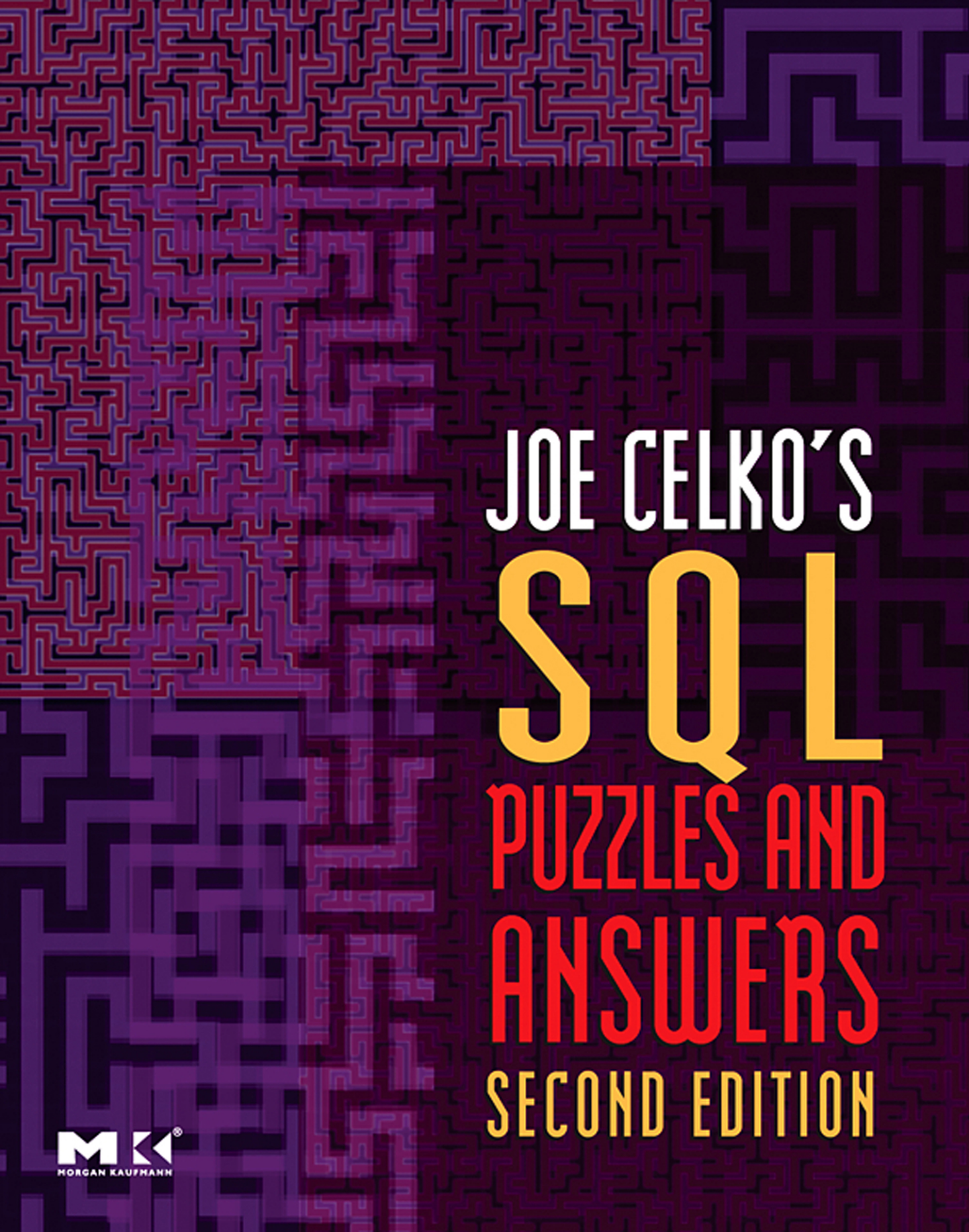

JOE CELKO'S
SQL
PUZZLES AND
ANSWERS
SECOND EDITION
MORGAN KAUFMANN

JOE CELKO'S
SQL PUZZLES & ANSWERS

Second Edition

The Morgan Kaufmann Series in Data Management Systems
Series Editor: Jim Gray, Microsoft Research

Joe Celko's Analytics and OLAP in SQL
Joe Celko

Data Preparation for Data Mining Using SAS
Mamdouh Refaat

Querying XML: XQuery, XPath, and SQL/XML in Context
Jim Melton and Stephen Buxton

Data Mining: Concepts and Techniques, Second Edition
Jiawei Han and Micheline Kamber

Database Modeling and Design: Logical Design, Fourth Edition
Toby J, Teorey, Sam S. Lightstone and Thomas P. Nadeau

Foundations of Multidimensional and Metric Data Structures
Hanan Samet

Joe Celko's SQL for Smarties: Advanced SQL Programming, Third Edition
Joe Celko

Moving Objects Databases
Ralf Hartmut Güting and Markus Schneider

Joe Celko's SQL Programming Style
Joe Celko

Data Mining, Second Edition: Concepts and Techniques
Ian Witten and Eibe Frank

Fuzzy Modeling and Genetic Algorithms for Data Mining and Exploration
Earl Cox

Data Modeling Essentials, Third Edition
Graeme C. Simsion and Graham C. Witt

Location-Based Services
Jochen Schiller and Agnès Voisard

Database Modeling with Microsft" Visio for Enterprise Architects
Terry Halpin, Ken Evans, Patrick Hallock, Bill Maclean

Designing Data-Intensive Web Applications
Stephano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera

Mining the Web: Discovering Knowledge from Hypertext Data
Soumen Chakrabarti

Advanced SQL: 1999—Understanding Object-Relational and Other Advanced Features
Jim Melton

Database Tuning: Principles, Experiments, and Troubleshooting Techniques
Dennis Shasha and Philippe Bonnet

SQL:1999—Understanding Relational Language Components
Jim Melton and Alan R. Simon

Information Visualization in Data Mining and Knowledge Discovery
Edited by Usama Fayyad, Georges G. Grinstein, and Andreas Wierse

Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery
Gerhard Weikum and Gottfried Vossen

Spatial Databases: With Application to GIS
Philippe Rigaux, Michel Scholl, and Agnes Voisard

Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design
Terry Halpin

Component Database Systems
Edited by Klaus R. Dittrich and Andreas Geppert

Managing Reference Data in Enterprise Databases: Binding Corporate Data to the Wider World
Malcolm Chisholm

Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies
Jim Melton and Andrew Eisenberg

Database: Principles, Programming, and Performance, Second Edition
Patrick and Elizabeth O'Neil

The Object Data Standard: ODMG 3.0
Edited by R. G. G. Cattell and Douglas K. Barry

Data on the Web: From Relations to Semistructured Data and XML
Serge Abiteboul, Peter Buneman, and Dan Suciu

Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations
Ian Witten and Eibe Frank

Joe Celko's SQL for Smarties: Advanced SQL Programming, Second Edition
Joe Celko

Joe Celko's Data and Databases: Concepts in Practice
Joe Celko

Developing Time-Oriented Database Applications in SQL
Richard T. Snodgrass

Web Farming for the Data Warehouse
Richard D. Hackathorn

Management of Heterogeneous and Autonomous Database Systems
Edited by Ahmed Elmagarmid, Marek Rusinkiewicz, and Amit Sheth

Object-Relational DBMSs: Tracking the Next Great Wave, Second Edition
Michael Stonebraker and Paul Brown,with Dorothy Moore

A Complete Guide to DB2 Universal Database
Don Chamberlin

Universal Database Management: A Guide to Object/Relational Technology
Cynthia Maro Saracco

Readings in Database Systems, Third Edition
Edited by Michael Stonebraker and Joseph M. Hellerstein

Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM
Jim Melton

Principles of Multimedia Database Systems
V. S. Subrahmanian

Principles of Database Query Processing for Advanced Applications
Clement T. Yu and Weiyi Meng

Advanced Database Systems
Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari

Principles of Transaction Processing
Philip A. Bernstein and Eric Newcomer

Using the New DB2: IBMs Object-Relational Database System
Don Chamberlin

Distributed Algorithms
Nancy A. Lynch

Active Database Systems: Triggers and Rules For Advanced Database Processing
Edited by Jennifer Widom and Stefano Ceri

Migrating Legacy Systems: Gateways, Interfaces, & the Incremental Approach
Michael L. Brodie and Michael Stonebraker

Atomic Transactions
Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete

Query Processing for Advanced Database Systems
Edited by Johann Christoph Freytag, David Maier, and Gottfried Vossen

Transaction Processing: Concepts and Techniques
Jim Gray and Andreas Reuter

Building an Object-Oriented Database System: The Story of O2
Edited by François Bancilhon, Claude Delobel, and Paris Kanellakis

Database Transaction Models for Advanced Applications
Edited by Ahmed K. Elmagarmid

A Guide to Developing Client/Server SQL Applications
Setrag Khoshafian, Arvola Chan, Anna Wong, and Harry K. T. Wong

The Benchmark Handbook for Database and Transaction Processing Systems, Second Edition
Edited by Jim Gray

Camelot and Avalon: A Distributed Transaction Facility
Edited by Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector

Readings in Object-Oriented Database Systems
Edited by Stanley B. Zdonik and David Maier

This Page Intentionally Left Blank

JOE CELKO'S
SQL PUZZLES & ANSWERS

Second Edition

Joe Celko

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
MORGAN KAUFMANN PUBLISHERS IS AN IMPRINT OF ELSEVIER

Publisher	Diane Cerra
Publishing Services Manager	George Morrison
Editorial Assistant	Asma Palmeiro
Cover Design	Side by Side Studios
Cover Image	Side by Side Studios
Cover Designer	Eric DeCicco
Composition	Multiscience Press, Inc.
Copyeditor	Multiscience Press, Inc.
Proofreader	Multiscience Press, Inc.
Indexer	Multiscience Press, Inc.
Interior printer	The Maple-Vail Book Manufacturing Group
Cover printer	Phoenix Color Corp.

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

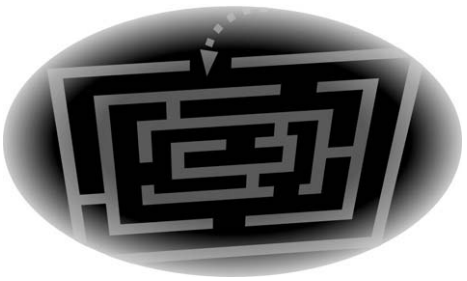
Library of Congress Cataloging-in-Publication Data

Application submitted.

ISBN-10 : 0-12-373596-3
ISBN-13: 978-0-12-373596-3

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
06 07 08 09 5 4 3 2 1

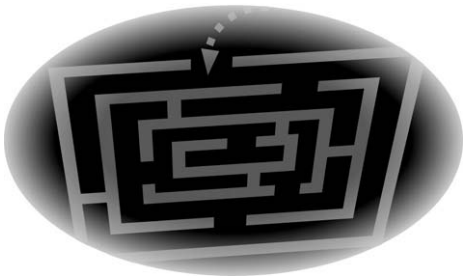


To chanticleer Michael—
I now have a convincing argument
against solipsism for you.

This Page Intentionally Left Blank

CONTENTS

This Page Intentionally Left Blank



Introduction

Back in the early and mid-1990s, I wrote regular magazine columns in *Database Programming & Design* and later in *DBMS* magazine. The gimmick I used to attract reader responses was to end each column with a SQL programming puzzle. Ten years later, those two magazines were consolidated into *Intelligent Enterprise*. My SQL puzzles moved to some smaller publications and then finally faded away. Today, I throw out a puzzle or two on the www.dbazine.com Web site and other places on the Internet rather than in print media.

Over the years, college students had all kinds of programming contests that used the procedural language du jour—C, Pascal, then Java and C++ today. There is not much for database programmers to test themselves against, except my little puzzle book.

I would often find my puzzles showing up in homework assignments because I was the only source that teachers knew about for SQL problems. I would then get an e-mail from a lazy student wanting me to do his homework for him, unaware of the source of the assignment.

Back in those early days, the de facto standard was SQL-86, and the SQL-92 standard was a design goal for the database vendors. Today, most vendors have gotten most of SQL-92 into their products. The design goal is now the SQL-99 standard's OLAP features.

A decade ago, college students took RDBMS courses, and becoming an SQL programmer required some expertise. SQL products were expensive and the best ones lived on mainframes.

Today, colleges are not teaching RDBMS theory in the undergrad curriculum. SQL is not as exotic as it once was, and you can get cheap or open-source SQL databases. The Internet is full of newsgroups where you can get help for particular products.

The bad news is that the quality of SQL programmers has gotten worse because people who have no foundations in RDBMS or training in SQL are being asked to write SQL inside their native programming languages.

This collection of puzzles includes the original puzzles, so that the original readers can look up their favorites. But now many of them have new solutions, some use the older syntax, and some use the newer features. Many of the original solutions have been cooked by other people over the years. The term "cooked" is a puzzler's term for finding a better solution than the proposer of the problem presented. The original book contained 50 puzzles; this edition has 75 puzzles.

In the first edition, I tried to organize the puzzles by categories rather than in chronological order or by complexity. This edition, I have given up my informal category scheme because it made no sense. A problem might be solved by a change to the DDL or a query, so should it be categorized as a DDL puzzle or a DML puzzle?

I have tried to credit the people involved with each problem, but if I missed someone, I apologize.

Acknowledgments, Corrections, and Future Editions

I will be glad to receive corrections, new tricks and techniques, and other suggestions for future editions of this book. Send your ideas to or contact me through the publisher, Morgan Kaufmann.

I would like to thank Diane Cerra of Morgan Kaufmann, David Kalman of *DBMS* magazine, Maurice Frank of *DBMS* magazine, David Stodder of *Database Programming & Design*, Phil Chapnick of Miller-Freeman, Frank Sweet of *Boxes & Arrows,* and Dana Farver at www.dbazine.com.

Special thanks to Richard Romley of Smith Barney for cooking so many of my early puzzles, all the people on CompuServe and SQL newsgroups who sent me e-mail all these years, and the people who are posting on the newsgroups today (I used your newsgroup handles, so people can search for your postings). These include, but are not limited to, Raymond D'Anjou, Dieter Noeth, Alexander Kuznetsov, Andrey

Odegov, Steve Kass,Tibor Karaszi, David Portas, Hugo Kornelis, Aaron Bertrand, Itzik Ben-Gan, Tom Moreau, Serge Rielau, Erland Sommarskog, Mikito Harakiri, Adam Machanic, and Daniel A. Morgan.

This Page Intentionally Left Blank

PUZZLE
1 FISCAL YEAR TABLES

Let's write some CREATE TABLE statements that are as complete as possible. This little exercise is important because SQL is a declarative language and you need to learn how to specify things in the database instead of in the code.

The table looks like this:

```
CREATE TABLE FiscalYearTable1
(fiscal_year INTEGER,
 start_date DATE,
 end_date DATE);
```

It stores date ranges for determining what fiscal year any given date belongs to. For example, the federal government runs its fiscal year from October 1 until the end of September. The scalar subquery you would use to do this table lookup is:

```
(SELECT F1.fiscal_year
  FROM FiscalYearTable1 AS F1
 WHERE outside_date BETWEEN F1.start_date AND F1.end_date)
```

Your assignment is to add all the constraints you can think of to the table to guarantee that it contains only correct information.

While vendors all have different date and time functions, let's assume that all we have is the SQL-92 temporal arithmetic and the function EXTRACT ([YEAR | MONTH | DAY] FROM <date expression>), which returns an integer that represents a field within a date.

Answer #1

1. First things first; make all the columns NOT NULL since there is no good reason to allow them to be NULL.
2. Most SQL programmers immediately think in terms of adding a PRIMARY KEY, so you might add the constraint PRIMARY KEY (fiscal_year, start_date, end_date) because the fiscal year is really another name for the pair (start_date, end_date). This is not enough, because it would allow this sort of error:

```
(1995, '1994-10-01', '1995-09-30')
(1996, '1995-10-01', '1996-08-30') <== error!
(1997, '1996-10-01', '1997-09-30')
(1998, '1997-10-01', '1997-09-30')
```

You could continue along the same lines and fix some problems by adding the constraints UNIQUE (fiscal_year), UNIQUE (start_date), and UNIQUE (end_date), since we do not want duplicate dates in any of those columns.

3. The constraint that almost everyone forgets to add because it is so obvious is:

   CHECK (start_date < end_date) or CHECK (start_date <= end_date), as is appropriate.

4. A better way would be to use the constraint PRIMARY KEY (fiscal_year) as before, but then since the start and end dates are the same within each year, you could use constraints on those column declarations:

```
CREATE TABLE FiscalYearTable1
(fiscal_year INTEGER  NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 CONSTRAINT valid_start_date
   CHECK ((EXTRACT (YEAR FROM start_date) = fiscal_year - 1)
            AND (EXTRACT (MONTH FROM start_date) = 10)
           AND CHECK (EXTRACT (DAY FROM start_date) = 01)),
 end_date DATE NOT NULL,
 CONSTRAINT valid_end_date
   CHECK ((EXTRACT (YEAR FROM end_date) = fiscal_year)
    AND (EXTRACT (MONTH FROM end_date) = 09)
    AND (EXTRACT (DAY FROM end_date) = 30)));
```

You could argue for making each predicate a separate constraint to give more detailed error messages. The predicates on the year components of the start_date and end_date columns also guarantee uniqueness because they are derived from the unique fiscal year.

5. Unfortunately, this method does not work for all companies. Many companies have an elaborate set of rules that involve taking into account the weeks, weekends, and weekdays involved.

They do this to arrive at exactly 360 days or 52 weeks in their accounting year. In fact, there is a fairly standard accounting practice of using a "4 weeks, 4 weeks, 5 weeks" quarter with some fudging at the end of the year; you can have a leftover week between 3 and 11 days. The answer is a FiscalMonth table along the same lines as this FiscalYears example.

A constraint that will work surprisingly well for such cases is:

```
CHECK ((end_date - start_date) = INTERVAL 359 DAYS)
```

where you adjust the number of days to fit your rules (i.e., 52 weeks * 7 days = 364 days). If the rules allow some variation in the size of the fiscal year, then replace the equality test with a BETWEEN predicate.

Now, true confession time. When I have to load such a table in a database, I get out my spreadsheet and build a table using the built-in temporal functions. Spreadsheets have much better temporal functions than databases, and there is a good chance that the accounting department already has the fiscal calendar in a spreadsheet.

PUZZLE
2 ABSENTEES

This problem was presented on the MS ACCESS forum on CompuServe by Jim Chupella. He wanted to create a database that tracks employee absentee rates. Here is the table you will use:

```
CREATE TABLE Absenteeism
(emp_id INTEGER NOT NULL REFERENCES Personnel (emp_id),
 absent_date DATE NOT NULL,
 reason_code CHAR (40) NOT NULL REFERENCES ExcuseList
(reason_code),
 severity_points INTEGER NOT NULL CHECK (severity_points
BETWEEN 1 AND 4),
 PRIMARY KEY (emp_id, absent_date));
```

An employee ID number identifies each employee. The reason_code is a short text explanation for the absence (for example, "hit by beer truck," "bad hair day," and so on) that you pull from an ever-growing and imaginative list, and severity point is a point system that scores the penalty associated with the absence.

If an employee accrues 40 severity points within a one-year period, you automatically discharge that employee. If an employee is absent more than one day in a row, it is charged as a long-term illness, not as a typical absence. The employee does not receive severity points on the second, third, or later days, nor do those days count toward his or her total absenteeism.

Your job is to write SQL to enforce these two business rules, changing the schema if necessary.

Answer #1

Looking at the first rule on discharging personnel, the most common design error is to try to drop the second, third, and later days from the table. This approach messes up queries that count sick days, and makes chains of sick days very difficult to find.

The trick is to allow a severity score of zero, so you can track the long-term illness of an employee in the Absenteeism table. Simply change the severity point declaration to "`CHECK (severity_points BETWEEN 0 AND 4)`" so that you can give a zero to those absences that do not count.

This is a trick newbies miss because storing a zero seems to be a waste of space, but zero is a number and the event is a fact that needs to be noted.

```
UPDATE Absenteeism
   SET severity_points= 0,
       reason_code = 'long term illness'
 WHERE EXISTS
        (SELECT *
           FROM Absenteeism AS A2
          WHERE Absenteeism.emp_id = A2.emp_id
            AND Absenteeism.absent_date = (A2.absent_date -
INTERVAL 1 DAY));
```

When a new row is inserted, this update will look for another absence on the day before and change its severity point score and reason_code in accordance with your first rule.

The second rule for firing an employee requires that you know what his or her current point score is. You would write that query as follows:

```
SELECT emp_id, SUM(severity_points)
  FROM Absenteeism
 GROUP BY emp_id;
```

This is the basis for a grouped subquery in the DELETE statement you finally want. Personnel with less than 40 points will return a NULL, and the test will fail.

```
DELETE FROM Personnel
 WHERE emp_id = (SELECT A1.emp_id
                   FROM Absenteeism AS A1
                  WHERE A1.emp_id = Personnel.emp_id
                  GROUP BY A1.emp_id
                 HAVING SUM(severity_points) >= 40);
```

The `GROUP BY` clause is not really needed in SQL-92, but some older SQL implementations will require it.

Answer #2

Bert Scalzo, a senior instructor for Oracle Corporation, pointed out that the puzzle solution had two flaws and room for performance improvements.

The flaws are quite simple. First, the subquery does not check for personnel accruing 40 or more severity points within a one-year period, as required. It requires the addition of a date range check in the `WHERE` clause:

```
DELETE FROM Personnel
 WHERE emp_id = (SELECT A1.emp_id
      FROM Absenteeism AS A1
          WHERE A1.emp_id = Personnel.emp_id
             AND absent_date
             BETWEEN CURRENT_TIMESTAMP - INTERVAL 365 DAYS
             AND CURRENT_TIMESTAMP
             GROUP BY A1.emp_id
             HAVING SUM(severity_points) >= 40);
```

Second, this SQL code deletes only offending personnel and not their absences. The related Absenteeism row must be either explicitly or implicitly deleted as well. You could replicate the above deletion for the Absenteeism table. However, the best solution is to add a cascading deletion clause to the Absenteeism table declaration:

```
CREATE TABLE Absenteeism
   (    ... emp_id INTEGER NOT NULL
                   REFERENCES Personnel(emp_id)
                   ON DELETE CASCADE,
        ...);
```

The performance suggestions are based on some assumptions. If you can safely assume that the `UPDATE` is run regularly and people do not change their departments while they are absent, then you can improve the `UPDATE` command's subquery:

```
UPDATE Absenteeism AS A1
   SET severity_points = 0,
       reason_code = 'long term illness'
 WHERE EXISTS
```

```
      (SELECT *
         FROM absenteeism as A2
        WHERE A1.emp_id = A2.emp_id
          AND (A1.absent_date + INTERVAL 1 DAY) =
A2.absent_date);
```

There is still a problem with long-term illnesses that span weeks. The current situation is that if you want to spend your weekends being sick, that is fine with the company. This is not a very nice place to work. If an employee reports in absent on Friday of week number 1, all of week number 2, and just Monday of week number 3, the `UPDATE` will catch only the five days from week number 2 as long-term illness. The Friday and Monday will show up as sick days with severity points. The subquery in the `UPDATE` requires additional changes to the missed-date chaining.

I would avoid problems with weekends by having a code for scheduled days off (weekends, holidays, vacation, and so forth) that carry a severity point of zero. A business that has people working weekend shifts would need such codes.

The boss could manually change the Saturday and Sunday "weekend" codes to "long-term illness" to get the `UPDATE` to work the way you described. This same trick would also prevent you from losing scheduled vacation time if you got the plague just before going on a cruise. If the boss is a real sweetheart, he or she could also add compensation days for the lost weekends with a zero severity point to the table, or reschedule an employee's vacation by adding absences dated in the future.

While I agreed that I left out the aging on the dates missed, I will argue that it would be better to have another `DELETE` statement that removes the year-old rows from the Absenteeism table, to keep the size of the table as small as possible.

The expression

```
(BETWEEN CURRENT_TIMESTAMP - INTERVAL 365 DAYS AND
CURRENT_TIMESTAMP)
```

could also be

```
(BETWEEN CURRENT_TIMESTAMP - INTERVAL 1 YEAR AND
CURRENT_TIMESTAMP),
```

so the system would handle leap years. Better yet, DB2 and some other SQL products have an AGE(date1) function, which returns the age in

years of something that happened on the date parameter. You would then write (AGE(absent_date) >= 1) instead.

Answer #3

Another useful tool for this kind of problem is a Calendar table, which has the working days that can count against the employee. In the 10 years since this book was first written, this has become a customary SQL programming practice.

```
SELECT A.emp_id,
       SUM(A.severity_points) AS absentism_score
  FROM Absenteeism AS A, Calendar AS C
 WHERE C1.cal_date = A.absent_date
   AND A.absent_date
       BETWEEN CURRENT_TIMESTAMP - INTERVAL 365 DAYS
           AND CURRENT_TIMESTAMP
   AND C1.date_type = 'work'
 GROUP BY emp_id
HAVING SUM(A.severity_points)>= 40;
```

Some people will also have a column in the Calendar table that Julianizes the working days. Holidays and weekends would carry the same Julian number as the preceding workday. For example (cal_date, Julian_workday) :

```
('2006-04-21', 42) - Friday
('2006-04-22', 42) - Saturday
('2006-04-23', 42) - Sunday
('2006-04-24', 43) - Monday
```

You do the math from the current date's Julian workday number to find the start of their adjusted one-year period.

PUZZLE
3 THE ANESTHESIA PUZZLE

Leonard C. Medal came up with this nifty little problem many years ago. Anesthesiologists administer anesthesia during surgeries in hospital operating rooms. Information about each anesthesia procedure is recorded in a table.

```
Procs
proc_id  anest_name start_time  end_time
====================================
   10    'Baker'    08:00       11:00
   20    'Baker'    09:00       13:00
   30    'Dow'      09:00       15:30
   40    'Dow'      08:00       13:30
   50    'Dow'      10:00       11:30
   60    'Dow'      12:30       13:30
   70    'Dow'      13:30       14:30
   80    'Dow'      18:00       19:00
```

Note that some of the times for a given anesthesiologist overlap. This is not a mistake. Anesthesiologists, unlike surgeons, can move from one operating room to another while surgeries are underway, checking on each patient in turn, adjusting dosages, and leaving junior doctors and anesthesia nurses to monitor the patients on a minute-to-minute basis.

Pay for the anesthesiologist is per procedure, but there's a catch. There is a sliding scale for remuneration for each procedure based on the maximum count of simultaneous procedures that an anesthesiologist has underway. The higher this count, the lower the amount paid for the procedure.

For example, for procedure #10, at each instant during that procedure Dr. Baker counts up how many total procedures in which he was concurrently involved. This maximum count for procedure #10 is 2. Based on the "concurrency" rules, Dr. Baker gets paid 75% of the fee for procedure #10.

The problem then is to determine for each procedure over its duration, the maximum, instantaneous count of procedures carried out by the anesthesiologist.

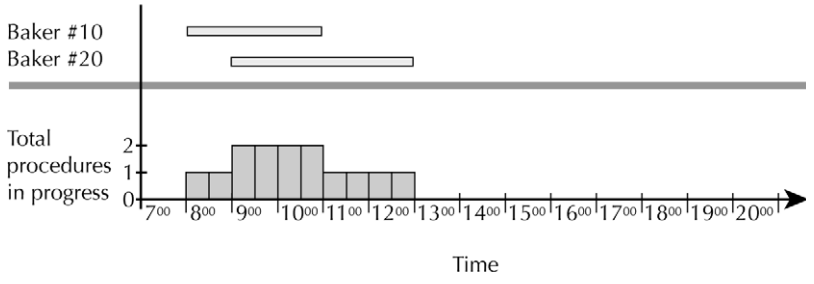
We can derive the answer graphically at first to get a better understanding of the problem.

Example 1 shows two overlapping procedures. The upper, Gantt-like graph displays the elapsed time of the procedure we are evaluating (the subject procedure) and all the doctor's other procedures that overlap in time.

The lower graph (Instantaneous Count of In-Progress Procedures) shows how many procedures are underway at each moment. It helps to think of a slide rule hairline moving from left to right over the Gantt chart while each procedure start or end is plotted stepwise on the lower chart.

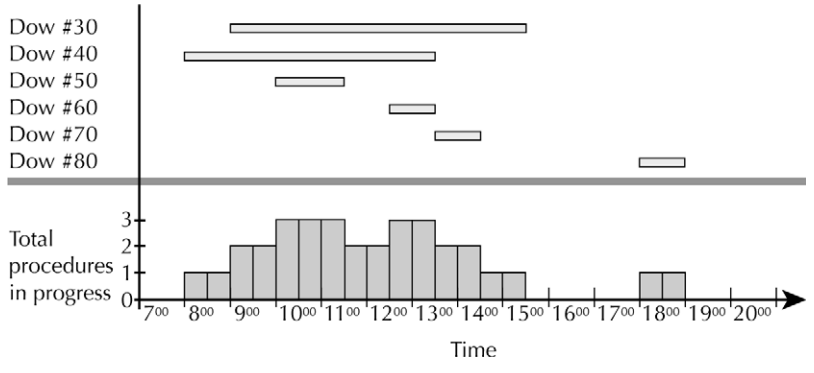
We can see in Example 1 by inspection that the maximum is 2.

Example 1—Dr. Baker, Proc #10



Example 2 shows a more complex set of overlapping procedures, but the principle is the same. The maximum, which happens twice, is 3.

Example 2—Dr. Dow, Proc #30



Note that the correct answer is not the number of overlapping procedures but the maximum instantaneous count. The puzzle is how to do this for each procedure using SQL. Here is the desired result for the sample data:

```
proc_id  max_inst_count
============================
     10             2
     20             2
     30             3
     40             3
     50             3
     60             3
     70             2
     80             1
```

Answer #1

The first step is to convert each procedure into two Events—a start and end event—and put them in a view. The `UNION` operator appends the set of end Events to the set of start Events. A (+1) indicates a start event and a (-1) indicates an end event.

The `WHERE` clauses assure that the procedures compared overlap and are for the same anesthesiologist. The `NOT` condition eliminates procedures that do not overlap the subject procedure.

```
CREATE VIEW Events (proc_id, comparison_proc, anest_name,
event_time, event_type)
AS SELECT P1.proc_id,
          P2.proc_id,
          P1.anest_name,
          P2.start_time,
          +1
     FROM Procs AS P1, Procs AS P2
    WHERE P1.anest_name = P2.anest_name
      AND NOT (P2.end_time <= P1.start_time
               OR P2.start_time >= P1.end_time)
 UNION
   SELECT P1.proc_id,
          P2.proc_id,
          P1.anest_name,
          P2.end_time,
          -1 AS event_type
     FROM Procs AS P1, Procs AS P2
    WHERE P1.anest_name= P2.anest_name
      AND NOT (P2.end_time <= P1.start_time
                OR P2.start_time >= P1.end_time);
```

The result is this view shown here for procedure #10 only and sorted by `event_time` for clarity. Notice that the same anesthesiologist can start more than one procedure at the same time.

```
Events
proc_id  comparison_proc  anest_name event_time  event_type

=============================================================
   10               10         Baker    08:00       +1
   10               20         Baker    09:00       +1
   10               10         Baker    11:00       -1
   10               20         Baker    13:00       -1
```

Now, for each set of Events with the same `proc_id` id, we can compute for each event the sum of the `event_types` for Events that occur earlier. This series of backward-looking sums gives us the values represented by each step in the step charts.

```
SELECT E1.proc_id, E1.event_time,
    (SELECT SUM(E2.event_type)
       FROM Events AS E2
      WHERE E2.proc_id = E1.proc_id
        AND E2.event_time < E1.event_time)
    AS instantaneous_count
  FROM Events AS E1
 ORDER BY E1.proc_id, E1.event_time;
```

The result of this query is shown here for procedure #10 only.

```
 proc_id instantaneous_count
================================
    10           NULL
    10            1
    10            2
    10            1
```

You could put this result set into a view called `ConcurrentProcs`, then query the view to get the maximum instantaneous count for each subject procedure using this statement:

```
SELECT proc_id,
       MAX(instantaneous_count) AS max_inst
  FROM ConcurrentProcs
 GROUP BY proc_id;
```

But you could also extract the desired result directly from the Events view. You could do this by merging the two select statements:

```
SELECT E1.proc_id,
   MAX((SELECT SUM(E2.event_type)
          FROM Events AS E2
         WHERE E2.proc_id = E1.proc_id
           AND E2.event_time < E1.event_time)) AS
max_inst_count
  FROM Events AS E1
 GROUP BY E1.proc_id;
```

However, it is illegal to put a subquery expression in an aggregate function in SQL-92, so you are depending on a vendor extension.

Answer #2

Richard Romley's single-query answer depends on the SQL-92 table query expressions in the FROM clause, so that what had been a VIEW can be folded into the query.

```
SELECT P3.proc_id, MAX(ConcurrentProcs.tally)
  FROM (SELECT P1.anest_name, P1.start_time, COUNT(*)
          FROM Procs AS P1
               INNER JOIN
               Procs AS P2
            ON P1.anest_name= P2.anest
               AND P2.start_time <= P1.start_time
               AND P2.stop_time > P1.start_time
         GROUP BY P1.anest_name, P1.start_time)
         AS ConcurrentProcs(anest_name, start_time, tally)
           INNER JOIN
           Procs AS P3
           ON ConcurrentProcs.anest_name= P3.anest
            AND P3.start_time <= ConcurrentProcs.start_time
             AND P3.stop_time > ConcurrentProcs.start_time
 GROUP BY P3.proc_id;
```

Answer #3

This answer came from Lex van de Pol (aavdpol@hotmail.com) on June 9, 2000. The idea is to loop through all procedures (P1); for each procedure P1 you look at procedures P2 where their start time lies in the interval of procedure P2. For each start time you found of P2, count the number of procedures (P3) that are ongoing on that time. Then, take the maximum count for a certain procedure P1.

For doing this, Lex first created this view:

```
CREATE VIEW Vprocs (id1, id2, total)
AS SELECT P1.prc_id, P2.prc_id, COUNT(*)
     FROM Procs AS P1, Procs AS P2, Procs AS P3
    WHERE P2.ant_name = P1.ant_name
      AND P3.ant_name = P1.ant_name
      AND P1.prc_start <= P2.prc_start
      AND P2.prc_start < P1.prc_end
      AND P3.prc_start <= P2.prc_start
      AND P2.prc_start < P3.prc_end
    GROUP BY P1.prc_id, P2.prc_id;
```

He then took the maximum for each procedure P1:

```
SELECT id1 AS proc_id, MAX(total) AS max_inst_count
  FROM Vprocs
 GROUP BY id1;
```

The funny thing is, you do not need to look at the end times for procedures P2.

Answer #4

Bert C. Hughes (bhughes@twincities.net) came up with a solution in Microsoft ACCESS, a proprietary near-SQL language. Here is his code translated into a single SQL statement:

```
SELECT P1.proc_id, P1.anest_name, MAX(E1.ecount) AS maxops
  FROM Procs AS P1,
   -- E1 is # of processes active at time for each
anesthetist
       (SELECT P2.anest_name, P2.start_time, COUNT(*)
```

```
          FROM Procs AS P1, Procs AS P2
         WHERE P1.anest_name = P2.anest_name
           AND P1.start_time <= P2.start_time
           AND P1.end_time > P2.start_time
         GROUP BY P2.anest_name, P2.start_time)
       AS E1(anest_name, etime, ecount)
 WHERE E1.anest_name= P1.anest_name
   AND E1.etime >= P1.start_time
   AND E1.etime < P1.end_time
 GROUP BY P1.proc_id, P1.anest;
```

Answer #5

Another approach is to set up a Clock table, since you probably round the billing to within a minute. That means we would have a table with (24 hours * 60 minutes) = 1,440 rows per day, or 525,600 rows; a year's worth of scheduling. But you can also set up a VIEW for the current day:

```
SELECT X.anest_name, MAX(X.proc_tally)
  FROM (SELECT P1.anest_name, COUNT(DISTINCT proc_id)
          FROM Procs AS P1, Clock AS C
         WHERE C1.clock_time BETWEEN P1.start_time
                             AND P1.end_time
         GROUP BY P1.anest_name)
       AS X(anest_name, proc_tally)
 GROUP BY X.anest_name);
```

This is just another version of the Calendar auxiliary table. This kind of table depends on a known granularity—Calendars work to the day, and Clocks to the minute, usually. You also can create a VIEW that uses a table of one day's clock ticks stored in minutes and the system constant CURRENT_DATE.

```
CREATE VIEW TodayClock (clock_time)
AS
SELECT CURRENT_DATE + ticks
  FROM DayTicks;
```

PUZZLE
4 SECURITY BADGES

Due to rightsizing (we never say downsizing or outsourcing) at your company, you are now the security officer and database administrator. You want to produce a list of personnel and their active security badge numbers. Each employee can have many badges, depending on how many job sites they are currently working, but only one of their badges will be active at a time. The default is that the most recently issued badge is assumed to be active because it will be issued at a new job site. The badge numbers are random to prevent counterfeiting. Your task is to produce a list of personnel, each with the relevant active badge number. Let's use 'A' for active and 'I' for inactive badge status.

Answer #1

From the specification, you know that each employee can have all but one badge set to inactive, so it would be nice to enforce that at the database level.

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL PRIMARY KEY,
 emp_name CHAR(30) NOT NULL,
 ...);

CREATE TABLE Badges
(badge_nbr INTEGER NOT NULL PRIMARY KEY,
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 issued_date DATE NOT NULL,
 badge_status CHAR(1) NOT NULL CHECK (badge_status IN ('A',
'I')),
     ...
 CHECK (1 <= ALL (SELECT COUNT(badge_status)
                    FROM Badges
                   WHERE badge_status = 'A'
                   GROUP BY emp_id))
 );
```

In fairness, I must point out that a lot of SQL implementations will gag on the final CHECK() clause on Badges because of the self-reference in the predicate, but it is legal SQL-92 syntax. You could drop that

CHECK() clause and allow an employee to have no active badge. That, however, would mean that you have to create a way of updating the badge status of the most recently issued badge to "A" for the employees.

```
UPDATE Badges
   SET badge_status = 'A'
 WHERE ('I' = ALL (SELECT badge_status
                     FROM Badges AS B1
                    WHERE emp_id = B1.emp_id))
   AND (issued_date = (SELECT MAX (issuedate)
                         FROM Badges AS B2
                        WHERE emp_id = B2.emp_id));
```

Again, I must point out that a lot of SQL implementations will also gag on this update because of the correlation names. The rule in SQL-92 is that the scope of the table name in the `UPDATE` is the whole statement, and the current row is used for the column values referenced. Therefore, you have to use the correlation names to see the rest of the table. Now the original query is really easy:

```
SELECT P.emp_id, empname, badge_nbr
  FROM Personnel AS P, Badges AS B
 WHERE B.emp_id = P.emp_id
   AND B.badge_status = 'A';
```

Answer #2

Another approach is to assign a sequence number to each of the badges using the `MIN()` or `MAX()` sequence number as the active badge:

```
CREATE TABLE Badges
(badge_nbr INTEGER NOT NULL PRIMARY KEY,
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
 issued_date DATE NOT NULL,
 badge_seq INTEGER NOT NULL
   CHECK (badge_seq > 0),
 UNIQUE (emp_id, badge_seq),
 ..
 );
```

Now create a view to show the active badge:

```
CREATE VIEW ActiveBadges (emp_id, badge_nbr)
AS
SELECT emp_id, MAX(badge_nbr)
   FROM Badges
  GROUP BY emp_id;
```

This approach also needs to have an update to reset the sequence numbering when badges are lost or retired. It is not required for the queries, but people feel better if they see the sequence, and it makes it easier to find the number of badges per employee.

```
UPDATE Badges
   SET badge_seq
       = (SELECT COUNT(*)
            FROM Badges AS B1
           WHERE Badges.emp_id = B1.emp_id
             AND Badges.badge_seq = B1.badge_seq;
```

PUZZLE
5 ALPHA DATA

How do you ensure that a column will have a single alphabetic-character–only string in it? That means no spaces, no numbers, and no special characters within the string.

In older procedural languages, you have to declare data fields with format constraints in the file declarations. The obvious examples are COBOL and PL/I. Another approach is to use a template to filter the data as you read; the FORTRAN-style `FORMAT` statement is the best-known example.

SQL made a strong effort to separate the logical view of data from the physical representation of it, so you don't get much help with specifying the physical layout of your data. When a programmer at our little shop came to me with this one, I came up with some really bad first tries using substrings and `BETWEEN` predicates in a `CHECK()` clause that was longer than the whole schema declaration.

Answer #1

I keep telling people to think in terms of whole sets and not in a "record at a time" mind-set when they write SQL. The trick is to think in terms of whole strings and not in a "character at a time" mind-set. The answer from the folks at Ocelot software is surprisingly easy:

```
CREATE TABLE Foobar
(no_alpha VARCHAR(6) NOT NULL
            CHECK (UPPER(no_alpha) = LOWER(no_alpha)),
 some_alpha VARCHAR(6) NOT NULL
e_alpha)),
 all_alpha VARCHAR(6) NOT NULL
           CHECK (UPPER(all_alpha) <> LOWER(all_alpha)
                 AND LOWER (all_alpha)
                     BETWEEN 'aaaaaa' AND 'zzzzzz'),
  ...);
```

These `CHECK()` constraints assume that you are using Standard SQL-92 case sensitivity. Letters have different uppercase and lowercase values, but other characters do not. This lets us edit a column for no alphabetic characters and some alphabetic characters.

Answer #2

However, trying to find a string of all alphabetic characters is difficult without using a vendor extension, such as a regular expression parser in the `LIKE` predicate.

```
 all_alpha VARCHAR(6) NOT NULL
            CHECK (TRANSLATE (all_alpha USING
one_letter_translation)
                   = 'xxxxxx')
```

The `one_letter_translation` is a translation that maps all letters to 'x.' This is standard SQL, but it is not a common function yet. The syntax for creating a translation is:

```
<translation definition> ::=
   CREATE TRANSLATION <translation name>
     FOR <source character set specification>
       TO <target character set specification> FROM
<translation source>;
```

I will not discuss details here.

Answer #3

The regular expression predicate in standard SQL is based on the POSIX syntax, but you will probably have to look at vendor particulars for your product.

```
all_alpha VARCHAR(6) NOT NULL
        CHECK (all_alpha SIMILAR TO '[a-zA-Z]')
```

PUZZLE

6 HOTEL RESERVATIONS

Scott Gammans posted a version of the following problem on the WATCOM Forum on CompuServe. Suppose you are the clerk at Hotel SQL, and you have the following table:

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL,
 guest_name CHAR(30) NOT NULL,
 PRIMARY KEY (room_nbr, arrival_date),
 CHECK (departure_date >= arrival_date));
```

Right now the `CHECK()` clause enforces the data integrity constraint that you cannot leave before you have arrived, but you want more. How do you enforce the rule that you cannot add a reservation that has an arrival date conflicting with the prior departure date for a given room? That is, no double bookings.

Answer #1

One solution requires a product to have the capability of using fairly complex SQL in the `CHECK()` clause, so you will find that a lot of implementations will not support it.

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 arrival_date DATE NOT NULL,
 departure_date DATE NOT NULL,
 guest_name CHAR(30),
 PRIMARY KEY (room_nbr, arrival_date),
 CHECK (departure_date >= arrival_date),
 CHECK (NOT EXISTS
           (SELECT *
              FROM Hotel AS H1, Hotel AS H2
             WHERE H1.room_nbr = H2.room_nbr
               AND H1.arrival_date
                   BETWEEN H2.arrival_date AND
H2.departure_date)));
```

Answer #2

Another solution is to redesign the table, giving a row for each day and each room, thus:

```
CREATE TABLE Hotel
(room_nbr INTEGER NOT NULL,
 occupy_date DATE NOT NULL,
 guest_name CHAR(30) NOT NULL,
 PRIMARY KEY (room_nbr, occupy_date, guest_name));
```

This does not need any check clauses, but it can take up disk space and add some redundancy. Given cheap storage today, this might not be a problem, but redundancy always is. You will also need to find a way in the `INSERT` statements to be sure that you put in all the room days without any gaps.

As an aside, in full SQL-92 you will have an `OVERLAPS` predicate that tests to see if two time intervals overlap a temporal version of the `BETWEEN` predicate currently in SQL implementations. Only a few products have implemented it so far.

Answer #3

Lothar Flatz, an instructor for Oracle Software Switzerland, made the objection that the clause "`H1.arrival_date BETWEEN H2.arrival_date AND H2.depatures`" does not allow for a guest name to arrive the same day another guest name leaves.

Since Oracle cannot put subqueries into `CHECK()` constraints and triggers would not be possible because of the mutating table problem, he used a `VIEW` that has a `WITH CHECK OPTION` to enforce the occupancy constraints:

```
CREATE VIEW HotelStays (room_nbr, arrival_date,
departure_date, guest_name)
AS SELECT room_nbr, arrival_date, departure_date,
guest_name
     FROM Hotel AS H1
    WHERE NOT EXISTS
       (SELECT *
          FROM Hotel AS H2
         WHERE H1.room_nbr = H2.room_nbr
           AND H2.arrival_date < H1.arrival_date
```

```
            AND H1.arrival_date < H2.departure_date)
    WITH CHECK OPTION;
```

For example,

```
INSERT INTO HotelStays
VALUES (1, '2008-01-01', '2008-01-03', 'Coe');
COMMIT;
INSERT INTO HotelStays
VALUES (1, '2008-01-03', '2008-01-05', 'Doe');
```

will give a `WITH CHECK OPTION` clause violation on the second `INSERT INTO` statement. This is a good trick for getting table-level constraints in a table on products without full SQL-92 features.

PUZZLE
7

KEEPING A PORTFOLIO

Steve Tilson sent this problem to me in November 1995:

I have a puzzle for you. Perhaps I cannot see the forest for the trees here, but this seems like a real challenge to solve in an elegant manner that does not result in numerous circular references.

Although this puzzle seems to be an entire system, my question is whether or not there is a way to eliminate the apparent circular references within the table design phase.

Let's say you must keep track of Portfolios in an organization for lookup or recall. There are various attributes attached, but only certain items are pertinent to the puzzle:

```
CREATE TABLE Portfolios
(file_id INTEGER NOT NULL PRIMARY KEY,
 issue_date DATE NOT NULL,
 superseded_file_id INTEGER NOT NULL REFERENCES Portfolios
(file_id),
 supersedes_file_id INTEGER NOT NULL REFERENCES
Portfolios(file_id));
```

Here is the puzzle:

- You need to keep track of which portfolio superseded the current portfolio.
- You need to keep track of which portfolio this portfolio has superseded.
- You need to be able to reinstate a portfolio (which has the effect of superseding a portfolio or portfolio chain, which results in a circular reference).
- You can track the dates by virtue of the `issue_date`, but another thorny issue results if a portfolio is reinstated!
- You need to be able to `SELECT` the most current portfolio regardless of the portfolio in a `SELECT` statement.
- You need to be able to reproduce an audit trail for a chain of documents.

Answer #1

Steve is still thinking in terms of pointer chains and procedural languages. Shame on him! We know this is a problem that deals with ordinal numbering, because we have the give-away words "successor" and "predecessor" in the specification. Let's apply what we know about nested sets instead.

First, create a table to hold all the information on each file:

```
CREATE TABLE Portfolios
(file_id INTEGER NOT NULL PRIMARY KEY,
 other_stuff ..);
```

Then create a table to hold the succession of the documents, with two special columns, chain and next, in it.

```
CREATE TABLE Succession
(chain INTEGER NOT NULL,
 next INTEGER DEFAULT 0 NOT NULL CHECK (next >= 0),
 file_id INTEGER NOT NULL REFERENCES Portfolios(file_id),
 suc_date NOT NULL,
 PRIMARY KEY(chain, next));
```

Imagine that the original document is the zero point on a line.

The next document that supersedes `_file_id` is a circle drawn around the point. The third document in the chain of succession is a second circle drawn around the first circle, and so forth. We show these nested sets with the next value, flattening the circles onto the number line starting at zero.

You have to create the new document row in Portfolios, then the succession table entry. The value of next in the successor is one greater than the highest next value in the chain. Nested sets!!

Here is some sample data where a chain of '22?' and '32?' documents are superseded by a single document, 999.

```
CREATE TABLE Portfolios
(file_id INTEGER NOT NULL PRIMARY KEY,
 stuff CHAR(15) NOT NULL);

INSERT INTO Portfolios
VALUES (222, 'stuff'),
```

```
       (223, 'old stuff'),
       (224, 'new stuff'),
       (225, 'borrowed stuff'),
       (322, 'blue stuff'),
       (323, 'purple stuff'),
       (324, 'red stuff'),
       (325, 'green stuff'),
       (999, 'yellow stuff');

CREATE TABLE Succession
(chain INTEGER NOT NULL,
 next INTEGER NOT NULL,
 file_id INTEGER NOT NULL REFERENCES Portfolios(file_id),
 suc_date NOT NULL,
 PRIMARY KEY(chain, next));

INSERT INTO Succession
VALUES (1, 0, 222, '2007-11-01'),
       (1, 1, 223, '2007-11-02'),
       (1, 2, 224, '2007-11-04'),
       (1, 3, 225, '2007-11-05'),
       (1, 4, 999, '2007-11-25'),
       (2, 0, 322, '2007-11-01'),
       (2, 1, 323, '2007-11-02'),
       (2, 2, 324, '2007-11-04'),
       (2, 3, 322, '2007-11-05'),
       (2, 4, 323, '2007-11-12'),
       (2, 5, 999, '2007-11-25');
```

To answer your queries:

- You need to be able to SELECT the most current portfolio regardless of the portfolio in a SELECT statement.

```
SELECT DISTINCT P1.file_id, stuff, suc_date
  FROM Portfolios AS P1, Succession AS S1
 WHERE P1.file_id = S1.file_id
   AND next = (SELECT MAX(next)
                 FROM Succession AS S2
                WHERE S1.chain= S2.chain);
```

I have to use the SELECT DISTINCT option in case two or more chains were superseded by a single document.

- You need to be able to reproduce an audit trail for a chain of documents.

```
SELECT chain, next, P1.file_id, stuff, suc_date
  FROM Portfolios AS P1, Succession AS S1
 WHERE S1.file_id = P1.file_id
 ORDER BY chain, next;
```

- You need to keep track of which portfolio superseded this portfolio.

```
SELECT S1.file_id, ' superseded ',
       S2.file_id, ' on ', S2.suc_date
  FROM Succession AS S1, Succession AS S2
 WHERE S1.chain = S2.chain
   AND S1.next = S2.next + 1
   AND S1.file_id = :my_file_id;  -- remove for all
portfolios
```

- You need to be able to reinstate a portfolio, which has the effect of superseding a portfolio or portfolio chain, which results in a circular reference.

```
BEGIN
-- Create a row for the new document
INSERT INTO Portfolios VALUES (1000, 'sticky stuff');

-- adds new_file_id to chain with :old_file_id anywhere in
it.
INSERT INTO Succession (chain, next, file_id, suc_date)
VALUES ((SELECT DISTINCT chain
           FROM Succession AS S1
          WHERE S1.file_id = :old_file_id),
        (SELECT MAX(next) + 1
           FROM Succession AS S1
          WHERE S1.chain = (SELECT DISTINCT chain
                              FROM Succession AS S2
                             WHERE file_id = :my_file_id)),
        :new_file_id, :new_suc_date);
END;
```

The problem here is that I allowed for a single file to supersede more than one existing file and for more than one file to supersede a single existing file. My chains are not really all that linear. This code blows up if `:old_file_id` is in more than one chain. You can fix it by asking for the chain number or the file_id of the document that the new file supersedes `_file_id`, but the SQL is ugly and I don't have time to work it out right now. You can try it.

- You can track the dates by virtue of the issue date, but another thorny issue results if a portfolio is reinstated!

No big deal with this schema. Do a `SELECT` on any particular `file_id` and look at the dates and next column to get the chain of events. You did not say if the succession date column values have to be in increasing order, along with the next column values. Is that true? If so, we need to add another `CHECK()` clause to handle this.

PUZZLE

8 SCHEDULING PRINTERS

Yogesh Chacha ran into a problem and sent it to me on CompuServe on September 12, 1996. Users in his shop usually end up using the wrong printer for printout, thus they decided to include a new table in the system that will derive the correct printer for each user at runtime. Their table looked like this:

```
CREATE TABLE PrinterControl
(user_id CHAR(10), -- null means free printer
 printer_name CHAR(4) NOT NULL PRIMARY KEY,
 printer_description CHAR(40) NOT NULL);
```

The rules of operation are that:

1. If the user has an entry in the table, he will pick the corresponding `printer_name`.
2. If the user is not in the table then, he is supposed to use one of the printers whose `user_id` is NULL.

Now, consider the following example:

```
PrinterControl
user_id        printer_name    printer_description
=======================================================
'chacha'    'LPT1'         'First floor's printer'
'lee'       'LPT2'         'Second floor's printer'
'thomas'    'LPT3'         'Third floor's printer'
NULL        'LPT4'         'Common printer for new user'
NULL        'LPT5'         'Common printer for new user'
```

When `'chacha'` executes the report he is entitled to use only LPT1, whereas a user named `'celko'` is expected to use either LPT4 or LPT5. In the first case, a simple query can pull out one row and it works fine; in the second case, you get two rows and cannot use that result.

Can you come up with a one-query solution?

Answer #1

I would answer that the problem is in the data. Look at the `user_id` column. The name tells you that it should be unique, but it has multiple NULLs in it. There is also another problem in the real world; you want to balance the printer loads between LPT4 and LPT5, so that one of them is not overused.

Do not write a fancy query; change the table:

```
CREATE TABLE PrinterControl
(user_id_start CHAR(10) NOT NULL,
 user_id_finish CHAR(10) NOT NULL,
 printer_name CHAR(4) NOT NULL,
 printer_description CHAR(40) NOT NULL
 PRIMARY KEY (user_id_start, user_id_finish));
```

Now, consider the following example:

```
PrinterControl
user_id_start    user_id_finish   printer_name
printer_description
============================================================
'chacha'   'chacha'        'LPT1'   'First floor's printer'
'lee'      'lee'           'LPT2'   'Second floor's printer'
'thomas'   'thomas'        'LPT3'   'Third floor's printer'
'aaaaaaaa' 'mzzzzzzzz'     'LPT4'   'Common printer #1 '
'naaaaaaa' 'zzzzzzzz'      'LPT5'   'Common printer #2'
```

The query then becomes:

```
SELECT MIN(printer_name)
  FROM PrinterControl
 WHERE :my_id BETWEEN user_id_start AND user_id_finish;
```

The trick is in the start and finish values, which partition the range of possible strings between '`aaa`...' and '`zzz`...' any way you wish. The '`celko`' user id qualified only for LPT4 because it falls alphabetically within that range of strings. A user '`norman`' is qualified only for LPT5. Careful choice of these ranges will allow you to distribute the printer loads evenly if you know what the user ids are going to be like.

I have assumed the common printers always will have higher LPT numbers. When 'chacha' goes to this table, he will get a result set of (LPT1, LPT4), and then pick the minimum value, LPT1, from it. A smart optimizer should be able to use the PRIMARY KEY index to speed up the query.

Answer #2

Richard Romley came up with a different solution:

```
SELECT COALESCE(MIN(printer_name),
       (SELECT MIN(printer_name)
          FROM PrinterControl AS P2
         WHERE user_id IS NULL))
  FROM PrinterControl AS P1
 WHERE user_id = :user_id;
```

This is trickier than it looks. You go to the outer WHERE clause with user_id = 'celko', an unregistered user, so you would think that you don't get any rows from the P1 copy of PrinterControl.

This is not true. While a query like:

```
SELECT col
  FROM SomeTable
 WHERE 1 = 2;
```

will return no rows, a query like:

```
SELECT MAX(col)
  FROM SomeTable
 WHERE 1 = 2;
```

will return one row containing one column (col) that is NULL. This is a funny characteristic of the aggregate functions on empty sets. Therefore,

```
SELECT COALESCE(MAX(col), something_else)
  FROM SomeTable
 WHERE 1 = 2;
```

will work. The WHERE clause is used only to resolve MAX(col) and not to determine whether or not to return rows; that is the job of the SELECT

clause. The aggregate `MAX(col)` will be `NULL` and will get returned. Therefore, the `COALESCE()` will work.

The bad news is that when I get back a row, this query is going to return the same printer over and over, instead of distributing the workload over all the unassigned printers. You can add an update statement to replace the `NULL` with the guest user, so that the next printer will be used.

Answer #3

This can be fixed with a simple repair:

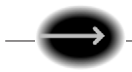
```
SELECT COALESCE(MIN(printer_name),
             (SELECT CASE
                     WHEN :user_id < 'n'
                     THEN 'LPT4'
                     ELSE 'LPT5' END
                FROM PrinterControl
               WHERE user_id IS NULL))
  FROM printer_control
  WHERE user_id = :user_id;
```

The flaw with this answer is that all common printer rows are being handled in the query, so you don't need them in the table at all. If you went that route, you would just remove everything after the `CASE` statement from the second query above. This would mean, however, that you never record information about the printers in the database. If you drop or add printers, you have to change the query, not the database where you are supposed to keep this information.

Answer #4

We can change the table design again to hold a flag for the type of printer:

```
CREATE TABLE PrinterControl
(user_id CHAR(10), -- null means free printer
 printer_name CHAR(4) NOT NULL PRIMARY KEY,
 assignable_flag CHAR(1) DEFAULT 'Y' NOT NULL
        CHECK (assignable_flag IN ('Y', 'N'),
 printer_description CHAR(40) NOT NULL);
```



We then update the table with:

```
UPDATE PrinterControl
   SET user_id = :guest_id
 WHERE printer_name
       = (SELECT MIN(printer_name)
            FROM PrinterControl
           WHERE assignable_flag = 'Y'
             AND user_id IS NULL);
```

Then you need to clear out the guest users at some point in time.

```
UPDATE PrinterControl
   SET user_id = NULL
 WHERE assignable_flag = 'Y';
```

PUZZLE
9 AVAILABLE SEATS

You have a restaurant with 1,000 seats. Whenever a waiter puts someone at a seat, he logs it in a table of seats (I was going to say "table of tables" and make this impossible to read). Likewise, when a guest finishes a meal, you remove the guest's seat number. You want to write a query to produce a list of the available seats in the restaurant, set up in blocks by their starting and ending seat numbers. Oh yes, the gimmick is that the database resides on a personal digital assistant and not a mainframe computer.

As part of the exercise, you must do this with the smallest amount of storage possible. Assume each seat number is an integer.

The first thought is to add a (free/occupied) flag column next to the seat-number column. The available seating query would be based on the flag. This would be 1,000 rows of one integer and one character for the whole restaurant and would work pretty well, but we have that minimal storage requirement. Darn!

Answer #1

The flag can be represented by a plus or minus on the seat number itself to save the extra column, but this is very bad relational practice; two attributes are being collapsed into one column. But it does keep us at 1,000 rows.

```
UPDATE Seats
   SET seat_nbr = -seat_nbr
 WHERE seat_nbr = :my_seat;
```

The same update statement can be used to put back into the Available list, and a "`SET seat_nbr = ABS(seat_nbr)`" will reset the restaurant at closing time.

Answer #2

The second thought is to create a second table with a single column of occupied seating and to move numbers between the occupied and available tables. That would require a total of 1,000 rows in both tables, which is a little weird, but it leads to the next answer.

Answer #3

Instead, we can use a single table and create seats 0 through 1,001 (0 and 1,001 do not really exist and are never assigned to a customer. They act as sentries on the edge of the real seating to make the code easier). Delete each seat from the table as it is occupied and insert it back when it is free again. The Restaurant table can get as small as the two dummy rows if all the seating is taken, but no bigger than 1,002 rows (2,004 bytes) if the house is empty.

This VIEW will find the first seat in a gap of open seats:

```
CREATE VIEW Firstseat (seat)
   AS SELECT (seat + 1)
        FROM Restaurant
       WHERE (seat + 1) NOT IN
       (SELECT seat FROM Restaurant)
       AND (seat + 1) < 1001;
```

Likewise, this `VIEW` finds the last seat in a gap of open seats:

```
CREATE VIEW Lastseat (seat)
   AS SELECT (seat - 1)
        FROM Restaurant
       WHERE (seat - 1) NOT IN
       (SELECT seat FROM Restaurant)
       AND (seat - 1) > 0;
```

Now, use these two `VIEW`s to show the blocks of empty seats:

```
SELECT F1.seat AS start, L1.seat AS finish.
          ((L1.seat - F1.seat) + 1) AS available
  FROM Firstseat AS F1, Lastseat AS L1
 WHERE L1.seat = (SELECT MIN(L2.seat)
                    FROM Lastseat AS L2
                   WHERE F1.seat <= L2.seat);
```

This query will also tell you how many available seats are in each block, a fact that could be handy for a waiter to know when seating groups. It is left as an exercise to the reader to write this as a single query without `VIEW`s.

Answer #4

Richard Romley combined the VIEWs into one query using the extended table with the rows 0 and 1,001 included:

```
 SELECT (R1.seat + 1) AS start,
        (MIN(R2.seat) - 1) AS finish
  FROM Restaurant AS R1
       INNER JOIN
       Restaurant AS R2
       ON R2.seat > R1.seat
 GROUP BY R1.seat
HAVING (R1.seat + 1) < MIN(R2.seat);
```

Answer #5

For variety you can use the new SQL-99 OLAP functions and get a bit more information:

```
SELECT X.seat_nbr, X.rn,
       (rn-seat_nbr) AS available_seat_cnt
  FROM (SELECT seat_nbr,
               ROW_NUMBER()
               OVER (ORDER BY seat_nbr)
          FROM Restaurant) AS X(seat_nbr, rn)
 WHERE rn <> seat_nbr;
```

The available_seat_cnt is the number of open seats less than the seat_nbr. This could be useful if the restaurant is broken into sections in some way.

PUZZLE
10 WAGES OF SIN

Luke Tymowski, a Canadian programmer, posted an interesting problem on the MS ACCESS Forum on CompuServe in November 1994. He was working on a pension fund problem. In SQL-92, the table involved would look like this:

```
CREATE TABLE Pensions
(sin CHAR(10) NOT NULL,
 pen_year INTEGER NOT NULL,
 month_cnt INTEGER DEFAULT 0 NOT NULL
     CHECK (month_cnt BETWEEN 0 AND 12),
 earnings DECIMAL (8,2) DEFAULT 0.00 NOT NULL);
```

The SIN column is the Social Insurance Number, which is something like the Social Security Number (SSN) used in the United States to identify taxpayers. The pen_year column is the calendar year of the pension, the `month_cnt` column is the number of months in that year the person worked, and earnings is the person's total earnings for the year.

The problem is to find the total earnings of each employee for the most recent 60 months of month_cnt in consecutive years. This number is used to compute the employee's pension. The shortest period going back could be 5 years, with 12 months in each year applying to the total `month_cnt`. The longest period could be 60 years, with 1 month in each year. Some people might work four years and not the fifth, and thus not qualify for a pension at all.

The reason this is a beast to solve is that "most recent" and "consecutive" are hard to write in SQL.

HINT: *For each employee in each year, insert a row even in the years in which the employee did not work. It not only makes the query easier, but you also have a record to update when you get in new information.*

Answer #1

This query will get me the starting and ending years of consecutive periods where (1) the employee worked (i.e., month_cnt greater than 0 months) and (2) the month_cnt totaled 60 or more.

```
CREATE VIEW PenPeriods (sin, start_year, end_year,
earnings_tot)
AS SELECT P0.sin, P0.pen_year, P1.pen_year,
   (SELECT SUM (earnings) -- total earnings for period
     FROM Pensions AS P2
    WHERE P2.sin = P0.sin
      AND P2.pen_year BETWEEN P0.pen_year AND P1.pen_year)
     FROM Pensions AS P0, Pensions AS P1
     WHERE P1.sin = P0.sin   -- self-join to make intervals
       AND P1.pen_year >= (P0.pen_year - 4) -- why sooner?
       AND 0 < ALL (SELECT month_cnt  -- consecutive months
                      FROM Pensions AS P3
                      WHERE P3.sin = P0.sin
                        AND P3.pen_year
                       BETWEEN P0.pen_year AND P1.pen_year)
      AND 60 <= (SELECT SUM (month_cnt) -- total more than
60
                   FROM Pensions AS P4
                  WHERE P4.sin = P0.sin
                    AND P4.pen_year
                        BETWEEN P0.pen_year AND
P1.pen_year);
```

The subquery expression in the SELECT list is a SQL-92 trick, but a number of products already have it.

The gimmick is that this will give you all the periods of 60 months or more. What we really want is the most recent end_year. I would handle this with the pension period view I just defined and a MAX(end_year) predicate:

```
SELECT *
  FROM PenPeriods AS P0
 WHERE end_year = (SELECT MAX(end_year)
                   FROM PenPeriods AS P1
                   WHERE P1.sin = P0.sin);
```

I can handle that with some ugly HAVING clauses in SQL-92, I could combine both those subquery predicates with an EXISTS clause, and so on.

As an exercise, you can try to add another predicate to the final subquery that says there does not exist a year between P0.pen_year and

P1.pen_year that is greater than the P4.pen_year and still gives a total of 60 or more consecutive months.

Answer #2

Most of the improved solutions I got via CompuServe were based on my original answer. However, Richard Romley sent in the best one and used a completely different approach. His answer used three copies of the Pensions table, ordered in time as P0, P1, and P2.

```
SELECT P0.sin,
       P0.pen_year AS start_year,
       P2.pen_year AS end_year,
       SUM (P1.earnings)
  FROM Pensions AS P0, Pensions AS P1, Pensions AS P2
 WHERE P0.month_cnt > 0
   AND P1.month_cnt > 0
   AND P2.month_cnt > 0
   AND P0.sin = P1.sin
   AND P0.sin = P2.sin
  AND P0.pen_year BETWEEN P2.pen_year-59 AND (P2.pen_year -
4)
   AND P1.pen_year BETWEEN P0.pen_year AND P2.pen_year
 GROUP BY P0.sin, P0.pen_year, P2.pen_year
HAVING SUM (P1.month_cnt) >= 60
   AND (P2.pen_year - P0.pen_year) = (COUNT (*) - 1);
```

Mr. Romley wrote: "This problem would have been easier if there were no rows allowed with (month_cnt = 0). I had to waste three WHERE clauses just to filter them out!" Another example of how a better data design would have made life easier!

The outstanding parts of this answer are the use of the BETWEEN predicate to look at durations in the range of 5 to 60 years (the minimum and maximum time needed to acquire 60 months of month_cnt) and the use of grouping columns in the last expression in the HAVING clause to guarantee consecutive years.

When I ran this query on WATCOM SQL 4.0, the query planner estimate was four times greater for Mr. Romley's solution than for my original solution, but his actually ran faster. I would guess that the plan estimation is being fooled by the three-way self-joins, which are usually very expensive.

Answer #3

In 1999, Dzavid Dautbegovic sent in the following response to the first edition of this book:

"Both solutions are excellent in their own way (your SQL-92, Richard's SQL-89). I must confess that my solution is too complicated and totally inelegant but much closer to Richard's answer. For me the biggest problem was to get sum of earnings in the first position. But I think that you need to change your second select if you want most recent end_year and only one solution per SIN. For this to work, we need to make a `VIEW` from Richard's solution."

```
SELECT PO.sin, PO.end_year,
       MAX(PO.start_year) AS laststart_year,
       MIN(PO.sumofearnings) AS minearnings,
       MIN(PO.sumofmonth_cnt) AS minmonth_cnt,
       MIN(PO.start_year) AS firststart_year,
       MAX(PO.sumofearnings) AS maxearnings,
       MAX(PO.sumofmonth_cnt) AS maxmonth_cnt
  FROM PensionsView AS PO
 WHERE end_year = (SELECT MAX(end_year)
                     FROM Pensions AS P1
                    WHERE P1.sin = PO.sin)
 GROUP BY PO.sin, PO.end_year;
```

Answer #4

Phil Sherman in April 2006 pointed out that this is an interesting problem because the answer should almost always be indeterminate.

How can you determine the most recent 60 months' salary in the following `CASE`? An employee works for 10 full years starting in January of the first year and six months in the final, 11th year. The most recent 60 months start in the middle of a year, in July. There is no data available in the database to show the salary earned during the first six months of the 60-month period. An average monthly salary for that year could be used, but that would not properly account for a pay raise that occurred in July of the first year used for the calculation.

This issue will occur any time the number of months needed to make 60 is less than the number of months worked in the earliest year that is used to build the 60-month period. The problem is worse for hourly workers who work different numbers of hours in different months.

Alan Samet sent in the following solution, under the assumption that getting the 60 months in consecutive years was pretty straightforward. However, he could not avoid using a subquery to find the most recent of those 60-month blocks. The subquery itself includes the results you're looking for, only it does not reduce it to the most recent block of years. He also added a column to the results to adjust the first year's earnings for the percentage that could apply to the pension (i.e., Person works 6 years, for a total of 71 months, subtract the first year's earnings * (11 / 12) from the `earnings_tot`). Here is his solution, with a Common Table Expression (CTE):

```
SELECT *,
       MOD(total_month_cnt,12) AS
nonutilized_first_year_month_cnt,
      (total_month_cnt % 12) / (first_year_month_cnt * 1.0)

             *first_year_earnings AS first_year_adjustment,
       earnings_tot - (MOD(total_month_cnt, 12))/
(first_year_month_cnt * 1.0)
         * first_year_earnings AS adjusted_earnings_tot
  FROM (SELECT P1.sin, P1.pen_year AS first_year
              P2.pen_year AS last_year,
              P1.month_cnt AS First_year_month_cnt,
              P1.earnings AS first_year_earnings,
              COUNT(P3.pen_year) AS year_cnt,
              SUM(P3.month_cnt) AS total_month_cnt,
              SUM(P3.earnings) AS earnings_tot,
         FROM Pensions AS P1
              INNER JOIN Pensions AS P2
              ON P1.sin = P2.sin
                INNER JOIN Pensions AS P3
                ON P1.sin = P3.sin
        WHERE P3.pen_year BETWEEN P1.pen_year
                          AND P2.pen_year
          AND P3.month_cnt > 0
        GROUP BY P1.sin, P1.pen_year, P2.pen_year,
P1.month_cnt, P1.earnings
      HAVING COUNT(P3.pen_year) = P2.pen_year - P1.pen_year
+ 1
              AND SUM(P3.month_cnt) BETWEEN 60 AND 60 +
P1.month_cnt - 1
       ) AS A;
 WHERE A.last_year
```

```
       = (SELECT MAX(last_year)
            FROM (SELECT P2.pen_year AS last_year
                    FROM Pensions AS P1
                         INNER JOIN Pensions AS P2
                         ON P1.sin = P2.sin
                           INNER JOIN Pensions AS P3
                           ON P1.sin = P3.sin
           WHERE P3.pen_year BETWEEN P1.pen_year AND
P2.pen_year
             AND P3.month_cnt > 0
             AND P1.sin = A.sin
           GROUP BY P1.sin, P1.pen_year, P2.pen_year,
P1.month_cnt
          HAVING COUNT(P3.pen_year) = P2.pen_year -
P1.pen_year + 1
             AND SUM(P3.month_cnt) BETWEEN 60 AND 60 +
P1.month_cnt - 1
         ) AS B
  );

WITH A(sin, first_year, last_year, first_year_month_cnt,
first_year_earnings, year_cnt, total_month_cnt,
earnings_tot)
AS (SELECT P1.sin, ROW_NUMBER() OVER (ORDER BY P1.sin),
           P2.pen_year, P1.pen_year, P2.pen_year,
           P1.month_cnt, P1.earnings,
           COUNT(P3.pen_year),
           SUM(P3.month_cnt), SUM(P3.earnings)
      FROM Pensions AS P1
           INNER JOIN Pensions AS P2
           ON P1.sin = P2.sin
              INNER JOIN Pensions ON P3
              ON P1.sin = P3.sin
     WHERE P3.pen_year BETWEEN P1.pen_year AND P2.pen_year
                           AND P3.month_cnt > 0
     GROUP BY P1.sin, P1.pen_year, P2.pen_year,
P1.month_cnt, P1.earnings
   HAVING COUNT(P3.pen_year) = P2.pen_year - P1.pen_year + 1
       AND SUM(P3.month_cnt) BETWEEN 60 AND 60 +
P1.month_cnt - 1
   )

SELECT sin, earnings_tot
  FROM A AS Parent
```

```
 WHERE NOT EXISTS
          (SELECT *
             FROM A
            WHERE sin = Parent.sin
              AND row_number > parent.row_number);
```

Dave Hughes came to the same answer. However, trying to find a way to limit the cumulative SUMs to range over consecutive runs of rows with (`month_cnt > 0`) seems to be impossible (at least in DB2s implementation). Assume one adds a "reset" column, which is `'Y'` when (`month_cnt = 0`) and `'N'` otherwise:

```
WITH P
AS (SELECT sin, pen_year, month_cnt,
           CASE month_cnt WHEN 0
           THEN 'Y' ELSE 'N' END AS month_reset,
           earnings
      FROM Pensions)
...
```

Unfortunately, this does not help much: you can not partition on (`sin, month_reset`) as you will wind up with nonconsecutive runs on `pen_year`. The rows and range clauses of the OLAP functions would not help either because we are not dealing with a fixed offset of `pen_year`.

If the aggregation-window could be limited by a search-condition instead of fixed row or key offsets it would be easy, but it does not look like that's possible. Hence, I do not think OLAP functions are the answer here.

Eventually I hit on the same idea that Paul used (joining the table to itself a couple of times, one to form the start of the range, another to form the end of the range, and a third to aggregate across the range). Naturally I ran into the same problem as Paul mentions in his post: that you wind up with several potential ranges of consecutive years that have at least 60 months, and you only want the last one.

Instead of using the `ROW_NUMBER()` function, Dave just used a second subquery with `MAX()` to fix this:

```
WITH Ranges (sin, first_year, last_year, earnings_tot)
AS (SELECT P1.sin, P1.pen_year, P2.pen_year,
           SUM(P3.earnigns) AS earnigns_tot
      FROM Pensions AS P1
```

```
           INNER JOIN Pensions AS P2
           ON P1.sin = P2.sin
              INNER JOIN Pensions AS P3
              ON P1.sin = P3.sin
     WHERE P3.pen_year BETWEEN P1.pen_year AND P2.pen_year
       AND P3.month_cnt > 0
      GROUP BY P1.sin, P1.pen_year, P2.pen_year,
P1.month_cnt
     HAVING SUM(P3.month_cnt) BETWEEN 60 AND 60 +
P1.month_cnt - 1
    AND COUNT(P3.pen_year) = P2.pen_year - P1.pen_year + 1),

LastRange (sin, last_year)
AS (SELECT sin, MAX(last_year)
      FROM Ranges
     GROUP BY sin)

SELECT R.*
  FROM Ranges AS R
       INNER JOIN LastRange AS L
       ON R.sin = L.sin
          AND R.last_year = L.last_year
```

Self-joined solution is more complete given that it includes the columns that would be required to normalize the result by excluding excess months from the first year, which is missing in Dave's answer.

Andrey Odegov pointed out that I had an answer in the January 1998 issue of *DBMS* magazine (http://www.dbmsmag.com/9801d06.html), but it lacked the current features. His rewrite of that query is:

```
WITH P(sin, pen_year, earnings)
AS(SELECT P1.sin, P1.pen_year, P1.earnings
     FROM Pensions AS P1, Pensions AS P2
    WHERE P1.sin = P2.sin
      AND P1.pen_year <= P2.pen_year
    GROUP BY P1.sin, P1.pen_year, P1.month_cnt, P1.earnings
   HAVING SUM(P2.month_cnt) - P1.month_cnt < 60)

SELECT sin, MIN(pen_year) AS set_year,
            MAX(pen_year) AS end_year,
       SUM(earnings) AS earnings_tot
  FROM P
 GROUP BY sin;
```

PUZZLE
11 WORK ORDERS

Cenk Ersoy asked this question on the Gupta Forum on CompuServe. In a factory, a project is described in a work order, which has a series of steps that it must go through. A `step_nbr` on the work order is either completed or awaiting the completion of one or more of the steps that come before it. His table looks like this:

```
CREATE TABLE Projects
(workorder_id CHAR(5) NOT NULL,
 step_nbr INTEGER NOT NULL CHECK (step_nbr BETWEEN 0 AND
1000),
 step_status CHAR(1) NOT NULL
    CHECK (step_status IN ('C', 'W')), -- complete, waiting
 PRIMARY KEY (workorder_id, step_nbr));
```

With some sample data like this:

```
Projects
workorder_id  step_nbr step_status
=================================
'AA100'         0       'C'
'AA100'         1       'W'
'AA100'         2       'W'
'AA200'         0       'W'
'AA200'         1       'W'
'AA300'         0       'C'
'AA300'         1       'C'
```

He would like to get the work orders where the `step_nbr` is zero and the step_status is '`C`', but all other legs for that work order have a `step_status` of 'W'. For example, the query should return only '`AA100`' in the sample data.

Answer #1

This is really fairly straightforward, but you have to reword the query specification into the passive voice to see the answer. Instead of saying, "all other legs for that work order have `step_status` of `Waiting`,"

instead say "Waiting is the step_status of all the nonzero legs," and the answer falls out immediately, thus:

```
SELECT workorder_id
 FROM Projects AS P1
WHERE step_nbr = 0
 AND step_status = 'C'
 AND 'W' = ALL (SELECT step_status
                 FROM Projects AS P2
                WHERE step_nbr <> 0
                 AND P1.workorder_id = P2.workorder_id);
```

Answer #2

Another rewording would be to say that we are looking for a work order group (i.e., a group of step_nbrs) that has certain properties in the step_status column for certain step_nbrs. Using a characteristic function in a SUM() will let us know if all the elements of the group meet the criteria; if they all do, then the count of the characteristic function is the size of the group.

```
SELECT workorder_id
  FROM Projects
 GROUP BY workorder_id
HAVING SUM(CASE
           WHEN step_nbr <> 0 AND step_status = 'W' THEN 1
           WHEN step_nbr = 0 AND step_status = 'C' THEN 1
           ELSE 0 END) = COUNT (step_nbr);
```

or if you do not have a CASE expression, you can use some algebra:

```
SELECT workorder_id
 FROM Projects AS P1
GROUP BY workorder_id
HAVING SUM(
     ((1-ABS(SIGN(step_nbr)) * POSITION('W' IN
step_status))
    + ((1-ABS(step_nbr)) * POSITION('C' IN step_status))
     ) = COUNT(step_nbr);
```

Since this query involves only one copy of the table and makes a single pass through it, it should be as fast as possible. There are also subtle

optimization tricks in the CASE expression. The CASE expression's WHEN clauses are tested in the order they appear, so you should arrange the WHEN clauses in order from mostly likely to occur to least likely to occur.

While not required by the standard, the terms of an AND predicate will also often execute in the order of their appearance when they are all join predicates (i.e., involve columns from two tables) or are all search arguments (i.e., a column from one table compared to a constant value—also called SARGs in the literature). Therefore you can put the SARG with the smallest datatype first to improve performance; integers compare faster than long CHAR(n).

Answer #3

A version of the second answer that avoided the subquery was sent by Mr. Francisco Moreno of Columbia, South America. The original version used the Oracle DECODE() function, but his query would translate into SQL-92 like this:

```
SELECT workorder_id
 FROM Projects
 GROUP BY workorder_id
HAVING COUNT(*) -- total rows in the workorder_id
    = COUNT(CASE WHEN leg || step_status = 'C'
           THEN 1
           ELSE NULL END) -- total 0 & 'C' rows
    + COUNT(CASE WHEN step_status = 'W'
           THEN 1
           ELSE NULL END); -- total 'W' rows
```

This puts the COUNT(*) on one side of a comparison operator and not in an expression. This can be a help for some optimizers.

One of my students (Stephan Gneist) found a nice, simple solution:

```
SELECT workorder_id
  FROM Projects
 WHERE step_status = 'C'
 GROUP BY workorder_id
HAVING SUM(step_nbr) = 0;
```

This solution exploits the NOT NULL and CHECK() constraints of the table definition and does not require any join. This illustrates the point that SQL is a combination of both DDL and DML.

PUZZLE
12 CLAIMS STATUS

Leonard C. Medal posted this problem on CompuServe. Patients make legal claims against a medical institution, and we record it in the Claims table:

```
Claims
claim_id patient_name
=====================
  10     'Smith'
  20     'Jones'
  30     'Brown'
```

Each claim has one or more defendants, usually physicians, recorded in the table `'Defendants'`:

```
Defendants
claim_id defendant_name
=======================
   10    'Johnson'
   10    'Meyer'
   10    'Dow'
   20    'Baker'
   20    'Meyer'
   30    'Johnson'
```

Each defendant associated with a claim has a history of legal events, where changes in the claim status of the defendant on a given claim are recorded:

```
LegalEvents
claim_id  defendant_name claim_status  change_date
================================================
   10    'Johnson'    'AP'   '1994-01-01'
   10    'Johnson'    'OR'   '1994-02-01'
   10    'Johnson'    'SF'   '1994-03-01'
   10    'Johnson'    'CL'   '1994-04-01'
   10    'Meyer'      'AP'   '1994-01-01'
   10    'Meyer'      'OR'   '1994-02-01'
```

```
10    'Meyer'      'SF'   '1994-03-01'
10    'Dow'        'AP'   '1994-01-01'
10    'Dow'        'OR'   '1994-02-01'
20    'Meyer'      'AP'   '1994-01-01'
20    'Meyer'      'OR'   '1994-02-01'
20    'Baker'      'AP'   '1994-01-01'
30    'Johnson'    'AP'   '1994-01-01'
```

Changes in claim status for each defendant occur in a known sequence, determined by law, as shown in the Claims status table:

```
ClaimStatusCodes
claim_status claim_status_desc claim_seq
=======================================
'AP'   'Awaiting review panel'     1
'OR'   'Panel opinion rendered'    2
'SF'   'Suit filed'                3
'CL'   'Closed'                    4
```

The claim status of a defendant (with regard to a given claim) is his or her latest claim status, which is the claim status with the highest claim sequence number. For certain legal reasons, legal events ordered by date do not always correspond to legal events ordered by claim sequence number.

The claim status of a claim is the claim status of the defendant having the lowest claim status of all the defendants involved in the claim. This makes the claim status a minimum of the maximums. For this sample data, the answer would be:

```
claim_id patient_name claim_status
==================================
  10   'Smith'   'OR'
  20   'Jones'   'AP'
  30   'Brown'   'AP'
```

The problem is to find the claim status of each claim and display it.

Answer #1

Mr. Medal's answer was a single SQL query that directly translated the description into code:

```
SELECT C1.claim_id, C1.patient, S1.claim_status
  FROM Claims AS C1, ClaimStatusCodes AS S1
 WHERE S1.claim_seq
       IN (SELECT MIN(S2.claim_seq)
             FROM ClaimStatusCodes AS S2
            WHERE S2.claim_seq
                  IN (SELECT MAX(S3.claim_seq)
                        FROM LegalEvents AS AS E1,
ClaimStatusCodes AS S3
                       WHERE E1.claim_status =
S3.claim_status
                         AND E1.claim_id = C1.claim_id
                       GROUP BY E1.defendant));
```

Answer #2

But there is another solution. It is easier to get the set of all claim status codes within a claim that all the defendants have obtained:

```
SELECT E1.claim_id, C1.patient, E1.claim_status
  FROM LegalEvents AS E1, Claims AS C1
 WHERE E1.claim_id = C1.claim_id
 GROUP BY E1.claim_id, C1.patient, E1.claim_status
HAVING COUNT(*) = (SELECT COUNT(DISTINCT defendant)
                     FROM LegalEvents AS E2
                    WHERE E1.claim_id = E2.claim_id)
 ORDER BY E1.claim_id;
```

But more to the point, look at the `ClaimStatusCodes` table and you will see that claim status and claim sequence numbers are both keys that point to the claim status description (actually, all three columns are keys). This is a code translation table.

If we dropped the two-letter claim status code domain and replaced it with the numeric claim sequence in all the places it occurs, we would get a result from which we can easily pick the maximum value of the claim sequence column in each claim. A quick way to do this translation is with a scalar subquery in the `SELECT` clause:

```
SELECT E1.claim_id, C1.patient, (SELECT claim_seq
                                   FROM ClaimStatusCodes AS S1
                                    WHERE S1.claim_status =
E1.claim_status)
```

```
  FROM LegalEvents AS E1, Claims AS C1
 WHERE E1.claim_id = C1.claim_id
 GROUP BY E1.claim_id, C1.patient, E1.claim_status
HAVING COUNT(*) = (SELECT COUNT(DISTINCT defendant)
                     FROM LegalEvents AS E2
                    WHERE E1.claim_id = E2.claim_id)
 ORDER BY E1.claim_id;
```

Answer #3

Sorin Shtirbu submitted a third answer:

```
SELECT claim_id, MAX(patient), MAX(T1.claim_status)
  FROM ClaimStatusCodes AS T1, Claims AS C1
 WHERE T1.claim_seq =
       (SELECT MAX(T2.claim_seq)
          FROM ClaimStatusCodes AS T2, LegalEvents AS E1
         WHERE T2.claim_status = E1.claim_status
           AND E1.claim_id = C1.claim_id
         GROUP BY E1.claim_id, defendant)
  GROUP BY claim_id, T1.claim_seq
 HAVING T1.claim_seq = MIN(T1.claim_seq)
```

It is a good question if this is more efficient or not. It might depend on the implementation: subquery versus a `GROUP BY`.

Answer #4

This answer came from Francisco Moreno, and it is designed to avoid subqueries by using the `JOIN` syntax of SQL-92.

Step 1
Insert a dummy row in the `claim_status` table:

```
INSERT INTO Claim_status (claim_status, claim_status_desc,
claim_seq)
VALUES ('XX', 'Dummy', 5);
```

Step 2
The query:

```
SELECT C1.claim_id, C1.patient,
        CASE MIN(S1.claim_seq)
         WHEN 2 THEN 'AP'
         WHEN 3 THEN 'OR'
         WHEN 4 THEN 'SF'
         ELSE 'CL' END
 FROM
  ((Claims AS C1
    INNER JOIN
    Defendants s AS D1
    ON C1.claim_id = D1.claim_id)
   CROSS JOIN
   Claim_status AS S1)
 LEFT OUTER JOIN
 LegalEvents AS E1
 ON C1.claim_id = E1.claim_id
    AND D1.'[ = E1.defendant
    AND S1.claim_status = E1.claim_status
 WHERE E1.claim_id IS NULL
 GROUP BY C1.claim_id, C1.patient;
```

PUZZLE
13 TEACHERS

Brendan Campbell posted an interesting problem on the Oracle User Group Forum on CompuServe in May 1996. He gave me permission to use it and to publish his alternative PL/SQL solution as a bad example, thus submitting himself to public humiliation and disgrace for the good of science. Donating your body is easy—you're dead—but donating your dignity is hard.

We want a query to pass to a report program that shows the names of the teachers for each course and each student. Now here's the catch: we physically have room for only two teacher names on the printout.

If there is only one teacher, display that teacher's name in the first `teacher_name` column and set the second column to blanks or NULL. If there are exactly two teachers, display both names in ascending order. If there are more than two teachers, the report will display the name of the first teacher in the first column and the string `'--more--'` in the second `teacher_name` column.

Assume the necessary data is in a table like this:

```
CREATE TABLE Register
(course_nbr INTEGER NOT NULL,
 student_name CHAR(10) NOT NULL,
 teacher_name CHAR(10) NOT NULL,
  ..);
```

Brenden's original solution was 70 lines long, while the pure SQL answer is about 12 lines of code in a single statement.

Answer #1

One method is to use the extrema functions and `UNION`s.

```
SELECT R1.course_nbr, R1.student_name,
MIN(R1.teacher_name), NULL
  FROM Register AS R1
 GROUP BY R1.course_nbr, R1.student_name
HAVING COUNT(*) = 1
UNION
```

```
SELECT R1.course_nbr, R1.student_name,
MIN(R1.teacher_name),
       MAX(R1.teacher_name)
  FROM Register AS R1
 GROUP BY R1.course_nbr, R1.student_name
HAVING COUNT(*) = 2
UNION
SELECT R1.course_nbr, R1.student_name,
MIN(R1.teacher_name), '--More--'
  FROM Register AS R1
 GROUP BY R1.course_nbr, R1.student_name
HAVING COUNT(*) > 2;
```

Now, to go into my usual painful details:

The first `SELECT` statement picks the `course_nbr/student_name` combinations with one and only one teacher_name. See why the `MIN()` works?

Without any competition, the only teacher_name will be the minimum by default. I like to use NULLs for missing values, but you could use a string constant instead.

The second `SELECT` statement picks the course_nbr/student_name combinations with two and only two teachers. The `MIN()` and `MAX()` functions work and order the names because there are only two teachers.

The third `SELECT` statement picks the `course_nbr/student_name` combinations with more than two teachers. I use the `MIN()` to get the first `teacher_name`, then a constant of `'more'` as per the report specification, in the second column.

The reason for this solution is that the original problem was given for a situation in Oracle and Oracle lacks certain SQL-92 features.

Answer #2

Richard Romley once more cooked my published solution by collapsing the two `SELECT` statements into a `CASE` expression with the extrema functions in SQL-92 syntax, like this:

```
SELECT course_nbr, student_name, MIN(teacher_name),
       CASE COUNT(*) WHEN 1 THEN NULL
                     WHEN 2 THEN MAX(teacher_name)
                     ELSE '--More--' END
   FROM Register
  GROUP BY course_nbr, student_name;
```

This version of the CASE expression can also be replaced with syntax, which is equivalent to:

```
CASE WHEN COUNT(*) = 1 THEN NULL
     WHEN COUNT(*) = 2 THEN MAX(teacher_name)
     ELSE '--More--' END
```

You will find that a CASE expression in the SELECT list is very handy for a display problem, as you will see in the next answer.

Answer #3

This is a bad way to lay out a report; what we really want is a list of all teachers, but without the course_nbr and student_name information repeated on each line. That is, those columns should be blank after the first line. This is easy in COBOL or any report writer. In SQL-92, it looks like this:

```
SELECT CASE WHEN teacher_name=(SELECT MIN(teacher_name)
FROM Register AS R1
        WHERE R1.course_nbr = R0.course_nbr
        AND R1.student_name = R0.student_name)
            THEN course_nbr
            ELSE '    ' END AS course_nbr_hdr,
        CASE WHEN teacher_name = (SELECT MIN(teacher_name)
                    FROM Register AS R1
                    WHERE R1.course_nbr = R0.course_nbr
            AND R1.student_name = R0.student_name)
            THEN student_name
            ELSE '    ' END AS student_name_hdr,
        teacher_name
   FROM Register
  ORDER BY teacher_name;
```

Having given this code, it is a bad idea. You are destroying First Normal Form (1NF) and doing something that should be done in the front end and not the database.

PUZZLE
14 TELEPHONE

Suppose you are trying to set up an office telephone directory with your new database publishing system, and you have the following tables:

```
CREATE TABLE Personnel
(emp_id INTEGER PRIMARY KEY,
 first_name CHAR(20) NOT NULL,
 last_name CHAR(20) NOT NULL);

CREATE TABLE Phones
(emp_id INTEGER NOT NULL,
 phone_type CHAR(3) NOT NULL
         CHECK (phone_type IN ('hom', 'fax')),
 phone_nbr CHAR(12) NOT NULL,
 PRIMARY KEY (emp_id, phone_type),
 FOREIGN KEY emp_id REFERENCES Personnel(emp_id));
```

The codes '`hom`' and '`fax`' indicate whether the number is the employee's home phone number or a fax number. You want to print out a report with one line per employee that gives both numbers, and shows a NULL if either or both numbers are missing.

I should note here that the FOREIGN KEY constraint on the Phones table means that you cannot list a telephone number for someone who is not an employee. The PRIMARY KEY looks a bit large until you stop and think about all the cases. Married personnel could share the same fax or home telephones, and a single line could be both voice and fax services.

Answer #1

You can do a lot of things wrong with this query. The first thought is to construct the home telephone information as a query in its own right. Because you want to see all the personnel, you need an `OUTER JOIN`:

```
CREATE VIEW Home_phones (last_name, first_name, emp_id,
home_phone)
AS SELECT E1.last_name, E1.first_name, E1.emp_id,
H1.phone_nbr
     FROM (Personnel AS E1
           LEFT OUTER JOIN
```

```
            Phones AS H1
            ON E1.emp_id = H1.emp_id
              AND H1.phone_type = 'hom');
```

Likewise, you could construct the fax information as a query, using the same approach:

```
CREATE VIEW Fax_phones (last_name, first_name, emp_id,
fax_phone)
AS SELECT E1.last_name, E1.emp_id, F1.phone_nbr
     FROM (Personnel AS E1
           LEFT OUTER JOIN
           Phones AS F1
           ON E1.emp_id = F1.emp_id
              AND F1.phone_type = 'fax');
```

It would seem reasonable to combine these two VIEWs to get:

```
SELECT H1.last_name, H1.first_name, home_phone, fax_phone
  FROM HomePhones AS H1, FaxPhones AS F1
  WHERE H1.emp_id = F1.emp_id;
```

But this does not work because it leaves out the "fax only" people. If you want to preserve both phone tables, you need a FULL OUTER JOIN, which might look like this:

```
SELECT H1.last_name, H1.first_name, home_phone, fax_phone
  FROM HomePhones AS H1
       FULL OUTER JOIN
       FaxPhones AS F1
       ON H1.emp_id = F1.emp_id;
```

But this still does not print the names of the "fax only" people who show up as NULLs because you are only printing the Home_phones people's names. The COALESCE() function will take care of that problem for you. It will take a list of expressions and return the first non-NULL value that it finds, reading from left to right:

```
SELECT COALESCE (H1.last_name, F1.last_name),
       COALESCE (H1.first_name, F1.first_name),
       home_phone, fax_phone
```

```
  FROM HomePhones AS H1
       FULL OUTER JOIN
       FaxPhones AS F1
       ON H1.emp_id = F1.emp_id;
```

Answer #2

The bad news with the previous solution is that this will work, but it will run like glue because it will probably materialize the `VIEW`s before using them. The real trick is to go back and see that the `FaxPhones` and `HomePhones VIEW`s are outer-joined to the `Personnel` table. You can factor out the `Personnel` table and combine the two `FROM` clauses to give:

```
SELECT E1.last_name, E1.first_name,
       H1.phone_nbr AS Home,
       F1.phone_nbr AS FAX
  FROM (Personnel AS E1
        LEFT OUTER JOIN
        Phones AS H1
        ON E1.emp_id = H1.emp_id
           AND H1.phone_type = 'hom)
       LEFT OUTER JOIN
       Phones AS F1
       ON E1.emp_id = F1.emp_id
          AND F1.phone_type = 'fax';
```

Because this gets all the tables at once, it should run a good bit faster than the false start. This is also a query that you cannot write easily using Sybase-, Oracle-, or Gupta-style extended equality `OUTER JOIN` syntax because that syntax cannot handle nesting of `OUTER JOIN`s.

Answer #3

Kishore Ganji, a database analyst at Liz Claiborne Cosmetics, proposed a solution that avoids the two views altogether. He originally gave his answer in Oracle, but it can be translated into SQL-92:

```
SELECT E1.emp_id, E1.first_name, E1.last_name,
       MAX (CASE WHEN P1.phone_type = 'hom'
                 THEN P1.phone_nbr
                  ELSE NULL) AS home_phone,
```

```
       MAX (CASE WHEN P1.phone_type = 'fax'
                 THEN P1.phone_nbr
                 ELSE NULL) AS fax_phone
  FROM Personnel AS E1
       LEFT OUTER JOIN
       Phones AS P1
       ON P1.emp_id = E1.emp_id
 GROUP BY E1.emp_id, E1.first_name, E1.last_name;
```

The CASE expression positions the telephone number in its proper column, then the MAX() and GROUP BY, consolidating them into one row in the result table.

Answer #4

This answer came from Francisco Moreno of Columbia:

```
SELECT P1.last_name, P1.first_name,
       (SELECT T1.phone_nbr
          FROM Phones AS T1
         WHERE T1.emp_id = P1.emp_id
           AND T1.phone_type = 'hom') AS home_phone,
       (SELECT T2.phone_nbr
          FROM Phones AS T2
         WHERE T2.emp_id = P1.emp_id
           AND T2.phone_type = 'fax') AS fax_phone
  FROM Personnel AS P1;
```

However, the scalar subqueries will not perform very well.

PUZZLE
15 FIND THE LAST TWO SALARIES

Jack Wells sent this perplexing SQL problem in June 1996 over CompuServe. His situation is pretty typical for SQL programmers who have to work with 3GL people. The programmers are writing a report on the employees, and they want to get information about each employee's current and previous salary status so that they can produce a report. The report needs to show the date of each person's promotion and the salary amount.

This is pretty easy if you can put each salary in one row in the result set and let the host program format it. That is the first programming problem for the reader, in fact.

Oh, I forgot to mention that the application programmers are a bunch of lazy bums who want to have both the current and previous salary information on one row for each employee. This will let them write a very simple cursor statement and print out the report without any real work on their part.

Jack spoke with Fabian Pascal, who runs the www.dbdebunk.com Web site, the week he was working on this problem, and Mr. Pascal replied that this query could not be done. He said "in a truly relational language it could be done, but since SQL is not relational it isn't possible, not even with SQL-92." Sounds like a challenge to me!

Oh, I forgot to mention an additional constraint on the query; Jack is working in Oracle. This product was not up to SQL-92 standards at the time (i.e., no proper `OUTER JOIN`s, no general scalar subexpressions, and so on), so his query has to run under the old SQL-89 rules.

Assume that we have this test data:

```
CREATE TABLE Salaries
(emp_name CHAR(10) NOT NULL,
 sal_date DATE NOT NULL,
 sal_amt DECIMAL (8,2) NOT NULL,
 PRIMARY KEY (emp_name, sal_date));

INSERT INTO Salaries
VALUES ('Tom', '1996-06-20', 500.00),
       ('Tom', '1996-08-20', 700.00),
       ('Tom', '1996-10-20', 800.00),
       ('Tom', '1996-12-20', 900.00),
       ('Dick', '1996-06-20', 500.00),
```

```
        ('Harry', '1996-07-20', 500.00),
        ('Harry', '1996-09-20', 700.00);
```

Tom has had two promotions, Dick is a new hire, and Harry has had one promotion.

Answer #1

First, let's do the easy problem. The answer is use the query I call a generalized extrema or "`top (n)`" function and put it in a `VIEW`, like this:

```
CREATE VIEW Salaries1 (emp_name, curr_sal_date,
curr_sal_amt)
AS SELECT S0.emp_name, S0.sal_date, MAX(S0.sal_amt)
     FROM Salaries AS S0, Salaries AS S1
    WHERE S0.sal_date >= S1.sal_date
    GROUP BY S0.emp_name, S0.sal_date
   HAVING COUNT(*) <= 2;

CREATE VIEW Salaries2 (emp_name, sal_date, sal_amt)
AS SELECT S0.emp_name, S0.sal_date, MAX(S0.sal_amt)
     FROM Salaries AS S0, Salaries AS S1
    WHERE S0.sal_date <= S1.sal_date
      AND S0.emp_name = S1.emp_name
    GROUP BY S0.emp_name, S0.sal_date
   HAVING COUNT(*) <= 2;

Results
emp_name        sal_date     sal_amt
===================================
'Dick'     '1996-06-20'     500.00
'Harry'    '1996-07-20'     500.00
'Harry'    '1996-09-20'     700.00
'Tom'      '1996-10-20'     800.00
'Tom'      '1996-12-20'     900.00
```

The S1 copy of the Salaries table determines the boundary of the subset of two or fewer salary changes for each employee. The `MAX()` function is a trick to get the salary amount column in the results. This gives you one row for each of the first two salary changes for each

employee. If the programmers were not so lazy, you could pass this table to them and let them format it for the report.

Answer #2

The real problem is harder. One way to do this within the limits of SQL-89 is to break the problem into two cases:

1. Employees with only one salary action
2. Employees with two or more salary actions

We know that every employee has to fall into one and only one of those cases. One solution is to `UNION` both of the sets together:

```
SELECT S0.emp_name, S0.sal_date, S0.sal_amt, S1.sal_date,
S1.sal_amt
  FROM Salaries AS S0, Salaries AS S1
 WHERE S0.emp_name = S1.emp_name
 AND S0.sal_date =
    (SELECT MAX(S2.sal_date)
       FROM Salaries AS S2
      WHERE S0.emp_name = S2.emp_name)
 AND S1.sal_date =
    (SELECT MAX(S3.sal_date)
       FROM Salaries AS S3
      WHERE S0.emp_name = S3.emp_name
        AND S3.sal_date < S0.sal_date)
UNION ALL
SELECT S4.emp_name, MAX(S4.sal_date), MAX(S4.sal_amt),
NULL, NULL
  FROM Salaries AS S4
 GROUP BY S4.emp_name
HAVING COUNT(*) = 1;

emp_name        sal_date     sal_amt   sal_date    sal_amt
=========================================================
'Tom'      '1996-12-20'   900.00   '1996-10-20'   800.00
'Harry'    '1996-09-20'   700.00   '1996-07-20'   500.00
'Dick'     '1996-06-20'   500.00      NULL         NULL
```

DB2 programmers will recognize this as a version of the OUTER JOIN done without an SQL-92 standard OUTER JOIN operator. The first SELECT statement is the hardest. It is a self-join on the Salaries table, with copy S0 being the source for the most recent salary information and copy S1 the source for the next most recent information. The second SELECT statement is simply a grouped query that locates the employees with one row. Since the two result sets are disjoint, we can use the UNION ALL instead of a UNION operator to save an extra sorting operation.

Answer #3

I got several answers in response to my challenge for a better solution to this puzzle. Richard Romley of Smith Barney sent in the following SQL-92 solution. It takes advantage of the subquery table expression to avoid VIEWs:

```
SELECT B.emp_name, B.maxdate, Y.sal_amt, B.maxdate2,
Z.sal_amt
FROM (SELECT A.emp_name, A.maxdate, MAX(X.sal_date) AS
maxdate2
          FROM (SELECT W.emp_name, MAX(W.sal_date) AS
maxdate
                   FROM Salaries AS W
                  GROUP BY W.emp_name) AS A
          LEFT OUTER JOIN Salaries AS X
           ON A.emp_name = X.emp_name
             AND A.maxdate > X.sal_date
          GROUP BY A.emp_name, A.maxdate) AS B
        LEFT OUTER JOIN Salaries AS Y
         ON B.emp_name = Y.emp_name
            AND B.maxdate = Y.sal_date
        LEFT OUTER JOIN Salaries AS Z
         ON B.emp_name = Z.emp_name
            AND B.maxdate2 = Z.sal_date;
```

If your SQL product supports common table expressions (CTEs), you can convert some of the subqueries into VIEWs for the table subqueries named A and B.

Answer #4

Mike Conway came up with an answer in Oracle, which I tried to translate into SQL-92 with mixed results. The problem with the translation was that the Oracle version of SQL did not support the SQL-92 standard `OUTER JOIN` syntax, and you have to watch the order of execution to get the right results. Syed Kadir, an associate application engineer at Oracle, sent in an improvement on my answer using the `VIEW` that was created in the first solution:

```
SELECT S1.emp_name, S1.sal_date, S1.sal_amt, S2.sal_date,
S2.sal_amt
  FROM Salaries1 AS S1, Salaries2 AS S2  -- use the view
 WHERE S1.emp_name = S2.emp_name
   AND S1.sal_date > S2.sal_date
UNION ALL
SELECT emp_name, MAX(sal_date), MAX(sal_amt), NULL, NULL
  FROM Salaries1
 GROUP BY emp_name
HAVING COUNT(*) = 1;
```

You might have to replace the last two columns with the expressions `CAST (NULL AS DATE)` and `CAST(NULL AS DECIMAL(8,2))` to assure that they are of the right datatypes for a `UNION`.

Answer #5

Jack came up with a solution using the relational algebra operators as defined in one of Chris Date's books on the www.dbdebunk.com Web site, which I am not going to post, since (1) the original problem was to be done in Oracle, and (2) nobody has implemented Relational Algebra. There is an experimental language called Tutorial D based on Relational Algebra, but it is not widely available.

The problem with the solution was that it created false data. All employees without previous salary records were assigned a previous salary of `0.00` and a previous salary date of `'1900-01-01'`, even though zero and no value are logically different and the universe did not start in 1900.

Fabian Pascal commented that "This was a very long time ago and I do not recall the exact circumstances, and whether my reply was properly represented or understood (particularly coming from Celko).

My guess is that it had something to do with inability to resolve such problems without a precise definition of the tables to which the query is to be applied, the business rules in effect for the tables, and the query at issue. I will let Chris Date to respond to PV's solution."

Chris Date posted a solution in his private language that was more compact than Jack's solution, and that he evaluated was "Tedious, but essentially straightforward," along with the remark "Regarding whether Celko's solution is correct or not, I neither know, nor care."

A version that replaces the outer join with a `COALESCE()` by Andrey Odegov:

```
SELECT S1.emp_name_id, S1.sal_date AS curr_date, S1.sal_amt
AS
curr_amt,
      CASE WHEN S2.sal_date <> S1.sal_date THEN S2.sal_date
END AS
prev_date,
       CASE WHEN S2.sal_date <> S1.sal_date THEN S2.sal_amt
END AS
prev_amt
  FROM Salaries AS S1
 INNER JOIN Salaries AS S2
    ON S2.emp_name_id = S1.emp_name_id
   AND S2.sal_date = COALESCE((SELECT MAX(S4.sal_date)
                                 FROM Salaries AS S4
                                WHERE S4.emp_name_id =
S1.emp_name_id
                                  AND S4.sal_date <
S1.sal_date),
S2.sal_date)
 WHERE NOT EXISTS(SELECT *
                    FROM Salaries AS S3
                   WHERE S3.emp_name_id = S1.emp_name_id
                     AND S3.sal_date > S1.sal_date);
```

Answer #6

One approach is to build a `VIEW` or `CTE` that gives all possible pairs of salary dates, and then filter them:

```
CREATE VIEW SalaryHistory (curr_date, curr_amt, prev_date,
prev_amt)
```

```
AS
SELECT S0.emp_name_id, S0.sal_date AS curr_date,
                  S0.sal_amt AS curr_amt,
                  S1.sal_date AS prev_date,
                  S1.sal_amt AS prev_amt
  FROM Salaries AS S0
       LEFT OUTER JOIN
       Salaries AS S1
       ON S0.emp_name_id = S1.emp_name_id
          AND S0.sal_date > S1.sal_date;
```

then use it in a self-join query:

```
SELECT S0.emp_name_id, S0.curr_date, S0.curr_amt,
S0.prev_date, S0.prev_amt
  FROM SalaryHistory AS S0
 WHERE S0.curr_date
       = (SELECT MAX(curr_date)
            FROM SalaryHistory AS S1
           WHERE S0.emp_name_id = S1.emp_name_id)
   AND (S0.prev_date
        = (SELECT MAX(prev_date)
             FROM SalaryHistory AS S2
            WHERE S0.emp_name_id = S2.emp_name_id)
          OR S0.prev_date IS NULL)
```

This is still complex, but that view might be useful for computing other statistics.

Answer #7

Here is another version of the `VIEW` approach from MarkC600 on the SQL Server Newsgroup. The `OUTER JOIN` has been replaced with a `RANK()` function from SQL:2003. Study this and see how the thought pattern is changing:

```
WITH SalaryRanks(emp_name, sal_date, sal_amt, pos)
AS
(SELECT emp_name, sal_date, sal_amt,
       RANK() OVER(PARTITION BY emp_name ORDER BY sal_date
DESC)
```

```
  FROM Salaries)
SELECT C.emp_name,
       C.sal_date AS curr_date, C.sal_amt AS curr_amt,
       P.sal_date AS prev_date, P.sal_amt AS prev_amt
  FROM SalaryRanks AS C
       LEFT OUTER JOIN
       SalaryRanks AS P
       ON P.emp_name = C.emp_name
          AND P.pos = 2
WHERE C.pos = 1;
```

Answer #8

Here is an SQL:2003 version, with OLAP functions and SQL-92 `CASE` expressions from Dieter Noeth:

```
SELECT S1.emp_name,
      MAX (CASE WHEN rn = 1 THEN sal_date ELSE NULL END) AS
curr_date,
       MAX (CASE WHEN rn = 1 THEN sal_amt ELSE NULL END) AS
curr_amt,
      MAX (CASE WHEN rn = 2 THEN sal_date ELSE NULL END) AS
prev_date,
       MAX (CASE WHEN rn = 2 THEN sal_amt ELSE NULL END) AS
prev_amt,
 FROM (SELECT emp_name, sal_date, sal_amt,
              RANK()OVER (PARTITION BY S1.emp_name ORDER BY
sal_date DESC)
        FROM Salaries) AS S1 (emp_name, sal_date, sal_amt,
rn)
WHERE rn < 3
GROUP BY S1.emp_name;
```

The idea is to number the rows within each employee and then to pull out the two most current values for the employment date. The other approaches build all the target output rows first and then find the ones we want. This query finds the raw rows first and puts them together last.

The table is used only once, no self-joins, but a hidden sort will be required for the `RANK()` function. This is probably not a problem in SQL engines that use contiguous storage or have indexing that will group the employee names together.

Answer #9

Here is another answer from Dieter Noeth using OLAP/CTE (tested on Teradata, but runs on MS-SQL 2005, too):

```
WITH CTE (emp_name, sal_date, sal_amt, rn)
AS
(SELECT emp_name, sal_date, sal_amt ,
ROW_NUMBER() OVER (PARTITION BY emp_name
ORDER BY sal_date DESC) AS rn - row numbering
FROM Salaries)
SELECT O.emp_name,
O.sal_date AS curr_date, O.sal_amt AS curr_amt,
I.sal_date AS prev_date, I.sal_amt AS prev_amt
FROM CTE  AS O
            LEFT OUTER JOIN
             CTE AS I
            ON O.emp_name = I.emp_name AND I.rn = 2
WHERE O.rn = 1;
```

Again, SQL:2003 using OLAP functions in Teradata:

```
SELECT emp_name, curr_date, curr_amt,
       prev_date, prev_amt
  FROM (SELECT emp_name,
       sal_date AS curr_date, sal_amt AS curr_amt,
       MIN(sal_date)
          OVER (PARTITION BY emp_name
                ORDER BY sal_date DESC
                ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
       AS prev_date,
       MIN(sal_amt)
          OVER (PARTITION BY emp_name
                ORDER BY sal_date DESC
                ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
       AS prev_amt,
        ROW_NUMBER() OVER (PARTITION BY emp_name ORDER BY
sal_date DESC) AS rn
FROM Salaries) AS DT
WHERE rn = 1;
```

This query would be easier if Teradata supported the `WINDOW` clause.

PUZZLE 16 MECHANICS

Gerard Manko at ARI posted this problem on CompuServe in April 1994. ARI had just switched over from Paradox to Watcom SQL (now part of Sybase). The conversion of the legacy database was done by making each Paradox table into a Watcom SQL table, without any thought of normalization or integrity rules—just copy the column names and data types. Yes, I know that as the SQL guru, I should have sent him to that ring of hell reserved for people who do not normalize, but that does not get the job done, and ARI's approach is something I find in the real world all the time.

The system tracks teams of personnel to work on jobs. Each job has a slot for a single primary mechanic and a slot for a single optional assistant mechanic. The tables involved look like this:

```
CREATE TABLE Jobs
(job_id INTEGER NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 ... );

CREATE TABLE Personnel
(emp_id INTEGER NOT NULL PRIMARY KEY,
 emp_name CHAR(20) NOT NULL,
 ... );

CREATE TABLE Teams
(job_id INTEGER NOT NULL,
 mech_type INTEGER NOT NULL,
 emp_id INTEGER NOT NULL,
 ... );
```

Your first task is to add some integrity checking into the Teams table. Do not worry about normalization or the other tables for this problem.

What you want to do is build a query for a report that lists all the jobs by `job_id`, the primary mechanic (if any), and the assistant mechanic (if any). Here are some hints: You can get the `job_ids` from `Jobs` because that table has all of the current jobs, while the `Teams` table lists only those jobs for which a team has been assigned. The same person can be assigned as both a primary and assistant mechanic on the same job.

Answer #1

The first problem is to add referential integrity. The Teams table should probably be tied to the others with `FOREIGN KEY` references, and it is always a good idea to check the codes in the database schema, as follows:

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL REFERENCES Jobs(job_id),
 mech_type CHAR(10) NOT NULL
     CHECK (mech_type IN ('Primary', 'Assistant')),
 emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
    ...);
```

Experienced SQL people will immediately think of using a `LEFT OUTER JOIN`, because to get the primary mechanics only, you could write:

```
SELECT Jobs.job_id, Teams.emp_id AS “primary”
  FROM Jobs LEFT OUTER JOIN Teams
            ON Jobs.job_id = Teams.job_id
 WHERE Teams.mech_type = 'Primary';
```

You can do a similar `OUTER JOIN` to the Personnel table to tie it to Teams, but the problem here is that you want to do two independent outer joins for each mechanic's slot on a team, and put the results in one table. It is probably possible to build a horrible, deeply nested self `OUTER JOIN` all in one `SELECT` statement, but you would not be able to read or understand it.

You could do the report with views for primary and assistant mechanics, and then put them together, but you can avoid all of this mess with the following query:

```
SELECT Jobs.job_id,
       (SELECT emp_id
          FROM Teams
         WHERE Jobs.job_id = Teams.job_id
           AND Teams.mech_type = 'Primary') AS "primary",
       (SELECT emp_id
          FROM Teams
         WHERE Jobs.job_id = Teams.job_id
```

```
          AND Teams.mech_type = 'Assistant') AS assistant
  FROM Jobs;
```

The reason that "primary" is in double quotation marks is that it is a reserved word in SQL-92, as in `PRIMARY KEY`. The double quotation marks make the word into an identifier. When the same word is in single quotation marks, it is treated as a character string.

One trick is the ability to use two independent scalar `SELECT` statements in the outermost `SELECT`. To add the employee's name, simply change the innermost `SELECT` statements.

```
SELECT Jobs.job_id,
       (SELECT name
          FROM Teams, Personnel
         WHERE Jobs.job_id = Teams.job_id
           AND Personnel.emp_id = Teams.emp_id
           AND Teams.mech_type = 'Primary') AS "primary",
       (SELECT name
          FROM Teams, Personnel
         WHERE Jobs,job_id = Teams,job_id
           AND Personnel.emp_id = Teams.emp_id
           AND Teams.mech_type = 'Assistant') AS Assistant
 FROM Jobs:
```

If you have an employee acting as both primary and assistant mechanic on a single job, then you will get that employee in both slots. If you have two or more primary mechanics or two or more assistant mechanics on a job, then you will get an error, as you should. If you have no primary or assistant mechanic, then you will get an empty `SELECT` result, which becomes a `NULL`. That gives you the outer joins you wanted.

Answer #2

Skip Lees of Chico, California, wanted to make the Teams table enforce the rules that:

1. A `job_id` has zero or one primary mechanics.
2. A `job_id` has zero or one assistant mechanics.
3. A `job_id` always has at least one mechanic of some kind.

Based on rule 3, there should be no time at which a job has no team members. On the face of it, this makes sense.

Therefore, team information will have to be entered before job records. Using a referential integrity constraint will enforce this constraint. Restrictions 1 and 2 can be enforced by making "job_id" and "mech_type" into a two-column PRIMARY KEY, so that a job_id could never be entered more than once with a given mech_type.

```
CREATE TABLE Jobs
(job_id INTEGER NOT NULL PRIMARY KEY REFERENCES Teams
(job_id),
 start_date DATE NOT NULL,
 ... );

 CREATE TABLE Teams
 (job_id INTEGER NOT NULL,
  mech_type CHAR(10) NOT NULL
  CHECK (mech_type IN ('Primary', 'Assistant')),
  emp_id INTEGER NOT NULL REFERENCES Personnel(emp_id),
  ...
 PRIMARY KEY (job_id, mech_type));
```

There is a subtle "gotcha" in this problem. SQL-92 says that a REFERENCES clause in the referencing table has to reference a UNIQUE or PRIMARY KEY column set in the referenced table. That is, the reference is to be to the same number of columns of the same datatypes in the same order. Since we have a PRIMARY KEY, (job_id, mech_type) is available in the Teams table in your answer.

Therefore, the job_id column in the Jobs table by itself cannot reference just the job_id column in the Teams table. You could get around this with a UNIQUE constraint:

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL UNIQUE,
 mech_type CHAR(10) NOT NULL
 CHECK (mech_type IN ('Primary', 'Assistant')),
 PRIMARY KEY (job_id, mech_type));
```

but it might be more natural to say:

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL PRIMARY KEY,
 mech_type CHAR(10) NOT NULL
 CHECK (mech_type IN ('primary', 'assistant')),
 UNIQUE (job_id, mech_type));
```

because `job_id` is what identifies the entity that is represented by the table. In actual SQL implementations, the `PRIMARY KEY` declaration can affect data storage and access methods, so the choice could make a practical difference in performance.

But look at what we have done! I cannot have both "`primary`" and "`assistant`" mechanics on one job because this design would require `job_id` to be unique.

Answer #3

Having primary and assistant mechanics is a property of a team on a job, so let's fix the schema:

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL REFERENCES Jobs(job_id),
 primary_mech INTEGER NOT NULL
   REFERENCES Personnel(emp_id),
 assist_mech INTEGER NOT NULL
   REFERENCES Personnel(emp_id),
 CONSTRAINT at_least_one_mechanic
 CHECK(COALESCE (primary_mech, assist_mech) IS NOT NULL),
    ...);
```

But this is not enough; we want to be sure that only qualified mechanics hold those positions:

```
CREATE TABLE Personnel
(emp_id INTEGER NOT NULL PRIMARY KEY,
 mech_type CHAR(10) NOT NULL
  CHECK (mech_type IN ('Primary', 'Assistant')),
 UNIQUE (emp_id, mech_type),
 ..);
```

So change the Teams again:

```
CREATE TABLE Teams
(job_id INTEGER NOT NULL REFERENCES Jobs(job_id),
 primary_mech INTEGER NOT NULL,
 primary_type CHAR(10) DEFAULT 'Primary' NOT NULL
   CHECK (primary_type = 'Primary')
   REFERENCES Personnel(emp_id, mech_type),
 assist_mech INTEGER NOT NULL
 assist_type CHAR(10) DEFAULT 'Assistant' NOT NULL
   CHECK (assist_type = 'Assistant')
   REFERENCES Personnel(emp_id, mech_type),
 CONSTRAINT at_least_one_mechanic
 CHECK(COALESCE (primary_mech, assist_mech) IS NOT NULL),
    ...);
```

Now it should work.

PUZZLE
17 EMPLOYMENT AGENCY

Larry Wade posted a version of this problem on the Microsoft ACCESS forum at the end of February 1996. He is running an employment service that has a database with tables for job orders, candidates, and their job skills. He is trying to do queries to match candidates to job orders based on their skill. The job orders take the form of a Boolean expression connecting skills. For example, find all candidates with manufacturing and inventory or accounting skills.

First, let's construct a table of the candidate's skills. You can assume that personal information about the candidate is in another table, but we will not bother with it for this problem.

```
CREATE TABLE CandidateSkills
(candidate_id INTEGER NOT NULL,
 skill_code CHAR(15) NOT NULL,
 PRIMARY KEY (candidate_id, skill_code));

INSERT INTO CandidateSkills
VALUES ((100, 'accounting'),
        (100, 'inventory'),
        (100, 'manufacturing'),
        (200, 'accounting'),
        (200, 'inventory'),
        (300, 'manufacturing'),
        (400, 'inventory'),
        (400, 'manufacturing'),
        (500, 'accounting'),
        (500, 'manufacturing'));
```

The obvious solution would be to create dynamic SQL queries in a front-end product for each job order, such as:

```
SELECT candidate_id, 'job_id #212' -- constant job id code
  FROM CandidateSkills AS C1, -- one correlation per skill
       CandidateSkills AS C2,
       CandidateSkills AS C3
 WHERE C1.candidate_id = C2.candidate_id
   AND C1.candidate_id = C3.candidate_id
   AND              -- job order expression created here
```

```
(C1.skill_code = 'manufacturing'
 AND C2.skill_code = 'inventory'
  OR C3.skill_code = 'accounting')
```

A good programmer can come up with a screen form to do this in less than a week. You then save the query as a `VIEW` with the same name as the `job_id` code. Neat and quick! The trouble is that this solution will give you a huge collection of very slow queries.

Got a better idea? Oh, I forgot to mention that the number of job titles you have to handle is over 250,000. The agency is using the DOT (Dictionary of Occupational Titles), an encoding scheme used by the U.S. government for statistical purposes.

Answer #1

If we were not worrying about so many titles, the problem would be much easier. You could use an integer as a bit string and set the positions in the string to 1 or 0 for each occupation. For example:

```
'accounting' = 1
'inventory'= 2
'manufacturing'= 4
 etc.
```

Thus (`'inventory' AND 'manufacturing'`) can be represented by (2+ 4) = 6. Unfortunately, with a quarter of a million titles, this approach will not work.

The first problem is that you have to worry about parsing the search criteria. Does "manufacturing and inventory or accounting" mean "(manufacturing AND inventory) OR accounting" or does it mean "manufacturing AND (inventory OR accounting)" when you search? Let's assume that ANDs have higher precedence.

Answer #2

Another solution is to put every query into a disjunctive canonical form; what that means in English is that the search conditions are written as a string of `AND`-ed conditions joined together at the highest level by `OR`s.

Let's build another table of job orders that we want to fill:

```
CREATE TABLE JobOrders
(job_id INTEGER NOT NULL,
 skill_group INTEGER NOT NULL,
 skill_code CHAR(15) NOT NULL,
 PRIMARY KEY (job_id, skill_group, skill_code));
```

The skill_group code says that all these skills are required—they are the AND-ed terms in the canonical form. We can then assume that each skill_group in a job order is OR-ed with the others for that job_id. Create the table for the job orders.

Now insert the following orders in their canonical form:

```
Job 1 = ('inventory' AND 'manufacturing') OR 'accounting'
Job 2 = ('inventory' AND 'manufacturing')
               OR ('accounting' AND 'manufacturing')
Job 3 = 'manufacturing'
Job 4 = ('inventory' AND 'manufacturing' AND 'accounting')
```

This translates into:

```
INSERT INTO JobOrders
VALUES  (1, 1, 'inventory'),
        (1, 1, 'manufacturing'),
        (1, 2, 'accounting'),
        (2, 1, 'inventory'),
        (2, 1, 'manufacturing'),
        (2, 2, 'accounting'),
        (2, 2, 'manufacturing'),
        (3, 1, 'manufacturing'),
        (4, 1, 'inventory'),
        (4, 1, 'manufacturing'),
        (4, 1, 'accounting');
```

The query is a form of relational division, based on using the skill_code and skill_group combinations as the dividend and the candidate's skills as the divisor. Since the skill groups within a job_id are OR-ed together, if any one of them matches, we have a hit.

```
SELECT DISTINCT J1.job_id, C1.candidate_id
  FROM JobOrders AS J1 INNER JOIN CandidateSkills AS C1
            ON J1.skill_code = C1.skill_code
```

```
 GROUP BY candidate_id, skill_group, job_id
HAVING COUNT(*) >= (SELECT COUNT(*)
                      FROM JobOrders AS J2
                     WHERE J1.skill_group = J2.skill_group
                       AND J1.job_id = J2.job_id);
```

The sample data should produce the following results:

```
 job_id candidate_id
====== ============
     1          100
     1          200
     1          400
     1          500
     2          100
     2          400
     2          500
     3          100
     3          300
     3          400
     3          500
     4          100
```

As job orders and candidates are changed, the query stays the same. You can put this query into a `VIEW` and then use it to find the job for which we have no candidates, candidates for which we have no jobs, and so on.

Answer #3

Another answer came from Richard Romley at Smith Barney. He then came up with an answer that does not involve a correlated subquery in SQL-92, thus:

```
SELECT J1.job_id, C1.candidate_id
  FROM (SELECT job_id, skill_grp, COUNT(*)
          FROM JobSkillRequirements
         GROUP BY job_id, skill_grp)
         AS J1(job_id, skill_grp, grp_cnt)
     CROSS JOIN
```

```
    (SELECT R1.job_id, R1.skill_grp, S1.candidate_id,
COUNT(*)
       FROM JobSkillRequirements AS R1, CandidateSkills AS
S1
      WHERE R1.skillid = S1.skillid
      GROUP BY R1.job_id, R1.skill_grp, S1.candidate_id)
    AS C1(job_id, skill_grp, candidate_id, candidate_cnt)

 WHERE J1.job_id = C1.job_id
   AND J1.skill_grp = C1.skill_grp
   AND J1.grp_cnt = C1.candidate_cnt
 GROUP BY J1.job_id, C1.candidate_id;
```

You can replace the subquery table expressions in the `FROM` with a `CTE` clause, but I am not sure if they will run better or not. Replacing the table expressions with two `VIEW`s for `C1` and `J1` is not a good option, unless you want to use those `VIEW`s in other places.

I am also not sure how well the three `GROUP BY` statements will work compared to the correlated subquery. The grouped tables will not be able to use any indexing on the original tables, so this approach could be slower.

PUZZLE
18 JUNK MAIL

You are given a table with the addresses of consumers to whom we wish to send junk mail. The table has a family (`fam`) column that links Consumers with the same street address (`con_id`). We need this because our rules are that we mail only one offer to a household. The column contains the `PRIMARY KEY` value of the first person who has this address. Here is a skeleton of the table.

```
Consumers
con_name   address con_id    fam
================================
 'Bob'      'A'      1     NULL
 'Joe'      'B'      3     NULL
 'Mark'     'C'      5     NULL
 'Mary'     'A'      2      1
 'Vickie'   'B'      4      3
 'Wayne'    'D'      6     NULL
```

We need to delete those rows where `fam` is `NULL`, but there are other family members on the mailing list. In the above example, I need to delete `Bob` and `Joe`, but not `Mark` and `Wayne`.

Answer #1

A first attempt might try to do too much work, but translating the English specification directly into SQL results in the following:

```
DELETE FROM Consumers
 WHERE fam IS NULL   -- this guy has a NULL family value
   AND EXISTS   -- ..and there is someone who is
       (SELECT *
          FROM Consumers AS C1
         WHERE C1.id <> Consumers.id  -- a different person
           AND C1.address = Consumers.address -- at same
address
           AND C1.fam IS NOT NULL); -- who has a family value
```

Answer #2

But if you think about it, you will see that the `COUNT(*)` for the household has to be greater than 1.

```
DELETE FROM Consumers
 WHERE fam IS NULL   -- this guy has a NULL family value
   AND (SELECT COUNT(*)
          FROM Consumers AS C1
         WHERE C1.address = Consumers.address) > 1;
```

The trick is that the `COUNT(*)` aggregate will include `NULL`s in its tally.

Answer #3

Another version of Answer #1 comes from Franco Moreno:

```
DELETE FROM Consumers
 WHERE fam IS NULL   -- this guy has a NULL family value
   AND EXISTS (SELECT *
                 FROM Consumers AS C1
                WHERE C1.fam = Consumers.id);
```

PUZZLE
19 TOP SALESPEOPLE

This problem came up in March 1995 at Database World, when someone came back from the IBM pavilion to talk to me. IBM had a DB2 expert with a whiteboard set up to answer questions, and this one had stumped her. The problem starts with a table of salespeople and the amount of their sales, which looks like this:

```
CREATE TABLE SalesData
(district_nbr INTEGER NOT NULL,
 sales_person CHAR(10) NOT NULL,
 sales_id INTEGER NOT NULL,
 sales_amt DECIMAL(5,2) NOT NULL);
```

The boss just came in and asked for a report that will tell him about the three biggest sales and salespeople in each district. Let's use this data:

```
SalesData
district_nbr sales_person sales_id sales_amt
=========================================
    1    'Curly'          5    3.00
    1    'Harpo'         11    4.00
    1    'Larry'          1   50.00
    1    'Larry'          2   50.00
    1    'Larry'          3   50.00
    1    'Moe'            4    5.00
    2    'Dick'           8    5.00
    2    'Fred'           7    5.00
    2    'Harry'          6    5.00
    2    'Tom'            7    5.00
    3    'Irving'        10    5.00
    3    'Melvin'         9    7.00
    4    'Jenny'         15   20.00
    4    'Jessie'        16   10.00
    4    'Mary'          12   50.00
    4    'Oprah'         14   30.00
    4    'Sally'         13   40.00
```

Answer #1

Unfortunately, there are some problems in the specification we got. Do we want the three largest sales (regardless of who made them) or the top three salespeople? There is a difference—look at district 1, where 'Larry' made all three of the largest sales, but the three best salespeople were 'Larry', 'Moe', and 'Harpo'.

What if more than three people sold exactly the same amount, as in district 2? If a district has less than three salespeople working in it, as in district 3, do we drop it from the report or not? Let us make the decision, since this is just a puzzle and not a production system, that the boss meant the three largest sales in each district, without regard to who the salespeople were. That query can be:

```
SELECT *
  FROM SalesData AS S0
 WHERE sales_amt IN (SELECT S1.sales_amt
                       FROM SalesData AS S1
                      WHERE S0.district_nbr = S1.district_nbr
                        AND S0.sales_amt <= S1.sales_amt
                      HAVING COUNT(*) <= 3)
 ORDER BY S0.district_nbr, S0.sales_person, S0.sales_id,
S0.sales_amt;
```

In SQL-92, a HAVING clause by itself treats the whole table as a single group. If your SQL does not like this, then replace the "sales_amt IN (SELECT sales_amt ..." with "sales_amt >= (SELECT MIN(sales_amt) ..." in the SELECT clause. If you do that, however, the HAVING clause will drop the district_nbrs with only one sales_amt, which is district_nbr 2 in this case—giving these results:

```
Results
district_nbr sales_person sales_id sales_amt
====================================
    1         'Larry'       1   50.00
    1         'Larry'       2   50.00
    1         'Larry'       3   50.00
    3         'Irving'     10    5.00
    3         'Melvin'      9    7.00
    4         'Mary'       12   50.00
    4         'Oprah'      14   30.00
    4         'Sally'      13   40.00
```

Now what if we wanted the top three salespeople in their districts, without regard to how many people were assigned to each district? We could modify the query like this:

```
SELECT DISTINCT district_nbr, sales_person
  FROM SalesData AS SO
 WHERE sales_amt <= (SELECT MAX(S1.sales_amt)
                       FROM SalesData AS S1
                      WHERE SO.district_nbr = S1.district_nbr
                        AND SO.sales_amt <= S1.sales_amt
                    HAVING COUNT(DISTINCT SO.sales_amt) <= 3);
```

and get these results. Please notice that you are getting the three largest sales.

```
Answer
district_nbr sales_person
====================
     1       'Harpo'
     1       'Moe'
     1       'Larry'
     2       'Dick'
     2       'Fred'
     2       'Harry'
     2       'Tom'
     3       'Irving'
     3       'Melvin'
     4       'Oprah'
     4       'Sally'
     4       'Mary'
```

Notice that four people are tied for the top three sales positions in district 2. Likewise, the lack of competition in district 3 gave us two salespeople in the top three.

Answer #2

With the addition of OLAP functions in SQL-99, life becomes very easy:

```
SELECT S1.district_nbr, S1.sales_person
  FROM (SELECT district_nbr, sales_person,
```

```
                DENSE_RANK()
                OVER (PARTITION BY district_nbr
                      ORDER BY sales_amt DESC)
           FROM SalesData)
      AS S1.(district_nbr, sales_person, rank_nbr)
 WHERE S1.rank_nbr <= 3;
```

Teradata, Oracle, DB2, and SQL Server 2005 support these OLAP functions. How you want to handle ties will determine which OLAP function you will use.

`RANK ()` assigns a sequential numbering to each row within a partition. If there are duplicate values, they all are assigned equal ranks and you can get gaps in the numbering.

`DENSE_RANK ()` also assigns a sequential rank to a row within a partition. However, `DENSE_RANK()` has no gaps while ties are assigned the same numbering.

`ROW_NUMBER()` assigns a unique sequential numbering to each row within a partition and does not care about duplicate values.

If an `ORDER BY` clause is not given in the partition, the number will be arbitrary. For example, given a partition with two values of `foo` and five rows:

```
foo ROW_NUMBER() RANK()  DENSE_RANK()
=====================================
'A'    1          1         1
'A'    2          1         1
'A'    3          1         1
'B'    4          4         2
'B'    5          4         2
```

PUZZLE
20 TEST RESULTS

A problem got posted on the CompuServe Sybase Forum in May 1995 by a Mr. Shankar. It had to do with a table of test results. This table tracks the progress of the testing by providing a completion date for each `test_step` in the test. The `test_steps` are not always done in order, and each test can have several `test_steps`. For example, the `'Reading Skills'` test might have five `test_steps` and the `'Math Skills'` test might have six `test_steps`. We can assume that the `test_steps` are numbered from 1 to whatever is needed.

```
CREATE TABLE TestResults
(test_name CHAR(20) NOT NULL,
 test_step INTEGER NOT NULL,
 comp_date DATE,    -- null means incomplete
 PRIMARY KEY (test_name, test_step));
```

The problem is to write a quick query to find those tests that have been completed.

Answer #1

I came up with the "obvious" answer:

```
SELECT DISTINCT test_name
  FROM TestResults AS T1
 WHERE NOT EXISTS
       (SELECT *
          FROM TestResults AS T2
         WHERE T1.test_name = T2.test_name
           AND T2.comp_date IS NULL);
```

This says that the test does not have any uncompleted `test_steps`. Can you think of a different way to do it?

Answer #2

Roy Harvey had a better and simpler solution, based on a completely different approach:

```
SELECT test_name
  FROM TestResults
 GROUP BY test_name
HAVING COUNT(*) = COUNT(comp_date);
```

This works because `COUNT(*)` will tally the `NULL`s in the `comp_date` columns (actually, it is counting whole rows), while `COUNT(comp_date)` will drop the `NULL`s before doing the tally.

This is a good trick that can be used when you need to compare one set to another. Carry this one step further and you can find out how complete a test is:

```
SELECT test_name,
       COUNT(*) AS test_steps_needed
       (COUNT(*) - COUNT(comp_date)) AS test_steps_missing
  FROM TestResults
 GROUP BY test_name
HAVING COUNT(*) <> COUNT(comp_date);
```

If you just need a list of the incomplete tests, without any information as to how much is still needed, then you can write:

```
SELECT DISTINCT test_name
  FROM TestResults
 WHERE comp_date IS NULL;
```

Instead of looking at each step one at a time, think about how the set as a whole will behave.

PUZZLE
21 AIRPLANES AND PILOTS

We have a table of pilots and the planes they can fly and a table of planes in the hangar. We want the names of the pilots who can fly every plane in the hangar.

```
CREATE TABLE PilotSkills
(pilot CHAR(15) NOT NULL,
 plane CHAR(15) NOT NULL,
 PRIMARY KEY (pilot, plane));

INSERT INTO PilotSkills
VALUES ('Celko', 'Piper Cub'),
       ('Higgins', 'B-52 Bomber'),
       ('Higgins', 'F-14 Fighter'),
       ('Higgins', 'Piper Cub'),
       ('Jones', 'B-52 Bomber'),
       ('Jones', 'F-14 Fighter'),
       ('Smith', 'B-1 Bomber'),
       ('Smith', 'B-52 Bomber'),
       ('Smith', 'F-14 Fighter'),
       ('Wilson', 'B-1 Bomber'),
       ('Wilson', 'B-52 Bomber'),
       ('Wilson', 'F-14 Fighter'),
       ('Wilson', 'F-17 Fighter');

CREATE TABLE Hangar
(plane CHAR(15) PRIMARY KEY);

INSERT INTO Hangar
VALUES ('B-1 Bomber'),
       ('B-52 Bomber'),
       ('F-14 Fighter');
```

The answer would be:

```
PilotSkills DIVIDED BY Hangar
pilot
=============================
'Smith'
'Wilson'
```

In this example, Smith and Wilson are the two pilots who can fly everything in the hangar. Notice that Higgins and Celko know how to fly a Piper Cub, but we don't have one right now. In Codd's original definition of relational division, having more rows than are called for is not a problem.

The important characteristic of a relational division is that the `CROSS JOIN` (Cartesian product) of the divisor and the quotient produces a valid subset of rows from the dividend. This is where the name comes from, since the `CROSS JOIN` acts like a multiplication operator.

Answer #1

The classic answer is to pull out a copy of almost any textbook and look up relational division. Chris Date's classic textbook is the usual choice, and it gives a template that you can copy for the problem. We are dividing the pilot's skill table (dividend) by the hangar (divisor) to get a list of pilot names (quotient).

Can you find another way, which uses a trick we have already seen?

```
SELECT DISTINCT pilot
  FROM PilotSkills AS PS1
 WHERE NOT EXISTS
           (SELECT *
              FROM Hangar
             WHERE NOT EXISTS
                       (SELECT *
                          FROM PilotSkills AS PS2
                         WHERE (PS1.pilot = PS2.pilot)
                           AND (PS2.plane = Hangar.plane)));
```

Answer #2

Look at the puzzle that came just before this problem. Roy Harvey's trick that we used in the Test Results Puzzle can be applied here. It is important to reuse tricks when you can.

Imagine that each pilot gets a set of stickers that he pastes to each plane in the hangar he can fly. If the number of planes in the hangar is the same as the number of stickers he used, then he can fly all the planes in the hangar. That becomes the query:

```
SELECT Pilot
  FROM PilotSkills AS PS1, Hangar AS H1
 WHERE PS1.plane = H1.plane
 GROUP BY PS1.pilot
HAVING COUNT(PS1.plane) = (SELECT COUNT(*) FROM Hangar);
```

The `WHERE` clause restricts the `PilotSkills` plane list to those that are in the hangar before each pilot is grouped and tallied. If pilots were limited to only a subset of the hangar planes, you could drop the `WHERE` clause and use two `COUNT (DISTINCT x)` expressions instead of two `COUNT(x)` expressions.

The nested `EXISTS()` predicates version of relational division was made popular by Chris Date's textbooks, while I am associated with popularizing the `COUNT(*)` version of relational division. The interesting difference between these two approaches is how they handle an empty hangar—a sort of "relational division by zero," if you will. Version #1 will return all the pilots, while version #2 will return an empty set. In his book *Introduction to Database Systems—6th Edition*, Date defined a divisional operator that behaves like #2, so I assume that he views this as the correct answer.

Answer #3

Another kind of relational division is exact relational division. The dividend table must match exactly to the values of the divisor without any extra values:

```
SELECT PS1.pilot
  FROM PilotSkills AS PS1
       LEFT OUTER JOIN
       Hangar AS H1
       ON PS1.plane = H1.plane
 GROUP BY PS1.pilot
HAVING COUNT(PS1.plane) = (SELECT COUNT(plane) FROM Hangar)
   AND COUNT(H1.plane) = (SELECT COUNT(plane) FROM Hangar);
```

This says that a pilot must have the same number of certificates as there are planes in the hangar, and these certificates all match to a plane in the hangar, not something else. The "something else" is shown by a created `NULL` from the `LEFT OUTER JOIN`.

Please do not make the mistake of trying to reduce the HAVING clause with a little algebra to:

```
HAVING COUNT(PS1.plane) = COUNT(H1.plane)
```

because it does not work; it will tell you that the hangar has (n) planes in it and the pilot is certified for (n) planes, but not that those two sets of planes are equal to each other.

The Winter 1996 edition of *DB2 On-Line Magazine* had an article entitled "Powerful SQL: Beyond the Basics" by Sheryl Larsen that gave the results of testing both methods. Her conclusion for DB2 was that the nested EXISTS() version is better when the quotient has less than 25% of the dividend table's rows, and the COUNT(*) version is better when the quotient is more than 25% of the dividend table.

PUZZLE
22 LANDLORD

Karen Gallaghar tried to use the following SQL (translated from the Microsoft ACCESS original) to do a report on who has paid their rent in an apartment complex:

```
SELECT *
  FROM Units AS U1
       LEFT OUTER JOIN
        (Tenants AS T1
         LEFT OUTER JOIN
         RentPayments AS RP1
          ON T1.tenant_id = RP1.tenant_id)
        ON U1.unit_nbr = T1.unit_nbr
 WHERE U1.complex_id = 32
   AND U1.unit_nbr = RP1.unit_nbr
   AND T1.vacated_date IS NULL
   AND ((RP1.payment_date >= :my_start_date
         AND RP1.payment_date < :my_end_date)
    OR RP1.payment_date IS NULL)
 ORDER BY U1.unit_nbr, RP1.payment_date;
```

What she wanted was a report with either `RentPayments` rows within the date range, or blank `RentPayments` rows for each unit/tenant combination. What happened was that she did not get blank rows where there are no `RentPayments` rows unless she dropped the `RentPayments` conditions. Can you see the problem and rewrite the query?

Answer #1

The trick is to think about what is persistent and what is transient in a problem with `OUTER JOIN`s. The unit and tenant pairs are the place to start: the unit stays but the tenants come and go, so you need to preserve the unit side of the `LEFT OUTER JOIN`. Once you have the unit and tenant pairs, ask the same question and conclude that rent payments may be missing, even when you have a tenant in a unit.

```
SELECT *  -- * is bad in real code
  FROM (Units AS U1
       LEFT OUTER JOIN Tenants AS T1
```

```
        ON U1.unit_nbr = T1.unit_nbr
           AND T1.vacated_date IS NULL
           AND U1.complex_id = 32)
        LEFT OUTER JOIN RentPayments AS RP1
        ON (T1.tenant_id = RP1.tenant_id
           AND U1.unit_nbr = RP1.unit_nbr)
    WHERE RP1.payment_date BETWEEN :my_start_date
          AND :my_end_date
       OR RP1.payment_date IS NULL;
```

The predicate (`T1.tenant_id = RP1.tenant_id AND U1.unit_nbr = RP1.unit_nbr`) is saying that a particular tenant has paid rent for a particular unit. This is to cover the situations where the same party rents more than one unit in the complex. You may assume that referential constraints prevent you from collecting rent from someone who does not have a unit. The use of a `BETWEEN` predicate makes code easier to read and maintain, but it means you have to adjust the ending date.

PUZZLE
23 MAGAZINE

This one was posted on the Sybase forum of CompuServe by Keith McGregor in November 1994. One of his end users came to him with the following query. After nearly three days of trial and error, he still did not have a clue how to tell her to do it. He could have done this in about 30 minutes using COBOL and flat files, but did not see any way to do it in SQL. This is a good exercise in switching from a procedural mind-set to a declarative one.

You are given the following tables for a magazine distribution database:

```
CREATE TABLE Titles
(product_id INTEGER NOT NULL PRIMARY KEY,
 magazine_sku INTEGER NOT NULL,
 issn INTEGER NOT NULL,
 issn_year INTEGER NOT NULL);

CREATE TABLE Newsstands
(stand_nbr INTEGER NOT NULL PRIMARY KEY,
 stand_name CHAR(20) NOT NULL);

CREATE TABLE Sales
(product_id INTEGER NOT NULL REFERENCES Titles(product_id),

 stand_nbr INTEGER NOT NULL REFERENCES
Newsstands(stand_nbr),
 net_sold_qty INTEGER NOT NULL,
 PRIMARY KEY(product_id, stand_nbr));
```

He needed to select the newsstand(s) where:

1. The average `net_sold_qty` is greater than 2 for both magazine titles 02667 and 48632 (if the average is 2 or less for either one, do not select the newsstand at all).

Or

2. The average net_sold_qty is greater than 5 for magazine 01107 (if this is true, select the newsstand regardless of the result of condition 1).

Answer #1

Let's create a VIEW of the three tables joined together that will give us the basic information we are after. Maybe this VIEW can be used for other reports later.

```
CREATE VIEW MagazineSales(stand_name, title, net_sold_qty)
AS SELECT Sales.stand_name, Titles.title, net_sold_qty
     FROM Titles, Sales, Newsstands
    WHERE Sales.stand_nbr = Newsstands.stand_nbr
      AND Titles.product_id = Sales.product_id;
```

Then we write the query from Hell:

```
SELECT stand_name
  FROM MagazineSales AS M0
 GROUP BY stand_name
HAVING  -- the two accept conditions
   ((SELECT AVG(net_sold_qty)
       FROM MagazineSales AS M1
      WHERE M1.stand_nbr = M0.stand_nbr
        AND magazine_sku = '01107') > 5)
     OR ((SELECT AVG(net_sold_qty)
           FROM MagazineSales AS M2
            WHERE M2.stand_nbr = M0.stand_nbr
              AND magazine_sku IN ('02667', '48632')) > 2)
 AND NOT -- the two reject conditions
      ((SELECT AVG(net_sold_qty)
          FROM MagazineSales AS M3
         WHERE M3.stand_nbr = M0.stand_nbr
           AND magazine_sku = '02667') < 2

       OR
        (SELECT AVG(net_sold_qty)
           FROM MagazineSales AS M4
          WHERE M4.stand_nbr = M0.stand_nbr
            AND magazine_sku = '48632') < 2);
```

For bonus points, can you simplify or improve this expression?

Hint: *DeMorgan's law might be useful, and it would help to have a decision table.*

Answer #2

In April 1995, Carl C. Federl, an independent consultant in Clarendon Hills, Illinois, proposed that the solution provided to this puzzle could be greatly simplified by using two techniques: First, create a `VIEW` of the average sales and include an `'EXISTS'` for the condition of two titles that both must exceed a threshold.

```
CREATE VIEW MagazineSales (stand_nbr, title, avg_qty_sold)
AS SELECT Sales.stand_nbr, Titles.title,
AVG(Sales.net_sold_qty)
     FROM Titles, Newsstands, Sales
    WHERE Titles.product_id = Sales.product_id
      AND Newsstands.stand_nbr = Sales.stand_nbr
      AND Titles.magazine_sku IN (01107,02667, 48632)
    GROUP BY Sales.stand_nbr, Titles.title;
```

Now the query is greatly reduced:

```
SELECT DISTINCT Newsstands.stand_name
  FROM MagazineSales AS M0, Newsstands AS N0
 WHERE N0.stand_nbr = M0.stand_nbr
   AND ((M0.magazine_sku = 1107 AND M0.avg_qty_sold > 5)
        OR (M0.magazine_sku = 2667 AND M0.avg_qty_sold > 2
            AND EXISTS (SELECT *
                          FROM MagazineSales AS Other
                         WHERE Other.magazine_sku = 48632
                        AND Other.stand_nbr = M0.stand_nbr
                          AND Other.avg_qty_sold > 2)));
```

In older versions of Sybase SQL and other databases, a `VIEW` with an aggregate that is joined will not produce the desired results. Today, the `VIEW` could be a `CTE` expression.

Instead of a `VIEW`, a temporary table must be used.

Answer #3

This answer came from Adam Thompson, manager of technical support at Doncar Systems, Inc., after he saw the first edition of this book. First, he added an inverse-lookup index on Titles.

This is pretty much essential to top-notch performance in a data-heavy environment. I have not mentioned indexing in this book, because that is not part of the SQL Standards.

```
CREATE INDEX Titles_magazine_sku ON Titles (title,
product_id);
```

Then he found a completely SQL-89 solution:

```
SELECT DISTINCT N1.stand_name
  FROM Newsstands AS N1
 WHERE N1.stand_nbr IN
        (SELECT S1.stand_nbr
           FROM Sales AS S1
          WHERE S1.product_id IN
                   (SELECT T1.product_id
                      FROM Titles AS T1
                     WHERE magazine_sku = 01107)
          GROUP BY S1.stand_nbr
         HAVING AVG(S1.net_sold_qty) > 5)
    OR (N1.stand_nbr IN (SELECT S1.stand_nbr
                           FROM Sales AS S1
                          WHERE S1.product_id IN
                                 (SELECT T1.product_id
                                    FROM Titles AS T1
                                   WHERE magazine_sku = 02667)
                          GROUP BY S1.stand_nbr
                         HAVING AVG(S1.net_sold_qty) > 2)
        AND N1.stand_nbr IN
                (SELECT S1.stand_nbr
                   FROM Sales AS S1
                  WHERE S1.product_id IN
                           (SELECT T1.product_id
                              FROM Titles AS T1
                             WHERE magazine_sku = 48632)
                  GROUP BY S1.stand_nbr
                 HAVING AVG(S1.net_sold_qty) > 2));
```

He tested this under SQL Anywhere v5.5, and with the addition of one index as indicated above in the schema, the entire thing runs off indices with only one full-table scan, on n1. I think that full-table scan is perfectly reasonable, given the query.

Note that the `GROUP BY`s inside the correlated subqueries need only the `stand_nbr` column and can omit `product_id` only because of the `WHERE` clause immediately preceding each instance. A more general solution would have been to have "`group by stand_nbr, product_id`" to allow for future expansion of the query terms. The ability to use a multiple-column row expression in a predicate is part of the SQL-92 standard, but it is not widely implemented yet.

Answer #4

Another solution uses the `GROUP BY` instead, but it requires the `CASE` statement:

```
SELECT N1.stand_name
  FROM Sales AS S1, Titles AS T1, Newsstands AS N1
 WHERE T1.title_id IN (02667, 48632,01107)
   AND S1.product_id = T1.product_id
 GROUP BY S1.stand_nbr
HAVING ((AVG(CASE WHEN T1.magazine_sku = 02667
                  THEN S1.net_sold_qty
                  ELSE NULL END) > 2
         AND
         AVG(CASE WHEN T1.magazine_sku = 48632
                  THEN S1.net_sold_qty
                  ELSE NULL END) > 2)
     OR AVG(CASE WHEN T1.magazine_sku = 01107
                 THEN S1.net_sold_qty
                 ELSE NULL END) > 5)
   AND S1.stand_nbr = N1.stand_nbr;
```

Answer #5

Richard Romley proposed several answers in SQL-92 syntax in September 1997.

```
SELECT N1.stand_name
  FROM (SELECT S1.stand_nbr
```

```
  FROM Sales AS S1
       INNER JOIN
       Titles AS T1
       ON S1.product_id = T1.product_id
 WHERE T1.title_id IN (02667, 48632,01107)
 GROUP BY S1.stand_nbr
HAVING (AVG(CASE
            WHEN T1.magazine_sku = 02667
            THEN S1.net_sold_qty
            ELSE NULL END) > 2
        AND AVG(CASE
                WHEN T1.magazine_sku = 48632
                THEN S1.net_sold_qty
                ELSE NULL END) > 2)
    OR AVG(CASE
            WHEN T1.magazine_sku = 01107
            THEN S1.net_sold_qty
            ELSE NULL END) > 5)
INNER JOIN
Newsstands AS N1
ON S1.stand_nbr = N1.stand_nbr;
```

He observed that:

1. The newsstands table has nothing to do with the solution of the problem. It is only needed to look up the stand name once the qualifying "`stand_nbr`" has been determined. It only confuses and complicates the solution by introducing it up front.
2. We only need to look up the stand name once for each qualifying `stand_nbr`. This should be done last, after the `stand_nbr` has been found. Suppose there are 10,000 `Sales` rows. In this solution we will do 10,000 `JOIN`s to the newsstands table! In the original solution we will do one `INNER JOIN`! Which do you think is better?
3. In this solution, the stand_name column must be included in the `GROUP BY` clause in order to include it in the `SELECT` list. This is bad for several reasons:
   - The `stand_name` column has nothing to do with the grouping and logically does not belong there.

- Any additional columns we wanted to include in the SELECT list having clauses or ORDER BYs would need to be added to the GROUP BY. This is illogical, because the GROUP BY should be used to create the aggregates necessary to solve the problem and include nothing else. Additional columns in the GROUP BY mask what the purpose of the GROUP BY is and make the query hard to read.
- Adding unnecessary columns to the GROUP BY represents a potentially huge performance hit. Just look at stand_nbr as an integer. We could add a dozen additional columns to the SELECT list, requiring a dozen additional columns in the GROUP BY. What effect do you think it will have forcing the server to do a GROUP BY on 10,000 rows with a dozen columns totaling a couple of hundred bytes each, versus one single-integer column?
- It is a maintenance nightmare. If you need to add another column to the SELECT list, you also need to add it to the GROUP BY, even though the request has nothing to do with the grouping of the query.

Answer #6

In July 1999, Francisco Moreno proposed a version that takes advantage of the set operators in SQL-92 syntax and a little algebra:

```
SELECT stand_name
  FROM Newsstands AS N1
 WHERE 1 = ANY ((SELECT SIGN(AVG(net_sold_qty) - 2)
                   FROM Sales AS S1
                  WHERE product_id IN (SELECT product_id
                                         FROM Titles
                                        WHERE magazine_sku = 2667)
                    AND S1.stand_nbr = S21.stand_nbr
                INTERSECT
                SELECT SIGN(AVG(net_sold_qty) - 2)
                  FROM Sales AS S2
                 WHERE product_id IN (SELECT product_id
                                        FROM Titles
                                       WHERE magazine_sku = 48632)
                   AND S2.stand_nbr = S21.stand_nbr)
               UNION
```

```
SELECT SIGN(AVG(net_sold_qty) - 5)
  FROM Sales AS S3
 WHERE product_id IN (SELECT product_id
                        FROM Titles
                       WHERE magazine_sku = 1107)
   AND S3.stand_nbr = S21. stand_nbr);
```

Answer #7

Mr. Kuznetsov also came up with a simple solution:

```
CREATE TABLE Titles
(product_id INTEGER NOT NULL PRIMARY KEY,
 magazine_sku INTEGER NOT NULL,
 issn INTEGER NOT NULL,
 issn_year INTEGER NOT NULL);

INSERT INTO Titles
VALUES (1, 12345, 1, 2006), (2, 2667, 1, 2006), (3, 48632,
1, 2006),
 (4, 1107, 1, 2006), (5, 12345, 2, 2006), (6, 2667, 2,
2006),
(7, 48632, 2, 2006), (8, 1107, 2, 2006);

CREATE TABLE Sales
(product_id INTEGER NOT NULL,
 stand_nbr INTEGER NOT NULL,
 net_sold_qty INTEGER NOT NULL);
-- stand 1
INSERT INTO Sales VALUES (1, 1, 1);
INSERT INTO Sales VALUES (2, 1, 4);
INSERT INTO Sales VALUES (3, 1, 1);
INSERT INTO Sales VALUES (4, 1, 1);
INSERT INTO Sales VALUES (5, 1, 1);
INSERT INTO Sales VALUES (6, 1, 2);
INSERT INTO Sales VALUES (7, 1, 1);
-- stand 2 meets the criteria
INSERT INTO Sales VALUES (4, 2, 5);
INSERT INTO Sales VALUES (8, 2, 6);
INSERT INTO Sales VALUES (3, 2, 1);
-- stand 3 meets the criteria
INSERT INTO Sales VALUES (1, 3, 1);
```

```
INSERT INTO Sales VALUES (2, 3, 3);
INSERT INTO Sales VALUES (3, 3, 3);
INSERT INTO Sales VALUES (4, 3, 1);
INSERT INTO Sales VALUES (5, 3, 1);
INSERT INTO Sales VALUES (6, 3, 3);
INSERT INTO Sales VALUES (7, 3, 3);
-- stand 4
INSERT INTO Sales VALUES (1, 4, 1);
INSERT INTO Sales VALUES (2, 4, 1);
INSERT INTO Sales VALUES (3, 4, 4);
INSERT INTO Sales VALUES (4, 4, 1);
INSERT INTO Sales VALUES (5, 4, 1);
INSERT INTO Sales VALUES (6, 4, 1);
INSERT INTO Sales VALUES (7, 4, 2);

SELECT stand_nbr
  FROM (SELECT stand_nbr,
               AVG(CASE WHEN title = 2667 THEN net_sold_qty
END),
              AVG(CASE WHEN title = 48632 THEN net_sold_qty
END),
               AVG(CASE WHEN title = 1107 THEN net_sold_qty
END) avg_1107
          FROM Sales, Titles
         WHERE Sales.product_id = Titles.product_id
         GROUP BY stand_nbr
        ) AS T (stand_nbr, avg_2667, avg_48632, avg_1107)
 WHERE avg_1107 > 5 OR (avg_2667 > 2 AND avg_48632 > 2);
```

A minor note: leaving off the `ELSE NULL` in a `CASE` expression is legal shorthand, but I prefer to use it as a placeholder for future updates and additions, as well as a reminder that a `NULL` is being created.

PUZZLE
24 ONE IN TEN

Alan Flancman ran into a problem with some legacy system data that had been moved over to an SQL database. The table looked like this:

```
CREATE TABLE MyTable
(keycol INTEGER NOT NULL,
 f1 INTEGER NOT NULL,
 f2 INTEGER NOT NULL,
 f3 INTEGER NOT NULL,
 f4 INTEGER NOT NULL,
 f5 INTEGER NOT NULL,
 f6 INTEGER NOT NULL,
 f7 INTEGER NOT NULL,
 f8 INTEGER NOT NULL,
 f9 INTEGER NOT NULL,
 f10 INTEGER NOT NULL);
```

The columns `f1` through `f10` were an attempt to flatten out an array into a table. What he wanted was an elegant way to test against the `f1` through `f10` columns to find the rows that had exactly one nonzero value in their columns.

How many different approaches can you find? We are looking for variety and not performance.

Answer #1

You could use the `SIGN()` function in Sybase and other SQL products. This function returns `-1`, `0`, or `+1` if the argument is negative, zero, or positive, respectively. Assuming that your numbers are zero or greater, you simply write:

```
SELECT *
  FROM MyTable
 WHERE SIGN(f1) + SIGN(f2) + ... + SIGN(f10) = 1;
```

to find a single nonzero value. If you can have negative values, then make the functions `SIGN(ABS(fn))`.

The SIGN(ABS()) function combination can be written with the CASE expression in SQL-92 as:

```
CASE WHEN x <> 0 THEN 1 ELSE 0 END
```

Answer #2

Since the fields are really an attempt to fake an array, you should put this table into First Normal Form (1NF), like this:

```
CREATE TABLE Foobar
(keycol INTEGER NOT NULL,
 i INTEGER NOT NULL CHECK (i BETWEEN 1 AND 10),
 f INTEGER NOT NULL,
 PRIMARY KEY (keycol, i));
```

The extra column i is really the subscript for the array. You now view the problem as finding an entity that has exactly nine zero-valued columns, instead of finding an entity that has exactly one nonzero-valued nonkey column. That is suddenly easy:

```
SELECT keycol
  FROM Foobar
 WHERE f = 0
 GROUP BY keycol
HAVING COUNT(*) = 9;
```

You can create a VIEW that has the structure of Foobar, but things are going to run pretty slowly unless you have a good optimizer:

```
CREATE VIEW Foobar (keycol, f)
AS SELECT keycol, f1 FROM MyTable WHERE f1 <> 0
   UNION
   SELECT keycol, f2 FROM MyTable WHERE f2 <> 0
   UNION
   ...
   UNION
   SELECT keycol, f10 FROM MyTable WHERE f10 <> 0 ;
```

Answer #3

This depends on a feature of SQL-92 that is not generally available yet. First, the code, then the explanation:

```
SELECT *
  FROM MyTable
 WHERE (f1, f2, ... , f10) IN
       (VALUES (f1, 0, 0, 0, 0, 0, 0, 0, 0, 0),
               (0, f2, 0, 0, 0, 0, 0, 0, 0, 0),
                ....
               (0, 0, 0, 0, 0, 0, 0, 0, 0, f10))
   AND (f1 + f2 + ... f10) > 0;
```

In SQL-92, you can use row and table constructors in comparison predicates. The `IN` predicate expands into a sequence of `OR`-ed equality predicates. The row-wise version of equality is then done on a position-by-position basis, where all corresponding values must be equal.

Answer #4

If one and only one column is nonzero, then there is a one set of nine columns that are all zeros.

```
SELECT *
  FROM MyTable
 WHERE 0 IN
         (VALUES (f2 + f3 + .. f10), -- pull out f1
                 (f1 + f3 + .. f10), -- pull out f2
                  ...
                 (f1 + f2 + .. f9)) -- pull out f10
   AND (f1 + f2 + ... f10) > 0;
```

Answer #5

In January 1999, Trevor Dwyer posted a similar problem he actually had on CompuServe. The differences were that his table had `NULL`s in it, instead of zeros. His problem was the need to test for any number of columns having a non-`NULL` value. This is very easy in SQL-92:

```
SELECT *
  FROM MyTable
 WHERE COALESCE(f1, f2, f3, f4, f5, f6, f7, f8, f9, f10)
       IS NOT NULL;
```

The `COALESCE()` function will return the first non-`NULL` it finds in the list. If the entire list is made up of `NULL`s, then it will return `NULL`.

Obviously, the original problem could be done by replacing each of the column expressions in the list with a call to a conversion function:

```
COALESCE (NULLIF (f1, 0), NULLIF (f2, 0), ..., NULLIF (f10,
0))
```

Answer #6

Frédéric Brouard (f.brouard@simog.com) came up with this answer:

```
SELECT *
  FROM MyTable
 WHERE
(f1+1)*(f2+1)*(f3+1)*(f4+1)*(f5+1)*(f6+1)*(f7+1)*(f8+1)*(f9
+1)*(f10+1)*(f2+1)= 2
```

PUZZLE
25 MILESTONE

This puzzle, in a slightly different form, came from Brian Young. His system tracks a series of dates (milestones) for each particular type of service (`service_type`) that they sell on a particular order (`my_order`).

These dates constitute the schedule for the delivery of the service and vary with the type of service they are delivering. Their management would like to see a schedule for each shop horizontally, which I must admit is a reasonable request, but it is really a job for the display functions in the front end and not the database. They also want to be able to specify which task code (`service_type`) to display.

Brian ran across a clever solution to this problem by Steve Roti in an SQL server book, but it relies on the `SUM` function and a multiplication by 1 to yield the correct result. (That Roti guy is very clever!) Unfortunately, this technique doesn't work with dates. So here is the table structure:

```
CREATE TABLE ServicesSchedule
(shop_id CHAR(3) NOT NULL,
 order_nbr CHAR(10) NOT NULL,
 sch_seq INTEGER NOT NULL CHECK (sch_seq IN (1,2,3)),
 service_type CHAR(2) NOT NULL,
 sch_date DATE,
 PRIMARY KEY (shop_id, order_nbr, sch_seq));
```

Where `sch_seq` is encoded as:

```
(1 = 'processed')
(2 = 'completed')
(3 = 'confirmed')
```

The data normally appears like this:

```
ServicesSchedule
shop_id  order_nbr  sch_seq  service_type sch_date
================================================
002  4155526710       1       01     '1994-07-16'
002  4155526710       2       01     '1994-07-30'
002  4155526710       3       01     '1994-10-01'
```

```
002  4155526711       1       01     '1994-07-16'
002  4155526711       2       01     '1994-07-30'
002  4155526711       3       01     NULL
```

This is the way they would like it to appear, assuming they want to look at (`service_type = 01`),

```
order_nbr   processed     completed     confirmed
=================================================
4155526710  '1994-07-16'  '1994-07-30'  '1994-10-01'
4155526711  '1994-07-16'  '1994-07-30'  NULL
```

Answer #1

If you only have an SQL-89 product instead of an SQL-92, you can do this with self-joins:

```
SELECT S0.order_nbr, S0.sch_date, S0.sch_date,
       S1.sch_date, S2.sch_date, S3.sch_date
  FROM ServicesSchedule AS S0, ServicesSchedule AS S1,
       ServicesSchedule AS S2, ServicesSchedule AS S3
 WHERE S0.service_type = :my_tos -- set task code
   AND S0.order_nbr = :my_order -- set order_nbr
   AND S1.order_nbr = S0.order_nbr AND S1.sch_seq = 1
   AND S2.order_nbr = S0.order_nbr AND S2.sch_seq = 2
   AND S3.order_nbr = S0.order_nbr AND S3.sch_seq = 3;
```

The problem is that for some SQL products, the self-joins are very expensive. This is probably the fastest answer on the old SQL products. Can you think of another way?

Answer #2

In SQL-92, this is easy and very fast with subquery expressions:

```
SELECT S0.order_nbr,
       (SELECT sch_date
          FROM ServicesSchedule AS S1
         WHERE S1.sch_seq = 1
           AND S1.order_nbr = S0.order_nbr) AS processed,
       (SELECT sch_date
```

```
         FROM ServicesSchedule AS S2
        WHERE S2.sch_seq = 2
          AND S2.order_nbr = S0.order_nbr) AS completed,
       (SELECT sch_date
         FROM ServicesSchedule AS S3
        WHERE S3.sch_seq = 3
          AND S3.order_nbr = S0.order_nbr) AS confirmed
  FROM ServicesSchedule AS S0
 WHERE service_type = :my_tos ; -- set task code
```

The trouble with this trick is that it might not be optimized in your SQL. This can be worse than the self-join.

Answer #3

You could try using `UNION ALL` operators and a work table to flatten out the original table. This is not usually a very good performer, but if the original table is very large, it can sometimes beat the self-join used in Answer #2.

```
INSERT INTO Work (order_nbr, processed, completed,
confirmed)
SELECT order_nbr, NULL, NULL, NULL
 FROM ServicesSchedule AS S0
 WHERE service_type = :my_tos -- set task code
UNION ALL
SELECT order_nbr, sch_date, NULL, NULL
  FROM ServicesSchedule AS S1
 WHERE S1.sch_seq = 1
   AND S1.order_nbr = :my_order
   AND service_type = :my_tos -- set task code
UNION ALL
 SELECT order_nbr, NULL, sch_date, NULL
  FROM ServicesSchedule AS S2
  WHERE S2.sch_seq = 2
   AND S2.order_nbr = :my_order
   AND service_type = :my_tos -- set task code
UNION ALL
 SELECT order_nbr, NULL, NULL, sch_date
  FROM ServicesSchedule AS S3
 WHERE S3.sch_seq = 3
```

```
   AND S3.order_nbr = :my_order
   AND service_type = :my_tos -- set task code
```

This simple UNION ALL statement might have to be broken down into four INSERTs. The final query is simply:

```
SELECT order_nbr, MAX(processed), MAX(completed),
MAX(confirmed)
  FROM Work
 GROUP BY order_nbr;
```

The MAX() function picks the highest non-NULL value in the group, which also happens to be the only non-NULL value in the group.

Answer #4

However, UNIONs can often be replaced by CASE expressions in SQL-92, which leads us to this solution:

```
SELECT order_nbr,
       (CASE WHEN sch_seq = 1
             THEN sch_date
             ELSE NULL END) AS processed,
       (CASE WHEN sch_seq = 2
             THEN sch_date END) AS
             ELSE NULL END) AS completed,
       (CASE WHEN sch_seq = 3
             THEN sch_date
             ELSE NULL END) AS confirmed
  FROM ServicesSchedule
 WHERE service_type = :my_tos
   AND order_nbr = :my_order;
```

or you can try this same query with a GROUP BY clause:

```
SELECT order_nbr,
       MAX(CASE WHEN sch_seq = 1 THEN sch_date ELSE NULL
END) AS processed,
       MAX(CASE WHEN sch_seq = 2 THEN sch_date ELSE NULL
END) AS completed,
       MAX(CASE WHEN sch_seq = 3 THEN sch_date ELSE NULL
END) AS confirmed
```

```
  FROM ServicesSchedule
 WHERE service_type=:my_tos
   AND order_nbr= :my_order
 GROUP BY order_nbr, service_type;
```

This is the preferred way in current SQL products, and now you can translate old code into this template when you see it.

PUZZLE
26 DATAFLOW DIAGRAMS

Tom Bragg posted a version of this problem on the CASE Forum on CompuServe. You have a table of dataflow diagrams (DFDs), which has the name of the diagram, the names of the bubbles in each diagram, and the labels on the flow lines. It looks like this:

```
CREATE TABLE DataFlowDiagrams
(diagram_name CHAR(10) NOT NULL,
 bubble_name CHAR(10) NOT NULL,
 flow_name CHAR(10) NOT NULL,
 PRIMARY KEY (diagram_name, bubble_name, flow_name));
```

To explain the problem, let's use this table:

```
DataFlowDiagrams
diagram_name  bubble_name  flow_name
===================================
Proc1    input   guesses
Proc1    input   opinions
Proc1    crunch  facts
Proc1    crunch  guesses
Proc1    crunch  opinions
Proc1    output  facts
Proc1    output  guesses
Proc2    reckon  guesses
Proc2    reckon  opinions
 ...
```

What we want to find is what flows do *not* go into each bubble within the diagrams. This will be part of a diagram validation routine that will search for missing dataflows. To make this easier, assume that all bubbles should have all flows. This would mean that `(Proc1, input)` is missing the `'facts'` flow, and that `(Proc1, output)` is missing the `'opinions'` flow.

Answer #1

We could use this SQL-92 query:

```
SELECT F1.diagram_name, F1.bubble_name, F2.flow_name
 FROM (SELECT F1.diagram_name, F1.bubble_name
         FROM DataFlowDiagrams AS F1
       CROSS JOIN
       SELECT DISTINCT F2.flow_name
         FROM DataFlowDiagrams AS F2)
       EXCEPT
     SELECT F3.diagram_name, F3.bubble_name, F3.flow_name

         FROM DataFlowDiagrams AS F3;
```

Basically, it makes all possible combinations of diagrams and flows, and then removes the ones we already have.

Answer #2

Another SQL-92 query would be:

```
SELECT F1.diagram_name, F1.bubble_name, F2.flow_name
 FROM (SELECT F1.diagram_name, F1.bubble_name
         FROM DataFlowDiagrams AS F1
       CROSS JOIN
       SELECT DISTINCT F2.flow_name
         FROM DataFlowDiagrams AS F2
        WHERE flow NOT IN (SELECT F3.flow_name
                             FROM DataFlowDiagrams AS F3
                            WHERE F3.diagram_name =
F1.diagram_name
                              AND F3.bubble_name =
F1.bubble_name)
  ORDER BY F1.diagram_name, F1.bubble_name, F2.flow_name;
```

Answer #3

Or to answer the puzzle in SQL-89, you will need to use `VIEW`s:

```
-- build a set of all the flows
CREATE VIEW AllDFDFlows (flow_name)
AS SELECT DISTINCT flow_name FROM DataFlowDiagrams;

-- attach all the flows to each row of the original table
```

```
 CREATE VIEW NewDFD (diagram_name, bubble_name, flow_name,
missingflow)
 AS SELECT DISTINCT F1.diagram_name, F1.bubble_name,
F1.flow, F2.flow_name
      FROM DataFlowDiagrams AS F1, AllDFDFlows AS F2
     WHERE F1.flow_name <> F2.flow_name;

 -- Show me the (diagram_name, bubble_name) pairs and
missing flow
 -- where the missing flow was not somewhere in the flow
column
 -- of the pair.
 SELECT DISTINCT diagram_name, bubble_name, missingflow
  FROM NewDFD AS ND1
 WHERE NOT EXISTS (SELECT *
                     FROM NewDFD AS ND2
                   WHERE ND1.diagram_name = ND2.diagram_name
                       AND ND1.bubble_name = ND2.bubble_name
                       AND ND1.flow = ND2.missingflow)
 ORDER BY diagram_name, bubble_name, missingflow;
```

I probably overdid the DISTINCTs, but you can experiment with it for execution speed. This should still run faster than moving all the rows across the network.

PUZZLE
27 FINDING EQUAL SETS

Set theory has two symbols for subsets. One is a "horseshoe" on its side (⊂), which means that set A is contained within set B and is sometimes called a proper subset. The other is the same symbol with a horizontal bar under it (⊆), which means "contained in or equal to," which is sometimes called just a subset or containment operator.

Standard SQL has never had an operator to compare tables against each other. Several college textbooks on relational databases mention a `CONTAINS` predicate that does not exist in standard SQL. Two such offenders are *An Introduction to Data Base Systems* by Bipin C. Desai (West Publishing, 1990, ISBN 0-314-66771-7) and *Fundamentals of Database Systems* by Elmasri and Navthe (Benjamin Cummings, 1989, ISBN 0-8053-0145-3). This predicate used to exist in the original System R, IBM's first experimental SQL system, but it was dropped from later SQL implementations because of the expense of running it.

The `IN()` predicate is a test for membership, not for subsets. For those of you who remember your high school set theory, membership is shown with a stylized epsilon with the containing set of the right side, thus ∈. Membership is for one element, while a subset is itself a set, not just an element.

Chris Date's puzzle in the December 1993 issue of *Database Programming & Design* magazine ("A Matter of Integrity, Part II" According to Date, December 1993) was to use a supplier and parts table to find pairs of suppliers who provide *exactly* the same parts. This is the same thing as finding two equal sets. Given his famous table:

```
CREATE TABLE SupParts
(sno CHAR(2) NOT NULL,
 pno CHAR(2) NOT NULL,
 PRIMARY KEY (sno, pno));
```

How many ways can you find to do this problem?

Answer #1

One approach would be to do a `FULL OUTER JOIN` on each pair of suppliers. Any parts that are not common to both would show up, but would have generated `NULL`s in one of the columns derived from the supplier who was not in the `INNER JOIN` portion. This tells you which

pairs are not matched, not who is. The final step is to remove these nonmatching pairs from all possible pairs.

```
SELECT SP1.sno, SP2.sno
  FROM SupParts AS SP1
       INNER JOIN
       SupParts AS SP2
       ON SP1.pno = SP2.pno
          AND SP1.sno < SP2.sno
EXCEPT
SELECT DISTINCT SP1.sno, SP2.sno
  FROM SupParts AS SP1
      FULL OUTER JOIN
      SupParts AS SP2
       ON SP1.pno = SP2.pno
          AND SP1.sno < SP2.sno)
 WHERE SP1.sno IS NULL
    OR SP2.sno IS NULL;
```

This is probably going to run very slowly. The `EXCEPT` operator is the SQL equivalent of set difference.

Answer #2

The usual way of proving that two sets are equal to each other is to show that set A contains set B, and set B contains set A. What you would usually do in standard SQL would be to show that there exists no element in set A that is not in set B, and therefore A is a subset of B. So the first attempt is usually something like this:

```
SELECT DISTINCT SP1.sno, SP2.sno
 FROM SupParts AS SP1, SupParts AS SP2
 WHERE SP1.sno < SP2.sno
  AND SP1.pno IN (SELECT SP22.pno
                    FROM SupParts AS SP22
                   WHERE SP22.sno = SP2.sno)
  AND SP2.pno IN (SELECT SP11.pno
                    FROM SupParts AS SP11
                   WHERE SP11.sno = SP1.sno));
```

Oops, this does not work because if a pair of suppliers has one item in common, they will be returned.

Answer #3

You can use the `NOT EXISTS` predicate to imply the traditional test mentioned in Answer #2.

```
SELECT DISTINCT SP1.sno, SP2.sno
 FROM SupParts AS SP1, SupParts AS SP2
 WHERE SP1.sno < SP2.sno
  AND NOT EXISTS (SELECT SP3.pno  -- part in SP1 but not in
SP2
                    FROM SupParts AS SP3
                   WHERE SP1.sno = SP3.sno
                     AND SP3.pno
                          NOT IN (SELECT pno
                                    FROM SupParts AS SP4
                                   WHERE SP2.sno = SP4.sno))
  AND NOT EXISTS (SELECT SP5.pno  -- part in SP2 but not in
SP1
                    FROM SupParts AS SP5
                   WHERE SP2.sno = SP5.sno
                     AND SP5.pno
                          NOT IN (SELECT pno
                                    FROM SupParts AS SP4
                                  WHERE SP1.sno = SP4.sno));
```

Answer #4

Instead of using subsets, I thought I would look for another way to do set equality. First, I join one supplier to another on their common parts, eliminating the situation where supplier 1 is the same as supplier 2, so that I have the intersection of the two sets. If the intersection has the same number of pairs as each of the two sets has elements, then the two sets are equal.

```
SELECT SP1.sno, SP2.sno
 FROM SupParts AS SP1
      INNER JOIN
      SupParts AS SP2
      ON SP1.pno = SP2.pno
```

```
         AND SP1.sno < SP2.sno
 GROUP BY SP1.sno, SP2.sno
HAVING (SELECT COUNT(*)   -- one to one mapping EXISTS
          FROM SupParts AS SP3
         WHERE SP3.sno = SP1.sno)
 = (SELECT COUNT(*)
         FROM SupParts AS SP4
        WHERE SP4.sno = SP2.sno);
```

If there is an index on the supplier number in the SupParts table, it can provide the counts directly as well as help with the join operation.

Answer #5

This is the same as Answer #4, but the GROUP BY has been replaced with a SELECT DISTINCT clause:

```
SELECT DISTINCT SP1.sno, SP2.sno
  FROM (SupParts AS SP1
        INNER JOIN
        SupParts AS SP2
        ON SP1.pno = SP2.pno
           AND SP1.sno < SP2.sno)
 WHERE (SELECT COUNT(*)
          FROM SupParts AS SP3
         WHERE SP3.sno = SP1.sno)
 = (SELECT COUNT(*)
         FROM SupParts AS SP4
        WHERE SP4.sno = SP2.sno);
```

Answer #6

This is a version of Answer #3, from Francisco Moreno, which has the NOT EXISTS predicate replaced by set difference. He was using Oracle, and its EXCEPT operator (called MINUS in their SQL dialect) is pretty fast.

```
SELECT DISTINCT SP1.sno, SP2.sno
 FROM SupParts AS SP1, SupParts AS SP2
 WHERE SP1.sno < SP2.sno
  AND NOT EXISTS (SELECT SP3.pno  -- part in SP1 but not in
SP2
                    FROM SupParts AS SP3
```

```
                    WHERE SP1.sno = SP3.sno
                    EXCEPT
                    SELECT SP4.pno
                     FROM SupParts AS SP4
                    WHERE SP2.sno = SP4.sno
  AND NOT EXISTS (SELECT SP5.pno  -- part in SP2 but notin
SP1
                     FROM SupParts AS SP5
                    WHERE SP2.sno = SP5.sno
                    EXCEPT
                    SELECT SP6.pno
                     FROM SupParts AS SP6
                    WHERE SP1.sno = SP6.sno);
```

Answer #7

Alexander Kuznetsov once more has a submission that improves the old "counting matches in a join" approach:

```
SELECT A.sno, B.sno AS sno1
  FROM (SELECT sno, COUNT(*), MIN(pno), MAX(pno)
          FROM SubParts GROUP BY sno)
       AS A(cnt, min_pno, max_pno)
       INNER JOIN
       (SELECT sno, COUNT(*), MIN(pno), MAX(pno)
          FROM SubParts GROUP BY sno)
       AS B(cnt, min_pno, max_pno)
-- four conditions filter out most permutations
       ON A.cnt = B.cnt
          AND A.min_pno = B.min_pno
          AND A.max_pno = B.max_pno
          AND A.sno < B.sno
-- Expensive inner select below does not have to execute for
every pair
 WHERE A.cnt
       = (SELECT COUNT(*)
            FROM SubParts AS A1,
                 SubParts AS B1
           WHERE A1.pno = B1.pno
             AND A1.sno = A.sno
             AND B1.sno = B.sno);
```

```
sn sn
=======
ab bb
aq pq
```

The clever part of this query is that most optimizers can quickly find the MIN() and MAX() values on a column because they are stored in the statistics table.

Answer #8

Let's look at notation and some of the usual tests for equality:

$((A \subseteq B) = (B \subseteq A)) \Rightarrow (A = B)$
$((A \cup B) = (B \cap A)) \Rightarrow (A = B)$

The first equation is really the basis for the comparisons that use joins. The second equation is done at the set level rather than the subset level, and it implies this answer:

```
SELECT DISTINCT 'not equal'
  FROM (SELECT * FROM A)
        INTERSECT
        SELECT * FROM B)
        EXCEPT
       (SELECT * FROM A)
        UNION
        SELECT * FROM B);
```

The idea is to return an empty set if tables A and B are equal. You have to be careful about using the ALL clauses on the set operators if you have duplicates. The good news is that these operators work with rows and not at the column level, so this template will generalize to any pairs of union-compatible tables. You do not have to know the column names.

PUZZLE
28 CALCULATE THE SINE FUNCTION

Let's assume that your SQL product does not have a sine function in its standard library. Can you write a query that will calculate the sine of a number in radians?

Answer #1

Just create a table with all the values you need:

```
CREATE TABLE Sine
(x REAL NOT NULL,
 sin REAL NOT NULL);

INSERT INTO Sine
VALUES (0.00, 0.0000),
 ...
       (0.75, 0.6816),
       (0.76, 0.6889);
 ...
  etc.
```

You can fill in this table with the help of a spreadsheet or a programming language with a good math library. You can now use this table in a scalar subquery:

```
(SELECT sin FROM Sine WHERE x = :myvalue)
```

Of course the table can get pretty big for some functions, but for smaller functions with a limited range of argument values, this is not a bad approach. The sine just happens to be a horrible choice since it is a continuous function defined over all real numbers.

Answer #2

Did you notice that if `:myvalue` in the first answer was not in the table, the subquery would be empty and hence return a `NULL`? This is not good.

If you get out an old calculus or trigonometry book, you will find out how your ancestors used tables in the days before there were calculators.

They had a mathematical technique called interpolation, which came in several flavors.

The easiest method is linear interpolation. Given two known values of a function, f(a) and f(b), you can approximate a third value of the function that lies between them. The formula is:

f(a) + (x-a) * ((f(b) - f(a))/ (b-a))

As an example, assume we want to find sin(0.754)

```
INSERT INTO Sine VALUES (0.75, 0.6816);
INSERT INTO Sine VALUES (0.76, 0.6889);
```

We plug in the formula and get:

```
0.6816 + (0.754 - 0.75) * ((0.6889 - 0.6816)/ (0.76-0.75))
= 0.68452
```

The actual answer is 0.68456, which means we are off by 0.00004, and that is not bad for an estimate in most cases. The trick is to put it in a query:

```
SELECT A.sin + (:myvalue - A.x)
        * ((B.sin - A.sin)/ (B.x - A.x))
  FROM Sine AS A, Sine AS B
 WHERE A.x = (SELECT MAX(x) FROM Sine WHERE x <= :myvalue)
   AND B.x = (SELECT MIN(x) FROM Sine WHERE x >= :myvalue);
```

You really need some more predicates to restrict the range of the function between zero and two pi, but that is a minor detail. There are other interpolation methods, but the idea is the same.

The lesson here is that SQL is a language designed to work with tables and joins, not computations. You should look for a table solution before you use a computational one. If you really want to get into the math behind interpolation, I would suggest a copy of *Interpolation* by J. F. Steffensen (Dover Publications, 2006, ISBN 0-486-45009-0).

PUZZLE
29 FIND THE MODE COMPUTATION

The only descriptive statistical function in SQL is the simple average, AVG(). While it is a common statistic, it is not the only one. The mean, the median, and the mode are all ways of measuring "central tendency" in a set of values. The mode is the most common value in a column in a table. Let's suppose that the table is named "Payroll" and has the check_nbr number and the amount of each check_nbr.

```
CREATE TABLE Payroll
(check_nbr INTEGER NOT NULL PRIMARY KEY,
 check_amt DECIMAL(8,2) NOT NULL,
 ... );
```

What we want to see is the most common check amount and the number of occurrences on the payroll. How would you write this query in SQL-89? In SQL-92? In SQL-99?

Answer #1

SQL-89 lacks the orthogonality that SQL-92 has, so the best way is probably to build a VIEW first:

```
CREATE VIEW AmtCounts
AS SELECT COUNT(*) AS check_cnt
     FROM Payroll
    GROUP BY check_amt;
```

then use the VIEW to find the most frequent check_amt amount:

```
SELECT check_amt, COUNT(*)
  FROM Payroll
 GROUP BY check_amt
HAVING COUNT(*) = (SELECT MAX(check_cnt)
                     FROM AmtCounts);
```

But this solution leaves a VIEW lying around the database schema. If you need it for something else, this is handy, but otherwise it is clutter. It would be better to do this in one statement without VIEWs.

Answer #2

The orthogonality of SQL-92 will allow you to fold the VIEW into a tabular subquery, thus:

```
SELECT check_amt, COUNT(*)AS check_cnt
  FROM Payroll
 GROUP BY check_amt
HAVING COUNT(*) = (SELECT MAX(check_cnt)
                     FROM (SELECT COUNT(*) AS check_cnt
                             FROM Payroll
                            GROUP BY check_amt));
```

The innermost SELECT statement has to be expanded completely before it passes the grouped table to its immediate containing SELECT statement. That statement finds the MAX() and then passes that single number to the outermost SELECT. There is a very good chance that the grouped table will be destroyed in this process.

If the optimizer were smart, it would have saved the first query to reuse in the final answer, but don't bet on it. Let's keep looking.

Answer #3

Here is another SQL-92 solution that will handle NULLs a bit differently than the last solution; can you tell me what the differences are?

```
SELECT PO.check_amt, COUNT(*)AS check_cnt
  FROM Payroll
 GROUP BY check_amt
HAVING COUNT(*) >= ALL (SELECT COUNT(*) AS check_cnt
                          FROM Payroll
                         GROUP BY check_amt);
```

The possible advantage of this answer is that since no MAX() function is used, there is a better chance that the grouped table will be preserved from one SELECT to be used by the other. Notice that the innermost SELECT is a projection of the outermost SELECT.

You should try all three solutions to see how your particular SQL implementation will perform with them.

Answer #4

You will find that many of the current versions of SQL have a `mode()` function in them now as part of the upcoming OLAP extensions, so this is not much of a question anymore. We can effectively replace the subquery with an OLAP function call.

```
SELECT check_amt,
       COUNT(*)OVER (PARTITION BY check_amt)
       AS check_cnt
  FROM Payroll;
```

However, I do not know if there is any particular performance advantage to this.

PUZZLE
30 AVERAGE SALES WAIT

Raymond Petersen asked me the following question: Given a Sales table with just the date of the sale and customer columns, is there any way to calculate the average number of days between sales for each customer in a single SQL statement? Use a simple table in which you can assume that nobody makes a sale to the same person on the same day:

```
CREATE TABLE Sales
(customer_name CHAR(5) NOT NULL,
 sale_date DATE NOT NULL,
 PRIMARY KEY (customer_name, sale_date));
```

Let's take a look at the date for the first week in June 1994:

```
Sales
customer_name  sale_date
=========================
'Fred'     '1994-06-01'
'Mary'     '1994-06-01'
'Bill'     '1994-06-01'
'Fred'     '1994-06-02'
'Bill'     '1994-06-02'
'Bill'     '1994-06-03'
'Bill'     '1994-06-04'
'Bill'     '1994-06-05'
'Bill'     '1994-06-06'
'Bill'     '1994-06-07'
'Fred'     '1994-06-07'
'Mary'     '1994-06-08'
```

The data shows that Fred waited one day, then waited five days, for an average of three days between his visits. Mary waited seven days, for an average of seven days. Bill is a regular customer every day.

Answer #1

The first impulse is to construct an elaborate `VIEW` that shows the number of days between each purchase for each customer. The first task

in this approach is to get the sales into a table with the current `sale_date` and the date of the last purchase:

```
CREATE VIEW Lastsales (customer_name, this_sale_date,
last_sale_date)
 AS SELECT S1.customer_name, S1.saledate,
     (SELECT MAX(sale_date)
      FROM Sales AS S2
      WHERE S2.saledate < S1.saledate
        AND S2.customer_name = S1.customer_name)
   FROM Sales AS S1, Sales AS S2;
```

This is a greatest lower bound query—we want the highest date in the set of dates for this customer that comes before the current date.

Now we construct a `VIEW` with the gap in days between this sale and the customer's last purchase. You could combine the two views into one statement, but it would be unreadable and would not optimize any better. Just to keep the code simple, assume that we have a `DAYS()` function that returns an integer to do the temporal math.

```
CREATE VIEW SalesGap (customer_name, gap)
AS
SELECT customer, DAYS(this_sale_date, last_sale_date)
  FROM Lastsales;
```

The final answer is one query:

```
 SELECT customer, AVG(gap)
   FROM SalesGap
  GROUP BY customer_name;
```

You could combine the two views into the `AVG()` parameter, but it would be totally unreadable, might blow up, and would run like molasses.

Answer #2

I showed you Answer #1 because it demonstrates how you can be too smart for your own good. Because we only look for the average number of days a customer waits between purchases, there is no need to build an

elaborate VIEW. Simply count the number of lapsed days and then divide by the number of sales.

```
SELECT customer_name, DAYS(MAX(sale_date) - MIN(sale_date))
/ (COUNT(*)-1.0) AS avg_gap
  FROM Sales
 GROUP BY customer
HAVING COUNT(*) > 1;
```

The (COUNT(*) -1.0) works because there is always one less gap than orders if you do not consider the time gap between the date of the last order and today's date. The decimal will cast the results to a numeric rather than an integer. The HAVING clause will remove from consideration customers who have made only one purchase. These one-shot customers can be included by changing MAX(sale_date) to CURRENT_DATE in the SELECT statement.

Incidentally, with either approach, you can have more than one sale per day per customer.

PUZZLE
31 BUYING ALL THE PRODUCTS

Software AG introduced an intelligent SQL query-writing product called Esperant in the mid-1990s. Using the keyboard and an interactive pick list, the user constructs an English sentence, which the machine turns into a series of target SQL queries.

Yes, natural-language queries are an old idea, but most of them have involved some preprogramming of English phrases to make them work. The amount of work that Esperant can do by itself is what makes it worth looking at. It will generate relational divisions, create views, and build complex transactions without any preprogramming.

Software AG's demo had a typical schema with tables of customers, orders, and order details.

```
CREATE TABLE Customers
(customer_id INTEGER NOT NULL PRIMARY KEY,
 acct_balance DECIMAL (12, 2) NOT NULL,
 ... );

CREATE TABLE Orders
(customer_id INTEGER NOT NULL,
 order_id INTEGER NOT NULL PRIMARY KEY,
 ... );

CREATE TABLE OrderDetails
(order_id INTEGER NOT NULL,
 item_id INTEGER NOT NULL,
 PRIMARY KEY(order_id, item_id),
 item_qty INTEGER NOT NULL,
 ... );

CREATE TABLE Products
(item_id INTEGER NOT NULL PRIMARY KEY,
 item_qty_on_hand INTEGER NOT NULL,
... );
```

Part of one sample problem was to find the average customer `acct_balance` for all customers who had orders for all products, and the average customer `acct_balance` for all customers who did not have orders for all of the products. Esperant did an impressive job, but it

generated a lot of VIEWs for portability. Using some of the SQL-92 constructs or making better use of the old SQL-89 constructs, can you improve this query?

Answer #1

The traditional answer is to use a deeply nested query. This query would translate into "Find the average for the set of customers for whom there is a product that is not in their orders" in English.

```
SELECT AVG(acct_balance)
  FROM Customers AS C1
 WHERE EXISTS
       (SELECT *
         FROM Products AS P1
         WHERE P1.item_id
               NOT IN (SELECT D1.item_id
                        FROM Orders AS O1, OrderDetails AS
D1
                         WHERE O1.customer_id =
C1.customer_id
                          AND O1.order_id = D1.order_id));
```

To get the average account balance of the customers, you could change the EXISTS() to NOT EXISTS().

Answer #2

Gillian Robertson, of Worcestershire, England, found a neat trick that saves some of the nesting of correlated subquery.

```
SELECT AVG(acct_balance)
  FROM Customers AS C1
 WHERE (SELECT COUNT(DISTINCT item_id) -- all products we
sell ...
          FROM Products)
       <> (SELECT COUNT(DISTINCT item_id)  -- versus what
he bought
             FROM Orders, OrderDetails
            WHERE Orders.customer_id = C1.customer_id
              AND Orders.order_id = OrderDetails.order_id);
```

This will find the average account balance of all customers who do not buy all products, by ensuring that the number of DISTINCT items that show up in their order details is not the number of DISTINCT items in the products list. Obviously, changing "<>" to "=" returns the customers who did order everything we sell.

Answer #3

Alex Kuznetsov realized that we need both answers (for those who ordered all products and for those who did not order all products), and we can get them in one query—simpler and better performing than issuing two queries.

```
SELECT AVG(acct_balance), ordered_all_desc
  FROM SELECT Customers.customer_id, acct_balance,
              CASE WHEN num_ordered_products =
num_all_products
              THEN 'ordered all'
              ELSE 'not ordered all' END
              AS ordered_all_desc
         FROM Customers
   INNER JOIN
   (SELECT customer_id, COUNT(DISTINCT item_id)
num_ordered_products
FROM Orders
     INNER JOIN
    OrderDetails ON Orders.order_id = OrderDetails.order_id

GROUP BY customer_id
) AS ordered_products
ON Customers.customer_id = ordered_products.customer_id
CROSS JOIN
(SELECT COUNT(DISTINCT item_id)
  FROM Products)AS AllProducts (all_product_cnt)
) AS T
GROUP BY ordered_all_desc;
```

PUZZLE
32 COMPUTING TAXES

Richard Romley sent this problem via CompuServe. This is a simplified version of a problem relating to tax calculations. I will define a tax area as being made up of multiple tax authorities. For example, a tax area might be a city, and the tax authorities for that city might be the city, the city's county, and the state. When you pay tax on a purchase in the city, the tax rate you pay is made up of the city tax, the county tax, and the state tax. Each of these taxing authorities changes its tax rate independently.

You have the following table:

```
CREATE TABLE TaxAuthorities
(tax_authority CHAR(10) NOT NULL,
 tax_area CHAR(10) NOT NULL,
 PRIMARY KEY (tax_authority, tax_area));
```

This is a hierarchy in which each tax area pays the multiple tax authorities to which it belongs.

```
TaxAuthories
tax_authority  tax_area
=======================
 'city1'       'city1'
 'city2'       'city2'
 'city3'       'city3'
 'county1'     'city1'
 'county1'     'city2'
 'county2'     'city3'
 'state1'      'city1'
 'state1'      'city2'
 'state1'      'city3'
```

This means that `city1` and `city2` are in `county1` of `state1`; `city3` is in `county2` of `state1`, and so forth. The other table you need is the tax rates, as follows. There is an assumption that the rates are additive in a direct, simple fashion.

```
CREATE TABLE TaxRates
(tax_authority CHAR(10) NOT NULL,
```

```
 effect_date DATE NOT NULL,
 tax_rate DECIMAL (8,2) NOT NULL,
 PRIMARY KEY (tax_authority, effect_date));
```

Populate this table as follows:

```
TaxRates
tax_authority  effect_date  tax_rate
=====================================
 'city1'       '1993-01-01'     1.0
 'city1'       '1994-01-01'     1.5
 'city2'       '1993-09-01'     1.5
 'city2'       '1994-01-01'     2.0
 'city2'       '1995-01-01'     2.0
 'city3'       '1993-01-01'     1.7
 'city3'       '1993-07-01'     1.9
 'county1'     '1993-01-01'     2.3
 'county1'     '1994-10-01'     2.5
 'county1'     '1995-01-01'     2.7
 'county2'     '1993-01-01'     2.4
 'county2'     '1994-01-01'     2.7
 'county2'     '1995-01-01'     2.8
 'state1'      '1993-01-01'     0.5
 'state1'      '1994-01-01'     0.8
 'state1'      '1994-07-01'     0.9
 'state1'      '1994-10-01'     1.1
```

This table is to be used for answering problems such as "What is the total tax rate for `city2` on November 1, 1994?" for the tax collector. The answer for this particular question would be:

```
city2 tax rate   = 2.0
county1 tax rate = 2.5
state1 tax rate  = 1.1
-----------------------
Total tax rate   = 5.6
```

Can you write a single SQL query to answer this question?

Answer #1

It is best to solve this problem in pieces. First, you want to find out who the taxing authorities for the city are, so you write a subquery:

```
(SELECT tax_authority
   FROM TaxAuthories AS A1
  WHERE A1.tax_area = 'city2')
```

which will result in the set (`'city2'`, `'county1'`, `'state1'`).

Next, you want to find out what the tax rates were on November 1, 1994, so you write another subquery:

```
(SELECT tax_authority, tax_rate
   FROM TaxRates AS R1
  WHERE R1.effect_date = (SELECT MAX (R2.effect_date)
                            FROM TaxRates AS R2
                           WHERE R2.effect_date <= '1994-11-01'))
```

Now, combine the two subqueries, do a summation, and put your constants in the `SELECT` list to make the final answer readable. Actually, I would change these constants into parameters to generalize the routine, but for now, let's stick to the original problem:

```
SELECT 'city2' AS city, '1994-11-01' AS effect_date,
       SUM (tax_rate) AS total_taxes
  FROM TaxRates AS R1
 WHERE R1.effect_date =
        (SELECT MAX (R2.effect_date)
           FROM TaxRates AS R2
          WHERE R2.dteffective <= '1994-11-01'
            AND R1.tax_authority = R2.tax_authority)
           AND R2.tax_authority IN (SELECT tax_authority
                                      FROM TaxAuthoriies AS A1
                                        WHERE A1.tax_area =
'city2')
 GROUP BY city, effect_date;
```

But wait! You can do more consolidation and move the second `AND` predicate to a deeper level of nesting, like this:

```
SELECT 'city2' AS city, '1994-11-01' AS effective_date,
        SUM(tax_rate) AS total_taxes
  FROM TaxRates AS R1
 WHERE R1.effect_date =
              (SELECT MAX (R2.effect_date)
                 FROM TaxRates AS R2
                WHERE R2.dteffective <= '1994-11-01'
                  AND R1.tax_authority = R2.tax_authority
                  AND R2.tax_authority
                        IN (SELECT tax_authority
                              FROM TaxAuthories AS A1
                             WHERE A1.tax_area = 'city2'
  GROUP BY city, effect_date;
```

Because the subquery is a noncorrelated, constant list, performance should be pretty good. And sure enough, when I look at the execution plan in WATCOM SQL, I found that the `R1` and `R2` tables were sequentially scanned, but the `A1` table used the primary key index. If I put indexes on the `TaxRates` table, I can get an even faster execution plan.

Answer #2

Diosdado Nebres, of Washington state, sent in an alternative solution to this puzzler:

```
SELECT SUM(T2.authtaxrate)
  FROM TaxAuthoriies AS T1, TaxRates AS T2
 WHERE T1.tax_area = 'city2'
   AND T2.tax_authority = T1.tax_authority
   AND T2.effect_date =
           (SELECT MAX(effect_date)
              FROM TaxRates
             WHERE tax_authority = T2.tax_authority
               AND effect_date <= '1994-11-01');
```

He eliminated the `GROUP BY`, which is a good move since the query will work as well, if not better, without it. And he replaced the deepest level of nesting with a `JOIN` between `TaxAuthority` and `TaxRates`. That greatly reduces the number of times the first subquery is executed.

Answer #3

Most current SQL programmers will recognize that the taxing authorities are in a hierarchy and would use a nested sets model. I will not bother to explain nested sets and the use of (lft, rgt) pairs for modeling hierarchies (see my book, *Joe Celko's Trees and Hierarchies in SQL for Smarties,* ISBN 1-55860-920-2), but the two tables are replaced by a single table:

```
CREATE TABLE TaxRates
(tax_authority CHAR(10) NOT NULL,
 lft INTEGER NOT NULL CHECK (lft > 0),
 rgt INTEGER NOT NULL,
  CHECK (lft < rgt),
 start_date DATE NOT NULL
 end_date DATE,  -- null is current rate
 tax_rate DECIMAL(8,2) NOT NULL,
 PRIMARY KEY (tax_authority, start_date)
);
```

The date pairs are the time ranges that a tax rate was in effect. They have to be nonoverlapping. The reason for using the ranges is so that historical rates can be computed more easily.

```
SELECT SUM(T2.taxrate)
  FROM TaxRates AS T1, TaxRates AS T2
 WHERE T1.tax_authority = :my_location
   AND :my_date BETWEEN T1.prior_date
               AND COALESCE (T1.effect_date, CURRENT_DATE)
   AND T1.lft BETWEEN T2.lft AND T2.rgt;
```

The `COALESCE()` will handle the current tax rates, if we do not have a future date for their expiration.

PUZZLE
33 COMPUTING DEPRECIATION

This is based on a problem posted by Gerhard F. Jilovec on CompuServe. He had a manufacturing company database from which he wished to compute depreciation on the machinery. To this end, his database has a table of machines, like this:

```
CREATE TABLE Machines
(machine_name CHAR(20) NOT NULL PRIMARY KEY,
 purchase_date DATE NOT NULL,
 initial_cost DECIMAL (10,2) NOT NULL,
 lifespan INTEGER NOT NULL);
```

Where the column `purchase_date` is just what you think—the purchase date of that machine. The `initial_cost` column is the initial cost of the machine. The lifespan column is the expected lifespan of the equipment given in days.

There is also a table of the cost of using a particular machine on a particular batch of work, defined as:

```
CREATE TABLE ManufactCosts
(machine_name CHAR(20) NOT NULL
    REFERENCES Machinery(machine_name),
 manu_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 manu_cost DECIMAL (6,2) NOT NULL,
 PRIMARY KEY (machine_name, manu_date, batch_nbr));
```

Where the `manu_date` column is the date that a particular batch was processed on that machine. The `manu_cost` is what it cost to manufacture that batch. A similar table of manufacturing hours tells us how much time each batch took. It looks like this:

```
CREATE TABLE ManufactHrs
(machine_name CHAR(20) NOT NULL REFERENCES Machines,
 manu_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 manu_hrs DECIMAL(4,2) NOT NULL,
 PRIMARY KEY (machine_name, manu_date, batch_nbr));
```

Your problem is to suggest a better design for the database. Then you are to write a query that will give us the average hourly cost of each machine to date for any day we choose.

Answer #4

Time and money were in separate tables in the original design because the data was collected separately from time cards and from the accounting department.

You should put manufacturing cost (`manu_cost`) and manufacturing hours (`manu_hrs`) in a single table, keyed by the machine, the date, and the batch number. If you can have hours without knowing the cost or cost without knowing the hours, your design might allow `NULL`s in those columns, but you will still have to watch your math. I would replace the two tables with:

```
CREATE TABLE ManufactHrsCosts
(machine_name CHAR(20) NOT NULL
     REFERENCES Machines(machine_name),
 manu_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 manu_hrs DECIMAL(4,2) NOT NULL,
 manu_cost DECIMAL (6,2) NOT NULL,
 PRIMARY KEY (machine_name, manu_date, batch_nbr));
```

Let's do an example with some data. We just bought a frammis cutter for $10,000 five days ago, and we were able to run seven batches on it. The lifetime for a frammis cutter is 1,000 days.

```
ManufactHrsCosts
machine_name manu_date    batch_nbr manu_hrs  manu_cost
=========================================================
'Frammis'   '1995-07-24'   101   2.5     123.00
'Frammis'   '1995-07-25'   102   2.5     125.00
'Frammis'   '1995-07-25'   103   2.0     110.00
'Frammis'   '1995-07-26'   104   2.5     125.00
'Frammis'   '1995-07-27'   105   2.5     120.00
'Frammis'   '1995-07-27'   106   2.5     120.00
'Frammis'   '1995-07-28'   107   2.5     125.00
```

On July 24, the first day of use, the average hourly cost was ($123.00 / 2.5) + $10.00 = $59.20 per hour. But by July 25, the second day of use, the average hourly cost was ($123.00 + $125.00 + $110.00 +(2 * $10.00))/(2.5 + 2.5 + 2.0 hrs) = $55.43, a considerable reduction. At the end of the first five days, the hourly cost is $52.82 for the frammis cutter.

While you could do this with other approaches, I like to create a `VIEW` for total cost and hours. I can use it for other daily reports.

```
CREATE VIEW TotHrsCosts (machine_name, manu_date, day_cost,
day_hrs)
 AS SELECT machine_name, manu_date, SUM(manu_cost),
SUM(manu_hrs)
      FROM ManufactHrsCosts
     GROUP BY machine_name, manu_date;
```

Let's assume we can compute the number of days between two `DATE` variables by subtraction. After that, your query is simply:

```
SELECT :mydate, M1.machine_name,
 (((initial_cost/lifespan) -- amortized cost per day
  * (:mydate - M1.purchase_date + 1)) -- days of life so
far)
 -- add the average hourly cost
  + (SELECT SUM(THC.daycost)/SUM(THC.dayhrs)
       FROM TotHrsCosts AS THC
      WHERE M1.machine_name = THC.machine_name)) AS
hourly_cost
 FROM Machines AS M1
WHERE :mydate BETWEEN M1.purchase_date AND manu_date;
```

Think about the `WHERE` clause predicate for a moment; it is a nice trick to avoid negative values in the first part of the calculations for hourly cost.

Answer #5

This came from Francisco Moreno, when he was a student in Colombia, who found a short solution, avoiding the view and the scalar subquery:

```
SELECT MAX(:mydate) AS my_date,
       F.machine_name,
```

```
       (MAX ((initial_cost/lifespan) *
              (:mydate - purchase_date + 1)
             ) + SUM(manucost)
       ) / SUM(manu_hrs) AS average_hour
FROM ManufactCosts AS F,
     Machines AS M
WHERE M.machine_name = F.machine_name
  AND purchase_date <= :mydate
  AND manu_date <= :mydate
GROUP BY F.machine_name;
```

PUZZLE
34 CONSULTANT BILLING

Brian K. Buckley posted a version of the following problem in November 1994, requesting assistance. He has three tables, declared as:

```
CREATE TABLE Consultants
(emp_id INTEGER NOT NULL,
 emp_name CHAR(10) NOT NULL);

INSERT INTO Consultants
VALUES (1, 'Larry'),
       (2, 'Moe'),
       (3, 'Curly');

CREATE TABLE Billings
(emp_id INTEGER NOT NULL,
 bill_date DATE NOT NULL,
 bill_rate DECIMAL (5,2));

INSERT INTO Billings
VALUES (1, '1990-01-01', 25.00);
       (2, '1989-01-01', 15.00),
       (3, '1989-01-01', 20.00),

       (1, '1991-01-01', 30.00);

CREATE TABLE HoursWorked
(job_id INTEGER NOT NULL,
 emp_id INTEGER NOT NULL,
 work_date DATE NOT NULL,
 bill_hrs DECIMAL(5, 2));

INSERT INTO HoursWorked
VALUES (4, 1, '1990-07-01', 3),
       (4, 1, '1990-08-01', 5),
       (4, 2, '1990-07-01', 2),
       (4, 1, '1991-07-01', 4);
```

He wanted a single query that would show a list of names and total charges for a given job. Total charges are calculated for each employee as

the hours worked multiplied by the applicable hourly billing rate. For example, the sample data shown would give the following answer:

```
Results
name   totalcharges
===================
'Larry'   320.00
'Moe'      30.00
```

since Larry would have ((3+5) hours * $25 rate + 4 hours * $30 rate) = $320.00 and Moe (2 hours * $15 rate) = $30.00.

Answer #1

I think the best way to do this is to build a `VIEW`, then summarize from it. The `VIEW` will be handy for other reports. This gives you the `VIEW`:

```
CREATE VIEW HourRateRpt (emp_id, emp_name, work_date,
bill_hrs, bill_rate)
AS
SELECT H1.emp_id, emp_name, work_date, bill_hrs,
       (SELECT bill_rate
          FROM Billings AS B1
         WHERE bill_date = (SELECT MAX(bill_date)
                                  FROM Billings AS B2
                                 WHERE B2.bill_date <=
H1.work_date
                                       AND B1.emp_id = B2.emp_id
                                        AND B1.emp_id =
H1.emp_id)))
     FROM HoursWorked AS H1, Consultants AS C1
    WHERE C1.emp_id = H1.emp_id;
```

Then your report is simply:

```
SELECT emp_id, emp_name, SUM(bill_hrs * bill_rate) AS
bill_tot
  FROM HourRateRpt
 GROUP BY emp_id, emp_name;
```

But since Mr. Buckley wanted it all in one query, this would be his requested solution:

```
SELECT C1.emp_id, C1.emp_name, SUM(bill_hrs) *
(SELECT bill_rate
   FROM Billings AS B1
  WHERE bill_date = (SELECT MAX(bill_date)
                       FROM Billings AS B2
                      WHERE B2.bill_date <= H1.work_date
                        AND B1.emp_id = B2.emp_id
                        AND B1.emp_id = H1.emp_id))
  FROM HoursWorked AS H1, Consultants AS C1
 WHERE H1.emp_id = C1.emp_id
 GROUP BY C1.emp_id, C1.emp_name;
```

This is not an obvious answer for a beginning SQL programmer, so let's talk about it. Start with the innermost query, which picks the effective date of each employee that immediately occurred before the date of this billing. The next level of nested query uses this date to find the billing rate that was in effect for the employee at that time; that is why the outer correlation name `B1` is used. Then, the billing rate is returned to the expression in the `SUM()` function and multiplied by the number of hours worked. Finally, the outermost query groups each employee's billings and produces a total.

Answer #2

Linh Nguyen sent in another solution:

```
SELECT name, SUM(H1.bill_hrs * B1.bill_rate)
  FROM Consultants AS C1, Billings AS B1, Hoursworked AS H1
 WHERE C1.emp_id = B1.emp_id
   AND C1.emp_id = H1.emp_id
   AND bill_date = (SELECT MAX(bill_date)
                      FROM Billings AS B2
                     WHERE B2.emp_id = C1.emp_id
                       AND B2.bill_date <= H1.work_date)
   AND H1.work_date >= bill_date
   GROUP BY name;
```

This version of the query has the advantage over the first solution in that it does not depend on subquery expressions, which are often slow. The moral of the story is that you can get too fancy with new features.

PUZZLE
35 INVENTORY ADJUSTMENTS

This puzzle is a quickie in SQL-92, but was very hard to do in SQL-89. Suppose you are in charge of the company inventory. You get requisitions that tell how many widgets people are putting into or taking out of a warehouse bin on a given date. Sometimes the quantity is positive (returns), and sometimes the quantity is negative (withdrawals).

```
CREATE TABLE InventoryAdjustments
(req_date DATE NOT NULL,
 req_qty INTEGER NOT NULL
    CHECK (req_qty <> 0),
 PRIMARY KEY (req_date, req_qty));
```

Your job is to provide a running balance on the quantity-on-hand as an SQL column. Your results should look like this:

```
Warehouse
req_date    req_qty    onhand_qty
================================
'1994-07-01'    100     100
'1994-07-02'    120     220
'1994-07-03'   -150      70
'1994-07-04'     50     120
'1994-07-05'    -35      85
```

Answer #1

SQL-92 can use a subquery in the SELECT list, or even a correlated query. The rules are that the result must be a single value (hence the name "scalar subquery"); if the query results are an empty table, the result is a NULL. This interesting feature of the SQL-92 standard sometimes lets you write an OUTER JOIN as a query within the SELECT clause. For example, the following query will work only if each customer has one or zero orders:

```
SELECT cust_nbr, cust_name,
            (SELECT order_amt
               FROM Orders
              WHERE Customers.cust_nbr = Orders.cust_nbr)
```

```
 FROM Customers;
```

and give the same result as:

```
SELECT cust_nbr, cust_name, order_amt
  FROM Customers
       LEFT OUTER JOIN
       Orders
       ON Customers.cust_nbr = Orders.cust_nbr;
```

In this problem, you must sum all the requisitions posted up to and including the date in question. The query is a nested self-join, as follows:

```
SELECT req_date, req_qty,
       (SELECT SUM(req_qty)
          FROM InventoryAdjustments AS A2
         WHERE A2.req_date <= A1.req_date)
       AS req_onhand_qty
  FROM iInventoryAdjustments AS A1
 ORDER BY req_date;
```

Frankly, this solution will run slowly compared to a procedural solution, which could build the current quantity-on-hand from the previous quantity-on-hand from a sorted file of records.

Answer #2

Jim Armes at Trident Data Systems came up with a somewhat easier solution than the first answer:

```
SELECT A1.req_date, A1.req_qty, SUM(A2.req_qty) AS
req_onhand_qty
  FROM InventoryAdjustments AS A2, InventoryAdjustments AS
A1
 WHERE A2.req_date <= A1.req_date
 GROUP BY A1.req_date, A1.req_qty
 ORDER BY A1.req_date;
```

This query works, but becomes too costly. Assume you have (n) requisitions in the table. In most SQL implementations, the `GROUP BY`

clause will invoke a sort. Because the GROUP BY is executed for each requisition date, this query will sort one row for the group that belongs to the first day, then two rows for the second day's requisitions, and so forth until it is sorting (n) rows on the last day.

The "SELECT within a SELECT" approach in the first answer involves no sorting, because it has no GROUP BY clause. Assuming no index on the requisition date column, the subquery approach will do the same table scan for each date as the GROUP BY approach does, but it could keep a running total as it does. Thus, we can expect the "SELECT within a SELECT" to save us several passes through the table.

Answer #3

The SQL:2003 standards introduced OLAP functions that will give you running totals as a function. The old SQL-92 scalar subquery becomes a function. There is even a proposal for a MOVING_SUM() option, but it is not widely available.

```
SELECT req_date, req_qty,
       SUM(req_qty)
       OVER (ORDER BY req_date DESC
             ROWS UNBOUNDED PRECEDING))
       AS req_onhand_qty
  FROM InventoryAdjustments
 ORDER BY req_date;
```

This is a fairly compact notation, but it also explains itself. I take the requisition date on the current row, and I total all of the requisition quantities that came before it in descending date order. This has the same effect as the old scalar subquery approach. Which would you rather read and maintain?

Notice also that you can change SUM() to AVG() or other aggregate functions with that same OVER() window clause. At the time of this writing, these are new to SQL, and I am not sure as to how well they are optimized in actual products.

PUZZLE
36 DOUBLE DUTY

Back in the early days of CompuServe, Nigel Blumenthal posted a notice that he was having trouble with an application. The goal was to take a source table of the roles that people play in the company, where 'D' means the person is a Director, 'O' means the person is an Officer, and we do not worry about the other codes. We want to produce a report with a code 'B', which means the person is both a Director and an Officer. The source data might look like this when you reduce it to its most basic parts:

```
Roles
person  role
=============
'Smith'   'O'
'Smith'   'D'
'Jones'   'O'
'White'   'D'
'Brown'   'X'
```

and the result set will be:

```
Result
person  combined_role
=====================
'Smith'    'B'
'Jones'    'O'
'White'    'D'
```

Nigel's first attempt involved making a temporary table, but this was taking too long.

Answer #1

Roy Harvey's first reflex response—written without measurable thought—was to use a grouped query. But we need to show the double-duty guys and the people who were just 'D' or just 'O' as well. Extending his basic idea, you get:

```
 SELECT R1.person, R1.role
   FROM Roles AS R1
  WHERE R1.role IN ('D', 'O')
  GROUP BY R1.person
 HAVING COUNT(DISTINCT R1.role) = 1
UNION
 SELECT R2.person, 'B'
   FROM Roles AS R2
  WHERE R2.role IN ('D', 'O')
  GROUP BY R2.person
 HAVING COUNT(DISTINCT R2.role) = 2
```

but this has the overhead of two grouping queries.

Answer #2

Leonard C. Medal replied to this post with a query that could be used in a `VIEW` and save the trouble of building the temporary table. His attempt was something like this:

```
SELECT DISTINCT R1.person,
 CASE WHEN EXISTS (SELECT *
                     FROM Roles AS R2
                    WHERE R2.person = R1.person
                      AND R2.role IN ('D', 'O'))
       THEN 'B'
       ELSE (SELECT DISTINCT R3.role
               FROM Roles AS R3
              WHERE R3.person = R1.person
                AND R3.role IN ('D', 'O'))
      END AS combined_role
  FROM Roles AS R1
 WHERE R1.role IN ('D', 'O');
```

Can you come up with something better?

Answer #3

I was trying to mislead you into trying self-joins. Instead you should avoid all those self-joins in favor of a `UNION`. The employees with a dual role will appear twice, so you are just looking for a row count of two.

```
SELECT R1.person, MAX(R1.role)
  FROM Roles AS R1
 WHERE R1.role IN ('D','O')
 GROUP BY R1.person
HAVING COUNT(*) = 1
UNION
SELECT R2.person, 'B'
  FROM Roles AS R2
 WHERE R2.role IN ('D','O')
 GROUP BY R2.person
HAVING COUNT(*) = 2;
```

In SQL-92, you will have no trouble putting a `UNION` into a `VIEW`, but some older SQL products may not allow it.

Answer #4

SQL-92 has a `CASE` expression and you can often use it as replacement. This leads us to the final simplest form:

```
SELECT person,
       CASE WHEN COUNT(*) = 1
            THEN role
            ELSE 'B' END
  FROM Roles
 WHERE role IN ('D','O')
 GROUP BY person;
```

The clause "`THEN role`" will work since we know that it is unique within a person because it has a count of 1. However, some SQL products might want to see "`THEN MAX(role)`" instead because "`role`" was not used in the `GROUP BY` clause, and they would see this as a syntax violation between the `SELECT` and the `GROUP BY` clauses.

Answer #5

Here is another trick with a `CASE` expression and a `GROUP BY`:

```
SELECT person,
       CASE WHEN MIN(role) <> MAX(role)
            THEN 'B' ELSE MIN(role) END
```

```
       AS combined_role
  FROM Roles
 WHERE role IN ('D','O')
 GROUP BY person;
```

Answer #6

Mark Wiitala used another approach altogether. It was the fastest answer available when it was proposed.

```
SELECT person,
       SUBSTRING ('ODB' FROM SUM (POSITION (role IN 'DO'))
FOR 1)
  FROM Person_Role
 WHERE role IN ('D','O')
 GROUP BY person;
```

This one takes some time to understand, and it is confusing because of the nested function calls. For each group formed by a person's name, the `POSITION()` function will return a 1 for `'D'` or a 2 for `'O'` in the role column. The `SUM()` of those results is then used in the `SUBSTRING()` function to convert a 1 back to `'D'`, a 2 back to `'O'`, and a 3 into `'B'`. This is a rather interesting use of conjugacy, the mathematical term where you use a transform and its inverse to make a problem easier. Logarithms and exponential functions are the most common examples.

PUZZLE
37 A MOVING AVERAGE

You are collecting statistical information stored by the quarter hour. What your customer wants is to get information by the hour—not on the hour. That is, we don't want to know what the load was at 00:00 hours, at 01:00 hours, at 02:00 hours, and so forth. We want the average load for the first four quarter hours (00:00, 00:15, 00:30, 00:45), for the next four quarter hours (00:15, 00:30, 00:45, 01:00), and so forth. This is called a moving average, and we will assume that the sample table looks like this:

```
CREATE TABLE Samples
(sample_time TIMESTAMP NOT NULL PRIMARY KEY,
 load REAL NOT NULL);
```

Answer #1

One way is to add another column to hold the moving average:

```
CREATE TABLE Samples
(sample_time TIMESTAMP NOT NULL PRIMARY KEY,
 moving_avg REAL NOT NULL DEFAULT 0
 load REAL DEFAULT 0 NOT NULL);
```

then update the table with a series of statements, like this:

```
UPDATE Samples
   SET moving_avg
       = (SELECT AVG(S1.load)
            FROM Samples AS S1
           WHERE S1.sample_time
                 IN (Samples.sample_time,
                     (Samples.sample_time - INTERVAL 15
MINUTES),
                     (Samples.sample_time - INTERVAL 30
MINUTES),
                     (Samples.sample_time - INTERVAL 45
MINUTES));
```

Answer #2

However, this is not the only way to write the UPDATE statement. The assumption that we are sampling exactly every 15 minutes is probably not true; there will be some sampling errors, so the timestamps could be a few minutes off. We could try for the hour time slot, instead of an exact match:

```
UPDATE Samples
   SET moving_avg
       = (SELECT AVG(S1.load)
            FROM Samples AS S1
           WHERE S1.sample_time
                 BETWEEN (Samples.sample_time - INTERVAL 1
HOUR)
                     AND Samples.sample_time);
```

Answer #3

That last update attempt suggests that we could use the predicate to construct a query that would give us a moving average:

```
SELECT S1.sample_time, AVG(S2.load) AS avg_prev_hour_load
  FROM Samples AS S1, Samples AS S2
 WHERE S2.sample_time
       BETWEEN (S1.sample_time - INTERVAL 1 HOUR)
           AND S1.sample_time
 GROUP BY S1.sample_time;
```

Is the extra column or the query approach better? The query is technically better because the UPDATE approach will denormalize the database. However, if the historical data being recorded is not going to change and computing the moving average is expensive, you might consider using the column approach.

Answer #4

We can also use the new SQL-99 OLAP functions. Create the table with time slots for all the measurements that you are going to make:

```
SELECT sample_time,
       AVG(load)
```

```
       OVER (ORDER BY sample_time DESC
             ROWS 4 PRECEDING)
  FROM Samples
 WHERE EXTRACT (MINUTE FROM sample_time) = 00;
```

The SELECT computes the running total over the preceding time slots, and the WHERE clause prunes out three of the four to display the desired sample points.

Another trick is to build a table of 15-minute points for a 24-hour period. You can then construct a VIEW that will update itself every day and save you from having a huge table.

```
CREATE VIEW DailyTimeSlots (slot_timestamp)
AS
SELECT CURRENT_DATE + CAST (tick AS MINUTES)
  FROM ClockTicks;
```

PUZZLE
38 JOURNAL UPDATING

This is a simple accounting puzzle. You are given a table that represents an accounting journal with transaction dates, transaction amounts, and the accounts to which they are applied. You are to find the number of days between each transaction and post that number of days on the first of the transactions, effectively giving you how many days until the next transaction against that account.

Assume that the table is very simple:

```
CREATE TABLE Journal
(acct_nbr INTEGER NOT NULL,
 trx_date DATE NOT NULL,
 trx_amt DECIMAL (10, 2) NOT NULL,
 duration INTEGER NOT NULL);
```

Answer #1

The first answer is to use a subquery expression to do the calculation and to determine when the most recent transaction occurred relative to the current date. With a little thought, that gives us this code:

```
UPDATE Journal
   SET duration =
      (SELECT CAST ((Journal.trx_date - J1.trx_date)
                       DAYS AS INTEGER)
         FROM Journal AS J1
        WHERE J1.acct_nbr = Journal.acct_nbr
          AND J1.trx_date =
                   (SELECT MIN(trx_date)
                      FROM Journal AS J2
                     WHERE J2.acct_nbr = Journal.acct_nbr
                       AND J2.trx_date > Journal.trx_date))
 WHERE EXISTS (SELECT *
                 FROM Journal AS J3
                WHERE J3.acct_nbr = Journal.acct_nbr
                  AND J3.trx_date > Journal.trx_date);
```

Since we did not say what happens to the latest transaction for each account, the WHERE clause will keep the UPDATE from touching those rows.

Answer #2

Look at this a bit closer. The J1 table contributes nothing and can be removed without affecting the results if we do a little tricky programming to produce the following:

```
UPDATE Journal
   SET duration
          = CAST ((Journal.trx_date -
                   (SELECT MIN(trx_date)
                      FROM Journal AS J1
                     WHERE J1.acct_nbr = Journal.acct_nbr
                       AND J1.trx_date > Journal.trx_date))
                  ) DAYS AS INTEGER)
 WHERE EXISTS (SELECT *
                 FROM Journal J2
                WHERE J2.acct_nbr = Journal.acct_nbr
                  AND J2.trx_date > Journal.trx_date);
```

This depends on the use of a scalar subquery expression inside a function call. By removing the unnecessary subquery, you reduce the I/O count by more than 50% in Sybase version 11! This is really not surprising because nested correlations increase the work exponentially, not linearly. Now we have two correlated queries but no nested ones.

The bad news is that as a programmer, you have to code the identical logic in two different places in the query. This is awkward and prone to errors, especially for future changes. The first time out, you will do a cut and paste in a text editor, but people tend to forget about that again when they are maintaining code.

Answer #3

One way around this could be to not use the WHERE clause at all. A COALESCE() function with your expression would leave things unchanged where there was no matchup:

```
UPDATE Journal
   SET duration
    = COALESCE (CAST ((Journal.trx_date -
                       (SELECT MIN(trx_date)
                          FROM Journal AS J1
                        WHERE J1.acct_nbr = Journal.acct_nbr
                            AND J1.trx_date >
                                Journal.trx_date))
                     ) DAYS AS INTEGER),
                Journal.duration);
```

This statement will result in a table scan of the Journal table. This may or may not work better than the second solution, depending on how your database engine releases pages that have been updated.

Answer #4

The best answer is to not do this at all. You can construct a `VIEW` with the new OLAP functions to get the preceding:

```
SELECT acct_nbr, trx_date,
      (trx_date -
       MAX(trx_date)
        OVER(ORDER BY trx_date DESC
             RANGE BETWEEN 1 PRECEDING
                   AND 1 PRECEDING)) DAY
       AS duration
  FROM Journal;
```

Since each product's temporal functions are different, you will probably have to change the code a bit.

PUZZLE
39 INSURANCE LOSSES

This puzzle came in my e-mail from Mike Gora. I changed the original problem a bit, but the idea still holds. You are given a table with the results of an insurance salesperson's appraisal of the possible losses a customer might suffer. To make the code easier, let's alphabetically name the dangers *a* through *o*. If a danger is not present for this customer, then we show that with a NULL. If a danger is present, then we give it a numeric rating. For example, a fireworks factory on a mountaintop has no danger of a flood, but the "explosion" factor is very high. Typically, only five or six of these attributes will have any values. The table looks like this:

```
CREATE TABLE Losses
(cust_nbr INTEGER NOT NULL PRIMARY KEY,
 a INTEGER, b INTEGER, c INTEGER, d INTEGER, e INTEGER,
 f INTEGER, g INTEGER, h INTEGER, i INTEGER, j INTEGER,
 k INTEGER, l INTEGER, m INTEGER, n INTEGER, o INTEGER);
```

Let's put one customer into the table so we will have someone to talk about:

```
INSERT INTO Losses
VALUES (99, 5, 10, 15, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL);
```

We have a second table that we use to determine the correct policy to sell to the customer based on his or her possible losses. That table looks like this:

```
CREATE TABLE Policy_Criteria
(criteria_id INTEGER NOT NULL,
 criteria CHAR(1) NOT NULL,
 crit_val INTEGER  NOT NULL,
 PRIMARY KEY (criteria_id, criteria, crit_val));

INSERT INTO Policy_Criteria VALUES (1, 'A', 5);
INSERT INTO Policy_Criteria VALUES (1, 'A', 9);
```

```
INSERT INTO Policy_Criteria VALUES (1, 'A', 14);
INSERT INTO Policy_Criteria VALUES (1, 'B', 4);
INSERT INTO Policy_Criteria VALUES (1, 'B', 10);
INSERT INTO Policy_Criteria VALUES (1, 'B', 20);
INSERT INTO Policy_Criteria VALUES (2, 'B', 10);
INSERT INTO Policy_Criteria VALUES (2, 'B', 19);
INSERT INTO Policy_Criteria VALUES (3, 'A', 5);
INSERT INTO Policy_Criteria VALUES (3, 'B', 10);
INSERT INTO Policy_Criteria VALUES (3, 'B', 30);
INSERT INTO Policy_Criteria VALUES (3, 'C', 3);
INSERT INTO Policy_Criteria VALUES (3, 'C', 15);
INSERT INTO Policy_Criteria VALUES (4, 'A', 5);
INSERT INTO Policy_Criteria VALUES (4, 'B', 21);
INSERT INTO Policy_Criteria VALUES (4, 'B', 22);
```

In English, this means that:

Policy 1 has criteria A = (5, 9, 14), B = (4, 10, 20)
Policy 2 has criteria B = (10, 19)
Policy 3 has criteria A = 5, B = (10, 30), C = (3, 15)
Policy 4 has criteria A = 5, B = (21, 22)

The Losses data for customer 99 has A = 5, B = 10, C = 15.

Therefore, the customer 99 could be offered policies 1, 2, and 3, but not 4. Policy 3 should be ranked the highest, because it matches the most qualifications and returned as the answer. Policy 1 should be second highest, and Policy 2 should be last, but let's not worry about presenting alternatives yet.

Answer #1

The trick in this problem is that the losses are presented as attributes in the Losses table and as values in the Policy Criteria table. This messes up the data model and means that you have to convert one table to match the other. I will pick the Losses table and flatten it out as shown below. This might be done with a `VIEW`, but I am going to show it as a working table:

```
CREATE TABLE LossDoneRight
(cust_nbr INTEGER NOT NULL,
 criteria CHAR(1) NOT NULL,
```

```
 crit_val INTEGER NOT NULL)
```

Here is how you transform values to and from attributes:

```
INSERT INTO LossDoneRight (cust_nbr, criteria, crit_val)
SELECT cust_nbr, 'A', a FROM Losses WHERE a IS NOT NULL
   UNION ALL
   SELECT cust_nbr, 'B', b FROM Losses WHERE b IS NOT NULL
   UNION
   SELECT cust_nbr, 'C', c FROM Losses WHERE c IS NOT NULL
   UNION
   SELECT cust_nbr, 'D', d FROM Losses WHERE d IS NOT NULL
   UNION
   SELECT cust_nbr, 'E', e FROM Losses WHERE e IS NOT NULL
   UNION
   SELECT cust_nbr, 'F', f FROM Losses WHERE f IS NOT NULL
   UNION
   SELECT cust_nbr, 'G', g FROM Losses WHERE g IS NOT NULL
   UNION
   SELECT cust_nbr, 'H', h FROM Losses WHERE h IS NOT NULL
   UNION SELECT cust_nbr, 'I', i FROM Losses
           WHERE i IS NOT NULL
   UNION
   SELECT cust_nbr, 'J', j FROM Losses WHERE j IS NOT NULL
   UNION
   SELECT cust_nbr, 'K', k FROM Losses WHERE k IS NOT NULL
   UNION
   SELECT cust_nbr, 'L', l FROM Losses WHERE l IS NOT NULL
   UNION
   SELECT cust_nbr, 'M', m FROM Losses WHERE m IS NOT NULL
   UNION
   SELECT cust_nbr, 'N', n FROM Losses WHERE n IS NOT NULL
   UNION
   SELECT cust_nbr, 'O', o FROM Losses WHERE o IS NOT NULL;
```

Now we have a relational division problem:

```
SELECT L1.cust_nbr, ' could use policy ', C1.criteria_id,
       COUNT(*) AS score
  FROM LossDoneRight AS L1, Policy_Criteria AS C1
 WHERE L1.criteria = C1.criteria
```

```
   AND L1.crit_val = C1.crit_val
 GROUP BY L1.cust_nbr, C1.criteria_id
HAVING COUNT(*) = (SELECT COUNT(*)
                     FROM LossDoneRight AS L2
                    WHERE L1.cust_nbr = L2.cust_nbr);
```

In English, you join the losses and criteria together. If the loss was able to match all the criteria (i.e., has the same count) in the Policy Criteria description, we keep it. It is a one-to-one mapping of the two tables, but one of them can have leftovers and the other cannot.

Answer #2

Mr. Gora then wrote that we were getting closer but were not there yet. This gives us the perfect matches, but life is not always that kind. Instead we want to rank how well the Loss and Policy criteria match, using the rules that:

1. The policy must have a subset of the criteria given in the loss—no extra criteria.
2. The policy gets a point for each criteria value that matches the loss value.

So under these rules, policy #3 scores a perfect 3 points, policy #1 gets 2 points, and policy #2 gets 1 point. However, policy #4 did not really match because it included criteria B but did not match the required value. This is not a problem. You just have to extend the `HAVING` clause a bit:

```
SELECT L1.loss_nbr, 'matches to ', C1.criteria_id,
       ' with a score of ', COUNT(*) AS score
  FROM LossDoneRight AS L1, Policy_Criteria AS C1
 WHERE L1.criteria = C1.criteria
   AND L1.crit_val = C1.crit_val
 GROUP BY L1.loss_nbr, C1.criteria_id
HAVING COUNT(*) <= (SELECT COUNT(*)
                      FROM LossDoneRight AS L2
                     WHERE L1.loss_nbr = L2.loss_nbr)
   AND COUNT(*) = (SELECT COUNT(DISTINCT C2.criteria)
                     FROM Policy_Criteria AS C2
```

```
                    WHERE C1.criteria_id = C2.criteria_id)
ORDER BY L1.loss_nbr, score;
```

The first test against `COUNT(*)` says that you have a match on some or all of the policy criteria in this loss on both criteria and values. The second test against `COUNT(*)` says that this matching subset was the same as the criteria in the policy—thus, policy #4 gets kicked for having criteria B, but not having a criteria value of 10.

I do not know what the execution speed will be, but it looks fairly tight to me. You might want to have indexes on `loss_nbr` and `criteria_id`, since they are used for grouping and the scalar subquery expressions.

PUZZLE
40 PERMUTATIONS

SQL is pretty good about letting you do CROSS JOINs to get all possible pairs (x, y) from two sets of elements with a simple query, for example:

```
SELECT x, y
 FROM BigX CROSS JOIN BigY;
```

But sometimes you would like to do this sort of thing horizontally instead of vertically. A permutation is an ordered arrangement of elements of a set. For example, if I have the set {1, 2, 3}, the permutations of those elements are (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1). The rule is that for (n) elements, you have a factorial number (n!) of permutations. What I would like is a query that returns one permutation per row from a set of the first seven integers (that will give us 5,040 rows). Try to make the answer easy to generalize for more numbers.

```
CREATE TABLE Elements
(i INTEGER NOT NULL PRIMARY KEY);
INSERT INTO Elements
VALUES (1), (2), (3), (4), (5), (6), (7);
```

Answer #1

The obvious and horrible answer is:

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i, E7.i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6,
       Elements AS E7
 WHERE E1.i NOT IN (E2.i, E3.i, E4.i, E5.i, E6.i, E7.i)
   AND E2.i NOT IN (E1.i, E3.i, E4.i, E5.i, E6.i, E7.i)
   AND E3.i NOT IN (E1.i, E2.i, E4.i, E5.i, E6.i, E7.i)
   AND E4.i NOT IN (E1.i, E2.i, E3.i, E5.i, E6.i, E7.i)
   AND E5.i NOT IN (E1.i, E2.i, E3.i, E4.i, E6.i, E7.i)
   AND E6.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E7.i)
   AND E7.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E6.i);
```

This monster predicate will guarantee that all column values in a row are unique. Execution time, however, is pretty bad.

Answer #2

An improvement on this query can be made by adding one more predicate to the WHERE clause:

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i, E7.i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6,
       Elements AS E7
 WHERE (E1.i + E2.i + E3.i + E4.i + E5.i + E6.i + E7.i)
       = 28
   AND E1.i NOT IN (E2.i, E3.i, E4.i, E5.i, E6.i, E7.i)
   AND E2.i NOT IN (E1.i, E3.i, E4.i, E5.i, E6.i, E7.i)
   AND E3.i NOT IN (E1.i, E2.i, E4.i, E5.i, E6.i, E7.i)
   AND E4.i NOT IN (E1.i, E2.i, E3.i, E5.i, E6.i, E7.i)
   AND E5.i NOT IN (E1.i, E2.i, E3.i, E4.i, E6.i, E7.i)
   AND E6.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E7.i)
   AND E7.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i, E6.i);
```

This improves things because most optimizers will see a predicate of the form <expression> = <constant> and will execute it before the AND-ed chain of IN() predicates. While not all rows that total 28 are a permutation, all permutations will total to 28 for this set of integers. When you have a factorial, you look for all the improvements you can get!

Answer #3

But let's carry the totals trick one step further. First, redefine the Elements table to have a weight for each element in the set:

```
CREATE TABLE Elements
(i INTEGER NOT NULL
 wgt INTEGER NOT NULL);
 INSERT INTO Elements
  VALUES (1, 1), (2, 2), (3, 4), (4, 8),
         (5, 16), (6, 32), (7, 64);
```

The weights are powers of 2, and we are about to write a bit vector in SQL with them. Now, the `WHERE` clause becomes:

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i, E7.i
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6,
       Elements AS E7
WHERE (E1.wgt + E2.wgt + E3.wgt + E4.wgt
       + E5.wgt + E6.wgt + E7.wgt) = 127;
```

This does the whole filtering job for you and the `IN()` predicates are all unnecessary. This answer also has another beneficial effect: the elements can now be of any datatype and are not limited just to integers.

Answer #4

Ian Young played with these solutions in MS SQL Server (both version 7.0 and 2000) and came up with the following conclusions for that product.

Well, the answer is not what you might expect. For Answer #1, the optimizer takes apart each of the predicates and applies the relevant parts on a join-by-join basis. So for the i-th join, the result has 7!/(7-i)! items.

The addition of the global constraint in Answer #2 leaves the overall approach the same, but makes it run a little (10% to 15%) slower.

Using the bit vector in Answer #3 means it cannot localize any constraints and is only able to filter the last cross join, taking (n^(n-1) * n) items down to n!. Result: the naive answer is about 5 to 10 times faster for seven items, and the bit vector approach is essentially unusable for nine items.

There are some improvements to the naive method, though.

Firstly, we are testing each of the constraints in two places, so we can reduce this to the upper or lower triangle—though this doesn't make much useful difference on MS SQL Server. More important, we are using seven cross joins to generate seven items when the last is uniquely constrained by the others. Better to drop the last join and calculate the value in the same way as the global constraint in Answer #2.

```
SELECT E1.i, E2.i, E3.i, E4.i, E5.i, E6.i,
       (28 - E1.i - E2.i - E3.i - E4.i - E5.i - E6.i)
       AS i
```

```
  FROM Elements AS E1, Elements AS E2, Elements AS E3,
       Elements AS E4, Elements AS E5, Elements AS E6
 WHERE E2.i NOT IN (E1.i)
   AND E3.i NOT IN (E1.i, E2.i)
   AND E4.i NOT IN (E1.i, E2.i, E3.i)
   AND E5.i NOT IN (E1.i, E2.i, E3.i, E4.i)
   AND E6.i NOT IN (E1.i, E2.i, E3.i, E4.i, E5.i)
```

Another possibility is to try to make it perform (n!) iterations, the minimum needed in theory, in the joins. This is possible if we view a string as a list of characters. It results in a query that is essentially an unrolled recursive function. Here, a string is accumulating the chosen values, and string c contains the remaining choices:

```
SELECT a || c
  FROM (SELECT a || SUBSTRING(c FROM i FOR 1),
                    STUFF(c, i, 1, '')
          FROM Elements,
               (SELECT a || SUBSTRING(c FROM i FOR 1),
                           STUFF(c, i, 1, '')
                  FROM Elements,
                     (SELECT a || SUBSTRING(c FROM i FOR 1),
                                  STUFF(c, i, 1, '')
                        FROM Elements,
                           (SELECT a || SUBSTRING(c FROM
                                                  i FOR 1),
                                        STUFF(c, i, 1, '')
                              FROM Elements,
                                (SELECT a || SUBSTRING(c
                                          FROM i FOR 1),
                                                STUFF(c,
                                                i, 1, '')
                                      FROM Elements,
                                           (SELECT
SUBSTRING('1234567', i, 1),

STUFF('1234567',
i, 1, '')
                                 FROM Elements
WHERE i <= 7) AS T1 (a,c)
                                                WHERE i <= 6) AS T2
(a,c)
```

```
                                        WHERE i <= 5) AS T3 (a,c)
                                  WHERE i <= 4) AS T4 (a,c)
                            WHERE i <= 3) AS T5 (a,c)
                      WHERE i <= 2) AS T6 (a,c);
```

The STUFF function is proprietary; it takes a target string, pulls it apart, and inserts another string at a given location. However, it is common in most SQLs, and it is easy to write as a user-defined function in products that do not have it.

There's as much work in the string operations as the extra loops of joins. And of course, it is additional work to separate and convert the characters of the string if this is what we want when we use the results.

Answer #5

I did not stop here. Looking at the query plan for this last approach, it appears to decide to unpick the accumulation, giving something operationally equivalent to this monstrosity:

```
SELECT SUBSTRING('1234567', a, 1) ||
       SUBSTRING(STUFF('1234567', a, 1, ''), b, 1) ||
       SUBSTRING(STUFF(STUFF('1234567', a, 1, ''), b, 1,
'', c,
1) ||
       SUBSTRING(STUFF(STUFF(STUFF('1234567',
         a, 1, ''), b, 1, ''), c, 1, ''), d, 1) ||
       SUBSTRING(STUFF(STUFF(STUFF(STUFF('1234567',
        a, 1, ''), b, 1, ''), c, 1, ''), d, 1, ''), e, 1) ||
       SUBSTRING(STUFF(STUFF(STUFF(STUFF(STUFF('1234567',
         a, 1, ''), b, 1, ''), c, 1, ''), d, 1, ''), e, 1,
'',
f, 1) ||
       STUFF(STUFF(STUFF(STUFF(STUFF(STUFF('1234567',

         a, 1, ''), b, 1, ''), c, 1, ''), d, 1, ''), e, 1,
'',
f, 1, '')
FROM (SELECT i
        FROM Elements
       WHERE i <= 7) AS T1 (a),
      (SELECT i
         FROM Elements
```

```
  WHERE i <= 6) AS T2 (b),
(SELECT i
   FROM Elements
  WHERE i <= 5) AS T3 (c),
(SELECT i
   FROM Elements
  WHERE i <= 4) AS T4 (d),
(SELECT i
   FROM Elements
  WHERE i <= 3) AS T5 (e),
(SELECT i
   FROM Elements
  WHERE i <= 2) AS T6 (f);
```

If you are interested in this kind of problem, you can get a survey on algorithms by Robert Sedgewick at http://www.princeton.edu/~rblee/ELE572Papers/p137-sedgewick.pdf. The algorithms are procedural, with loops or recursion, but you might be able to translate them into recursive CTEs.

PUZZLE
41 BUDGETING

Mark Frontera at LanSoft, Inc., in Miami, Florida, posted this problem in September 1995. He has budgeting information that consists of the following three tables: the items to be paid for, the estimated amounts to be spent on them, and the actual amounts spent on them. I am going to skip the DDL and post the data, since they are simple.

Notice that some items are covered by more than one check, and sometimes one check covers several items.

```
Items
item_nbr  item_descr
=====================
  10     'Item 10'
  20     'Item 20'
  30     'Item 30'
  40     'Item 40'
  50     'item 50'

Actuals
item_nbr  actual_amt   check_nbr
================================
  10       300.00         '1111'
  20       325.00         '2222'
  20       100.00         '3333'
  30       525.00         '1111'

Estimates
item_nbr  estimated_amt
=======================
   10     300.00
   10      50.00
   20     325.00
   20     110.00
   40      25.00
```

I would like the following output from a single query:

```
Results
item_nbr    item_descr   actual_tot estimate_tot  check_nbr
=================================================
  10        'item 10'     300.00   350.00    '1111'
  20        'item 20'     425.00   435.00    'Mixed'
  30        'item 30'     525.00     NULL    '1111'
  40        'item 40'       NULL    25.00     NULL
```

Item 50 from the Items table is not to be shown, because there was no record for it in either the Actuals or the Estimates table. The column `actual_tot` is the total of actual amounts for that item; the column `estimate_tot` is the total of estimated amounts for that item.

Answer #1

I think that this schema needs some work, but you can do this with scalar subqueries and some tricky code.

```
SELECT I1.item_nbr, I1.item_descr,
      (SELECT SUM (A1.actual_amt)
         FROM Actuals AS A1
        WHERE I1.item_nbr = A1.item_nbr) AS tot_act,
      (SELECT SUM (E1.estimated_amt)
         FROM Estimates AS E1
        WHERE I1.item_nbr = E1.item_nbr) AS estimate_tot,
      (SELECT CASE WHEN COUNT(*) = 1
                   THEN MAX(check_nbr)
                   ELSE 'Mixed' END
         FROM Actuals) AS A2
       WHERE I1.item_nbr = A2.item_nbr
       GROUP BY item_nbr) AS check_nbr
  FROM Items AS I1
 WHERE actual_tot IS NOT NULL
    OR estimate_tot IS NOT NULL;
```

The trick is in the scalar subqueries. The first two calculate the total actual amounts and the total estimated amounts as if they were part of a `GROUP BY` and `LEFT OUTER JOIN`.

The final subquery is trickier. The query finds all of the Actuals that are associated with the item under consideration in the result table and makes a group from them. If the group is empty (no checks issued),

then the subquery returns a single NULL, and we display that NULL. If the group has one check in it, then the CASE expression will return that single check number. The MAX() function is a safety check to guarantee that you have a scalar result from the subquery; you might not need it in all SQL-92 implementations. If there is more than one check actually issued on the item, then the COUNT(*) is greater than 1, and you get the string 'Mixed' instead of a string that represents the unique check number.

Answer #2

You can replace the subqueries with LEFT OUTER JOINs:

```
SELECT I1.item_nbr, I1.item_descr,
       SUM(A1.actual_amt) AS tot_act,
       SUM(E1.estimated_amt) AS estimate_tot,
       (SELECT CASE WHEN COUNT(check_nbr) = 1
                    THEN MAX(check_nbr)
                    ELSE 'Mixed' END
          FROM Actuals) AS check_nbr
  FROM (items AS I1
        LEFT OUTER JOIN
        Actuals AS A1
        ON I1.acct_nbr = A1.acct_nbr)
           LEFT OUTER JOIN
           Estimates AS E1
           ON I1.acct_nbr = E1.acct_nbr
  GROUP BY I1.item_nbr, I1.item_descr;
```

PUZZLE
42 COUNTING FISH

Let's go fishing! A fish and game warden is trying to find the average of something that is not there. This is not quite as strange as it first sounds, nor quite as simple. The warden is collecting sample data on fish in the following table:

```
CREATE TABLE Samples
(sample_id INTEGER NOT NULL,
 fish_name CHAR(20) NOT NULL,
 found_tally INTEGER NOT NULL,
 PRIMARY KEY (sample_id, fish_name));

INSERT INTO Samples
VALUES (1, 'minnow', 18),
       (1, 'pike', 7),
       (2, 'pike', 4),
       (2, 'carp', 3),
       (3, 'carp', 9),
         ... ;

CREATE TABLE SampleGroups
(group_id INTEGER NOT NULL,
 group_descr CHAR(20) NOT NULL,
 sample_id INTEGER NOT NULL
    REFERENCES Samples(sample_id),
 PRIMARY KEY (group_id, sample_id));

INSERT INTO SampleGroups
VALUES (1, 'muddy water', 1),
       (1, 'muddy water', 2),
       (2, 'fresh water' 1),
       (2, 'fresh water', 3),
       (2, 'fresh water', 4),
         ...;
```

Notice that a sample can be grouped many ways; sample 1 is fresh muddy-water fish.

The warden needs to get the average number of each species of fish in the sample groups. For example, group number one (`'muddy water'`) has

samples 1 and 2; you could use the parameters (`:my_fish_name = 'minnow'`) and (`:my_group = 1`) to find the average number of minnows in sample group 1, as follows:

```
SELECT fish_name, AVG(found_tally)
  FROM Samples
 WHERE sample_id IN (SELECT sample_id
                       FROM SampleGroups
                      WHERE group_id = :my_group)
   AND fish_name = :my_fish_name
 GROUP BY fish_name;
```

This query will give you an average of 18 minnows, which is wrong. There were no minnows for `sample_id = 2` within group 1, so the average is ((18 + 0)/2) = 9. The other approach is to do several steps to get the correct answer: first use a `SELECT` statement to get the number of samples involved, then use another `SELECT` to get the sum, and finally manually calculate the average. Is there a way to do it in one `SELECT` statement?

Answer #1

The obvious answer is to enter a count of 0 for each `fish_name` under each `sample_id` instead of letting it be missing. This approach will let you use the original query. You can create the missing rows with the following statement:

```
INSERT INTO Samples
SELECT M1.sample_id, M2.fish_name, 0
  FROM Samples AS M1, Samples AS M2
 WHERE NOT EXISTS
       (SELECT *
          FROM Samples AS M3
         WHERE M1.sample_id = M3.sample_id
           AND M2.fish_name = M3.fish_name);
```

Answer #2

Unfortunately, it turns out that there are more than 100,000 different species of fish and tens of thousands of samples. This trick would fill up

more disk space than the warden has. You need to use SQL tricks to get it into one statement:

```
SELECT fish_name, SUM(found_tally)/
           (SELECT COUNT(sample_id)
              FROM SampleGroups
             WHERE group_id = :my_group)
  FROM Samples
 WHERE fish_name = :my_fish_name
 GROUP BY fish_name;
```

The scalar subquery query is really using the rule that an average is the total of the values divided by the number of occurrences. But the SQL is a little tricky.

The `SUM()` in the dividend returns a `NULL` when it has an empty set. This will make the fraction (quotient) become `NULL`. The scalar subquery expression in the divisor returns `NULL` when its result is an empty set. However, the `COUNT(<expression>)` aggregate function inside the subquery will return a zero when it has an empty set as its parameter.

The only way that the `COUNT(<expression>)` aggregate function will return a `NULL` is from a table that has only `NULL`s in it. But we have declared all the tables to be without `NULL`s, so we are safe.

Answer #3

Anilbabu Jaiswal of Kansas City submitted a slightly different Oracle version, which translates into SQL-92 as:

```
SELECT fish_name, AVG(COALESCE(found_tally, 0))
  FROM Samples AS SA
       LEFT OUTER JOIN
       SampleGroups AS SG
       ON SA.sample_id = SG.sample_id
         AND SA.fish_name = :my_fish_name
         AND group_id = :my_group
 GROUP BY fish_name;
```

The `COALESCE` function will inspect its parameter list and return the first non-`NULL` value, so this converts the `AVG()` parameter from `NULL` to zero. Most people seem to have trouble with the idea that an aggregate can handle an expression, not just a single column, as a parameter. The

other good trick in this solution is doing a `LEFT OUTER JOIN` on two columns instead of just one. This is very handy because the primary key of a table is not always just one column.

PUZZLE
43 GRADUATION

Richard Romley created this problem based on the logic of a more complicated problem. It is a good example of how we need to learn to analyze problems differently with ANSI/ISO SQL-92 stuff than we did before. There are some really neat solutions that didn't exist before if we learn to think in terms of the new features; the same thing applies to SQL-99 features. This solution takes advantage of derived tables, `CASE` statements, and outer joins based on other than equality—all in one query.

In this problem, `student_names` represents students who take courses for which they receive credits. Each course belongs to a credit category. The Categories table lists each `credit_cat` and the minimum necessary credits required in that credit_cat to graduate. The "`CreditsEarned`" table has a row for each course completed showing the `student_name`, `credit_cat`, and number of credits earned. (This would more logically contain `student_name`, course, and credits, and the `credit_cat` would be looked up in the Courses table—but for this problem I simplified the definition slightly.) The first problem is to generate a list with all `student_names` who are eligible to graduate—that is, those students who have completed at least the minimum required credits in all categories. Then, generate a list with all `student_names` who are not eligible to graduate. But best yet is to combine these two and generate a single list of all `student_names` showing in the appropriate column whether or not each student is eligible to graduate.

```
EligibleReport
student_name   grad nograd
=====================
Bob              X
Joe         X
John             X
Mary        X

CREATE TABLE Categories
(credit_cat CHAR(1) NOT NULL,
 rqd_credits INTEGER NOT NULL);

CREATE TABLE CreditsEarned  -- no primary key
(student_name CHAR(10) NOT NULL,
```

```
 credit_cat CHAR(1) NOT NULL,
 credits INTEGER NOT NULL);

INSERT INTO Categories
VALUES (('A', 10),
        ('B', 3),
        ('C', 5));

INSERT INTO CreditsEarned
VALUES ('Joe', 'A', 3), ('Joe', 'A', 2), ('Joe', 'A', 3),
       ('Joe', 'A', 3), ('Joe', 'B', 3), ('Joe', 'C', 3),
       ('Joe', 'C', 2), ('Joe', 'C', 3),
       ('Bob', 'A', 2), ('Bob', 'C', 2), ('Bob', 'A', 12),
       ('Bob', 'C', 4),
       ('John', 'A', 1), ('John', 'B', 100),
      ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'A', 1),
       ('Mary', 'A', 1), ('Mary', 'A', 1),('Mary', 'A', 1),
      ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'A', 1),
      ('Mary', 'A', 1), ('Mary', 'A', 1), ('Mary', 'B', 1),
      ('Mary', 'B', 1), ('Mary', 'B', 1), ('Mary', 'B', 1),
      ('Mary', 'B', 1), ('Mary', 'B', 1), ('Mary', 'B', 1),
      ('Mary', 'C', 1), ('Mary', 'C', 1), ('Mary', 'C', 1),
      ('Mary', 'C', 1), ('Mary', 'C', 1), ('Mary', 'C', 1),
       ('Mary', 'C', 1), ('Mary', 'C', 1);
```

This is the best solution I can come up with:

```
SELECT X.student_name,
       CASE WHEN COUNT(C1.credit_cat)
                 >= (SELECT COUNT(*) FROM Categories)
            THEN 'X'
            ELSE ' ' END AS grad,
       CASE WHEN COUNT(C1.credit_cat)
                < (SELECT COUNT(*) FROM Categories)
            THEN 'X'
            ELSE ' ' END AS nograd
  FROM (SELECT student_name, credit_cat, SUM(credits) AS
cat_credits
          FROM CreditsEarned
         GROUP BY student_name, credit_cat) AS X
      LEFT OUTER JOIN
```

```
       Categories AS C1
      ON X.credit_cat = C1.credit_cat
      AND X.cat_credits >= C1.rqd_credits
 GROUP BY X.student_name

Results
student_name  grad nograd
--------------------------
Bob                X
Joe           X
John               X
Mary          X
```

The derived table `X` contains a row for each student, credit category, and total credits for that (`student_name`, `credit_cat`) combination. The key to this solution is in the next step—the `LEFT OUTER JOIN` to `credit_cat` on `credit_cat` and (`credits >= required credits`). By then grouping on `student_name`, `COUNT(C1.credit_cat)` will tell me, for the categories in which the student took any courses, in how many he or she has at least the minimum required credits for graduation. By comparing this to the total number of categories, I can determine if the student is eligible to graduate and put the "`X`" in the appropriate column.

This automatically handles the situation where a student may have taken no courses in a particular credit category. `COUNT(C1.credit_cat)` will only count categories in which at least the minimum number of credits have been earned.

PUZZLE
44 PAIRS OF STYLES

Abbott de Rham posted this problem on the ACCESS Forum in September 1996. He gets data from sales slips that show pairs of items in the order they are collected at the point of sale. The table looks like this:

```
CREATE TABLE SalesSlips
(item_a INTEGER NOT NULL,
 item_b INTEGER NOT NULL,
 PRIMARY KEY(item_a, item_b),
 pair_tally INTEGER NOT NULL);
```

The table is arranged by the style that shows up first on an order as `item_a`, and `item_b` is always the item that came after `item_a` on the order. The table will also include pairs where the paired values are the same style.

```
SalesSlips
item_a    item_b   pair_tally
================================
12345      12345        12
12345      67890         9
67890      12345         5
```

For some of his reports, he would like to sum all pairs and their reciprocals together with a result set showing only one entry per pair:

```
Pairs
item_a    item_b   pair_tally
================================
12345      12345        12
12345      67890        14
```

He had no trouble getting rows with reciprocals added together with a self-join, but he could not get rid of the duplicate rows:

```
SELECT S0.item_a, S0.item_b, SUM(S0.pair_tally +
S1.pair_tally) AS pair_tally
  FROM SalesSlips AS S0, SalesSlips AS S1
```

```
 WHERE S0.item_b = S1.item_a
   AND S0.item_a = S1.item_b
 GROUP BY S0.item_a, S0.item_b, S1.item_a, S1.item_b;
```

This returned the false results:

```
Results
item_a    item_b   pair_tally
============================
12345   12345        24
12345   67890        14
67890   12345        14
```

He had considered writing code to seek the reciprocal, add the value, and delete a record while skipping style pairs with the same style numbers. He was hoping for an SQL solution instead.

Answer #1

The existing query can be easily patched up:

```
SELECT S0.item_a, S0.item_b, SUM(S0.pair_tally +
S1.pair_tally) AS pair_tally,
  FROM SalesSlips AS S0, SalesSlips AS S1
 WHERE S0.item_a <= S0.item_b
   AND S0.item_a = S1.item_b
   AND S0.item_b = S1.item_a
 GROUP BY S0.item_a, S0.item_b, S1.item_a, S1.item_b;
```

The self-join will be expensive and you really do not need it; you can write this instead:

```
SELECT CASE WHEN item_a <= item_b
            THEN item_a
            ELSE item_b END AS s1,
       CASE WHEN item_a <= item_b
            THEN item_b
            ELSE item_a END AS s2,
       SUM (pair_tally)
  FROM SalesSlips
 GROUP BY s1, s2;
```

Frankly, this is not supposed to work because the column names `s1` and `s2` come into existence after the `GROUP BY` and therefore cannot be used by it. However, lots of products support this syntax because they improperly create the `SELECT` list first, and then fill it. The correct SQL-92 version would use a tabular subquery:

```
SELECT s1, s2, SUM(pair_tally)
  FROM (SELECT CASE WHEN item_a <= item_b
                    THEN item_a
                    ELSE item_b END,
               CASE WHEN item_a <= item_b
                    THEN item_b
                    ELSE item_a END,
               pair_tally
          FROM SalesSlips) AS Report (s1, s2, pair_tally)
 GROUP BY s1, s2;
```

Answer #2

In SQL-89, you would have to put the tabular subquery expression in a `VIEW` and then use the `VIEW` in another query. It is really the same code, but broken into separate steps and with the advantage that the `VIEW` can be reused for other reports.

```
CREATE VIEW Report (s1, s2, pair_tally)
AS SELECT CASE WHEN item_a <= item_b
               THEN item_a
               ELSE item_b END,
          CASE WHEN item_a <= item_b
               THEN item_b
               ELSE item_a END,
         pair_tally
    FROM SalesSlips;

SELECT s1, s2, SUM(pair_tally)
  FROM Report
 GROUP BY s1, s2;
```

Answer #3

But the best way is to update the database itself and make `item_a` the smallest of the two code numbers, before doing the query, so this is not an issue:

```
UPDATE SalesSlips
   SET item_a = item_b,
       item_b = item_a
 WHERE item_a > item_b;
```

You could also do this with a `TRIGGER` on insertion, but that would mean writing proprietary procedural code. The real answer is to mop the floor (these updates) and then to fix the leak with a `CHECK()` constraint:

```
CREATE TABLE SalesSlips
(item_a INTEGER NOT NULL,
 item_b INTEGER NOT NULL,
 PRIMARY KEY(item_a, item_b),
  CHECK (item_a <= item_b)
 pair_tally INTEGER NOT NULL);
```

PUZZLE
45 PEPPERONI PIZZA

A good classic accounting problem is to print an aging report of old billings. Let's use the Friends of Pepperoni, who have a charge card at our pizza joint. It would be nice to find out if you should have let club members charge pizza on their cards.

You have a table of charges that contains a member identification number (`cust_id`), a date (`bill_date`), and an amount (`pizza_amt`). None of these is a key, so there can be multiple entries for a customer, with various dates and amounts. This is an old-fashioned journal file, done as an SQL table.

What you are trying to do is get a sum of amounts paid by each member within an age range. The ranges are 0 to 30 days old, 31 to 60 days old, 61 to 90 days old, and everything over 90 days old. This is called an aging report on account receivables, and you use it to see what the Friends of Pepperoni program is doing to you.

Answer #1

You can write a query for each age range with `UNION ALL` operators, like this:

```
SELECT cust_id, '0-30 days = ' AS age, SUM (pizza_amt)
  FROM Friends Of Pepperoni
 WHERE bill_date BETWEEN CURRENT_DATE
            AND (CURRENT_DATE - INTERVAL 30 DAY)
 GROUP BY cust_id
UNION ALL
SELECT cust_id, '31-60 days = ' AS age, SUM (pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date BETWEEN (CURRENT_DATE - INTERVAL 31 DAY)
            AND (CURRENT_DATE - INTERVAL 90 DAY)
 GROUP BY cust_id
UNION ALL
SELECT cust_id, '61-90 days = ' AS age, SUM(pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date BETWEEN (CURRENT_DATE - INTERVAL 61 DAY)
            AND (CURRENT_DATE - INTERVAL 90 DAY)
 GROUP BY cust_id
UNION ALL
```

```
SELECT cust_id, '90+ days = ' AS age, SUM(pizza_amt)
  FROM FriendsOfPepperoni
 WHERE bill_date < CURRENT_DATE - INTERVAL 90 DAY) GROUP BY
cust_id
ORDER BY cust_id, age;
```

Using the second column to keep the age ranges as text makes sorting within each customer easier because the strings are in temporal order. This query works, but it takes awhile. There must be a better way to do this in SQL-92.

Answer #2

Do not use UNIONs when you can use a CASE expression instead. The UNIONs will make multiple passes over the table, and the CASE expression will make only one.

```
SELECT cust_id,
     SUM(CASE WHEN bill_date
              BETWEEN CURRENT_TIMESTAMP - INTERVAL 30 DAYS
                  AND CURRENT_TIMESTAMP
              THEN pizza_amt ELSE 0.00) AS age1,
     SUM(CASE WHEN bill_date
              BETWEEN CURRENT_TIMESTAMP - INTERVAL 60 DAYS
                AND CURRENT_TIMESTAMP - INTERVAL 31 DAYS
              THEN pizza_amt ELSE 0.00) AS age2,
     SUM(CASE WHEN bill_date
              BETWEEN CURRENT_TIMESTAMP - INTERVAL 90 DAYS
                      AND CURRENT_TIMESTAMP - INTERVAL 61
DAYS
              THEN pizza_amt ELSE 0.00) AS age3,
     SUM(CASE WHEN bill_date
                   < CURRENT_TIMESTAMP - INTERVAL 91 DAYS
              THEN pizza_amt ELSE 0.00) AS age4
 FROM FriendsofPepperoni;
```

Using the CASE expression to replace UNIONs is a handy trick.

Answer #3

You can avoid both UNIONs and CASE expressions by creating a CTE or derived table with the ranges for the report.

```
WITH ReportRanges(day_count, start_cnt, end_cnt)
AS (VALUES ('under Thirty days', 00, 30),
           ('Sixty days', 31, 60),
           ('Ninty days', 61, 90))
SELECT F1.cust_id, R1.day_count, SUM(pizza_amt)
  FROM FriendsofPepperoni AS F1
       LEFT OUTER JOIN
       ReportRanges AS R1
       ON F1.bill_date
          BETWEEN CURRENT_TIMESTAMP - start_cnt DAY
              AND CURRENT_TIMESTAMP - end_cnt DAY;
```

This is easier to maintain and extend than the `CASE` expression. It can also be faster with indexing. Remember, SQL is designed for joins and not computations.

PUZZLE
46 SALES PROMOTIONS

You have just gotten a job as the sales manager for a department store. Your database has two tables. One is a calendar of the promotional events the store has had, and the other is a list of the sales that have been made during the promotions. You need to write a query that will tell us which clerk had the highest amount of sales for each promotion, so we can pay that clerk a performance bonus.

```
CREATE TABLE Promotions
(promo_name CHAR(25) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 CHECK (start_date <= end_date));

Promotions
promo_name                   start_date      end_date
=======================================================
'Feast of St. Fred'         '1995-02-01'  '1995-02-07'
'National Pickle Pageant'   '1995-11-01'  '1995-11-07'
'Christmas Week'            '1995-12-18'  '1995-12-25'

CREATE TABLE Sales
(ticket_nbr INTEGER NOT NULL PRIMARY KEY,
 clerk_name CHAR (15) NOT NULL,
 sale_date DATE NOT NULL,
 sale_amt DECIMAL (8,2) NOT NULL);
```

Answer #1

The trick in this query is that we need to find out what each employee sold during each promo and finally pick the highest sum from those groups. The first part is a fairly easy `JOIN` and `GROUP BY` statement.

The final step of finding the largest total sales in each grouping requires a fairly tricky `HAVING` clause. Let's look at the answer first, and then explain it.

```
SELECT S1.clerk_name, P1.promo_name, SUM(S1.amount) AS
sales_tot
  FROM Sales AS S1, Promotions AS P1
```

```
 WHERE S1.saledate BETWEEN P1.start_date AND P1.end_date
 GROUP BY S1.clerk_name, P1.promo_name
HAVING SUM(amount) >=
     ALL (SELECT SUM(amount)
            FROM Sales AS S2
           WHERE S2.clerk_name <> S1.clerk_name
             AND S2.saledate
                 BETWEEN (SELECT start_date
                            FROM Promotions AS P2
                           WHERE P2.promo_name =
P1.promo_name)
                     AND (SELECT end_date
                            FROM Promotions AS P3
                           WHERE P3.promo_name =
P1.promo_name)
           GROUP BY S2.clerk_name);
```

We want the total sales for the chosen clerk and promotion to be equal or greater than the other total sales of all the other clerks during that promotion. The predicate "`S2.clerk_name <> S1.clerk_name`" excludes the other clerks from the subquery total. The subquery expressions in the `BETWEEN` predicate make sure that we are using the right dates for the promotion.

The first thought when trying to improve this query is to replace the subquery expressions in the `BETWEEN` predicate with direct outer references, like this:

```
 SELECT S1.clerk_name, P1.promo_name, SUM(S1.amount) AS
sales_tot
   FROM Sales AS S1 Promotions AS P1
  WHERE S1.saledate BETWEEN P1.start_date AND P1.end_date
  GROUP BY S1.clerk_name, P1.promo_name
HAVING SUM(amount) >=
  ALL (SELECT SUM(amount)
         FROM Sales AS S2
        WHERE S2.clerk_name <> S1.clerk_name
          AND S2.saledate          -- Error !!
              BETWEEN P1.start_date AND P1.end_date
        GROUP BY S2.clerk_name);
```

But this will not work—and if you know why, then you really know your SQL. Cover the rest of this page and try to figure it out before you read further.

Answer #2

The "GROUP BY S1.clerk_name, P1.promo_name" clause has created a grouped table whose rows contain only aggregate functions and two grouping columns. The original working table built in the FROM clause ceased to exist and was replaced by this grouped working table, so the start_date and end_date also ceased to exist at that point.

However, the subquery expressions work because they reference the outer table P1 while it is still available, since the query works from the innermost subqueries outward and not the grouped table.

If we were looking for sales performance between two known, constant dates, then the second query would work when we replaced P1.start_date and P1.end_date with those constants.

Two readers of my column sent in improved versions of this puzzle. Richard Romley and J. D. McDonald both noticed that the Promotions table has only key columns if we assume that no promotions overlap, so that using (promo_name, start_date, end_date) in the GROUP BY clause will not change the grouping. However, it will make the start_date and end_date available to the HAVING clause, thus:

```
SELECT S1.clerk_name, P1.promo_name, SUM(S1.amount) AS sales_tot
   FROM Sales AS S1 Promotions AS P1
  WHERE S1.saledate BETWEEN P1.start_date AND P1.end_date
  GROUP BY P1.promo_name, P1.start_date, P1.end_date,
S1.clerk_name
 HAVING SUM(S1.amount) >
    ALL (SELECT SUM(S2.amount)
           FROM Sales AS S2
          WHERE S2.Saledate BETWEEN P1.start_date AND P1.end_date
            AND S2.clerk_name <> S1.clerk_name
          GROUP BY S2.clerk_name);
```

Alternatively, you can reduce the number of predicates in the HAVING clause by making some simple changes in the subquery, thus:

```
  ...
 HAVING SUM(S1.amount) >=
```

```
    ALL (SELECT SUM(S2.amount)
           FROM Sales AS S2
          WHERE S2.Saledate BETWEEN P1.start_date AND
P1.end_date
          GROUP BY S2.clerk_name);
```

I am not sure if there is much difference in performance between the two, but the second is cleaner.

Answer #3

The new common table expression (CTE) makes it easier to aggregate data at multiple levels:

```
WITH ClerksTotals (clerk_name, promo_name, sales_tot)
AS
(SELECT S1.clerk_name, P1.promo_name, SUM(S1.amount)
  FROM Sales AS S1, Promotions AS P1
 WHERE S1.saledate BETWEEN P1.start_date AND P1.end_date
 GROUP BY S1.clerk_name, P1.promo_name)

SELECT C1.clerk_name, C1.promo_name
  FROM ClerksTotals AS C1
 WHERE C1.sales_tot
       = (SELECT MAX(C2.sales_tot)
            FROM ClerksTotals AS C2
           WHERE C1.promo_name = C2.promo_name);
```

This is fairly tight code and should be easy to maintain.

PUZZLE
47 BLOCKS OF SEATS

The original version of this puzzle came from Bob Stearns at the University of Georgia and dealt with allocating pages on an Internet server. I will reword it as a block of seat reservations in the front row of a theater. The reservations consist of the reserver's name and the `start_seat` and `finish_seat` seat numbers of his block.

The rule of reservation is that no two blocks can overlap. The table for the reservations looks like this:

```
CREATE TABLE Reservations
(reserver CHAR(10) NOT NULL PRIMARY KEY,
 start_seat INTEGER NOT NULL,
 finish_seat INTEGER NOT NULL);

Reservations
reserver start_seat  finish_seat
================================
'Eenie'      1       4
'Meanie'     6       7
'Mynie'     10      15
'Melvin'    16      18
```

What you want to do is put a constraint on the table to ensure that no reservations violating the overlap rule are ever inserted. This is harder than it looks unless you do things in steps.

Answer #1

The first solution might be to add a `CHECK()` clause. You will probably draw some pictures to see how many ways things can overlap, and you might come up with this:

```
CREATE TABLE Reservations
(reserver CHAR(10) NOT NULL PRIMARY KEY,
 start_seat INTEGER NOT NULL,
 finish_seat INTEGER NOT NULL,
 CHECK (start_seat <= finish_seat),

 CONSTRAINT No_Overlaps
```

```
  CHECK (NOT EXISTS
          (SELECT R1.reserver
             FROM Reservations AS R1
            WHERE Reservations.start_seat BETWEEN
R1.start_seat AND R1.finish_seat
              OR Reservations.finish_seat BETWEEN
R1.start_seat AND R1.finish_seat));
```

This is a neat trick that will also handle duplicate start and finish seat pairs with different reservers, as well as overlaps.

The two problems are that intermediate SQL-92 does not allow subqueries in a `CHECK()` clause, but full SQL-92 does allow them. So this trick is probably not going to work on your current SQL implementation. If you get around that problem, you might find that you have trouble inserting an initial row into the table.

The `PRIMARY KEY` and `NOT NULL` constraints are no problem. However, when the engine does the `CHECK()` constraint, it will make a copy of the empty Reservations table in the subquery under the name `R1`.

Now things get confusing. The `R1.start_seat` and `R1.finish_seat` values cannot be `NULL`s, according to the `CREATE TABLE` statement, but `D1` is empty, so they have to be `NULL`s in the `BETWEEN` predicates.

There is a very good chance that this self-referencing is going to confuse the constraint checker, and you will never be able to insert a first row into this table. The safest bet is to declare the table, insert a row or two, and add the `No_Overlaps` constraint afterward. You can also defer a constraint, and then turn it back on when you leave the session.

PUZZLE
48 UNGROUPING

Sissy Kubu sent me a strange question on CompuServe. She has a table like this:

```
CREATE TABLE Inventory
(goods CHAR(10) NOT NULL PRIMARY KEY,
 pieces INTEGER NOT NULL CHECK (pieces >= 0));
```

She wants to deconsolidate the table; that is, get a `VIEW` or nontable with one row for each piece. For example, given a row with (`'CD-ROM'`, `3`) in the original table, she would like to get three rows with (`'CD-ROM'`, `1`) in them. Before you ask me, I have no idea why she wants to do this; consider it a training exercise.

Since SQL has no "`UN-COUNT(*) ... DE-GROUP BY..`" operators, you will have to use a cursor or the vendor's 4GL to do this. Frankly, I would do this in a report program instead of an SQL query, since the results will not be a table with a key. But let's look for weird answers since this is an exercise.

Answer #1

The obvious procedural way to do this would be to write a routine in your SQL's 4GL that reads a row from the Inventory table, and then write the value of good to the second table in a loop driven by the value of pieces.

This will be pretty slow, since it will require (`SELECT SUM(pieces) FROM Inventory`) single-row insertions into the working table.

Can you do better?

Answer #2

I always stress the need to think in terms of sets in SQL. The way to build a better solution is to do repeated self-insertion operations using a technique based on the "Russian peasant's algorithm," which was used for multiplication and division in early computers. You can look it up in a history of mathematics text or a computer science book—it is based on binary arithmetic and can be implemented with right and left shift operators in assembly languages.

You are still going to need a 4GL to do this, but it will not be so bad. First, let's create two working tables and one for the final answer:

```
CREATE TABLE WorkingTable1 - no key possible!!
(goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL);

CREATE TABLE WorkingTable2
(goods CHAR(10) NOT NULL,
pieces INTEGER NOT NULL);

CREATE TABLE Answer
(goods CHAR(10) NOT NULL,
 pieces INTEGER NOT NULL);
```

Now start by loading the goods that have only one piece in inventory into the answer table:

```
INSERT INTO Answer
SELECT * FROM Inventory WHERE pieces = 1;
```

Now put the rest of the data into the first working table:

```
INSERT INTO WorkingTable1
SELECT * FROM Inventory WHERE pieces > 1;
```

This block of code will load the second working table with pairs of rows that each has half (or half plus one) piece counts of those in the first working table:

```
INSERT INTO WorkingTable2
SELECT goods, FLOOR(pieces/2.0)
 FROM WorkingTable1
WHERE pieces > 1
UNION ALL
SELECT goods, CEILING(pieces/2.0)
 FROM WorkingTable1
WHERE pieces > 1;
```

The `FLOOR(x)` and `CEILING(x)` functions return, respectively, the greatest integer that is lower than *x* and smallest integer higher than *x*. If

your SQL does not have them, you can write them with rounding and truncation functions. It is also important to divide by (2.0) and not by 2, because this will make the result into a decimal number.

Now harvest the rows that have gotten down to a piece count of one and clear out the first working table:

```
INSERT INTO Answer
SELECT *
  FROM WorkingTable2
 WHERE pieces = 1;

DELETE FROM WorkingTable1;
```

Exchange the roles of `WorkingTable1` and `WorkingTable2`, and repeat the process until both working tables are empty. That is simple straightforward procedural coding. The way that the results shift from table to table is interesting to follow. Think of these diagrams as an animated cartoon:

Step 1: Load the first working table, harvesting any goods that already had a piece count of one.

```
WorkingTable1              WorkingTable2
goods      pieces          goods      pieces
==================         ==================
'Alpha'         4
'Beta'          5
'Delta'        16
'Gamma'        50
```

The row (`'Epsilon', 1`) goes immediately to Answer table.

Step 2: Halve the piece counts and double the rows in the second working table. Empty the first working table.

```
WorkingTable1              WorkingTable2
goods      pieces          goods      pieces
==================         ==================
                           'Alpha'         2
                           'Alpha'         2
                           'Beta'          2
```

```
                                'Beta'          3
                                'Delta'         8
                                'Delta'         8
                                'Gamma'        25
                                'Gamma'        25
```

Step 3: Repeat the process until both working tables are empty.

```
WorkingTable1                   WorkingTable2
goods       pieces              goods       pieces
=========== =======             =========== =======
'Alpha'         1
'Alpha'         1
'Alpha'         1
'Alpha'         1
'Beta'          1   'Alpha' and 'Beta' are ready to harvest
'Beta'          1
'Beta'          1
 --------------
'Beta'          2
'Delta'         4
'Delta'         4
'Delta'         4
'Delta'         4
'Gamma'        12
'Gamma'        12
'Gamma'        13
'Gamma'        13
```

The cost of completely emptying a table is usually very low.

Likewise, the cost of copying sets of rows (which are in physical blocks of disk storage that can be moved as whole buffers) from one table to another is much lower than inserting one row at a time.

The code could have been written to leave the results in one of the working tables, but this approach allows the working tables to get smaller and smaller so that you get better buffer usage. This algorithm uses (`SELECT SUM(pieces) FROM Inventory`) rows of storage and (`log2((SELECT MAX(pieces) FROM Inventory)) + 1`) moves, which is pretty good on both counts.

Answer #3

Peter Lawrence suggested another answer on CompuServe to the "uncount" problem. First, create a Sequence auxiliary table that contains all integers up to at least the maximum number of pieces (n):

```
CREATE TABLE Sequence (seq INTEGER NOT NULL PRIMARY KEY);
INSERT INTO Sequence VALUES (1), (2), ..., (n);
```

Or you can use:

```
INSERT INTO Sequence(seq)
WITH Digits (digit)
AS (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9))
SELECT D1.digit + 10*D2 + 100*D3 + ..
  FROM Digits AS D1, Digits AS D2, ..
 WHERE D1.digit + 10*D2 + 100*D3 + .. > 0;
```

Select the "uncount" as follows:

```
SELECT goods, 1 AS tally, seq
  FROM Inventory AS I1, Sequence AS S1
 WHERE I1.pieces >= S1.seq;
```

Note that the predicate "`T1.seq >= 1`" is redundant because of the `CHECK()` clause on the table declaration. I choose to leave it in this statement because (1) not all tables are declared with such clauses, and (2) it might help the optimizer.

The results should be:

```
Results
goods     tally  seq
====================
'CD-ROM'    1     1
'CD-ROM'    1     2
'CD-ROM'    1     3
'Printer'   1     1
'Printer'   1     2
...
```

Mr. Lawrence finds a table like Sequence above very useful and also frequently has a temporal table containing, say, every hour of a date/time range. This can be used for similar queries such as selecting every hour that someone was in the office when all the database contains is the start and end times.

I like this answer, and the simple `JOIN` should be faster than my elaborate shuffle between two working tables. Mr. Lawrence was not the only reader of my *DBMS* column to find a solution using this method.

Answer #4

Mary Attenborough also came up with the same solution, but her twist was a novel way of generating the table of consecutive numbers. This is another version of the Russian peasant's algorithm. Vinicius Mello then improved this method of creating the working table further by simplifying the math involved. The procedure looks like this:

```
BEGIN
DECLARE maxnum INTEGER NOT NULL;
DECLARE ntimes INTEGER NOT NULL;
DECLARE increment INTEGER NOT NULL;

INSERT INTO Sequence VALUES ((1), (2));

-- the count of rows in Sequence doubles each loop
SET maxnum = (SELECT MAX(pieces) FROM Inventory);
SET increment = 2;

WHILE increment < maxnum
   DO INSERT INTO Sequence
      SELECT seq + increment FROM Sequence;
      SET increment = increment + increment;
  END WHILE;
```

If we decide to make Sequence permanent, instead of loading it with a procedure, then we will need to see that some of the work gets done, leaving the items with a piece count greater than the highest `seq` still intact, thus:

```
SELECT goods, 1 AS tally, seq
  FROM Inventory AS I1, Sequence AS T1
```

```
 WHERE I1.pieces >= T1.seq
   AND T1.seq BETWEEN 1 AND MAX(I1.pieces);
```

A second approach would be to reject the whole query if we have a piece count greater than the highest `seq`, thus:

```
SELECT goods, 1 AS tally, seq
  FROM Inventory AS I1, Sequence AS T1
 WHERE I1.pieces >= T1.seq
   AND (SELECT MAX(I2.pieces) FROM Inventory AS I2)
       <= (SELECT MAX(T2.seq) FROM Sequence AS T2);
```

The subquery expressions are known to be constant for the life of the query, so the optimizer can do them once by going to an index in the case of the Sequence and with a table scan in the case of the Inventory table, since it is not likely to be indexed on the piece count.

Use a query to find the maximum number of pieces, and create only enough copies of digits in the `FROM` clause to build numbers that will cover it.

Answer #5

Another approach is to use a `JOIN` to a table with duplicate rows:

```
CREATE TABLE Repetitions - no key
(pieces INTEGER NOT NULL,
 one INTEGER DEFAULT 1 NOT NULL
    CHECK (one = 1));

INSERT INTO Repetitions
VALUES (2,1), (2,1), (3,1), (3,1), (3,1)..;

INSERT INTO WorkingTable
SELECT goods, one
  FROM Repetitions AS R1
       CROSS JOIN
       Inventory AS I1
 WHERE I1.pieces = R1.pieces
   AND I1.pieces > 1;
```

If the piece count is higher than the Repetitions table limits, run the insertion until no more rows are added.

Notes on Complexity

Let's compare the two approaches, assuming an Inventory table with (m) total rows and a maximum quantity of (n) for one or more items. Let the final table have (r) rows in it, where `r = (SELECT SUM(pieces) FROM Inventory)`. This implies that `r <= (m*n)`.

The Russian peasant algorithm approach will require $O(\log_2(n))$ iterations to solve the problem. It cuts the problem in half with each pass and has no join or search costs. The Russian peasant algorithm also has a cost for writing and rewriting the rows of one table into another. Each row is written and rewritten $\log_2(\text{pieces})$ times, which would total to $O(\log_2(r))$ for the whole table. This gives us a total cost of $O(\log_2(n) + \log_2(r))$.

The "sequential join" approach will require the time to build the Sequence plus the time to do the join. The iterative Sequence builds will be O(n). The join of Inventory and Sequence will be O(m * n), because this will be a cross join that has to be reduced. With a good index, this could be reduced to O(r) by avoiding some of the values in Sequence that are not needed. Therefore, the total cost will be O(n + r), which is higher than the Russian peasant's algorithm approach.

Having said all of that, the Russian peasant's algorithm will probably not be as fast in the real world as the sequential join approach because there is a really high cost to physically writing and rewriting the rows of one table into another. You would have the same (or worse) problems with just one table in which you updated an existing row's piece count by half and then inserted a new row in the table with the "other half" as its piece count.

PUZZLE
49 WIDGET COUNT

You get a production report from production centers that have a date, a production center code, and how many widgets were produced from each batch of raw materials sent to the center that day. It looks like this:

```
CREATE TABLE Production
(production_center INTEGER NOT NULL,
 wk_date DATE NOT NULL,
 batch_nbr INTEGER NOT NULL,
 widget_cnt INTEGER NOT NULL,
 PRIMARY KEY (production_center, wk_date, batch_nbr));
```

The boss comes in and wants to know the average number of widgets produced in all batches by date and production center. You say "No problem" and do it. The next day your boss comes back and wants the same data separated into three equal-sized batch groups. This sort of breakdown is important for certain types of statistical analysis of production work.

In other words, if on February 24, in production center 42, you processed 21 batches, your report will show the average number of widgets made from the first seven batches, the second seven batches, and the last seven batches. Write a query that will show, by work production center and date, the batch groups and the average number of widgets in each group.

Answer #1

The first query is straightforward:

```
SELECT production_center, wk_date, COUNT(batch_nbr),
AVG(widget_cnt)
  FROM Production
 GROUP BY production_center, wk_date;
```

You have to make some assumptions about the second query. I am assuming batches are numbered from 1 to (n), starting over every day. If the number of batches is not divisible by three, then do a best fit that accounts for all batches. Using the `CASE` expression in SQL-92, you can find which third a `batch_nbr` is contained in, using a `VIEW`, as follows:

```
CREATE VIEW Prod3 (production_center, wk_date, widget_cnt,
third)
AS SELECT production_center, wk_date, widget_cnt,
          CASE WHEN batch_nbr <= cont/3 THEN 1
               WHEN batch_nbr > (2 * cont)/3 THEN 3
               ELSE 2 END
     FROM Production, V1
    WHERE V1.production_center =
Production.production_center
      AND V1.wk_date = Production.wk_date;
```

If you do not have this in your SQL, then you might try something like this:

```
CREATE VIEW Prod3 (production_center, wk_date, third,
batch_nbr, widget_cnt)
   AS SELECT production_center, wk_date, 1, batch_nbr,
widget_cnt
        FROM Production AS P1
       WHERE batch_nbr <= (SELECT MAX(batch_nbr)/3
                             FROM Production AS P2
                            WHERE P1.production_center =
P2.production_center
                              AND P1.wk_date = P2.wk_date)
   UNION
   SELECT production_center, wk_date, 2, batch_nbr,
widget_cnt
     FROM Production AS P1
    WHERE batch_nbr > (SELECT MAX(batch_nbr)/3
                         FROM Production AS P2
                        WHERE P1.production_center =
P2.production_center
                          AND P1.wk_date = P2.wk_date)
      AND batch_nbr <= (SELECT 2 * MAX(batch_nbr)/3
                         FROM Production AS P2
                        WHERE P1.production_center =
P2.production_center
                          AND P1.wk_date = P2.wk_date)
   UNION
   SELECT production_center, wk_date, 3, batch_nbr,
widget_cnt
     FROM Production AS P1
    WHERE batch_nbr > (SELECT 2 * MAX(batch_nbr)/3
```

```
                          FROM Production AS P2
                         WHERE P1.production_center =
P2.production_center
                           AND P1.wk_date = P2.wk_date);
```

Answer #2

Either way, you end up with the final query:

```
SELECT production_center, wk_date, third, COUNT(batch_nbr),
AVG(widget_cnt)
  FROM Prod3
 GROUP BY production_center, wk_date, third;
```

PUZZLE
50 TWO OF THREE

We are putting together an anthology book with contributions from many other books (identified by their International Standard Book Number (ISBN)). As it works out, we want to find all authors who have articles in exactly two out of three categories in the book for a specified set of three categories that we put into the query as parameters.

```
CREATE TABLE AnthologyContributors
(isbn CHAR(10) NOT NULL,
 contributor CHAR(20) NOT NULL,
 category INTEGER NOT NULL,
 ...,
 PRIMARY KEY (isbn, contributor));
```

Answer #1

The first thought is that this is a simple `GROUP BY` query that would look like this:

```
SELECT isbn, contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
 GROUP BY isbn, contributor
HAVING COUNT(*) = 2;
```

But this will not work for two reasons. First, a `GROUP BY` on the key of a table gives you groups of one row. Second, one contributor might have made two contributions in only one area, but they will both be counted. What you needed was a `COUNT (DISTINCT <expression>)` aggregate function. This last problem is easy to fix with a `COUNT (DISTINCT A1.category) = 2;`

```
SELECT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
 GROUP BY contributor
HAVING COUNT(DISTINCT A1.category) = 2;
```

Can you find other ways of doing this without using a `GROUP BY`? I am not recommending any of the following solutions; the point of this exercise is to make you appreciate the `GROUP BY` clause.

Answer #2

The specification does not tell if we want *any* two of the three categories, or if we want them in a particular order (that is, category 1 and category 2, but not category 3). The latter is actually easy to do:

```
 SELECT A1.isbn, :cat_1, :cat_2
   FROM AnthologyContributors AS A1,
        AnthologyContributors AS A2
  WHERE A1.isbn = A2.isbn -- self-join table
    AND A1.category = :cat_1 -- category #1 first
    AND A2.category = :cat_2 -- category #2 second
    AND NOT EXISTS (SELECT *    -- but no category #3
anywhere
                      FROM AnthologyContributors AS A3
                     WHERE A1.isbn = A3.isbn
                       AND A3.category = :cat_3);
```

Answer #3

But the query to find any two out of three has to rely on some tricky coding. This answer will not tell you which two of the three is missing, however:

```
SELECT isbn, contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
   AND EXISTS
       (SELECT *
          FROM AnthologyContributors AS A2,
         WHERE A2.category IN (:cat_1, :cat_2, :cat_3)
           AND A1.category < A2.category
           AND A1.isbn = A2.isbn
           AND NOT EXISTS
                  (SELECT *
                     FROM AnthologyContributors AS A3,
                    WHERE A3.category
                         IN (:cat_1, :cat_2, :cat_3)
```

```
                    AND A1.isbn = A3.isbn
                    AND (A1.category <> A3.category
                        OR A2.category <> A3.category)));
```

To find the contributors who have something in all three categories, just change the `NOT EXISTS` to `EXISTS`.

To find the contributors who have only one category:

```
SELECT isbn, contributor, :cat_1
  FROM AnthologyContributors AS A1,
 WHERE A1.category = :cat_1
   AND NOT EXISTS
       (SELECT *
          FROM AnthologyContributors AS A2
         WHERE A2.category IN (:cat_2, :cat_3)
           AND A1.isbn = A2.isbn
           AND A1.category <> A2.category);
```

which is a "collapsed version" of the "two of three" query.

Answer #4

Until here, everything is okay. But take a look at Answer #3, with the restriction not to use `GROUP BY`. The answer is a little wrong because you are forgetting the condition that joins the contributor.

```
SELECT DISTINCT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE 2 = (SELECT COUNT(DISTINCT A2.category)
              FROM AnthologyContributors AS A2
             WHERE A1.contr_num = A2.contr_num
               AND A1.contributor = A2.contributor
               AND A2.category IN (:cat_1, :cat_2, :cat_3));
```

Answer #5

This is the best answer in the bunch. It neatly handles the requirement that we have the first two categories and not the third.

```
SELECT DISTINCT contributor, :cat_1, :cat_2
  FROM AnthologyContributors AS A1
```

```
 WHERE (SELECT SUM(DISTINCT
                     CASE WHEN category = :cat_1
                          THEN 1
                          WHEN category = :cat_2
                          THEN 2
                          WHEN category = :cat_3
                          THEN -3 ELSE NULL END)) = 3
          FROM AnthologyContributors AS A2
         WHERE A1.contr_num = A2.contr_num
           AND A1.contributor = A2.contributor
           AND A2.category IN (:cat_1, :cat_2, :cat_3));
```

Of course, it is possible again to use the `GROUP BY` clause:

```
SELECT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE A1.category IN (:cat_1, :cat_2, :cat_3)
 GROUP BY contributor
HAVING (SELECT SUM(DISTINCT
                     CASE WHEN category = :cat_1
                          THEN 1
                          WHEN category = :cat_2
                          THEN 2
                          WHEN category = :cat_3
                          THEN -3 ELSE NULL END)) = 3;
```

Answer #6

Find the contributors who have *all three* categories. This is straightforward using the same basic pattern:

```
SELECT DISTINCT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE (SELECT COUNT(DISTINCT A2.category)
          FROM AnthologyContributors AS A2
         WHERE A1.contr_num = A2.contr_num
           AND A1.contributor = A2.contributor
          AND A2.category IN (:cat_1, :cat_2, :cat_3)) = 3;
```

The GROUP BY version is straightforward. If the question was to find the contributors who have something in any of the three categories, the answer is:

```
SELECT DISTINCT contributor, :cat_1, :cat_2, :cat_3
  FROM AnthologyContributors AS A1
 WHERE category IN (:cat_1, :cat_2, :cat_3);
```

PUZZLE
51 BUDGET VERSUS ACTUAL

C. Conrad Cady posted a simple SQL problem on the CompuServe Gupta Forum. He has two tables, Budgeted and Actual, which describe how a project is being done. Budgeted has a one-to-many relationship with Actual. The tables are defined like this:

```
CREATE TABLE Budgeted
(task INTEGER NOT NULL PRIMARY KEY,
 category INTEGER NOT NULL,
 est_cost DECIMAL(8,2) NOT NULL);

CREATE TABLE Actual
(voucher DECIMAL(8,2) NOT NULL PRIMARY KEY,
 task INTEGER NOT NULL REFERENCES Budgeted(task),
 act_cost DECIMAL(8,2) NOT NULL);
```

He wants a Budgeted versus Actual comparison for each category. This is easier to see with an example:

```
Budgeted
task category est_cost
======================
    1    9100  100.00
    2    9100   15.00
    3    9100    6.00
    4    9200    8.00
    5    9200   11.00

Actual
voucher task act_cost
=====================
    1     1     10.00
    2     1     20.00
    3     1     15.00
    4     2     32.00
    5     4      8.00
    6     5      3.00
    7     5      4.00
```

The output he wants is this:

```
Result
category estimated   spent
=========================
   9100    121.00    77.00
   9200     19.00    15.00
```

The $121.00 is the sum of the `est_cost` of the three task items in category 9100. The $77.00 is the sum of the `act_cost` of the four voucher items related to those three task items (three amounts are related to the first item, one to the second, and none to the third).

He tried the query

```
SELECT category, SUM(est_cost) AS estimated,
                 SUM(act_cost) AS spent
 FROM (Budgeted LEFT OUTER JOIN Actual
      ON Budgeted.task = Actual.task)
 GROUP BY category;
```

and he got:

```
Result
category estimated   spent
=========================
   9100   321.00   77.00
   9200    31.00   15.00
```

The problem is that the $100.00 is counted three times in the `JOIN`, giving $321.00 instead of $121.00, and the $11.00 is counted twice, giving $31.00 instead of $19.00 in the `JOIN`.

Is there a simple, single piece of SQL that will give him the output he wants, given the above tables?

Answer #1

Bob Badour suggested that he can get the required result by creating a view in SQL-89:

```
CREATE VIEW cat_costs (category, est_cost, act_cost)
AS SELECT category, est_cost, 0.00
```

```
    FROM Budgeted
  UNION
  SELECT category, 0.00, act_cost
    FROM Budgeted, Actual
   WHERE Budgeted.task = Actual.task;
```

followed by the query:

```
SELECT category, SUM(est_cost), SUM(act_cost)
  FROM cat_costs
 GROUP BY category;
```

In SQL-92, we can join the total amounts spent on each task to the category in the Budgeted table, like this:

```
SELECT B1.category, SUM(est_cost), SUM(spent)
  FROM Budgeted AS B1
       LEFT OUTER JOIN
       (SELECT task, SUM(act_cost) AS spent
          FROM Actual AS A1
         GROUP BY task)

       ON A1.task = B1.task
 GROUP BY B1.category;
```

The `LEFT OUTER JOIN` will handle situations where no money has been spent yet. If you have a transitional SQL that does not allow subqueries in a `JOIN`, then extract the subquery shown here and put it in a `VIEW`.

Answer #2

Here is an answer from Francisco Moreno of Colombia that uses a scalar subquery with a `GROUP BY` clause. Notice that the `MIN()` and `MAX()` values are in the containing query:

```
SELECT category, SUM(B1.est_cost) AS estimated,
       (SELECT SUM(T1.act_cost)
          FROM Actual AS T1
         WHERE T1.task
```

```
                BETWEEN MIN(B1.task) AND MAX(B1.task)) AS
spent
  FROM Budgeted AS B1
 GROUP BY category;
```

PUZZLE
52 PERSONNEL PROBLEM

Daren Race was trying to aggregate the results from an aggregate result set using Gupta's SQLBase and could not think of any way other than using a temporary table or a `VIEW`. This is an example of what he was doing:

```
Personnel:
emp_name  dept_id
=================
'Daren' 'Acct'
'Joe'   'Acct'
'Lisa'  'DP'
'Helen' 'DP'
'Fonda' 'DP'
```

Then he viewed the data as an aggregate by `dept_id`:

```
SELECT dept_id, COUNT(*)
 FROM Personnel
 GROUP BY dept_id;
```

The results will be:

```
Result
dept_id COUNT(*)
===============
 Acct    2
 DP      3
```

Then he wanted to find the average department size! The way he did this was to use a `VIEW`:

```
CREATE VIEW DeptView (dept_id, tally)
AS SELECT dept_id, COUNT(*)
     FROM Personnel
    GROUP BY dept_id;
```

Then:

```
SELECT AVG(tally) FROM DeptView;
```

He asked if anyone on the Centura (nee Gupta) Forum on CompuServe could think of a way of doing this without using temporary tables (or `views`). He got two answers, namely:

```
SELECT AVG(DISTINCT dept_id)
 FROM Personnel;
```

and

```
SELECT COUNT(*) / COUNT(DISTINCT dept_id)
 FROM Personnel;
```

Your problem is to tell me what is wrong with each of them.

Answer #1

The first answer makes absolutely no sense at all. The department codes are alphabetic and not numeric. This has nothing to do with the number of people in each department.

The second answer is really much better and will give us the right results for this data. We have a `COUNT(*) = 5`, and `COUNT(DISTINCT dept_id) = 2`, so the answer we get is (2.5), just as we wished.

But now we hire three new employees, Larry, Moe, and Curly, who are not yet assigned to a department, and our table looks like this:

```
Personnel:
emp_name  dept_id
===============
 Daren  Acct
 Joe    Acct
 Lisa   DP
 Helen  DP
 Fonda  DP
 Larry  NULL
 Moe    NULL
 Curly  NULL
```

We now have a `COUNT(*) = 8`, but `COUNT(DISTINCT dept_id) = 2` because the function will drop all the `NULL`s before it counts the `dept_ids`,

so the answer we get is 4. The problem is that we need to decide what to do with Larry, Moe, and Curly. Possible ways to do that are:

1. Each one of the new guys is in his own new department (5 departments).
2. They are all in a special department shown by `NULL` (3 departments).
3. Three in DP.
4. One in DP, two in Accounting.
5. Three in Accounting.
6. One in Accounting, two in DP.
7. One in DP, two in a new department.
8. One in Accounting, two in a new department.
9. One in DP, two in a new department.
10. One in Accounting, one in DP, and one in a new department.
11. One in Accounting, one in a new department, one in a second new department.
12. One in DP, one in a new department, one in a second new department.

This puts the average department size somewhere between (8/2) = 4.0 and (8/5) = 1.6 people. If Mr. Race had stuck to his original method, we would have gotten:

```
Result
dept_id COUNT(*)
===============
 Acct    2
 DP      3
 NULL    3
```

and a final result of 1.5 as before because the `NULL`s would form a group by themselves in the `VIEW`, but then been dropped out by the average in the final query.

PUZZLE
53 COLLAPSING A TABLE BY COLUMNS

Robert Brown proposed this one in 2004. Suppose I have the following table:

```
CREATE TABLE Foobar
(lvl INTEGER NOT NULL PRIMARY KEY,
 color VARCHAR(10),
 length INTEGER,
 width INTEGER,
 hgt INTEGER);

INSERT INTO Foobar
VALUES (1, 'RED', 8, 10, 12),
       (2, NULL, NULL, NULL, 20),
       (3, NULL, 9, 82, 25),
       (4, 'BLUE', NULL, 67, NULL),
       (5, 'GRAY', NULL, NULL, NULL);
```

I want to write a query that will return me a view collapsed from "bottom-to-top" in order of level (lvl = 1 is the top, lvl = 5 is the bottom). That means that the sample data would return:

```
('GRAY', 9, 67, 25)
```

The principle is that looking from the bottom level up in each column, we first see 'GRAY' for color, 9 for length, 67 for width, and 25 for height. In other words, any non-null row in a lower level overrides the value set at a higher level.

Answer #1

John Gilson came up with two answers:

```
-- Option 1 uses scalar subqueries


SELECT (SELECT color FROM Foobar WHERE lvl = M.lc) AS color,
       (SELECT length FROM Foobar WHERE lvl = M.ll) AS
length,
```

```
       (SELECT width FROM Foobar WHERE lvl = M.lw) AS width,
       (SELECT hgt FROM Foobar WHERE lvl = M.lh) AS hgt
  FROM (SELECT MAX(CASE WHEN color IS NOT NULL THEN lvl END)
AS lc,
             MAX(CASE WHEN length IS NOT NULL THEN lvl END)
AS ll,
              MAX(CASE WHEN width IS NOT NULL THEN lvl END)
AS lw,
               MAX(CASE WHEN hgt IS NOT NULL THEN lvl END)
AS lh
          FROM Foobar) AS M;

-- Option 2
SELECT MIN(CASE WHEN Foobar.lvl = M.lc THEN Foobar.color
END) AS color,
       MIN(CASE WHEN Foobar.lvl = M.ll THEN Foobar.length
END) AS length,
       MIN(CASE WHEN Foobar.lvl = M.lw THEN Foobar.width
END) AS width,
      MIN(CASE WHEN Foobar.lvl = M.lh THEN Foobar.hgt END)
AS hgt
  FROM (SELECT MAX(CASE WHEN color IS NOT NULL THEN lvl END)
AS lc,
             MAX(CASE WHEN length IS NOT NULL THEN lvl END)
AS ll,
              MAX(CASE WHEN width IS NOT NULL THEN lvl END)
AS lw,
               MAX(CASE WHEN hgt IS NOT NULL THEN lvl END)
AS lh
          FROM Foobar) AS M
       INNER JOIN
       Foobar

       ON Foobar.lvl IN (M.lc, M.ll, M.lw, M.lh)
```

Answer #2

This was my attempt. `COALESCE` is executed in the order written, so I can go from bottom to top easily to get the first non-NULL back.

```
SELECT COALESCE (F5.color, F4.color, F3.color, F2.color,
F1.color) AS color,
      COALESCE (F5.length, F4.length, F3.length, F2.length,
F1.length) AS length,
```

```
       COALESCE (F5.width, F4.width, F3.width, F2.width,
F1.width) AS width,
       COALESCE (F5.hgt, F4.hgt, F3.hgt, F2.hgt, F1.hgt) AS
hgt
  FROM Foobar AS F1, Foobar AS F2, Foobar AS F3,
       Foobar AS F4, Foobar AS F5
 WHERE F1.lvl = 1
   AND F2.lvl = 2
   AND F3.lvl = 3
   AND F4.lvl = 4
   AND F5.lvl = 5;
```

PUZZLE
54 POTENTIAL DUPLICATES

Ronny Weisz posted this problem he was having with an application in DB2. The store has approximately 20,000 customers, and over time they notice data problems due to typos, inaccurate data entry, separate listings for different family members (in their context the family is really a single customer), and so forth. Occasionally they try to clean up their Customers table by preparing a report with potential duplications.

The definition of a potential duplicate is that the rows have the same surname and match two of the five columns related to street address: `first_name`, `street_address`, `city_name`, `state_code`, and `phone_nbr`. The first attempt was something like this:

```
CREATE VIEW Dups (custnbr, last_name, first_name,
street_address, city_name, state_code, phone_nbr, m)
AS
SELECT C0.custnbr, C0. last_name, C0.first_name,
C0.street_address, C0.city_name,
       C0.state_code, C0.phone_nbr,
       (CASE WHEN C0.first_name = C1.first_name THEN 1 ELSE
0 END)
       + (CASE WHEN C0.street_address = C1.street_address
THEN 1 ELSE 0 END)
       + (CASE WHEN C0.city_name = C1.city_name THEN 1 ELSE
0 END)
       + (CASE WHEN C0.state_code = C1.state_code THEN 1
ELSE 0 END)
       + (CASE WHEN C0.phone_nbr = C1.phone_nbr THEN 1 ELSE
0 END) AS m
   FROM Customers AS C1, Customers AS C0
WHERE C0.custnbr <> C1.custnbr
  AND C0.last_name = C1.last_name);

SELECT DISTINCT *
  FROM Dups
 WHERE m >= 2
 ORDER BY surname;
```

You need the `DISTINCT` since a customer can appear more than once. For example, if `A1` matches `A3`, `A5`, and `A6`, then there will be three `A1`

rows in Dups. But performance is not great, so your job is to find an improvement.

Answer #3

Johannes Becher of CODATA in Germany came up with the following answer:

```
SELECT D0.cust_nbr
   FROM Dups AS D0
  WHERE EXISTS (SELECT D1.id
                  FROM Dups AS D1
                 WHERE D0.surname = D1.surname
                   AND D0.cust_nbr <> D1.cust_nbr
                   AND (CASE WHEN D0.first_name =
D1.first_name
                             THEN 1 ELSE 0 END)
                     + (CASE WHEN D0.street_address =
D1.street_address
                             THEN 1 ELSE 0 END)
                    + (CASE WHEN D0.city_name = D1.city_name
                             THEN 1 ELSE 0 END)
                     + (CASE WHEN D0.state_code =
D1.state_code
                             THEN 1 ELSE 0 END)
                    + (CASE WHEN D0.phone_nbr = D1.phone_nbr
                             THEN 1 ELSE 0 END) >= 2);
```

Note that he added the (`D0.cust_nbr <> D1.cust_nbr`) clause so that every row would not qualify as a duplicate of itself. Also, he moved the `CASE` expressions from the `SELECT` list, where it was a calculated column, to the `WHERE` clause. This made the addition an expression in a predicate where an optimizer could do shortcut evaluation. In a shortcut evaluation, as soon as you know the predicate is `TRUE` or `FALSE`, then you stop calculating and return an answer. As a calculated column, it had to be evaluated completely so it could be materialized in the `VIEW`.

Going one step further, if you know the optimizer's order of evaluation (left to right or right to left) of the `CASE` expressions and the columns mostly to be matched, then you can arrange the expressions for even better performance. For example, `city_name` and `state_code` are probably not often misspelled and should come last, while people and street names are often wrong.

Having said all of this, the best way to handle this kind of problem is to use specialized packages that are designed for cleaning up mailing lists. You can take a look at the offerings from Melissa Data (http://www.melissadata.com/).

PUZZLE
55 PLAYING THE PONIES

You have just become the database manager for your bookie. He keeps records on horse races for statistical purposes, and his basic table looks like this:

```
CREATE TABLE RacingResults
(track_id CHAR(3) NOT NULL,
 race_date DATE NOT NULL,
 race_nbr INTEGER NOT NULL,
 win_name CHAR(30) NOT NULL,
 place_name CHAR(30) NOT NULL,
 show_name CHAR(30) NOT NULL,
 PRIMARY KEY (track_id, race_nbr_date, race_nbr));
```

The `track_id` column is the name of the track where the race was held, `race_date` is when it was held, `race_nbr` is number of the each race, and the other three columns are the names of the horses that won, placed, or showed for that race. If you do not know these terms, won means that the horse was in first place; placed means that the horse was in first or second place, and showed means the horse was in first, second, or third place.

Your bookie comes to you one day and wants to know how many times each horse was in the money. What SQL query do you write for this?

Answer #1

The phrase "in the money" means that the horse won, placed, or showed in a race—we don't care which. The first step is to build a `VIEW` with the aggregate information, thus:

```
CREATE VIEW InMoney (horse, tally, position) AS
SELECT win_name, COUNT(*), 'win_name'
  FROM RacingResults
 GROUP BY win_name
UNION ALL
SELECT place_name, COUNT(*), 'place_name'
  FROM RacingResults
 GROUP BY place_name
```

```
UNION ALL
SELECT show_name, COUNT(*), 'show_name'
  FROM RacingResults
 GROUP BY show_name;
```

Now use that view to get the final summary:

```
SELECT horse, SUM(tally)
  FROM InMoney
 GROUP BY horse;
```

There are two reasons for putting those string constants in the SELECT lists. The first is so that we will not drop duplicates incorrectly in the UNION ALL. The second reason is so that if the bookie wants to know how many times each horse finished in each position, you can just change the query to:

```
SELECT horse, position, SUM(tally)
  FROM InMoney
 GROUP BY horse, position;
```

Answer #2

If you have a table with all the horses in it, you can write the query as:

```
 SELECT H1.horse, COUNT(*)
   FROM HorseNames AS H1, RacingResults AS R1
  WHERE H1.horse IN (R1.win_name, P1.place_name,
R1.show_name)
  GROUP BY H1.horse;
```

If you use an OUTER JOIN, you can also see the horse that did not show up in the RacingResults table. There is an important design principle demonstrated here; it is hard to tell if something is an entity or an attribute. A horse is an entity and therefore should be in a table. But the horse's name is also used as a value in three columns in the RacingResults table.

Answer #3

Another approach that requires a table of the horses' names is to build the totals with scalar subqueries.

```
SELECT H1.horse,
       (SELECT COUNT(*)
          FROM RacingResults AS R1
         WHERE R1.win_name = H1.horse)
       + (SELECT COUNT(*)
            FROM RacingResults AS R1
           WHERE R1.place_name = H1.horse)
       + (SELECT COUNT(*)
            FROM RacingResults AS R1
           WHERE R1.show_name = H1.horse)
  FROM Horses AS H1;
```

While this works, it is probably going to be expensive to execute.

PUZZLE 56 HOTEL ROOM NUMBERS

Ron Hiner put this question on CompuServe. He had a data conversion project where he needed to automatically assign some values to use as part of the PRIMARY KEY to a table of hotel rooms.

The floor number is part of the PRIMARY KEY is the FOREIGN KEY to another table of floors within the building. The part of the hotel key we need to create is the room number, which has to be a sequential number starting at x01 for each floor number x. The hotel is small enough that we know we will only have three-digit numbers. The table is defined as follows:

```
CREATE TABLE Hotel
(floor_nbr INTEGER NOT NULL,
 room_nbr INTEGER,
 PRIMARY KEY (floor_nbr, room_nbr),
 FOREIGN KEY floor_nbr REFERENCES Bldg(floor_nbr);
```

Currently, the data in the working table looks like this:

```
floor_nbr  room_nbr
===================
    1  NULL
    1  NULL
    1  NULL
    2  NULL
    2  NULL
    3  NULL
```

WATCOM (and other versions of SQL back then) had a proprietary NUMBERS(*) function that begins at 1 and returns an incremented value for each row that calls it. The current SQL Standard now has a DENSE_RANK () OVER(<window expression>) function that makes this easy to compute, but can you do it the old-fashion way?

Is there an easy way via the numbering functions (or some other means) to automatically populate the room_nbr column? Mr. Hiner was thinking of somehow using a "GROUP BY floor_nbr" clause to restart the numbering back at 1.

Answer #1

The WATCOM support people came up with this approach. First, make one updating pass through the whole database, to fill in the `room_nbr` numbers. This trick will not work unless you can guarantee that the Hotel table is updated in sorted order. As it happens, WATCOM can guarantee just that with a clause on the `UPDATE` statement, thus:

```
UPDATE Hotel
   SET room_nbr = (floor_nbr*100)+NUMBER(*)
 ORDER BY floor_nbr;
```

which would give these results:

```
room_nbr
==========
   1   101
   1   102
   1   103
   2   204
   2   205
   3   306
```

followed by:

```
UPDATE Hotel
   SET room_nbr = (room_nbr - 3)
 WHERE floor_nbr = 2;

UPDATE Hotel
   SET room_nbr = (room_nbr - 5)
 WHERE floor_nbr = 3;
```

which would give the correct results:

```
floor_nbr room_nbr
==========
   1   101
   1   102
   1   103
   2   201
   2   202
   3   301
```

Unfortunately, you have to know quite a bit about the number of rooms in the hotel. Can you do better without having to use the `ORDER BY` clause?

Answer #2

I would use SQL to write SQL statements. This is a neat trick that is not used enough. Just watch your quotation marks when you do it and remember to convert numeric values to characters, thus:

```
SELECT DISTINCT
       'UPDATE Hotel SET room_nbr = ('
       || CAST (floor_nbr AS CHAR(1))
       || '* 100)+NUMBER(*) WHERE floor_nbr = '
       || CAST (floor_nbr AS CHAR(1)) || ';'
 FROM Hotel;
```

This statement will write a result table with one column that has a test, like this:

```
UPDATE Hotel SET room_nbr = (floor_nbr*100)+NUMBER(*) WHERE
floor_nbr = 1;
UPDATE Hotel SET room_nbr = (floor_nbr*100)+NUMBER(*) WHERE
floor_nbr = 2;
UPDATE Hotel SET room_nbr = (floor_nbr*100)+NUMBER(*) WHERE
floor_nbr = 3;
  ...
```

Copy this column as text to your interactive SQL tool or into a batch file and execute it. This does not depend on the order of the rows in the table.

You could also put this into the body of a stored procedure and pass the `floor_nbr` as a parameter. You are only going to do this once, so writing and compiling procedure is not going to save you anything.

Answer #3

What was such a problem in older SQLs is now trivial in SQL-99.

```
UPDATE Hotel
   SET room_nbr
      = (floor_nbr * 100)
         + ROW_NUMBER()OVER (PARTITION BY floor_nbr);
```

PUZZLE
57 GAPS—VERSION ONE

This is a classic SQL programming problem that comes up often in newsgroups. In the simplest form, you have a table of unique numbers and you want to either find out if they are in sequence or find the gaps in them. Let's construct some sample data.

```
CREATE TABLE Numbers (seq INTEGER NOT NULL PRIMARY KEY);
INSERT INTO Numbers
VALUES (2), (3), (5), (7), 8), (14), (20);
```

Answer #1

Finding out if you have a sequence from 1 to (n) is very easy. This will not tell you where the gaps are, however.

```
SELECT CASE WHEN COUNT(*) = MAX(seq)
       THEN 'Sequence' ELSE 'Not Sequence' END FROM Numbers;
```

The math for the next one is obvious, but this test does not check that the set starts at one (or at zero). It is only for finding if a gap exists in the range.

```
SELECT CASE WHEN COUNT(*) + MIN(seq) - 1 = MAX(seq)
       THEN 'Sequence' ELSE 'Not Sequence' END FROM Numbers;
```

Answer #2

This will find the starting and ending values of the gaps. But you have to add a sentinel value of zero to the set of `Numbers`.

```
SELECT N1.seq+1 AS gap_start, N2.seq-1 AS gap_end
  FROM Numbers AS N1, Numbers AS N2
 WHERE N1.seq +1 < N2.seq
   AND (SELECT SUM(seq)
          FROM Numbers AS Num3
         WHERE Num3.seq BETWEEN N1.seq AND N2.seq)
         = (N1.seq + N2.seq);
```

Bad starts are common in this problem. For example, this query will return only the start of a gap and one past the maximum value in the Numbers table, and it misses 1 if it is not in Numbers.

```
-- does not work; only start of gaps
SELECT N1.seq + 1
  FROM Numbers AS N1
       LEFT OUTER JOIN
       Numbers AS N2
       ON N1.seq = N2.seq -1
 WHERE N2.seq IS NULL;
```

A more complicated but accurate way to find the first missing number is:

```
--first missing seq
SELECT CASE WHEN MAX(seq) = COUNT(*)
       THEN MAX(seq) + 1
       WHEN MIN(seq) < 1
       THEN 1
       WHEN MAX(seq) <> COUNT(*)
       THEN (SELECT MIN(seq)+1
               FROM Numbers
              WHERE (seq + 1)
               NOT IN (SELECT seq FROM Numbers))
       ELSE NULL END
  FROM Numbers;
```

The first case adds the next value to Numbers if there is no gap. The second case fills in the value 1 if it is missing. The third case finds the lowest missing value.

Answer #3

Let's use the usual Sequence auxiliary table and one of the SQL-99 set operators:

```
SELECT X.seq
  FROM ((SELECT seq FROM Sequence AS S1)
        EXCEPT ALL
        (SELECT seq FROM Numbers AS N1
```

```
        WHERE seq <= (SELECT MAX(seq) FROM Numbers))
) AS X(seq);
```

Notice that I used `EXCEPT ALL`, since there are no duplicates in either table. You cannot trust the optimizer to always pick up on that fact when a feature is this new.

PUZZLE
58 GAPS—VERSION TWO

Here is a second version of the classic SQL programming problem of finding gaps in a sequence. How many ways can you do it? Can you make it better with SQL-92 and SQL-99 features?

```
CREATE TABLE Tickets
(buyer_name CHAR(5) NOT NULL,
 ticket_nbr INTEGER DEFAULT 1 NOT NULL
          CHECK (ticket_nbr > 0),
 PRIMARY KEY (buyer_name, ticket_nbr));

INSERT INTO Tickets
VALUES ('a', 2), ('a', 3), ('a', 4),
       ('b', 4),
       ('c', 1), ('c', 2), ('c', 3), ('c', 4), ('c', 5),
       ('d', 1), ('d', 6), ('d', 7), ('d', 9),
       ('e', 10);
```

Answer #1

Tom Moreau, another well-known SQL author in Toronto, came up with this solution that does not use a `UNION ALL`. It finds buyers with a gap in the tickets they hold, but it does not "fill in the holes" for you. For example, Mr. D is holding (1, 6, 7, 9) so he has gaps for (2, 3,4, 5, 8), but Tom did not count Mr. A because there is no gap within the range he holds.

```
SELECT buyer_name
  FROM Tickets
 GROUP BY buyer_name
HAVING NOT (MAX(ticket_nbr) - MIN(ticket_nbr) <= COUNT
(*));
```

If we can assume that there is a relatively small number of tickets, then you could use a table of sequential numbers from 1 to (n) and write:

```
SELECT DISTINCT T1.buyer_name, S1.seq
  FROM Tickets AS T1, Sequence AS S1
 WHERE seq <= (SELECT MAX(ticket_nbr) -- SET the range
```

```
                FROM Tickets AS T2
               WHERE T1.buyer_name = T2.buyer_name)
  AND seq NOT IN (SELECT ticket_nbr -- get missing numbers
                    FROM Tickets AS T3
                   WHERE T1.buyer_name = T3.buyer_name);
```

Another trick here is to add a zero to act as a boundary when 1 is missing from the sequence. In standard SQL-92, you could write the `union all` expression directly in the `FROM` clause.

Answer #2

A Liverpool fan proposed this query:

```
SELECT DISTINCT T1.buyer_name, S1.seq
  FROM Tickets AS T1, Sequence AS S1
 WHERE NOT EXISTS
      (SELECT *
         FROM Tickets AS T2
        WHERE T2.buyer_name = T1.buyer_name
          AND T2.ticket_nbr = S1);
```

but it lacks an upper limit on the `Sequence.seq` value used.

Answer #3

Omnibuzz avoided the `DISTINCT` and came up with this query, which does put a limit on the `Sequence`.

```
SELECT T2.buyer_name, T2.ticket_nbr
  FROM (SELECT T1.buyer_name, S1.seq AS ticket_nbr
          FROM (SELECT buyer_name, MAX(ticket_nbr)
                  FROM Tickets
                 GROUP BY buyer_name)
               AS T1(buyer_name, max_nbr),
               Sequence AS S
         WHERE S1.seq <= max_nbr
        ) AS T2
       LEFT OUTER JOIN
       Tickets AS T3
       ON T2.buyer_name = T3.buyer_name
```

```
          AND T2.ticket_nbr = T3.ticket_nbr
        WHERE T3.buyer_name IS NULL;
```

Answer #4

Dieter Noeth uses the SQL:1999 OLAP functions to calculate the "previous value." If the difference to the "current" value is greater than 1 there's a gap. Since `Sequence` starts at 1, we need the `COALESCE` to add a dummy "`prev_value`" of 1.

```
SELECT buyer_name, (prev_nbr + 1) AS gap_start,
       (ticket_nbr - 1) AS gap_end
  FROM (SELECT buyer_name, ticket_nbr,
               COALESCE(MIN(ticket_nbr) OVER (PARTITION BY
buyer_name
                        ORDER BY ticket_nbr
                        ROWS BETWEEN 1 PRECEDING AND 1
PRECEDING), 0)
          FROM Tickets
       ) AS DT(buyer_name, ticket_nbr, prev_nbr)
 WHERE (ticket_nbr - prev_nbr) > 1;
```

Answer #5

Omnibuzz came up with another one using a common table expression (CTE), no sequence table, and no `DISTINCT`.

```
WITH CTE(buyer_name, ticket_nbr)
AS
(SELECT buyer_name, MAX(ticket_nbr)
   FROM Tickets
  GROUP BY buyer_name
UNION ALL
SELECT buyer_name, ticket_nbr - 1
  FROM CTE
 WHERE ticket_nbr - 1 >= 0
)

SELECT A.buyer_name, A.ticket_nbr
  FROM CTE AS A
       LEFT OUTER JOIN
       Tickets AS B
```

```
      ON A.buyer_name = B.buyer_name
         AND A.ticket_nbr = B.ticket_nbr
WHERE B.buyer_name IS NULL;
```

Notice that this is a recursive CTE expression that generates the complete range of ticket numbers. The main `SELECT` statement is doing a set difference with an `OUTER JOIN`.

PUZZLE
59 MERGING TIME PERIODS

When you have a timesheet, you often need to merge contiguous or overlapping time periods. This can be a problem to do in a simple query, so be careful as this is not easy to follow or understand:

```
CREATE TABLE Timesheets
(task_id CHAR(10) NOT NULL PRIMARY KEY,
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 CHECK(start_date <= end_date));

INSERT INTO Timesheets
VALUES (1, '1997-01-01', '1997-01-03'),
       (2, '1997-01-02', '1997-01-04'),
       (3, '1997-01-04', '1997-01-05'),
       (4, '1997-01-06', '1997-01-09'),
       (5, '1997-01-09', '1997-01-09'),
       (6, '1997-01-09', '1997-01-09'),
       (7, '1997-01-12', '1997-01-15'),
       (8, '1997-01-13', '1997-01-14'),
       (9, '1997-01-14', '1997-01-14'),
       (10, '1997-01-17', '1997-01-17');
```

Answer #1

```
SELECT T1.start_date, MAX(T2.end_date)
  FROM Timesheets AS T1, Timesheets AS T2
WHERE T1.start_date < T2.end_date
   AND NOT EXISTS
       (SELECT *
          FROM Timesheets AS T3, Timesheets AS T4
         WHERE T3.end_date < T4.start_date
           AND T3.start_date >= T1.start_date
           AND T3.end_date <= T2.end_date
           AND T4.start_date >= T1.start_date
           AND T4.end_date <= T2.end_date
           AND NOT EXISTS
               (SELECT *
                  FROM Timesheets AS T5
```

```
              WHERE T5.start_date BETWEEN
T3.start_date AND T3.end_date
                                                AND T5.end_date
BETWEEN T4.start_date AND T4.end_date))
GROUP BY T1.start_date
HAVING t1.start_date = MIN(t2.start_date);

Results:
start_date      end_date
=======================
1997-01-01    1997-01-05
1997-01-04    1997-01-05
1997-01-06    1997-01-09
1997-01-12    1997-01-15
```

Answer #2

It is a long query, but check the execution time.

```
SELECT X.start_date, MIN(Y.end_date) AS end_date
  FROM (SELECT T1.start_date
          FROM Timesheets AS T1
               LEFT OUTER JOIN
               Timesheets AS T2
               ON T1.start_date > T2.start_date
                  AND T1.start_date <= T2.end_date
         GROUP BY T1.start_date
        HAVING COUNT(T2.start_date) = 0) AS X(start_date)
       INNER JOIN
       (SELECT T3.end_date
          FROM Timesheets AS T3
               LEFT OUTER JOIN
               Timesheets AS T4
               ON T3.end_date >= T4.start_date
                  AND T3.end_date < T4.end_date
         GROUP BY T3.end_date
        HAVING COUNT(T4.start_date) = 0) AS Y(end_date)
    ON X.start_date <= Y.end_date
 GROUP BY X.start_date;
```

```
Results
start_date      end_date
-------------------------
1997-01-01     1997-01-05
1997-01-06     1997-01-09
1997-01-12     1997-01-15
1997-01-17     1997-01-17
```

Answer #3

```
SELECT X.start_date, MIN(X.end_date) AS end_date
  FROM (SELECT T1.start_date, T2.end_date
          FROM Timesheets AS T1, Timesheets AS T2,
Timesheets AS T3
         WHERE T1.end_date <= T2.end_date
         GROUP BY T1.start_date, T2.end_date
        HAVING COUNT (CASE
                      WHEN (T1.start_date > T3.start_date
                           AND T1.start_date <= T3.end_date)
                            OR (T2.end_date = T3.start_date
                              AND T2.end_date < T3.end_date)
                      THEN 1 ELSE 0 END) = 0) AS X
 GROUP BY X.start_date;

Results
start_date     end_date
========================
1997-01-01    1997-01-05
1997-01-06    1997-01-09
1997-01-12    1997-01-15
1997-01-17    1997-01-17
```

There is a lot of logic crammed into that little query.

PUZZLE
60 BARCODES

In a recent posting on www.swug.org, a regular contributor posted a T-SQL function that calculates the checksum digit of a standard, 13-digit barcode. The algorithm is a simple weighted sum method (see *Data & Databases*, Section 15.3.1, if you do not know what that means). Given a string of 13 digits, you take the first 12 digits of the string of the barcode, use a formula on them, and see if the result is the 13th digit. The rules are simple:

1. Sum each digit in an odd position to get S1.
2. Sum each digit in an odd position to get S2.

Subtract S2 from S1, do a modulo 10 on the sum, and then compute the absolute positive value. The formula is ABS(MOD(S1-S2), 10) for the barcode checksum digit.

Here is the author's suggested function code translated from T-SQL in standard SQL/PSM:

```
CREATE FUNCTION Barcode_CheckSum(IN my_barcode CHAR(12))
RETURNS INTEGER
 BEGIN
 DECLARE barcode_checkres INTEGER;
 DECLARE idx INTEGER;
 DECLARE sgn INTEGER;
 SET barcode_checkres = 0;
-- check if given barcode is numeric
 IF IsNumeric(my_barcode) = 0
 THEN RETURN -1;
 END IF;
-- check barcode length
 IF CHAR_LENGTH(TRIM(BOTH ' ' FROM my_barcode))<> 12
 THEN RETURN -2;
 END IF;
-- compute barcode checksum algorithm
 SET idx = 1;
 WHILE idx <= 12
 DO  -- Calculate sign of digit
  IF MOD(idx, 2) = 0
```

```
  THEN SET sgn = -1;
  ELSE SET sgn = +1;
  END IF;
  SET barcode_checkres = barcode_checkres +
    CAST(SUBSTRING(my_barcode FROM idx FOR 1) AS INTEGER)
         * sgn;
  SET idx = idx + 1;
 END WHILE;

 -- check digit
 RETURN ABS(MOD(barcode_checkres, 10));
END;
```

Let's see how it works:

```
barcode_checkSum('283723281122')
= ABS (MOD(2-8 + 3-7 + 2-3 + 2-8 + 1-1 + 2-2), 10))
= ABS (MOD(-6 -4 -1 -6 + 0 + 0), 10)
= ABS (MOD(-17, 10))
= ABS(-7) = 7
```

Answer #1

Okay, where to begin? Notice the creation of unnecessary local variables, the assumption of an `IsNumeric()` function taken from T-SQL dialect, and the fact that the check digit is supposed to be a character in the barcode and not an integer separated from the barcode. We have three `IF` statements and a `WHILE` loop in the code. This is about as procedural as you can get.

In fairness, SQL/PSM does not handle errors by returning negative numbers, but I don't want to get into a lesson on the mechanism used, which is quite different from the one used in T-SQL.

Why use all that procedural code? Most of it can be replaced by declarative expressions. Let's start with the usual Sequence auxiliary table in place of the loop, nest function calls, and use `CASE` expressions to remove `IF` statements.

The rough pseudoformula for conversion is:

- A procedural loop becomes a sequence set:

```
FOR seq FROM 1 TO n DO f(seq);
    => SELECT f(seq) FROM Sequence WHERE seq <= n;
```

- A procedural selection becomes a `CASE` expression:

```
IF.. THEN .. ELSE
    => CASE WHEN.. THEN .. ELSE.. END;
```

- A series of assignments and function calls become a nested set of function calls:

```
DECLARE x <type>;
SET x = f(..);
SET y = g(x);
 ..;
  => f(g(x))
```

Answer #2

Here is a translation of those guidelines into a first shot at a rewrite:

```
CREATE FUNCTION Barcode_CheckSum(IN my_barcode CHAR(12))
RETURNS INTEGER
BEGIN
 IF barcode NOT SIMILAR TO '%[^0-9]%'
 THEN RETURN -1;
 ELSE RETURN
 (SELECT ABS(SUM((CAST (SUBSTRING(barcode
                            FROM S.seq FOR 1) AS INTEGER)
        * CASE MOD(S.seq)WHEN 0 THEN 1 ELSE -1 END)))
   FROM Sequence AS S
  WHERE S.seq <= 12);
END IF;
END;
```

The `SIMILAR TO` regular expression predicate is a cute trick worth mentioning. It is a double-negative that assures the input string is all digits in all 12 positions. Remember that an oversized string will not fit into the parameter and will give you an overflow error, while a short string will be padded with blanks.

Answer #3

But wait! We can do better:

```
CREATE FUNCTION Barcode_CheckSum(IN my_barcode CHAR(12))
RETURNS INTEGER
RETURN
 (SELECT ABS(SUM((CAST (SUBSTRING(barcode
                            FROM S.seq FOR 1) AS INTEGER)
        * CASE MOD(S.seq)WHEN 0 THEN 1 ELSE -1 END)))
   FROM Sequence AS S
  WHERE S.seq <= 12)
    AND barcode NOT SIMILAR TO '%[^0-9]%';
```

This will return a `NULL` if there is an improper barcode. It is only one SQL statement, so we are doing pretty well. There are some minor tweaks, like this:

```
CREATE FUNCTION Barcode_CheckSum(IN my_barcode CHAR(12))
RETURNS INTEGER
RETURN
 (SELECT ABS(SUM(CAST(SUBSTRING(barcode
        FROM Weights.seq FOR 1) AS INTEGER)
        * Weights.wgt))
   FROM (VALUES (CAST(1 AS INTEGER), CAST(-1 AS INTEGER)),
(2, +1), (3, -1), (4, +1), (5, -1),
        (6, +1), (7, -1), (8, +1), (9, -1), (10, +1), (11,
-1), (12, +1)) AS weights(seq, wgt)
  WHERE barcode NOT SIMILAR TO '%[^0-9]%');
```

Another cute trick in standard SQL is to construct a table constant with a `VALUES()` expression. The first row in the table expression establishes the datatypes of the columns by explicit casting.

Answer #4

What is the best solution? The real answer is none of the above. The point of this exercise was to come up with a *set-oriented, declarative* answer. We have been writing functions to check a condition. What we want is a `CHECK()` constraint for the barcode. Try this instead:

```
CREATE TABLE Products
(..
 barcode CHAR(13) NOT NULL
  CONSTRAINT all_numeric_checkdigit
  CHECK (barcode NOT SIMILAR TO '%[^0-9]%')
  CONSTRAINT valid_checkdigit
  CHECK (
         (SELECT ABS(SUM(CAST(SUBSTRING(barcode
                     FROM Weights.seq FOR 1) AS INTEGER)
         * Weights.wgt))
   FROM (VALUES (CAST(1 AS INTEGER), CAST(-1 AS INTEGER)),
(2, +1), (3, -1), (4, +1), (5, -1),
        (6, +1), (7, -1), (8, +1), (9, -1), (10, +1), (11,
-1), (12, +1)) AS weights(seq, wgt)
 = CAST(SUBSTRING(barcode FROM 13 FOR 1) AS INTEGER)),
..
 );
```

This will keep bad data out of the schema, which is something that a function cannot do. The closest thing you could do would be to have a trigger that fires on insertion. The reason for splitting the code into two constraints is to provide better error messages. That is how we think in SQL.

PUZZLE
61 SORT A STRING

Tony Wilton posted this quick problem in 2003. We are currently writing a stored procedure that generates a string of characters in the form 'CABBDBC'. We need to be able to sort this string alphabetically, that is, "ABBBCCD" as the results. There are no library functions to do this.

Let's make the assumption that when Tony said "in the form," he means the string is always CHAR(7) and always made up of elements of the four letters {'A', 'B', 'C', 'D'}.

Answer #1

The first answers proposed by people were for a procedure with a WHILE loop written in a proprietary 4GL. They used bubble sorts on SUBSTRING (gen_str FROM i FOR 1) within the string, so that you would have to call on the procedure one row at a time.

If the string is a fixed length, then you can use a Bose-Nelson solution that will be insanely fast. This sort generates the swap pairs that you need to consider to order the string. I am not going to go into the details since it is a bit more math than we want to look at right now, but you can find them in my book *SQL for Smarties*. The procedure would be a chain of UPDATE statements.

Answer #2

If there is a relatively small set of generated string, use a look-up table.

```
CREATE TABLE SortMeFast
(unsorted_string CHAR(7) NOT NULL PRIMARY KEY,
 sorted_string CHAR(7) NOT NULL);
```

Fike's algorithm can give you the permutations to load into the table, and this will let you process an entire set of unsorted strings at once, instead of calling a function on one row at a time.

Answer #3

If the set of characters is small, count the A's, generate a string of them; count the B's, generate a string of them; and concatenate it to the first string. Keep going for the rest of the alphabet.

Let us assume that we have a REPLICATE(<string expression>, <n>) function that will create a string of (n) copies of the <string expression>. Also assume a REPLACE(<target string>,<old string>,<new string>) that will replace the <old string> with the <new string> wherever it appears in the <target string>.

```
BEGIN
DECLARE instring CHAR(7);
SET instring = 'DCCBABA';

 REPLICATE ('A', (DATA_LENGTH(instring)-
DATA_LENGTH(REPLACE(instring,'A',''))))
 || REPLICATE ('B', (DATA_LENGTH(instring)-
DATA_LENGTH(REPLACE(instring,'B',''))))
 || REPLICATE ('C', (DATA_LENGTH(instring)-
DATA_LENGTH(REPLACE(instring,'C',''))))
 || REPLICATE ('D', (DATA_LENGTH(instring)-
DATA_LENGTH(REPLACE(instring,'D',''))))

END;
```

You can do this for the whole alphabet and it will fly. What you do is change the current search letter to an empty string, and then compare the length of the reduced string to the original to get the number of occurrences of that letter. That count is used by REPLICATE() to build the output string.

This expression can also be put into a VIEW, so there is absolutely no procedural code in the schema. SQL programmers too often come from a procedural programming background and cannot think this way. When I showed this to a LISP programmer, however, his response was "Of course, how else?"

PUZZLE
62 REPORT FORMATTING

SQL is a data retrieval language and not a report writer. Unfortunately, people do not seem to know this and are always trying to do things for which SQL was not intended. One of the most common ones is to arrange a list of values into a particular number of columns for display.

The original version of this puzzle came from Richard S. Romley at Smith Barney, who many of you know as the man who routinely cooks my SQL puzzle solutions. He won a bet from a coworker who said it could not be done. The problem is to first create a simple one-column table "Names" with a single unique "name" column and populate it as follows:

```
CREATE TABLE Names
(name VARCHAR(15) NOT NULL PRIMARY KEY);

INSERT INTO Names
VALUES ('Al'), ('Ben'), ('Charlie'),
       ('David'), 'Ed'), ('Frank'),
       ('Greg'), ('Howard'), ('Ida'),
       ('Joe'), ('Ken'), ('Larry'),
       ('Mike');
```

A simple "`SELECT name FROM Names ORDER BY name;`" returns the original list in alphabetic order, but suppose you wanted to display the names three across, like this:

```
Results
name1   name2   name3
=======================
Al      Ben     Charlie
David   Ed      Frank
Greg    Howard  Ida
Joe     Ken     Larry
Mike    NULL    NULL
```

or four across:

```
Results
name1   name2   name3   name4
==============================
Al      Ben     Charlie David
Ed      Frank   Greg    Howard
Ida     Joe     Ken     Larry
Mike    NULL    NULL    NULL
```

or any other number across? Can you write single SQL statements that will generate each of these results?

Answer #1

The best approach is to start with a simple two-across solution and explain it:

```
SELECT N1.name AS name1, MIN(N2.name) AS name2
  FROM Names AS N1
       LEFT OUTER JOIN
       Names AS N2
       ON N1.name < N2.name
 WHERE N1.name
       IN (SELECT A.name
             FROM Names AS A
                  INNER JOIN
                  Names AS B
                  ON A.name <= B.name
            GROUP BY A.name
           HAVING MOD(COUNT(B.name), 2) =
                (SELECT MOD(COUNT(*), 2) FROM Names))
 GROUP BY N1.name
 ORDER BY N1.name;
```

The self `OUTER JOIN` will put the lower alphabetical ranked name in the first column. The `MIN()` aggregate function will then pick the lowest remaining name from the table, excluding `N1.name`.

The `WHERE` clause is the real trick. We want to find the values of `N1.name` that will start each row in the desired result table and use that list of names to define the result set. In this case, that would be the first name (`'Al'`), third name (`'Charlie'`), and so on, in the alphabetized list. This is all done with a `MOD()` function. The `MOD()` function was not

part of the official SQL-92, so technically we should have been writing this out with integer arithmetic. But it is such a common vendor extension, and it does show up in the SQL-99 standard, that I do not mind using it.

Start with an experimental table that looks like this:

```
 SELECT A.name
   FROM Names AS A
        INNER JOIN
        Names AS B
        ON A.name <= B.name
  GROUP BY A.name;
```

Using four names, the ungrouped table would look like this:

```
A.name   B.name
 ==================
 Al       Al
 Al       Ben
 Al       Charlie
 Al       David
--------------------
 Ben      Ben
 Ben      Charlie
 Ben      David
--------------------
 Charlie Charlie
 Charlie David
--------------------
 David    David
```

The predicate `(MOD(COUNT(A.name), 2) = 0` will find what we want. This is fine for even numbers, but if we have an odd number of people (insert 'Ed' into the example), we need to get that "orphaned" row into the result table. You can do this by knowing the total number of rows in the original table and using it to adjust the selection of the first column in the final result table. I am skipping some algebra, but you can work it out easily.

Instead of doing the cases for three and four across, let's jump directly to five across to show how the solution generalizes:

```
SELECT N1.name,
       MIN(N2.name) AS name2,
       MIN(N3.name) AS name3,
       MIN(N4.name) AS name4,
       MIN(N5.name) AS name5
  FROM (Names AS N1
        LEFT OUTER JOIN
        Names AS N2
       ON N1.name < N2.name)
        LEFT OUTER JOIN
        Names AS N3
        ON N1.name < N2.name
           AND N2.name < N3.name
        LEFT OUTER JOIN
        Names AS N4
        ON N1.name < N2.name
          AND N2.name < N3.name
          AND N3.name < N4.name
        LEFT OUTER JOIN
        Names AS N5
        ON N1.name < N2.name
           AND N2.name < N3.name
           AND N3.name < N4.name
           AND N4.name < N5.name
 WHERE N1.name IN (SELECT A.name
                     FROM Names AS A
                          INNER JOIN
                          Names AS B
                          ON A.name <= B.name
                   GROUP BY A.name
                  HAVING MOD(COUNT(B.name), 5) =
                        (SELECT MOD(COUNT(*), 5)
                            FROM Names))
 GROUP BY N1.name;
```

Answer #2

Another shorter version of the above query is as follows:

```
 SELECT N3.name, MIN(N4.name), MIN(N5.name), MIN(N6.name),
MIN(N7.name)
   FROM (SELECT N1.name
```

```
            FROM Names AS N1
                 INNER JOIN
                 Names AS N2
                 ON N1.name >= N2.name
             GROUP BY N1.name
            HAVING MOD (COUNT(*), 5) = 1) AS N3(name)
          LEFT OUTER JOIN
          Names AS N4
          ON N3.name < N4.name
             LEFT OUTER JOIN
             Names AS N5
             ON N4.name < N5.name
                 LEFT OUTER JOIN
                 Names AS N6
                 ON N5.name < N6.name
                    LEFT OUTER JOIN
                    Names AS N7
                    ON N6.name < N7.name
  GROUP BY N3.name;
```

Answer #3

Nayan Raval sent an e-mail to me after this puzzle appeared in the February 1997 issue of *DBMS*. He started to think about the repeated use of `(N1.name < N2.name)` in the `LEFT OUTER JOIN` clauses of Answer #1 and realized that only the first one is required. That is, the following produced the same output on my system:

```
-- same code up to ...
FROM (Names AS N1
      LEFT OUTER JOIN
      Names AS N2
      ON N1.name < N2.name)
         LEFT OUTER JOIN
         Names AS N3
         ON N2.name < N3.name
            LEFT OUTER JOIN
            Names AS N4
            ON N3.name < N4.name
               LEFT OUTER JOIN
               Names AS N5
```

```
                ON N4.name < N5.name
-- rest of the code is the same
```

Answer #4

Along the same lines, Dautbegovic Dzavid of Bosnia-Herzegovina observed that

```
...
IN (SELECT N1.name
      FROM Names AS N1.
           INNER JOIN
           Names AS N2
           ON N1.name <= N2.name
      GROUP BY N1.name
     HAVING MOD(COUNT(N2.name), 2)
            = (SELECT MOD(COUNT(*), 2) FROM Names)) ...
```

could better be replaced by:

```
...
IN (SELECT N1.name
      FROM Names AS N1
           INNER JOIN
           Names AS N2
           ON N1.name >= N2.name
     GROUP BY N1.name
   HAVING MOD(COUNT(*), 2) = 1)
...
```

Answer #5

Dmitry Sizintsev came up with an alternative solution. Here is his solution for `N = 5`; it's very different from the one given by Richard Romley, and avoids using a five-way self-`JOIN`.

```
SELECT MAX(name1), MAX(name2), MAX(name3), MAX(name4),
MAX(name5)
  FROM ( -- start of monster table query
       (SELECT (COUNT(*) - 1) / 5),
       (SELECT MAX(N1.name)
          FROM Names AS N3
```

```
          WHERE N1.name <= N3.name
         HAVING MOD(COUNT(*), 5)
                = (SELECT MOD(COUNT(*), 5)
                     FROM Names)),
        (SELECT MAX(N1.name)
           FROM Names AS N3
          WHERE N1.name <= N3.name
         HAVING MOD (COUNT(*), 5)
                = (SELECT MOD((COUNT(*) - 1), 5)
                     FROM Names)),
        (SELECT MAX(N1.name)
           FROM Names AS N3
          WHERE N1.name <= N3.name
         HAVING MOD(COUNT(*), 5)
                = (SELECT MOD((COUNT(*) - 2), 5)
                     FROM Names)),
        (SELECT MAX(N1.name)
           FROM Names AS N3
          WHERE N1.name <= N3.name
         HAVING MOD(COUNT(*), 5)
                = (SELECT MOD((COUNT(*) - 3), 5)
                     FROM Names)),
        (SELECT MAX(N1.name)
           FROM Names AS N3
          WHERE N1.name <= N3.name
         HAVING MOD(COUNT(*), 5)
                = (SELECT MOD((COUNT(*) - 4), 5)
                     FROM Names))
  FROM Names AS N1
       INNER JOIN
       Names AS N2
       ON N1.name >= N2.name
 GROUP BY N1.name)
 AS XO(cnt, name1, name2, name3, name4, name5)
 GROUP BY cnt;
```

Answer #6

I e-mailed these answers to Richard Romley on March 12, 2000, and he immediately had a cook on the puzzle that he had not shared:

```
--For 3 columns...

SELECT FirstCol.name AS name1,
       MAX(CASE WHEN OtherCols.cnt = 2
                THEN OtherCols.final_name
                ELSE NULL END) AS name2,
       MAX(CASE WHEN OtherCols.cnt = 3
                THEN OtherCols.final_name
                ELSE NULL END) AS name3
  FROM (SELECT N1.name
          FROM Names AS N1, Names AS N2
         WHERE N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD(COUNT(*), 3) = 1) AS FirstCol(name)
       LEFT OUTER JOIN
       (SELECT N3.name, N5.name, COUNT(*)
          FROM Names AS N3, Names AS N4, Names AS N5
         WHERE N3.name < N5.name
           AND N4.name BETWEEN N3.name AND N5.name
         GROUP BY N3.name, N5.name) AS OtherCols(name,
final_name, cnt)
       ON FirstCol.name = OtherCols.name
 GROUP BY FirstCol.name
 ORDER BY FirstCol.name;

For 5 columns...

SELECT FirstCol.name AS name1,
       MAX(CASE WHEN OtherCols.cnt = 2
                THEN OtherCols.final_name
                ELSE NULL END) AS name2,
       MAX(CASE WHEN OtherCols.cnt = 3
                THEN OtherCols.final_name
                ELSE NULL END) AS name3,
       MAX(CASE WHEN OtherCols.cnt = 4
                THEN OtherCols.final_name
                ELSE NULL END) AS name4,
       MAX(CASE WHEN OtherCols.cnt = 5
                THEN OtherCols.final_name
                ELSE NULL END) AS name5
  FROM (SELECT N1.name
          FROM Names AS N1, Names AS N2
```

```
         WHERE N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD(COUNT(*), 5) = 1) AS FirstCol(name)
       LEFT OUTER JOIN
       (SELECT N3.name, N5.name, COUNT(*)
          FROM Names AS N3, Names AS N4, Names AS N5
         WHERE N3.name < N5.name
           AND N4.name BETWEEN N3.name AND N5.name
         GROUP BY N3.name, N5.name) AS OtherCols(name,
final_name, cnt)
       ON FirstCol.name = OtherCols.name
 GROUP BY FirstCol.name
 ORDER BY FirstCol.name;

For 6 columns...

SELECT FirstCol.name AS name1,
       MAX(CASE WHEN OtherCols.cnt = 2
                THEN OtherCols.final_name
                ELSE NULL END) AS name2,
       MAX(CASE WHEN OtherCols.cnt = 3
                THEN OtherCols.final_name
                ELSE NULL END) AS name3,
       MAX(CASE WHEN OtherCols.cnt = 4
                THEN OtherCols.final_name
                ELSE NULL END) AS name4,
       MAX(CASE WHEN OtherCols.cnt = 5
                THEN OtherCols.final_name
                ELSE NULL END) AS name5,
       MAX(CASE WHEN OtherCols.cnt = 6
                THEN OtherCols.final_name
                ELSE NULL END) AS name6
  FROM (SELECT N1.name
          FROM Names AS N1, Names AS N2
         WHERE N1.name >= N2.name
         GROUP BY N1.name
        HAVING MOD COUNT(*), 6) = 1) AS FirstCol
       LEFT OUTER JOIN
       (SELECT N3.name, N5.name AS final_name, COUNT(*) AS
cnt
          FROM Names AS N3, Names AS N4, Names AS N5
         WHERE N3.name < N5.name
```

```
          AND N4.name BETWEEN N3.name AND N5.name
        GROUP BY N3.name, N5.name) AS OtherCols
    ON FirstCol.name = OtherCols.name
GROUP BY FirstCol.name
ORDER BY FirstCol.name;
```

Each additional column requires only adding the additional column in the `SELECT` list and changing the modulus in the `MOD()` function. The rest of the query remains the same.

PUZZLE
63 CONTIGUOUS GROUPINGS

Donald Halloran proposed this simple-looking problem:

```
CREATE TABLE T
(num INTEGER NOT NULL PRIMARY KEY,
 data CHAR(1) NOT NULL);

INSERT INTO T
VALUES (1, 'a'),
       (2, 'a'),
       (3, 'b'),
       (6, 'b'),
       (8, 'a');
```

The aim is to group the results into neighboring ranges with the start and end of the range, and the data in the range in this example becomes:

```
low  high  data
====================
1      2    'a'
3      6    'b'
8      8    'a'
```

His solution is as follows, but he had a feeling that something was redundant in what he wrote, given that the algorithm seems to be two steps but the SQL looks like three:

1. For each row (r), find the first row (R) where r.num > r.num and r.data <> r.data.
2. group (r) by (R).

Answer #1

Here is the first attempt:

```
SELECT MIN(T1.num) AS low,
       MAX(T1.num) AS high,
       T1.data
```

```
  FROM T AS T1
       LEFT OUTER JOIN
       T AS T2
       ON T2.num
           = (SELECT MIN(num)
                FROM T
               WHERE num > T1.num
                 AND data <> T1.data)
GROUP BY T1.data, T2.num;
```

Answer #2

I came up with another version that uses the `ALL()` predicate to check on the contents of the range of low and high numbers.

```
SELECT X.data, MIN(X.low) AS low, X.high
  FROM (SELECT T1.data, T1.num, MAX(T2.num)
          FROM T AS T1, T AS T2
         WHERE T1.num <= T2.num
           AND T1.data
               = ALL(SELECT T3.data
                       FROM T AS T3
                      WHERE T3.num BETWEEN T1.num
                                       AND T2.num)
         GROUP BY T1.data, T1.num)
       AS X(data, low, high)
GROUP BY X.data, X.high;
```

Just offhand, I think this is not as good as the original version.

Answer #3

Steve Kass proposed this answer, but did not know whether it's faster, but this is another approach (a clustered index on (`num`, `data`) helps).

```
SELECT MIN(num) AS low, MAX(num) AS high, data
  FROM (SELECT A.num,
               SUM(CASE WHEN A.data = B.data THEN 1 ELSE 0
END)
                   - COUNT(B.num) AS ct,
               A.data
          FROM T AS A, T AS B
```

```
         WHERE A.num >= B.num
         GROUP BY A.num, A.data
        ) AS A (num, ct, data)
GROUP BY data, ct;
```

He is using a little math to determine that a range has only one data value in it.

PUZZLE
64 BOXES

This is an interesting puzzle from Mikito Harakiri. Imagine that you have a Cartesian space, and you are filling it with n-dimensional boxes. The boxes are modeled as shown below:

```
CREATE TABLE Boxes
(box_id INTEGER NOT NULL PRIMARY KEY,
 dimension_nbr INTEGER NOT NULL,
 low INTEGER NOT NULL,
 high INTEGER NOT NULL);
```

The problem is to find all the pairs of boxes that intersect (e.g., the 3D cubes):

```
A = {(x,0,2),(y,0,2),(z,0,2)}
B = {(x,1,3),(y,1,3),(z,1,3)}
C = {(x,10,12),(y,0,4),(z,0,100)}
```

Boxes A and B intersect, but box C intersects neither A nor B. Bonus point: is there anything special about this kind of query?

Answer #1

This is from Bob Badour:

```
SELECT B1.box_id AS box1, B2.box_id AS box2
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND NOT EXISTS
       (SELECT *
          FROM Boxes AS B3, Boxes AS B4
         WHERE B3.box_id = B1.box_id
           AND B4.box_id = B2.box_id
           AND B4.dimension_nbr = B3.dimension_nbr
           AND (B4.high < B3.low OR B4.low > B3.high)
       )
GROUP BY B1.box_id, B2.box_id;
```

Mikito Harakiri thought that this was interesting because it is a generalization of relational division. Now, relational division expressed in SQL is either a query with two levels of nested subqueries, or one level of nested subqueries with `EXCEPT` operators, or the query with counting subquery in the `HAVING` clause. Bob's query is none of those.

```
SELECT B1.box_id AS box1, B2.box_id AS box2, B1.dim
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.low BETWEEN B2.low AND B2.high
   AND B1.dim = B2.dim
 GROUP BY B1.box_id, B2.box_id
HAVING COUNT(*)
  = (SELECT COUNT(*)
       FROM Boxes AS B3
      WHERE B3.box_id = B1.box_id
        AND B3.dim = B1.dim);
```

Answer #2

Try a slightly different approach. Begin with one dimension and stronger DDL:

```
CREATE TABLE Boxes
(box_id CHAR (1) NOT NULL,
 dim CHAR(1) NOT NULL,
 PRIMARY KEY (box_id, dim),
 low INTEGER NOT NULL,
 high INTEGER NOT NULL,
 CHECK (low < high));

INSERT INTO Boxes VALUES ('A', 'x', 0, 2);
INSERT INTO Boxes VALUES ('B', 'x', 1, 3);
INSERT INTO Boxes VALUES ('C', 'x', 10, 12);

--in 1 dimension
SELECT B1.box_id, B2.box_id
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND (B1.high - B1.low) + (B2.high - B2.low)
> ABS(B1.high - B2.low);
```

This says that two lines segments overlap when their combined lengths are less than their span in the dimension. I am using math rather than BETWEEN-ness.

Cubes A={(x,0,2),(y,0,2),(z,0,2)} and B={(x,1,3),(y,1,3),(z,1,3)} intersect, while box C={(x,10,12),(y,0,4),(z,0,100)}. Let's go to two dimensions:

```
INSERT INTO Boxes VALUES ('A', 'y', 0, 2);
INSERT INTO Boxes VALUES ('B', 'y', 1, 3);
INSERT INTO Boxes VALUES ('C', 'y', 0, 4);

--in 2 dimension: first shot:
SELECT B1.box_id, B2.box_id, B1.dim
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND B1.dim = B2.dim
   AND (B1.high - B1.low) + (B2.high - B2.low)
ABS(B1.high - B2.low);
```

Now look for a common area in (x,y) by having overlaps in both dimensions:

```
SELECT B1.box_id, B2.box_id
  FROM Boxes AS B1, Boxes AS B2
 WHERE B1.box_id < B2.box_id
   AND B1.dim = B2.dim
   AND (B1.high - B1.low) + (B2.high - B2.low)
       > ABS(B1.high - B2.low)
 GROUP BY B1.box_id, B2.box_id
 HAVING COUNT(B1.dim) = 2;

--3 dimensions:
INSERT INTO Boxes VALUES ('A', 'z', 0, 2);
INSERT INTO Boxes VALUES ('B', 'z', 1, 3);
INSERT INTO Boxes VALUES ('C', 'z', 0, 100);
```

Now change the HAVING clause to

```
COUNT(B1.dim) = 3
```

or

```
(SELECT COUNT (DISTINCT dim) FROM Boxes)
```

if you do not know the dimension of the space you are using.

PUZZLE
65 AGE RANGES FOR PRODUCTS

David Poole posted what he called a simple problem with which he was struggling. Given an inventory of products, he has prices broken down by age ranges.

```
CREATE TABLE PriceByAge
(product_id INTEGER NOT NULL,
 low_age INTEGER NOT NULL,
 high_age INTEGER NOT NULL,
 CHECK (low_age < high_age),
 product_price DECIMAL (12,4) NOT NULL,
PRIMARY KEY (product_id, low_age));

INSERT INTO PriceByAge ('Product1', 5, 15, 20.00);
INSERT INTO PriceByAge ('Product1', 16, 60, 18.00);
INSERT INTO PriceByAge ('Product1', 65, 150, 17.00);
INSERT INTO PriceByAge ('Product2', 1, 5, 20.00);
 ...etc
```

You are also given a table containing various persons' ages. This table should be a `VIEW`, based on birthdates, but let me be sloppy.

```
CREATE TABLE Buyers
(person_name VARCHAR(20) NOT NULL PRIMARY KEY,
 age INTEGER NOT NULL CHECK (age > 0));
```

What he wanted was to bring back all products that could satisfy all ages.

In the sample data, if I had an age of 4 in the list of ages, then no rows for products that did not have an age band that included 4 were to be returned.

Answer #1

Define "all ages." I will guess you mean a range of 1 to (`max_age`), say 150, since Elizabeth Israel, also known as Ma Pampo, was born on January 27, 1875 and is believed to be the world's oldest living human

being, so we should be safe. A quick way is to use your auxiliary Sequence table.

```
SELECT P.product_id
  FROM PriceByAge AS P, Sequence AS S
 WHERE S.seq BETWEEN P.low_age AND P.high_age
   AND S.seq <= 150
 GROUP BY P.product_id
HAVING COUNT(seq) = 150;
```

PUZZLE
66 SUDOKU

I thought that Sudoku, the current puzzle fad, would be a good SQL programming problem. You start with a 9×9 grid that is further divided into nine 3×3 regions. Some of the cells will have a digit from 1 to 9 in them at the start of the puzzle. Your goal is to fill in all the cells with more digits such that each row, column, and region contains one and only one instance of each digit.

Strangely, the puzzle appeared in the United States in 1979, then caught on in Japan in 1986 and became an international fad in 2005. Most newspapers today carry a daily Sudoku.

How can we do this in SQL? Well we can start by modeling the grid as an `(i, j)` array with a value in the cell. The first attempt usually does not have the region information as one of the columns. The regions do not have names in the puzzle, so we need a way to give them names.

```
CREATE TABLE SudokuGrid
(i INTEGER NOT NULL
  CHECK (i BETWEEN 1 AND 9),
 j INTEGER NOT NULL
  CHECK (j BETWEEN 1 AND 9),
 val INTEGER NOT NULL
  CHECK (val BETWEEN 1 AND 9),
 region_nbr INTEGER NOT NULL,
 PRIMARY KEY (i, j, val));
```

Now we need to fill it. Each `(i, j)` cell needs to start with all nine digits, so we build a table of the digits 1 to 9 and do `CROSS JOIN`s. But how do we get a region number?

An obvious name would be the position of the region by `(x, y)` coordinates, where `x` = {1, 2, 3} and `y` = {1, 2, 3}. We can then make them into one number by making `x` the tens place and `y` the units place, so we get {11,12, 13, 21, 22, 23, 31, 32, 33} for the regions. The math for this depends on integer arithmetic, but it is not really hard.

```
INSERT INTO SudokuGrid (i, j, val, region_nbr)
SELECT D1.d, D2.d, D3.d,
       10*((D1.d+2)/3) + ((D2.d+2)/3) AS region_nbr
  FROM Digits AS D1
       CROSS JOIN Digits AS D2
```

```
       CROSS JOIN Digits AS D3;
```

We will need a procedure to insert the known values and clear out that value in the rows, columns, and regions. As we remove more and more values, we hope to get a table with 81 cells that is the unique solution for the puzzle.

Answer #1

The first attempt is usually to write three delete statements, one for rows, one for columns, and one for the region.

```
BEGIN
DELETE FROM SudokuGrid -- rows
WHERE :my_i = i
 AND :my_j <> j
 AND :my_val = val;

DELETE FROM SudokuGrid -- columns
WHERE :my_i <> i
 AND :my_j = j
 AND :my_val = val;

DELETE FROM SudokuGrid -- region
WHERE i <> :my_i
 AND j <> :my_j
 AND region_nbr = 10*((:my_i+2)/3) + ((:my_j+2)/3)
 AND :my_val = val);
END;
```

Answer #2

But this is a waste of execution time. Why use three statements when you can write it in one? Let's do a brute force code merge:

```
DELETE FROM SudokuGrid
 WHERE (((:my_i = i AND j <> :my_j)
        OR (:my_i <> i AND j = :my_j))
     AND :my_val = val)
   OR (i <> :my_i
       AND j <> :my_j
```

```
      AND region_nbr = 10*((:my_i+2)/3) + ((:my_j+2)/3)
      AND :my_val = val);
```

Those nested ORs are ugly! The expression (:my_val = val) appears twice. Get a drink, step back, and consider that the (i, j) pairs can relate to our input in one of four mutually exclusive ways, which require that we remove a value from a cell or leave it. That implies a CASE expression instead of the nested ANDs and ORs.

```
DELETE FROM SudokuGrid
 WHERE CASE WHEN :my_i = i AND :my_j = j -- my cell
            THEN 'Keep'
            WHEN :my_i = i AND :my_j <> j -- row
            THEN 'Delete'
            WHEN :my_i <> i AND :my_j = j -- column
            THEN 'Delete'
            WHEN i <> :my_i AND j <> :my_j -- square
                AND region_nbr
                     = 10*(:my_i+2)/3) + (:my_j+2)/3)
            THEN 'Delete'
            ELSE NULL END = 'Delete'
      AND :my_val = val);
```

Answer #3

Test it and find out that this is wrong!! We need to pay special attention to the cell where we know the value; that means two cases.

```
DELETE FROM SudokuGrid
 WHERE CASE WHEN :my_i = i AND :my_j = j AND my_val = val
            THEN 'Keep'
            WHEN :my_i = i AND :my_j = j AND my_val <> val
            THEN 'Delete'
            WHEN :my_i = i AND :my_j <> j -- row
            THEN 'Delete'
            WHEN :my_i <> i AND :my_j = j -- column
            THEN 'Delete'
            WHEN i <> :my_i AND j <> :my_j -- square
                AND region_nbr
                     = 10*(:my_i+2)/3) + (:my_j+2)/3)
            THEN 'Delete'
```

```
        ELSE NULL END = 'Delete'
AND :my_val = val);
```

The next improvement might be to put the known cells into their own table so we have a history of the puzzle. But let's leave that as a problem for the reader.

PUZZLE
67 STABLE MARRIAGES PROBLEM

This is a classic programming problem from procedural language classes. The setup is fairly simple; you have a set of potential husbands and an equally sized set of potential wives. We want to pair them up into stable marriages.

What is a stable marriage? In 25 words or less, a marriage in which neither partner can do better. You have a set of *n* men and a set of *n* women. All the men have a preference scale for all the women that ranks them from 1 to *n* without gaps or ties. The women have the same ranking system for the men.

The goal is to pair off the men and women into *n* marriages such that there is no pair in your final arrangement where Mr. X and Ms. Y are matched to each other when they both would rather be matched to someone else.

For example, let's assume the husbands are ("Joe Celko," "Brad Pitt") and the wives are ("Jackie Celko," "Angelina Jolie"). If Jackers got matched to Mr. Pitt, she would be quite happy. And I would enjoy Ms. Jolie's company. However, Mr. Pitt and Ms. Jolie can both do better than us. Once they are paired up they will stay that way, leaving Jackers and I still wed.

The classic Stable Marriage algorithms usually are based on backtracking. These algorithms try a combination of couples, and then attempt to fix any unhappy matches. When the algorithm hits on a situation where nobody can improve their situation, they stop and give an answer.

Two important things to know about this problem: (1) there is always a solution, and (2) there is often more than one solution. Doing this in SQL is really hard because SQL does not backtrack.

Remember that a stable marriage is not always a happy marriage. In fact, in this problem, while there is always at least one arrangement of stable marriages in any set, you most often find many different pairings that produce a set of stable marriages. Each set of marriages will tend to maximize either the happiness of the men or the women.

Let's show a small example in SQL with four couples:

```
CREATE TABLE Husbands
(man CHAR(5) NOT NULL,
 woman CHAR(5) NOT NULL,
 ranking INTEGER NOT NULL CHECK (ranking > 0),
```

```
  PRIMARY KEY (man, woman));

INSERT INTO Husbands VALUES ('Abe', 'Joan', 1);
INSERT INTO Husbands VALUES ('Abe', 'Kathy', 2);
INSERT INTO Husbands VALUES ('Abe', 'Lynn', 3);
INSERT INTO Husbands VALUES ('Abe', 'Molly', 4);

INSERT INTO Husbands VALUES ('Bob', 'Joan', 3);
INSERT INTO Husbands VALUES ('Bob', 'Kathy', 4);
INSERT INTO Husbands VALUES ('Bob', 'Lynn', 2);
INSERT INTO Husbands VALUES ('Bob', 'Molly', 1);

INSERT INTO Husbands VALUES ('Chuck', 'Joan', 3);
INSERT INTO Husbands VALUES ('Chuck', 'Kathy', 4);
INSERT INTO Husbands VALUES ('Chuck', 'Lynn', 2);
INSERT INTO Husbands VALUES ('Chuck', 'Molly', 1);

INSERT INTO Husbands VALUES ('Dave', 'Joan', 2);
INSERT INTO Husbands VALUES ('Dave', 'Kathy', 1);
INSERT INTO Husbands VALUES ('Dave', 'Lynn', 3);
INSERT INTO Husbands VALUES ('Dave', 'Molly', 4);

CREATE TABLE Wives
(woman CHAR(5) NOT NULL,
 man CHAR(5) NOT NULL,
 ranking INTEGER NOT NULL CHECK (ranking > 0),
  PRIMARY KEY (man, woman));

INSERT INTO Wives VALUES ('Joan', 'Abe', 1);
INSERT INTO Wives VALUES ('Joan', 'Bob', 3);
INSERT INTO Wives VALUES ('Joan', 'Chuck', 2);
INSERT INTO Wives VALUES ('Joan', 'Dave', 4);
INSERT INTO Wives VALUES ('Kathy', 'Abe', 4);
INSERT INTO Wives VALUES ('Kathy', 'Bob', 2);
INSERT INTO Wives VALUES ('Kathy', 'Chuck', 3);
INSERT INTO Wives VALUES ('Kathy', 'Dave', 1);

INSERT INTO Wives VALUES ('Lynn', 'Abe', 1);
INSERT INTO Wives VALUES ('Lynn', 'Bob', 3);
INSERT INTO Wives VALUES ('Lynn', 'Chuck', 4);
INSERT INTO Wives VALUES ('Lynn', 'Dave', 2);

INSERT INTO Wives VALUES ('Molly', 'Abe', 3);
```

```
INSERT INTO Wives VALUES ('Molly', 'Bob', 4);
INSERT INTO Wives VALUES ('Molly', 'Chuck', 1);
INSERT INTO Wives VALUES ('Molly', 'Dave', 2);
```

The pairing of:

```
('Abe', 'Lynn')
('Bob', 'Joan')
('Chuck', 'Molly')
('Dave', 'Kathy')
```

does not work. There is a "blocking pair" in (`'Abe', 'Joan'`). Abe is Joan's first choice and he is her first choice, as shown by the rows:

```
Wives ('Joan', 'Abe', 1)
Husbands ('Abe', 'Joan', 1)
```

but they are matched to others. A simple swap will give us a stable situation:

```
('Abe', 'Joan')
('Bob', 'Lynn')
('Chuck', 'Molly')
('Dave', 'Kathy')
```

If you use a backtracking algorithm, you do not have to generate all possible marriage sets. Once you found a blocking pair, you would never have to create it again. This is considerably faster than the combinatory explosion that SQL must generate and filter. The only advantage with SQL—and it is weak—is that the algorithms for this problem will usually stop at the first success. They do not generate the full solution set, as SQL does.

This answer is from Richard Romley. Simply explained, it generates all possible marriages and filters out the failures. But there are some neat little optimizing tricks in the code.

```
DROP TABLE Wife_perms;
CREATE TABLE Wife_perms
(wife CHAR(5) NOT NULL PRIMARY KEY,
tally INTEGER NOT NULL);

INSERT INTO Wife_perms VALUES ('Joan', 1);
```

```
INSERT INTO Wife_perms VALUES ('Kathy', 2);
INSERT INTO Wife_perms VALUES ('Lynn', 4);
INSERT INTO Wife_perms VALUES ('Molly', 8);
```

Answer #1

The query for finding stable marriages is:

```
SELECT W1.wife AS abe_wife, W2.wife AS bob_wife,
W3.wife AS check_wife, W4.wife AS dave_wife
FROM Wife_perms AS W1, Wife_perms AS W2,
Wife_perms AS W3, Wife_perms AS W4
WHERE (W1.tally + W2.tally + W3.tally + W4.tally) = 15
AND NOT EXISTS
(SELECT *>
FROM Husbands AS H1, Husbands AS H2,
Wives AS W1, Wives AS W2
WHERE H1.man = H2.man
AND H1.ranking > H2.ranking
AND (H1.man || H1.woman) IN
('Abe' || W1.wife, 'Bob' || W2.wife,
'Chuck' || W3.wife, 'Dave' || W4.wife)
AND H2.woman = W1.woman
AND W1.woman = W2.woman
AND W1.ranking > W2.ranking
AND (W1.man || W1.woman) IN
('Abe' || W1.wife, 'Bob' || W2.wife,
'Chuck' || W3.wife, 'Dave' || W4.wife));
```

The first predicate generates all the permutations of wives in the husband columns and the `EXISTS()` checks for "blocking pairs" in the row. This query will take some time to run, especially on a small machine, and it will break down when you have a value of *n* too large for the permutation trick.

The other optimization trick is the list of concatenated strings to see that the blocking pair is in the row that was just constructed. A shorter version of this trick is replacing

```
SELECT *
  FROM Foo as F1, Bar as B1
 WHERE F1.city = B1.city
   AND F1.state = B1.state;
```

with

```
SELECT *
  FROM Foo as F1, Bar as B1
 WHERE F1.city || F1.state = B1.city || B1.state;
```

to speed up a query. I will rationalize that the concatenated name is atomic because it has meaning in itself that would be destroyed if it is split apart. Things like (longitude, latitude) pairs are also atomic in this sense.

There were only 4! (= 24) possible marriage collections, so this ran pretty fast, even on a small machine. Now extend the problem to a set of couples where (n = 8); you now have 8! (= 40,320) possible marriage collections. And only a small number of the rows will be in the final answer set.

Answer #2

Here is the code for the Stable Marriages problem with n = 8:

```
CREATE TABLE Husbands
(man CHAR(2) NOT NULL,
woman CHAR(2) NOT NULL,
ranking INTEGER NOT NULL CHECK (ranking > 0),
PRIMARY KEY (man, woman));

CREATE TABLE Wives
(woman CHAR(2) NOT NULL,
man CHAR(2) NOT NULL,
ranking INTEGER NOT NULL CHECK (ranking > 0),
PRIMARY KEY (woman, man));

INSERT INTO Husbands VALUES ('h1', 'w1', 5);
INSERT INTO Husbands VALUES ('h1', 'w2', 2);
INSERT INTO Husbands VALUES ('h1', 'w3', 6);
INSERT INTO Husbands VALUES ('h1', 'w4', 8);
INSERT INTO Husbands VALUES ('h1', 'w5', 4);
INSERT INTO Husbands VALUES ('h1', 'w6', 3);
INSERT INTO Husbands VALUES ('h1', 'w7', 1);
INSERT INTO Husbands VALUES ('h1', 'w8', 7);
```

```
INSERT INTO Husbands VALUES ('h2', 'w1', 6);
INSERT INTO Husbands VALUES ('h2', 'w2', 3);
INSERT INTO Husbands VALUES ('h2', 'w3', 2);
INSERT INTO Husbands VALUES ('h2', 'w4', 1);
INSERT INTO Husbands VALUES ('h2', 'w5', 8);
INSERT INTO Husbands VALUES ('h2', 'w6', 4);
INSERT INTO Husbands VALUES ('h2', 'w7', 7);
INSERT INTO Husbands VALUES ('h2', 'w8', 5);

INSERT INTO Husbands VALUES ('h3', 'w1', 4);
INSERT INTO Husbands VALUES ('h3', 'w2', 2);
INSERT INTO Husbands VALUES ('h3', 'w3', 1);
INSERT INTO Husbands VALUES ('h3', 'w4', 3);
INSERT INTO Husbands VALUES ('h3', 'w5', 6);
INSERT INTO Husbands VALUES ('h3', 'w6', 8);
INSERT INTO Husbands VALUES ('h3', 'w7', 7);
INSERT INTO Husbands VALUES ('h3', 'w8', 5);

INSERT INTO Husbands VALUES ('h4', 'w1', 8);
INSERT INTO Husbands VALUES ('h4', 'w2', 4);
INSERT INTO Husbands VALUES ('h4', 'w3', 1);
INSERT INTO Husbands VALUES ('h4', 'w4', 3);
INSERT INTO Husbands VALUES ('h4', 'w5', 5);
INSERT INTO Husbands VALUES ('h4', 'w6', 6);
INSERT INTO Husbands VALUES ('h4', 'w7', 7);
INSERT INTO Husbands VALUES ('h4', 'w8', 2);

INSERT INTO Husbands VALUES ('h5', 'w1', 6);
INSERT INTO Husbands VALUES ('h5', 'w2', 8);
INSERT INTO Husbands VALUES ('h5', 'w3', 2);
INSERT INTO Husbands VALUES ('h5', 'w4', 3);
INSERT INTO Husbands VALUES ('h5', 'w5', 4);
INSERT INTO Husbands VALUES ('h5', 'w6', 5);
INSERT INTO Husbands VALUES ('h5', 'w7', 7);
INSERT INTO Husbands VALUES ('h5', 'w8', 1);

INSERT INTO Husbands VALUES ('h6', 'w1', 7);
INSERT INTO Husbands VALUES ('h6', 'w2', 4);
INSERT INTO Husbands VALUES ('h6', 'w3', 6);
INSERT INTO Husbands VALUES ('h6', 'w4', 5);
INSERT INTO Husbands VALUES ('h6', 'w5', 3);
INSERT INTO Husbands VALUES ('h6', 'w6', 8);
INSERT INTO Husbands VALUES ('h6', 'w7', 2);
INSERT INTO Husbands VALUES ('h6', 'w8', 1);

INSERT INTO Husbands VALUES ('h7', 'w1', 5);
INSERT INTO Husbands VALUES ('h7', 'w2', 1);
```

```
INSERT INTO Husbands VALUES ('h7', 'w3', 4);
INSERT INTO Husbands VALUES ('h7', 'w4', 2);
INSERT INTO Husbands VALUES ('h7', 'w5', 7);
INSERT INTO Husbands VALUES ('h7', 'w6', 3);
INSERT INTO Husbands VALUES ('h7', 'w7', 6);
INSERT INTO Husbands VALUES ('h7', 'w8', 8);

INSERT INTO Husbands VALUES ('h8', 'w1', 2);
INSERT INTO Husbands VALUES ('h8', 'w2', 4);
INSERT INTO Husbands VALUES ('h8', 'w3', 7);
INSERT INTO Husbands VALUES ('h8', 'w4', 3);
INSERT INTO Husbands VALUES ('h8', 'w5', 6);
INSERT INTO Husbands VALUES ('h8', 'w6', 1);
INSERT INTO Husbands VALUES ('h8', 'w7', 5);
INSERT INTO Husbands VALUES ('h8', 'w8', 8);

INSERT INTO Wives VALUES ('w1', 'h1', 6);
INSERT INTO Wives VALUES ('w1', 'h2', 3);
INSERT INTO Wives VALUES ('w1', 'h3', 7);
INSERT INTO Wives VALUES ('w1', 'h4', 1);
INSERT INTO Wives VALUES ('w1', 'h5', 4);
INSERT INTO Wives VALUES ('w1', 'h6', 2);
INSERT INTO Wives VALUES ('w1', 'h7', 8);
INSERT INTO Wives VALUES ('w1', 'h8', 5);

INSERT INTO Wives VALUES ('w2', 'h1', 4);
INSERT INTO Wives VALUES ('w2', 'h2', 8);
INSERT INTO Wives VALUES ('w2', 'h3', 3);
INSERT INTO Wives VALUES ('w2', 'h4', 7);
INSERT INTO Wives VALUES ('w2', 'h5', 2);
INSERT INTO Wives VALUES ('w2', 'h6', 5);
INSERT INTO Wives VALUES ('w2', 'h7', 6);
INSERT INTO Wives VALUES ('w2', 'h8', 1);

INSERT INTO Wives VALUES ('w3', 'h1', 3);
INSERT INTO Wives VALUES ('w3', 'h2', 4);
INSERT INTO Wives VALUES ('w3', 'h3', 5);
INSERT INTO Wives VALUES ('w3', 'h4', 6);
INSERT INTO Wives VALUES ('w3', 'h5', 8);
INSERT INTO Wives VALUES ('w3', 'h6', 1);
INSERT INTO Wives VALUES ('w3', 'h7', 7);
INSERT INTO Wives VALUES ('w3', 'h8', 2);

INSERT INTO Wives VALUES ('w4', 'h1', 8);
INSERT INTO Wives VALUES ('w4', 'h2', 2);
INSERT INTO Wives VALUES ('w4', 'h3', 1);
INSERT INTO Wives VALUES ('w4', 'h4', 3);
```

```
INSERT INTO Wives VALUES ('w4', 'h5', 7);
INSERT INTO Wives VALUES ('w4', 'h6', 5);
INSERT INTO Wives VALUES ('w4', 'h7', 4);
INSERT INTO Wives VALUES ('w4', 'h8', 6);

INSERT INTO Wives VALUES ('w5', 'h1', 3);
INSERT INTO Wives VALUES ('w5', 'h2', 7);
INSERT INTO Wives VALUES ('w5', 'h3', 2);
INSERT INTO Wives VALUES ('w5', 'h4', 4);
INSERT INTO Wives VALUES ('w5', 'h5', 5);
INSERT INTO Wives VALUES ('w5', 'h6', 1);
INSERT INTO Wives VALUES ('w5', 'h7', 6);
INSERT INTO Wives VALUES ('w5', 'h8', 8);

INSERT INTO Wives VALUES ('w6', 'h1', 2);
INSERT INTO Wives VALUES ('w6', 'h2', 1);
INSERT INTO Wives VALUES ('w6', 'h3', 3);
INSERT INTO Wives VALUES ('w6', 'h4', 6);
INSERT INTO Wives VALUES ('w6', 'h5', 8);
INSERT INTO Wives VALUES ('w6', 'h6', 7);
INSERT INTO Wives VALUES ('w6', 'h7', 5);
INSERT INTO Wives VALUES ('w6', 'h8', 4);

INSERT INTO Wives VALUES ('w7', 'h1', 6);
INSERT INTO Wives VALUES ('w7', 'h2', 4);
INSERT INTO Wives VALUES ('w7', 'h3', 1);
INSERT INTO Wives VALUES ('w7', 'h4', 5);
INSERT INTO Wives VALUES ('w7', 'h5', 2);
INSERT INTO Wives VALUES ('w7', 'h6', 8);
INSERT INTO Wives VALUES ('w7', 'h7', 3);
INSERT INTO Wives VALUES ('w7', 'h8', 7);

INSERT INTO Wives VALUES ('w8', 'h1', 8);
INSERT INTO Wives VALUES ('w8', 'h2', 2);
INSERT INTO Wives VALUES ('w8', 'h3', 7);
INSERT INTO Wives VALUES ('w8', 'h4', 4);
INSERT INTO Wives VALUES ('w8', 'h5', 5);
INSERT INTO Wives VALUES ('w8', 'h6', 6);
INSERT INTO Wives VALUES ('w8', 'h7', 1);
INSERT INTO Wives VALUES ('w8', 'h8', 3);

-- wife permutations
CREATE TABLE Wife_perms
(wife CHAR(2) NOT NULL PRIMARY KEY,
 tally INTEGER NOT NULL);
```

```
INSERT INTO Wife_perms VALUES ('w1', 1);
INSERT INTO Wife_perms VALUES ('w2', 2);
INSERT INTO Wife_perms VALUES ('w3', 3);
INSERT INTO Wife_perms VALUES ('w4', 4);
INSERT INTO Wife_perms VALUES ('w5', 5);
INSERT INTO Wife_perms VALUES ('w6', 6);
INSERT INTO Wife_perms VALUES ('w7', 7);
INSERT INTO Wife_perms VALUES ('w8', 8);
```

This query produces the correct results:

```
SELECT W1.wife AS h1, W2.wife AS h2, W3.wife AS h3,
W4.wife AS h4, W5.wife AS h5, W6.wife AS h6, W7.wife AS h7,
W8.wife AS h8
FROM Wife_perms AS W1, Wife_perms AS W2, Wife_perms AS W3,
Wife_perms AS W4, Wife_perms AS W5, Wife_perms AS W6,
Wife_perms AS W7 , Wife_perms AS W8
WHERE (W1.tally + W2.tally + W3.tally + W4.tally +
W5.tally + W6.tally + W7.tally + W8.tally) = 255
AND NOT EXISTS
(SELECT *
FROM Husbands AS H1, Husbands AS H2,
Wives AS W1, Wives AS W2
WHERE H1.man = H2.man
AND H1.ranking > H2.ranking
AND (H1.man || H1.woman) IN
('h1' || H1.wife, 'h2' || H2.wife,
'h3' || H3.wife, 'h4' || H4.wife,
'h5' || H5.wife, 'h6' || H6.wife,
'h7' || H7.wife, 'h8' || H8.wife)
AND H2.woman = W1.woman
AND W1.woman = W2.woman
AND W1.ranking > W2.ranking
AND (W1.man || W1.woman) IN
('h1' || H1.wife, 'h2' || H2.wife,
'h3' || H3.wife, 'h4' || H4.wife,
'h5' || H5.wife, 'h6' || H6.wife,
'h7' || H7.wife, 'h8' || H8.wife));
```

The final results are:

```
h1  h2  h3  h4  h5  h6  h7  h8
================================
w2  w4  w3  w1  w7  w5  w8  w6
w2  w4  w3  w8  w1  w5  w7  w6
w3  w6  w4  w1  w7  w5  w8  w2
w3  w6  w4  w8  w1  w5  w7  w2
w6  w3  w4  w1  w7  w5  w8  w2
w6  w3  w4  w8  w1  w5  w7  w2
w6  w4  w3  w1  w7  w5  w8  w2
w6  w4  w3  w8  w1  w5  w7  w2
w7  w4  w3  w8  w1  w5  w2  w6
```

This example was taken from Niklaus Wirth's book (see the references list at the end of this puzzle).

Answer #3

The key was to maximize the performance of the `NOT EXISTS` test in the final `SELECT`. Unstable is a table of the unstable relationships—in a form designed to be used as the object of the `LIKE` clause in the final query.

```
CREATE TABLE Unstable
(bad_marriage CHAR(16) NOT NULL);

INSERT INTO Unstable
SELECT DISTINCT
       CASE WHEN W.man = 'h1' THEN W.woman
            WHEN Y.man = 'h1' THEN Y.woman
                              ELSE '__' END
     || CASE WHEN W.man = 'h2' THEN W.woman
            WHEN Y.man = 'h2' THEN Y.woman
                              ELSE '__' END
     || CASE WHEN W.man = 'h3' THEN W.woman
            WHEN Y.man = 'h3' THEN Y.woman
                              ELSE '__' END
     || CASE WHEN W.man = 'h4' THEN W.woman
            WHEN Y.man = 'h4' THEN Y.woman
                              ELSE '__' END
     || CASE WHEN W.man = 'h5' THEN W.woman
            WHEN Y.man = 'h5' THEN Y.woman
                              ELSE '__' END
```

```
     || CASE WHEN W.man = 'h6' THEN W.woman
             WHEN Y.man = 'h6' THEN Y.woman
                               ELSE '__' END
     || CASE WHEN W.man = 'h7' THEN W.woman
             WHEN Y.man = 'h7' THEN Y.woman
                               ELSE '__' END
     || CASE WHEN W.man = 'h8' THEN W.woman
             WHEN Y.man = 'h8' THEN Y.woman
                               ELSE '__' END
  FROM Husbands AS W, Husbands AS X,
       Wives AS Y, Wives AS Z
 WHERE W.man = X.man
   AND W.ranking > X.ranking
   AND X.woman = Y.woman
   AND Y.woman = Z.woman
   AND Y.ranking > Z.ranking
   AND Z.man = W.man

SELECT A.name AS h1, B.name AS h2, C.name AS h3, D.name AS
h4,
       E.name AS h5, F.name AS h6, G.name AS h7, H.name AS
h8
  FROM wife_hdr AS a, wife_hdr AS b, wife_hdr AS c, wife_hdr
AS d,
       wife_hdr AS e, wife_hdr AS f, wife_hdr AS g, wife_hdr
AS h
 WHERE B.name NOT IN (A.name)
   AND C.name NOT IN (A.name, B.name)
   AND D.name NOT IN (A.name, B.name, C.name)
   AND E.name NOT IN (A.name, B.name, C.name, D.name)
   AND F.name NOT IN (A.name, B.name, C.name, D.name,
E.name)
   AND G.name NOT IN (A.name, B.name, C.name, D.name,
E.name, F.name)
   AND H.name NOT IN (A.name, B.name, C.name, D.name,
E.name, F.name, G.name)
   AND NOT EXISTS
      (SELECT * FROM Unstable
        WHERE A.name || B.name || C.name || D.name
              || E.name || F.name || G.name || H.name
             LIKE bad_marriage)
```

```
h1   h2   h3   h4   h5   h6   h7   h8
---- ---- ---- ---- ---- ---- ---- ----
w3   w6   w4   w8   w1   w5   w7   w2
w6   w4   w3   w8   w1   w5   w7   w2
w6   w3   w4   w8   w1   w5   w7   w2
w3   w6   w4   w1   w7   w5   w8   w2
w6   w4   w3   w1   w7   w5   w8   w2
w6   w3   w4   w1   w7   w5   w8   w2
w7   w4   w3   w8   w1   w5   w2   w6
w2   w4   w3   w8   w1   w5   w7   w6
w2   w4   w3   w1   w7   w5   w8   w6
```

When Richard timed this, he got 40,000 rows in 4 seconds at an average throughput of around 10,000 rows per second. I don't think I'm going to do any better than that.

The first predicate generates all the permutations of wives in the husband columns, and the `EXISTS()` checks for blocking pairs in the row. This query will take some time to run, especially on a small machine, and it will break down when you have a value of *n* too large for the permutation trick.

REFERENCES

Gusfierld, Dan, and Irving, Robert W., *The Stable Marriage Problem: Structure & Algorithms,* ISBN 0-262-07118-5.

Knuth, Donald E., *CRM Proceedings & Lecture Notes, Vol #10,* "Stable Marriage and Its Relation to Other Combinatorial Problems," ISBN 0-8218-0603-3.

This booklet, which reproduces seven expository lectures given by the author in November 1975, is a gentle introduction to the analysis of algorithms using the beautiful theory of stable marriages as a vehicle to explain the basic paradigms of that subject.

Wirth, Niklaus, Section 3.6, *Algorithms + Data Structures = Programs,* ISBN 0-13-022418-9.

This section gives an answer in Pascal and a short analysis of the algorithm. In particular, I used his data for my example. He gives several answers that give a varying "degrees of happiness" for husbands and wives.

PUZZLE
68 CATCHING THE NEXT BUS

Suppose a man randomly runs to the bus station and catches the next bus leaving the station. Assume there are only two bus routes in this town, and call them "A" and "B." The schedule for each route spaces the bus trips one hour apart. Your first thought may be that the random traveler would spend approximately equal time on both routes, but the truth is that he spends far more time on route A than on route B. Why?

The A bus leaves on the hour, and the B bus leaves at five minutes after the hour. In order to catch the B bus, the traveler has to be in the station between the hour and five after the hour. The rest of the time, he will sit and wait for the A bus to arrive on the hour.

The problem of finding the next bus leaving the station is a fairly easy bit of SQL. Here is a simple table for an imaginary bus line schedule. It gives the route number and the departure and arrival times for one day, without regard to the departure or destination points:

```
CREATE TABLE Schedule
(route_nbr INTEGER NOT NULL,
 depart_time TIMESTAMP NOT NULL,
 arrive_time TIMESTAMP NOT NULL,
 CHECK (depart_time < arrive_time),
 PRIMARY KEY (route_nbr, depart_time));

INSERT INTO Schedule
VALUES (3, '2006-02-09 10:00', '2006-02-09 14:00'),
       (4, '2006-02-09 16:00', '2006-02-09 17:00'),
       (5, '2006-02-09 18:00', '2006-02-09 19:00'),
       (6, '2006-02-09 20:00', '2006-02-09 21:00'),
       (7, '2006-02-09 11:00', '2006-02-09 13:00'),
       (8, '2006-02-09 15:00', '2006-02-09 16:00'),
       (9, '2006-02-09 18:00', '2006-02-09 20:00');
```

If you want to catch the next bus at "2006-02-09 15:30," you should return `(route_nbr = 4)`. If the time is "2006-02-09 16:30," then you should return route numbers 4 and 9, since both of them will depart at the same time.

Answer #1

Okay, how can we solve the problem? The computational way to do so is to find the departure times that occur on or after the time you arrived at the station:

```
SELECT E1.event_id, E1.depart_time, E1.arrive_time
  FROM Events AS E1
 WHERE E1.depart_time
       = (SELECT MIN(E2.depart_time)
            FROM Events AS E2
           WHERE :my_time <= E2.depart_time);
```

This will work fine because the primary key should give you fast access to the departure time column for computing the `MIN()`.

Answer #2

The next question that comes to mind is: Can we find a way to do this without computations?

Let's try adding a column for the waiting period before the departure time in a given day. Route 3 is the first bus out, so if you get to the station any time between midnight and 10:00 hrs on February 9, that is the next bus out. If you get to the station between 10:00 and 11:00 hrs, you will take Route 7, and so forth.

```
CREATE TABLE Schedule
(route_nbr INTEGER NOT NULL,
 wait_time TIMESTAMP NOT NULL,
 depart_time TIMESTAMP NOT NULL,
 arrive_time TIMESTAMP NOT NULL,
 CHECK (depart_time < arrive_time),
 PRIMARY KEY (route_nbr, depart_time));
```

Now the table looks like this:

```
INSERT INTO Schedule
VALUES (3, '2006-02-09 00:00', '2006-02-09 10:00', '2006-
02-09 14:00'),
       (7, '2006-02-09 10:00', '2006-02-09 11:00', '2006-02-
09 13:00'),
```

```
      (4, '2006-02-09 15:00', '2006-02-09 16:00', '2006-02-
09 17:00'),
      (5, '2006-02-09 16:00', '2006-02-09 18:00', '2006-02-
09 19:00'),
      (6, '2006-02-09 18:00', '2006-02-09 20:00', '2006-02-
09 21:00'),
      (8, '2006-02-09 11:00', '2006-02-09 15:00', '2006-02-
09 16:00'),
      (9, '2006-02-09 16:00', '2006-02-09 18:00', '2006-02-
09 20:00');
```

The query can now be done without a subquery:

```
SELECT E1.event_id, E1.depart_time, E1.arrive_time
  FROM Events AS E1
 WHERE :my_time BETWEEN E1.wait_time AND depart_time;
```

Will this run better? Well, it will require an extra timestamp for each row in the schedule. That means (365 days per year * (n) routes in the system * size of a timestamp) extra storage from the first version of the table; this is not that bad. In the case of a bus schedule, you can assume that it will remain constant for at least several months. The real question is how much trouble is computing the wait time? It will be about the same as running the first query once and using the data as part of an `UPDATE` or `INSERT`.

It only takes a little adjusting of your own mindset to start seeing table-driven answers for database problems.

PUZZLE
69 LIFO-FIFO INVENTORY

Imagine a very simple inventory of one kind of item, widgets, into which we add stock once a day. This inventory is then used to fill orders that also come in once a day. Here is the table for this toy problem:

```
CREATE TABLE WidgetInventory
(receipt_nbr INTEGER NOT NULL PRIMARY KEY,
 purchase_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT
NULL,
 qty_on_hand INTEGER NOT NULL
   CHECK (qty_on_hand >= 0),
 unit_price DECIMAL (12,4) NOT NULL);
```

with the following data:

```
 WidgetInventory
 receipt_nbr purchase_date qty_on_hand unit_price
 ================================================
  1          '2005-08-01'   15         10.00
  2          '2005-08-02'   25         12.00
  3          '2005-08-03'   40         13.00
  4          '2005-08-04'   35         12.00
  5          '2005-08-05'   45         10.00
```

The business now sells 100 units on 2005-08-05. How do you calculate the value of the stock sold? There is not one right answer, but here are some options:

1. Use the current replacement cost, which is $10.00 per unit as of 2005-08-05. That would mean the sale cost us only $1,000.00 because of a recent price break.
2. Use the current average price per unit. We have a total of 160 units, for which we paid a total of $1,840.00; that gives us an average cost of $11.50 per unit, or $1,150.00 in total inventory costs.

3. Use LIFO, which stands for "last in, first out." We start by looking at the most recent purchases and work backward through time

```
 2005-08-05: 45 * $10.00 = $450.00 and 45 units
 2005-08-04: 35 * $12.00 = $420.00 and 80 units
 2005-08-03: 20 * $13.00 = $260.00 and 100 with 20 units
left over
```

for a total of $1,130.00 in inventory cost.

4. Use FIFO, which stands for "first in, first out." We start by looking at the earliest purchases and work forward through time

```
 2005-08-01: 15 * $10.00 = $150.00 and 15 units
 2005-08-02: 25 * $12.00 = $300.00 and 40 units
 2005-08-03: 40 * $13.00 = $520.00 and 80 units
 2005-08-04: 20 * $12.00 = $240.00 with 15 units left over
```

for a total of $1,210.00 in inventory costs.

The first two scenarios are trivial to program.

```
CREATE VIEW (current_replacement_cost)
AS
SELECT unit_price
  FROM WidgetInventory
 WHERE purchase_date
       = (SELECT MAX(purchase_date) FROM WidgetInventory);

CREATE VIEW (average_replacement_cost)
AS
SELECT SUM(unit_price * qty_on_hand)/SUM(qty_on_hand)
  FROM WidgetInventory;
```

LIFO and FIFO are more interesting because they involve looking at matching the order against blocks of inventory in a particular order.

Answer #1

Consider this view:

```
CREATE VIEW LIFO (stock_date, unit_price, tot_qty_on_hand,
tot_cost)
AS
SELECT W1.purchase_date, W1.unit_price,
SUM(W2.qty_on_hand), SUM(W2.qty_on_hand *
W2.unit_price)
  FROM WidgetInventory AS W1,
       WidgetInventory AS W2
 WHERE W2.purchase_date <= W1.purchase_date
 GROUP BY W1.purchase_date, W1.unit_price;
```

A row in this view tells us the total quantity on hand, the total cost of the goods in inventory, and what we were paying for items on each date. The quantity on hand is a running total. We can get the LIFO cost with this query:

```
SELECT (tot_cost - ((tot_qty_on_hand - :order_qty) *
unit_price))
        AS cost
  FROM LIFO AS L1
 WHERE stock_date
       = (SELECT MIN(stock_date)
            FROM LIFO AS L2
           WHERE tot_qty_on_hand >= :order_qty);
```

This is straight algebra and a little logic. You need to find the most recent date when we had enough (or more) quantity on hand to meet the order. If, by dumb blind luck, there is a day when the quantity on hand exactly matched the order, return the total cost as the answer. If the order was for more than we have in stock, then return nothing. If we go back to a day when we had more in stock than the order was for, look at the unit price on that day, multiply by the overage, and subtract it.

Answer #2

Alternatively, you can use a derived table and a `CASE` expression. The `CASE` expression computes the cost of units that have a running total quantity less than the `:order_qty` and then performs algebra on the

final block of inventory, which would put the running total over the limit. The outer query does a sum on these blocks:

```
SELECT SUM(W3.v) AS cost
  FROM (SELECT W1.unit_price
              * CASE WHEN SUM(W2.qty_on_hand) <= :order_qty
                     THEN W1.qty_on_hand
                     ELSE :order_qty
                          - (SUM(W2.qty_on_hand) -
W1.qty_on_hand)
                     END
          FROM WidgetInventory AS W1,
               WidgetInventory AS W2
         WHERE W1.purchase_date <= W2.purchase_date
         GROUP BY W1.purchase_date, W1.qty_on_hand,
W1.unit_price
        HAVING (SUM(W2.qty_on_hand) - W1.qty_on_hand) <=
:order_qty)
       AS W3(v);
```

FIFO can be found with a similar `VIEW` or derived table:

```
CREATE VIEW FIFO (stock_date, unit_price, tot_qty_on_hand,
tot_cost)
AS
SELECT W1.purchase_date, W1.unit_price,
       SUM(W2.qty_on_hand), SUM(W2.qty_on_hand *
W2.unit_price)
 FROM WidgetInventory AS W1,
       WidgetInventory AS W2
 WHERE W2.purchase_date <= W1.purchase_date
 GROUP BY W1.purchase_date, W1.unit_price;
```

with the corresponding query:

```
SELECT (tot_cost - ((tot_qty_on_hand - :order_qty) *
unit_price)) AS cost
  FROM FIFO AS F1
 WHERE stock_date
       = (SELECT MIN (stock_date)
            FROM FIFO AS F2
           WHERE tot_qty_on_hand >= :order_qty);
```

These queries and VIEWs only told us the value of the widget inventory. Notice that we never actually shipped anything from the inventory.

Answer #3

How do we write the UPDATE statements that let us change this simple inventory?

What we did not do in part 1 was to actually update the inventory when the widgets were shipped out. Let's build another VIEW that will make life easier:

```
CREATE VIEW StockLevels (purchase_date, previous_qty,
current_qty)
AS
SELECT W1.purchase_date,
       SUM(CASE WHEN W2.purchase_date < W1.purchase_date
                THEN W2.qty_on_hand ELSE 0 END),
       SUM(CASE WHEN W2.purchase_date <= W1.purchase_date
                THEN W2.qty_on_hand ELSE 0 END)
  FROM WidgetInventory AS W1,
       WidgetInventory AS W2
 WHERE W2.purchase_date <= W1.purchase_date
 GROUP BY W1.purchase_date, W1.unit_price;

StockLevels
purchase_date previous_qty current_qty
=======================================
'2005-08-01'       0    15
'2005-08-02'      15    40
'2005-08-03'      40    80
'2005-08-04'      80   115
'2005-08-05'     115   160
```

Using CASE expressions will save you a self-join.

```
CREATE PROCEDURE RemoveQty (IN my_order_qty INTEGER)
LANGUAGE SQL
BEGIN
IF my_order_qty > 0
THEN
```

```
UPDATE WidgetInventory
  SET qty_on_hand
      = CASE
        WHEN my_order_qty
             >= (SELECT current_qty
                   FROM StockLevels AS L
                  WHERE L.purchase_date
                        = WidgetInventory.purchase_date)
        THEN 0
        WHEN my_order_qty
             < (SELECT previous_qty
                  FROM StockLevels AS L
                 WHERE L.purchase_date
                       = WidgetInventory.purchase_date)
        THEN WidgetInventory.qty_on_hand
        ELSE (SELECT current_qty
                FROM StockLevels AS L
               WHERE L.purchase_date =
WidgetInventory.purchase_date)
              - my_order_qty END;
END IF;

-- remove empty bins
DELETE FROM WidgetInventory
 WHERE qty_on_hand = 0;
END;
```

Another inventory problem is how to fill an order with the smallest or greatest number of bins. This assumes that the bins are not in order, so you are free to fill the order as you wish. Using the fewest bins would make less work for the order pickers. Using the greatest number of bins would clean out more storage in the warehouse.

For example, with this data, you could fill an order for 80 widgets by shipping out bins (1, 2, 3) or bins (4, 5). These bins happen to be in date and bin number order in the sample data, but that is not required.

Mathematicians call it (logically enough) a bin-packing problem, and it belongs to the NP-complete family of problems. This kind of problem is too hard to solve for the general case because the work requires trying all the combinations, and this increases too fast for a computer to do it.

However, there are "greedy algorithms" that are often nearly optimal. The idea is to begin by taking the "biggest bite" you can until you have

met or passed your goal. In procedural languages, you can backtrack when you go over the target amount and try to find an exact match or apply a rule that dictates from which bin to take a partial pick.

Answer #4

This is not easy in SQL, because it is a declarative, set-oriented language. A procedural language can stop when it has a solution that is "good enough," while an SQL query tends to find all of the correct answers no matter how long it takes. If you can put a limit on the number of bins you are willing to visit, you can fake an array in a table:

```
CREATE TABLE Picklists
(order_nbr INTEGER NOT NULL PRIMARY KEY,
 goal_qty INTEGER NOT NULL
   CHECK (goal_qty > 0),
 bin_nbr_1 INTEGER NOT NULL UNIQUE,
 qty_on_hand_1 INTEGER DEFAULT 0 NOT NULL
  CHECK (qty_on_hand_1 >= 0),
 bin_nbr_2 INTEGER NOT NULL UNIQUE,
 qty_on_hand_2 INTEGER DEFAULT 0 NOT NULL
  CHECK (qty_on_hand_2 >= 0),
 bin_nbr_3 INTEGER NOT NULL UNIQUE,
 qty_on_hand_3 INTEGER DEFAULT 0 NOT NULL
  CHECK (qty_on_hand_3 >= 0),
 CONSTRAINT not_over_goal
  CHECK (qty_on_hand_1 + qty_on_hand_2 + qty_on_hand_3
          <= goal_qty)
CONSTRAINT bins_sorted
  CHECK (qty_on_hand_1 >= qty_on_hand_2
         AND qty_on_hand_2 >= qty_on_hand_3));
```

Now you can start stuffing bins into the table. This query will give you the ways to fill or almost fill an order with three or fewer bins. The first trick is to load some empty dummy bins into the table. If you want at most (n) picks, then add (n-1) dummy bins:

```
INSERT INTO WidgetInventory VALUES (-1, '1990-01-01', 0
,0.00);
INSERT INTO WidgetInventory VALUES (-2, '1990-01-02', 0
,0.00);
```

The following code shows how to build a CTE or VIEW with the possible pick lists:

```
CREATE VIEW PickCombos(total_pick, bin_1, qty_on_hand_1,
                  bin_2, qty_on_hand_2,
                  bin_3, qty_on_hand_3)
AS
SELECT DISTINCT
      (W1.qty_on_hand + W2.qty_on_hand + W3.qty_on_hand) AS
total_pick,
       CASE WHEN W1.receipt_nbr < 0
            THEN 0 ELSE W1.receipt_nbr END AS bin_1,
W1.qty_on_hand,
       CASE WHEN W2.receipt_nbr < 0
            THEN 0 ELSE W2.receipt_nbr END AS bin_2,
W2.qty_on_hand,
       CASE WHEN W3.receipt_nbr < 0
            THEN 0 ELSE W3.receipt_nbr END AS bin_3,
W3.qty_on_hand
 FROM WidgetInventory AS W1,
      WidgetInventory AS W2,
      WidgetInventory AS W3
WHERE W1.receipt_nbr NOT IN (W2.receipt_nbr,
W3.receipt_nbr)
  AND W2.receipt_nbr NOT IN (W1.receipt_nbr,
W3.receipt_nbr)
  AND W1.qty_on_hand >= W2.qty_on_hand
  AND W2.qty_on_hand >= W3.qty_on_hand;
```

Now you need a procedure to find the pick combination that meets or comes closest to a certain quantity.

```
CREATE PROCEDURE OverPick (IN goal_qty INTEGER)
LANGUAGE SQL
BEGIN
IF goal_qty > 0
THEN
SELECT goal_qty, total_pick, bin_1, qty_on_hand_1,
      bin_2, qty_on_hand_2,
      bin_3, qty_on_hand_3
  FROM PickCombos
 WHERE total_pick
```

```
        = (SELECT MIN (total_pick)
             FROM PickCombos
            WHERE total_pick >= goal_qty)
END IF;
END;
```

With the SQL-99 syntax, the `VIEW` could be put into a CTE and produce a query without a `VIEW`. With the current data and goal of 73 widgets, you can find two picks that together equal 75, namely {3, 4} and {4, 2, 1}.

I will leave it as an exercise for the reader to find a query that underpicks a target quantity.

PUZZLE
70 STOCK TRENDS

You are not supposed to put a calculated column in a table in a pure SQL database. And as the guardian of pure SQL, I should oppose this practice. Well, I do oppose it, but it can be handy and you can do it through `VIEW`s, so it is technically okay.

The first type is values calculated from columns in the same row. In the days when we used punch cards, you would take a deck of cards, run them through a machine that would do the multiplication and addition, and then punch the results in the right-hand side of the cards. For example, the total cost of a line in an order could be described as (closing_price * quantity).

The reason for this calculation was simple; the machines that processed punch cards had no secondary storage, so the data had to be kept on the cards themselves. All 80 columns of a punch card are read into main storage at once, so this was faster than doing the math in the electromechanical hardware.

There is truly no reason for doing this today; it is much faster to recalculate the data than it is to read the results from secondary storage. Disk storage runs in microseconds and CPUs run in nanoseconds.

The second type of calculated data uses data in the same table, but not always in the same row in which it will appear. The third type uses data in the same database in multiple tables.

These last two types are used when the cost of the calculation is higher than the cost of a simple read. In particular, data warehouses love to have this type of data in them to save time.

When and how you do something is important in SQL. Here is an example, based on a thread in an SQL server discussion group. I am changing the table around a bit, and not telling you the names of the guilty parties involved, but the idea still holds. You are given a table that looks like this and you need to calculate a column based on the value in another row of the same table:

```
CREATE TABLE StockHistory
(ticker_sym CHAR(5) NOT NULL,
 sale_date DATE NOT NULL DEFAULT CURRENT_DATE,
 closing_price DECIMAL (10,4) NOT NULL,
 trend INTEGER NOT NULL DEFAULT 0
  CHECK(trend IN(-1, 0, 1))
 PRIMARY KEY (ticker_sym, sale_date));
```

It records the closing price of many different stocks. The trend column is +1 if the closing price increased from the last reported selling closing price, 0 if it stayed the same, and -1 if it dropped in closing price. The trend column is the problem, not because it is hard to compute, but because it can be done several different ways. Let's look at the methods for doing this calculation.

Answer #1

You can write a trigger that will fire after the new row is inserted. While there is an ISO standard SQL/PSM language for writing triggers, the truth is that every vendor has a proprietary trigger language, and they are not compatible. In fact, you will find many different features from product to product and totally different underlying data models. If you decide to use triggers, you will be using proprietary, nonrelational code and have to deal with several problems.

One problem is what a trigger does with a bulk insertion. Given this statement that inserts two rows at the same time:

```
INSERT INTO StockHistory (ticker_sym, sale_date,
closing_price)
 VALUES ('XXX', '2000-04-01', 10.75),
   ('XXX', '2000-04-03', 200.00);
```

`trend` will be set to zero in both of these new rows using the `DEFAULT` clause. But can the trigger see these rows and figure out that the 2000-04-03 row should have a +1 trend or not? Maybe or maybe not, because the new rows are not always committed before the trigger is fired. Also, what should that status of the 2000-04-01 row be? That depends on an already existing row in the table.

But assume that the trigger worked correctly. Now, what if you get this statement:

```
INSERT INTO StockHistory (ticker_sym, sale_date,
closing_price)
VALUES ('XXX', '2000-04-02', 313.25);
```

Did your trigger change the trend in the 2000-04-03 row or not? If I drop a row, does your trigger change the trend in the affected rows? Probably not.

As an exercise, write some trigger code for this problem.

Answer #2

You can do this with insertion statements. I admit I am showing off a bit, but here is one way of inserting data one row at a time. Let me put the statement into a stored procedure:

```
CREATE PROCEDURE NewStockSale
(new_ticker_sym CHAR(5) NOT NULL,
 new_sale_date DATE NOT NULL DEFAULT CURRENT_DATE,
 new_closing_price DECIMAL (10,4) NOT NULL)

INSERT INTO StockHistory (ticker_sym, sale_date,
closing_price, trend)
VALUES (new_ticker_sym, new_sale_date,
   new_closing_price,
   SIGN(new_closing_price
   - (SELECT H1.closing_price
   FROM StockHistory AS H1
  WHERE H1.ticker_sym = StockHistory.ticker_sym
    AND H1.sale_date
   = (SELECT MAX(sale_date)
   FROM StockHistory AS H2
  WHERE H2.ticker_sym = H1.ticker_sym
    AND H2.sale_date < H1.sale_date)
    ))) AS trend
);
```

This is not as bad as you first think. The innermost subquery finds the sale just before the current sale, and then returns its closing price. If the old closing price minus the new closing price is positive, negative, or zero, the `SIGN()` function can compute the value of `trend`. Yes, I was showing off a little bit with this query.

The problem with this is much the same as the triggers. What if I delete a row or add a new row between two existing rows? This statement will not do a thing about changing the other rows.

But there is another problem; this stored procedure is good for only one row at a time. That would mean that at the end of the business day, I would have to write a loop that put one row at a time into the `StockHistory` table.

Your next exercise is to improve this stored procedure.

Answer #3

You can UPDATE the table. You already have a default value of 0 in the trend column, so you could just write an UPDATE statement based on the same logic we have been using.

```
 UPDATE StockHistory
    SET trend
   = SIGN(closing_price -
    (SELECT H1.closing_price
  FROM StockHistory AS H1
 WHERE H1.ticker_sym = StockHistory.ticker_sym
   AND H1.sale_date =
  (SELECT MAX(sale_date)
FROM StockHistory AS H2
    WHERE H2.ticker_sym = H1.ticker_sym
 AND H2.sale_date < H1.sale_date)));
```

While this statement does the job, it will recalculate the trend column for the entire table. What if we only looked at the columns that had a zero? Better yet, what if we made the trend column NULL-able and used the NULLs as a way to locate the rows that need the updates?

```
UPDATE StockHistory
   SET trend = ...
 WHERE trend IS NULL;
```

But this does not solve the problem of inserting a row between two existing dates. Fixing that problem is your third exercise.

Use a VIEW.

This approach will involve getting rid of the trend column in the StockHistory table and creating a VIEW on the remaining columns:

```
CREATE TABLE StockHistory
(ticker_sym CHAR(5) NOT NULL,
 sale_date DATE NOT NULL DEFAULT CURRENT_DATE,
 closing_price DECIMAL (10,4) NOT NULL,
 PRIMARY KEY (ticker_sym, sale_date));

 CREATE VIEW StockTrends (ticker_sym, sale_date,
closing_price, trend)
```

```
 AS SELECT H1.ticker_sym, H1.sale_date, H1.closing_price,
  SIGN(MAX(H2.closing_price) - H1.closing_price)
 FROM StockHistory AS H1, StockHistory AS H2
WHERE H1.ticker_sym = H2.ticker_sym
  AND H2.sale_date < H1.sale_date
GROUP BY H1.ticker_sym, H1.sale_date, H1.closing_price;
```

This approach will handle the insertion and deletion of any number of rows, in any order. The `trend` column will be computed from the existing data each time. The primary key is also a covering index for the query, which helps performance. A covering index contains all of the columns that used the `WHERE` clause of a query.

The major objection to this approach is that the `VIEW` can be slow to build each time if `StockHistory` is a large table.

Answer #4

The OLAP functions in SQL-99 make this so much easier:

```
CREATE VIEW StockTrends (ticker_sym, sale_date,
closing_price, trend)
 AS
SELECT H1.ticker_sym, H1.sale_date, H1.closing_price,
       SIGN (H1.closing_price -
       MAX(closing_price)
       OVER(PARTITION BY ticker_sym
       ORDER BY sale_date DESC
       RANGE BETWEEN 1 PRECEDING AND 1 PRECEDING))
  FROM StockHistory;
```

PUZZLE
71 CALCULATIONS

This problem was posted on a newsgroup in 2006, long after people should not be writing code this bad. The goal is to select three different values in the same row based on three conditions, but it does the math improperly and did not include any DDL. I will start with the original query text:

```
SELECT DISTINCT SUM(A.calc_rslt_val + A.calc_adj_val),
                SUM(A.unit_rslt_val + A.unit_adj_val),
                SUM(OT1.calc_rslt_val + OT1.calc_adj_val),
                SUM(OT1.unit_rslt_val + OT1.unit_adj_val),
                SUM(OT2.calc_rslt_val + OT2.calc_adj_val),
                SUM(OT2.unit_rslt_val + OT2.unit_adj_val)
 FROM Table1 AS A, Table1 AS OT1, Table1 AS OT2, Table2 AS B
 WHERE OT1.emp_id = A.emp_id
   AND OT2.emp_id = A.emp_id
   AND OT1.pin_num = B.pin_num
   AND OT2.pin_num = B.pin_num
   AND A.empl_rcd = 0
   AND A.pin_num = B.pin_num
   AND A.emp_id = 'xxxxxx'
   AND B.pin_num IN ('52636','52751','52768')
   AND A.pin_num = '52636' AND OT1.pin_num = '52751' AND
OT2.pin_num = '52768'
```

Answer #1

USASQL posted a reply that assumed you'll need to use `UNION`s to do something like:

```
SELECT SUM(a_val1), SUM(a_val2), SUM(OT1_val1),
SUM(OT1_val2),
        SUM(ot2_val1), SUM(ot2_val2)
((SELECT DISTINCT
SUM(A.calc_rslt_val + A.calc_adj_val) AS a_val1,
SUM(A.unit_rslt_val + A.unit_adj_val) AS a_val2,
0 AS OT1_val1, 0 AS OT1_val2, 0 AS ot2_val1
0 AS ot2_val1
```

```
   FROM Table1 AS A,Table1 AS OT1, Table1 AS OT2, Table2 AS
B
WHERE --put in condition for a select only
UNION
SELECT DISTINCT 0, 0, SUM(OT1.calc_rslt_val +
OT1.calc_adj_val), SUM(OT1.unit_rslt_val +
OT1.unit_adj_val), 0, 0
FROM Table1 AS A, Table1 AS OT1, Table1 AS OT2, Table2 AS B
WHERE --put in condition for OT1 SELECT only
UNION
SELECT DISTINCT 0, 0, 0, 0, SUM(OT2.calc_rslt_val +
OT2.calc_adj_val),
       SUM(OT2.unit_rslt_val + OT2.unit_adj_val)
  FROM Table1 AS A, Table1 AS OT1, Table1 AS OT2, Table2 AS
B
 WHERE --put in condition for OT1 SELECT only
)
```

Answer #2

This nightmare had no DDL, so people do not have to guess the keys, constraints, declarative referential integrity, data types, and so on. Cleaning it up to make it readable was an effort. Next, look at the logic you have and the way that you were causing a cross product; do you remember algebra?

```
a = b, b = c, c = 2
```

Reduces to

```
a = 2, b = 2, c = 2
```

And let's drop out the table that is never used, except to cause a `CROSS JOIN` problem. Here is a clean version of the original code:

```
SELECT DISTINCT SUM(F1.calc_rslt_val + F1.calc_adj_val) AS
calc_1,
   SUM(F1.unit_rslt_val + F1.unit_adj_val) AS unit_1,
   SUM(f2.calc_rslt_val + f2.calc_adj_val) AS calc_2,
   SUM(f2.unit_rslt_val + f2.unit_adj_val) AS unit_2,
   SUM(f3.calc_rslt_val + f3.calc_adj_val) AS calc_3,
   SUM(f3.unit_rslt_val + f3.unit_adj_val) AS unit_3
```

```
 FROM Foobar AS F1, Foobar AS f2, Foobar AS f3
WHERE F1.empl_id = 'xxxxxx'
 AND f2.empl_id = 'xxxxxx'
 AND f3.empl_id = 'xxxxxx'
 AND F1.empl_rcd = 0
 AND F1.pin_nbr = '52636'
 AND f2.pin_nbr = '52751'
 AND f3.pin_nbr = '52768';
```

See how easy it is to read now? The computed columns need names, too.

Answer #3

But the right way to do it is probably like this. Without DDL, we can only guess as to the NULLs, keys, and what the columns mean, of course:

```
SELECT F1.pin_nbr,
       SUM(F1.calc_rslt_val + F1.calc_adj_val) AS calc_val,
       SUM(F1.unit_rslt_val + F1.unit_adj_val) AS unit_val,
 FROM Foobar AS F1
WHERE F1.empl_id = 'xxxxxx'
 AND F1.empl_rcd = 0
 AND F1.pin_nbr IN ('52636', '52751', '52768')
GROUP BY F1.pin_nbr;
```

What is the basic principle of a tiered architecture? You do all display formatting in the front end and not in the back end. Arrange the columns across the page in the front end.

This will run at least three times faster than Answer #2.

PUZZLE
72 SCHEDULING SERVICE CALLS

This was posted by Sammy on an SQL server newsgroup. He wants help with his design and needs to make sure no employees are is booked on a service call when they are off duty.

The process is pretty straightforward. A client calls to book an appointment, then the dispatcher inputs the desired date and time and searches for the first available employee to book. Once at the client's location, the employee can sell additional SKUs, and all SKUs are based on 30-minute time blocks.

Sammy was after advice on the design of the schema. His first attempt was pretty bad. He used `IDENTITY` (a nonrelational proprietary autonumbering "feature" in T-SQL) as a key. His data element names violated ISO-11179 rules—he even put a "-table" suffix on table names (Department of Redundancy Department)! It was something like this:

```
CREATE TABLE CallsTable
(call_id INTEGER IDENTITY(1,1) NOT NULL
           PRIMARY KEY, --proprietary data type
 client_id INTEGER NOT NULL,
 employee_id INTEGER NOT NULL, -- hard to remember
 call_date SMALLDATETIME NOT NULL, --proprietary data type
 durration INTEGER NOT NULL,
 start_time SMALLDATETIME NOT NULL, --proprietary data type
 start_time INTEGER NOT NULL,
 end_time INTEGER NOT NULL);
```

Answer #1

He had computed columns and did not understand how the `DATETIME` datatype in T-SQL works. Why is a date or time column defined as `INTEGER`? You only need two of the three start, end, and duration columns to compute the third.

The original table design allows for double booking, since you had to store a completed job. The right way to do this is to leave the ending time as a `NULL` that you can `COALESCE()` to the current timestamp.

The rest of the schema was just as bad. Columns were absurdly long and invited garbage data. He numbered his personnel with another `IDENTITY`, avoiding a legally required employee identifier. The table names were singular, to show that he thinks a table is a file. Do you say

"Personnel" or "Employees" when you think of the set of something? That is why you use collective names.

Let's try again, using standard SQL (note that DATETIME in T-SQL becomes TIMESTAMP, which is not the same thing in T-SQL) and ISO-11179 naming conventions:

```
CREATE TABLE ScheduledCalls
(client_id INTEGER NOT NULL
    REFERENCES Clients (client_id),
 scheduled_start_time TIMESTAMP NOT NULL,
 PRIMARY KEY (client_id, emp_id, start_time),
 scheduled_end_time TIMESTAMP NOT NULL,
  CHECK (scheduled_start_time < scheduled_end_time),
 emp_id CHAR(9) DEFAULT '{xxxxxxx}' NOT NULL
    REFERENCES Personnel (emp_id));
```

Notice the use of a dummy employee id '{xxxxxxx}' to hold a slot so the dispatcher can try to find someone. You will need to put the dummy value into the Personnel table and make sure that he is available 24×7 so that integrity checks will work. The curly brackets are a trick to make him sort to the bottom of the displays and reports. Allowing the column to be NULL-able will also work, but I wanted to demonstrate this programming trick.

```
CREATE TABLE Clients
(client_id INTEGER NOT NULL PRIMARY KEY,
 first_name VARCHAR(15) NOT NULL,
 last_name VARCHAR(20) NOT NULL,
 phone_nbr CHAR(15) NOT NULL,
 phone_nbr_2 CHAR(15),
 client_street VARCHAR(35) NOT NULL,
 client_city_name VARCHAR(20) NOT NULL);

CREATE TABLE Personnel
(emp_id CHAR(9) NOT NULL PRIMARY KEY,
 first_name VARCHAR(15) NOT NULL,
 last_name VARCHAR(20) NOT NULL,
 home_phome_nbr CHAR(15) NOT NULL,
 cell_phone_nbr CHAR(15) NOT NULL,
 street_addr VARCHAR(35) NOT NULL,
 city_name VARCHAR(20) NOT NULL,
```

```
 zip_code CHAR(5) NOT NULL);

CREATE TABLE Services
(client_id INTEGER NOT NULL REFERENCES Clients,
 emp_id CHAR(9) NOT NULL REFERENCES Personnel,
 start_time DATETIME NOT NULL,
  FOREIGN KEY (client_id, emp_id, start_time)
  REFERENCES (client_id, emp_id, start_time),
 end_time DATETIME, -- null is an open job
 CHECK (start_time)< end_time),
 sku INTEGER NOT NULL,
PRIMARY KEY (client_id, emp_id, start_time, sku)
);
```

Notice the long natural key. If you do not declare it that way, you will have no data integrity. But newbies will get scared and use things like IDENTITY as a key and never worry about data integrity.

```
CREATE TABLE Inventory
(sku INTEGER NOT NULL PRIMARY KEY,
 stock_descr VARCHAR(50) NOT NULL,
 tax_rate DECIMAL(5,3) NOT NULL,
 duration INTEGER NOT NULL);
```

The real trick is to create a Personnel Schedule table that holds all available dates for each employee.

```
CREATE TABLE PersonnelSchedule
(emp_id CHAR(9) NOT NULL
 REFERENCES Personnel(emp_id),
 avail_start_time DATETIME NOT NULL,
 avail_end_time DATETIME NOT NULL,
  CHECK (avail_start_time < avail_end_time),
PRIMARY KEY (emp_id, avail_start_time));
```

Answer #2

We need someone with available time between the scheduled periods for the job. In this query, the available time must overlap or exactly contain the service call period. The dummy employee is a handy trick to let the dispatcher see a list of available employees via the PK-FK relationship.

```
SELECT P.emp_id,
       S.client_id,
       S.scheduled_start_time,
       S.scheduled_end_time,
  FROM ScheduledCalls AS S,
       PersonnelSchedule AS P
 WHERE S.emp_id = '{xxxxxxx}'
   AND P.emp_id <> '{xxxxxxx}'
   AND S.scheduled_start_time
       BETWEEN P.avail_start_time
           AND P.avail_end_time;
   AND S.scheduled_end_time
       BETWEEN P.avail_start_time
           AND P.avail_end_time;
```

But beware! This will produce all of the available personnel. We will have to leave it to the dispatcher to make the actual assignments.

PUZZLE
73 A LITTLE DATA SCRUBBING

This came in as a data-scrubbing problem from "Stange" at SQL ServerCentral.com. He is importing data from a source that sends him rows with all NULLs. And, no, the source cannot be modified to get rid of these rows on the other side of the system. After staging this data into SQL, he wants to identify the NULL rows and remove them. His fear was that he would have to hard-code:

```
SELECT *
  FROM Staging
 WHERE col1 IS NULL
   AND col2 IS NULL
   AND col3 IS NULL
    etc.
   AND col100 IS NULL;
```

Answer #1

He was playing around with passing the <tablename> as a parameter and then interfacing with the Schema Information tables to identify all columns in said table, get a list of them, then build the query and see if all of these columns are NULL or not. In SQL Server, that would look something like this, but each SQL product would be a little different:

```
SELECT *
  FROM syscolumns
 WHERE id
       = (SELECT id
            FROM sysobjects
           WHERE name = <tablename>);
```

Answer #2

Chris Teter and Jesper both proposed a highly proprietary looping cursor that went through the schema information tables to build dynamic SQL and execute. This was not only nonrelational and highly proprietary, but also very slow.

I am not going to print their code here for the reasons just given, but it shows how programmers fall into a procedural mind-set.

Answer #3

Just do a "cut and paste" from the system utility function that will give you all the column names and drop it into this statement template: Any SQL product will have such a function (e.g., `EXEC sp_columns` in SQL Server).

```
DELETE FROM Staging
 WHERE COALESCE
 (col1, col2, col3, .., col100) IS NULL;
```

This is about as fast as it will get. It also demonstrates that it helps to read the manual and find out what the SQL vendors have given you. Most of these utilities will define a `<column>` and its options (`NULL`-able, `DEFAULT`, key, indexed, etc.) in one row, so you just lift out the name and add a comma after it.

It takes less than five seconds, even for large tables. You will spend more time writing code that will probably fail when the next release of our database comes out and the schema information tables are a little different.

However, you will have to remember to update your SQL every time there is a change to the table or your query will fail every time you have a new release of your system, which will happen much more often than releases of your schema information tables.

PUZZLE
74 DERIVED TABLES OR NOT?

Allen Davidson was trying to join three tables with two LEFT OUTER JOINs and an INNER JOIN to get the SUM() of a few of the columns. Can his query be rewritten to avoid the derived tables?

```
CREATE TABLE Accounts
(acct_nbr INTEGER NOT NULL PRIMARY KEY);
INSERT INTO Accounts VALUES(1), (2), (3), (4);
```

Please notice that the following, Foo and Bar, are not tables, since they have no keys.

```
CREATE TABLE Foo
(acct_nbr INTEGER NOT NULL
  REFERENCES Accounts(acct_nbr),
 foo_qty INTEGER NOT NULL);

INSERT INTO Foo VALUES (1, 10);
INSERT INTO Foo VALUES (2, 20);
INSERT INTO Foo VALUES (2, 40);
INSERT INTO Foo VALUES (3, 80);

CREATE TABLE Bar
(acct_nbr INTEGER NOT NULL
  REFERENCES Accounts(acct_nbr),
  bar_qty INTEGER NOT NULL);
INSERT INTO Bar VALUES (2, 160);
INSERT INTO Bar VALUES (3, 320);
INSERT INTO Bar VALUES (3, 640);
INSERT INTO Bar VALUES (3, 1);
```

His proposed query:

```
SELECT A.acct_nbr,
       COALESCE(F.foo_qty, 0) AS foo_qty_tot,
       COALESCE(B.bar_qty, 0) AS bar_qty_tot
  FROM Accounts AS A
       LEFT OUTER JOIN
```

```
        (SELECT acct_nbr, SUM(foo_qty) AS foo_qty
           FROM Foo
          GROUP BY acct_nbr) AS F
        ON F.acct_nbr = A.acct_nbr
           LEFT OUTER JOIN
           (SELECT acct_nbr, SUM(bar_qty) AS bar_qty
              FROM Bar
             GROUP BY acct_nbr) AS B
           ON F.acct_nbr = B.acct_nbr;
```

This does just fine, but are there other answers?

```
Results
acct_nbr  foo_qty_tot bar_qty_tot
==================================
1            10             0
2            60           160
3            80           961
4             0             0
```

Answer #1

R. Sharma found a way to avoid one derived table, but not both:

```
SELECT A.acct_nbr,
       COALESCE(SUM(F.foo_qty), 0) AS foo_qty_tot,
       COALESCE(MAX(B.bar_qty), 0) AS bar_qty_tot
  FROM (SELECT * FROM Accounts) AS A
       LEFT OUTER JOIN
       (SELECT * FROM Foo) AS F
       ON A.acct_nbr = F.acct_nbr
          LEFT OUTER JOIN
          (SELECT acct_nbr, SUM(bar_qty) AS bar_qty
             FROM Bar
            GROUP BY acct_nbr) AS B
          ON A.acct_nbr = B.acct_nbr
 GROUP BY A.acct_nbr;
```

This will work since the derived table will always get one row per account number so the `MAX()` will ensure the right value. The first one, a derived table, won't be needed because of the one-to-many relationship

between accounts and Foo and the grouping done on `Accounts.acct_nbr`.

Answer #2

Here is my answer. First, assemble the two nontables with the little-used `FULL OUTER JOIN`, which will give you a table with Foo and Bar combined and then we add the Account information.

```
SELECT A.acct_nbr,
       COALESCE (SUM(F.foo_qty), 0) AS foo_qty_tot,
       COALESCE (SUM(B.bar_qty), 0) AS bar_qty_tot
  FROM Accounts AS A
       LEFT OUTER JOIN
        (Foo AS F
         FULL OUTER JOIN
         Bar AS B
         ON F.acct_nbr = B.acct_nbr)
       ON A.acct_nbr = F.acct_nbr
 GROUP BY A.acct_nbr;
```

The other queries have started with the accounts, added nontable Foo, and then added nontable Bar to the mix. Notice that the `OUTER JOIN` is a table! Wow! Maybe those RDBMS principles are useful after all.

I am hoping that the Foo-Bar `JOIN` table will be relatively small, so the `OUTER JOIN` will be quick and they can go into main storage.

PUZZLE
75 FINDING A PUB

This is a common problem for simple maps. The original version of this problem was based on the location of pubs in a city, so that when we got kicked out of one pub, we could locate nearby ones to which to crawl.

The map we use is an (x, y) Cartesian system, which looks like this:

```
CREATE TABLE PubMap
(pub_id CHAR(5) NOT NULL PRIMARY KEY,
 x INTEGER NOT NULL,
 y INTEGER NOT NULL);
```

What I would like is an efficient method for finding the group of points within a neighborhood.

Answer #1

The immediate solution is to use the Cartesian distance formula, $d = \sqrt{((x1-x2)^2 + (y1-y2)^2)}$, and to define a neighborhood as a certain radius from the pub, as the crow flies.

```
SELECT B.pub_id, B.x, B.y
  FROM PubMap AS A,
       PubMap AS B
 WHERE :my_pub <> B.pub_id
   AND SQRT (POWER((A.x - B.x), 2)
             + POWER((A.y - B.y), 2))
       <= :crawl_distance;
```

But if you look at the math, you can save yourself some of the calculation costs. A little algebra tells us that we can square both sides.

```
SELECT B.pub_id, B.x, B.y
  FROM PubMap AS A,
       PubMap AS B
 WHERE :my_pub <> B.pub_id
   AND :my_pub = A.pub_id
   AND (POWER((A.x - B.x), 2)
             + POWER((A.y - B.y), 2))
```

```
        <= POWER(:crawl_distance, 2);
```

Squaring a number is usually pretty fast, since it can now be done as integer multiplications.

Answer #2

If you are willing to give up a direct distance (circular neighborhood model) and look for a square neighborhood, the math gets easier:

```
SELECT A.pub_id, B.pub_id
  FROM PubMap AS A, PubMap AS B
 WHERE :my_pub <> B.pub_id
   AND :my_pub = A.pub_id
   AND ABS(A.x - B.x) <= :distance
   AND ABS(A.y - B.y) <= :distance;
```

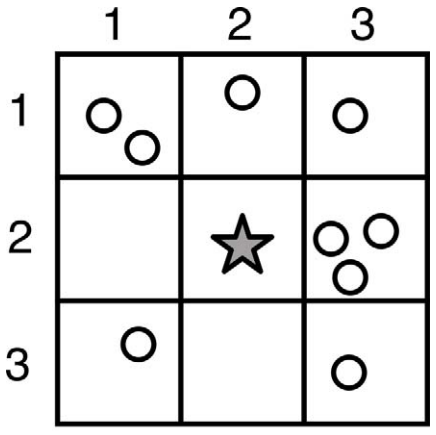
Answer #3

Another approach that is inspired by the square neighborhoods approach is to divide the plane into a large square grid. Each grid cell holds several nodes—think of a typical city map. When a node is located in one quadrant, then the nine adjacent cells centered on his cell would have the nearest neighbor, so I only did a distance formula on the points in those cells.

```
CREATE TABLE PubMap
(pub_id CHAR(5) NOT NULL PRIMARY KEY,
 x INTEGER NOT NULL,
 y INTEGER NOT NULL,
 cell_x INTEGER NOT NULL,
 cell_y INTEGER NOT NULL);
```

It meant carrying an extra pair of cell coordinates, but it saved searching all the nodes—the nodes of 9 cells versus approximately 150,000 nodes in the whole map.

```
SELECT N2.pub_id, N2.x, N2.y
  FROM PubMap AS N1, PubMap AS N2
 WHERE :my_pub <> N2.pub_id
   AND :my_pub = N1.pub_id
```



A pub (star) and its neighbors by grid

```
AND N2.cell_x IN (N1.cell_x-1, N1.cell_x, N1.cell_x+1)
AND N2.cell_y IN (N1.cell_y-1, N1.cell_y, N1.cell_y+1);
```

Use this as a derived table for a limited neighborhood in the first query in place of the B alias, and you will have a good answer for a large map.

This Page Intentionally Left Blank

Index

D

This Page Intentionally Left Blank

ABOUT THE AUTHOR

Joe Celko is a noted consultant and lecturer, and one of the most-read SQL authors in the world. He is well known for his 10 years of service on the ANSI SQL standards committee, his column in *Intelligent Enterprise* magazine (which won several Reader's Choice Awards), and the war stories he tells to provide real-world insights into SQL programming. His best-selling books include *Joe Celko's SQL for Smarties: Advanced SQL Programming, second edition; Joe Celko's SQL Puzzles and Answers;* and *Joe Celko's Trees and Hierarchies in SQL for Smarties.*

This Page Intentionally Left Blank