including SQL statements and output and some screenshots from the Database Creation Assistant (DBCA). You can also use the Oracle Universal Installer to create CDBs and PDBs while Oracle Cloud Control and SQL Developer can create PDBs only. When you use the create database command, there is a new clause, "enable_pluggable_databases," that you must use to create a multitenant database. The command will, by default, create a non-container database unless this new syntax is used. You also have to add the new enable_pluggable_databases=true parameter to the initialization file. Once the new CDB has been created, you have to run several SQL scripts, some of which are old favorites and some which are new. Several other new options to the create database command are shown. Examples are shown for cloning a PDB, converting a non-PDB into a PDB, and several other PDB operations. This is one of the longest sections of the book and contains a lot of information that could be part of the exam.

While the steps to convert a non-PDB into a PDB and plug it into a CDB were already reviewed, that process assumed the non-PDB was a 12*c* database. If you don't upgrade an existing

11*g* database to 12*c*, you can still bring it into a CDB using the expdb and impdb utilities. This section reviews the steps needed to do this. You can also use the DBMS_PDB package or Golden Gate Replication.

Chapter 3: Managing CDBs and PDBs

This chapter covers how to work with the CDBs and PDBs once they have been created. You can connect to the root container, the CDB, or one of the PDBs using SQL*Plus connect, or you can connect to one container and switch to another using the alter session set container syntax. When you create a PDB, a default service is created. Connecting without a service will default to the root container. A new view, CDB_SERVICES, shows all the services that have been created in the CDB. The concept of the current container is explained, including how it plays with users that exist across the CDB and all the PDBs (common users) and users that only exist in a PDB (local users). DDL can be executed in all containers, under specific circumstances. Examples are given for connecting as well as the needed entries in the files listener.ora and tnsnames.ora. The various ways to switch between containers are shown.

The CDB and all of the PDBs share a single instance, which means you can't shut a PDB down; you can only start up or shut down the entire CDB. The various stages of startup for the CDB, and the PDBs status along the way, are shown in great detail. The various PDB modes are covered, from mounted to open with various read and write options. Instead of startup or shutdown, you open and close a PDB. The steps are explained for shutting down a CDB, and what happens to the PDBs is covered.

Since the CDB and all of the PDBs share a single instance, this also means they all share a single init.ora file or SPFILE. Most initialization parameters affect the CDB and all of the PDBs, but there are about 200 that can be changed for a PDB. These are seen in the ISPDB_MODIFIABLE column of the V$PARAMETER view. An example is given with the SQL to change the DDL_LOCK_TIMEOUT parameter for a specific PDB. For these parameters, when changed for a specific PDB, the change is written to the database tables so we don't need the SCOPE=BOTH syntax. Many examples are given for using the alter pluggable database command to open, close, and work with datafiles while connected to a PDB. You can also use the familiar shutdown and startup commands while connected to a PDB. In the context of a PDB, shutdown takes the pluggable database to the mount state, while startup opens the PDB. The way we can see information about the CDB and PDBs has changed and we now have CDB_xxx views along with the familiar DBA_xxx views. Because we have multiple containers, these views have a new CON_ID column that corresponds to each container database. In this section I learned something that I had not seen in all the other material I had studied so far. I had always been confused about why the root container would have con_id=0 in some places and con_id=1 in others. Here it was explained that con_id=0 is for data pertaining to the entire CDB, including all the PDBs, while con_id=1 is for data pertaining only to the root container. The seed database has con_id=2, and all of the user-created PDBs will have con_id values of 3-254, since you can create up to 252 PDBs. This is an example of why it is good to read multiple sources.

The CDB and each PDB have their own set of tablespaces. They all have the SYSTEM and SYSAUX tablespaces, for example. Some tablespaces, such as UNDO, are only in the CDB and are shared to all the PDBs. You can select from PDB_TABLESPACES to see the tablespaces in a specific PDB, or from CDB_TABLESPACES to see all tablespaces in the CDB and all PDBS, with the con_id column identifying which tablespaces belong to which container database. How to create tablespaces in the CDB and PDBs is discussed as well as how to manage temporary tablespaces. The CDB has a temporary tablespace that is also the default temporary tablespace for all the PDBs, but you can also create a temporary tablespace in each PDB as well.

In 12*c*, we have common database users that exist in the CDB and in all PDBs as well as the typical local database users that are unique to each PDB. It can be confusing to discuss users, roles, and privileges because it all depends on what the context is. If we are talking about a common user in the CDB, it is different from a local user in a PDB. This section covers all of this. I still find it confusing, but this material did help. It is also confusing that all common users must have usernames that start with C## (or c##), but there are also Oracle-supplied common users (SYS and SYSTEM for example) that don't follow this rule. It is only common users that you create that must follow this rule. Also note that you can have common users—and common privileges—but this doesn't mean that privilege has been assigned to the common user in all the PDBs.

Chapter 4: Security New Features

With the increased need to audit Oracle databases, 12*c* provides enhancements to what we had in 11*g*. The new unified audit trail makes auditing simpler by combining various auditing sources into one audit trail table. The unified audit trail gathers audit information from many sources, including Recovery Manager, Database Vault, Label Security, Data Pump, and SQL*Loader. We are told that the new unified audit features consolidate audit information while providing better, simpler security with less performance overhead. These new features can be applied to one or more PDBs or to the entire CDB.

This new feature is activated by default and the architecture is covered. Note that the audit records are now written to the SGA and periodically flushed to the UNIFIED_AUDIT_TRAIL table.

Oracle supplies default unified audit policies, and you can create your own. The process to do so is illustrated with examples of local and common audit policies. Making changes to the policies as well as dropping them is shown. You can configure auditing to capture application context data to go along with the audit data. Examples of viewing audit policies are also given.

To support the least privilege principle, 12*c* has several new administrative system privileges. SYSBACKUP for RMAN tasks, SYSDG for DataGuard, and SYSKM for administration of transparent data encryption. The specifics are documented of what each of these privileges allows a user to do. There are new operating system groups as well—osbackup, osdg, and oskm—so you can set up OS-level groups that will have the respective new privilege. These new privileges require changes to the password file; how to deal with this is covered in detail, with SQL examples.

The new privilege analysis features help to keep track of database user privileges. The DBMS_PRIVILEGE_CAPTURE package can be used to document which users have which privileges. We are shown how to set up and use privilege analysis policies as

well as generate the report of all that is found. SQL is shown for all these steps.

In the past, data redaction was done at the application level; now it can be done in the database. This means that the data redaction policies are executed just before the data is returned to the application by the database. You don't have to worry if all applications redact data in the same way. Some database activities can't have redaction applied, such as RMAN, export and import, and replication. Patching the database and upgrading also have this restriction.

Chapter 5: Performance Tuning Enhancements

Adaptive Execution Plans converts the optimizer into an adaptive optimizer; this is also called "adaptive query optimization." This means that the optimizer can make adjustments at runtime to improve the execution time, via adaptive execution plans and adaptive statistics. For the former, the optimizer starts with an initial plan, the default plan, and then adjusts the plan using actual execution statistics. For the latter, the optimizer can, for complex query predicates, supplement the existing table statistics with adaptive statistics, which can be dynamic statistics, automatic reoptimization, and SQL plan directives. Examples are shown of how to set this up.

In 12*c*, the optimizer can collect statistics online during bulk data loads. Also new is the ability to perform concurrent statistics collection, which means executing multiple statistics-gathering jobs at the same time. Parallel processing can be used to further enhance this process. Two new types of histograms, top frequency and hybrid, assist the optimizer when dealing with skewed data. For top frequency histograms, if the column has more distinct values than the number of buckets you choose, a second histogram is created that looks at the most common column values. For hybrid histograms, the features of a height-based and frequency histogram are combined to provide more accurate estimates. Extended statistics help when the where clause has multiple columns from the same table, by allowing statistics to be computed for groups of columns.

Adaptive SQL Plan Management in 12*c* is an extension of what was offered in 11*g*. SQL execution plans are captured over time to make sure that any changes made do not result in degraded performance. This allows the optimizer to reproduce an execution plan as needed. The execution plan history has all of the different plans that the optimizer has used over time for a given SQL statement. The optimizer uses the SQL Management Base (SMB) in the SYSAUX tablespace to store the plan history, the SQL itself, and the SQL plan baselines. Specific features of the DBMS_SPM package are documented.

Chapter 6: Administration and Management New Features

While resource management is not new in 12*c*, there are new features to handle the PDBs inside the CDB. This means that we need to manage at both the CDB and the PDB levels. At the CDB level, where all the PDBs are competing for resources, you assign shares to each PDB. Each PDB starts with a default value of one share. If you have four PDBs, by default each has one share, so each PDB gets 25% of the overall resources. Each PDB can have a resource plan, but only one, and each resource plan can only apply to one PDB. For the CDB, the resource plan controls the

allocation of CPU usage and parallel execution servers that each PDB can use, and for both, you specify a percentage limit. The steps to create, enable, and manage a CDP resource plan are shown with examples. Since each PDB is very similar to a traditional non-multitenant database (Oracle calls this a "legacy database"), each PDB has consumer groups, limited to eight such groups. Identical procedures are used to create these consumer groups as used in a legacy database.

Multiprocess Multithreaded Architecture is new in 12*c*. This allows some of the background processes to be configured to run as OS threads. Before, these processes ran as OS processes. This improves the performance of parallel processing and reduces the amount of CPU and memory used. We are told that in some cases, this can reduce by half the amount of memory used. Specifically, the PMON, DBWx, VKTM, and PSP background processes can run as OS threads. To configure this feature you need to set THREADED_EXECUTION=TRUE; this parameter is FALSE by default. This new architecture is described as well as the new columns of the V$PROCESS table that show the OS process IDs and the OS thread IDs. Smart Flash Cache extends the functionality provided in 11*g*. The restriction of only having a single device has been removed, and you can have up to 16 devices.

Table enhancements include invisible columns and multiple indexes on the same set of columns. Invisible columns allow application developers to enhance an application while users continue to work. There are limitations: you can't create invisible columns on external, cluster, and temporary tables. Creating and displaying invisible columns is shown with examples. There are many requirements for creating multiple indexes on the same set of columns. For example, only one of these indexes can be visible at a time. This also means that the optimizer can't choose among these multiple indexes; it can only use the one that is visible.

12*c* provides online redefinition of tables with VPD, dropping indexes and constraints, making an index unusable, and setting a column to UNUSED. You can also configure the timeout period for online redefinition. Online operations will lock the table briefly at the end of the redefinition operation, but they have to wait for any pending DML statements. The timeout you can set defines how long the redefinition operation will wait for DML to complete. You can also execute several DDL operations online: drop index, drop constraint, alter index unusable, and set column unused. All of these use the new keyword ONLINE in the DDL statement.

SQL Enhancements include increasing the maximum size of character columns to 32k. Note that this needs to be done with care to avoid row chaining issues. The Database Migration Assistant is a new utility to help migrate databases to Unicode; it reduces the time that previously was needed to run the CCSSCAN and CSALTER utilities. In 12*c*, all LOB columns default to use SecureFiles. Additional options have been added to limit the rows returned by a query using keywords FETCH and OFFSET, and you can specify the percentage of the rows returned.

Chapter 7: RMAN Backup and Recovery New Features

RMAN can back up and recover the whole CDB and all the PDBs it contains, the CDB alone, or one or more of the PDBs that belong to the CDB. You use the keyword DATABASE to tell RMAN you want to back up or recover the entire CDB and all the PDBs it contains. You use the keyword PLUGGABLE DATABASE (isn't this two words?) to tell RMAN to back up or recover one or more PDBs. Note that if you want to back up just the CDB alone, you use the syntax backup pluggable database "CDB$ROOT," which I think is confusing. When is the CDB a pluggable database? Apparently when dealing with RMAN. Many examples with the needed SQL and the resulting output are shown.

You can use Flashback for a CDB much as you would for a non-CDB, but there are restrictions. If you use Flashback on the CDB, all the PDBs in that CDB will go back in time as well. A detailed example of using Flashback to take a CDB back to a specific SCN is presented.

RMAN in 12*c* has the new SYSBACKUP administrative privilege, and you can execute SQL commands directly from within RMAN without having to use quotes. RMAN in 12*c* also supports multisection backups of image copies, which allows you to break up large image copies into smaller pieces. Database duplication in RMAN has been improved so that you can compress and encrypt the data during the duplication process. Many examples of using SQL within RMAN are covered. Extensive coverage is provided for using RMAN to transport tablespaces and databases, and to do this between different platforms.

Flashback Data Archive (FDA) has been enhanced so that you can get a history of all the changes made to a set of tables with one command. This also helps when using the Flashback Query functionality. User context tracking helps identify the users that made changes to a table, and tables, such as those in Exadata, compressed with Hybrid Columnar Compression (HCC), are now supported. If you have been using triggers or other methods to track changes, this data can be imported into FDA. You can also optimize the FDA history tables to reduce storage using various forms of compression, de-duplication, and compression tiering.

Chapter 8: Storage and Data Loading

In 12*c*, Data Pump can make a full transportable export of a database, which means that all of the data and objects are exported. This makes it easier to move an entire database to another platform and can be useful when upgrading a database. Exporting views as tables is now supported, and you can compress data during import. Migrating LOBs to SecureFile LOBs during import is also supported in 12*c*. Use cases, restrictions, and examples are all presented here. Logging can be disabled during import to reduce the time needed.

SQL*Loader now has an Express Mode where you can load data without a control file. You only need to specify the username and the table name; all other needed parameters use default values. There are, of course, restrictions. The table being loaded must have columns that are only character, number, or date time data types, and the input data file can only have delimited character data. Oracle 12*c* introduces an identity column and SQL*Loader supports this as well.

You can move table partitions and subpartitions online in 12*c*, and compression can be applied in the same operation. As these operations are done online, this means they won't interfere with any DML happening on the same partition. A new partitioning option, reference partitioning with interval-partitioned parent tables, is now available. When needed, you can merge and truncate multiple partitions at the same time. Multiple examples of the SQL used for these new options are provided.

Improvements to indexing for partitioned tables include the

ability to create partial indexes on only some of a table's partitions; dropping or truncating causes index maintenance to be done later, asynchronously, to improve performance. When creating indexes on partitioned tables, the keywords INDEXING FULL and INDEXING PARTIAL are provided. Those partitions not included in the index will have a value of OFF for the indexing property. Partial indexes reduce the storage space needed and can improve the performance of operations that load and query data.

Chapter 9: ILM: Automatic Data Optimization, In-Database Archiving, and Temporal Validity

Information Lifecycle Management (ILM) means compressing data and moving it to lower-cost storage as it ages and is accessed less often. This is done to reduce storage costs and improve performance for the data that is accessed the most. To get this done, we use partitioning, compression, and a tiered storage strategy that has different storage platforms that trade off speed and cost. Another factor driving ILM is the need to keep more and more data for a longer time due to increased regulatory requirements. As we need to keep older data longer, we have more need to compress and store the affected data on low-cost storage. The vast amounts of unstructured data used by current applications make this even more urgent.

In 12*c* we have support for ILM in the new Automatic Data Optimization (ADO) features. Using ADO you create policies that determine which data will be compressed and moved to different storage tiers. These policies use the new Heat Map to determine which data is hot (frequently used) and cold (not used as often). Data that is not popular will be voted off the island! These policies can be specified for the row, segment, or tablespace levels. The Heat Map relies on activity tracking, which tracks data changes at the row level. A great deal of detail about ADO is covered.

Another way to deal with old data or data that is not being actively used is to archive it inside the database. In 12*c* we have Hybrid Columnar Compression (HCC) that is provided to support in-database archiving. This compression typically requires Exadata storage. Using HCC you can get 15 to 50 times compression, referred to as "warehouse" and "archive compression" levels respectively.

Temporal Validity is another ILM feature, where each row in a table is marked as active or not. Queries won't return rows that aren't marked active. The start and end dates or timestamps that you set up determine when a row is marked as inactive. This feature also supports specific applications that need to know when specific data became invalid. Dealing with insurance policies would be an example of such an application.

Appendix: About the Download

The text says you can download a test simulator for Exam 1Z0-062, which is not the exam this book is preparing us for. When you actually use the link that is in the text, you go to a test simulator for 1Z0-060, which is the correct exam. I downloaded and installed this exam simulator and it works well. I will be using this as I prepare for the exam.

Conclusion

When I began this book review, I commented that the OCP exam questions need to be read very carefully. I've included two examples of questions from the book's two-minute drill sections. While these are not actual exam questions, they are great examples of the mentality you need to have while studying for the exam.

Example question 1:

You can use Enterprise Manager Database Express to manage:

A. All instances of a RAC database

B. A single-instance database

C. An entire Oracle environment

D. A single instance that belongs to a RAC database

Initially I said A, B, and D were correct. I thought you would use EM Database Express on each node to manage all instances of a RAC database, but the question is asking what could be managed from a single EM Database Express instance, which means only B and D are correct. I'm not asking you to agree with the way the OCP exam questions are set up; I want you to know what you need to look out for in order to do well.

Example question 2:

Which of the following are legitimate configurations in Oracle Database 12*c*?

A One or more PDBs with no CDB

B. A CDB with no PDBs

C. A CDB with one or more PDBs

D. A non-CDB

Initially I said B and C were correct, but I forgot that the seed DB is a PDB and it's created as part of CDB. You can have a CDB with no user-created PDBs, but the seed PDB will still be present—so B is not correct. I also forgot that in 12*c* you can create a non-container, old-style database, so D is correct, and this means that C and D are correct.

Overall, I would recommend this book to anyone who is preparing for this OCP exam, especially for the practice questions. As you can see, you need to be familiar with the way the questions and answers are written in order to do well on the exam. ▲

Brian Hitchcock works for Oracle Corporation where he has been supporting Fusion Middleware since 2013. Before that, he supported Fusion Applications and the Federal OnDemand group. He was with Sun Microsystems for 15 years (before it was acquired by Oracle Corporation) where he supported Oracle databases and Oracle Applications. His contact information and all his book reviews and presentations are available at **www.brianhitchcock.net/oracle-dbafmw/**. *The statements and opinions expressed here are the author's and do not necessarily represent those of Oracle Corporation.*

Slideware in Technical Presentations: Motor, Action!

by Stéphane Faroult

About ten to twelve years ago, a number of presentation specialists started to rebel against the dictatorship of bullet points—bullet points so kindly recommended by presentation software whenever a new presentation is created. Blog articles, books, and even slide decks began to flourish, advocating minimal text and emotional images instead of dull lists that bore everybody. These days, one of the most worn-out clichés of all presentations about presentation software is a full-screen picture of a sleeping audience.

If this trend has enthused a number of would-be-visionary managers, the reaction has been more lukewarm among people who had figures to report in business meetings—and among technical folks. An online review of a famous book on presentations summarizes rather well the feelings of many techies:

I give presentations to audiences of Ph.D.'s in computer science, and four-star Generals. This book is clearly aimed at audiences with shallower minds and lesser technical educations. If I followed this—and almost all other "how to make better presentations/slides"—book I would never be allowed to present to my audiences again.

I won't comment on "shallower minds," but anybody who has had to give a technical presentation would probably agree. It's hard to explain a complex architecture with cute photographs, or to teach professionals or students how to code without showing a good quantity of the stuff—which is, basically, text. Even at technical conferences, where the audience often expects both to learn and to be entertained, some substance is required. Analogies may introduce a welcome element of fun, but you needn't scratch the surface much to find the hard layer; with the exception of a few general talks, adopting the "motivational format" for technical presentations is doomed from the start. As a result, after a quick makeover that may be limited to a better choice of font and color scheme, many perplexed technical folks put down *How to Create Dazzling Slides* and return to what they have always done. Which is, probably, a mistake (except for the fonts and colors).

Between the rock of "emotional content" that has little appeal to rational minds and the hard place of plain information, presented as a list or otherwise, the latter is more comforting. But do you go to a talk to be informed? My personal answer is no. Information is what I find (and try to sort out) when searching the web. What I'm looking for in a talk, seminar, or lecture is an understanding. Whenever you try to make someone understand a technical point in a one-to-one interaction and grab a piece of paper, chances are that if, indeed, a list pops up, you are far more likely to draw shapes one by one and comment on them as you draw, add arrows, underline, grab another piece of paper, and scribble again. You aren't drawing clean slides.

The original sin of slideware is that the slide is the unit, and the unit choice is almost never challenged: designers teach you how to improve your slides and others insist on illustrating a story with slides, but in everybody's mind it's always click, next slide, click, next slide, click, next slide. Animation is frowned upon, as everybody knows that animation is what you add to pep up boring text in the futile hope that it will distract somebody from smartphone games.

In fact, you can give a movie-like quality to a slide presentation—not that roller-coaster feeling of sweeping transitions between fixed slides, but by thinking in sequences instead of individual slides. I have seen countless PL/SQL procedures that developers were trying to make faster by "tuning" individual—and often flawless—SQL statements when the whole logic of database accesses was shabby; in many ways, the obsession with single slides belongs to the same respect of divisions that shouldn't be regarded as untouchable. Switching from one slide to the next need not appear as such. Many slide transitions are totally invisible between two identical slides; change a single element, and only this element will be affected. In many cases, you have the choice between a transition and an animation to achieve one visual effect; sometimes, when you want, for instance, to displace or grow an object on your slide, animation is required. Generally speaking, the number of slides is irrelevant. Clicks don't need to trigger a slide change; clicks need to trigger a visual change that you can comment on. Everything that you would draw on a piece of paper to explain how something works to a single person can be turned into a visual element. These can be added one by one to the slide, with movement or arrows just as you would draw them. Projected on a big screen, you can explain them to hundreds or thousands of people the same way you would explain them to one person. Show what you see in your head.

Try it: you'll be surprised—and your audience will be thankful. ▲

Stéphane Faroult is a database consultant and college instructor who has talked at conferences in Europe and in the U.S.; given seminars in Asia; published video tutorials on YouTube; and taught at colleges in France, Canada, and the U.S. Following his three books on SQL, his latest book, Getting the Message Across *(Apress), demonstrates through examples how to prepare PowerPoint presentations that don't look like PowerPoint presentations.*

Database Specialists: DBA Pro Service

DBA PRO BENEFITS

- *Cost-effective and flexible extension of your IT team*
- *Proactive database maintenance and quick resolution of problems by Oracle experts*
- *Increased database uptime*
- *Improved database performance*
- *Constant database monitoring with Database Rx*
- *Onsite and offsite flexibility*
- *Reliable support from a stable team of DBAs familiar with your databases*

CUSTOMIZABLE SERVICE PLANS FOR ORACLE SYSTEMS

Keeping your Oracle database systems highly available takes knowledge, skill, and experience. It also takes knowing that each environment is different. From large companies that need additional DBA support and specialized expertise to small companies that don't require a full-time onsite DBA, flexibility is the key. That's why Database Specialists offers a flexible service called DBA Pro. With DBA Pro, we work with you to configure a program that best suits your needs and helps you deal with any Oracle issues that arise. You receive cost-effective basic services for development systems and more comprehensive plans for production and mission-critical Oracle systems.

DBA Pro's mix and match service components

Access to experienced senior Oracle expertise when you need it
We work as an extension of your team to set up and manage your Oracle databases to maintain reliability, scalability, and peak performance. When you become a DBA Pro client, you are assigned a primary and secondary Database Specialists DBA. They'll become intimately familiar with your systems. When you need us, just call our toll-free number or send email for assistance from an experienced DBA during regular business hours. If you need a fuller range of coverage with guaranteed response times, you may choose our 24 x 7 option.

24 x 7 availability with guaranteed response time
For managing mission-critical systems, no service is more valuable than being able to call on a team of experts to solve a database problem quickly and efficiently. You may call in an emergency request for help at any time, knowing your call will be answered by a Database Specialists DBA within a guaranteed response time.

Daily review and recommendations for database care
A Database Specialists DBA will perform a daily review of activity and alerts on your Oracle database. This aids in a proactive approach to managing your database systems. After each review, you receive personalized recommendations, comments, and action items via email. This information is stored in the Database Rx Performance Portal for future reference.

Monthly review and report
Looking at trends and focusing on performance, availability, and stability are critical over time. Each month, a Database Specialists DBA will review activity and alerts on your Oracle database and prepare a comprehensive report for you.

Proactive maintenance
When you want Database Specialists to handle ongoing proactive maintenance, we can automatically access your database remotely and address issues directly — if the maintenance procedure is one you have pre-authorized us to perform. You can rest assured knowing your Oracle systems are in good hands.

Onsite and offsite flexibility
You may choose to have Database Specialists consultants work onsite so they can work closely with your own DBA staff, or you may bring us onsite only for specific projects. Or you may choose to save money on travel time and infrastructure setup by having work done remotely. With DBA Pro we provide the most appropriate service program for you.

DatabaseSpecialists

CALL 1-888-648-0500 TO DISCUSS A SERVICE PLAN

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED