

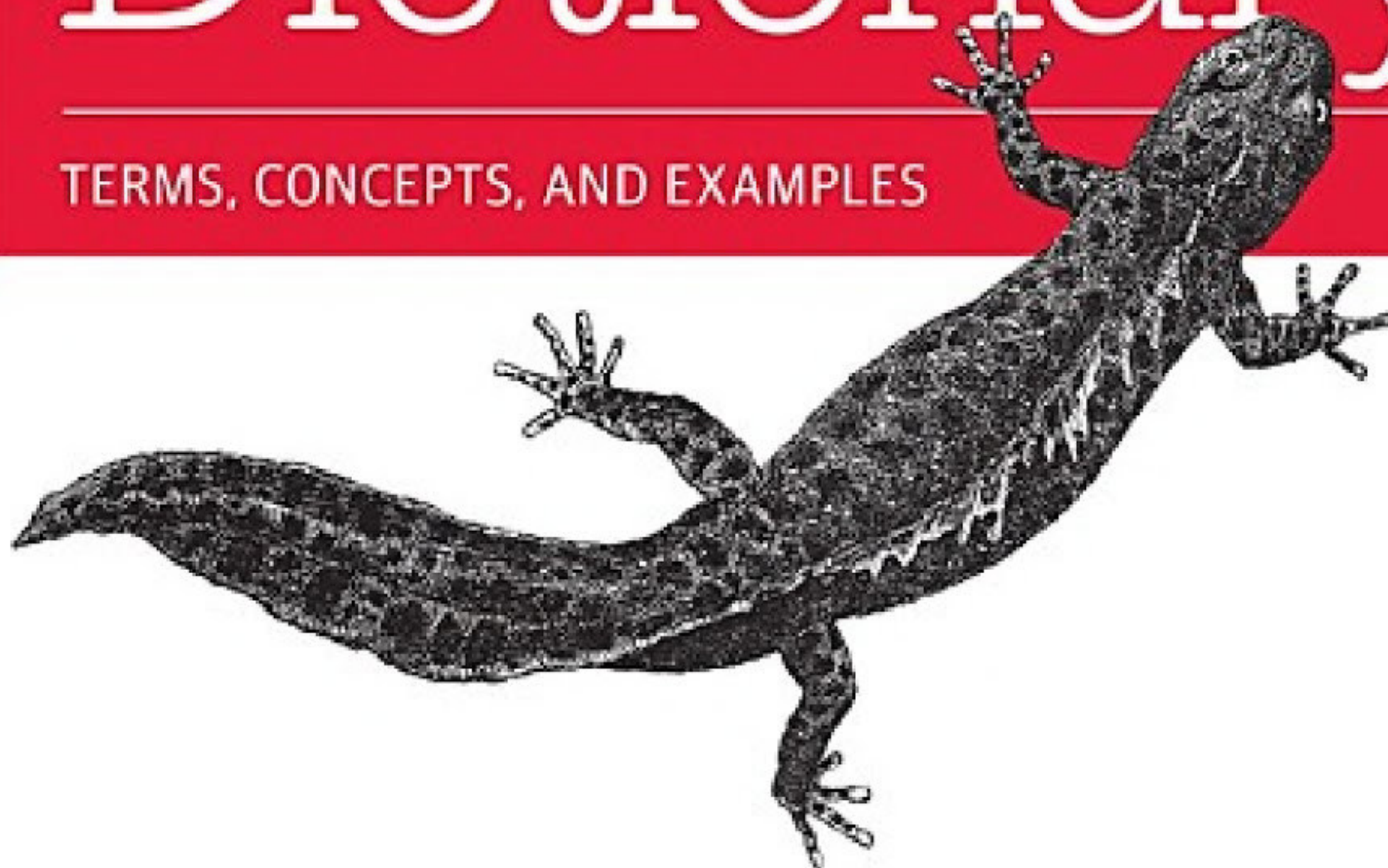
O'REILLY®

New &
Expanded



The New Relational Database Dictionary

TERMS, CONCEPTS, AND EXAMPLES



C. J. Date

The New Relational Database Dictionary

by C. J. Date

Copyright © 2016 C. J. Date. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc.,

1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>).

For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Revision History:

2015-12-15 First release.

See <http://oreilly.com/catalog/errata.csp?isbn=9781491951736> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *The New Relational Database Dictionary* and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-491-95173-6

[LSI]

*Thy gift, thy tables, are within my brain
Full characterized with lasting memory,
Which shall above that idle rank remain
Beyond all date, even to eternity.*

—William Shakespeare: *Sonnet 122*



“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.”

—Lewis Carroll: *Through the Looking-Glass and What Alice Found There*



*Myself when young did eagerly frequent
Doctor and Saint, and heard great Argument
About it and about; but evermore
Came out by the same Door as in I went.*

—Edward Fitzgerald: *The Rubáiyát of Omar Khayyam*



Lexicographer *A writer of dictionaries, a harmless drudge*

—Dr Johnson: *A Dictionary of the English Language*



To all keepers of the true relational flame

About the Author

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is best known for his book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004), which has sold some 900,000 copies at the time of writing and is used in several hundred colleges and universities worldwide. He is also the author of many other books on database management, the following among them:

- From Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, with Hugh Darwen, 2007)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007) and *Database Explorations: Essays on The Third Manifesto and Related Topics* (with Hugh Darwen, 2010)
- From Ventus: *Go Faster! The TransRelationalTM Approach to DBMS Implementation* (2002, 2011)
- From O'Reilly: *Database Design and Relational Theory: Normal Forms and All That Jazz* (2012); *View Updating and Relational Theory: Solving the View Update Problem* (2013); *Relational Theory for Computer Professionals: What Relational Databases Are Really All About* (2013); and *SQL and Relational Theory: How to Write Accurate SQL Code* (3rd edition, 2015)
- From Morgan Kaufmann: *Time and Relational Theory: Temporal Data in the Relational Model and SQL* (with Hugh Darwen and

Nikos A. Lorentzos, 2014)

Mr. Date was inducted into the Computing Industry Hall of Fame in 2004. He enjoys a reputation that is second to none for his ability to explain complex technical subjects in a clear and understandable fashion.

Introduction

This dictionary contains over 1,700 entries dealing with issues, terms, and concepts involved in, or arising from use of, the relational model of data. Most of the entries include not only a definition as such—often several definitions, in fact—but also an illustrative example (sometimes more than one). What’s more, I’ve tried to make those entries as clear, precise, and accurate as I can; they’re based on my own best understanding of the material, an understanding I’ve gradually been honing over some 45 years of involvement in this field.

I’d also like to stress the fact that the dictionary is, as advertised, relational. To that end, I’ve deliberately omitted many topics that are only tangentially connected to relational databases as such (in particular, topics that have to do with database technology in general, as opposed to relational databases specifically); for example, I have little or nothing to say about security, recovery, or concurrency matters. I’ve also omitted certain SQL topics that—despite the fact that SQL is supposed to be a relational language—aren’t really relational at all (cursors, outer join, and SQL’s various “retain duplicates” options are examples here). At the same time, I’ve deliberately included a few nonrelational topics in order to make it clear that, contrary to popular opinion, the topics in question are indeed nonrelational (index is a case in point here).

I must explain too that this is a dictionary with an attitude. It’s my very firm belief that the relational model is the right and proper foundation for database technology and will remain so for as far out as

anyone can see, and many of the definitions in what follows reflect this belief. As I said in my book *SQL and Relational Theory: How to Write Accurate SQL Code* (3rd edition, O’Reilly Media Inc., 2015):

In my opinion, the relational model is rock solid, and “right,” and will endure. A hundred years from now, I fully expect database systems still to be based on Codd’s relational model. Why? Because the foundations of that model—namely, set theory and predicate logic—are themselves rock solid in turn. Elements of predicate logic in particular go back well over 2000 years, at least as far as Aristotle (384–322 BCE).

Partly as a consequence of this state of affairs, I haven’t hesitated to mark some term or concept as *deprecated* if I believe there are good reasons to avoid it, even if the term or concept in question is in widespread use at the time of writing. Materialized view is a case in point here.

The Suppliers-and-Parts Database

Many of the examples used to illustrate the definitions are based on the familiar (not to say hackneyed) suppliers-and-parts database. I apologize for dragging out this old warhorse yet one more time, but as I’ve said many times before, I believe that using the same example—or essentially the same example, at any rate—in a variety of different publications can be a help, not a hindrance, in learning. Here are the relvar definitions for that database (and if you don’t know what a relvar is, then please see the pertinent dictionary entry!):

```
VAR S BASE RELATION
  { SNO      SNO ,
    SNAME    NAME ,
```

```

    STATUS INTEGER ,
    CITY CHAR }
KEY { SNO } ;

```

```

VAR P BASE RELATION
{ PNO PNO ,
  PNAME NAME ,
  COLOR COLOR ,
  WEIGHT WEIGHT ,
  CITY CHAR }
KEY { PNO } ;

```

```

VAR SP BASE RELATION
{ SNO SNO ,
  PNO PNO ,
  QTY QTY }
KEY { SNO , PNO }
FOREIGN KEY { SNO } REFERENCES S
FOREIGN KEY { PNO } REFERENCES P ;

```

These definitions are expressed in a language called **Tutorial D** (see the section “Technical Issues” below for further explanation). The semantics are as follows:

- Relvar S represents *suppliers under contract*. Each supplier has one supplier number (SNO), unique to that supplier; one name (SNAME), not necessarily unique; one status value (STATUS); and one location (CITY). Attributes SNO, SNAME, STATUS, and CITY are of types SNO, NAME, INTEGER, and CHAR, respectively.
- Relvar P represents *kinds of parts*. Each kind of part has one part number (PNO), which is unique; one name (PNAME); one color (COLOR); one weight (WEIGHT); and one location where parts of that kind are stored (CITY). Attributes PNO, PNAME,

COLOR, WEIGHT, and CITY are of types PNO, NAME, COLOR, WEIGHT, and CHAR, respectively.

- Relvar SP represents *shipments* (it shows which parts are shipped, or supplied, by which suppliers). Each shipment has one supplier number (SNO), one part number (PNO), and one quantity (QTY). There's at most one shipment at any given time for a given supplier and given part, and so the combination of supplier number and part number is unique to the shipment in question. Attributes SNO, PNO, and QTY are of types SNO, PNO, and QTY, respectively.

Fig. 1 shows a set of sample values for these relvars. Examples in the body of the dictionary assume those specific values, where applicable.

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

SNO	PNO	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Fig. 1: The suppliers-and-parts database—sample values

Alphabetization

For alphabetization purposes, I've followed these rules:

1. Blanks precede numerals.
2. Numerals precede letters.
3. Uppercase precedes lowercase.
4. Punctuation symbols (parentheses, hyphens, underscores, etc.) are treated as blanks.

Technical Issues

1. Keywords, variable names, and the like are set in all uppercase throughout.
2. Coding examples are expressed, mostly, in a language called **Tutorial D**. Now, I believe those examples are reasonably self-explanatory, but in any case that language is largely defined in the dictionary itself in the entries for the various relational operators (projection, join, and so on). A comprehensive description of the language can be found if needed in the book *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition), by C. J. Date and Hugh Darwen (Addison-Wesley, 2007). To elaborate briefly: As its subtitle indicates, that book—the *Manifesto* book for short—also introduces and explains *The Third Manifesto*, which is a precise though somewhat formal definition of the relational model and a supporting type theory

(including a comprehensive model of type inheritance). In particular, that book uses the name **D** as a generic name for any language that conforms to the principles laid down by *The Third Manifesto*. Any number of distinct languages could qualify as a valid **D**; sadly, however, SQL isn't one of them, which is why coding examples are expressed for the most part in **Tutorial D** and not SQL. (**Tutorial D** is, of course, a valid **D**; in fact, it was expressly designed to be suitable as a vehicle for illustrating and teaching the ideas of *The Third Manifesto*.)

Note: **Tutorial D** has been revised and extended somewhat since the *Manifesto* book was first published. A description of the current version can be found in the book *Database Explorations: Essays on The Third Manifesto and Related Topics*, by C. J. Date and Hugh Darwen (Trafford, 2010)—available online at the *Manifesto* website www.thethirdmanifesto.com.¹ What's more, that *Explorations* book also includes some proposals for extending the language still further (e.g., to incorporate explicit foreign key support), proposals that for the purposes of this dictionary I assume to have been adopted.

3. Following on from the previous point, I should make it clear that definitions in this dictionary are intended to conform fully to the relational model as defined by *The Third Manifesto*. As a consequence, you might find certain aspects of those definitions a trifle surprising—for example, the assertion in the entry for deferred checking that such checking is logically flawed. As I've said, this is a dictionary with an attitude.
4. The notion of *set* is ubiquitous in the database world. On paper,

a set is typically represented by a comma separated list (or “commalist”) of items denoting the elements that constitute the set in question, the whole enclosed in braces, as here: $\{a,b,c\}$. (Blanks appearing immediately before the first item or any comma, or immediately after the last item or any comma, are ignored.) Throughout this dictionary, therefore, I use braces to enclose commalists of items whenever the items in question are meant to denote the elements of some set, implying among other things that (a) the order in which the items appear within that commalist is immaterial and (b) if some item appears more than once, it’s treated as if it appeared just once.

5. **Tutorial D** in particular uses braces to enclose the commalist of argument expressions in certain n -adic (prefix) operator invocations. If the operator in question is idempotent, as in the case of, e.g., JOIN, then the argument expression commalist truly does represent a set of arguments, and the remarks of the previous paragraph apply unconditionally. For other operators, however, the argument expression commalist represents a *bag* of arguments, not a set—in which case the order in which the argument expressions appear is still immaterial, but repetition has significance (despite the fact that **Tutorial D** and this dictionary do still both use braces in such a context). For example, the operator XOR (“exclusive OR”)—meaning the version of that operator defined in this dictionary, at any rate—isn’t idempotent. As a consequence, the **Tutorial D** expressions

XOR { TRUE , FALSE }

and

XOR { TRUE , FALSE , TRUE }

aren't logically equivalent—the first returns TRUE and the second FALSE.

6. The notion of *logic* is, of course, also ubiquitous in the database world. The relational model in particular is firmly based on logic. More precisely, it's based on conventional two-valued logic ("2VL"), and all references to logic in this dictionary should be taken as referring to that logic specifically, except very occasionally where the context demands otherwise. *Note:* As these remarks suggest, many of the dictionary entries do have to do with concepts from logic. Unfortunately, logic texts (and logicians) vary widely not just in the terminology they use but also, in some cases, in the substance of their definitions. The definitions I give are the ones I find most appropriate myself, but be warned that they're sometimes at odds with others you can find in the literature.
7. *A note on the relational operators:* Perhaps unfortunately, it has become standard practice in the database world to use terms such as projection, join, and so on in two somewhat different senses. To be specific, they're used to refer sometimes to those operators as such and sometimes to the results obtained when those operators are invoked. I've followed this practice myself in this dictionary on occasion, and hope it won't lead to confusion.
8. In fact, it has become standard practice to use terms such as projection, join, and so on in another sense also. By definition, these operators apply to relation values specifically. In particular,

of course, they apply to the values that happen to be the current values of relvars. It thus clearly makes sense to talk about, e.g., the join of relvars $R1$ and $R2$, meaning the relation r that results from taking the join of the current values $r1$ and $r2$, respectively, of those two relvars. In some contexts, however (normalization, for example, also view processing), it turns out to be convenient to use expressions like “the join of relvars $R1$ and $R2$ ” in a slightly different sense. To be specific, we might say, loosely but very conveniently, that some *relvar*, R say, is the join of relvars $R1$ and $R2$ —meaning, more precisely, that the value of R is equal at all times to the join of the values of $R1$ and $R2$ at the time in question. In a sense, therefore, we can talk in terms of joins of relvars per se, rather than just in terms of joins of current values of relvars. Analogous remarks apply to all of the relational operations.

9. Regarding projection in particular, please note that **Tutorial D** treats projection as having very high precedence, in order to reduce the number of parentheses that might otherwise be required in relational expressions. For example, the **Tutorial D** expression

$$SP \text{ JOIN } S \{ SNO \}$$

is defined to be equivalent to

$$SP \text{ JOIN } (S \{ SNO \})$$

and not

$$(SP \text{ JOIN } S) \{ SNO \}$$

10. Talk of projection raises yet another point. Here's the definition from the pertinent dictionary entry:

Let relation r have attributes called $A1, A2, \dots, An$ (and possibly others). Then (and only then) the expression $r\{A1, A2, \dots, An\}$ denotes the projection of r on $\{A1, A2, \dots, An\}$, and it returns the relation with heading $\{A1, A2, \dots, An\}$ and body consisting of all tuples t such that there exists a tuple in r that has the same value for attributes $A1, A2, \dots, An$ as t does.

Now, if the result has heading $\{A1, A2, \dots, An\}$, then by definition each of those Ai 's is an <attribute name, type name> pair. But in the projection expression $r\{A1, A2, \dots, An\}$, each of those Ai 's is just an attribute name. (The syntax works because attribute names are unique within the pertinent heading and thus imply the associated type names.) So there's a kind of punning going on here: The very same symbol Ai is being used to denote slightly different things in different contexts.

Generalizing slightly from the foregoing remarks, please understand that the term *attribute* is sometimes used in the body of the dictionary to mean an attribute name rather than an attribute as such; likewise, the term *heading* is sometimes used to mean a set of attribute names rather than a set of attributes as such. I apologize if you find this state of affairs confusing, but once again it's fairly standard practice.

Note: While I'm on the subject of headings, I should mention that in previous versions of this dictionary, headings were denoted $\{H\}$; in the present version, by contrast, they're denoted simply H (i.e., the enclosing braces have been dropped).

11. There's another convention I need to mention (yet again it's fairly standard, but it's worth spelling out in detail in order to avoid any possible confusion). It's illustrated by, e.g., the entry for *joinable*, which includes the following sentence:

Relations r_1, r_2, \dots, r_n ($n \geq 0$) are joinable if and only if for all i and j , relations r_i and r_j are joinable ($1 \leq i \leq n, 1 \leq j \leq n$).

Consider the opening part of this sentence—"Relations r_1, r_2, \dots, r_n ($n \geq 0$) are joinable." Here the case $n = 0$ is to be understood as meaning, not that there exists a relation, not mentioned in the commalist, called r_0 , but rather that the commalist is empty—i.e., there aren't in fact any relations at all.

Similarly, consider the closing part of the sentence—"relations r_i and r_j are joinable ($1 \leq i \leq n, 1 \leq j \leq n$)." Here the case $n = 0$ is to be understood as meaning that there aren't any i 's or j 's, and hence that there are no relations r_i and r_j .

12. I'd also like to draw your attention to still another standard convention, followed throughout this dictionary (and in fact spelled out explicitly in the pertinent dictionary entries): viz., I use the generic term *update* in lowercase to refer to—among other things—the familiar INSERT, DELETE, and UPDATE operators considered collectively. By contrast, when I want to refer to the UPDATE operator as such, I'll set it in uppercase ("all caps") as just shown.

13. Certain of the definitions and examples make use of a simplified notation for tuples. For example, consider the SP tuple shown in Fig. 1 for supplier S1 and part P1. A formal **Tutorial D**

representation of that tuple might look like this:

```
TUPLE { SNO SNO('S1') , PNO PNO('P1') , QTY QTY(300)
}
```

In the simplified notation under discussion, however, the same tuple would be represented thus:

```
<S1, P1, 300>
```

—or, very occasionally, sometimes even thus:

```
S1 P1 300
```

14. This dictionary has almost nothing to say about distributed databases or related matters. The reason is that the whole point about a distributed database as far as the relational model is concerned is that it's supposed to look exactly like a nondistributed database! In other words, all of the problems of distributed databases (and problems there most certainly are) are, at least in an ideal system, problems of physical implementation, not problems of the logical model.
15. Finally, please note that all references to SQL in this dictionary are to the version of that language defined by the official SQL standard. As you might be aware, however, that standard has been through several versions, or editions, over the years. The version current at the time of writing—and the version on which references to SQL in this dictionary are based—is the 2011 version (“SQL:2011”). Here's the formal reference:

International Organization for Standardization (ISO), *Database*

Publishing History and Structure of This Edition

This is the third version, or edition, of this dictionary; the first (with the title *The Relational Database Dictionary*) was published by O'Reilly Media Inc. in 2006, and the second (with the title *The Relational Database Dictionary, Extended Edition*) by Apress in 2008. The following remarks are taken from the introduction to that second edition:

It's a fact of life that dictionaries always expand from one edition to the next. The first edition of this dictionary had just over 600 entries; this one has over 900—an almost 50 percent increase. New entries include atomic relvar, attribute reference, cardinality constraint, class, computational completeness, connection trap, default, field, Great Divide, overriding, referential cycle, safe expression, stored procedure, and many others. I've also taken the opportunity to improve (and in a few cases correct) several of the existing entries; examples here include derived relation, essentiality, fifth normal form, foreign key, JD implied by superkeys, NAND, NOR, ordering, and pointer. No entries have been removed!

One thing I was slightly surprised to discover in working on this edition was the extent to which database concepts rely, ultimately, on certain mathematical terms and constructs. As a result, I decided to include a few somewhat mathematical entries; examples here include boolean algebra, group, inverse, nonnegative, partial ordering, and mathematical (as opposed to relational model) definitions for relation and tuple. The

relevance of such entries might not be immediately apparent, but I felt it was useful to collect them together in one place in order to serve as a convenient reference for anyone who wishes to delve a little more deeply into the precise meaning and origins of a term like relational algebra (or the term relation itself, come to that).

The foregoing remarks, suitably amended, apply to this new edition as well, but with even more force (which is why I decided to use the slightly revised, but I believe merited, title *The New Relational Database Dictionary*). There are now over 1,700 entries in total (an almost 90% increase over the previous edition); new ones include axiom of choice, constant reference, disjoint INSERT, domain of discourse, double negation, exclusive union, individual constant, logical difference, mediator, possibly nondeterministic, primary key attribute, Query-By-Example, repeating field, scalar operator, and tuple product. In addition, numerous existing entries have been expanded and improved (and occasionally corrected), cosmetic improvements have been made throughout, and many more examples have been included.

But the foregoing remarks are far from being the whole of the story. Indeed, the major reason for the increase in size in this edition is that I decided to include, this time around, both (a) definitions arising from the underlying theory of types—including those having to do with the concept of type inheritance in particular—and (b) definitions arising from the use of interval types in particular. Thus, the dictionary is now divided into three parts, as follows:

- *Part I*: Given that relations have attributes and attributes have

types (also called domains), it's clear that relational theory does rely on, or assume, a supporting type theory. But nowhere does it say what that theory has to look like. In other words, relational theory and type theory are, at least to a first approximation, completely independent of one another. At the same time, it's quite difficult—certainly less than fully satisfactory, at least—to define and illustrate relational concepts properly without saying something about the underlying theory of types. Thus, Part I of this new dictionary (“Types and Relations”), which effectively subsumes the previous edition in its entirety, now contains numerous entries having to do with that type theory specifically. (Those entries, like the ones having to do with relational theory as such, are all intended to conform to the prescriptions laid down by *The Third Manifesto*. As you'll soon see, however, the inclusion of such entries inevitably led to the inclusion of several further entries dealing with concepts from the world of object orientation (OO). But those entries too are intended to conform to the prescriptions of *The Third Manifesto*, inasmuch as it makes sense for them to do so.)

- *Part II:* As mentioned earlier in these introductory notes, the *Manifesto* book not only defines a theory of types as such, it builds on that theory to define a model of type inheritance (“the *Manifesto* model”).² Part II of the dictionary (“Inheritance”) deals with terms and concepts arising in connection with that model. The definitions and examples in that part of the dictionary are intended to conform to that model specifically. More details can be found in the *Manifesto* book.

- *Part III*: Finally, Part III of the dictionary (“Intervals”) deals with terms and concepts arising in connection with the theory of intervals. Interval theory provides the formal underpinnings for the support of data of any of a variety of interval types; in particular, it supports the pragmatically important case of temporal data specifically. The definitions and examples in this part of the dictionary are intended to conform to the theory presented in the book *Time and Relational Theory: Temporal Data in the Relational Model and SQL*, by C. J. Date, Hugh Darwen, and Nikos A. Lorentzos (Morgan Kaufmann, 2014), where further details can be found.

Note: All three parts include a few additional remarks of an introductory nature that are specific to the part in question.

Acknowledgments

This dictionary was Jonathan Gennick’s brainchild. Indeed, Jonathan originally intended to write it himself, and I’m very grateful to him for stepping out of the limelight, as it were, and letting me steal his idea and run with it as I’ve done. Jonathan and I have very different writing styles, and what follows is no doubt a long way from what he originally had in mind; but I hope it at least does justice to his overall vision. I’d also like to thank Apress (publisher of the second edition) for allowing me to return to O’Reilly Media Inc. (publisher of the first edition) with this vastly expanded new version, and my friends and colleagues Hugh Darwen and (for Part III in particular) Nikos Lorentzos for numerous helpful comments and much technical assistance over the past several years. It goes without saying that any remaining errors and infelicities

are my own responsibility.

C. J. Date
Healdsburg, California 2015

Part I

Types and Relations

Several of the entries appearing in this part of the dictionary—primarily ones having to do with type theory—are expanded or elaborated on in Part II (“Inheritance”). Such entries are marked “Without inheritance” in what follows (and the corresponding expanded entries in Part II are marked “With inheritance” accordingly).



0-adic (*Of an operator or predicate*) Niladic. *Contrast* 0-ary.

0-ary (*Of a heading, key, tuple, relation, etc.*) Of degree zero. *Contrast* 0-adic.

0-place (*Of a predicate*) Niladic.

0-tuple The empty tuple; the tuple of degree zero.

1NF First normal form.

2NF Second normal form.

2VL Two-valued logic.

3NF Third normal form.

3VL Three-valued logic.

4NF Fourth normal form.

4VL Four-valued logic.

5NF Fifth normal form.

6NF Sixth normal form.



A A relationally complete (q.v.), “reduced instruction set” version of relational algebra with just two primitive operators—REMOVE (essentially projection on all attributes but one), q.v., and an algebraic analog of either NOR or NAND, q.v. The name **A** (note the boldface) is a doubly recursive acronym: It stands for *ALGEBRA*, which in turn stands for *A Logical Genesis Explains Basic Relational Algebra*. As this expanded name suggests, the algebra **A** is designed in such a way as to emphasize its close relationship to, and solid foundation in, the discipline of predicate logic, q.v. Further details can be found in the *Manifesto* book. *Note:* That book uses solid arrowheads to delimit **A** operator names, as in (e.g.) ◀NOR▶, in order to distinguish those operators from operators with the same name in predicate logic or **Tutorial D** or both, but those arrowheads are deliberately omitted here. More to the point, the *Manifesto* book doesn’t actually define either NOR or NAND as a primitive **A** operator; rather, it defines **A** as supporting explicit NOT, OR, and AND operators, q.v. But it then

goes on to show that (a) either OR or AND could be removed without loss, and (b) NOT and whichever of OR and AND is retained could be collapsed into a single operator—NOT and OR into NOR, or NOT and AND into NAND—and thus no serious harm is done by thinking of either NOR or NAND (like REMOVE) as a primitive operator of **A**.

abelian group *See* group (mathematics). *Note: Abelian* (after the mathematician Niels Henrik Abel) is pronounced “ah beel’ ian,” with the stress on the second syllable.

ABS A scalar operator that returns the absolute value of its argument (which must be of some numeric type).

Examples: The expressions ABS(+5) and ABS(−5) both denote ABS invocations, and they both return the absolute value 5.

absolute complement *See* complement (set theory).

absorption Let $Op1$ and $Op2$ be dyadic operators, and assume for definiteness that they’re expressed in infix style. Then $Op1$ absorbs $Op2$ if and only if, for all x and y , $x Op1 (x Op2 y) = x$.

Examples: In logic, each of OR and AND absorbs the other, because $x OR (x AND y)$ and $x AND (x OR y)$ both reduce to—i.e., are logically equivalent to—just x . Analogously, in set theory and relational algebra, each of union and intersection absorbs the other.

abstract algebra *See* algebra.

abstract data type Same as abstract type, in any of the senses of this latter term.

abstract type (*Without inheritance*) Type. *Caveat:* The term is

sometimes used to refer to some specific kind of type (especially one that isn't built in), but a strong case can be made that all types are or should be “abstract,” at least in the sense that their physical representation is hidden from the user.

access path Usually a physical access path, q.v. The term is sometimes used to refer to a “logical” access path also, but this latter term really has no precise definition.

actual operand An argument. *Contrast* formal operand.

ad hoc polymorphism *See* overloading.

additive identity *See* Laws of Algebra.

additive inverse *See* Laws of Algebra.

ADT Abstract data type.

aggregate (*Noun*) An aggregate value, q.v.

aggregate operator A read-only operator that derives a single value, typically but not necessarily a scalar value, from some aggregate value. The aggregate value in question is either a set or a bag of individual values (all of the same type in each case), typically but not necessarily the set or bag of values of some specified attribute of some specified relation, and typically but not necessarily a set or bag of scalar values specifically.

Examples: Let ST1, ST2, ST3, and ST4 be variables of declared type INTEGER. First of all, then, the following statement assigns to ST1 the sum of the status values for suppliers in London:

```
ST1 := SUM ( S WHERE CITY = 'London' , STATUS ) ;
```

The SUM invocation here has two arguments, denoted by a relational expression (q.v.) and an attribute reference (q.v.), respectively. With reference to the definition given above, (a) the first of these arguments is the “specified relation” (in the example, it’s the relation that’s the current value of the expression `S WHERE CITY = 'London'`), and (b) the second is the “specified attribute” (in the example, it’s attribute `STATUS`). Given the sample values shown in Fig. 1, therefore, the aggregate value over which the sum is computed is the bag $\{20,20\}$ of `STATUS` values in the relation that’s the current value of the expression `S WHERE CITY = 'London'`, and the SUM invocation in the example thus returns the value 40.

In contrast to the previous example, the following statement assigns to `ST2` the value 20, not 40, because the aggregate value over which the sum is computed in this case is the singleton set of `STATUS` values $\{20\}$ (since it’s obtained from the projection on $\{\text{STATUS}\}$ of the relation that’s the current value of the expression `S WHERE CITY = 'London'`):

```
ST2 := SUM ( ( S WHERE CITY = 'London' ) { STATUS } ,  
STATUS ) ;
```

Typical aggregate operators include `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`. For `SUM` and `AVG`, the aggregate argument must consist of values of some numeric type; for `MAX` and `MIN`, it must consist of values of some ordered type. *Note:* `COUNT` is slightly special—it simply returns the cardinality of its aggregate argument and thus neither needs nor permits a second argument. Also, **Tutorial D** in particular allows the expression denoting the second argument (and the immediately preceding comma) to be omitted anyway—i.e., even if the aggregate operator is something other than `COUNT`—if the first

argument is a relation of degree one (i.e., a unary relation), in which case the second argument expression is understood by default to be an attribute reference denoting the sole attribute of that unary relation. The foregoing assignment to *ST2* could thus be abbreviated as follows:

```
ST2 := SUM ( ( S WHERE CITY = 'London' ) { STATUS } ) ;
```

By way of another example, consider the following assignment:

```
ST3 := SUM ( S WHERE CITY = 'London' , 2 * STATUS ) ;
```

This statement assigns to *ST3* twice the sum of the status values for suppliers in London. As this example suggests, the expression denoting the second argument isn't necessarily limited to being a simple attribute reference but in fact can be arbitrarily complex. Nor does it necessarily have to contain any attribute references, though in practice it usually will (*see* open expression).

Note: Despite the foregoing, we can in fact assume without loss of generality that the expression denoting the second argument—when there is a second argument—is indeed a simple attribute reference after all, thanks to the availability of the **EXTEND** operator, q.v. For example, the **SUM** invocation in the assignment above to *ST3* is logically equivalent to the following:

```
SUM ( ( EXTEND S WHERE CITY = 'London' : { X := 2 *  
STATUS } ) , X )
```

Simpler (“*n*-adic”) versions of the aggregate operators are also available, in which the aggregate value argument (a set or bag of individual values) is represented by a simple commalist of argument expressions. For example, the following assignment makes use of the *n*-

adic version of SUM (note the use of braces rather than parentheses to enclose the argument expression commalist):

```
ST4 := SUM { X , Y , Z } ;
```

The result in this case is the sum of the current values of variables X, Y, and Z, whatever they might happen to be.

Additional aggregate operators supported by **Tutorial D** include (a) AND, OR, XOR, and EQUIV, q.v. (for aggregates consisting of values of type BOOLEAN) and (b) UNION, XUNION, D_UNION, JOIN, and INTERSECT, q.v. (for aggregates consisting of values of some relation type).

Note: Let *AggOp* be an aggregate operator other than COUNT, and let *agg* be the aggregate value over which some given invocation of *AggOp* is to be evaluated. If *agg* is of cardinality one, the result of the invocation in question is the single value contained in *agg*. If *agg* is of cardinality zero (i.e., if *agg* is empty), and if all three of the following are true—

- a. The invocation in question is essentially just shorthand for repeated invocation of some dyadic operator *Op*
- b. An identity value, q.v., exists for *Op*
- c. The semantics of *AggOp* don't demand that the result of an invocation be a value actually appearing in *agg*

—then

- d. The result of the invocation in question is the applicable identity value.

For example, suppose the operator `SUM` is invoked on an aggregate value consisting of a set or bag of values of type `INTEGER`. Since (a) `SUM` is essentially just shorthand for repeated invocation of the scalar operator “+”, and (b) an identity value—viz., 0—exists for “+” on integers, the result if the aggregate value is empty is the integer 0. By contrast, the `AVG`, `MAX`, and `MIN` of an empty set or bag are undefined, because (a) for `AVG`, no appropriate identity value exists and (b) for `MAX` and `MIN`, the result is supposed to be a value actually appearing in the aggregate argument, and no such value exists (but see further discussion below).

As for `COUNT`, the foregoing remarks can be interpreted to apply to that operator as well by noting that any given `COUNT` invocation is logically equivalent to, and indeed defined to be shorthand for, a certain `SUM` invocation. For example, the `COUNT` invocation

```
COUNT ( S WHERE CITY = 'London' )
```

is logically equivalent to the following `SUM` invocation:

```
SUM ( S WHERE CITY = 'London' , 1 )
```

To return to `MAX` and `MIN` for a moment: Actually there’s an argument that says the `MAX` and `MIN` of an empty aggregate shouldn’t be undefined after all. For definiteness, consider `MAX` specifically. Let `MAX2` be a dyadic operator that returns the larger of its two arguments (in other words, `MAX2{x1,x2}` returns `x1` if `x1 ≥ x2` and `x2` otherwise). Then (a) any given `MAX` invocation is essentially just shorthand for repeated invocation of `MAX2`, and (b) `MAX2` clearly has an identity value, viz., “negative infinity” (meaning the minimum

value of the pertinent type); so we might reasonably define *MAX* to return that identity value if its aggregate argument is empty. Likewise, we might reasonably define *MIN* to return “positive infinity” (the maximum value of the pertinent type) if its aggregate argument is empty. Perhaps the best approach in practice would be to provide both versions of *MAX*—they are, after all, different operators—and let the user decide. We might even provide a third version, one that takes an additional argument x , where x is supplied by the user and is the value to be returned if the aggregate argument is empty.

Incidentally, it’s worth noting that (contrary to popular opinion, perhaps) SQL doesn’t support aggregate operators at all. It does support the notion of a summary, q.v., but aggregate operator invocations and summaries aren’t the same thing—there’s a logical difference (q.v.) between them, as explained under summary.

aggregate type In general, a nonscalar type for which the user visible components are usually required all to be of the same type. For example, array and relation types might be regarded as aggregate types, but tuple types usually wouldn’t be.

aggregate value Either a set or a bag of individual values (all of the same type in each case)—typically but not necessarily the set or bag of values of some specified attribute of some specified relation, and typically but not necessarily a set or bag of scalar values specifically. *See* aggregate operator.

ALGEBRA *See* A.

algebra 1. Generically, a formal system consisting of (a) a set of elements and (b) a set of read-only operators that apply to those

elements, such that those elements and operators together satisfy certain laws and properties (almost certainly closure, probably commutativity and associativity, and so on); also known as an *algebraic structure* or an *abstract algebra*. The word algebra itself derives from Arabic *al-jabr*, meaning a resetting (of something broken) or a combination. *Note:* The foregoing definition is admittedly not very precise, but the term just doesn't seem to have a very precise definition, not even in mathematics. Note in particular that not all algebras abide by The Laws of Algebra, q.v.!—for example, matrix algebra does not. *See also* boolean algebra. 2. Relational algebra specifically, q.v. (if the context demands).

algebra of sets *See* boolean algebra (second definition).

alias Strongly deprecated term sometimes used in SQL contexts to mean either a tuple calculus range variable, q.v., or the name of such a variable. The term *table alias* (also deprecated) is also sometimes used with the same meaning. *See also* correlation name.

ALL Keyword sometimes used as an alternative spelling for the aggregate operator **AND** (*see* aggregate operator).

ALL BUT *See* projection.

all key Relvar R is “all key” if and only if the entire heading of R is a key (in which case it's the only key, necessarily). Equivalently, R is all key if and only if no proper subset of the heading is a key. Note that if R is all key, then it certainly has no nonkey attributes (q.v.), but the converse is false—a relvar can have no nonkey attributes and yet not be all key.

ALPHA A proposal, due to Codd, for a concrete relational language based on tuple calculus; also known as Data Sublanguage ALPHA. ALPHA as such was never implemented, but its ideas were influential on the design of several languages that were, including QBE, QUEL, and (to a much lesser extent) SQL.

alternate key Loosely, a key that isn't a primary key, q.v. More precisely, let relvar R have keys K_1, K_2, \dots, K_n (and no others), and let some K_i ($1 \leq i \leq n$) be chosen as the primary key for, or of, R ; then each K_j ($1 \leq j \leq n, j \neq i$) is an alternate key for, or of, R . The term isn't much used.

AND 1. A connective, q.v. (*see* conjunction). 2. An aggregate operator, q.v. *Note:* AND as conventionally understood is a logical operator (and this observation applies to both of the foregoing definitions); however, the algebra \mathbf{A} , q.v., includes an operator it calls AND that—by definition—is a relational operator (in fact, it's just natural join).

antecedent *See* implication.

antijoin Term sometimes used as a synonym for semidifference, q.v. The term is deprecated, slightly, because the operator is really “anti” semijoin, q.v., not “anti” join as such.

antisymmetry *See* partial ordering. Note that antisymmetry and asymmetry aren't the same thing—the former is as defined under partial ordering, the latter just means lack of symmetry.

ANY Keyword sometimes used as an alternative spelling for the aggregate operator OR (*see* aggregate operator).

appearance (*Of a value*) An occurrence or “instance” of a value in

some context. Observe that there's a logical difference between a value as such (*see value*) and an appearance of that value in some context—for example, as the current value of some variable or as an attribute value within the current value of some tuplevar or relvar. Of course, every appearance of a value has an implementation that consists of some internal or physical representation, q.v., of the value in question (and distinct appearances of the same value might have distinct physical representations). Thus, there's also a logical difference between an appearance of a value, on the one hand, and the physical representation of that appearance, on the other; there might even be a logical difference between the physical representations used for distinct appearances of the same value. All of that being said, however, it's usual to abbreviate *physical representation of an appearance of a value* to just *appearance of a value*, or (more often) to just *value*, so long as there's no risk of ambiguity. Note, however, that *appearance of a value* is a model concept, whereas *physical representation of an appearance* is an implementation concept—users certainly might need to know whether (for example) two variables contain appearances of the same value, but they don't need to know whether those two appearances use the same physical representation.

Example: Let N1 and N2 be variables of declared type INTEGER. After the following assignments, then, N1 and N2 both contain an appearance of the integer value 3. The corresponding physical representations might or might not be the same (for example, N1 might use a base two representation and N2 a base ten representation), but either way it's of no concern to the user.

```
N1 := 3 ;
```

```
N2 := 3 ;
```


application relvar *See* relvar.

argument (*Without inheritance*) The actual operand that replaces—i.e., is substituted for—some parameter of some operator when the operator in question is invoked. That argument must be of the same type as the parameter it replaces. Note that there's a logical difference between an argument per se and the expression that denotes it (i.e., the argument expression, q.v.). To be specific, the argument per se is either a value or a variable; if the pertinent parameter is subject to update, then the argument is—in fact, must be—a variable specifically, denoted by some variable reference, otherwise it's a value and can be denoted by an arbitrarily complex expression (possibly just a variable reference). *Contrast* parameter.

Examples: Let operator DOUBLE be defined as follows:

```
OPERATOR DOUBLE ( X INTEGER ) RETURNS INTEGER ;  
    RETURN ( 2 * X ) ;  
END OPERATOR ;
```

X here is a parameter, of declared type INTEGER. Let N be a variable of declared type INTEGER. Then, e.g., DOUBLE (N+1) is an invocation of DOUBLE, and the value of the expression N+1 at the time of that invocation is an argument—in fact, the sole argument—to that invocation. What's more, that invocation is itself an expression, and it can appear wherever an integer literal can appear (because, thanks to the RETURNS clause, q.v., operator DOUBLE returns a value of type INTEGER when it's invoked).

Now suppose by contrast that DOUBLE is defined to be an update operator instead of a read-only one, as follows (observe that the RETURNS clause has been replaced by an UPDATES clause and the

RETURN statement has been replaced by an assignment):

```
OPERATOR DOUBLE ( X INTEGER ) UPDATES { X } ;  
    X := 2 * X ;  
END OPERATOR ;
```

Now the parameter X is subject to update, and any argument corresponding to X must be a variable specifically. What's more, the only way `DOUBLE` can now be invoked is by means of an explicit `CALL` statement (or equivalent), as here:

```
CALL DOUBLE ( N ) ;
```

In this example, the variable N —not the value of that variable, observe—is the argument to the invocation. Moreover, note carefully that `DOUBLE (N)` here isn't an expression, and it can't appear “wherever an integer literal can appear.” Note too that, e.g.,

```
CALL DOUBLE ( N + 1 ) ;
```

would be a syntax error, because $N+1$ isn't a variable reference.

argument expression An expression denoting an argument (q.v.) to some operator invocation.

arity Degree, q.v. The term isn't much used, except in formal or academic contexts.

Armstrong's axioms / Armstrong's inference rules (*For FDs*) Let X , Y , and Z denote sets of attributes; also, let XZ denote the set theory union of X and Z , and similarly for YZ , etc. Then Armstrong's axioms (also known as Armstrong's inference rules) are as follows:

- a. If $X \supseteq Y$, then $X \rightarrow Y$ (the reflexivity rule).
- b. If $X \rightarrow Y$, then $XZ \rightarrow YZ$ (the augmentation rule).
- c. If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ (the transitivity rule).

These rules are both sound and complete (*see* soundness; completeness).

Examples: The FD $X \rightarrow Y$ is implied by the FD $X \rightarrow YZ$. To be specific, it can be derived from this latter FD using Armstrong's axioms, thus: (a) $X \rightarrow YZ$ (given); (b) $YZ \rightarrow Y$ by reflexivity; hence (c) $X \rightarrow Y$ by transitivity.

By way of a second example, given the FDs $X \rightarrow Y$ and $Z \rightarrow W$, it can be shown using Armstrong's axioms that the FD $XV \rightarrow YW$ (where V is the set theory difference $Z - Y$ between Z and Y , in that order) is implied by those given FDs. (This example, which is due to Darwen, can be regarded as another inference rule. It has the interesting property that the augmentation and transitivity rules, as well as several other rules not discussed here, are all special cases.)

arrow *See* functional dependency.

arrow out of An FD of the form $A \rightarrow B$ is sometimes referred to, informally, as “an arrow out of A ” (or, even more informally, as an arrow out of the attribute(s) constituting A —especially if A is of degree one).

assignment (*Without inheritance*) An operator, denoted “:=” in **Tutorial D**, that assigns a value (the source, denoted by an expression) to a variable (the target, denoted by a variable reference); also, the operation performed when that operator is invoked. The source and

target must be of the same type, and the operation overall is required to abide by (a) *The Assignment Principle*, q.v. (always), as well as (b) **The Golden Rule**, q.v. (if applicable). *Note:* Every update operator invocation is logically equivalent to some assignment—possibly a multiple assignment, q.v.—in the second of the senses just defined. See *also* multiple assignment; relational assignment; tuple assignment.

Assignment Principle After assignment of value v to variable V , the comparison $v = V$ is required to evaluate to TRUE.

associative addressing Addressing by value instead of position. All addressing is associative in the relational model, implying among other things that pointers, q.v., are outlawed (and hence implying further that no database relvar can have an attribute of any pointer type).

associativity Let Op be a dyadic operator, and assume for definiteness that it's expressed in infix style. Then Op is associative if and only if, for all x , y , and z , $x Op (y Op z) = (x Op y) Op z$.

Examples: In ordinary arithmetic, addition (“+”) is associative, because

$$x + (y + z) = (x + y) + z$$

for all numbers x , y , and z . Likewise, “| |” (string concatenation) is associative, because

$$x || (y || z) = (x || y) || z$$

for all strings x , y , and z . In the same kind of way, UNION and JOIN are associative in relational algebra (by contrast, MINUS is not). Likewise, OR and AND are associative in logic (by contrast, IMPLIES is not). *Note:* Each of the associative operators mentioned in these

examples except for “| |” is also commutative, q.v. Another example of an operator that’s associative but not commutative is the (conventionally unnamed) dyadic connective in logic that simply returns the value of its first argument. *See also* left associativity; right associativity.

atomic predicate A simple predicate, q.v.

atomic projection *See* atomic relvar; FD preservation.

atomic proposition A simple proposition, q.v.

atomic relvar A relvar that can’t be nonloss decomposed into independent projections. *Note:* The term *independent projection* is being used here in a specific technical sense (*see* FD preservation). Note too that the term *atomic relvar* is deprecated, somewhat, because it’s likely to be confused with the term *irreducible relvar* (*see* irreducible, second definition). While it’s true that irreducible relvars are certainly atomic, the converse is false—a relvar can be atomic without being irreducible (*see* the example below). The concept is seldom needed, anyway; thus, it’s probably best just to spell out the meaning as and when necessary.

Example: Suppose relvar SP is subject to a constraint to the effect that part P1 (only) is always supplied in a quantity in the range 1-100, part P2 (only) is always supplied in a quantity in the range 101-200, and so on; then the FD {QTY} → {PNO} holds in that relvar. (This particular constraint isn’t satisfied by the sample values in Fig. 1, of course. Indeed, the example overall is highly contrived; however, it suffices for the purpose at hand.) This revised version of SP can be nonloss decomposed into its projections on {SNO,QTY} and {QTY,PNO} (and it can’t be nonloss decomposed in any other way,

other than trivially); in fact, the relvar isn't in BCNF, q.v., because {QTY} isn't a superkey (it is, however, in 3NF, q.v., and in fact in EKNF, q.v., also). Those two projections—i.e., on {SNO,QTY} and {QTY,PNO}—are atomic. They're also in BCNF (the keys are {SNO,QTY} and {QTY}, respectively). However, they aren't independent, because the FD {SNO,PNO} → {QTY}, which holds in SP, isn't preserved in the decomposition. Relvar SP, revised as above, is thus atomic (*see* FD preservation) but not irreducible. Note that it follows from this example that the objectives of (a) decomposing into BCNF projections and (b) decomposing into independent projections, though both generally desirable, can sometimes be in conflict.

atomic statement (*Programming languages*) Syntactically, a statement that contains no other statements nested inside itself (*contrast* compound statement); semantically, a statement that's guaranteed either to execute in its entirety or to have no effect (other than returning a status code or equivalent, perhaps). All syntactically atomic statements are semantically atomic in the relational model, except possibly if the statement in question represents an invocation of a user defined operator, q.v. (The converse is false, incidentally; an important counterexample is provided by multiple assignment, q.v., which is semantically atomic but not syntactically so.) *Note:* A statement might execute in its entirety and yet have no lasting effect, owing to the fact that its execution will necessarily be part of some transaction (q.v) and that transaction might subsequently be rolled back.

atomic type Somewhat deprecated term for a scalar type, q.v.

atomic value Old fashioned and somewhat deprecated term for a scalar value, q.v.

attribute Very loosely, a column; more precisely, an <attribute name, type name> pair, though it's common to ignore the type name in informal contexts. (Ignoring the type name in this way is acceptable when the heading, q.v., of which the attribute in question is a component is known, because the relational model requires attribute names within any given heading to be unique, and the attribute names thus effectively imply the corresponding type names.)

Examples: In the suppliers-and-parts database, (a) the pair <SNAME,NAME> is an attribute of relvar S, and (b) the pair <SNO,SNO> is an attribute—in fact, a “common attribute,” q.v.—of both relvar S and relvar SP. We might also say, more simply but less formally, just that (a) SNAME is an attribute of relvar S and (b) SNO is an attribute—a “common attribute”—of both relvar S and relvar SP. Attributes SNAME and SNO are of declared types NAME and SNO, respectively.

Caveat: The foregoing is the relational meaning of the term *attribute*. Be aware, however, that some systems, including SQL systems in particular (also certain OO systems), use the term with a meaning or meanings rather different from that ascribed to it here.

attribute assignment An assignment in which the target is specified syntactically by means of an attribute reference, q.v. Attribute assignments are permitted in **Tutorial D** only in the context of an invocation of EXTEND, SUMMARIZE, or UPDATE.

Example: Consider the following UPDATE statement:

```
UPDATE S WHERE SNO = SNO('S1') : { STATUS := 10 , CITY :=  
'Rome' } ;
```

This UPDATE statement contains two attribute assignments, viz.,

STATUS := 10 and CITY := 'Rome'.

attribute constraint A specification, conceptually part of a relvar constraint, q.v., to the effect that a given attribute of a given relvar is of a given type.

Example: Attribute SNAME of relvar S is declared to be of type NAME—i.e., it's constrained to contain values of type NAME only. Any operation (necessarily an update operation) that attempts to assign a value to relvar S in which some tuple contains a value for attribute SNAME that's not of type NAME will fail (and moreover will do so, ideally, at compile time).

attribute extractor An operator for extracting the value of a specified attribute from a specified tuple (*attribute value extractor* would be a more accurate term).

Example: Let t denote the supplier tuple shown in Fig. 1 for supplier S1. Then the following **Tutorial D** expression extracts the status value 20 (an integer) from that tuple:

```
STATUS FROM  $t$ 
```

STATUS here is an attribute reference, q.v. *Note:* SQL uses dot qualification, q.v., for such purposes (as well as for other purposes, beyond the scope of this dictionary). Here's the SQL analog of the foregoing **Tutorial D** example (though here, of course, t must be understood as denoting an SQL row, not a tuple):

```
 $t$ .STATUS
```

attribute level redundancy *See* redundancy.

attribute reference Syntactically, an attribute name (possibly dot qualified, though never so in **Tutorial D**). An attribute reference denotes either an attribute as such or the value of the attribute in question (frequently, though not invariably, within some specific tuple in each case), as the context demands. Note in particular that such a reference certainly denotes an attribute as such if it's used to specify the target for some attribute assignment within some **EXTEND**, **SUMMARIZE**, or **UPDATE** invocation.

Examples: Consider the following **UPDATE** statement:

```
UPDATE P WHERE CITY = 'London' :  
    { WEIGHT := 2 * WEIGHT , CITY := 'Oslo' } ;
```

This statement contains two attribute assignments (q.v.) and four attribute references, viz., **CITY** (twice) and **WEIGHT** (also twice). Imagine the overall **UPDATE** being executed by processing the tuples of relvar **P** one by one in some sequence, and let t be the tuple currently being processed. Within the overall statement, then, (a) the first appearance of **CITY** and the second appearance of **WEIGHT** currently denote the **CITY** value and the **WEIGHT** value, respectively, within t ; (b) the first appearance of **WEIGHT** and the second appearance of **CITY** currently denote the **WEIGHT** attribute as such and the **CITY** attribute as such, respectively, within t . See the example under **UPDATE** for further explanation.

attribute reference FROM Tutorial D syntax for an attribute extractor, q.v.

attribute renaming See renaming.

attribute type See attribute. *Note:* Attributes can be of essentially any

type whatsoever, except that (a) no attribute can be of a type that's defined, directly or indirectly, in terms of the type of the tuple or relation of which it's a part (*see* recursively defined type); (b) no database relvar can have an attribute of any pointer type (*see* pointer).

attribute value *See* tuple value.

attribute value extractor *See* attribute extractor.

audit trail A special file or database, possibly but not necessarily integrated with the recovery log (q.v.), in which the system keeps track of database operations performed by users, with a view to assisting in the detection of actual or attempted security breaches, among other things. Further details are beyond the scope of this dictionary (but see the discussion of logged time in Part III).

augmentation *See* Armstrong's axioms.

automatic action An action carried out by the DBMS on the user's behalf without having been explicitly requested by the user in question. Compensatory actions, q.v., are an important special case.

automatic definition (*Without inheritance*) Defining a scalar type T automatically causes certain associated operators to be defined as well. The operators in question are assignment (“:=”), equality (“=”), and at least one selector, q.v., and at least one set of THE_ operators, q.v. *Note:* If operator Op is automatically defined in this way as an operator associated with type T , code to implement Op might or might not be automatically defined as well. In particular, for “:=” and “=” it probably will be, whereas for selectors and THE_ operators it might not. If it isn't, however, then whatever agency (either the system or some user)

is responsible for defining type T must also define that code—in effect, as part of the process of defining T . Note too that operators analogous to the ones that are the subject of this entry are “automatically defined” for tuple and relation types as well, even though such types are generated (*see* type generator) instead of being explicitly defined.

automatic optimization *See* optimization.

axiom Something assumed to be true, available for use in deriving further truths (i.e., theorems, q.v.; *see also* proof). An axiom is a special case of a theorem. In a database, the tuples in the base relations can be regarded as axioms, because they represent propositions that are assumed to be true (*see Closed World Assumption*). *Note:* In a formal system, it’s usually desirable that the axioms all be independent of one another, meaning none of them is derivable from the rest. For precisely analogous reasons, it’s usually desirable in a database that there be no redundancy, q.v. (or at least no uncontrolled redundancy, q.v.).

Example: The tuple $\langle S1, \text{Smith}, 20, \text{London} \rangle$ in the base relation that’s the current value of base relvar S represents the presumably true proposition *Supplier S1 is under contract, is named Smith, has status 20, and is located in city London*. This proposition thus serves as an axiom with respect to (the current value of) the suppliers-and-parts database.

axiom of choice An axiom of set theory to the effect that, given a set S of nonempty, pairwise disjoint sets $s1, s2, \dots, sn$, there exists a set of n elements $x1, x2, \dots, xn$ such that each xi is an element of si ($i = 1, 2, \dots, n$). The axiom implies among other things that, given some set s , it must be possible to choose an arbitrary element x from that set (*see* ZO). *Note:* The axiom of choice is obviously and intuitively valid (and

noncontroversial) so long as the sets s_1, s_2, \dots, s_n , and S are all finite, but can be (and has been) questioned otherwise.

axiom of extension An axiom of set theory, to the effect that two sets are equal, and hence are in fact the same set, if and only if they contain the same elements.



bag Very informally, “a set that permits duplicates”; more precisely, a collection of objects, called elements, in which the same element can appear any number of times. An example is the collection $\{x,y,y,y,z,z\}$, which can alternatively be written as, e.g., $\{y,y,x,z,y,z\}$, since bags, like sets, have no ordering to their elements. The number of times a given element appears in a given bag is the multiplicity (of that element with respect to that bag). *Note:* As the foregoing text indicates, a bag is usually represented on paper by a commalist of items denoting the elements that constitute the bag in question, that whole commalist then being enclosed in braces. **Tutorial D** in particular uses braces to enclose the commalist of argument expressions in certain n -adic operator invocations when the argument expression commalist in question denotes a bag of arguments (as well as when it denotes a set). For example, the **Tutorial D** expression $\text{SUM } \{1,2,2\}$ denotes an invocation of the n -adic version of the aggregate operator SUM (see aggregate operator), and it returns 5, not 3.

The set theory operations of inclusion, union, intersection, difference, exclusive union (also known as symmetric difference), and product—but not complement—can all be generalized to apply to bags, as follows. First, inclusion. Let b_1 and b_2 be bags, and let element

x appear exactly $n1$ times in $b1$ and exactly $n2$ times in $b2$ ($n1 \geq 0, n2 \geq 0$). Then bag $b1$ includes bag $b2$ (" $b1 \supseteq b2$ ") if and only if $n1 \geq n2$ for all such elements x ; further, $b2$ is included in $b1$ (" $b2 \subseteq b1$ ") if and only if $b1$ includes $b2$, and $b1$ is equal to $b2$ (" $b1 = b2$ ") if and only if each of $b1$ and $b2$ includes the other. *Note:* All of the terms associated with set inclusion (subset, proper subset, and so on) have analogs in connection with bag inclusion (subbag, proper subbag, and so on).

Now let Op be union, intersection, difference, or exclusive union, and let b be the bag obtained by applying Op to bags $b1$ and $b2$ (in that order, in the case of difference), where as before element x appears exactly $n1$ times in $b1$ and exactly $n2$ times in $b2$ ($n1 \geq 0, n2 \geq 0$). Then element x appears exactly n times in b , where n is:

- $MAX\{n1, n2\}$ if Op is union
- $MIN\{n1, n2\}$ if Op is intersection
- $MAX\{n1 - n2, 0\}$ if Op is difference
- $ABS(n1 - n2)$ if Op is exclusive union

In no case does b contain any other elements.

Again let elements $x1$ and $x2$ appear exactly $n1$ times in $b1$ and exactly $n2$ times in $b2$, respectively ($n1 \geq 0, n2 \geq 0$), and let b be the product of $b1$ and $b2$, in that order. Then the ordered pair $\langle x1, x2 \rangle$ appears exactly $n1 * n2$ times as an element of b , and b contains no other elements.

Finally, there are two further operations, union plus and intersection star (also known by a variety of other names), that have no counterpart in set theory. Let b be the bag obtained by applying one of

these operations to bags $b1$ and $b2$, where once again element x appears exactly $n1$ times in $b1$ and exactly $n2$ times in $b2$ ($n1 \geq 0, n2 \geq 0$). Then x appears exactly n times as an element of b , where n is:

- $n1+n2$ if Op is union plus
- $n1*n2$ if Op is intersection star

(and b contains no other elements).

Examples: Let $b1$ and $b2$ be the bags $\{w,w,x,x,y\}$ and $\{x,y,y,y,z,z\}$, respectively. Then the following expressions yield the indicated results:

- $b1 \text{ UNION } b2 = \{w, w, x, x, y, y, y, z, z\}$
- $b1 \text{ INTERSECT } b2 = \{x, y\}$
- $b1 \text{ MINUS } b2 = \{w, w, x\}$
- $b2 \text{ MINUS } b1 = \{y, y, z, z\}$
- $b1 \text{ XUNION } b2 = \{w, w, x, y, y, z, z\}$
- $b1 \text{ TIMES } b2 = \{ \langle w, x \rangle, \langle w, x \rangle, \langle x, x \rangle, \langle x, x \rangle, \langle y, x \rangle, \langle w, y \rangle, \langle w, y \rangle, \langle x, y \rangle, \langle x, y \rangle, \langle y, y \rangle, \langle w, y \rangle, \langle w, y \rangle, \langle x, y \rangle, \langle x, y \rangle, \langle y, y \rangle, \langle w, y \rangle, \langle w, y \rangle, \langle x, y \rangle, \langle x, y \rangle, \langle y, y \rangle, \langle w, z \rangle, \langle w, z \rangle, \langle x, z \rangle, \langle x, z \rangle, \langle y, z \rangle, \langle w, z \rangle, \langle w, z \rangle, \langle x, z \rangle, \langle x, z \rangle, \langle y, z \rangle \}$

- $b1 \text{ UNION+ } b2 = \{w, w, x, x, x, y, y, y, y, z, z\}$
- $b1 \text{ INTERSECT* } b2 = \{x, x, y, y, y\}$

A note on SQL: SQL tables in general contain bags (not sets) of rows, and SQL supports certain bag operations on such tables. To be specific, it supports bag intersection and bag difference, through its operators `INTERSECT ALL` and `EXCEPT ALL`, respectively. It also supports union plus, through its operator `UNION ALL`. It doesn't support bag exclusive union, intersection star, or (oddly enough) true bag union. As for bag product, SQL's regular product operator—which is supported in a variety of syntactic styles, including, for example, the `CROSS` version of SQL's explicit `JOIN` operator—in fact represents an extended or expanded form of bag product, much as `TIMES` in **Tutorial D** represents an extended or expanded form of the set theory product operator. *See cartesian product.*

bag inclusion *See bag.*

bag membership (*Of an element*) The property of appearing in some given bag; the operation of testing for that property. Like set membership, q.v., bag membership is usually denoted by the symbol “ \in ” (sometimes pronounced *epsilon*, because it's a variant form of the lowercase Greek letter epsilon—i.e., “ ϵ ”—which is the first letter of the Greek word meaning “is”); thus, the boolean expression $x \in b$ —which is logically equivalent to the expression $\{x\} \subseteq b$ —returns `TRUE` if and only if element x does in fact appear at least once in bag b . *Note:* The expression $x \in b$ is logically equivalent to the expression $b \ni x$, where the symbol “ \ni ” denotes containment (the inverse of

membership, in effect).

bag operator *See* bag.

bang bang A relational operator, denoted in **Tutorial D** by the symbol “!!”. *See* image relation for further explanation.

base relation The value of a given base relvar at a given time. *Contrast* derived relation.

Examples: The relations that are the values of relvars S, P, and SP at some given time.

base relvar A relvar not defined in terms of others (*contrast* derived relvar.). *Note:* It’s a popular misconception that base relvars are physically stored, in the sense that they correspond directly to physically stored files and their tuples and attributes correspond directly to records and fields within those files (*see* direct image). But the relational model deliberately has nothing to say about physical storage; in particular, it categorically doesn’t say that base relvars, as such, are physically stored—not in the foregoing sense and not in any other sense, either. The only requirement is that there must be some defined mapping between whatever is physically stored and what’s perceived by the user (i.e., base relvars or derived relvars or a mixture of both).

Examples: Relvars S, P, and SP in the suppliers-and-parts database.

base table SQL analog of either a base relation or a base relvar, as the context demands. *See also* table.

base type (*Without inheritance*) Synonym for primitive type, q.v.

BCNF Boyce/Codd normal form.

behavior Term sometimes used (especially in OO contexts) to refer to the operators that apply to values and variables of some given type.

bi-implication Logical equivalence.

BI-IMPLIES Same as EQUIV.

bijection / bijective mapping Terms used interchangeably to mean a mapping, or function, from set $s1$ to set $s2$ such that each element of $s2$ is the image of exactly one element of $s1$; equivalently, a mapping that is both an injection and a surjection (in other words, a one to one correspondence, in the strict sense of that term, from $s1$ to $s2$). Also known as a bijective or “one to one onto” mapping. Note that if a given mapping is bijective, then it has an inverse mapping that’s bijective as well.

Examples: The mapping from integers x to their successors $x+1$ is a bijection from the set of all integers to itself. So is the inverse mapping from integers x to their predecessors $x-1$.

binary (*Of a heading, key, tuple, relation, etc.*) Of degree two. *Contrast* dyadic.

binding 1. In logic, quantifying a free variable, thereby converting it into to a bound variable. 2. (*Without inheritance*) In the programming context, the term *binding* has a variety of meanings—a name might be bound to a variable at compile time; a variable might be bound to a storage location at run time; a variable might be bound to a type at assignment time; and so on.

body A set of tuples all of the same type—especially, the set of tuples appearing in a given relation, or in a given relvar at a given time. Every

subset of a body is itself a body.

Examples: The set of tuples appearing in relvar S at some given time; any subset of that set (including the empty subset in particular).

BOOLEAN A scalar data type (the only one required by the relational model, and thus, in a relational DBMS, necessarily a system defined type), containing just two values: two truth values, to be precise, denoted in **Tutorial D** by the literals TRUE and FALSE, respectively.

boolean algebra 1. (*Simple case*) The truth values TRUE and FALSE, together with the logical operators NOT, OR, and AND, q.v. 2. (*General case*) Let s be a set; let " \leq " be a partial ordering, q.v., on s ; and let a monadic operator " \neg " ("complement") and distinct dyadic operators " $+$ " ("addition") and " $*$ " ("multiplication") be defined on s , such that (a) " \neg " satisfies the closure and involution laws; (b) " $+$ " and " $*$ " satisfy the closure, commutative, associative, distributive, idempotence, and absorption laws (meaning, in the case of the distributive law in particular, that each of " $+$ " and " $*$ " distributes over the other); and (c) " \neg ", " $+$ ", and " $*$ " together satisfy De Morgan's Laws, q.v. Let s also contain two elements 0 and 1 such that (a) 0 is the identity for " $+$ "; (b) 1 is the identity for " $*$ "; and (c) for all elements x in s , $0 \leq x \leq 1$. Then the combination of s and the operators " \leq ", " \neg ", " $+$ ", and " $*$ " is a boolean algebra. *Note:* Although they're usually referred to in this context as addition and multiplication, respectively, it must be clearly understood that " $+$ " and " $*$ " aren't necessarily the operators referred to by those names in conventional arithmetic.

Example (second definition only): Let s be an arbitrary set; let $P(s)$ be the power set (q.v.) of s ; and let " \leq ", " \neg ", " $+$ ", and " $*$ " denote set inclusion, set complement, set union, and set intersection, respectively

(“set complement” here meaning the relative complement, q.v., with respect to the set s). Then the combination of that power set $P(s)$ —not the set s , observe—and the operators “ \leq ”, “ \neg ”, “ $+$ ”, and “ $*$ ” as just defined is a boolean algebra, in which the empty set and the set s itself serve as the required additive identity and multiplicative identity, respectively. In other words, the familiar algebra of sets is in fact a boolean algebra.

boolean expression A logical expression, q.v.

boolean operator A logical operator, q.v. (especially one of the connectives, q.v.).

boolean value A value of type BOOLEAN, q.v.; in other words, a truth value (either TRUE or FALSE, in 2VL).

bound variable Within a predicate, q.v., a variable—more precisely, an occurrence of a reference to some variable—that either (a) appears within the scope of a quantifier that explicitly specifies that variable or (b) is that explicit specification itself. (The term *variable* is used here in the sense of logic, not in the programming language sense.) *Contrast* free variable.

Examples: Let the symbols x and y denote integers. Then the following expressions are both predicates, and x appears as a bound variable, twice, in each of them:

EXISTS x ($x > 3$)

EXISTS x ($x > 3$) AND $y < 7$

The first of these predicates is in fact a proposition, q.v., and its meaning is: *There exists an integer x such that x is greater than three* (a

proposition that evaluates to TRUE, as it happens). By contrast, the second predicate is not a proposition, because it involves a free variable (namely, y) as well as two bound ones; thus, it has no truth value. *Note:* Instantiating that second predicate—i.e., substituting an argument value for the free variable, or parameter, y —will convert it into a proposition, and that proposition will have a truth value, of course. For example, substituting the argument value 2 will yield the true proposition EXISTS x ($x > 3$) AND $2 < 7$. However (to repeat), the predicate as such has no truth value.

Turning to a database example, the following is a query (“Get suppliers who supply at least one part”) on the suppliers-and-parts database, expressed in tuple calculus, q.v.:

```
{ S } WHERE EXISTS SP ( SP.SNO = S.SNO )
```

The boolean expression following the keyword WHERE here is a predicate, and the references to SP in that predicate are bound (by contrast, the reference to S is free). Note, however, that in this particular example the symbols S and SP denote not only variables in the sense of logic but also variables in the conventional programming language sense—but that’s because we’ve indulged in a certain sleight of hand, as it were. Here’s an expanded version of the same example that should help clarify matters:

```
SX RANGES OVER { S } ;  
SPX RANGES OVER { SP } ;
```

```
{ SX } WHERE EXISTS SPX ( SPX.SNO = SX.SNO )
```

Here SX and SPX have been explicitly declared as range variables (q.v.)—in other words, they’re variables in the sense of logic—ranging

over (the current values of) relvars S and SP , respectively. Now it's the references to SPX that are bound and the reference to SX that's free (in the predicate following the keyword `WHERE` in both cases). In effect, what happened in the first version of the example was that we were appealing to a syntax rule that allowed a relvar name to be used to denote an implicitly defined range variable that ranges over (the current value of) the relvar with the same name. Note that SQL includes a syntax rule of exactly this kind.

Note: Let R be a range variable reference that occurs prior to the `WHERE` clause—i.e., in the proto tuple, q.v.—within some tuple calculus expression. If R also occurs in the predicate in that `WHERE` clause (which it usually but not invariably will), then it must be free, not bound, in that predicate. Observe that these remarks apply in particular to the references to the range variable SX in the example shown above.

Boyce/Codd normal form “The” normal form with respect to functional dependencies (FDs). Relvar R is in Boyce/Codd normal form (BCNF) if and only if every FD that holds in R is implied by some superkey of R —equivalently, if and only if for every nontrivial FD $X \rightarrow Y$ that holds in R , X is a superkey for R . Every BCNF relvar is in 3NF (and in fact in EKNF, q.v.). *Note:* Although being in BCNF clearly doesn't preclude being in the next higher normal form (4NF) as well, the term *BCNF* is often used loosely to refer to a relvar that's in BCNF and not in 4NF.

Example: With the normal forms it's often more instructive to show a counterexample rather than an example per se. Suppose, therefore, that relvar SP has an additional attribute `SNAME`, representing the name of the applicable supplier; suppose also that

supplier names are necessarily unique (i.e., no two suppliers ever have the same name at the same time). Then this revised version of SP has two keys, {SNO,PNO} and {SNAME,PNO}, and every subset of the heading—{QTY} in particular—is (of course) functionally dependent on both of them. However, the FDs {SNO} → {SNAME} and {SNAME} → {SNO} also hold in this relvar; these FDs are certainly not trivial, nor are they “arrows out of superkeys,” and so this version of relvar SP isn’t in BCNF (though it is in 3NF, and in fact in EKNF, q.v.).

brute force join A rather unsophisticated join implementation technique, involving an exhaustive comparison of each tuple from the first operand relation with each tuple from the second. Sometimes known as a nested loops join; this terminology is deprecated, however, since all join implementation techniques involve nested loops of some kind.

built in System defined. *Contrast* user defined.

business rule A declaration of some kind, usually expressed in natural language, that’s supposed to capture some aspect of what the data in the database means or how it’s constrained. There’s no consensus on any more precise definition of the term, but most if not all writers would probably agree (a) that relvar predicates, q.v., are an important special case and (b) that business rules other than relvar predicates map formally to integrity constraints, q.v.

Examples: Consider the suppliers-and-parts database. The predicate for suppliers is *Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY* (see the example under relvar predicate for further discussion). Along with this

predicate, there'll be rules that specify what type of information is denoted by the associated parameters—for example, a rule to the effect that the STATUS parameter (“status values”) denotes values expressed in integers. Then there'll be rules that constrain the values those parameters can take for a given supplier considered in isolation—for example, a rule that says status values must lie in the range 1 to 100, inclusive. There'll also be rules that constrain the set of suppliers taken as a whole, independent of other “entities” that might also be represented in the database—for example, a rule to the effect that supplier numbers must be unique. Finally, there'll be rules that constrain suppliers considered in combination with certain other entities—for example, a rule to the effect that every shipment must involve some known supplier, or a rule to the effect that no supplier with status less than 20 can supply part P6.

Note: The set of all business rules that apply in some given context—for example, the set of rules that apply to a given database, or to a given enterprise in its entirety—is sometimes referred to as the conceptual schema (for the context in question). However, this latter term resembles the term *business rule* itself in that it too has no universally agreed precise definition.



calculus 1. Generically, a system of formal computation (the Latin word *calculus* means a pebble, perhaps used in counting or some other form of reckoning). 2. Relational calculus specifically, q.v. (if the context demands).

candidate key Loosely, a unique identifier. More precisely, let K be a

subset of the heading of relvar R ; then K is a candidate key (key for short) for, or of, R if and only if (a) no possible value for R contains two distinct tuples with the same value for K (the uniqueness property), while (b) the same can't be said for any proper subset of K (the irreducibility property). Note that every relvar, base or derived, does have at least one key. Note too that, by definition, keys are sets of attributes (and key values are therefore tuples); however, if the set of attributes constituting some key K contains just one attribute A , then it's common, though strictly incorrect, to speak informally of that attribute A per se as being that key. Note further that if K is a key for relvar R , then the functional dependency $K \rightarrow X$ necessarily holds in R for all subsets X of the heading of R . Note finally that the qualifier *candidate* is a hangover from earlier times when more of a distinction was made between primary and alternate keys and a generic term was required to cover both. It could be dropped without serious loss, and usually is. *See also* alternate key; key constraint; primary key. *Contrast* subkey; superkey.

Examples: In the suppliers-and-parts database, {SNO}, {PNO}, and {SNO,PNO} are the sole keys for relvars S, P, and SP, respectively. Note that {SNAME} isn't a key for S, because SNAME values aren't necessarily unique (even though the sample values shown in Fig. 1 do happen to be unique). Note too that, e.g., {SNO,CITY} isn't a key for S either, because although its values are necessarily unique, it isn't irreducible—we could remove the CITY attribute, and what would be left would still have the uniqueness property. (Irreducibility is desirable because, among other things, the system would be enforcing the wrong integrity constraint without it. In the case at hand, for example, it wouldn't be enforcing the constraint that supplier numbers are

“globally” unique, but merely the weaker constraint that they’re unique within each city.)

canonical form Given a set $s1$, together with a stated notion of equivalence among the elements of that set, subset $s2$ of $s1$ is a set of canonical forms for $s1$ if and only if every element $x1$ of $s1$ is equivalent to just one element $x2$ of $s2$ under that notion of equivalence (and that element $x2$ is said to be the canonical form for the element $x1$). The set $s2$ taken as a whole is also sometimes said to be the canonical form for the set $s1$ as such. Various “interesting” properties that apply to $s1$ also apply to $s2$; thus, we can study just the “small” set $s2$, not the “large” set $s1$, in order to prove a variety of interesting theorems or results. *Note:* It would be usual to require also that every element of $s2$ be equivalent (under the stated notion of equivalence) to at least one element of $s1$. Note also that the set of all elements $x1$ of $s1$ that are equivalent to some specific element $x2$ of $s2$ in fact constitutes an equivalence class, q.v.

Example: Let $s1$ be the set of nonnegative integers $\{0,1,2,\dots\}$ and let two such integers be equivalent if and only if they leave the same remainder on division by five. Then we can define $s2$ to be the set $\{0,1,2,3,4\}$. (Note in particular that $s2$ here is finite while $s1$ is infinite.) As for an “interesting” theorem that applies in this example, let $x1, y1,$ and $z1$ be any three elements of $s1$, and let their canonical forms in $s2$ be $x2, y2,$ and $z2$, respectively; then the product $y1 * z1$ is equivalent to $x1$ if and only if the product $y2 * z2$ is equivalent to $x2$.

cardinality The number of elements in a bag or (especially) set; hence, of a relation, the number of tuples in the body of that relation. Also used (a) of a relvar, to mean the cardinality of the relation that’s the

value of that relvar at a given time; (b) of an attribute of a relation or relvar, to mean the cardinality of the set of distinct values of that attribute appearing in the body of that relation or relvar (at a given time, in the case of a relvar). Of course, the cardinality of attribute A of relation r is the same as the cardinality of the projection $r\{A\}$ of that relation on that attribute; definition (b) here is thus strictly redundant.

Examples: In Fig. 1, (a) the cardinality of the relation that's the current value of relvar SP is twelve (and the cardinality of relvar SP is thus currently twelve also); (b) the cardinality of attribute SNO in that relation is four (and the cardinality of that attribute in relvar SP is thus currently four also).

Note: Since types are sets (*see* type), types in particular have a cardinality: viz., the number of distinct values of the type in question. For example, the cardinality of type SNO is a count of all possible supplier numbers.

cardinality constraint 1. A constraint on the cardinality of a given relvar (a special case of a relvar constraint, q.v.); for example, a constraint to the effect that there can never be more than ten suppliers at any one time. 2. Let r be a relationship (q.v.) from set $s1$ to set $s2$, and let $x1$ and $x2$ be typical elements of $s1$ and $s2$, respectively. In E/R modeling (q.v.) and similar design schemes, then, the following are all cardinality constraints that can be specified for each of $s1$ and $s2$: 1, 0..1, 0.. m , 1.. m . (Other notations are also used.) For definiteness, assume the constraint in question has been specified for set $s2$; then that constraint indicates how many $x2$'s correspond to any given $x1$ in relationship r . The various specifications have the following meanings: 1 means there must be exactly one such $x2$; 0..1 means there must be at most one such $x2$; 0.. m means there can be any number of such $x2$'s,

from zero to some unspecified upper bound m ; and $1..m$ means there can be any number of such x_2 's, from one to some unspecified upper bound m . *Note:* The terms *optional participation* and *mandatory participation* are sometimes used to refer to the case where the lower bound is 0 and the case where it's 1, respectively; however, there's no universal agreement on what these terms mean, and they're probably best avoided.

cartesian join Same as cartesian product.

cartesian product 1. (*Dyadic case*) Let relations r_1 and r_2 have no attribute names in common. Then (and only then) the expression r_1 TIMES r_2 denotes the cartesian product of r_1 and r_2 , and it returns the relation with heading the set theory union of the headings of r_1 and r_2 and body the set of all tuples t such that t is the set theory union of a tuple from r_1 and a tuple from r_2 . 2. (*N-adic case*) Let relations r_1, r_2, \dots, r_n ($n \geq 0$) be such that no two of them have any attribute names in common. Then (and only then) the expression TIMES $\{r_1, r_2, \dots, r_n\}$ denotes the cartesian product of r_1, r_2, \dots, r_n , and it returns the relation with heading the set theory union of the headings of r_1, r_2, \dots, r_n and body the set of all tuples t such that t is the set theory union of a tuple from r_1 , a tuple from r_2 , ..., and a tuple from r_n . *Note:* The relational cartesian product operator differs in several respects from the mathematical or set theory operator of the same name, q.v., and is sometimes explicitly said to be an expanded, or extended, cartesian product for that reason. *See also* tuple product.

Example: The expression $S\{SNO\}$ TIMES $P\{PNO\}$ denotes the cartesian product of the projections on $\{SNO\}$ and $\{PNO\}$, respectively, of the relations that are the current values of relvars S and

P, respectively. That product is a relation of type RELATION {SNO SNO, PNO PNO}. Moreover, if the current values of relvars S and P are s and p , respectively, the body of that relation contains (a) all possible tuples of the form $\langle sno, pno \rangle$ such that the tuple $\langle sno \rangle$ appears in s and the tuple $\langle pno \rangle$ appears in p and (b) no other tuples. (Given the values in Fig. 1, the result has cardinality 30.)

Note: TIMES is actually a special case of JOIN, as the following alternative definitions make explicit: 1. (*Dyadic case*) If and only if $r1$ and $r2$ have no attribute names in common, the expression $r1$ TIMES $r2$ denotes the cartesian product of $r1$ and $r2$, and it reduces to $r1$ JOIN $r2$. In the foregoing example, therefore, the expression $S\{SNO\}$ TIMES $P\{PNO\}$ is logically equivalent to the expression $S\{SNO\}$ JOIN $P\{PNO\}$. 2. (*N-adic case*) If and only if no two of $r1, r2, \dots, rn$ ($n \geq 0$) have any attribute names in common, the expression TIMES $\{r1, r2, \dots, rn\}$ denotes the cartesian product of $r1, r2, \dots, rn$, and it reduces to JOIN $\{r1, r2, \dots, rn\}$. In the foregoing dyadic example, therefore, the expression $S\{SNO\}$ TIMES $P\{PNO\}$ —which could alternatively have been written TIMES $\{S\{SNO\}, P\{PNO\}\}$ —is logically equivalent to the expression JOIN $\{S\{SNO\}, P\{PNO\}\}$.

cartesian product (bag theory) See bag.

cartesian product (set theory) The cartesian product of two sets $s1$ and $s2$, $s1 \times s2$, is the set of all ordered pairs of elements $\langle x1, x2 \rangle$ such that the first element of the pair, $x1$, is an element of $s1$ and the second element of the pair, $x2$, is an element of $s2$. *Note:* This definition can obviously be extended to apply to any number of sets (and is so, tacitly, in the mathematical definition of a relation, q.v.).

cascading Performing an update of the same general kind as, but in

addition to, some explicitly requested update; hence, a compensatory action, q.v. (but an important special case). Cascading a delete operation is a typical example. Note, however, that such cascading should occur, if and when logically required, regardless of the concrete syntactic form in which the original update request is expressed. For example, an update expressed as a pure relational assignment (using “:=”), q.v., should nevertheless cause a cascade delete to be performed—assuming a pertinent cascade DELETE rule has been defined in the first place, of course.

CAST Shorthand for `CAST_AS_T` for some T .

CAST_AS_T Let T be a scalar type. Then `CAST_AS_T` is an operator for mapping values of some scalar type T' to corresponding values of type T (i.e., for performing what’s loosely called type conversion—specifically, conversion from type T' to type T). *Note:* Type T here is said to be the target type. *See also* coercion.

Example: Let variables `N` and `C` be of declared types `INTEGER` and `CHAR`, respectively. Then `CAST_AS_CHAR (N)` casts or “converts” the current value of `N` to character string form, and `CAST_AS_INTEGER (C)` casts or “converts” the current value of `C` to integer form. (In the latter case, of course, the operation will fail if the current value in question isn’t a character string representation of some integer.)

Observe that the argument to `CAST_AS_T` will typically be allowed to be of different types on different invocations; in other words, the operator will typically be overloaded (*see* overloading). Observe also that the number of `CAST` operators actually needed in any given situation can sometimes be reduced by good type design. For

example, consider temperatures. A good design will involve a single `TEMPERATURE` type, together with operators (namely, selectors and `THE_` operators) to expose a Celsius representation, a Fahrenheit representation, and so on (*see* types vs. units). A bad design would involve different types—`CELSIUS`, `FAHRENHEIT`, and so on— together with a set of `CAST` operators to convert between them.

catalog Within a given database, a set of database relvars that describe the database in question. *Note:* The catalog includes descriptions of the catalog relvars themselves; in other words, the catalog is self-describing. It's sometimes said to contain metadata, q.v. Catalog relvars are usually updated not by explicit assignment operations but rather by more user friendly data definition operators, q.v. (which are nevertheless essentially just shorthand for certain relational assignments—often multiple assignments, q.v.).

catalog relvar A special kind of database relvar, q.v. (probably but not necessarily a base relvar), forming part of the database catalog. *See* catalog.

***Cautious Design Principle** See Principle of Cautious Design.*

cell Term sometimes used to refer to a row and column intersection in a table; not to be confused with the content of the cell in question. *Note:* The concept of “cells” makes sense in connection with the idea that a table is a picture of a relation (*see* table) but not in connection with the idea that a table *is* such a relation, which is why this definition is framed in terms of tables and not relations. It's true that we might think, very informally, of some relation in terms of “tuple and attribute intersections,” but we can't sensibly regard those intersections as being

somehow distinct from their content. (Take the content away from a relation and nothing remains; as Lewis Carroll might have remarked, a relation without its content would be like a grin without a cat.)

chase See chase algorithm.

chase algorithm An algorithm for determining whether some specified dependency (q.v.) d is a logical consequence of some specified set of dependencies D . In outline (and speaking somewhat loosely), the algorithm works by defining a relation r containing sample tuples conforming to the premises (q.v.) of d and repeatedly applying the dependencies of D to r (possibly adding further tuples to r in the process). Then:

- If d is an equality generating dependency (q.v.) and this process causes the pertinent equality condition to be satisfied, then d is a logical consequence of D .
- If d is a tuple generating dependency (q.v.) and this process causes the pertinent conclusion tuple(s) to be generated, then d is a logical consequence of D .
- Otherwise r is a relation that satisfies the dependencies of D but doesn't satisfy d ; r thus serves as a counterexample to show that d isn't a logical consequence of D .

Examples: Here are a couple of very simple examples of the chase in action. First, consider a heading consisting of attributes A , B , and C (and no others). Let AB denote the set $\{A,B\}$, and similarly for AC . Let \mathcal{J} and F be the JD $\bowtie \{AB,AC\}$ and the FD $A \rightarrow B$, respectively. Here then is a proof that \mathcal{J} is a logical consequence of F :


1. \mathcal{J} says “If $\langle a1, b1, c1 \rangle$ and $\langle a1, b2, c2 \rangle$ appear, then $\langle a1, b1, c2 \rangle$ and $\langle a1, b2, c1 \rangle$ appear.” So these two tuples—call them $t1$ and $t2$, respectively—represent the premises of \mathcal{J} :

$t1$: $a1$ $b1$ $c1$
 $t2$: $a1$ $b2$ $c2$

2. If these tuples appear, then we have $b1 = b2$, thanks to F , and so the tuples

$t3$: $a1$ $b1$ $c2$
 $t4$: $a1$ $b2$ $c1$

“also” appear (“also” in quotation marks because $t3$ and $t4$ are basically just $t1$ and $t2$ in disguise, as it were, shown in reverse order). But “if $t1$ and $t2$ appear, then $t3$ and $t4$ appear” is exactly what \mathcal{J} says; i.e., $t3$ and $t4$ are the conclusion of \mathcal{J} , given $t1$ and $t2$ as premises. So \mathcal{J} is a logical consequence of F . *Note:* This result is basically Heath’s Theorem, q.v.

By way of a second example, again let \mathcal{J} and F be the JD  $\{AB, AC\}$ and the FD $A \rightarrow B$, respectively. Here then is a proof that F doesn’t follow from \mathcal{J} (i.e., the converse of Heath’s Theorem is false):

1. F says “If $\langle a1, b1, c1 \rangle$ and $\langle a1, b1, c2 \rangle$ appear, then $b1 = b2$.” So these two tuples $t1$ and $t2$ represent the premises of F :

$t1$: $a1$ $b1$ $c1$
 $t2$: $a1$ $b2$ $c2$

2. If these tuples appear, then the following tuples also appear, thanks to \mathcal{J} :

$t3$: $a1$ $b1$ $c2$

$t_4 : a_1 \ b_2 \ c_1$

Observe now that tuples t_1 - t_4 taken together satisfy \mathcal{J} without requiring that $b_1 = b_2$. They thus constitute (the body of) a relation that satisfies \mathcal{J} but not F . So F isn't a logical consequence of \mathcal{J} .

child / child table Deprecated, because inappropriate, terms sometimes used in SQL contexts to mean (the SQL analog of) a referencing relvar, q.v.

class 1. (*Mathematics*) Term usually used just as a synonym for set. However, it's also used to refer to certain collections—specifically, collections in which the elements are themselves sets—that aren't regarded as legitimate sets for some reason. For example, the (infinite) collection C of all sets is regarded by some mathematicians as a class but not a set. (One argument against regarding C as a set is that the cardinality of the power set, q.v., of any given set s is always greater than that of s ; thus, if s is in fact C , the collection of all sets, then there's apparently at least one collection of sets that's of greater cardinality than s , which is a contradiction.) *See also* equivalence class.

2. (*OO*) Term used to mean, variously, (a) a type; (b) the implementation or physical representation of some type; (c) a type and one of its implementations in combination; (d) the set of all values of some type currently in use; and possibly (e) other things besides.

closed expression *See* open expression.

closed WFF A WFF, q.v., that denotes a proposition. *Contrast* open WFF.

Closed World Assumption Loosely, the assumption that everything stated or implied by the database is true and everything else is false. More precisely, let relvar R have predicate P (see relvar predicate). Then *The Closed World Assumption* (CWA) says (a) if tuple t appears in R at time T , then the instantiation of P corresponding to t is assumed to be true at time T ; conversely, (b) if tuple t has the same heading as R but doesn't appear in R at time T , then the instantiation of P corresponding to t is assumed to be false at time T . Loosely speaking, in other words, tuple t appears in relvar R at a given time if and only if it satisfies the predicate for R at that time. What's more, it follows that if proposition p is represented by a tuple that appears in some relation that can be derived from the relations that are the values of the database relvars at time T —see derived relation—then proposition p is true at time T (which is why the phrase “or implied” appears in the original loose characterization). *Contrast Open World Assumption.* *Caveat:* Be aware that very different interpretations of the term “closed world” can be found in the general computing literature—even in the database literature specifically, sometimes.

Examples: The tuple $\langle S1, P1, 300 \rangle$ currently appears in relvar SP; we can therefore assume that it's currently the case that supplier S1 supplies part P1 in quantity 300. By contrast, the tuple $\langle S5, P6, 250 \rangle$ doesn't currently appear in that relvar, though presumably it could; we can therefore assume that it's currently not the case that supplier S5 supplies part P6 in quantity 250.

As for an example of implied information, the tuple $\langle S3 \rangle$ currently appears in the projection of relvar SP on {SNO}; we can therefore assume that it's currently the case that supplier S3 supplies some part in some quantity. By contrast, the tuple $\langle S5 \rangle$ doesn't currently appear

in that projection, though presumably it could; we can therefore assume that it's currently not the case that supplier S5 supplies any part in any quantity.

Note: It follows from the CWA that if relvars $R1$ and $R2$ have predicates $P1$ and $P2$, respectively, and if $P1$ and $P2$ are both currently satisfied by the same tuple t , then t must currently appear in both $R1$ and $R2$. As a rule of thumb, it's a good idea to design the database in such a way as to ensure that $P1$ and $P2$ are specific enough to preclude such a situation (so long as $R1$ and $R2$ are both base relvars, at any rate).

closure 1. (*Of algebras in general*) See Laws of Algebra. 2. (*Of relational algebra in particular*) The property that the result of every relational algebra operation is a relation. 3. (*Of a set of FDs*) The set of all FDs implied by the given set (*see* Armstrong's axioms). 4. (*Of a set of attributes*) Loosely, the set of all attributes functionally dependent on those in the given set. More precisely, let H be a heading, let F be a set of FDs with respect to H , and let Z be a subset of H . Then the closure Z^+ of Z under F is the maximal subset C of H such that the FD $Z \rightarrow C$ is implied by the FDs in F (again *see* Armstrong's axioms).

closure, transitive *See* transitive closure.

CNF Conjunctive normal form.

Codd, E. F. The inventor of the relational model. See especially the papers (a) "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599, August 19th, 1969 (Codd's very first publication on the relational model); (b) "A Relational Model of Data for Large Shared Data Banks," *CACM*

13, No. 6, June 1970 (a revised and extended version of that first paper); and (c) “Relational Completeness of Data Base Sublanguages,” in Randall Rustin (ed.), *Data Base Systems: Courant Computer Science Symposia 6*, Prentice-Hall (1972). The last of these papers in particular contains formal definitions of a relational calculus (actually a tuple calculus, q.v.) and a relational algebra (“Codd’s relational algebra,” q.v.), as well as of Codd’s reduction algorithm, q.v. *Note*: The 1969 paper was republished in *ACM SIGMOD Record* 38, No. 1 (March 2009); the 1970 paper was republished in *Milestones of Research—Selected Papers 1958-1982 (CACM 25th Anniversary Issue)*, *CACM* 26, No. 1 (January 1983) and elsewhere. The 1972 paper has never been republished in hard copy form but can be found on the web.

Codd’s reduction algorithm An algorithm for reducing a given tuple calculus expression to a logically equivalent expression of Codd’s relational algebra. Among other things, the algorithm relies on the fact that—speaking a trifle loosely (*see* division)—the operators project and divide are algebraic counterparts to the existential quantifier and the universal quantifier, respectively, of tuple calculus. Note that the existence of such an algorithm suffices to show that Codd’s algebra is relationally complete, q.v.

Codd’s relational algebra Codd’s first few papers all included definitions of certain operators of an algebraic nature, but the exact set of operations defined varied somewhat from one paper to the next, and so did the precise definitions. As a consequence, it’s a little difficult to say exactly what’s meant by the term “Codd’s relational algebra.” But most writers would agree that it does at least include the following operators in some shape or form: cartesian product, union, intersection, difference, restriction, projection, natural and theta join,

and division. Note that extension and aggregate operators are definitely not included. Nor are relational comparisons of any kind.

codomain *See* function.

coercion Implicit type conversion (usually best avoided). Note that implicit conversion will be possible only when explicit conversion is also possible (unless the types involved are both system defined; a badly designed language might conceivably support coercion, but not explicit conversion, between such types). *Note:* Elsewhere—e.g., in its definitions of the various relational operations—this dictionary assumes for simplicity that coercions aren't supported.

collection (*Of an attribute, type, value, or variable; noun used as an adjective; not much used in the relational context*) A special case of nonscalar, q.v., in which the user visible component parts are usually required all to be of the same type. For example, array and relation types might be regarded as collection types, but tuple types usually wouldn't be. The term is also used as a noun, in which case it serves as an abbreviation for any or all of *collection type* or *collection value* or *collection variable*, as the context demands. *See also* aggregate.

collection type Same as aggregate type. *See* collection.

column 1. Term used variously to refer to the SQL analog of (a) an attribute of some relation or relvar, or (b) the bag or set of values of some attribute of some relation or relvar, or (c) the type of some attribute of some relation or relvar, or sometimes even (d) an attribute of some tuple or tuplevar or (e) the value of some attribute of some tuple or tuplevar or (f) the type of some attribute of some tuple or tuplevar (as the context demands). 2. More generally, a picture of an

attribute (on paper, for example). *See also* cell; row; table.

common attribute An attribute that's common to two or more relations and/or relvars and/or tuples and/or tuplevars.

Examples: In the suppliers-and-parts database, (a) $\langle \text{SNO}, \text{SNO} \rangle$ is a common attribute for relvars S and SP; (b) $\langle \text{PNO}, \text{PNO} \rangle$ is a common attribute for relvars SP and P; and (c) $\langle \text{CITY}, \text{CHAR} \rangle$ is a common attribute for relvars S and P. We might also say, more simply but less formally, just that (a) SNO is a common attribute for S and SP, (b) PNO is a common attribute for SP and P, and (c) CITY is a common attribute for S and P.

commutative group *See* group (mathematics).

commutativity Let Op be a dyadic operator, and assume for definiteness that it's expressed in infix style. Then Op is commutative if and only if, for all x and y , $x Op y = y Op x$.

Examples: In ordinary arithmetic, addition (“+”) is commutative, because

$$x + y = y + x$$

for all x and y . By contrast, subtraction (“-”) is not commutative. In the same kind of way, UNION and JOIN are commutative in relational algebra while MINUS is not. Likewise, OR and AND are commutative in logic while IMPLIES is not. *Note:* It so happens that all of the commutative operators just mentioned are also associative, q.v. By contrast, the logical operators NAND and NOR, q.v., are examples of operators that are commutative but not associative. So too is COMPOSE, q.v.

comparison A boolean expression of the form $(exp1) \theta (exp2)$, where $exp1$ and $exp2$ are expressions of the same type T and θ is any comparison operator that makes sense for values of type T (certainly “=” or “≠”, perhaps “<” and “>” also, and so on). *Note:* The parentheses enclosing $exp1$ and $exp2$ in the comparison might not be needed in practice.

compensating action / compensatory action Terms used interchangeably to mean an update performed automatically in addition to some explicitly requested update, with the aim of avoiding some integrity violation that might otherwise occur. Cascading a delete operation in order to avoid a referential integrity violation is a typical example; so too is the update performed on some underlying base relvar in response to some requested view update. Note that such compensatory actions should be performed, if and when logically required, regardless of the concrete syntactic form in which the original update request is expressed. For example, an INSERT operation expressed as a pure relational assignment (using “:=”), q.v., should nevertheless cause the compensatory action for that INSERT to be performed—assuming, of course, that such an action has been defined in the first place.

Note: Compensatory actions should be specified declaratively, and users should generally be aware of them (that is, users should generally know when their update requests are shorthand for some more extensive set of actions), for otherwise they might perceive an apparent violation of *The Assignment Principle*, q.v. Note too, however, that—at least with regard to the compensatory actions needed in connection with view updating—the system should in fact be able to work out for itself what compensatory actions are needed, implying that the

required declarative specifications can and should be provided by the system. *See also* controlled redundancy; multiple assignment. *Contrast* triggered procedure.

complement Let relation r have heading H and body B . Then the complement of r is the relation with heading H and body consisting of all tuples with heading H not appearing in B .

complement (set theory) The complement—also known as the absolute complement—of a set s is the set of all elements not appearing in s . *Note:* The difference $s1 - s2$ between sets $s1$ and $s2$, in that order, is sometimes referred to as the relative complement of $s2$ with respect to $s1$ (*see* difference); thus, the absolute complement of s is in fact the relative complement of s with respect to the universal set, q.v. *See also* boolean algebra (second definition).

complementarity 1. (*Logic*) The disjunction of a predicate and its negation is a tautology, q.v.; the conjunction of a predicate and its negation is a contradiction, q.v. 2. (*Set theory*) The union of a set and its complement is the universal set, q.v.; the intersection of a set and its complement is the empty set, q.v.

Example (first definition only): The following identities are just a representation of the foregoing logic laws in symbolic form, but they might be a little easier to understand than the prose versions:

$$p \text{ OR } (\text{ NOT } p) \equiv \text{ TRUE}$$

$$p \text{ AND } (\text{ NOT } p) \equiv \text{ FALSE}$$

completeness (*Of a formal system; not to be confused with computational, relational, or truth functional completeness, q.v.*) A formal system is

complete if and only if, given a set s of sentences of the system, all sentences implied by those in s can be derived using the rules of inference of that system (i.e., all tautologies are theorems). *See also* soundness.

component 1. (*Of a JD*) *See* join dependency. 2. (*Of a possrep*) *See* possrep. 3. (*Of a tuple*) *See* tuple component.

COMPOSE *See* composition.

composite attribute / compound attribute Deprecated terms used interchangeably to mean a combination of two or more attributes. The terms are deprecated in part because a “composite” or “compound” attribute isn’t actually an attribute at all.

composite key / compound key Terms used interchangeably to mean a key consisting of two or more attributes. *Contrast* simple key.

Example: In the suppliers-and-parts database, SP is the only relvar with a composite key (namely, {SNO,PNO}).

composite predicate / compound predicate Terms used interchangeably to mean a predicate that involves at least one connective. *Contrast* simple predicate.

composite proposition / compound proposition Terms used interchangeably to mean a proposition that involves at least one connective. *Contrast* simple proposition.

composite statement / compound statement (*Programming languages*) Terms used interchangeably to mean a statement that contains other statements syntactically nested inside itself. *Contrast* atomic statement.

Examples: Conventional IF, DO, WHILE, and CASE statements; BEGIN – END statement blocks; multiple assignment statements (q.v.); and many others.

composition 1. (*Dyadic case*) Let relations $r1$ and $r2$ be joinable, q.v., and let their common attributes be called $A1, A2, \dots, Am$ ($m \geq 0$). Then (and only then) the expression $r1$ COMPOSE $r2$ denotes the composition of $r1$ and $r2$, and it returns the relation denoted by the expression $(r1$ JOIN $r2)$ {ALL BUT $A1, A2, \dots, Am$ }. See also tuple composition. *Note:* Dyadic COMPOSE is unusual, in a sense, in that it's commutative but not associative. 2. (*N-adic case*) Let relations $r1, r2, \dots, rn$ ($n \geq 0$) be n -way joinable, q.v., and let the attributes common to at least two of those relations be called $A1, A2, \dots, Am$ ($m \geq 0$). Then (and only then) the expression COMPOSE $\{r1, r2, \dots, rn\}$ denotes the composition of $r1, r2, \dots, rn$, and it returns the relation denoted by the expression (JOIN $\{r1, r2, \dots, rn\}$) {ALL BUT $A1, A2, \dots, Am$ }. *Caveat:* This definition is motivated by a desire to preserve commutativity (of a kind)—more precisely, to preserve the property that the value of the expression COMPOSE $\{r1, r2, \dots, rn\}$ is independent of the order in which relations $r1, r2, \dots, rn$ are specified. It also has the property that the expression COMPOSE $\{r1, r2\}$ is logically equivalent to the expression $r1$ COMPOSE $r2$ (i.e., the n -adic version degenerates to its dyadic counterpart in the special case where $n = 2$). On the other hand, the operator isn't associative; in other words, the expressions COMPOSE $\{r1, \text{COMPOSE } \{r2, r3\}\}$, COMPOSE $\{\text{COMPOSE } \{r1, r2\}, r3\}$, and COMPOSE $\{r1, r2, r3\}$ aren't logically equivalent. Thus, n -adic COMPOSE as here defined isn't just shorthand for repeated dyadic COMPOSE; rather, it's a logically distinct operator. Contrast the situation with, e.g., n -adic JOIN, which is shorthand for

repeated dyadic JOIN.

Example: The expression $S\{SNO,CITY\}$ COMPOSE $P\{PNO,CITY\}$ denotes the composition of the projections on $\{SNO,CITY\}$ and $\{PNO,CITY\}$, respectively, of the relations that are the current values of relvars S and P , respectively. That composition is a relation of type $RELATION \{SNO SNO, PNO PNO\}$. Moreover, if the current values of relvars S and P are s and p , respectively, the body of that relation consists of all tuples of the form $\langle sno, pno \rangle$ such that sno is a supplier number appearing in s , pno is a part number appearing in p , and supplier sno and part pno are located in the same city.

computable function A function that can be computed by a Turing machine in a finite number of steps).

computational completeness A language is computationally complete if and only if it supports the computation of all computable functions.

Examples: C++; PL/I; SQL; **Tutorial D**; and many others. Codd's relational algebra, q.v., is an example of a language that's not computationally complete (basically because it includes no support for either EXTEND, q.v., or aggregate operators, q.v.).

conceptual design Synonym for conceptual modeling; in other words, the process, or the result of the process, of producing a conceptual schema, q.v. Note, however, that the boundaries between conceptual design and logical design are far from being hard and fast, and might not exist at all in some cases.

conceptual modeling Term sometimes used as a synonym for the process of conceptual design, q.v. *See also* semantic modeling.

conceptual schema *See* business rule.

conclusion In logic, that which a proof proves or an attempted proof attempts to prove. *See* in particular equality generating dependency; tuple generating dependency.

conditional expression A logical expression, q.v.

conditional operator A logical operator, q.v. (especially one of the connectives, q.v.).

conjunct A predicate that's ANDed with zero or more others.

conjunction 1. (*Dyadic case*) If and only if p and q are predicates, their conjunction (p) AND (q) is a predicate also. Let (ip) AND (iq) be an invocation of that predicate, where ip and iq are invocations of p and q , respectively. Then that invocation (ip) AND (iq) evaluates to TRUE if and only if ip and iq both evaluate to TRUE. *Note:* The parentheses enclosing p and q in the predicate, and ip and iq in the invocation, might not be needed in practice. 2. (*N-adic case*) Let p_1, p_2, \dots, p_n ($n \geq 0$) be predicates; then (and only then) the conjunction AND $\{p_1, p_2, \dots, p_n\}$ is defined to be shorthand for the expression (p_1) AND (p_2) AND ... AND (p_n) . (*Note* that this expression evaluates to TRUE if $n = 0$, because TRUE is the identity with respect to AND.) *See also* universal quantifier.

conjunctive normal form A predicate is in conjunctive normal form, CNF, if and only if it's of the form (p_1) AND (p_2) AND ... AND (p_n) , where none of the conjuncts $(p_1), (p_2), \dots, (p_n)$ involves any ANDs—more precisely, where each of p_1, p_2, \dots, p_n is a disjunction of literals (*see* literal, second definition). *Note:* The parentheses enclosing the

individual predicates p_1, p_2, \dots, p_n might not be needed in practice.

connection trap A term used by some writers to refer to an alleged flaw in the relational model. By way of illustration, consider the expression $(S \text{ JOIN } P) \{SNO, PNO\}$. This expression denotes a relation, r say, that—given the sample values in Fig. 1—happens to contain the tuple $\langle S2, P5 \rangle$, because supplier S2 and part P5 are both located in the same city, Paris. Now, from the fact that this tuple appears in r , it obviously can't be inferred (at least, not validly) that supplier S2 supplies part P5—the predicate for r is *Supplier SNO and part PNO are located in the same city*, not *Supplier SNO supplies part PNO* (speaking a trifle loosely). However, it's claimed by certain writers that users will nevertheless make that invalid inference, and hence that the relational model is flawed because it lets users fall into that trap. But it should be clear from the example that the flaw lies not with the model but with a failure on the part of those users—or those writers, perhaps—to understand the semantics of join properly. (Indeed, the flaw, such as it is, really has nothing to do with the relational model as such. Instead, it has to do with the intrinsic nature of data.) *Note:* As the example suggests, the term *connection trap* is typically regarded as an issue that arises in connection with join specifically (indeed, some writers even refer to it as *the join trap* for that reason); however, similar issues can clearly arise in connection with other operations also.

connective A read-only monadic or dyadic logical operator. There are exactly 20 connectives in two-valued logic, four monadic and 16 dyadic (corresponding directly to the four possible monadic and 16 possible dyadic truth tables). The connectives most frequently encountered in practice are NOT (negation), OR (disjunction), AND (conjunction), IMPLIES (implication), and EQUIV (equivalence); others include

NAND, NOR, and XOR, q.v. *Note:* A variety of other symbols and keywords, some but not all of which are mentioned in this dictionary, are also used to denote these connectives. *See also* *nVL*; truth functional completeness; two-valued logic; three-valued logic.

Here for the record are truth tables for the connectives of two-valued logic. First the monadic ones:

—		—
T		T
F		T

—		—
T		T
F		F

		NOT		
—		—		—
T		F		
F		T		

—		—
T		F
F		F

And here are the dyadic ones (using, for typographic reasons, IF for IMPLIES and IFF for EQUIV):

		T	F
—		—	—
T		T	T
F		T	T

IF		T	F
—		—	—
T		T	F
F		T	T

NAND		T	F
—		—	—
T		F	T
F		T	T

		T	F
—		—	—
T		F	F
F		T	T

OR		T	F
—		—	—
T		T	T
F		T	F

		T	F
—		—	—
T		T	F
F		T	F

XOR		T	F
—		—	—
T		F	T
F		T	F

		T	F
—		—	—
T		F	F
F		T	F

		T	F
—		—	—
T		T	T
F		F	T

IFF		T	F
—		—	—
T		T	F
F		F	T

		T	F
—		—	—
T		F	T
F		F	T

NOR		T	F
—		—	—
T		F	F
F		F	T

		T	F
—		—	—
T		T	T
F		F	F

AND		T	F
—		—	—
T		T	F
F		F	F

		T	F
—		—	—
T		F	T
F		F	F

		T	F
—		—	—
T		F	F
F		F	F

consequent *See* implication.

consistency Loosely, a synonym for integrity, q.v.; sometimes used more specifically to refer to the state of a database that conforms to just those declared integrity constraints that have to do with controlled redundancy, q.v. Note, however, that there's an important distinction to be drawn between what might be called formal consistency and informal consistency. To elaborate:

- (*Formal consistency*) Formally speaking, a database is in a state of consistency if and only if it conforms to all declared integrity constraints—and the term *consistency*, unqualified, is usually taken to mean consistency in this formal (or logical) sense, unless the context demands otherwise. *Note:* It follows from this definition that a database is formally inconsistent if and only if there's some declared constraint it should satisfy but doesn't. Equivalently, a database is formally inconsistent if and only if it's self-contradictory—meaning that it asserts, either explicitly or implicitly, that some proposition p and its negation $\text{NOT}(p)$ are both true. The relational model requires databases to be consistent in this formal sense at all times (where “at all times” effectively means at statement boundaries or, loosely, “at semicolons”). Consistency in this sense is necessary but not sufficient for correctness, q.v. *See also* atomic statement; controlled redundancy; integrity.
- (*Informal consistency, also known as “eventual” consistency*) Consistency in the foregoing formal sense isn't necessarily the same thing as consistency as conventionally understood in the real world (meaning consistency as understood outside the realm

of databases in particular). For example, suppose there are two items A and B in the database that, in the real world, are supposed to have the same value (they might both be the selling price for some given commodity, stored twice in the database to improve availability). If A and B in fact have different values at some given time, we might certainly say, informally, that there's an inconsistency in the database at that time. But that "inconsistency" is an inconsistency as far as the system is concerned if and only if the system has been told that A and B are supposed to be equal—i.e., if and only if " $A = B$ " has been declared as a formal constraint. If it hasn't, then (a) the fact that A and B are unequal at some time doesn't in itself constitute a consistency violation as far as the system is concerned, and (b) importantly, the system will nowhere rely on an assumption that A and B are equal. Thus, if all we want is for A and B to be equal "eventually"—i.e., if we're content for that requirement to be handled outside the database system by some application program—then all we have to do as far as the system is concerned is omit any declaration of " $A = B$ " as a formal constraint.

Examples (formal consistency): Suppose there's an integrity constraint on the suppliers-and-parts database to the effect that part weights must be positive. However, suppose the database were to show some part as having a negative weight (not possible, of course, if the DBMS is enforcing constraints properly). Then the database would be inconsistent (and a fortiori incorrect).

By way of a second example, suppose there's an integrity constraint in effect that says that every part must be supplied by at least

one supplier (i.e., the projections $P\{PNO\}$ and $SP\{PNO\}$ must be equal). However, suppose the database were to show part P7 as represented in relvar P but not in relvar SP (not possible, of course, if the DBMS is enforcing constraints properly). Again, then, the database would be inconsistent (and a fortiori incorrect).

consistent 1. (*Logic*) A set of predicates is consistent if and only if there exists at least one set of arguments that can be substituted for the parameters of those predicates in such a way that every resulting proposition evaluates to TRUE. 2. (*Database*) See consistency. Note, however, that consistency in the database sense is really nothing more than a special case of consistency in the sense of logic, where the predicates involved are simply the predicates that apply to the particular database in question.

Examples (logical consistency): Let the symbols x , y , and z denote integers. Then the predicates $x > y$ and $y > z$ form a consistent set, while the predicates $x > y$, $y > z$, and $z > x$ do not.

constant 1. (*Logic*) See individual constant. 2. (*Programming languages*) A value, especially one that's given a name that's not just a simple literal representation of the value as such; not to be confused with a literal, q.v.

Examples (second definition only): See relation constant.

constant reference (*Programming languages*) Syntactically, the name of a named constant (q.v.), used to denote the corresponding value. It can be regarded as an invocation of a read-only operator—and hence as an expression, q.v.—where the read-only operator in question is essentially “Return [the value of] the specified constant.” Like all expressions, therefore, it can appear wherever a literal of the

appropriate type can appear.

CONSTRAINT (*Without inheritance*) A **Tutorial D** keyword, used in connection with the definition of type constraints (q.v.) for scalar types. (It's also used in connection with database constraints, q.v.) Let T be such a type. Then the definition of type T must include at least one POSSREP specification (q.v.), and that POSSREP specification must include exactly one CONSTRAINT specification (either explicitly or implicitly; CONSTRAINT TRUE is assumed if nothing is specified explicitly).

Example: Let ELLIPSE be a scalar type. Then the corresponding type definition might look like this (irrelevant details omitted):

```
TYPE ELLIPSE
    POSSREP { A LENGTH , B LENGTH , CTR POINT
              CONSTRAINT A ≥ B } }
;
```

In other words, ellipses are such that they can possibly be represented by two lengths a and b and a point ctr , where a is the length of the ellipse's major semiaxis, b is the length of its minor semiaxis, ctr is its center, and $a \geq b$ (see the introduction to Part II of the dictionary for further discussion). *Note:* The user defined types LENGTH and POINT have already been defined (at least, let's assume so for the sake of the example). Also, the constraint $B > 0$ ought by rights to be specified as well but has been omitted to keep the example simple.

Now let e be a scalar value. Then e is of type ELLIPSE if and only if the following constraint—call it *ETC*—is satisfied: The value e can possibly be represented by a length a , a length b , and a point ctr , such that $a \geq b$. *ETC* here is the type constraint for type ELLIPSE. Note, therefore, that the CONSTRAINT specification as such doesn't

define the type constraint in its entirety, though it's often referred to informally as if it did.

constraint An integrity constraint, q.v. Usually understood to mean a database constraint specifically (i.e., not a type constraint), unless the context demands otherwise.

constraint inference The process of determining the constraints that hold in a given derived relvar or are satisfied by a given derived relation.

constructor function Term used in OO contexts for the operator that creates a new “instance” of a given object type (*see* instance, first definition; *see also* mutable object).

containment Generally, the relationship between a container and the things it contains; in particular, the relationship between a bag or set and its elements. The containment relationship is the inverse of the membership relationship, q.v. Containment is sometimes denoted by the symbol “ \ni ”; thus, the boolean expression $X \ni x$ —which is logically equivalent to both of the expressions $x \in X$ and $X \supseteq \{x\}$ —returns TRUE if and only if X does in fact contain x . *Contrast* inclusion.

Examples: A relation contains a heading and a body; a heading contains attributes; a body contains tuples; a tuple contains tuple components; a tuple component contains an attribute value; and so on.

contradiction A predicate whose every possible invocation is guaranteed to yield FALSE, regardless of what arguments are substituted for its parameters. *Note:* A contradiction in logic isn't quite the same thing as a contradiction in ordinary discourse. Loosely, we

might say a contradiction in ordinary discourse is something that implies that some proposition p and its negation $\text{NOT}(p)$ are both true; in logic, by contrast, it's anything that's "always false." Thus, propositions of the form $p \text{ AND } \text{NOT}(p)$ are certainly contradictions in the logical sense, but so are propositions of the form, e.g., $p \text{ AND FALSE}$, and so is the proposition consisting of just the literal FALSE itself. *Contrast* tautology.

Examples: Let $p1$ be the predicate (actually a proposition) $2+2 = 5$; let $p2$ be the predicate $x > x$, where x denotes an arbitrary integer; and let $p3$ be the predicate $(p) \text{ AND } (\text{NOT}(p))$, where p denotes an arbitrary predicate. Then $p1$, $p2$, and $p3$ are all contradictions. Note that a contradiction isn't necessarily a proposition, even though (like some propositions) it does unequivocally evaluate to FALSE . For example, $x > x$ isn't a proposition; rather, it's a predicate with exactly one parameter.

contrapositive The implicational predicates $\text{IF } (p) \text{ THEN } (q)$ and $\text{IF } (\text{NOT}(q)) \text{ THEN } (\text{NOT } (p))$ are contrapositives of each other. Any given implication and its contrapositive are logically equivalent.

Example: Consider the predicates—actually propositions—*If it's raining, then the streets are getting wet* and *If the streets aren't getting wet, then it isn't raining*. Each of these is the contrapositive of the other, and, clearly, each is logically equivalent to the other.

controlled redundancy Redundancy, q.v., is controlled if (a) it does exist (and the user is aware of it) but (b) it's guaranteed never to lead to any formal inconsistencies in the database. Uncontrolled redundancy, q.v., can be a problem, but controlled redundancy shouldn't be. As a general rule, databases shouldn't involve any uncontrolled redundancy.

Example: Suppose there's a business rule to the effect that all suppliers in the same city must have the same status. Of course, the sample value shown for relvar S in Fig. 1 doesn't satisfy this rule; however, it would do so if we changed the status for supplier S2 from 10 to 30, so let's suppose, just for the sake of the example, that this change has in fact been made. Then the fact that the status associated with Paris is 30 appears twice, and so there's some redundancy. (By contrast, if the status for supplier S2 were left at 10 instead of being changed to 30, then the database would be formally inconsistent, and hence incorrect.) So to say that the database involves some redundancy is to say that some specific business rule is supposed to hold, and hence that some specific integrity constraint is supposed to apply (though the converse is false, of course—not all integrity constraints have to do with controlling redundancy as such). For example, the “same status” constraint might be stated thus:

```
CONSTRAINT CRX COUNT ( S { CITY } ) = COUNT ( S { CITY ,  
STATUS } ) ;
```

Stating this constraint explicitly serves to inform the user that the redundancy exists; enforcing it serves to ensure that it won't lead to any formal inconsistencies, thereby guaranteeing that the redundancy in question is controlled. *Note:* Of course, enforcing such constraints should be done by the DBMS, not by the user. In some cases, it might even be possible for the DBMS to “propagate updates” appropriately in order to keep the data formally consistent (*see* compensatory action).

correct *See* correctness.

correctness (*Of a database*) The property of truly reflecting the state of affairs that exists in the real world (*see* the example under relvar

predicate for further discussion). *Contrast* consistency.

correlation name SQL term denoting (the SQL analog of) either a tuple calculus range variable, q.v., or the name of such a variable, as the context demands.

COUNT 1. Loosely, a synonym for cardinality, q.v. 2. An aggregate operator, q.v.

cover (*Of a set of FDs*) If $s1$ and $s2$ are sets of FDs, then $s2$ is a cover for $s1$ if and only if every FD implied by $s1$ is implied by those in $s2$ (*see Armstrong's axioms*). *Note*: Some writers use the term *cover* in a stronger sense, to mean a set of FDs that's equivalent to some given set (*see equivalence*).

cross join / cross product Terms sometimes used to mean cartesian product, q.v.

CWA *The Closed World Assumption*.

cyclic ordering Let s be a set. Loosely speaking, then, a cyclic ordering on s is like a linear ordering (q.v.) on s , except that it wraps around in such a way that what would otherwise be the first element is considered the immediate successor of what would otherwise be the last element. An example is provided by the hours of the day (0, 1, 2, ..., 23), where the the available values can be thought of as being arranged around the circumference of a clockface and every value thus has both a successor and a predecessor. Note that “<” and “>” both degenerate to “≠” in a cyclic ordering.



D Generic name (note the boldface) used to refer to any language that conforms to the principles laid down by *The Third Manifesto*. *Contrast Tutorial D*.

D_INSERT See disjoint INSERT.

D_UNION See disjoint union.

data (*Plural noun treated as singular*) An encoded representation of some set of propositions, assumed by convention to be true ones.

data definition operator An operator that either defines some database object, such as a base relvar or a view or a snapshot or a constraint, or deletes (“drops”) or updates such a definition; in other words, an operator that updates the catalog. *Note:* Dropping a definition effectively causes the corresponding object to be dropped as well, of course (at least as far as the user is concerned), and is usually described in such terms. For example, **Tutorial D** provides an operator called for psychological reasons DROP CONSTRAINT (not “drop constraint definition”).

Examples: See the definitions of relvars S, P, and SP. Other examples could be an operation to add an attribute to one of those relvars, or an operation to define a constraint on those relvars, or an operation to delete any of these definitions. *Note:* Strictly speaking, the first of the foregoing examples—“adding an attribute” to some relvar, say relvar S—has the effect of dropping the original relvar with that name and introducing a new one with the same name but an extended heading, at the same time preserving, somehow, the current information content of, and the constraints that apply to, the original relvar. Details of how this effect might be achieved are beyond the

scope of this dictionary.

data independence The ability to change either the physical or the logical design of a database without having to make corresponding changes in the way the database is perceived by users (thereby protecting investment in, among other things, existing user training and existing applications). The terms *physical data independence* and *logical data independence* refer to the two cases. Both involve having two sets of definitions and mappings between them, such that (a) if the physical design changes, physical data independence is preserved by changing the mapping between the physical design and the logical design, and (b) if the logical design changes, logical data independence is preserved by defining a mapping between the old logical design and the new one (or, equivalently, by changing the mapping between the logical design and the physical design). *Note:* If the logical design changes, the new logical design will consist of views of relvars in the old logical design—at least conceptually, if not in actual fact. Thus, logical data independence in particular implies the need to be able to update views, q.v.

data manipulation operator Loosely, an operator that isn't a data definition operator. However, the distinction isn't hard and fast; in fact, it's quite difficult to find an operator that doesn't, in the last analysis, "manipulate" data of some kind (unless it's a read-only operator, possibly; some writers might claim that update operators are the only ones that actually "manipulate" data). The term is really a hangover from prerelational systems, where it arguably made a little more sense than it does now; in relational contexts, it's probably better avoided.

data model 1. An abstract, self-contained, logical definition of the data structures, data operators, and so forth, that together make up the abstract machine with which users interact (*contrast* implementation).
2. A model of the persistent data of some particular enterprise (in other words, a conceptual or logical database design).

Examples: For the first definition, the most obvious example is of course the relational model itself. As for the second definition, any conceptual or logical database design will suffice as an example.

Note: There's a nice analogy that can help explain the difference between the two definitions, as follows: A data model in the first sense is like a programming language, whose constructs can be used to solve many specific problems but in and of themselves have no direct connection with any such specific problem; a data model in the second sense is like a specific program written in that language—it uses the facilities provided by the model, in the first sense of that term, to solve some specific problem. Note also that we can usefully characterize the distinction between a data model in that first sense and an implementation (q.v.) of that model by saying the model is what the user has to know, while the implementation is what the user doesn't have to know.

data modeling Term sometimes used—with reference to the second meaning of the term *data model* specifically, q.v., though never very precisely defined—to describe either the conceptual or the logical design process. *See* conceptual design; logical design.

data sublanguage A language that provides database support for one or more host languages, q.v., in which its statements can be embedded or from which they can be invoked.

Example: SQL is an obvious case in point; application programs that access an SQL database are usually written in some host language but invoke certain SQL operations, either in “embedded” form or via some kind of call level interface, to obtain the necessary database functionality.

Data Sublanguage ALPHA *See* ALPHA.

data type Same as type.

database Strictly, a database value, q.v.; more commonly used, in this dictionary in particular, to refer to what would more accurately be called a database variable, q.v. *Note:* We assume throughout this dictionary that databases are relational, barring explicit statements to the contrary. Be aware, however, that the term *database* is used in nonrelational contexts to mean a variety of other things—for example, a collection of data as physically stored. It’s also used, all too frequently, to mean a DBMS, but this particular usage is strongly deprecated. (If we call the DBMS a database, what do we call the database?)

database assignment An operation that assigns a database value to a database variable; in other words, any operation that updates the database. For further explanation, *see* database variable; multiple assignment.

database catalog *See* catalog.

database constraint 1. (“*A*” *database constraint*) Formally, any constraint that isn’t a type constraint; informally, any constraint that refers to two or more distinct relvars (also, and better, known as a

multirelvar constraint, q.v.). *Note:* These definitions aren't meant to be equivalent in any sense—they refer to two distinct concepts. 2. (*“The” database constraint*) The logical AND of all constraints, other than type constraints, that apply to a given database (*the* database constraint—sometimes called the *total* database constraint, for emphasis—for the database in question). *Note:* It follows from this second definition that one constraint that applies to every database is the degenerate (“default”) constraint TRUE. *See also* relvar constraint.

Examples: First, the key and foreign key constraints specified in the definition of the suppliers-and-parts database are all database constraints. Second, here are some more database constraints that might also apply to that database:

```
CONSTRAINT C1 IS_EMPTY ( S WHERE STATUS < 1 OR STATUS >
100 ) ;
/* status values must be in the range 1 to 100 inclusive
*/
```

```
CONSTRAINT C2 IS_EMPTY ( P WHERE CITY = 'London'
AND COLOR ≠ COLOR('Red') ) ;
/* parts in London must be red */
```

```
CONSTRAINT C3 IS_EMPTY
( ( S JOIN SP ) WHERE STATUS < 20 AND PNO =
PNO('P6') ) ;
/* no supplier with status less than 20 can supply part
P6 */
```

Here for interest is an alternative formulation of constraint C1 that makes use of the AND aggregate operator, q.v.:

```
CONSTRAINT C1 AND ( S , STATUS ≥ 1 AND STATUS ≤ 100 ) ;
/* status values must be in the range 1 to 100 inclusive
*/
```

This same style could also be used with constraints C2 and C3, of course.

Finally, suppose for the sake of the example that the specified key and foreign key constraints, together with constraints C1-C3 above, are the only database constraints that apply to the suppliers-and-parts database. Then the logical AND of all of them is “the” (total) database constraint for that database.

database design See logical database design; physical database design. *Note:* The unqualified term *database design*, or sometimes even just *design*, is usually taken to mean logical database design specifically, unless the context demands otherwise. See also conceptual design.

database management system The software system (abbreviated DBMS, plural DBMSs) that manages, and in particular handles all access to, some database or collection of databases. *Note:* A relational DBMS in particular can be thought of, or even defined, as an implementation of the relational model. *Contrast* database.

database programming language A programming language that includes fully integrated (“native”) database support. *Contrast* data sublanguage; host language.

Examples: **Tutorial D** might be regarded as a fully fledged database programming language, except that it currently includes no exception handling and no I/O support. A similar remark applies to SQL; SQL is widely thought of as just a data sublanguage, q.v., but with the introduction in the 1992 version of the standard (“SQL:1992”) of such features as local variables, exception handling, IF, CASE, WHILE, CALL, RETURN, and assignment (SET) statements, it too became a fully fledged database programming

language (except that, like **Tutorial D**, it currently includes no I/O facilities).

database relation The value of a given database relvar at a given time.

Database Relativity Principle See *Principle of Database Relativity*.

database relvar See relvar.

database statistics Metadata, typically kept in the catalog, that (among other things) might be helpful to the optimizer, q.v.

Examples: Relvar and attribute cardinalities; minimum, maximum, and average attribute values; attribute value frequencies; index selectivities; and so on.

database value Either the actual (i.e., current) or some possible “state” for some database; in other words, a collection of relations, those relations being actual or possible values for the applicable relvars. Abstractly, therefore, a database value can be thought of as a collection of propositions (assumed by convention to be true ones), those propositions being represented by the tuples in the applicable relations. *Contrast* database variable.

Example: The relations (i.e., relation values) shown in Fig. 1 constitute the “state” of the suppliers-and-parts database that happens to be current at this time. But if we were to look at that database at some different time, we would probably see a different state. In other words, the database is really a variable—a database variable, to be precise, meaning a variable whose values are database values (see database variable). Moreover, the tuples in the relations that are the values of relvars S, P, and SP at any given time represent propositions—propositions that are assumed to be true at that time—so, as the

foregoing definition indicates, the database at the time in question can be thought of, a trifle loosely, as a collection of true propositions.

database variable Loosely, a container for relvars; more accurately, a variable whose value at any given time is a database value. Strictly speaking, there's a logical difference, analogous to that between relation values and relation variables, between database values and database variables; thus, what we usually call a database is really a variable (typically a rather large one), and updating that database has the effect of replacing one value of that variable by another such value, where the values in question are database values and the variable in question is a database variable. More precisely still, a database is really a tuple variable, with one attribute (relation valued) for each relvar in the database in question. Note, therefore, that a database isn't really a set of relation variables, despite the fact that we usually think of it that way; rather, the relvars within any given database are really pseudovariables, q.v. All of that being said, however, we bow to traditional usage in this dictionary (most of the time, at any rate) and use the term *database* to refer to both database values and database variables, relying on context to make clear which is intended. *See also* database; database value.

Examples: For an example of a database value, see Fig. 1. As for the matter of a database really being a tuple variable, the suppliers-and-parts database in particular can be thought of as a tuple variable (SPDB, say) of the following tuple type:

```
TUPLE { S RELATION { SNO SNO , SNAME NAME ,
                    STATUS INTEGER , CITY
                    CHAR } ,
        P RELATION { PNO PNO , PNAME NAME , COLOR COLOR
        ,
```

```

        WEIGHT WEIGHT , CITY CHAR
    } ,
        SP RELATION { SNO SNO , PNO PNO , QTY QTY } }

```

It follows that, e.g., the following relational update on tuplevar SPDB
—

```
DELETE SP WHERE QTY < QTY(150) ;
```

—is really shorthand for the following tuple update:

```
UPDATE SPDB : { SP := SP WHERE NOT ( QTY < QTY(150) ) } ;
```

And this statement in turn is shorthand for the following tuple assignment:

```

SPDB := TUPLE { S      ( S FROM SPDB ) ,
                P      ( P FROM SPDB ) ,
                SP ( ( SP FROM SPDB ) WHERE NOT ( QTY <
QTY(150) ) ) } ;

```

As previously indicated, therefore, the names S, P, and SP really denote pseudovariabes, q.v. Note, however, that if we're to be able to write explicit database assignments as in the foregoing example, then databases—or database variables, rather—like SPDB will certainly have to have user visible names, which in **Tutorial D** they don't (at least, not as the language is currently defined). Thus, database assignments in **Tutorial D** have to be expressed in the form of relational assignments (in general, multiple assignments) to the relvar(s) within the database in question. For further explanation, see multiple assignment.

Incidentally, assuming the name SPDB is indeed user visible, then **Tutorial D** would certainly allow the foregoing tuple assignment to be

written in the form of an explicit tuple UPDATE statement as shown above, thus—

```
UPDATE SPDB : { SP := SP WHERE NOT ( QTY < QTY(150) ) } ;
```

—or even as follows:

```
UPDATE SPDB : { DELETE SP WHERE QTY < QTY(150) } ;
```

Finally, note that a database isn't just a set of relvars—rather, it's a set of relvars that are subject to a certain constraint (viz., the pertinent total database constraint). And it seems reasonable to require the database to be *fully connected* (and hence to form a coherent whole); in other words, it seems reasonable to require the total database constraint to be such that every relvar in the database is logically connected to every other (not necessarily directly, of course). The following definition is intended as an aid in formalizing this requirement. Let DB be a set of relvars, and let TC be the logical AND of all constraints that mention any relvar in DB . Assume without loss of generality that TC is in conjunctive normal form. Now let A and B be distinct relvars in DB . Then A and B are logically connected if and only if there exist relvars R_1, R_2, \dots, R_n in DB ($n > 0$, A and R_1 not necessarily distinct, R_n and B not necessarily distinct) such that there's at least one conjunct in TC that mentions both A and R_1 , at least one that mentions both R_1 and R_2 , ..., and at least one that mentions both R_n and B . *Note:* It should be clear that if a given database isn't fully connected in the foregoing sense, then the relvars it contains can be partitioned into two or more disjoint sets, each of which is fully connected.

DBMS Database Management System; plural DBMSs.

dbvar A database variable, q.v. The term isn't much used, though perhaps it should be.

DCO Domain check override, q.v.

De Morgan's Laws 1. (*Logic*) The negation of the disjunction of predicates p and q is logically equivalent to the conjunction of the negations of p and q ; the negation of the conjunction of predicates p and q is logically equivalent to the disjunction of the negations of p and q . 2. (*Set theory*) The complement of the union of sets $s1$ and $s2$ is equal to the intersection of the complements of $s1$ and $s2$; the complement of the intersection of sets $s1$ and $s2$ is equal to the union of the complements of $s1$ and $s2$.

Example (first definition only): The following identities are just a representation of the foregoing logic laws in symbolic form, but they might be a little easier to understand than the prose versions:

$$\text{NOT } ((p) \text{ OR } (q)) \equiv (\text{NOT } (p)) \text{ AND } (\text{NOT } (q))$$

$$\text{NOT } ((p) \text{ AND } (q)) \equiv (\text{NOT } (p)) \text{ OR } (\text{NOT } (q))$$

decidability (*Of a formal system*) A formal system is decidable if and only if, given an arbitrary sentence s , it can be determined mechanically whether s is a sentence of the system.

Examples: Propositional calculus is decidable; predicate calculus is not.

declared Term often used as a synonym for *defined* or *specified*.

declared possrep See possible representation. *Note:* The unqualified

term *possrep* is used almost invariably to refer to a declared possrep specifically.

declared type (*Without inheritance*) Type. *Note:* The following more specific definitions are logically correct but reduce, in the absence of support for inheritance, merely to saying that—as already indicated—the declared type of some item x is just the type of x , as this latter term is usually understood. 1. (*Of a constant, variable, attribute, or parameter*) The type specified when the constant, variable, attribute, or parameter in question is declared. 2. (*Of a read-only operator*) The type of the result, specified when the operator in question is declared (*see RETURNS*). 3. (*Of an expression*) The type of the outermost operator involved in the expression in question; in other words, the type of the operator whose execution is last in sequence (logically speaking, at any rate) in evaluating the expression in question.

Examples:

- First, the declared type of the literal 5 is INTEGER.
- Second, let variables E and R be defined as follows:

```
VAR E ELLIPSE ;
```

```
VAR R RECTANGLE ;
```

Then the declared types of these variables are (the presumably user defined types) ELLIPSE and RECTANGLE, respectively.

- Next, let ER be the relation type

```
RELATION { E ELLIPSE , R RECTANGLE }
```

Then the declared type of attribute R within relation type ER is

RECTANGLE.

- Finally, let the specification signature (q.v.) for operator MOVE be:

```
MOVE ( ELLIPSE , RECTANGLE ) RETURNS ELLIPSE
```

Then the declared type of that operator is ELLIPSE, and the declared types of the first and second parameter to that operator are ELLIPSE and RECTANGLE, respectively.

Note: Declared types are always known at compile time. Also, note in particular that x can have an empty declared type—*see* empty type—only if x is an attribute of some tuple type or some relation type.

decomposition Nonloss decomposition, q.v. (unless the context demands otherwise).

deductive axiom Term occasionally used to mean a rule of inference.

DEE Shorthand for TABLE_DEE.

default value Let A be an attribute of relvar R . Barring explicit rules to the contrary, then, a default value (default for short) can optionally be declared for A ; that value, a say, will then be used as the value for attribute A in any tuple for which no value is specified explicitly when the tuple in question is entered into relvar R .

Example: Suppose attribute STATUS of relvar S has default value 10. Then the following INSERT might be valid, syntactically speaking:

```
INSERT S RELATION { TUPLE { SNO      SNO('S6') ,
                           SNAME    NAME('Lopez') ,
                           CITY     'Madrid' } } ;
```

The relation that's actually inserted will look like this:

```
RELATION { TUPLE { SNO      SNO('S6') ,
                   SNAME    NAME('Lopez') ,
                   STATUS   10 ,
                   CITY     'Madrid' } }
```

Note: **Tutorial D** has no support for default values at the time of writing, and the foregoing INSERT on relvar S would thus currently not be valid in **Tutorial D**.

deferred checking Checking a database integrity constraint at some time (typically commit time) later than the time when an update is performed that might cause it to be violated. The relational model rejects such checking as logically flawed. *Contrast* immediate checking.

deferred constraint A database integrity constraint for which the checking is deferred (*see* deferred checking). The relational model rejects such constraints as logically flawed. *Contrast* immediate constraint.

degree The number n ($n \geq 0$) of attributes in a given heading, key, tuple, relation (etc.). *See also* arity.

Examples: The degrees of relvars S, P, and SP are four, five, and three, respectively; the degrees of the corresponding keys (one per relvar) are one, one, and two, respectively.

DELETE Loosely, an operator—shorthand for a certain relational assignment—that deletes specified tuples from a specified relvar. The syntax is:

```
DELETE R rx
```

Here R is a relvar reference (syntactically, just a relvar name) and rx is a relational expression (denoting some relation r of the same type as R), and the effect is to delete the tuples of r from R . In other words, the DELETE invocation just shown is shorthand for the following explicit assignment:

```
R := R MINUS rx
```

It follows that an attempt via DELETE to delete a tuple that's not present in the first place is not considered an error (*contrast* included DELETE).

Examples: The statement

```
DELETE SP RELATION
      { TUPLE { SNO SNO('S3') , PNO PNO('P2') , QTY
QTY(200) } ,
      TUPLE { SNO SNO('S1') , PNO PNO('P1') , QTY
QTY(400) } } ;
```

is shorthand for the following explicit assignment statement:

```
SP := SP MINUS RELATION
      { TUPLE { SNO SNO('S3') , PNO PNO('P2') ,
QTY QTY(200) } ,
      TUPLE { SNO SNO('S1') , PNO PNO('P1') ,
QTY QTY(400) } } ;
```

Given the sample values shown in Fig. 1, however, this assignment will delete just one tuple, not two (speaking a trifle loosely), because the tuple $\langle S1, P1, 400 \rangle$ doesn't currently appear in relvar SP.

By way of another example, the statement

```
DELETE S WHERE CITY = 'London' ;
```


is shorthand for the following relational assignment statement:

```
S := S MINUS ( S WHERE CITY = 'London' ) ;
```

Note: Strictly speaking, this second example is shorthand for a DELETE statement of the same form as the first example that might look like this:

```
DELETE S S WHERE CITY = 'London' ;
```

It's clear, however, that if as in this example the expression denoting the set of tuples to be deleted from relvar R takes the form R WHERE bx (where WHERE TRUE is assumed if no WHERE clause is specified explicitly), then (a) there's no point in mentioning R twice in concrete syntax, and (b) the question of attempting to delete tuples not present in the first place simply doesn't arise. Indeed, this common special case can be defined more simply as shorthand for the following:

```
R := R WHERE NOT ( bx )
```

For example, the second DELETE statement shown above is shorthand for:

```
S := S WHERE NOT ( CITY = 'London' ) ;
```

DELETE anomaly Same as deletion anomaly.

DELETE rule A rule specifying the action to be taken by the DBMS automatically—typically but not necessarily a compensatory action, q.v.—to ensure that DELETE operations on a given relvar don't violate any associated multivariable constraint, q.v. Foreign key DELETE rules (e.g., cascade) are an important special case. Note, however, that such automatic actions should occur, if and when

logically required, regardless of the concrete syntactic form in which the original DELETE request is expressed. For example, a DELETE request expressed as a pure relational assignment (using “:=”), q.v., should nevertheless cause the action specified by the pertinent DELETE rule to be performed—assuming, of course, that such a rule has been defined in the first place.

delete set *See* relational assignment.

deletion anomaly Term originally used (though never very precisely defined) to refer to the fact that DELETE operations on a relvar that’s subject to FD redundancy, q.v., can sometimes “delete too much.” E.g., suppose for the sake of the example that relvar *S* is subject to the FD {CITY} → {STATUS}. Of course, the sample value shown for that relvar in Fig. 1 doesn’t satisfy this FD; however, it would do so if we changed the status for supplier S2 from 10 to 30, so let’s suppose, just for the sake of the example, that this change has in fact been made (though actually it has no effect on the specific anomaly to be discussed). Here then is a deletion anomaly: If we delete the tuple for supplier S5 (the only supplier in Athens), we lose the fact that the status for Athens is 30. *Note:* A relvar that’s in BCNF, q.v., is guaranteed to be free of deletion anomalies in this “FD redundancy” sense.

The term *deletion anomaly* is also used in connection with relvars that are subject to JD redundancy, q.v.; in this case, however, the concept is more precisely defined. To be specific, let the JD \mathcal{J} hold in relvar *R*; then *R* suffers from a deletion anomaly with respect to \mathcal{J} if and only if there exists a relation *r* containing a tuple *t* such that (a) *r* satisfies \mathcal{J} and (b) the relation *r'* whose body is obtained from that of *r*

by removing t violates \mathcal{J} . *Note:* A relvar that's in ETNF, q.v., is guaranteed to be free of deletion anomalies in this "JD redundancy" sense.

Finally, this latter definition can be generalized, as follows: Relvar R suffers from a deletion anomaly if and only if (a) there exists a single-relvar constraint C on R and (b) there exists a relation r containing a tuple t such that r satisfies C and the relation r' whose body is obtained from that of r by removing t violates C . *Note:* A relvar that's in DK/NF, q.v., is guaranteed to be free of deletion anomalies in this generalized sense.

denormalization Replacing a set of relvars R_1, R_2, \dots, R_n by their join R , such that (a) for all i ($i = 1, 2, \dots, n$) the projection of R on the attributes of R_i at any given time is guaranteed to be equal to R_i at the time in question, and usually also such that (b) R is at a lower level of normalization than at least one of R_1, R_2, \dots, R_n . Denormalization is generally done for performance reasons; however, it typically has the effect of increasing redundancy, q.v., thereby increasing (a) the amount of integrity checking that has to be done, by the user or the system or both (thereby, incidentally, undermining the performance advantage that was the justification for doing the denormalization in the first place), or (b) the likelihood that certain update anomalies, q.v., will occur, or (c) both. It can also increase the complexity of certain queries. *Contrast* unnormalized. *Note:* Denormalization, at least to a level below ETNF, q.v., is always contraindicated from a logical point of view. Sometimes it can't reasonably be avoided, however, given the level of technology found in today's commercial products.

Example: A denormalization that might be applied to the suppliers-and-parts database would be to replace relvars S and SP by their join

(SSP, say). Relvars S and SP could then be derived by projecting relvar SSP on the attributes of S and the attributes of SP, respectively. Note that S and SP are both in 5NF (in fact, SP is in 6NF), while SSP isn't even in 2NF. Note too, however, that such a denormalization would be valid only if S and SP are both true projections of SSP—in other words, if and only if every supplier number appearing in relvar S at any given time also appears in relvar SP at that same time (and vice versa, of course)—which isn't guaranteed to be the case (and indeed isn't the case, given the sample values in Fig. 1).

dependant / dependent Terms used interchangeably to mean the set of attributes on the right side of an FD or MVD. *Contrast* determinant.

Example: In the FD $\{SNO, PNO\} \rightarrow \{QTY\}$, which holds in relvar SP, $\{QTY\}$ is the dependant and $\{SNO, PNO\}$ is the determinant.

dependence / dependency Terms used generically and interchangeably to mean an integrity constraint, typically but not necessarily an EQD or IND or JD or MVD or (especially) FD specifically. *See also* generalized dependency.

dependency preservation FD preservation, q.v.; occasionally, analogous preservation of some other kind of dependency.

dependency theory A body of theory, built on top of—i.e., relying on certain features of—the relational model and having to do with the formal properties of FDs, MVDs, and JDs among other things, that can be used to help with the process of logical database design (though not limited to that purpose alone).

dereferencing *See* referencing.

derived relation Loosely, a relation defined in terms of others. More precisely, let s be a set of relations. Then relation r is derived (or, perhaps more accurately, derivable) from the relations in s if and only if it doesn't itself appear in s but can be obtained by means of some relational expression from those that do. *Contrast* base relation. *Note:* The phrase "those that do" here is meant to be understood as referring to those relations that appear in s and those relations *only*. The reason is that *any* relation x can be "derived from the relations in s " by means of (e.g.) an expression of the form $r\{ \} \text{ JOIN } exp$, where (a) r denotes some nonempty relation in s and (b) exp is a relation literal whose value is precisely the desired relation x . In other words, the introduction of relation literals into such derivation expressions isn't allowed.

Example: Consider the expression $S \text{ JOIN } SP$. If the current values of relvars S and SP are s and sp , respectively, this expression defines the derived relation that is the join of s and sp .

derived relvar A relvar defined in terms of others by means of some relational expression; more specifically, a view or snapshot, q.v. (the only kinds of derived relvars supported at the time of writing). *Contrast* base relvar.

Examples: See snapshot; view.

descriptor Metadata that describes, e.g., a relvar or an attribute or a constraint.

design dilemma See relvar vs. type.

designator A name, possibly complex, used in a predicate to designate some specific object (as opposed to a parameter, which doesn't designate a specific object but instead stands for an arbitrary value of

the pertinent type). For example, in the predicate *The cardinality of relvar S is n*, the phrase “relvar S” is a designator, designating the relation that’s the current value of the suppliers relvar (by contrast, *n* is a parameter). Similarly, in the predicates—actually propositions—*Earth has a moon* and *Earth has a satellite*, “a moon” and “a satellite” are both designators (designating the same object, as it happens).

determinant The set of attributes on the left side of an FD or MVD. *Contrast* dependant.

Example: See dependant.

difference (*Without inheritance*) Let relations $r1$ and $r2$ be of the same type T . Then (and only then) the expression $r1$ MINUS $r2$ denotes the difference between $r1$ and $r2$ (in that order), and it returns the relation of type T with body the set of all tuples t such that t appears in $r1$ and not in $r2$. *Note:* The relational difference operator differs in certain respects from the mathematical or set theory operator of the same name, q.v.; in fact, it’s a special case of semidifference, q.v.

Example: The expression $S\{CITY\}$ MINUS $P\{CITY\}$ denotes the difference between (a) the relation that’s the projection on $\{CITY\}$ of the current value of relvar S and (b) the relation that’s the projection on $\{CITY\}$ of the current value of relvar P (in that order). That difference is a relation r of type $RELATION \{CITY \ CHAR\}$. Moreover, if the current values of relvars S and P are s and p , respectively, then the body of that relation r consists of all tuples of the form $\langle c \rangle$ that appear in $s\{CITY\}$ and not $p\{CITY\}$ —meaning c is a current supplier city that isn’t also a current part city. Note that the expression $S\{CITY\}$ MINUS $P\{CITY\}$ is logically equivalent to the expression $S\{CITY\}$ NOT MATCHING $P\{CITY\}$ —or to either of

the simpler expressions $S\{CITY\}$ NOT MATCHING P and $(S$ NOT MATCHING $P)$ $\{CITY\}$, come to that. (NOT MATCHING is **Tutorial D** syntax for the semidifference operator, q.v.)

difference (bag theory) *See* bag.

difference (set theory) The difference between two sets $s1$ and $s2$ (in that order), $s1 - s2$, is the set of all elements x such that x is an element of $s1$ and not an element of $s2$. *Note:* The difference $s1 - s2$ is also known as the relative complement (q.v.) of $s2$ with respect to $s1$.

direct image A somewhat unsophisticated style of implementation, found in most if not all of today's mainstream database products, in which what's physically stored is effectively just a direct image of what the user logically sees. In other words (and simplifying slightly), relvars are stored as physical files, and tuples and attributes are stored as records and fields within those files. *Contrast* TransRelationalTM Model.

direct proof *See* proof.

direct reasoning *See* *modus ponens*.

directed relationship A relationship (in the sense of the third definition of that term, q.v.) from one set to another.

discernibility Distinguishability. *See* indiscernibility; *see also* *Principle of Identity of Indiscernibles*.

discriminant *See* discriminated union (set theory).

discriminated union (set theory) Let $s1 = \{a1, a2, \dots, am\}$ and $s2 = \{b1, b2, \dots, bn\}$ be sets. Define sets $s1'$ and $s2'$ as follows:

$$s1' = \{ \langle a1, 1 \rangle, \langle a2, 1 \rangle, \dots, \langle am, 1 \rangle \}$$

$$s2' = \{ \langle b1, 2 \rangle, \langle b2, 2 \rangle, \dots, \langle bn, 2 \rangle \}$$

Observe that (a) $s1'$ and $s2'$ are sets of ordered pairs, one such pair for each element of $s1$ or $s2$, as applicable; (b) the first element of each such pair is an element from $s1$ or $s2$, as applicable; and (c) the second element of each such pair (the discriminant) is either 1 or 2, indicating which of $s1$ and $s2$ that first element is taken from. Then the discriminated union of $s1$ and $s2$ is the set theory union—the disjoint union, in fact—of $s1'$ and $s2'$.

Note: The foregoing definition is essentially the one given in the literature. However, it suffers from the weakness—surely unintended, and certainly undesirable—that the operator thus defined won't be commutative, unless there's some systematic way of assigning discriminants that guarantees that $s1$ and $s2$ are assigned discriminants 1 and 2, respectively, and not the other way around. Be that as it may, note too that the operator as here defined is dyadic; however, it would clearly be possible to define an n -adic version if desired.

Caveat: Be aware that discriminated union is sometimes referred to in the literature, rather unfortunately, as disjoint union. That is (to spell the point out), the discriminated union of $s1$ and $s2$ is sometimes referred to in the literature as the disjoint union of $s1$ and $s2$ as such, instead of as the disjoint union of $s1'$ and $s2'$.

disjoint 1. (*Of bags or sets*) Having no elements in common. 2. (*Of relations all of the same type*) Having no tuples in common. 3. (*Of types*) Having no value in common. *Note:* Distinct types are always disjoint, except possibly if inheritance is supported (see Part II of this

dictionary). *Contrast* overlapping.

disjoint INSERT Loosely, an operator, D_INSERT (shorthand for a certain relational assignment), that inserts specified tuples into a specified relvar, just so long as the tuples in question don't already appear in that relvar. The syntax is:

```
D_INSERT R rx
```

Here R is a relvar reference (syntactically, just a relvar name) and rx is a relational expression (denoting some relation r of the same type as R), and the effect is to insert the tuples of r into R , just so long as none of those tuples is already present in R . In other words, the D_INSERT invocation just shown is shorthand for the following explicit assignment:

```
R := R D_UNION rx
```

It follows that an attempt via D_INSERT to insert a tuple that's already present is an error (*contrast* INSERT).

Example: The statement

```
D_INSERT SP RELATION
      { TUPLE { SNO SNO('S3') , PNO PNO('P1') , QTY
QTY(150) } ,
      TUPLE { SNO SNO('S4') , PNO PNO('P5') , QTY
QTY(400) } } ;
```

is shorthand for the following relational assignment statement:

```
SP := SP D_UNION RELATION
      { TUPLE { SNO SNO('S3') , PNO PNO('P1') , QTY
QTY(150) } ,
      TUPLE { SNO SNO('S4') , PNO PNO('P5') , QTY
```


QTY(400) } } ;

Given the sample values shown in Fig. 1, this assignment will fail—more precisely, the implicit D_UNION invocation will fail—and no updating will be done, because the tuple <S4,P5,400> already appears in relvar SP.

disjoint union A variant on the relational union operator, q.v., in which the operand relations are required to be disjoint, q.v. In other words, if (a) relations $r1$ and $r2$ are of the same type T , and (b) they have no tuples in common, then (and only then) the expression $r1$ D_UNION $r2$ denotes the disjoint union of $r1$ and $r2$, and it reduces to $r1$ UNION $r2$. *Note:* An n -adic version of this operator could also be defined (and is so, in **Tutorial D**). Note too that a version of the operator could be defined to apply to sets in general as well as to relations in particular; in fact, elsewhere in this dictionary, such an operator is indeed assumed to exist. Note finally that disjoint union can also be used as an aggregate operator, q.v. *Contrast* discriminated union.

Example: Consider the expression S{CITY} D_UNION P{CITY}. If the current values of relvars S and P are as shown in Fig. 1, this expression will raise a run-time error, because some supplier cities are also part cities. If such were not the case, however, the expression would then be logically equivalent to S{CITY} UNION P{CITY}.

disjunct A predicate that's ORed with zero or more others.

disjunction 1. (*Dyadic case*) If and only if p and q are predicates, their disjunction (p) OR (q) is a predicate also. Let (ip) OR (iq) be an invocation of that predicate, where ip and iq are invocations of p and q ,

respectively. Then that invocation (ip) OR (iq) evaluates to TRUE if and only if at least one of ip and iq evaluates to TRUE. *Note:* The parentheses enclosing p and q in the predicate, and ip and iq in the invocation, might not be needed in practice. 2. (*N-adic case*) Let p_1, p_2, \dots, p_n ($n \geq 0$) be predicates; then (and only then) the disjunction OR $\{p_1, p_2, \dots, p_n\}$ is defined to be shorthand for the expression (p_1) OR (p_2) OR ... OR (p_n) . (Note that this expression evaluates to FALSE if $n = 0$, because FALSE is the identity with respect to OR.) *See also* existential quantifier.

disjunctive normal form A predicate is in disjunctive normal form, DNF, if and only if it's of the form (p_1) OR (p_2) OR ... OR (p_n) , where none of the disjuncts $(p_1), (p_2), \dots, (p_n)$ involves any ORs—more precisely, where each of p_1, p_2, \dots, p_n is a conjunction of literals (*see* literal, second definition).

DISTINCT *See* SELECT expression.

distinct type (SQL) *See* user defined type (SQL).

distributivity 1. (*Monadic over dyadic*) Let operators Op_1 and Op_2 be monadic and dyadic, respectively, and assume for definiteness that they're expressed in prefix and infix style, respectively. Then Op_1 distributes over Op_2 if and only if, for all x and y , $Op_1(x Op_2 y) = (Op_1(x)) Op_2 (Op_1(y))$. 2. (*Dyadic over dyadic*) Let operators Op_1 and Op_2 both be dyadic, and assume for definiteness that they're expressed in infix style. Then Op_1 distributes over Op_2 if and only if, for all x, y , and z , $x Op_1 (y Op_2 z) = (x Op_1 y) Op_2 (x Op_1 z)$.

Examples: 1. (*Monadic over dyadic*) In ordinary arithmetic, nonnegative square root (“ $\sqrt{\quad}$ ”) distributes over multiplication (“ $*$ ”),

because

$$\sqrt{\ (x * y) } = (\sqrt{ x }) * (\sqrt{ y })$$

for all x and y . (By contrast, “ $\sqrt{\ }$ ” does not distribute over “ $+$ ”.) In the same kind of way, restriction distributes over UNION, INTERSECT, and MINUS in relational algebra. 2. (*Dyadic over dyadic*) In ordinary arithmetic, multiplication (“ $*$ ”) distributes over addition (“ $+$ ”), because

$$x * (y + z) = (x * y) + (x * z)$$

for all x , y , and z . (By contrast, “ $+$ ” does not distribute over “ $*$ ”.) In the same kind of way, each of UNION and INTERSECT distributes over the other in relational algebra. Likewise, each of OR and AND distributes over the other in logic.

DIVIDEBY See Great Divide; Small Divide; *see also* division.

division Over the years several logically distinct relational division operators (i.e., operators that “divide” one relation by another) have been defined—so many, in fact, that it’s probably better not to use the term at all, or at least to state explicitly in any given context which particular operator is intended. Two such operators are defined in this dictionary, the Great Divide and the Small Divide, q.v. *Note: Tutorial D* does currently support both of these operators, but they’re in the process of being dropped, since (as is shown under Great Divide and Small Divide) their functionality can be obtained by a variety of other, and psychologically preferable, means.

DK/NF Domain-key normal form.

DNF Disjunctive normal form.

domain Type. *Note:* Earlier relational writings favored the term *domain*; more recent ones favor the term *type* instead.

domain (mathematics) See function; relation (mathematics).

domain calculus A form of relational calculus in which the range variables range over domains (i.e., types) instead of relations and thus denote values from those domains. *Note:* Domain calculus and tuple calculus, q.v., are expressively equivalent, because for every expression of the former there's a logically equivalent expression of the latter and vice versa. In fact, they're both relationally complete, q.v.

Example: Here's a domain calculus formulation of the query "Get supplier names for suppliers who supply at least one part" (see tuple calculus for a tuple calculus analog):

```
NX RANGES OVER { NAME } ;
SX RANGES OVER { SNO } ;
PX RANGES OVER { PNO } ;

{ NX } WHERE EXISTS SX ( EXISTS PX ( S { SNO SX , SNAME
NX } AND
SP { SNO SX , PNO PX
} ) )
```

In stilted English: "Get names NX where there exist a supplier number SX and a part number PX such that a tuple with supplier number SX and supplier name NX appears in relvar S and a tuple with the same supplier number SX and part number PX appears in relvar SP." As you can see, this particular example is somewhat clumsier than its tuple calculus counterpart (see tuple calculus), but there are cases where the reverse is true.

domain check override An ad hoc and logically flawed—and

therefore deprecated—mechanism for performing comparisons between values of different types. (It's flawed because it's based on a confusion over the logical difference between types and representations.)

domain constraint *See* domain-key normal form.

domain-key normal form The “ultimate” normal form, in the following special (and limited) sense: Relvar R is in domain-key normal form (DK/NF) if and only if every single-relvar constraint that holds in R is implied by the domain and key constraints that hold in R , where (a) the phrase “every single-relvar constraint” includes but isn't limited to FDs and JDs, q.v., in particular, and (b) a “domain constraint” in this context is a constraint to the effect that values of a given attribute are taken from some prescribed set of values—for example, a constraint on relvar S to the effect that STATUS values must be in the range 1-100 inclusive. Every DK/NF relvar is in 5NF, though not necessarily in 6NF. *Note:* A relvar in DK/NF is guaranteed to be free of insertion and deletion anomalies as defined elsewhere in this dictionary; however, the concept is mainly of academic interest, because relvars can easily be fully normalized—i.e., in 5NF or even 6NF—and still not be in DK/NF. In other words, DK/NF isn't always achievable. What's more, the question “Exactly when can it be achieved?” has still not been answered.

Example: As noted under Boyce/Codd normal form, with the normal forms it's often more instructive to show a counterexample rather than an example per se. Suppose, therefore, that shipments are subject to a constraint to the effect that odd numbered parts can be supplied only by odd numbered suppliers and even numbered parts

only by even numbered suppliers. (This example is very contrived, of course, but it suffices for the purpose at hand.) Then that constraint is clearly not implied by the domain and key constraints that hold in relvar SP, and so the relvar isn't in DK/NF; yet it's certainly in 6NF.

domain of discourse Same as universe of discourse.

domain relational calculus Domain calculus, q.v.

dot qualification In tuple calculus and languages based on it, a dot qualified name is an expression of the form $R.A$, where R is the name of a range variable and A is the name of an attribute of the relation r over which R ranges. Such an expression serves as an attribute reference, q.v.; it denotes the value of attribute A (or possibly attribute A as such) within the particular tuple of r to which R currently refers. Dot qualification is used for disambiguation purposes in tuple calculus—also in SQL—but not in domain calculus or relational algebra (these latter use attribute (re)naming and/or name scoping to achieve an equivalent effect). *Note:* Since it's directly based on relational algebra, **Tutorial D** in particular has no dot qualification.

Example: The following tuple calculus formulation of the query “Get suppliers who supply at least one part” makes use of two dot qualified names, SPX.SNO and SX.SNO:

```
SX RANGES OVER { S } ;  
SPX RANGES OVER { SP } ;  
  
{ SX } WHERE EXISTS SPX ( SPX.SNO = SX.SNO )
```

Here for comparison is a relational algebra (**Tutorial D**) formulation of the same query:

The “matching” here is done on the basis of attribute SNO (since that attribute is the only one common to relvars S and SP). *See* semijoin.

double arrow *See* multivalued dependency.

double arrow out of An MVD of the form $A \twoheadrightarrow B$ is sometimes referred to, informally, as “a double arrow out of A ” (or, even more informally, as a double arrow out of the attributes constituting A —especially if A is of degree one).

double bang Same as bang bang.

double negation (*Logic*) Same as involution.

double underlining A convention used in pictures like Fig. 1 for indicating or highlighting primary key attributes. To elaborate, there are two cases to consider: (a) The relation depicted is a sample value for some relvar R (this case is illustrated by Fig. 1); (b) the relation depicted is a sample value for some relational expression rx , where rx is something other than a simple relvar reference (i.e., just the pertinent relvar name, syntactically speaking). In the first case, double underlining simply indicates that a primary key PK has been declared for R and the pertinent attribute is part of PK . In the second case, rx can be thought of as the defining expression for some temporary relvar R (equivalently, it can be thought of as a view defining expression and R as the corresponding view); then double underlining indicates that a primary key PK could in principle be declared for R and the pertinent attribute is part of PK .

DRC Domain relational calculus.

drop See data definition operator.

dual 1. (*Logic*) The duals of AND, OR, TRUE, and FALSE are OR, AND, FALSE, and TRUE, respectively (NOT is its own dual). More generally, let exp be a logical expression involving no connectives other than NOT, AND, and OR, and let exp' be obtained from exp by replacing every occurrence of AND, OR, TRUE, and FALSE by its dual; then exp and exp' are duals of each other. *Note:* Since every logical expression is logically equivalent to one involving no connectives other than NOT, AND, and OR, it follows that every logical expression has a dual. Note too that logical expressions $exp1$ and $exp2$ are logically equivalent if and only if their duals $exp1'$ and $exp2'$ are logically equivalent. 2. (*Set theory*) The duals of intersection, union, the universal set, and the empty set are union, intersection, the empty set, and the universal set, respectively (complement is its own dual). More generally, let exp be a set theory expression involving no operators other than complement, intersection, and union, and let exp' be obtained from exp by replacing every occurrence of intersection, union, the universal set, and the empty set by its dual; then exp and exp' are duals of each other. *Note:* Since every set theory expression is logically equivalent to one involving no operators other than complement, intersection, and union, it follows that every set theory expression has a dual. Note too that set theory expressions $exp1$ and $exp2$ are logically equivalent if and only if their duals $exp1'$ and $exp2'$ are logically equivalent. See also *Duality Principle*.

dual mode principle The principle that any relational operation that can be invoked interactively can also be invoked from an application program and vice versa.

Duality Principle 1. (*Logic*) Let exp be a tautology of the form $p \equiv q$, and let exp' be obtained from exp by replacing every appearance of AND, OR, TRUE, and FALSE by its dual, q.v.; then exp' is a tautology. 2. (*Set theory*) Let exp be a theorem of the form $p = q$, and let exp' be obtained from exp by replacing every appearance of intersection, union, the universal set, and the empty set by its dual, q.v.; then exp' is a theorem.

Examples: Each of De Morgan's Laws (q.v.) is a tautology, and each is the dual of the other.

DUM Shorthand for TABLE_DUM.

duplicate Let a and a' be appearances (q.v.) in some context of values v and v' , respectively. Then a and a' are duplicates of each other if and only if v and v' are equal (in other words, if and only if v and v' are the very same value). *Note:* It should be clear from this definition that the well known dictum to the effect that no relation ever contains duplicate tuples really means no relation ever contains duplicate *appearances* of the *same* tuple—though we stay with the less precise formulation elsewhere in this dictionary (for the most part, at any rate), for reasons of familiarity. Observe that since (a) relations never contain duplicate tuples and (b) every relational operation yields a relation, the DBMS is required to eliminate redundant duplicate tuples—meaning, more precisely, redundant appearances of the same tuple—from the result of any such operation, if such duplicates would otherwise appear (i.e., as artifacts of the algorithm used to implement the operation in question).

Examples (duplicate elimination): Given the sample values shown in Fig. 1, the projection on {CITY} of the current value of relvar S has

cardinality three, not five; similarly, the union of (a) the projection on {CITY} of the current value of relvar S and (b) the projection on {CITY} of the current value of relvar P has cardinality four, not eleven. (Note that projection and union are the only relational operators defined in this part of the dictionary for which duplicate elimination is a consideration.)

duplicate elimination Term used ubiquitously to mean what would more accurately be called duplication elimination. *See* duplicate.

dyadic Of an operator, having exactly two operands; of a predicate, being defined in terms of exactly two parameters. *Contrast* binary.



E/R Entity/relationship.

E/R diagram *See* entity/relationship diagram.

E/R model *See* entity/relationship model.

E/R modeling *See* entity/relationship modeling.

E-relation / E-relvar *See* RM/T.

EKNF Elementary key normal form.

element *See* bag; set.

elementary key Let K be a subset of the heading of relvar R . Then K is an elementary key for, or of, relvar R if and only if (a) it's a key for R and (b) there exists some subset A of the heading of R such that the FD

$K \rightarrow A$ is nontrivial and irreducible. See elementary key normal form.

Examples: 1. Suppose relvar SP has, instead of the usual QTY attribute, an attribute CITY, representing the city of the applicable supplier. The sole key of this revised version of SP is still {SNO,PNO}; however, it's not an elementary key, because the only nontrivial FD that holds with that key as determinant is {SNO,PNO} \rightarrow {CITY}, which isn't irreducible (because the FD {SNO} \rightarrow {CITY} also holds). 2. Suppose now that relvar SP has an attribute CITY (supplier city) as well as—not instead of—the usual QTY attribute. The sole key is still {SNO,PNO}. Now, however, that key is elementary, because the FD {SNO,PNO} \rightarrow {QTY}, which certainly holds, is both nontrivial and irreducible.

elementary key normal form Relvar R is in elementary key normal form (EKNF) if and only if, for every nontrivial FD $X \rightarrow Y$ that holds in R , (a) X is a superkey or (b) Y is a subkey of some elementary key (q.v.). Every EKNF relvar is in 3NF.

Example: As noted under Boyce/Codd normal form, with the normal forms it's often more instructive to show a counterexample rather than an example per se. Suppose, therefore, that relvar SP has, instead of the usual QTY attribute, an attribute SNAME, representing the name of the applicable supplier; suppose also that supplier names are necessarily unique (i.e., no two distinct suppliers ever have the same name at the same time). Then this revised version of SP has two keys, {SNO,PNO} and {SNAME,PNO}. However, these keys aren't elementary keys, because the only nontrivial FDs that hold with one of these keys as determinant are {SNO,PNO} \rightarrow {SNAME} and {SNAME,PNO} \rightarrow {SNO}, and these FDs are both reducible (in both cases PNO can be dropped from the determinant without loss). So the

relvar is subject to two nontrivial FDs, $\{SNO\} \rightarrow \{SNAME\}$ and $\{SNAME\} \rightarrow \{SNO\}$, in which the determinant isn't a superkey and the dependant isn't a subkey of an elementary key. So this version of relvar SP isn't in EKNF (though it is in 3NF).

embedded dependency A dependency that's satisfied by some projection of some relation but not by the relation itself, or—more important—a dependency that holds in some projection of some relvar but not in the relvar itself. *Note:* If F is an FD that holds in some projection of relvar R , then F certainly holds in R itself; thus, embedded dependencies aren't FDs, by definition.

Example: Consider relvar CTXD, with attributes C (course), T (teacher), X (textbook), and D (days) and predicate *Teacher T spends D days with textbook X on course C*. Let the sole key for that relvar be $\{C, T, X\}$. Assume also that for a given course, the set of teachers and the set of textbooks are quite independent of each other. Then CTXD is in 6NF—it can't be nonloss decomposed at all, other than trivially—but its projection on $\{C, T, X\}$ is subject to the embedded multivalued dependencies $\{C\} \twoheadrightarrow \{T\}$ and $\{C\} \twoheadrightarrow \{X\}$.

empty (*Of a bag or set*) Having no elements.

empty bag The bag with no elements (note that there's exactly one such); written $\{ \}$ or \emptyset . *Note:* Of course, the empty bag and the empty set, q.v., are logically indistinguishable—though if B and S are variables of some bag type and some set type, respectively, they won't “compare equal” even if their values are the empty bag and the empty set, respectively. (In fact, of course, such a comparison wouldn't even be syntactically legal, precisely because the comparands are of different types.)

empty database 1. A database containing only empty relvars. 2. A database containing no relvars at all. (Of course, the second definition here is just a special case—but an important special case—of the first.)

empty foreign key A foreign key of degree zero. Note that the corresponding target key will necessarily be of degree zero also (*see* empty key), and the pertinent referential constraint—from relvar $R2$ to relvar $R1$, say—will therefore be satisfied if and only if either $R1$ is nonempty or $R2$ is empty or both. *Note:* Either or both of $R1$ and $R2$ here might in fact be “hypothetical views,” in the sense of that term explained under, e.g., foreign key constraint.

empty heading The heading of degree zero (note that there’s exactly one such).

empty key A key of degree zero. Note that a relvar with an empty key can’t have any other keys apart from the empty one, thanks to the key irreducibility requirement, q.v. Note too that such a relvar can’t contain more than one tuple, thanks to the key uniqueness requirement, q.v. Declaring relvar R to have an empty key is thus a convenient way of stating a cardinality constraint, q.v., to the effect that R must never contain more than one tuple.

empty possrep A possrep with no components. If type T has an empty possrep, then (a) T can’t have any possreps apart from that empty one; (b) the associated set of THE_ operators is also empty, a fortiori; (c) T has exactly one value, v say; (d) T has exactly one associated—and necessarily niladic—selector operator, S say; and (e) the sole legal invocation of S , viz., $S()$, returns that value v .

empty range *See* existential quantifier; UNIQUE; universal quantifier.

empty relation Slightly imprecise term used to refer to a relation with an empty body. Given a relation type T , there's exactly one empty relation of that type: viz., the relation of type T that contains no tuples at all. Note that two relations can both be empty and yet not equal; to be specific, they'll be equal if and only if they're of the same type. *Contrast* universal relation.

Example: Suppose relvars S and P are both currently empty; that is, their current values s and p are both empty relations. Then s and p aren't equal, even though their bodies are equal, precisely because they're of different types (equivalently, because their headings aren't equal).

empty relvar A relvar whose current value is an empty relation.

empty restriction A restriction of a given relation r that contains no tuples (i.e., is equal to r WHERE FALSE); especially, a restriction of the form r WHERE c , where c is a contradiction, q.v. *Note:* The term is also used of a relvar.

Examples: Given the sample values in Fig. 1, the expressions S WHERE STATUS = 25 and S WHERE STATUS \neq STATUS both denote empty restrictions (the second necessarily so, because STATUS \neq STATUS is a contradiction).

empty set The set with no elements (note that there's exactly one such); written $\{ \}$ or \emptyset . The empty set is a subset of every set. All theorems, properties, definitions, etc., that apply to sets in general apply to the empty set in particular; for example, relation headings and bodies are both defined to be sets (of attributes and tuples, respectively), and so each is allowed to be the empty set in particular. *See* nullology.

empty tuple The tuple of degree zero (note that there's exactly one such).

empty type (*Without inheritance*) A type with no values. This concept is of crucial importance if type inheritance is supported—see Part II of this dictionary—but perhaps not otherwise.

encapsulated Scalar—though it's not always obvious from the literature (especially the OO literature) that *scalar* is indeed what the term means. For example, here's a typical definition (it's taken from James Martin and James J. Odell, *Object-Oriented Methods: A Foundation*, Prentice-Hall, 1998):

[Encapsulation is a] protective encasement that permits access to an object's data only via specifically assigned operations. With encapsulation, an object's interface is stated in terms of its permissible operations. All other implementation details about the object are hidden from the user. This is why the term encapsulation is often used interchangeably with *information hiding*.

And here's another (this one is from Douglas K. Barry, *The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases*, Wiley Publishing, 1996):

[Encapsulation is] the separation of the external aspects of an object from the object's internal implementation.

As you can see, the emphasis in both of these definitions is on what the conventional database literature would call data independence, q.v. (physical data independence, to be specific). But such data independence is intrinsic to the very notion of scalar data, so it's not clear why there's so much emphasis—at least in some circles—on the

concept of encapsulation as such.

Note: The term *encapsulated* is also used, especially in OO contexts, to refer to the physical bundling, or packaging together, of code and data (or operator definitions and data representation definitions, to be a little more precise about the matter). But to use the term in this way is to mix model and implementation considerations; the user shouldn't care, and shouldn't need to care, whether code and data are physically bundled together or not.

entity A thing. *Note:* It's frequently suggested that there should be a one to one correspondence between "entities of interest" and tuples in base relvars. The suggestion is hard to sustain, however, given that the term *entities of interest* has no precise definition. (Of course, the same is true of the term *entity* itself, come to that.)

entity integrity A rule, articulated in certain of Codd's writings, to the effect that attributes of primary keys in base relvars don't allow nulls. However, since (a) relvars, base or otherwise, don't necessarily have to have primary keys at all (*see* primary key) and (b) rules that apply to base relvars but not to other kinds are more than a little suspect anyway (because they violate *The Principle of Interchangeability*, q.v.), the entity integrity rule could be, and in fact has been, dropped without serious loss. We mention it here mainly for historical reasons. In any case, it refers to a concept, null, that is totally incompatible with the relational model; it would thus require major revision anyway before any suggestion that it be kept could be seriously entertained.

entity modeling *See* semantic modeling.

entity/relationship diagram A picture intended to explicate the

logical or conceptual design of a given database at a level of abstraction in which many details—in particular, details of the underlying types and almost all integrity constraints—are omitted. (The most important constraints not omitted are, typically, key and foreign key constraints.) Such pictures can be helpful in connection with the design process, but they're certainly not, as some people seem to think, a total solution to the design problem.

entity/relationship model A set of conventions for drawing entity/relationship diagrams, q.v. *Note:* Actually, there's no consensus on exactly what the entity/relationship model consists of—different writers define it in different ways. Thus, the term is best thought of as referring to a family of similar but distinct schemes.

entity/relationship modeling Using some form of entity/relationship model, q.v., as a tool to assist in the database design process.

enumerated type A type whose definition specifies the legal values of the type by simply enumerating or listing them.

Example: Here's a **Tutorial D** definition for a type called WEEKDAY (irrelevant details omitted):

```
TYPE WEEKDAY POSSREP
  { WD CHAR CONSTRAINT WD ∈
    { 'Sun' , 'Mon' , 'Tue' , 'Wed' , 'Thu' , 'Fri' ,
      'Sat' } } ;
```

EQ Same as EQUIV.

EQD Equality dependency.

equality (*Without inheritance*) A truth valued or logical operator (“=”).

Two values are equal if and only if they're the very same value; that is, the comparison $v1 = v2$ (where $v1$ and $v2$ are values) evaluates to TRUE if and only if $v1$ and $v2$ are in fact the very same value. For example, the integer 3 is equal to the integer 3 and not the integer 4 or any other integer (and not to anything else either, for that matter). Note that it follows from this definition that if $v1 = v2$ evaluates to TRUE, then $v1$ and $v2$ must be of the same type T . It also follows that if (a) there exists an operator Op (other than “=” itself) with a parameter P such that (b) two successful invocations of Op —invocations that are identical in all respects except that the argument corresponding to P is the value $v1$ in one invocation and the value $v2$ in the other—are distinguishable in their effect, then (c) $v1 = v2$ must evaluate to FALSE. *Note:* The equality operator (which is defined for every type, necessarily) is also known, especially in logic contexts, as *identity*. See also bag membership; duplicate; equivalence; identity; overloading; set membership; relation equality; tuple equality; and elsewhere.

equality dependency An expression of the form $rx = ry$, where rx and ry are relational expressions of the same type; it can be read as “The relations denoted by rx and ry are equal” (in other words, they're one and the same relation). An important special case is as follows: Let $R1$ and $R2$ be relvars, not necessarily distinct. Let $X1$ and $X2$ be subsets of the heading of $R1$ and the heading of $R2$, respectively, such that there exists a possibly empty set of attribute renamings on $R1$ that maps $X1$ into $X1'$, say, where $X1'$ and $X2$ contain exactly the same attributes (in other words, $X1'$ and $X2$ are in fact one and the same). Further, let $R1$ and $R2$ be subject to the constraint that, at all times, (a) every tuple $t1$ in $R1$ has an $X1'$ value that's the $X2$ value for at least one tuple $t2$ in $R2$

at the time in question, and (b) every tuple t_2 in R_2 has an X_2 value that's the X_1' value for at least one tuple t_1 in R_1 at the time in question. Then that constraint is an equality dependency (EQD for short)—very loosely, an EQD “on” R_1 and R_2 . *Note:* EQDs shouldn't be confused with equality generating dependencies, q.v.; in fact, they're a special case of inclusion dependencies, q.v.

Example: Suppose the suppliers-and-parts database is subject to a constraint to the effect that every part must be supplied by at least one supplier:

```
CONSTRAINT EQDX P { PNO } = SP { PNO } ;
/* every part must be supplied */
```

This constraint is an EQD “on” relvars P and SP (and it's satisfied by the sample values shown in Fig. 1).

Note: The comparands in an EQD can be specified by means of arbitrarily complex expressions. As a consequence, all possible database constraints (in the more formal sense of that term, q.v.) can in fact be expressed as equality dependencies! To elaborate, let C be such a constraint; let s be a set of tuples (all of the same type) that together violate C ; let r be the relation whose body is s ; and let rx be a relational expression denoting r . Then r must be empty, and C must thus conceptually be of the form $IS_EMPTY(rx)$. But $IS_EMPTY(rx)$ is logically equivalent to each of the following expressions—

```
 $rx \{ \} = TABLE\_DUM$ 
```

```
 $rx = rx \text{ WHERE } FALSE$ 
```

—and each of these expressions is an EQD. (The subexpression $rx\{ \}$ in the first of these equivalent expressions denotes the projection of

relation r on the empty set of attributes. Such a projection evaluates, necessarily, either to TABLE_DEE, if r is nonempty, or to TABLE_DUM otherwise.)

equality generating dependency An expression of the form $\{t1, t2, \dots, tn\} / a = b$; it can be read as “If tuples $t1, t2, \dots, tn$ appear (in some given relvar at some given time), then a and b must be equal.” Tuples $t1, t2, \dots, tn$ are the premises of the dependency and $a = b$ is the conclusion. Observe that FDs in particular are equality generating dependencies—not the only possible kind, but the only kind considered in this dictionary—because they take the basic form “If certain tuples appear (in some given relvar at some given time), then certain attributes within those tuples must have equal values.” *Note:* Equality generating dependencies should not be confused with equality dependencies, q.v. *Contrast* tuple generating dependency.

equijoin A theta join, q.v., in which theta is “=”.

Example: The following expression represents the equijoin of suppliers and parts on cities:

```
( ( S RENAME { CITY AS SC } )  
  TIMES  
  ( P RENAME { CITY AS PC } ) ) WHERE SC = PC
```

Observe the need to rename at least one of the two CITY attributes before we can apply the operator TIMES, q.v. (the example renames them both, for symmetry).

Note: The result of an equijoin necessarily has two attributes—SC and PC, in the example—whose values are equal in every tuple. If one of those two attributes is projected away and the other then renamed back to CITY, the result is the natural join (q.v.) of suppliers and parts

(so natural join can be defined in terms of cartesian product, restriction, projection, and renaming).

EQUIV 1. A connective, q.v. 2. An aggregate operator, q.v. *Note:* In practice, the equivalence connective is often represented by the symbol “ \equiv ”. For further explanation, *see* equivalence (sixth and seventh definitions). *Contrast* XOR.

equivalence 1. (*General*) Let x and y be elements of some set, and let that set be partitioned into a set of equivalence classes, q.v. Then x and y are equivalent (in symbols, $x \equiv y$) if and only if they're members of the same equivalence class. 2. (*Logical*) *See* logical equivalence. 3. (*Truth functional*) *See* truth functional equivalence. 4. (*Information*) *See* information equivalence. 5. (*Sets of FDs*) Two sets of FDs are equivalent if and only if each is a cover for the other. *Note:* Any given set of FDs always has at least one equivalent set that's irreducible. *See* irreducible, fourth definition. 6. (*Connective, dyadic case*) If and only if p and q are predicates, the equivalence (p) EQUIV (q) is a predicate also. Let (ip) EQUIV (iq) be an invocation of that predicate, where ip and iq are invocations of p and q , respectively. Then that invocation (ip) EQUIV (iq) evaluates to TRUE if and only if ip and iq both evaluate to TRUE or both evaluate to FALSE. In other words, (p) EQUIV (q) is equivalent to $((p)$ IMPLIES $(q))$ AND $((q)$ IMPLIES $(p))$. It's also equivalent to NOT $((p)$ XOR $(q))$. *Note:* The parentheses enclosing p and q in the predicate, and ip and iq in the invocation, might not be needed in practice. For further discussion, *see* truth functional equivalence; *contrast* logical equivalence. 7. (*Connective, n-adic case*) Let p_1, p_2, \dots, p_n ($n \geq 0$) be predicates. Then (and only then) the equivalence EQUIV $\{p_1, p_2, \dots, p_n\}$ is a predicate also; and if ip_1, ip_2, \dots, ip_n are invocations of p_1, p_2, \dots, p_n , respectively, then the invocation

$\text{EQUIV } \{ip1, ip2, \dots, ipn\}$ returns TRUE if and only if exactly m of the invocations $ip1, ip2, \dots, ipn$ return FALSE, where m is even. *Caveat:* This definition is motivated by a desire to preserve associativity; to be specific, it has the property that the expressions $\text{EQUIV } \{p1, \text{EQUIV } \{p2, p3\}\}$, $\text{EQUIV } \{\text{EQUIV } \{p1, p2\}, p3\}$, and $\text{EQUIV } \{p1, p2, p3\}$ are all truth functionally equivalent. On the other hand, it also has the property that $\text{EQUIV } \{p1, p2, p3\}$ and $\text{NOT } (\text{XOR } \{p1, p2, p3\})$, as this latter expression is defined in this dictionary, are *not* truth functionally equivalent. It would be possible to come up with a different and possibly more intuitive definition, according to which the invocation $\text{EQUIV } \{ip1, ip2, \dots, ipn\}$ returns TRUE if and only if all n of the invocations $ip1, ip2, \dots, ipn$ return the same truth value. However, the two definitions are themselves clearly not equivalent (!); in other words, they define two logically distinct operators (though they both reduce to the simple dyadic case if $n = 2$, as is surely to be desired).

equivalence class A subset s' of some given set s with the property that the elements of s' are (a) all equivalent to one another, under some stated definition of equivalence, and (b) not equivalent to any other element of s , under that same definition of equivalence. (Note the relevance of this concept to the relational grouping operation, q.v.; *see also* image relation.) Observe that (a) equivalence classes are pairwise disjoint, and (b) together, they partition the values in the given set s . For a more formal definition, *see* equivalence relation. *See also* canonical form.

Examples: 1. Let s be the set of all positive integers, and define positive integers x and y to be equivalent if and only if they have the same number of digits in conventional decimal notation (no leading zeros). Then the subset of s containing all one-digit integers is an

equivalence class under this definition of equivalence; so too are the subsets consisting of all two-digit integers, all three-digit integers, and so on. 2. Consider the set of parts currently represented by relvar P . Define two such parts to be equivalent if and only if they're of the same color. Then the set of all red parts currently represented in P is an equivalence class under this definition of equivalence; so too is the set of all blue parts, and so is the set of all yellow parts, and so on. 3. Consider the set of tuples in the current value of relvar SP . Define two such tuples to be equivalent if and only if they contain the same SNO value. Then the set of all such tuples for supplier number $S1$ is an equivalence class under this definition of equivalence; so too is the set of all such tuples for supplier $S2$, and so is the set of all such tuples for supplier $S3$, and so on.

equivalence relation Let r be a binary relation. Then r is an equivalence relation if and only if it's reflexive (q.v.), symmetric (q.v.), and transitive (q.v.). Further, let x be a value such that the tuple $\langle x, y \rangle$ appears in r for some y . Given that value x , then, the set of all such corresponding values y is an equivalence class with respect to r —namely, that specific equivalence class that corresponds to the given value x (*see* equivalence class). Observe that if r_y is the set of all y values appearing in r , then every value in r_y appears in exactly one equivalence class with respect to r —in other words, as noted under equivalence class, equivalence classes are pairwise disjoint, and together they partition the pertinent set of values.

essential tuple Tuple t is essential in relation r if and only if it's not redundant in r . *Contrast* redundant tuple.

essential tuple normal form Relvar R is in essential tuple normal

form (ETNF) if and only if every relation r that's a legitimate value for R is such that every tuple is essential in r —equivalently, if and only if (a) R is in BCNF and (b) for every JD \mathcal{f} that holds in R , at least one component of \mathcal{f} is a superkey for R . Every ETNF relvar is in 4NF. Also, it's easy to see that if relvar R is in BCNF and has at least one simple key (q.v.), then it's in ETNF.

Example: As noted under Boyce/Codd normal form, with the normal forms it's often more instructive to show a counterexample rather than an example per se. Consider, therefore, relvar SPJ, with attributes SNO (supplier number), PNO (part number), and JNO (project number), and predicate *Supplier SNO supplies part PNO to project JNO*. Let that relvar be all key (i.e., let no proper subset of the heading be a key). Let the relvar also be subject to the constraint that if (a) supplier sno supplies part pno and (b) part pno is supplied to project jno and (c) project jno is supplied by supplier sno , then (d) supplier sno supplies part pno to project jno . Then SPJ is equal to the join of its projections on {SNO,PNO}, {PNO,JNO}, and {JNO,SNO}—in other words, the join dependency

$$\bowtie \{ \{ \text{SNO} , \text{PNO} \} , \{ \text{PNO} , \text{JNO} \} , \{ \text{JNO} , \text{SNO} \} \}$$

holds in SPJ—and so that relvar can be nonloss decomposed into those three projections. Since no component of that JD is a superkey (the sole superkey being the entire heading), relvar SPJ isn't in ETNF, though it is in 4NF.

Note: The ETNF definition refers to “every JD that holds in R .” In checking whether some relvar R is in fact in ETNF, however, it's easy to see that it's sufficient just to check those JDs that have been explicitly declared for R . In fact, it's sufficient just to check those JDs

that have been explicitly declared for R and are irreducible (see irreducible JD).

essentiality Let DM be a data model in the first sense of that term, q.v., and let DS be a data structure supported by DM . Let dm be a data model in the second sense of that term, constructed in accordance with the features provided by DM , and let dm include an occurrence ds of DS . Let db be a database conforming to dm . If removal from db of the data corresponding to ds would cause a loss of information from db , then ds is essential in dm (and, loosely, DS is essential in DM).

Examples: 1. Consider a hierarchic analog of the suppliers-and-parts database, in which (a) suppliers are represented by records with fields SNO, SNAME, STATUS, and CITY, (b) shipments are represented by records with fields SNO, PNO, and QTY, and (c) there's a hierarchic "link" connecting each supplier record to the corresponding shipment records. (The "link" can be thought of as a pointer chain that starts at the pertinent supplier record, runs through all of the corresponding shipment records in some order, and finally connects back to the supplier record in question.) Then that link is inessential—there's no information that can be obtained from the database using it that can't alternatively be obtained without it. 2. Suppose the foregoing hierarchic design is modified in such a way as to remove the SNO field from the shipment records, while leaving everything else unchanged. Then the link is now essential (for without it, there's no way to tell which shipments correspond to which suppliers).

Note: Hierarchic and other nonrelational systems provide numerous different ways of representing data, any or all of which can be used "essentially"—links and pointers, record ordering, repeating

groups, and so forth. By contrast, relational systems provide just one way (viz., relations themselves), and so relations themselves are the sole essential information carrier in relational systems. Now, if data model *DM* provides n distinct ways, essential or inessential, of representing information, then it's axiomatic that *DM* must also support n distinct sets of operators. However, there's nothing useful that can be done if $n > 1$ that can't be done if $n = 1$ (and $n = 1$ is the minimum, of course). And for the relational model, we do have $n = 1$; that is, the relational model supports just one data structure, the relation itself, and that data structure is clearly essential, since if it were removed that model would be incapable of representing anything at all. However, since the relational model is in fact capable of representing absolutely any data whatsoever, any data model that supports relations in some shape or form as well as some additional data structure *DS* must be such that either relations are inessential or *DS* is. But if relations are inessential, then *DS* must be effectively equivalent to relations anyway!—in which case it could be argued that it's really *DS* that's inessential, not relations. What's more, a data model that doesn't "support relations in some shape or form" is unlikely in the extreme; even SQL could be said to support relations if various SQL idiosyncrasies—nulls, anonymous columns, duplicate rows, etc.—are avoided. Thus, for example, pointers (object IDs), bags, lists, and arrays could all be removed from the so called object model without any loss of representational power. Indeed, the fact that they're not removed is prima facie evidence that "the object model" fails to distinguish properly between model and implementation issues.

ETNF Essential tuple normal form.

eventual consistency See consistency.

EVERY Keyword sometimes used as an alternative spelling for the aggregate operator **AND** (*see* aggregate operator).

example value (*Without inheritance*) Let T be a scalar type other than *omega* (*see* Part II of this dictionary). Then *The Third Manifesto* requires an example value of type T to be specified when T is defined, in order to ensure that T is nonempty. In the case of user defined types, **Tutorial D** uses the keyword **INIT** for this purpose, as here:

```
TYPE WEEKDAY ... INIT ( WEEKDAY('Sun') ) ;
```

Use of the keyword **INIT** here reflects an assumption that, in practice, otherwise uninitialized variables of type T will almost certainly be initialized to the example value defined for type T .

Note: In type definitions elsewhere in this dictionary, example values would mostly just be a distraction and are therefore usually omitted.

EXCEPT SQL analog of **MINUS**.

exclusive OR 1. (*Dyadic case*) If and only if p and q are predicates, their “exclusive OR” $(p) \text{ XOR } (q)$ is a predicate also. Let $(ip) \text{ XOR } (iq)$ be an invocation of that predicate, where ip and iq are invocations of p and q , respectively. Then that invocation $(ip) \text{ XOR } (iq)$ evaluates to **TRUE** if and only if exactly one of ip and iq evaluates to **TRUE**. In other words, $(p) \text{ XOR } (q)$ is equivalent to $\text{NOT}((p) \text{ EQUIV } (q))$. *Note:* The parentheses enclosing p and q in the predicate, and ip and iq in the invocation, might not be needed in practice. 2. (*N-adic case*) Let p_1, p_2, \dots, p_n ($n \geq 0$) be predicates. Then (and only then) the “exclusive OR” $\text{XOR } \{p_1, p_2, \dots, p_n\}$ is a predicate also; and if ip_1, ip_2, \dots, ip_n are

invocations of p_1, p_2, \dots, p_n , respectively, then the invocation $\text{XOR}\{ip_1, ip_2, \dots, ip_n\}$ returns TRUE if and only if exactly m of the invocations ip_1, ip_2, \dots, ip_n return TRUE, where m is odd. *Caveat:* This definition is motivated by a desire to preserve associativity; to be specific, it has the property that the expressions $\text{XOR}\{p_1, \text{XOR}\{p_2, p_3\}\}$, $\text{XOR}\{\text{XOR}\{p_1, p_2\}, p_3\}$, and $\text{XOR}\{p_1, p_2, p_3\}$ are all truth functionally equivalent. On the other hand, it also has the property that $\text{XOR}\{p_1, p_2, p_3\}$ and $\text{NOT}(\text{EQUIV}\{p_1, p_2, p_3\})$, as this latter expression is defined in this dictionary, are *not* truth functionally equivalent. It would be possible to come up with a different and possibly more intuitive definition, according to which the invocation $\text{XOR}\{ip_1, ip_2, \dots, ip_n\}$ returns TRUE if and only if exactly one of the invocations ip_1, ip_2, \dots, ip_n returns TRUE. However, the two definitions are themselves clearly not equivalent; in other words, they define two logically distinct operators (though they both reduce to the simple dyadic case if $n = 2$, as is surely to be desired).

exclusive union 1. (*Dyadic case*) Let relations r_1 and r_2 be of the same type T . Then (and only then) the expression $r_1 \text{XUNION} r_2$ denotes the exclusive union of r_1 and r_2 , and it returns the relation of type T with body the set of all tuples t such that t appears in exactly one of r_1 and r_2 . 2. (*N-adic case*) Let relations r_1, r_2, \dots, r_n ($n \geq 0$) all be of the same type T . Then (and only then) the expression $\text{XUNION}\{r_1, r_2, \dots, r_n\}$ denotes the exclusive union of r_1, r_2, \dots, r_n , and it returns the relation of type T with body the set of all tuples t such that t appears in exactly m of r_1, r_2, \dots, r_n , where m is odd (and possibly different for different tuples t). *Note:* If $n = 0$, (a) some syntactic mechanism, not shown here, is needed to specify the pertinent type T and (b) the result is the empty relation, q.v., of that type. Note too (a)

that exclusive union (which is also known as symmetric difference) is to exclusive OR as union is to inclusive OR, and (b) that the relational exclusive union operator differs in certain respects from the mathematical or set theory operator of the same name, q.v. Note finally that exclusive union can also be used as an aggregate operator, q.v.

Example: The expression $S\{CITY\} \text{ XUNION } P\{CITY\}$ denotes the exclusive union of the projections on $\{CITY\}$ of the relations that are the current values of relvars S and P . That exclusive union is a relation r of type $RELATION \{CITY \text{ CHAR}\}$. Moreover, if the current values of relvars S and P are s and p , respectively, the body of that relation r consists of all tuples of the form $\langle c \rangle$ that appear in either $s\{CITY\}$ or $p\{CITY\}$ but not both—meaning c is either a current supplier city that's not a current part city or vice versa. Note that the expression $S\{CITY\} \text{ XUNION } P\{CITY\}$ is logically equivalent to the expression $(S\{CITY\} \text{ MINUS } P\{CITY\}) \text{ UNION } (P\{CITY\} \text{ MINUS } S\{CITY\})$.

exclusive union (bag theory) *See* bag.

exclusive union (set theory) The set of all elements appearing in either but not both of two given sets. *Note:* The foregoing definition could be extended to apply to any number of sets, thus: The exclusive union of sets s_1, s_2, \dots, s_n ($n \geq 0$) is the set of all values v such that v appears in exactly m of s_1, s_2, \dots, s_n , where m is odd (and possibly different for different values v).

existential quantifier Let $p(x)$ be a predicate with a parameter x ; then $\text{EXISTS } x (p(x))$ is a predicate, and it means “There exists at least one argument value v that can be substituted for the parameter x such that

$p(v)$ evaluates to 'TRUE.'" In this example, EXISTS x is an existential quantifier, and x is an existentially quantified bound variable, q.v. *Note:* Some writers refer to EXISTS by itself as the quantifier; the literature is not consistent on this point. More important, note that if $v1, v2, \dots, vn$ are all of the possible argument values in the foregoing example, then EXISTS $x (p(x))$ is defined to be shorthand for OR $\{(p(v1)), (p(v2)), \dots, (p(vn))\}$ (see disjunction, second definition). Observe in particular that this expression evaluates to FALSE if $n = 0$ (i.e., if the bound variable x has an empty range), because FALSE is the identity with respect to OR. Observe further that the expression EXISTS $x (p(x))$ is logically equivalent to the expression NOT (FORALL x (NOT $(p(x))$)). See also EXISTS; UNIQUE; *contrast* universal quantifier.

Examples: See bound variable; domain calculus; free variable; tuple calculus; and elsewhere.

EXISTS See existential quantifier. *Note:* In the literature (but not in this dictionary), EXISTS is often represented by a backward E, thus: \exists . The keyword is also sometimes used as an alternative spelling for the aggregate operator OR (see aggregate operator). For example, the aggregate operator invocation OR (S, STATUS > 10), which means "At least one supplier has status greater than 10," might alternatively, and intuitively very reasonably, be written thus: EXISTS (S, STATUS > 10).

expanded cartesian product See cartesian product.

explicit dependency A dependency—e.g., an FD or JD, or some more general constraint—that's explicitly declared for some relvar, and is thereby required to hold in that relvar. *Contrast* implicit dependency.

explicit dynamic variable *See* instance.

expressible database *See Principle of Database Relativity.*

expressible relation Any relation that, given a particular set of relations, either is contained in that set or can be derived from those that are (*see* derived relation).

expressible relvar Any relvar that, given a particular set of relvars, either is contained in that set or can be derived from those that are (*see* derived relvar).

expression (*Without inheritance*) In a programming language, a read-only operator invocation; a construct that denotes a value; in effect, a rule for computing, or determining, the value in question. Every expression is of some type—namely, the type of the value it denotes. Literals (q.v.), constant references (q.v), and variable references (q.v.) are all considered to be read-only operator invocations and thus all constitute legal expressions. *See also* closed expression; open expression; *contrast* statement.

Examples: $X+Y$ is an expression; in fact, it's an invocation of the operator “+”, and it denotes the value that's the sum of the current values of the variables X and Y . By contrast,

$$Z := X + Y ;$$

is a statement; it assigns the value denoted by the expression $X+Y$ appearing on the right side to the variable Z referenced on the left side. *Note:* In both of the foregoing examples, X and Y are variable references and thus themselves constitute (sub)expressions in turn.

expression transformation Transforming a given expression into

another expression that's logically equivalent to the given expression and thus denotes the same value. The process applies to relational expressions in particular, where it's sometimes called "query rewrite." Query rewrite is typically done for performance reasons; it can be done either by the user or—much more important—by the system (*see optimizer*). *Note:* The term *query rewrite* is also used in certain commercial products with a somewhat more limited meaning. *Caveat lector.*

Example: The relational expression $(r1 \text{ WHERE } bx1) \text{ JOIN } (r2 \text{ WHERE } bx2)$, where $r1$ and $r2$ are relations and $bx1$ and $bx2$ are restriction conditions, q.v., is logically equivalent to the relational expression $(r1 \text{ JOIN } r2) \text{ WHERE } (bx1) \text{ AND } (bx2)$; therefore, either of these relational expressions can be transformed into the other. Transforming the second into the first is likely to be advantageous from a performance standpoint, because the first means doing the restrictions before the join; thus, it's likely that the input relations to the join will be smaller and the output will be smaller too. In fact, this transformation could make the difference between keeping the result of the join in main memory and having to spill it out to secondary storage.

expressive completeness A database design is expressively complete if and only if it's capable of representing all facts about the real world that need to be represented.

Example: Consider the suppliers-and-parts database. That database is expressively complete (or let's agree so for the sake of the example, at least). Now suppose we were to replace relvars S and SP by their join (SSP, say). Then the resulting design wouldn't be expressively complete, because it would be incapable of representing information

concerning suppliers (such as supplier S5 in Fig. 1) who currently supply no parts.

EXTEND *See* extension.

extended cartesian product *See* cartesian product.

Extensible Markup Language *See* XML.

extension 1. (*Relational algebra, first form*) Let relation r not have an attribute called A . Then (and only then) the expression $\text{EXTEND } r : \{A := \text{exp}\}$ denotes an extension of r , and it returns the relation with heading the heading of r extended with attribute A and body the set of all tuples t such that t is a tuple of r extended with a value for A that's computed by evaluating the expression exp on that tuple of r . *See also* tuple extension; WITH. 2. (*Relational algebra, second form*) Let relation r have an attribute called A . Then (and only then) the expression $\text{EXTEND } r : \{A := \text{exp}\}$ denotes an extension of r , and it returns the relation with heading the same as that of r and body the set of all tuples t such that t is derived from a tuple of r by replacing the value of A by a value that's computed by evaluating the expression exp on that tuple of r . Again, *see also* tuple extension; WITH. 3. (*Predicate*) Let p be a predicate; then the extension of p consists of all full instantiations of p (i.e., all propositions that can be derived from p by full instantiation) that evaluate to TRUE. 4. (*Relation*) Following on from the previous definition, let r be a relation. Then the heading of r can be regarded as representing a predicate (*see* relation predicate), and the body of r can be regarded as representing the extension of that predicate. Hence, the term *extension* is also sometimes used to refer to the body of a relation. *Contrast* intension. 5. (*Set theory*) *See* axiom of extension.