/THEORY/IN/PRACTICE

The Art of SQL

Stéphane Faroult
with Peter Robson

O'REILLY®

The Art of SQL
by Stéphane Faroult with Peter Robson

Copyright © 2006 O'Reilly Media, Inc. All rights reserved. Printed in the United States of America.

Published by O'Reilly Media, Inc. 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or c*orporate@oreilly.com*.

Editor: Jonathan Gennick
Production Editors: Jamie Peppard and Marlowe Shaeffer
Copyeditor: Nancy Reinhardt
Indexer: Ellen Troutman Zaig
Cover Designer: Mike Kohnke
Interior Designer: Marcia Friedman
Illustrators: Robert Romano, Jessamyn Read, and Lesley Borash

Printing History:

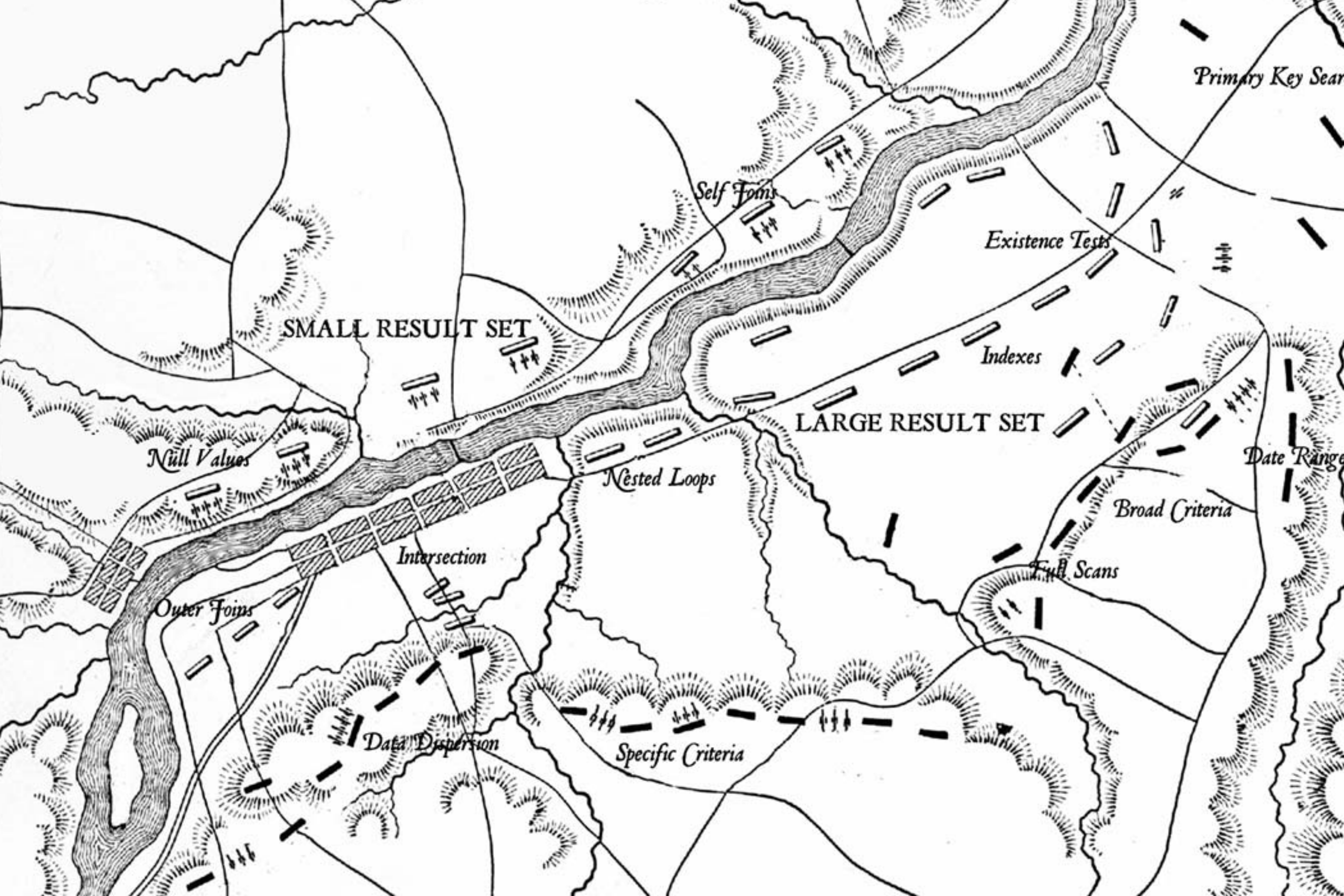
March 2006: First Edition.

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

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

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00894-5
[M]



CHAPTER SIX

The Nine Situations

Recognizing Classic SQL Patterns

Je pense que pour conserver la clarté dans le récit d'une action de guerre, il faut se borner à…ne raconter que les faits principaux et décisifs du combat.

To preserve clarity in relating a military action, I think one ought to be content with… reporting only the facts that affected the decision.

—Général Baron de Marbot (1782–1854)

Mémoires, Book I, xxvi

Any SQL statement that we execute has to examine some amount of data before identifying a result set that must be either returned or changed. The way that we have to attack that data depends on the circumstances and conditions under which we have to fight the battle. As I discuss in Chapter 4, our attack will depend on the amount of data from which we retrieve our result set and on our forces (the filtering criteria), together with the volume of data to be retrieved.

Any large, complicated query can be divided into a succession of simpler steps, some of which can be executed in parallel, rather like a complex battle is often the combination of multiple engagements between various distinct enemy units. The outcome of these different fights may be quite variable. But what matters is the final, overall result.

When we come down to the simpler steps, even when we do not reach a level of detail as small as the individual steps in the execution plan of a query, the number of possibilities is not much greater than the individual moves of pieces in a chess game. But as in a chess game, combinations can indeed be very complicated.

This chapter examines common situations encountered when accessing data in a properly normalized database. Although I refer to queries in this chapter, these example situations apply to updates or deletes as well, as soon as a `where` clause is specified; data must be retrieved before being changed. When filtering data, whether it is for a simple query or to update or delete some rows, the following are the most typical situations—I call them the *nine situations*—that you will encounter:

- Small result set from a few tables with specific criteria applied to those tables
- Small result set based on criteria applied to tables other than the data source tables
- Small result set based on the intersection of several broad criteria
- Small result set from one table, determined by broad selection criteria applied to two or more additional tables
- Large result set
- Result set obtained by self-joining on one table
- Result set obtained on the basis of aggregate function(s)
- Result set obtained by simple searching or by range searching on dates
- Result set predicated on the absence of other data

This chapter deals with each of these situations in turn and illustrates them with either simple, specific examples or with more complex real-life examples collected from different programs. Real-life examples are not always basic, textbook, one- or two-table affairs. But the overall pattern is usually fairly recognizable.

As a general rule, what we require when executing a query is the filtering out of any data that does not belong in our final result set as soon as possible; this means that we must apply the most efficient of our search criteria as soon as possible. Deciding which criterion to apply first is normally the job of the optimizer. But, as I discuss in Chapter 4, the optimizer must take into account a number of variable conditions, from the physical implementation of tables to the manner in which we have written a query. Optimizers do not always "get it right," and there are things we can do to facilitate performance in each of our nine situations.

Small Result Set, Direct Specific Criteria

The typical online transaction-processing query is a query returning a small result set from a few tables and with very specific criteria applied to those tables. When we are looking for a few rows that match a selective combination of conditions, our first priority is to pay attention to indexes.

The trivial case of a single table or even a join between two tables that returns few rows presents no more difficulty than ensuring that the query uses the proper index. However, when many tables are joined together, and we have input criteria referring to, for instance, two distinct tables `TA` and `TB`, then we can either work our way from `TA` to `TB` or from `TB` to `TA`. The choice depends on how fast we can get rid of the rows we do not want. If statistics reflect the contents of tables with enough accuracy, the optimizer should, hopefully, be able to make the proper decision as to the join order.

When writing a query to return few rows, and with direct, specific criteria, we must identify the criteria that are most efficient at filtering the rows; if some criteria are highly critical, before anything else, we must make sure that the columns corresponding to those criteria are indexed and that the indexes can be used by the query.

Index Usability

You've already seen in Chapter 3 that whenever a function is applied to an indexed column, a regular index cannot be used. Instead, you would have to create a functional index, which means that you index the result of the function applied to the column instead of indexing the column.

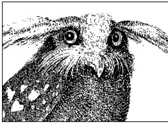
Remember too that you don't have to explicitly invoke a function to see a function applied; if you compare a column of a given type to a column or literal value of a different type, the DBMS may perform an implicit type conversion (an implicit call to a conversion function), with the performance hit that one can expect.

Once we are certain that there are indexes on our critical search criteria and that our query is written in such a way that it can take full advantage of them, we must distinguish between unique index fetches of a single row, and other fetches—non-unique index or a range scan of a unique index.

Query Efficiency and Index Usage

Unique indexes are excellent when joining tables. However, when the input to a query is a primary key and the value of the primary key is not a primitive input to the program, then you may have a poorly designed program on your hands.

What I call *primitive input* is data that has been fed into the program, either typed in by a user or read from a file. If the primary key value has been derived from some primitive input and is itself the result of a query, the odds are very high that there is a massive design flaw in the program. Because this situation often means that the output of one query is used as the input to another one, you should check whether the two queries can be combined.



Excellent queries don't necessarily come from excellent programs.

Data Dispersion

When indexes are not unique, or when a condition on a unique index is expressed as a range, for instance:

```
where customer_id between ... and ...
```

or:

```
where supplier_name like 'SOMENAME%'
```

the DBMS must perform a range scan. Rows associated with a given key may be spread all over the table being queried, and this is something that a cost-based optimizer often understands. There are therefore cases when an index range scan would require the DBMS kernel to fetch, one by one, a large number of table data pages, each with very few rows of relevance to the query, and when the optimizer decides that the DBMS kernel is better off scanning the table and ignoring the index.

You saw in Chapter 5 that many database systems offer facilities such as table partitions or clustered indexes to direct the storage of data that we would like to retrieve together. But the mere nature of data insertion processes may well lead to clumping of data. When we associate a timestamp with each row and do mostly inserts into a table, the chances are that most rows will be inserted next to one another (unless we have taken special measures to limit contention, as I discuss in Chapter 9). The physical proximity of the inserted rows is not an absolute necessity and, in fact, the notion of order as such is totally foreign to relational algebra. But, in practice, it is what may happen. Therefore,

when we perform a range scan on the index on the timestamp column to look for index entries close together in time, the chances are that the rows in question will be close together too. Of course, this will be even truer if we have tweaked the storage so as to get such a result.

Now, if the value of a key bears no relation to any peculiar circumstance of insertion nor to any hidden storage trick, the various rows associated with a key value or with a range of key values can be physically placed anywhere on disk. The keys in the index are always, by construction, held in sorted order. But the associated rows will be randomly located in the table. In practice, we shall have to visit many more blocks to answer a query involving such an index than would be the case were the table partitioned or the index clustered. We can have, therefore, two indexes on the same table, with strictly identical degrees of selectivity, one of which gives excellent results, and the other one, significantly worse results, a situation that was mentioned in Chapter 3 and that it is now time to prove.

To illustrate this case I have created a 1,000,000–row table with three columns `c1`, `c2`, and `c3`, `c1` being filled with a sequence number (1 to 1,000,000), `c2` with all different random numbers in the range 1 to 2,000,000, and `c3` with random values that can be, and usually are, duplicated. On face value, and from a logical point of view, `c1` and `c2` are both unique and therefore have identical selectivity. In the case of the index on column `c1`, the order of the rows in the table matches the order in the index. In a real case, some activity against the table might lead to "holes" left by deletions and subsequently filled with out-of-order records due to new insertions. By contrast, the order of the rows in the table bears no relation to the ordering of the keys in the index on `c2`.

When we fetch `c3`, based on a range condition of the type:

```
where column_name between some_value and some_value + 10
```

it makes a significant difference whether we use `c1` and its associated index (the *ordered* index, where keys are ordered as the rows in the table) or `c2` and its associated index (the *random* index), as you can see in Figure 6-1. Don't forget that we have such a difference because additional accesses to the table are required in order to fetch the value of `c3`; there would be no difference if we had two composite indexes, on `(c1, c3)` and `(c2, c3)`, because then we could return everything from an index in which the keys are ordered.

The type of difference illustrated in Figure 6-1 also explains why sometimes performance can degrade over time, especially when a new system is put into production with a considerable amount of data coming from a legacy system. It may happen that the initial data loading imposes some physical ordering that favors particular queries. If a few months of regular activity subsequently destroys this order, we may suffer over this period a mysterious 30–40% degradation of performance.

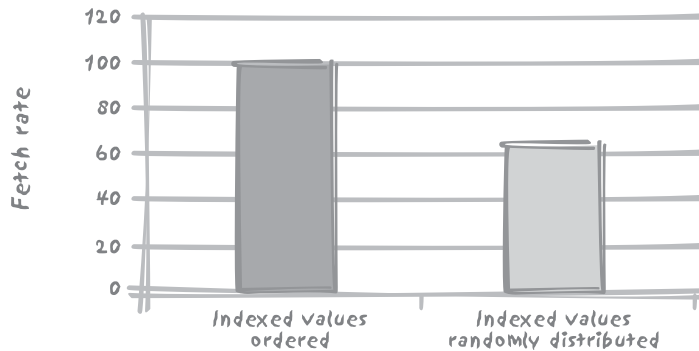
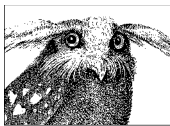


FIGURE 6-1. *Difference of performance when the order in the index matches the order of the rows in the table*

It should be clear by now that the solution "can't the DBAs reorganize the database from time to time?" is indeed a fudge, not a solution. Database reorganizations were once quite in vogue. Ever-increasing volumes, 99.9999% uptime requirements and the like have made them, for the most part, an administrative task of the past. If the physical implementation of rows really is crucial for a critical process, then consider one of the self-organizing structures discussed Chapter 5, such as clustered indexes or index-organized tables. But keep in mind that what favors one type of query sometimes disadvantages another type of query and that we cannot win on all fronts.



Performance variation between comparable indexes may be due to physical data dispersion.

Criterion Indexability

Understand that the proper indexing of specific criteria is an essential component of the "small set, direct specific criteria" situation. We can have cases when the result set is small and some criteria may indeed be quite selective, but are of a nature that isn't suitable for indexing: the following real-life example of a search for differences among different amounts in an accounting program is particularly illustrative of a very selective criterion, yet unfit for indexing.

In the example to follow, a table named `glreport` contains a column named `amount_diff` that ought to contain zeroes. The purpose of the query is to track accounting errors, and identify where `amount_diff` isn't zero. Directly mapping ledgers to tables and applying a logic that dates back to a time when these ledgers where inked with a quill is rather questionable when using a modern DBMS, but unfortunately one encounters

questionable databases on a routine basis. Irrespective of the quality of the design, a column such as amount_diff is typical of a column that should not be indexed: ideally amount_diff should contain nothing but zeroes, and furthermore, it is obviously the result of a denormalization and the object of numerous computations. Maintaining an index on a column that is subjected to computations is even costlier than maintaining an index on a static column, since a modified key will "move" inside the index, causing the index to undergo far more updates than from the simple insertion or deletion of nodes.

All specific criteria are not equally suitable for indexing. In particular, columns that are frequently updated increase maintenance costs.

Returning to the example, a developer came to me one day saying that he had to optimize the following Oracle query, and he asked for some expert advice about the execution plan:

```
select
    total.deptnum,
    total.accounting_period,
    total.ledger,
    total.cnt,
    error.err_cnt,
    cpt_error.bad_acct_count
from
 -- First in-line view
 (select
      deptnum,
      accounting_period,
      ledger,
      count(account) cnt
 from
      glreport
 group by
      deptnum,
      ledger,
      accounting_period) total,
 -- Second in-line view
 (select
     deptnum,
     accounting_period,
     ledger,
     count(account) err_cnt
 from
     glreport
 where
     amount_diff <> 0
```

```
 group by
     deptnum,
     ledger,
     accounting_period) error,
 -- Third in-line view
 (select
     deptnum,
     accounting_period,
     ledger,
     count(distinct account) bad_acct_count
 from
     glreport
 where
     amount_diff <> 0
 group by
     deptnum,
     ledger,
     accounting_period
 ) cpt_error
where
   total.deptnum = error.deptnum(+) and
   total.accounting_period = error.accounting_period(+) and
   total.ledger = error.ledger(+) and
   total.deptnum = cpt_error.deptnum(+) and
   total.accounting_period = cpt_error.accounting_period(+) and
   total.ledger = cpt_error.ledger(+)
order by
   total.deptnum,
   total.accounting_period,
   total.ledger
```

For readers unfamiliar with Oracle-specific syntax, the several occurrences of (+) in the outer query's `where` clause indicate outer joins. In other words:

```
select whatever
from ta,
     tb
where ta.id = tb.id (+)
```

is equivalent to:

```
select whatever
from ta
     outer join tb
             on tb.id = ta.id
```

The following SQL*Plus output shows the execution plan for the query:

```
10:16:57 SQL> set autotrace traceonly
10:17:02 SQL> /

37 rows selected.

Elapsed: 00:30:00.06
```

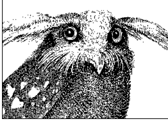
```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
                    (Cost=1779554 Card=154 Bytes=16170)
   1    0   MERGE JOIN (OUTER) (Cost=1779554 Card=154 Bytes=16170)
   2    1     MERGE JOIN (OUTER) (Cost=1185645 Card=154 Bytes=10780)
   3    2       VIEW (Cost=591736 Card=154 Bytes=5390)
   4    3         SORT (GROUP BY) (Cost=591736 Card=154 Bytes=3388)
   5    4           TABLE ACCESS (FULL) OF 'GLREPORT'
                            (Cost=582346 Card=4370894 Bytes=96159668)
   6    2       SORT (JOIN) (Cost=593910 Card=154 Bytes=5390)
   7    6         VIEW (Cost=593908 Card=154 Bytes=5390)
   8    7           SORT (GROUP BY) (Cost=593908 Card=154 Bytes=4004)
   9    8             TABLE ACCESS (FULL) OF 'GLREPORT'
                              (Cost=584519 Card=4370885 Bytes=113643010)
  10    1     SORT (JOIN) (Cost=593910 Card=154 Bytes=5390)
  11   10       VIEW (Cost=593908 Card=154 Bytes=5390)
  12   11         SORT (GROUP BY) (Cost=593908 Card=154 Bytes=5698)
  13   12           TABLE ACCESS (FULL) OF 'GLREPORT'
                            (Cost=584519 Card=4370885 Bytes=161722745)


Statistics
----------------------------------------------------------
        193  recursive calls
          0  db block gets
    3803355  consistent gets
    3794172  physical reads
       1620  redo size
       2219  bytes sent via SQL*Net to client
        677  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
         17  sorts (memory)
          0  sorts (disk)
         37  rows processed
```

I must confess that I didn't waste too much time on the execution plan, since its most striking feature was fairly apparent from the text of the query itself: it shows that the table `glreport`, a tiny 4 to 5 million–row table, is accessed three times, once per subquery, and each time through a full scan.

Nested queries are often useful when writing complex queries, especially when you mentally divide each step, and try to match a subquery to every step. But nested queries are not silver bullets, and the preceding example provides a striking illustration of how easily they may be abused.

The very first inline view in the query computes the number of accounts for each department, accounting period, and ledger, and represents a full table scan that we cannot avoid. We need to face realities; we have to fully scan the table, because we are including all rows when we check how many accounts we have. We need to scan the table once, but do we absolutely need to access it a second or third time?



If a full table scan is required, indexes on the table become irrelevant.

What matters is to be able to not only have a very analytic view of processing, but also to be able to stand back and consider what we are doing in its entirety. The second inline view counts exactly the same things as the first one—except that there is a condition on the value of `amount_diff`. Instead of counting with the `count()` function, we can, at the same time as we compute the total count, add 1 if `amount_diff` is not 0, and 0 otherwise. This is very easy to write with the Oracle-specific `decode(u, v, w, x)` function or using the more standard `case when u = v then w else x end` construct.

The third inline view filters the same rows as the second one; however, here we want to count distinct account numbers. This counting is a little trickier to merge into the first subquery; the idea is to replace the account numbers (which, by the way, are defined as `varchar2`* in the table) by a value which is totally unlikely to occur when `amount_diff` is 0; `chr(1)` (Oracle-speak to mean *the character corresponding to the ASCII value 1*) seems to be an excellent choice (I always feel a slight unease at using `chr(0)` with something written in C like Oracle, since C terminates all character strings with a `chr(0)`). We can then count how many distinct accounts we have and, of course, subtract one to avoid counting the dummy `chr(1)` account.

So this is the suggestion that I returned to the developer:

```
select  deptnum,
        accounting_period,
        ledger,
        count(account) nb,
        sum(decode(amount_diff, 0, 0, 1)) err_cnt,
        count(distinct decode(amount_diff, 0, chr(1), account)) - 1
                                     bad_acct_count
 from
      glreport
 group by
       deptnum,
       ledger,
       accounting_period
```

My suggestion was reported to be four times as fast as the initial query, which came as no real surprise since the three full scans had been replaced by a single one.

Note that there is no longer any `where` clause in the query: we could say that the condition on `amount_diff` has "migrated" to both the logic performed by the `decode()` function inside the select list and the aggregation performed by the `group by` clause. The

* To non-Oracle users, the `varchar2` type is, for all practical purposes, the same as the `varchar` type.

replacement of a filtering condition that *looked* specific with an aggregate demonstrates that we are here in another situation, namely a result set obtained on the basis of an aggregate function.

In-line queries can simplify a query, but can result in excessive and duplicated processing if used without care.

Small Result Set, Indirect Criteria

A situation that is superficially similar to the previous one is when you have a small result set that is based on criteria applied to tables *other than* the data source tables. We want data from one table, and yet our conditions apply to other, related tables from which we don't want any data to be returned. A typical example is the question of "which customers have ordered a particular item" that we amply discussed earlier in Chapter 4. As you saw in Chapter 4, this type of query can be expressed in either of two ways:

- As a regular join with a `distinct` to remove duplicate rows that are the result, for instance, of customers having ordered the same item several times
- By way of either a correlated or uncorrelated subquery

If there is some particularly selective criterion to apply to the table (or tables) from which we obtain the result set, there is no need to say much more than what has been said in the previous situation "Small Result Set, Direct Specific Criteria": the query will be driven by the selective criterion. and the same reasoning applies. But if there is no such criterion, then we have to be much more careful.

To take a simplified version of the example in Chapter 4, identifying the customers who have ordered a Batmobile, our typical case will be something like the following:

```
select distinct orders.custid
from orders
     join orderdetail
        on (orderdetail.ordid = orders.ordid)
     join articles
        on (articles.artid = orderdetail.artid)
where articles.artname = 'BATMOBILE'
```

In my view it is much better, because it is more understandable, to make explicit the test on the presence of the article in a customer's orders by using a subquery. But should that subquery be correlated or uncorrelated? Since we have no other criterion, the answer should be clear: uncorrelated. If not, one would have to scan the `orders` table and fire the subquery for each row—the type of big mistake that passes unnoticed when we start with a small `orders` table but becomes increasingly painful as the business gathers momentum.

The uncorrelated subquery can either be written in the classic style as:

```
select distinct orders.custid
from orders
where ordid in (select orderdetails.ordid
                from orderdetail
                     join articles
                       on (articles.artid = orderdetail.artid)
                where articles.artname = 'BATMOBILE')
```

or as a subquery in the `from` clause:

```
select distinct orders.custid
from orders,
     (select orderdetails.ordid
      from orderdetail
           join articles
             on (articles.artid = orderdetail.artid)
      where articles.artname = 'BATMOBILE') as sub_q
where sub_q.ordid = orders.ordid
```

I find the first query more legible, but it is really a matter of personal taste. Don't forget that an `in()` condition on the result of the subquery implies a `distinct` and therefore a sort, which takes us to the fringe of the relational model.

Where using subqueries, think carefully before choosing either a correlated or uncorrelated subquery.

Small Intersection of Broad Criteria

The situation we talk about in this section is that of a small result set based on the intersection of several broad criteria. Each criterion individually would produce a large result set, yet the intersection of those individual, large sets is a very small, final result set returned by the query.

Continuing on with our query example from the preceding section, if the existence test on the article that was ordered is not selective, we must necessarily apply some other criteria elsewhere (otherwise the result set would no longer be a small result set). In this case, the question of whether to use a regular join, a correlated subquery, or an uncorrelated subquery usually receives a different answer depending on both the relative "strength" of the different criteria and the existing indexes.

Let's suppose that instead of checking people who have ordered a Batmobile, admittedly not our best-selling article, we look for customers who have ordered something that I hope is much less unusual, in this case some soap, but purchased last Saturday. Our query then becomes something like this:

```
select distinct orders.custid
from orders
     join orderdetail
        on (orderdetail.ordid = orders.ordid)
     join articles
        on (articles.artid = orderdetail.artid)
where articles.artname = 'SOAP'
  and <selective criterion on the date in the orders table>
```

Quite logically, the processing flow will be the reverse of what we had with a selective article: get the article, then the order lines that contained the article, and finally the orders. In the case we're currently discussing, that of orders for soap, we should first get the small number of orders placed during the relatively short interval of time, and then check which ones refer to the article soap. From a practical point of view, we are going to use a totally different set of indexes. In the first case, ideally, we would like to see one index on the article name and one on the article identifier in the orderdetail table, and then we would have used the index on the primary key ordid in the orders table. In the case of orders for soap, what we want to find is an index on the date in orders and then one on orderid in orderdetail, from which we can use the index on the primary key of articles—assuming, of course, that in both cases using the indexes is the best course to take.

The obvious natural choice to get customers who bought soap last Saturday would appear to be a correlated subquery:

```
select distinct orders.custid
from orders
where <selective criterion on the date in the orders table>
  and exists (select 1
              from orderdetail
                  join articles
                    on (articles.artid = orderdetail.artid)
                where articles.artname = 'SOAP'
                  and orderdetails.ordid = orders.ordid)
```

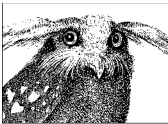
In this approach, we take for granted that the correlated subquery executes very quickly. Our assumption will prove true only if orderdetail is indexed on ordid (we shall then get the article through its primary key artid; therefore, there is no other issue).

You've seen in Chapter 3 that indexes are something of a luxury in transactional databases, due to their high cost of maintenance in an environment of frequent inserts, updates, and deletes. This cost may lead us to opt for a "second-best" solution. The absence of the vital index on orderdetail and good reason for not creating further indexes might prompt us to consider the following:

```
select distinct orders.custid
from orders,
     (select orderdetails.ordid
      from orderdetail,
           articles
```

```
        where articles.artid = orderdetail.artid
          and articles.artname = 'SOAP') as sub_q
  where sub_q.ordid = orders.ordid
    and <selective criterion on the date in the orders table>
```

In this second approach, the index requirements are different: if we don't sell millions of articles, it is likely that the condition on the article name will perform quite satisfactorily even in the absence of any index on `artname`. We shall probably not need any index on the column `artid` of `orderdetail` either: if the article is popular and appears in many orders, the join between `orderdetail` and `articles` is probably performed in a more efficient manner by hash or merge join, rather than by a nested loop that would need such an index on `artid`. Compared to the first approach, we have here a solution that we could call a *low index* solution. Because we cannot afford to create indexes on each and every column in a table, and because we usually have in every application a set of "secondary" queries that are not absolutely critical but only require a decent response time, the *low index* approach may perform in a perfectly acceptable manner.



Adding one extra search criterion to an existing query can completely change a previous construct: ***a modified query is a new query.***

Small Intersection, Indirect Broad Criteria

An *indirect criterion* is one that applies to a column in a table that you are joining only for the purpose of evaluating the criterion. The retrieval of a small result set through the intersection of two or more broad criteria, as in the previous situation "Small Intersection of Broad Criteria," is often a formidable assignment. Obtaining the intersection of the large intermediary result sets by joining from a central table, or even through a chain of joins, makes a difficult situation even more daunting. This situation is particularly typical of the "star schema" that I discuss in some detail in Chapter 10, but you'll also encounter it fairly frequently in operational databases. When you are looking for that rare combination of multiple nonselective conditions on the columns of the row, you must expect to perform full scans at some point. The case becomes particularly interesting when several tables are involved.

The DBMS engine needs to start from somewhere. Even if it can process data in parallel, at some point it has to start with one table, index, or partition. Even if the resulting set defined by the intersection of several huge sets of data is very small, a boot-strapping full table scan, and possibly two scans, will be required—with a nested loop, hash join, or merge join performed on the result. The difficulty will then be to identify which

combination of tables (not necessarily the smallest ones) will result in the least number of rows from which the final result set will be extracted. In other words, we must find the weakest point in the line of defense, and once we have eliminated it, we must concentrate on obtaining the final result set.

Let me illustrate such a case with a real-life Oracle example. The original query is a pretty complicated query, with two tables each appearing twice in the `from` clause. Although none of the tables is really enormous (the biggest one contains about 700,000 rows), the problem is that none of the nine parameters that are passed to the query is really selective:

```
select (data from ttex_a,
                  ttex_b,
                  ttraoma,
                  topeoma,
                  ttypobj,
                  ttrcap_a,
                  ttrcap_b,
                  trgppdt,
                  tstg_a)
from ttrcapp ttrcap_a,
     ttrcapp ttrcap_b,
     tstg tstg_a,
     topeoma,
     ttraoma,
     ttex ttex_a,
     ttex ttex_b,
     tbooks,
     tpdt,
     trgppdt,
     ttypobj
where ( ttraoma.txnum = topeoma.txnum )
  and ( ttraoma.bkcod = tbooks.trscod )
  and ( ttex_b.trscod = tbooks.permor )
  and ( ttraoma.trscod = ttrcap_a.valnumcod )
  and ( ttex_a.nttcod = ttrcap_b.valnumcod )
  and ( ttypobj.objtyp = ttraoma.objtyp )
  and ( ttraoma.trscod = ttex_a.trscod )
  and ( ttrcap_a.colcod = :0 ) -- not selective
  and ( ttrcap_b.colcod = :1 ) -- not selective
  and ( ttraoma.pdtcod = tpdt.pdtcod )
  and ( tpdt.risktyp = trgppdt.risktyp )
  and ( tpdt.riskflg = trgppdt.riskflg )
  and ( tpdt.pdtcod = trgppdt.pdtcod )
  and ( trgppdt.risktyp = :2 ) -- not selective
  and ( trgppdt.riskflg = :3 ) -- not selective
  and ( ttraoma.txnum = tstg_a.txnum )
  and ( ttrcap_a.refcod = :5 ) -- not selective
  and ( ttrcap_b.refcod = :6 ) -- not selective
  and ( tstg_a.risktyp = :4 ) -- not selective
  and ( tstg_a.chncod = :7) -- not selective
  and ( tstg_a.stgnum = :8 ) -- not selective
```

When run with suitable parameters (here indicated as :0 to :8), the query takes more than 25 seconds to return fewer than 20 rows, doing about 3,000 physical I/Os and hitting data blocks 3,000,000 times. Statistics correctly represent the actual contents of tables (one of the very first things to check), and a query against the data dictionary gives the number of rows of the tables involved:

```
TABLE_NAME                   NUM_ROWS
-------------------------- ----------
ttypobj                           186
trgppdt                           366
tpdt                             5370
topeoma                         12118
ttraoma                         12118
tbooks                          12268
ttex                           102554
ttrcapp                        187759
tstg                           702403
```

A careful study of the tables and of their relationships allows us to draw the enemy position of Figure 6-2, showing our weak criteria represented as small arrows, and tables as boxes the size of which approximately indicates the number of rows. One thing is especially remarkable: the central position of the `ttraoma` table that is linked to almost every other table. Unfortunately, all of our criteria apply elsewhere. By the way, an interesting fact to notice is that we are providing two values to match columns `risktyp` and `riskflg` of `trgppdt`—which is joined to `tpdt` on those very two columns, plus `pdtcod`. In such a case, it can be worth contemplating reversing the flow—for example, comparing the columns of `tpdt` to the constants provided, and only then pulling the data from `trgppdt`.

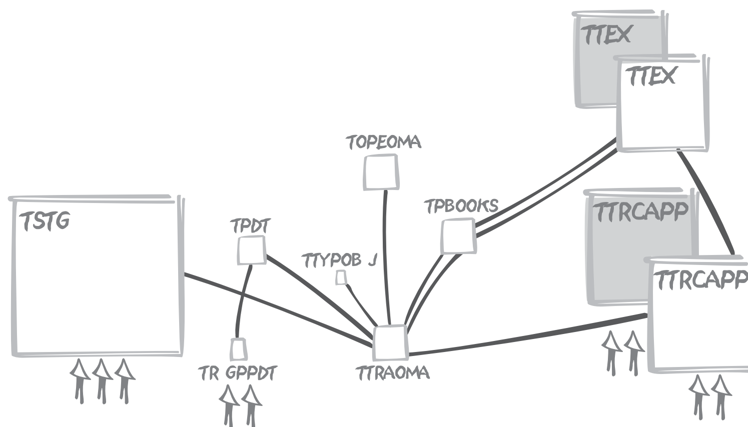


FIGURE 6-2. *The enemy position*

Most DBMS allow you to check the execution plan chosen by the optimizer, either through the `explain` command or sometimes by directly checking in memory how something has been executed. When this query took 25 seconds, the plan, although not especially atrocious, was mostly a full scan of `ttraoma` followed by a series of nested loops,

using the various indexes available rather efficiently (it would be tedious to detail the numerous indexes, but suffice to say that all columns we are joining on are correctly indexed). Is this full scan the reason for slowness? Definitely not. A simple test, fetching all the rows of `ttraoma` (without displaying them to avoid the time associated with displaying characters on a screen) proves that it takes just a tiny fraction, hardly measurable, of the elapsed time for the overall query.

When we consider the weak criteria we have, our forces are too feeble for a frontal attack against `tstg`, the bulk of the enemy troops, and even `ttrcap` won't lead us very far, because we have poor criteria against each instance of this table, which intervenes twice in the query. However, it should be obvious that the key position of `ttraoma`, which is relatively small, makes an attack against it, as a first step, quite sensible—precisely the decision that the optimizer makes without any prompting.

If the full scan is not to blame, then where did the optimizer go wrong? Have a look at Figure 6-3, which represents the query as it was executed.

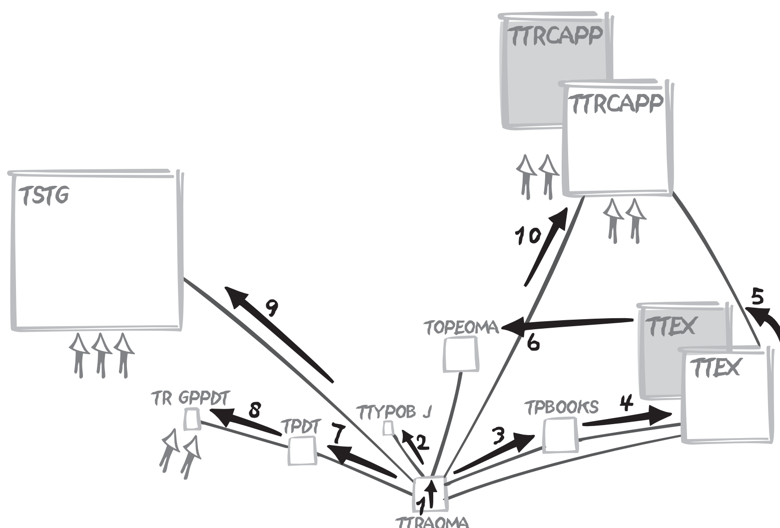


FIGURE 6-3. *What the optimizer chose to do*

When we check the order of operations, it all becomes obvious: our criteria are so bad, on face value, that the optimizer chose to ignore them altogether. Starting with a pretty reasonable full scan of `ttraoma`, it then chose to visit all the smallish tables gravitating around `ttraoma` before ending with the tables to which our filtering criteria apply. This approach is the mistake. It is likely that the indexes of the tables we first visit look much more efficient to the optimizer, perhaps because of a lower average number of table rows per key or because the indexes more closely match the order of the rows in the tables. But postponing the application of our criteria is not how we cut down on the number of rows we have to process and check.

Once we have taken `ttraoma` and hold the key position, why not go on with the tables against which we have criteria instead? The join between those tables and `ttraoma` will help us eliminate unwanted rows from `ttraoma` before proceeding to apply joins with the other tables. This is a tactic that is likely to pay dividends since—and this is information we have but that is unknown to the optimizer—we know we should have, in all cases, very few resulting rows, which means that our combined criteria should, through the joins, inflict heavy casualties among the rows of `ttraoma`. Even when the number of rows to be returned is larger, the execution path I suggest should still remain relatively efficient.

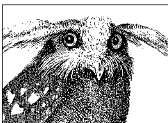
How then can we force the DBMS to execute the query as we want it to? It depends on the SQL dialect. As you'll see in Chapter 11, most SQL dialects allow directives, or hints, to the optimizer, although each dialect uses different syntax for such hints—telling the optimizer, for instance, to take on the tables in the same order as they are listed in the `from` clause. The trouble with hints is that they are more imperative than their name suggests, and every hint is a gamble on the future—a bet that circumstances, volumes, database algorithms, hardware, and the rest will evolve in such a way that our forced execution path will forever remain, if not absolutely the best, at least acceptable. In the particular case of our example, since nested loops using indexes are the most efficient choice, and because nested loops don't really benefit from parallelism, we are taking a rather small risk concerning the future evolution of our tables by ordering tables as we want them processed and instructing the optimizer to obey. Explicitly forcing the order followed to visit tables was the approach actually taken in this real-life case, which resulted in a query running in a little less than one second, with hardly fewer physical I/Os than before (2,340 versus 3,000—not too surprising since we start with a full scan of the very same table) but since we "suggested" a more efficient path, logical I/Os fell dramatically—to 16,500, down from over 3,000,000—with a noticeable result on the response time.

Remember that you should heavily document anything that forces the hand of the DBMS.

Explicitly forcing the order in which to visit tables by using optimizer directives is a heavy-handed approach. A more gentle way to obtain the same result from the optimizer, provided that it doesn't savagely edit our SQL clauses, may be to nest queries in the `from` clause, thus suggesting associations like parentheses would in a numerical expression:

```
select (select list)
from (select ttraoma.txnum,
             ttraoma.bkcod,
             ttraoma.trscod,
             ttraoma.pdtcod,
             ttraoma.objtyp,
             ...
       from ttraoma,
            tstg tstg_a,
            ttrcapp ttrcap_a
      where tstg_a.chncod = :7
        and tstg_a.stgnum = :8
        and tstg_a.risktyp = :4
        and ttraoma.txnum = tstg_a.txnum
        and ttrcap_a.colcod = :0
        and ttrcap_a.refcod = :5
        and ttraoma.trscod = ttrcap_a.valnumcod) a,
      ttex ttex_a,
      ttrcapp ttrcap_b,
      tbooks,
      topeoma,
      ttex ttex_b,
      ttypobj,
      tpdt,
      trgppdt
where ( a.txnum = topeoma.txnum )
 and ( a.bkcod = tbooks.trscod )
 and ( ttex_b.trscod = tbooks.permor )
 and ( ttex_a.nttcod = ttrcap_b.valnumcod )
 and ( ttypobj.objtyp = a.objtyp )
 and ( a.trscod = ttex_a.trscod )
 and ( ttrcap_b.colcod = :1 )
 and ( a.pdtcod = tpdt.pdtcod )
 and ( tpdt.risktyp = trgppdt.risktyp )
 and ( tpdt.riskflg = trgppdt.riskflg )
 and ( tpdt.pdtcod = trgppdt.pdtcod )
 and ( tpdt.risktyp = :2 )
 and ( tpdt.riskflg = :3 )
 and ( ttrcap_b.refcod = :6 )
```

It is often unnecessary to be very specific about the way we want a query to be executed and to multiply esoteric hints; the right initial guidance is usually enough to put an optimizer on the right track. Nested queries making explicit some table associations have the further advantage of being quite understandable to a qualified human reader.



A confused query can make the optimizer confused. Clarity and suggested joins are often enough to help the optimizer provide good performance.

Large Result Set

The situation of a large result set includes any result, irrespective of how it is obtained (with the exception of the explicit cases discussed here) that might be described as "large" or, in other words, a result set which it would be sensible to generate in a batch environment. When you are looking for a very large number of rows, even if this number looks like a fraction of the total number of rows stored in the tables involved in the query, conditions are probably not very selective and the DBMS engine must perform full scans, except perhaps in some very special cases of data warehousing, which are discussed in Chapter 10.

When a query returns tens of thousand of rows, whether as the final result or an intermediate step in a complex query, it is usually fairly pointless to look for a subtle use of indexes and fast jumps from an index to the table rows of interest. Rather, it's time to hammer the data remorselessly through full scans, usually associated with hash or merge joins. There must, however, be intelligence behind the brute force. We always must try to scan the objects, whether they are tables, indexes, or partitions of either tables or indexes, for which the ratio of data returned to data scanned is highest. We must scan objects for which filtering is the most coarse, because the best justification for the "effort" of scanning is to make it pay by a rich data harvest. A situation when a scan is unavoidable is the major exception to the rule of trying to get rid of unnecessary data as soon as possible; but we must fall back to the usual rule as soon as we are done with the unavoidable scans.

As ever, if we consider scanning rows of no interest to us as useless work, we must minimize the number of blocks we access. An approach often taken is to minimize accesses by hitting indexes rather than tables—even if the total volume of indexes is often bigger than the volume of data, each individual index is usually much smaller than its underlying table. Assuming that an index contains all the required information, scanning the index rather than the table makes a lot of sense. Implementation techniques such as adding columns to an index to avoid visiting the table can also show their worth.

Processing very large numbers of rows, whether you need to return them or simply have to check them, requires being very careful about what you do when you process each row. Calling a suboptimal, user-defined function, for instance, is not extremely important when you do it in the select list of a query that returns a small result set or when it comes as an additional criterion in a very selective `where` clause. But when you call such a function hundreds of thousands of times, the DBMS is no longer forgiving, and a slight awkwardness in the code can bring your server to its knees. This is a time for lean and mean code.

One key point to watch is the use of subqueries. Correlated subqueries are the death toll of performance when we are processing massive amounts of rows. When we can identify several subqueries within a query, we must let each of them operate on a distinct and "self-sufficient" subset, removing any dependence of one subquery on the result set of another. Dependencies between the various datasets separately obtained must be solved at the latest stage of query execution through hash joins or set operators.

Relying on parallelism may also be a good idea, but only when there are very few concurrently active sessions—typically in a batch job. Parallelism as it is implemented by a DBMS consists in splitting, when possible, one query into multiple subtasks, which are run in parallel and coordinated by a dedicated task. With a very high number of users, parallelism comes naturally with many similar tasks being executed concurrently, and adding DBMS parallelism to de facto parallelism often makes throughput worse rather than better. Generally speaking, processing very large volumes of information with a very high number of concurrent sessions qualifies as a situation in which the best you can aim for is an honorable fight and in which the solution is often to throw more hardware into the ring.

Response times are, lest we forget about the various waits for the availability of a resource in the course of processing, mostly dependent on the amount of data we have to browse through. But don't forget that, as you saw in Chapter 4, the subjective vision of an end user may be utterly different from a cold analysis of the size of the haystack: the only interest to the end user is the needle.

Self-Joins on One Table

In a correctly designed relational database (third normal form or above), all non-key columns are about the key, the whole key, and nothing but the key, to use an excellent and frequently quoted formula.* Each row is both logically consistent and distinct from all other rows in the same table. It is this design characteristic that enables join relationships to be established within the same table. You can therefore select in the same query different (not necessarily disjoint) sets of rows from the same table and join them as if those rows came from several different tables. In this section, I'll discuss the simple self-join and exclude the more complex examples of nested hierarchies that I discuss later in Chapter 7.

Self-joins—tables joined to themselves—are much more common than hierarchies. In some cases, it is simply because the data is seen in an identical way, but from two

* I have seen this elegant formula credited only once—to a 1983 paper by William Kent, available at *http://www.bkent.net*.

different angles; for instance, we can imagine that a query listing air flights would refer to the airports table twice, once to find the name of the departure airport, and once to find the name of the arrival airport. For example:

```
select f.flight_number,
       a.airport_name departure_airport,
       b.airport_name arrival_airport
from flights f,
     airports a,
     airports b
where f.dep_iata_code = a.iata_code
  and f.arr_iata_code = b.iata_code
```

In such a case, the usual rules apply: what matters is to ensure that highly efficient index access takes place. But what if the criteria are such that efficient access is not possible? The last thing we want is to do a first pass on the table, then a second one to pick up rows that were discarded during the first pass. In that case, what we should do is a single pass, collect all the rows of interest, and then use a construct such as the case statement to display separately rows from the two sets; I show examples of this "single-pass" approach in Chapter 11.

There are subtle cases that only superficially look like the airport case. Imagine that we store in some table cumulative values taken at regular intervals* and we want to display by how much the counter increased between two successive snapshots. In such a case, we have a relationship between two different rows in the same table, but instead of having a strong relationship coming from another table, such as the flights table that links the two instances of airports together, we have a weak, internal relationship: we define that two rows are related not because their keys are associated in another table, but because the timestamp of one row happens to be the timestamp which immediately follows the timestamp of another row.

For instance, if we assume that snapshots are taken every five minutes, with a timestamp expressed in seconds elapsed since a reference date, we might issue the following query:

```
select a.timestamp,
       a.statistic_id,
      (b.counter - a.counter)/5 hits_per_minute
from hit_counter a,
     hit_counter b
where b.timestamp = a.timestamp + 300
  and b.statistic_id = a.statistic_id
order by a.timestamp, a.statistic_id
```

* This is exactly what happens when you collect values from the V$ views in Oracle, which contain monitoring information.

There is a significant flaw in this script: if the second snapshot has not been taken *exactly* five minutes after the first one, down to the second, we may be unable to join the two rows. We may therefore choose to express the join condition as a range condition. For example:

```
select a.timestamp,
        a.statistic_id,
       (b.counter - a.counter) * 60 /
           (b.timestamp - a.timestamp) hits_per_minute
from  hit_counter a,
      hit_counter b
where b.timestamp between a.timestamp + 200
                      and a.timestamp + 400
  and b.statistic_id = a.statistic_id
order by a.timestamp, a.statistic_id
```

One side effect of this approach is the risk of having bigger data gaps than needed when, for one reason or another (such as a change in the sampling frequency), two successive records are no longer collected between 200 and 400 seconds of each other.

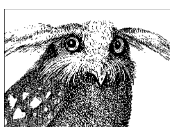
We may play it even safer and use an OLAP function that operates on windows of rows. It is indeed difficult to imagine something less relational in nature, but such a function can come in handy as the final shine on a query, and it can even make a noticeable difference in performance. Basically, OLAP functions allow the consideration of different subsets of the final result set, through the use of the `partition` clause. Sorts, sums, and other similar functions can be applied separately to these individual result subsets. We can use the `row_number( )` OLAP function to create one subset by `statistic_id`, and then assign to each different statistic successive integer numbers that increase as timestamps do. When these numbers are generated by the OLAP function, we can join on both `statistic_id` and two sequential numbers, as in the following example:

```
select a.timestamp,
       a.statistic_id,
       (b.counter - a.counter) * 60 /
              (b.timestamp - a.timestamp)
from (select timestamp,
             statistic_id,
             counter,
             row_number( ) over (partition by statistic_id
                                order by timestamp) rn
      from hit_counter) a,
      (select timestamp,
              statistic_id,
              counter,
              row_number( ) over (partition by statistic_id
                                 order by timestamp) rn
      from hit_counter) b
where b.rn = a.rn + 1
  and a.statistic_id = b.statistic_id
order by a.timestamp, a.statistic_id
```

We may even do better—about 25% faster than the previous query—if our DBMS implements, as Oracle does, a `lag(column_name, n)` OLAP function that returns the *n*th previous value for `column_name`, on the basis of the specified partitioning and ordering:

```
select timestamp,
       statistic_id,
       (counter - prev_counter) * 60 /
       (timestamp - prev_timestamp)
from (select timestamp,
             statistic_id,
             counter,
             lag(counter, 1) over (partition by statistic_id
                                   order by timestamp) prev_counter,
             lag(timestamp, 1) over (partition by statistic_id
                                     order by timestamp) prev_timestamp
     from hit_counter) a
order by a.timestamp, a.statistic_id
```

In many cases we don't have such symmetry in our data, as is shown by the flight example. Typically, a query looking for all the data associated with the smallest, or the largest, or the oldest, or the most recent value of a specific column, first needs to find the actual smallest, largest, oldest, or most recent value in the column used for filtering (this is the first pass, which compares rows), and then search the table again in a second pass, using as a search criterion the value identified in the first pass. The two passes can be made (at least superficially) into one through the use of OLAP functions that operate on sliding windows. Queries applied to data values associated to timestamps or dates are a special case of sufficient importance to deserve further discussion later in this chapter as the situation "Simple or Range Searching on Dates."



When multiple selection criteria are applied to different rows in the same table, functions that operate on sliding windows may be of assistance.

Result Set Obtained by Aggregation

An extremely common situation is the case in which the result set is a dynamically computed summary of the detailed data from one or more main tables. In other words, we are facing an *aggregation* of data. When data is aggregated, the size of the result set isn't dependent on the precision of the criteria that are provided, but merely on the cardinality of the columns that we group by. As in the first situation of the small result set obtained through precise criteria (and as you'll see again in Chapter 11), aggregate functions (or *aggregates*) are also often quite useful for obtaining in a single pass on the table results that are not truly aggregated but that would otherwise require self-joins and

multiple passes. In fact, the most interesting SQL uses of aggregates are not the cases in which sums or averages are an obvious part of the requirements, but situations in which a clever use of aggregates provides a pure SQL alternative to a procedural processing.

I stress in Chapter 2 that one of the keys to efficient SQL coding is a swashbuckling approach to code execution, testing for success after the deed rather than executing preliminary queries to check if, by chance, the really useful query we want to execute may fail: you cannot win a swimming race by tiptoeing carefully into the water. The other key point is to try to pack as much "action" as possible into an SQL query, and it is in respect to this second key point that aggregate functions can be particularly useful.

Much of the difficulty of good SQL programming lies in seeing how a problem can translate, not into a succession of queries to a database, but into very few queries. When, in a program, you need a lot of intermediate variables to hold values you get from the database before reinjecting them into the database as input to other queries, and if you perform against those variables nothing but very simple tests, you can bet that you have the algorithm wrong. And it is a striking feature of poorly written SQL programs to see the high number of lines of code outside of SQL queries that are simply devoted to summing up, multiplying, dividing, and subtracting inside loops what is painfully returned from the database. This is a totally useless and utterly inefficient job: we have SQL aggregate functions for that sort of work.

> **NOTE**
>
> **Aggregate functions are very useful tools for solving SQL problems (and we will revisit them in Chapter 11, when I talk about stratagems); however, it often appears to me that developers use only the least interesting aggregate function of all, namely `count()`, the real usefulness of which is often, at best, dubious in most programs.**

Chapter 2 shows that using `count(*)` to decide whether to update an existing row or insert a new one is wasteful. You can misuse `count(*)` in reports as well. A test for existence is sometimes implemented as a mock-Boolean value such as:

```
case count(*)
when 0 then 'N'
else 'Y'
end
```

Such an implementation gets, when rows are found, all the rows that match the condition in order to obtain a precise count, whereas finding only one is enough to decide whether `Y` or `N` must be displayed. You can usually write a much more effective statement by using a construct that either limits the number of rows returned or tests for existence, effectively stopping processing as soon as a row that matches the condition is found.

But when the question at hand is about the most, the least, the greatest, or even the first or the last, it is likely that aggregate functions (possibly used as OLAP functions) will provide the best answer. If you believe that aggregate functions should be used only when counts, sums, maxima, minima, or averages are explicitly required, then you risk seriously underusing them.

Interestingly, aggregate functions are extremely narrow in scope. If you exclude the computation of maximum and minimum values, the only thing they can really do is simple arithmetic; a `count( )` is nothing more than adding 1s for each row encountered. Similarly, the computation of `avg( )` is just, on one hand, adding up the values in the column it is applied to and, on the other hand, adding 1s, and then dividing.

But it is sometimes wonderful what you can do with simple sums. If you're mathematically inclined, you'll remember how easily you can switch between sums and products by the magic of logarithms and power functions. And if you're logically inclined, you know well how much `OR` owes to sums and `AND` to products.

I'll show the power of aggregation with a simple example. Assume that we have a number of shipments to make and that each shipment is made of a number of different orders, each of which has to be separately prepared; it is only when each order in a shipment is complete that the shipment itself is ready. The problem is how to detect when all the orders comprising a shipment are complete.

As is so often the case, there are several ways to determine the shipments that are complete. The worst approach would probably be to loop on all shipments, inside a second loop on each shipment count how many orders have `N` as value for the `order_complete` column, and return shipment IDs for which the count is `0`. A much better solution would be to recognize the test on the nonexistence of an `N` value for what it is, and use a subquery, correlated or uncorrelated; for instance:

```
select shipment_id
from shipments
where not exists (select null from orders
                  where order_complete = 'N'
                    and orders.shipment_id = shipments.shipment_id)
```

This approach is pretty bad if we have no other condition on the `shipments` table. Following is a query that may be much more efficient if we have a large `shipments` table and few uncompleted orders:

```
select shipment_id
from shipments
where shipment_id not in (select shipment_id
                          from orders
                          where order_complete = 'N')
```

This query can also be expressed as follows, as a variant that an optimizer may like better but that wants an index on the column shipment_id of the table orders:

```
select shipments.shipment_id
from shipments
    left outer join orders
          on orders.shipment_id = shipments.shipment_id
          and orders.order_complete = 'N'
where orders.shipment_id is null
```

Another alternative is a massive set operation that will operate on the primary key index of shipments on one hand, and that will perform a full table scan of orders on the other hand:

```
select shipment_id
from shipments
except
select shipment_id
from orders
where order_complete = 'N'
```

Be aware that not all DBMS implement the except operator, sometimes known as minus.

But there is still another way to express our query. What we are doing, basically, is to return the identifiers of all shipments for which a logical AND operation on all orders which have been completed returns TRUE. This kind of operation happens to be quite common in the real world. As hinted previously, there is a very strong link between AND and multiplication, and between OR and addition. The key is to convert flags such as Y and N to 0s and 1s. This conversion is a trivial operation with the case construct. To get just order_complete as a 0 or 1 value, we can write:

```
select shipment_id,
       case when order_complete = 'Y' then 1
                                      else 0
       end flag
from orders
```

So far, so good. If we always had a fixed number of orders per shipment, it would be easy to sum the calculated column and check if the result is the number of orders we expect. However, what we want here is to multiply the flag values per shipment and check whether the result is 0 or 1. That approach works, because even one incomplete order, represented by a 0, will cause the final result of all the multiplication to also be 0. The multiplication can be done with the help of logarithms (although 0s are not the easiest values to handle with logarithms). But in this particular case, our task is even easier.

What we want are the shipments for which the first order is completed and the second order is completed and…the *n*th order is completed. Logic and the laws of de Morgan* tell us that this is exactly the same as stating that we *do not* have (first order not completed or second order not completed…or *n*th order not completed). Since their kinship to sums makes ORs much easier to process with aggregates than ANDs, checking that a list of conditions linked by OR is false is much easier than checking that a list of conditions linked by AND is true. What we must consider as our true predicate is "the order is not completed" rather than the reverse, and convert the order_complete flag to 1 if it is N, and 0 if it is Y. In that way, we can easily check that we have 0s (or yeses) everywhere by summing up values—if the sum is 0, then all orders are completed; otherwise, we are at various stages of incompletion.

Therefore we can also express our query as:

```
select shipment_id
from (select shipment_id,
             case when order_complete = 'N' then 1
                                                 else 0
             end flag
      from orders) s
group by shipment_id
having sum(flag) =0
```

And it can be expressed in an even more concise way as:

```
select shipment_id
from orders
group by shipment_id
having sum(case when order_complete = 'N' then 1
                                                else 0
           end) =0
```

There is another way to write this query that is even simpler, using another aggregate function, and without any need to convert flag values. Noticing that Y is, from an alphabetical point of view, greater than N, it is not too difficult to infer that if all values are Y then the minimum will necessarily be Y too. Hence:

```
select shipment_id
from orders
group by shipment_id
having min(order_complete) = 'Y'
```

* The India-born Augustus de Morgan (1806–1871) was a British mathematician who contributed to many areas of mathematics, but most significantly to the field of logic. The de Morgan laws state that the complement of the intersection of any number of sets equals the union of their complements and that the complement of the union of any number of sets equals the intersection of their complements. If you remember that SQL is about sets, and that negating a condition returns the complement of the result set returned by the initial condition (if you have no null values), you'll understand why these laws are particularly useful to the SQL practitioner.

This approach of depending on `Y` to be greater than `N` may not be as well grounded mathematically as the flag-to-number conversion, but it is just as efficient.

Of course we must see how the query that uses a `group by` and a condition on the minimum value for `order_complete` compares to the other versions that use subqueries or `except` instead of an aggregate function. What we can say is that it has to fully sort the `orders` table to aggregate the values and check whether the sum is or is not 0. As I've specified the problem, this solution involving a non-trivial use of an aggregate function is likely to be faster than the other queries, which hit two tables (`shipments` and `orders`), and usually less efficiently.

I have made an extensive use of the `having` clause in the previous examples. As already mentioned in Chapter 4, a common example of careless SQL statements involves the use of the `having` clause in aggregate statements. Such an example is illustrated in the following (Oracle) query, which attempts to obtain the sales per product per week during the past month:

```
select product_id,
       trunc(sale_date, 'WEEK'),
       sum(sold_qty)
from sales_history
group by product_id, trunc(sale_date, 'WEEK')
having trunc(sale_date, 'WEEK') >= add_month(sysdate, -1)
```

The mistake here is that the condition expressed in the `having` clause doesn't depend on the aggregate. As a result, the DBMS has to process all of the data in `sales_history`, sorting it and aggregating against each row, before filtering out ancient figures as the last step before returning the required rows. This is the kind of mistake that can go unnoticed until `sales_history` grows really big. The proper approach is, of course, to put the condition in a `where` clause, ensuring that the filtering occurs at an early stage and that we are working afterwards on a much reduced set of data.

I should note that when we apply criteria to views, which are aggregated results, we may encounter exactly the same problem if the optimizer is not smart enough to reinject our filter *before* aggregation.

You can have slightly more subtle variants of a filter applied later than it should be. For instance:

```
select customer_id
from orders
where order_date < add_months(sysdate, -1)
group by customer_id
having sum(amount) > 0
```

In this query, the following condition looks at first glance like a reasonable use of `having`:

```
having sum(amount) > 0
```

However, this use of having does not really make sense if amount is always a positive quantity or zero. In that event, we might be better using the following condition:

```
where amount > 0
```

We have two possibilities here. Either we keep the group by:

```
select customer_id
from orders
where order_date < add_months(sysdate, -1)
  and amount > 0
group by customer_id
```

or we notice that group by is no longer required to compute any aggregate and replace it with a distinct that in this case performs the same task of sorting and eliminating duplicates:

```
select distinct customer_id
from orders
where order_date < add_months(sysdate, -1)
  and amount > 0
```

Placing the condition in the where clause allows unwanted rows to be filtered at an earlier stage, and therefore more effectively.

Aggregate as little data as you can.

Simple or Range Searching on Dates

Among search criteria, dates (and times) hold a particular place that is all their own. Dates are extremely common, and more likely than other types of data to be subjected to range conditions, whether they are bounded ("between this date and that date") or only partially bounded ("before this date"). Very often, and what this situation describes, the result set is derived from searches against date values referenced to the current date (e.g., "six months earlier than the current date," etc.).

The example in the previous section, "Result Set Obtained by Aggregation," refers to a sales_history table; our condition was on an amount, but it is much more common with this type of table to have conditions on date, especially to get a snapshot of the data either at a given date or between two dates. When you are looking for a value on a given date in a table containing historical data, you must pay particular attention to the way you identify current data. The way you handle current data may happen to be a special case of data predicated on an aggregate condition.

I have already pointed out in Chapter 1 that the design of a table destined to store historical data is a tricky affair and that there is no easy, ready-made solution. Much depends on what you plan to do with your data, whether you are primarily interested in current values or in values as of a particular date. The best solution also depends on how fast data becomes outdated. If you are a retailer and wish to keep track of the wares you sell, it is likely that, unless your country suffers severe hyper-inflation, the rate of change of your prices will be pretty slow. The rate of change will be higher, possibly much higher, if you are recording the price of financial instruments or monitoring network traffic.

To a large extent, what matters most with history tables is how much historical data you keep on average per item: you may store a lot of historical information for very few items, or have few historical records for a very large number of items, or anything in between. The point here is that the selectivity of any item depends on the number of items being tracked, the frequency of sampling (e.g., either once per day or every change during the day), and the total time period over which the tracking takes place (infinite, purely annual, etc.). We shall therefore first consider the case when we have many items with few historical values, then the opposite case of few items with a rich history, and then, finally, the problem of how to represent the current value.

Many Items, Few Historical Values

If we don't keep an enormous amount of historical data per item, the identification of an item is quite selective by itself. Specifying the item under study restricts our "working set" to just a few historical rows, and it then becomes fairly easy to identify the value at a given reference date (the current or a previous date) as the value recorded at the closest date prior to the reference date. In this case, we are dealing once again with aggregate values.

Unless some artificial, surrogate key has been created (and this is a case where there is no real need for a surrogate key), the primary key will generally be a composite key on the identifier of items (`item_id`) and the date associated with the historical value (`record_date`). We mostly have two ways of identifying the rows that store values that were current as of a given reference date: subqueries and OLAP functions.

Using subqueries

If we are looking for the value of one particular item as of a given date, then the situation is relatively simple. In fact, the situation is deceptively simple, and you'll often encounter a reference to the value that was current for a given item at a given date coded as:

```
select whatever
from hist_data as outer
where outer.item_id = somevalue
  and outer.record_date = (select max(inner.record_date)
                           from hist_data as inner
                           where inner.item_id = outer.item_id
                             and inner.record_date <= reference_date)
```

It is interesting to see what the consequences of this type of construct suggest in terms of the execution path. First of all, the inner query is correlated to the outer one, since the inner query references the `item_id` of the current row returned by the outer query. Our starting point is therefore the outer query.

Logically, from a theoretical point of view, the order of the columns in a composite primary key shouldn't matter much. In practice, it is critical. If we have made the mistake of defining the primary key as (`record_date, item_id`) instead of (`item_id, record_date`), we desperately need an additional index on `item_id` for the inner query; otherwise, we will be unable to efficiently descend the tree-structured index. And we know how costly each additional index can be.

Starting with our outer query and finding the various rows that store the history of `item_id`, we will then use the current value of `item_id` to execute the subquery each time. Wait! This inner query depends only on `item_id`, which is, by definition, the same for all the rows we check! The logical conclusion: we are going to execute exactly the same query, returning exactly the same result for each historical row for `item_id`. Will the optimizer notice that the query always returns the same value? The answer may vary. It is better not to take the chance.

There is no point in using a correlated subquery if it always returns the same value for all the rows for which it is evaluated. We can easily uncorrelate it:

```
select whatever
from hist_data as outer
where outer.item_id = somevalue
  and outer.record_date = (select max(inner.record_date)
                           from hist_data as inner
                           where inner.item_id = somevalue
                             and inner.record_date <= reference_date)
```

Now the subquery can be executed without accessing the table: it finds everything it requires inside the primary key index.

It may be a matter of personal taste, but a construct that emphasizes the primary key is arguably preferable to the preceding approach, if the DBMS allows comparing several columns to the output of a subquery (a feature that isn't supported by all products):

```
select whatever
from hist_data as outer
where (outer.item_id, outer.record_date) in
                          (select inner.item_id, max(inner.record_date)
                           from hist_data as inner
                           where inner.item_id = somevalue
                             and inner.record_date <= reference_date
                           group by inner.item_id)
```

The choice of a subquery that precisely returns the columns matching a composite primary key is not totally gratuitous. If we now need to return values for a list of items,

possibly the result of another subquery, this version of the query naturally suggests a good execution path. Replace *somevalue* in the inner query by an `in()` list or a subquery, and the overall query will go on performing efficiently under the very same assumptions that each item has a relatively short history. We have also replaced the equality condition by an `in` clause: in most cases the behavior will be exactly the same. As usual, it is at the fringes that you encounter differences. What happens if, for instance, the user mistyped the identification of the item? The `in()` will return that no data was found, while the equality may return a different error.

Using OLAP functions

With databases, OLAP functions such as `row_number()` that we have already used in the *self-joins* situation can provide a satisfactory and sometimes even a more efficient way to answer the same question "what was the current value for one particular item at a given date?" (remember that OLAP functionality does, however, introduce a distinctly non-relational aspect to the proceedings*).

> **NOTE**
>
> **OLAP functions belong to the non-relational layer of SQL. They represent the final, or almost final, step in query execution, since they have to operate on the post-retrieval result set after the filtering has completed.**

With a function such as `row_number()` we can assign a degree of freshness (one meaning most recent) to the data by ranking on date:

```
select row_number() over (partition by item_id
                          order by record_date desc) as freshness,
       whatever
from hist_data
where item_id = somevalue
  and record_date <= reference_date
```

Selecting the freshest data is then simply a matter of only retaining the rows with a value of one for `freshness`:

```
select x.<suitable_columns>
from (select row_number() over (partition by item_id
                                order by record_date desc) as freshness,
             whatever
      from hist_data
      where item_id = somevalue
        and record_date <= reference_date) as x
where x.freshness = 1
```

* ...even if the term OLAP was coined by Dr. E.F. Codd himself in a 1993 paper.

In theory, there should be hardly any difference between the OLAP function approach and the use of subqueries. In practice, an OLAP function hits the table only once, even if the usual sorting happens behind the scene. There is no need for additional access to the table, even a fast one that uses the primary key. The OLAP function approach may therefore be faster (albeit only slightly so).

Many Historical Values Per Item

The picture may be different when we have a very large number of historical values—for instance, a monitoring system in which metrics are collected at a rather high frequency. The difficulty here lies in the fact that all the intermediate sorting required for identifying the value at or nearest a given date may have to operate on a really large amount of data.

Sorting is a costly operation. If we apply the principles of Chapter 4, the only way we have to reduce the thickness of the non-relational layer is by doing a bit more work at the relational level—by increasing the amount of filtering. In such a case, it is very important to *narrow* our scope by bracketing the date (or time) more precisely for which we want the data. If we only provide an upper boundary, then we shall have to scan and sort the full history since the beginning of ages. If data is collected at a high frequency, it is then reasonable to give a lower limit. If we succeed in restraining the "working set" of rows to a manageable size, we are back to the case in which we have relatively few historical values per item. If specifying both an upper boundary (such as the current date) *and* a lower boundary isn't an option, our only hope is in partitioning per item; operating on a single partition will take us closer to the "large result set" situation.

Current Values

When we are predominantly interested in the most recent or current values, it is very tempting to design a way to avoid either the nested subquery or the OLAP function (which both entail a sort), and hit the proper values directly. We mentioned in Chapter 1 that one solution to this problem is to associate each value with some "end date"—the kind of "best before" you find on your cereal boxes—and to say that for current values that end date is far, far away into the future (let's say December 31, 2999). We also mentioned that there were some practical issues associated with such a design and the time has now come to explore these issues.

With a fixed date, it certainly becomes extremely easy to find the current value. Our query simply becomes:

```
select whatever
from hist_data
where item_id = somevalue
  and record_date = fixed_date_in_the future
```

We then hit the right row, spot on, through the primary key. And of course, nothing prevents us from using either subqueries or OLAP functions whenever we need to refer to a date other than the current one. There are, however, two main drawbacks to this approach—an obvious one and a more subtle one:

- The obvious drawback is that each insertion of a new historical value will first require updating what used to be the current value with, for example, today's date, to mean that it used to be the current value until today. Then the new value can be inserted with the later date, to mean that it is now the current value until further notice. This process leads to double the amount of work, which is bad enough. Moreover, since in the relational theory the primary key is what identifies a row, the combination (`item_id`, `record_date`) can be unique but cannot be the primary key since we have to partially update it. We therefore need a surrogate key to be referenced by foreign keys (identity column or sequence), which further complicates programs. The trouble with big historical tables is that usually, to grow that big, they also undergo a high rate of insertion. Does the benefit of faster querying offset the disadvantage of inserting more slowly? It's difficult to say, but definitely a question worth asking.
- The subtle drawback has to do with the optimizer. The optimizer relies on statistics that may be of variable detail, with the result that it is not unusual for it to check the lowest and highest value in a column to try to assess the spread of values. Let us say that our historical table contains values since January 1, 2000. Our data will therefore consist of perhaps 99.9% historical data, spread over several, but relatively few, years, and 0.1% of current data, officially as of December 31, 2999. The view of the optimizer will be of data spread over one millennium. This skewness on the part of the optimizer view of the data range is because it is being misled by the upper boundary date in the query (`"and record_date =` *`fixed_date_in_the future`*`"`). The problem is then that when you search for something other than current values (for instance if you want to collect variations over time for statistical purposes), the optimizer may well incorrectly decide that since you are accessing such a tiny fraction of the millennium, then using indexes is the thing to do, but what you really need is to scan the data. Skewness can lead to totally wrong execution plans, which are not easy to correct.

You must understand your data and your data distributions if you are to understand how the optimizer views your system.

Result Set Predicated on Absence of Data

It is a common occurrence to look for rows in one table for which there is no matching data in another table—usually for identifying exceptions. There are two solutions people most often think of when having to deal with this type of problem: using either `not in ()`

with an uncorrelated subquery or `not exists ()` with a correlated subquery. Popular wisdom says that you should use `not exists`. Since a correlated subquery is efficient when used to mop up after the bulk of irrelevant data has been cleared out by efficient filtering, popular wisdom has it right when the subquery comes after the strong forces of efficient search criteria, and totally wrong when the subquery happens to be the only criterion.

One sometimes encounters more exotic solutions to the problem of finding rows in one table for which there is no matching data in another. The following example is a real-life case that monitoring revealed to be one of the costliest queries performed against a database (note that question marks are placeholders, or *bind variables*, for constant values that are passed to the query on successive executions):

```
insert into ttmpout(custcode,
                    suistrcod,
                    cempdtcod,
                    bkgareacod,
                    mgtareacod,
                    risktyp,
                    riskflg,
                    usr,
                    seq,
                    country,
                    rating,
                    sigsecsui)
select distinct custcode,
                ?,
                ?,
                ?,
                mgtareacod,
                ?,
                ?,
                usr,
                seq,
               country,
               rating,
               sigsecsui
from ttmpout a
where a.seq = ?
  and 0 = (select count(*)
           from ttmpout b
           where b.suistrcod = ?
             and b.cempdtcod = ?
             and b.bkgareacod = ?
             and b.risktyp = ?
             and b.riskflg = ?
             and b.seq = ?)
```

This example must not be understood as an implicit unconditional endorsement of temporary tables! As a passing remark, I suspect that the `insert` statement was part of a loop. Proper performance improvement would probably be achieved by removing the loop.

An insertion into a table based on a `select` on the very same table as in the current example is a particular and yet not uncommon case of self-reference, an insertion derived from existing rows and conditional on the absence of the row to be created.

Using `count(*)` to test whether something exists or doesn't exist is a bad idea: to count, the DBMS must search and find all rows that match. We should use `exists` in such a case, which stops as soon as the first match is encountered. Arguably, it does not make much difference if the filtering criterion happens to be the primary key. But it may make a very significant difference in other cases—and anyway from a semantic point of view there is no reason to say this:

```
and 0 = (select count(*) ...)
```

when we mean this:

```
and not exists (select 1 ...)
```

If we use `count(*)` as a test for existence, we may be lucky enough to benefit from the "invisible hand" of a smart optimizer, which will turn our query into something more suitable. But this will not necessarily be the case, and it will never be the case if the rows are counted into some variable as an independent step, because then even the smartest of optimizers cannot guess for which purpose we are counting: the result of the `count()` could be a critical value that absolutely has to be displayed to the end user!

In such a case when we want to create new, unique rows derived from rows already present in the table, however, the right construct to use is probably a set operator such as `except` (sometimes known as `minus`).

```
insert into ttmpout(custcode,
                    suistrcod,
                    cempdtcod,
                    bkgareacod,
                    mgtareacod,
                    risktyp,
                    riskflg,
                    usr,
                    seq,
                    country,
                    rating,
                    sigsecsui)
(select custcode,
        ?,
        ?,
        ?,
        mgtareacod,
        ?,
        ?,
        usr,
        seq,
```

```
               country,
               rating,
               sigsecsui
        from ttmpout
        where seq = ?
        except
        select custcode,
               ?,
               ?,
               ?,
               mgtareacod,
               ?,
               ?,
               usr,
               seq,
               country,
               rating,
               sigsecsui
        from ttmpout
        where suistrcod = ?
          and cempdtcod = ?
          and bkgareacod = ?
          and risktyp = ?
          and riskflg = ?
          and seq = ?)
```

The big advantage of set operators is that they totally break the time frame imposed by subqueries, whether they are correlated or uncorrelated. What does *breaking the time frame* mean? When you have correlated subqueries, you must run the outer query, and then you must execute the inner query for each row that passes through all other filtering criteria. Both queries are extremely dependent on each other, since the outer query feeds the inner one.

The picture is slightly brighter with uncorrelated subqueries, but not yet totally rosy: the inner query must be executed, and in fact completed, before the outer query can step in and gather steam (something similar occurs even if the optimizer chooses to execute the global query as a hash join, which is the smart thing for it to do, because to execute a hash join, the SQL engine first has to scan one of the tables involved to build a hash array).

With set operators, on the contrary, whether they are `union`, `intersect` or `except`, none of the components in the query depends on any other. As a result, the different parts of the query can run in parallel. Of course, parallelism is of hardly any benefit if one of the steps is very slow while all the others are very fast; and it will be of no benefit at all if much of the work in one part is strictly identical to the work in another part, because then you are duplicating, rather than sharing, the work between processes. But in a favorable case, it is much more efficient to have all parts run in parallel before the final step, which combines the partial result sets—divide and rule.

There is an additional snag with using set operators: they require each part of the query to return compatible columns—an identical number of columns of identical types. A case such as the following (another real-life case, coming from a billing program) is typically unsuited to set operators:

```
select whatever, sum(d.tax)
from invoice_detail d,
     invoice_extractor e
where (e.pga_status = 0
       or e.rd_status = 0)
 and suitable_join_condition
 and (d.type_code in (3, 7, 2)
      or (d.type_code = 4
          and d.subtype_code not in
               (select trans_code
                from trans_description
                where trans_category in (6, 7))))
group by what_is_required
having sum(d.tax) != 0
```

I am always fascinated by the final condition:

```
sum(d.tax) != 0
```

and the way it evokes yellow brick roads and fantasy worlds where taxes are negative. A condition such as:

```
and d.tax > 0
```

might have been more appropriate in the `where` clause, as already demonstrated.

In such a case a set operator would be rather awkward, since we would have to hit the `invoice_detail` table—as we can guess, not a lightweight table—several times. However, depending on the selectivity of the various criteria provided, typically if `type_code=4` is a rare and therefore selective attribute condition, an `exists` might be more appropriate than a `not in ()`. If, however, `trans_description` happens to be, at least relatively, a small table, then there is no doubt that trying to improve the query by playing on the existence test alone is a dead end.

Another interesting way to express nonexistence—and often quite an efficient one—is to use outer joins. The purpose of outer joins is basically to return, in a join, all information from one table, including rows for which no match is found in the joined table. As it happens, when we are looking for data that has no match in another table, it is precisely these rows that are of interest to us. How can we identify them? By checking the joined table columns: when there is no match, they are replaced with null values.

Something such as:

```
select whatever
from invoice_detail
where type_code = 4
```

```
    and subtype_code not in
                (select trans_code
                 from trans_description
                 where trans_category in (6, 7))
```

can therefore be rewritten:

```
select whatever
from invoice_detail
     outer join trans_description
             on trans_description.trans_category in (6, 7)
            and trans_description.trans_code = invoice_detail.subtype_code
where trans_description.trans_code is null
```

I have purposely included the condition on trans_category in the join clause. Whether it should rightly appear in this clause or in the where clause is debatable but, in fact, filtering before the join or after the join is result-neutral (of course, from a performance point of view, it can make a difference, depending on the relative selectivity of this condition and of the join condition itself). However, we have no such latitude with the condition on the null value, since this is something that can only be checked *after* the join.

Apart from the fact that the outer join may in some cases require a distinct, in practice there should be very little difference between checking the absence of data through an outer join or a not in () uncorrelated subquery, since the column which is used for the join happens to be the very same column that is compared to the result set of the subquery. But SQL is famous for being a language in which the manner of the query expression often has a very real effect on the pattern of execution, even if the theory says otherwise. It all depends on the degree of sophistication of the optimizer, and whether it processes both types of queries in a similar way or not. In other words, SQL is not a truly declarative language, even if the enhancement of optimizers with each new version slowly improves its reliability.

Before closing this topic, watch out for the perennial SQL party-poopers—null values. Although in an in () subquery a null value joining the flow of non-null values does not bother the outer query, with a not in () subquery, any null value returned by the inner query causes the not in () condition to be evaluated as false. It does not cost much to ensure that a subquery returns no null value—and doing so will save you a lot of grief.

Data sets can be compared using various techniques, but outer joins and set operators are likely to be efficient.