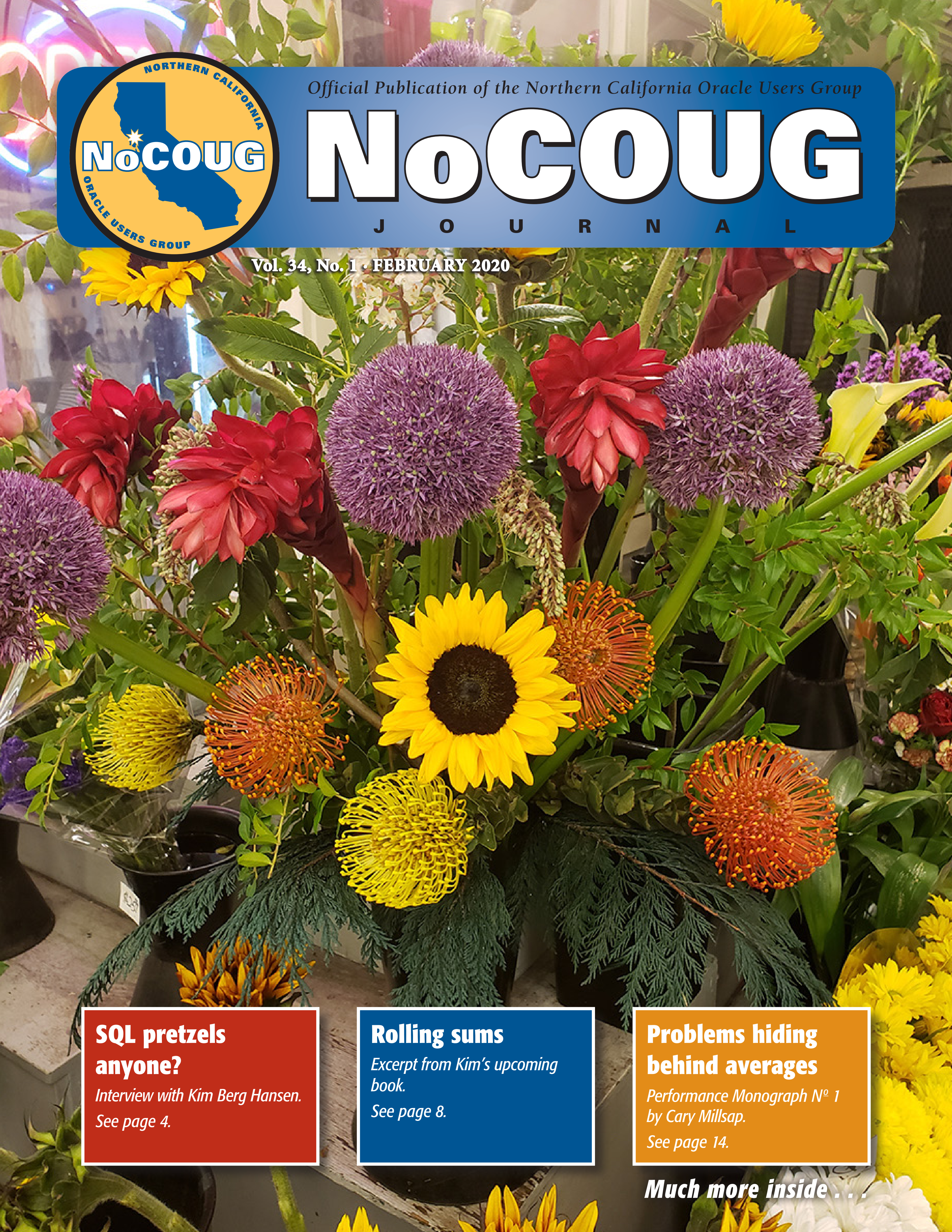
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 34, No. 1 · FEBRUARY 2020



SQL pretzels anyone?

Interview with Kim Berg Hansen.

See page 4.

Rolling sums

Excerpt from Kim's upcoming book.

See page 8.

Problems hiding behind averages

Performance Monograph Nº 1 by Cary Millsap.

See page 14.

Much more inside . . .

Amazon Aurora

Performance and availability of commercial-grade databases at 1/10th the cost.

Learn more: aws.amazon.com/aurora

aws

Professionals at Work

First there are the IT professionals who write for the *Journal*. A very special mention goes to Brian Hitchcock, who has written dozens of book reviews over a 12-year period.

Next, the *Journal* is professionally copyedited and proofread by veteran copyeditor Karen Mead of Creative Solutions. Karen polishes phrasing and calls out misused words (such as "reminiscences" instead of "reminisces"). She dots every i, crosses every t, checks every quote, and verifies every URL.

Then, the *Journal* is expertly designed by graphics duo Kenneth Lockerbie and Richard Repas of San Francisco-based Giraffex.

And, finally, the *Journal* is printed and shipped to us. This is the 133rd issue of the *NoCOUG Journal*. Enjoy! ▲

Table of Contents

Publication Notices and Submission Format

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

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

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

2020 NoCOUG Board

ADVERTISING RATES

The *NoCOUG Journal* is published quarterly.

Size	Per Issue	Per Year
Quarter Page	$125	$400
Half Page	$250	$800
Full Page	$500	$1,600
Inside Cover	$750	$2,400

Personnel recruitment ads are not accepted.

journal@nocoug.org

SQL by Example

with Kim Berg Hansen

Kim Berg Hansen

Your book *Practical Oracle SQL* will be published soon. Yet Another SQL Book?

I've been working with Oracle SQL since 2000, and if someone asks me where I learned SQL, the answer is many places: books by Tom Kyte (my guru) and others, the *SQL Reference Manual* (that I use daily), conference presentations by experienced developers, blogs, googling, and much more. But even all of that would not help if I didn't simultaneously simply try writing SQL myself, see where I went wrong, and then try again and again and again.

One thing I have noticed in my learning process is that almost all teaching examples are nicely short and sweet in order to facilitate understanding. This is fine as such, but it also sometimes means that it can be harder to relate to daily work. Making the leap from having understood a small example to applying the same technique in a larger and real context is an acquired skill. Personally I think I achieved this skill in university, where we students had to research and learn much more by ourselves as opposed to the one-way teacher-to-student schooling I had before entering university.

When I went to work, I had the good fortune of working for 16 years at a retail company where the philosophy was *never* to adapt business practice to whatever the software was capable of but instead *always* to customize the software to make the daily business go smarter and smoother. We always went by the philosophy "of course it is possible to solve; we just need to figure out how." In this atmosphere I had plenty of practical real tasks to practice on, trying out SQL and changing it piece by piece until I had something that solved the task at hand. It was a time when I improved my SQL daily by using the skills of researching, finding examples, learning from them, and then applying them to real tasks.

When I have presented at conferences about some of these solutions that I developed during those happy years, I have several times had audience members approach me afterwards, telling me that suddenly they saw the light and understood, for example, how analytic functions could help in their work. Until then, they had seen it as some SQL extension that was smart and fancy, but they couldn't relate it to their own tasks that they had to solve.

And so it dawned on me that not everybody learns things the way I do. Quite a few would watch a presentation or read a book chapter or a blog post on how to use analytic functions to get a running total of salaries in SCOTT.EMP—but when they got back to work it didn't help them to figure out that this technique actually can help them to create a warehouse picking list by First-In-First-Out (FIFO) in a single SQL statement. The leap from the simple SCOTT.EMP example to real life was too great.

The majority of SQL books that I have seen fall into two categories (grossly oversimplified, I know, but it makes my point):

- *Definitive reference guides.*
  Books that try to cover every single bit of syntax in the SQL language with new editions for every new database version. Books that potentially can replace the official *SQL Reference Manual*, just more user friendly and going about showing SQL in a way that's more conducive to learning.

- *SQL 101 (and maybe 102 and 103) books.*
  Books that start by assuming you know nothing of SQL and start with SELECT 'hello, world' FROM DUAL. Books that really teach SQL in a structured way from the basics and sometimes also move on to somewhat more advanced topics.

Both categories of books most often use simple examples, just like presentations and blog posts. A reader can start with a *SQL 101*–style book and get fairly confident in using basic SQL as it was in the SQL-92 standard. The reader might then buy a *reference*-style book, see some more advanced things, and then get lost trying to apply the advanced concepts to real life.

What I felt was missing was a book that assumes the reader already knows SQL basics and is ready to learn something more advanced than SQL-92 but needs to learn it in a form that relates

"Almost all teaching examples are nicely short and sweet in order to facilitate understanding. This is fine as such, but it also sometimes means that it can be harder to relate to daily work. Making the leap from having understood a small example to applying the same technique in a larger and real context is an acquired skill."

"I see quite a few developers chugging along with SQL-92 and never progressing to using analytic functions, let alone row pattern matching and other modern additions to the SQL language, and I want to push those developers further into modern SQL."

to the reader's daily activities. Unfortunately I see quite a few developers chugging along with SQL-92 and never progressing to using analytic functions, let alone row pattern matching and other modern additions to the SQL language, and I want to push those developers further into modern SQL.

So having had success with developers understanding more advanced concepts by *presenting* about the real solutions I had developed, I decided to use the same teaching concept in book form.

Okay, so a *different* way. Exactly what do you mean?

You'll find loads of examples, based on a fictitious company called *Good Beer Trading Co.* (As a card-carrying member of Danish Beer Enthusiasts Association I couldn't resist the temptation to use beer in my example schema, but you can use the book even if you don't drink alcohol—there is no bottle included when you buy the book . . .) For this company I have created a schema with tables and data for inventory, orders, purchases, webshop, and more; all such tables might have been part of the application of a real company.

In each chapter I have a task as it might have been put to the application developers of the *Good Beer Trading Co*; I then show how to solve that task with SQL, explaining in steps how I create that SQL, starting small and building on it until I have a working statement that most often does not fit on a single PowerPoint slide. The statements I demonstrate are not trivial examples, but they look more like something a developer might have to personally develop on the job.

What are some of those tasks?

As I said, all chapters (except Chapter 6) have an objective of showing a task that is relevant for real application development. Although the specific examples are shown from the viewpoint of the fictitious *Good Beer Trading Co*, the techniques can be applied to many other applications. The chapters are divided into three parts based on the SQL technique used to solve the task.

The first part deals with solutions that use a variety of SQL constructs, covering many techniques: inline view correlation, set operations, with clause and with clause functions, recursive subquery factoring and model clause iteration, pivoting and unpivoting, as well as splitting and creating delimited text.

Solving tasks with analytic functions is the topic of the second part. The focus is on demonstrating practical tasks that can be solved extremely efficiently, walking through using analytic functions for tasks such as top-n questions, warehouse picking with rolling sums, analyzing activity logs, and two types of forecasting.

Finally, the third part covers using match_recognize not only for the row pattern matching for which it was designed but also for tasks that might not at first glance seem appropriate. The covered tasks include finding up-and-down patterns, grouping consecutive data, merging date ranges, finding abnormal peaks, bin fitting, and tree branch calculations.

Why should database developers bother? Can't they get by with basic SQL and an ORM framework?

SQL is evolving like any other programming language. The SQL language you'll find in the Oracle versions of this millennium includes a whole lot more than SQL-92. Listen to Andy Mendelsohn talking about multi-model converged database architecture, and you'll realize that you can work with relational data, JSON, XML, NoSQL, and big data—basically you-name-it—all that you can work with using SQL and PL/SQL.

That's a much-too-huge topic to cover in a single book. But even just sticking to the good old relational world, SQL has grown over the years with functionality that makes it possible to create much more performant SQL, solving much more complex problems than SQL-92 can manage.

Take, for example, analytic functions that have been my favorite since I started working with Oracle SQL. I saw a quote (source unknown) from a conference presentation: "If you write on your CV that you know SQL, but you do not use analytic functions, then you are lying." Personally I would just hate to solve SQL

"'If you write on your CV that you know SQL, but you do not use analytic functions, then you are lying.' Personally I would just hate to solve SQL tasks without having the use of analytic functions— they make it so much easier when your logic needs data from multiple rows, which is often the case."

tasks without having the use of analytic functions— they make it so much easier when your logic needs data from multiple rows, which is often the case.

I mentioned earlier the case of a FIFO warehouse picking list. Suppose you're the developer and I give you this task:

- ➤ You have a set of orderlines from multiple orders that need to be picked from the warehouse as one batch.
- ➤ For each of the products in the orderlines you need to pick from the locations in the warehouse where the oldest purchased inventory is stored (the First-In-First-Out principle).
- ➤ The resulting picking list needs to be output in an optimal driving order, so the operator doesn't waste time by driving the picking trolley back and forth in the warehouse.

Given that specification, would you say that you probably would need to retrieve orderlines and loop over them procedurally, do a query of inventory for each product and find the oldest,

"Analytic functions have been available since version 8i or thereabouts, and it has been my go-to solution ever since whenever I need to develop logic that crosses row boundaries. But SQL hasn't stopped evolving since 8i; for example, from version 12.2 match_recognize has been added to my toolbox for cases where even analytic function SQL would be too convoluted."

and then sort the resultant collection in the desired driving order?

You have another way?

Yes, that entire logic can be developed in a single SQL statement with analytic functions. Not even a particularly large statement, just 78 lines, but it's an example that is too large for most typical books or blog posts. My Chapter 13 consists of stepping through developing this statement from a smaller statement that just does the FIFO part on a single order and building it up step by step adding driving order and batch picking, until at the end of the chapter you can build basically an entire picking list application in just one SQL statement.

"Expertise comes from practice. Confidence comes from familiarity. You should just go ahead and write slightly more complex SQL tomorrow, then slightly more the day after, and so on. Over time it will become as familiar to you as whatever other language you've worked in for years, and you will say to yourself, 'What was I afraid of?'"

This is a great example of putting business logic in your SQL, rather than just using SQL to pull out data and then applying logic procedurally. And no framework, no matter how much they promise you can code great applications without knowing SQL, can build logic like that. At least I don't know of a framework that can—but I am very sure that if a framework allowed you to configure and declare similar logic without SQL, then that would be *more* convoluted to use than it would be just to bite the bullet and learn SQL and be happy.

An example like that with efficient use of analytic functions is just the tip of the iceberg. Analytic functions have been available since version 8*i* or thereabouts, and it has been my go-to solution ever since whenever I need to develop logic that crosses row boundaries. But SQL hasn't stopped evolving since 8*i*; for example, from version 12.2 match_recognize has been added to my toolbox for cases where even analytic function SQL would be too convoluted. With analytic functions and match_recognize you can build quite complex logic in very efficient SQL statements.

The theme of the international NoCOUG SQL challenges is *"pretzels created by someone who is experimenting with hallucinogens."*[1] Are your solutions like those pretzels?

Maybe you are under the impression that if SQL is slightly more complex than a two-table join, then it is only for geniuses to attempt and you won't even try it. I assure you this is not the case.

Expertise comes from practice. Confidence comes from familiarity. You should just go ahead and write slightly more complex SQL tomorrow, then slightly more the day after, and so on. Over time it will become as familiar to you as whatever other language you've worked in for years, and you will say to yourself, "What was I afraid of?"

You need a slightly different mindset when coding SQL: you think about the whole set of data instead of just handling one row at a time. But it's not a mindset that only special minds can achieve; it's a mindset everybody can acquire. The requirement is merely to start practicing; without doing "wax on, wax off," the Karate Kid would never have learned to be good at what he did.

And that's the core reason I wrote the book: I want more developers to acquire this mindset, and I am confident it will give developers (who already know a bit of SQL) a jumpstart in their journey toward *really* using the power of SQL.

If they end up with the attitude that "of course it is possible to solve in SQL," their bosses will be happy because they save a lot on cloud credits or licenses when the efficient SQL code uses much less CPU. And the developers will be happier because it is much more fun really exercising the brain cells to find a good solution.

Then I will be happy too and can say, "Mission accomplished!"

Before making the buying decision, I always check out the author. We've talked a lot about the book and why you wrote it, but not about you . . .

When I was young in the mid-'80s I originally wanted to work with electronics, but almost by coincidence I got to try computer programming. I discovered the programs I wrote worked well—unlike the electronics projects I soldered that often failed. So that led to the purchase of a Commodore VIC-20 with 5 kilobytes RAM and many hours of programming in Commodore Basic, trying to fit as much code into that small memory as possible.

[1] Steven Feuerstein was asked the following question in an interview published in the May 2006 issue of the *NoCOUG Journal*: *"SQL is a set-oriented non-procedural language; i.e., it works on sets and does not specify access paths. PL/SQL on the other hand is a record-oriented procedural language, as is very clear from the name. What is the place of a record-oriented procedural language in the relational world?"* Steven replied: *"Its place is proven: SQL is not a complete language. Some people can perform seeming miracles with straight SQL, but the statements can end up looking like pretzels created by someone who is experimenting with hallucinogens. We need more than SQL to build our applications, whether it is the implementation of business rules or application logic. PL/SQL remains the fastest and easiest way to access and manipulate data in an Oracle RDBMS, and I am certain it is going to stay that way for decades."*

"You need a slightly different mindset when coding SQL: you think about the whole set of data instead of just handling one row at a time. But it's not a mindset that only special minds can achieve; it's a mindset everybody can acquire. The requirement is merely to start practicing; without doing 'wax on, wax off,' the Karate Kid would never have learned to be good at what he did."

It was clear where my talent was, so I abandoned electronics and went on to study computer science at Odense University, financing it with a summer job as sheriff of Legoredo (a western town within Legoland in Denmark). There I learned programming in Modula-2 and C, but more importantly, I learned to be methodical both when learning something new and when developing programs. That was invaluable when I started working for a consulting company as a developer making customizations for customers, specifically in the Danish ERP software Concorde XAL that was the predecessor of Microsoft Dynamics AX. That software could run on different databases, one of which was Oracle, so that gave me my first introduction to Oracle SQL and PL/SQL. Since then I've worked extensively with those two languages, with SQL being closest to my heart.

Book writing is the culmination of my knowledge sharing, which I do by blogging at my **kibeha.dk** website, presenting at various Oracle User Group conferences, and being the SQL quizmaster at the Oracle Dev Gym at **devgym.oracle.com**. The last I enjoy doing, as many developers learn SQL better if a bit of fun and games is involved. Even experienced people discover new corners of the SQL language that they hadn't stumbled upon before, and several have told me they use the Dev Gym to practice and prepare for certification exams.

It sounds like your life is centered around SQL. Don't you have a life outside SQL?

But of course. A brain needs to recuperate from all the hard thinking work and doing creative solutions all the time, so I'm careful to have time where my brain thinks of anything but SQL.

I'm married—we had our 25th anniversary last spring—and it's perfect in the sense that my wife enjoys gardening and I enjoy cooking. She relaxes working with perennials, while I turn off the SQL side of my brain in the afternoon, plan what to cook, go grocery shopping, and cook dinner for the both of us. (My brain is kept just sufficiently occupied to not consciously work on SQL, so quite often the background processing during dishwashing suddenly pops up with the answer to a problem that was bugging me during the day.)

And as I mentioned, I'm also a beer enthusiast. Our association has about 9,000 members throughout Denmark, divided into 50 local chapters. My local chapter meets every other month for beer tastings and some good food. Also I've just barely started to try to brew my own beer, but it'll be a while before I gain real experience in that area.

Most evenings and other free time here and there I will spend with a good book—preferably science fiction or fantasy. Mostly I simply enjoy classic sci-fi authors; I'm not keeping much up to date with new authors. My all-time favorite is Robert Heinlein, but I also like Spider Robinson, Terry Pratchett, Anne McCaffrey, Isaac Asimov, Larry Niven, Jerry Pournelle, and many more. I re-read books many times. Sure, I use my smartphone a good deal like most do, but I'd rather grab a book from the bookcase and read a chapter at night instead of browsing social media.

"[I am] the SQL quizmaster at the Oracle Dev Gym at devgym.oracle.com [which] I enjoy doing, as many developers learn SQL better if a bit of fun and games is involved. Even experienced people discover new corners of the SQL language that they hadn't stumbled upon before, and several have told me they use the Dev Gym to practice and prepare for certification exams."

I hope the *Journal* readers grab your book. Thanks for your time, Kim.

My pleasure. I hope you and your readers enjoy the book. ▲

Kim Berg Hansen lives in Middelfart, Denmark. He is a certified Oracle OCE in SQL and an Oracle ACE director, and speaks at user group conferences in Europe and the USA. His motivation comes from peers who say "Now I understand" after his explanations and from end users who "can't live without" his application coding.

"If they end up with the attitude that 'of course it is possible to solve in SQL,' their bosses will be happy because they save a lot on cloud credits or licenses when the efficient SQL code uses much less CPU. And the developers will be happier because it is much more fun really exercising the brain cells to find a good solution."

BOOK EXCERPT

Rolling Sums to Forecast Reaching Minimums

with Kim Berg Hansen

Kim Berg Hansen

This article is a preview from the upcoming book Practical Oracle SQL: Mastering the Full Power of Oracle Database *by Apress, March 24, 2020, ISBN 978-1484256169; Copyright © 2020 by Kim Berg Hansen. Reprinted with permission.*

If you have a steady consumption rate it is easy to forecast how far you can go with that rate; for example, if you know your car on average drives 20 kilometers per liter of fuel and it has 30 liters left in the tank, you can simply multiply to know that you can drive 600 kilometers before you run out of fuel.

But if the consumption is not steady, you need something else. If the Good Beer Trading Co sells a particular seasonal Christmas beer, it is not simply a steady 100 beers sold per month—June will sell very few of those beers, while December sells hundreds. For such a case you estimate (perhaps using the techniques of the previous chapter) what you think you are going to sell and store it as a *forecast* or sales budget.

Once you have forecast that you are going to sell 150 in January, 100 in February, 250 in March, etc., you need to figure out that the 400 you have in stock in your inventory will dwindle to 250 by end of January and to 150 by end of February—and be sold out a little later than the middle of March. Figuring this out is the topic of this chapter.

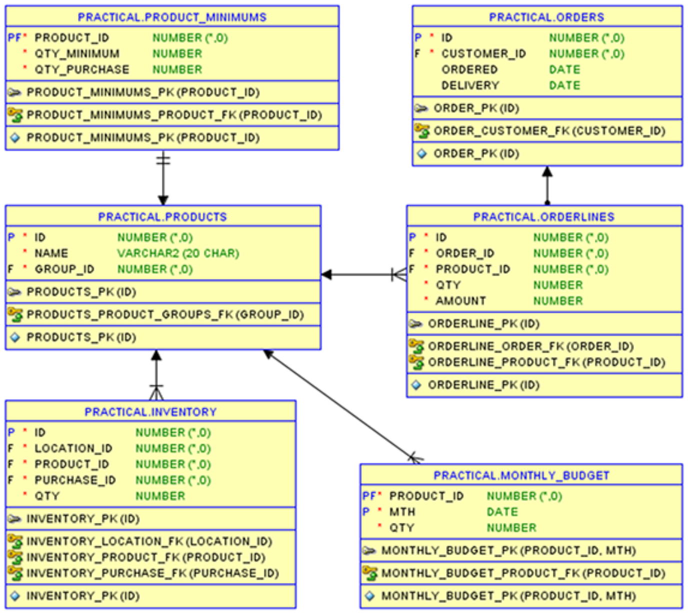
Inventory, budget and order

In the Good Beer Trading Co example, I'm going to demonstrate the case of forecasting when the inventory reaches zero (or a minimum), given that I know how many beers are in stock (waiting to be picked from the inventory) and how many beers are budgeted to be sold (assumed to be picked at some point.)

I'll use month as the time granularity, budgeting sales quantities per month. For this demonstration purpose I don't need to go to weekly or daily data, but you can easily adapt the methods to finer time granularity if you need it. I will use the data in the tables shown in Figure 16-1.

From table `inventory` I know what quantity of each beer is in stock; table `monthly_budget` shows me the quantity of each beer that is expected to sell per month; and table `orderlines` reveals how much has been ordered (but *not yet* picked and therefore not yet taken from the stock). I'll get back to table `product_minimums` later in the chapter.

You'll notice the `inventory` table contains quantities per location (I used the table in the FIFO picking in chapter 13), but for this purpose I just need the total quantity in stock per beer.


PRACTICAL.PRODUCT_MINIMUMS
PF* PRODUCT_ID NUMBER (*,0)
* QTY_MINIMUM NUMBER
* QTY_PURCHASE NUMBER
PRODUCT_MINIMUMS_PK (PRODUCT_ID)
PRODUCT_MINIMUMS_PRODUCT_FK (PRODUCT_ID)
PRODUCT_MINIMUMS_PK (PRODUCT_ID)
PRACTICAL.ORDERS
P * ID NUMBER (*,0)
F * CUSTOMER_ID NUMBER (*,0)
ORDERED DATE
DELIVERY DATE
ORDER_PK (ID)
ORDER_CUSTOMER_FK (CUSTOMER_ID)
ORDER_PK (ID)
PRACTICAL.PRODUCTS
P * ID NUMBER (*,0)
* NAME VARCHAR2 (20 CHAR)
F * GROUP_ID NUMBER (*,0)
PRODUCTS_PK (ID)
PRODUCTS_PRODUCT_GROUPS_FK (GROUP_ID)
PRODUCTS_PK (ID)
PRACTICAL.ORDERLINES
P * ID NUMBER (*,0)
F * ORDER_ID NUMBER (*,0)
F * PRODUCT_ID NUMBER (*,0)
* QTY NUMBER
* AMOUNT NUMBER
ORDERLINE_PK (ID)
ORDERLINE_ORDER_FK (ORDER_ID)
ORDERLINE_PRODUCT_FK (PRODUCT_ID)
ORDERLINE_PK (ID)
PRACTICAL.INVENTORY
P * ID NUMBER (*,0)
F * LOCATION_ID NUMBER (*,0)
F * PRODUCT_ID NUMBER (*,0)
F * PURCHASE_ID NUMBER (*,0)
* QTY NUMBER
INVENTORY_PK (ID)
INVENTORY_LOCATION_FK (LOCATION_ID)
INVENTORY_PRODUCT_FK (PRODUCT_ID)
INVENTORY_PURCHASE_FK (PURCHASE_ID)
INVENTORY_PK (ID)
PRACTICAL.MONTHLY_BUDGET
PF* PRODUCT_ID NUMBER (*,0)
P * MTH DATE
* QTY NUMBER
MONTHLY_BUDGET_PK (PRODUCT_ID, MTH)
MONTHLY_BUDGET_PRODUCT_FK (PRODUCT_ID)
MONTHLY_BUDGET_PK (PRODUCT_ID, MTH)
Figure 16-1. The tables used in the examples within this chapter

To make that easier, I create the view `inventory_totals` in Listing 16-1 aggregating the inventory per beer:

Listing 16-1. View of total inventory per product

```
SQL> create or replace view inventory_totals
  2  as
  3  select
  4     i.product_id
  5   , sum(i.qty) as qty
  6  from inventory i
  7  group by i.product_id;

View INVENTORY_TOTALS created.
```

Similarly, for the quantities in stock I do not need specific order lines; I just need to include how many of each beer each month, so I aggregate those figures in view `monthly_orders` in Listing 16-2:

Listing 16-2. View of monthly order totals per product

```
SQL> create or replace view monthly_orders
  2  as
  3  select
  4     ol.product_id
  5   , trunc(o.ordered, 'MM') as mth
  6   , sum(ol.qty) as qty
  7  from orders o
  8  join orderlines ol
  9     on ol.order_id = o.id
 10  group by ol.product_id, trunc(o.ordered, 'MM');

View MONTHLY_ORDERS created.
```

Those are the tables and views I'm going to be using; now I'll show the data in them.

The data

I'll use two beers for the examples of this chapter: Der Helle Kumpel and Hazy Pink Cloud. They represent the total inventory shown in Listing 16-3:

Listing 16-3. The inventory totals for two products

```
SQL> select it.product_id, p.name, it.qty
  2  from inventory_totals it
  3  join products p
  4     on p.id = it.product_id
  5  where product_id in (6520, 6600)
  6  order by product_id;

PRODUCT_ID  NAME             QTY
6520        Der Helle Kumpel  400
6600        Hazy Pink Cloud   100
```

This is the total beer in stock as of January 1, 2019. Then I have a monthly sales budget for the year 2019 (Listing 16-4):

Listing 16-4. The 2019 monthly budget for the two beers

```
SQL> select mb.product_id, mb.mth, mb.qty
  2  from monthly_budget mb
  3  where mb.product_id in (6520, 6600)
  4  and mb.mth >= date '2019-01-01'
  5  order by mb.product_id, mb.mth;

PRODUCT_ID  MTH         QTY
6520        2019-01-01  45
6520        2019-02-01  45
6520        2019-03-01  50
...
6520        2019-10-01  50
6520        2019-11-01  40
6520        2019-12-01  40
6600        2019-01-01  20
6600        2019-02-01  20
6600        2019-03-01  20
...
6600        2019-10-01  20
6600        2019-11-01  20
6600        2019-12-01  20

24 rows selected.
```

Product 6520 is expected to sell a bit more in the summer months, while product 6600 is expected to sell a steady 20 per month.

But I don't just have the expected quantities; in Listing 16-5 I also have the quantities that have already been ordered in the first months of 2019:

Listing 16-5. The current monthly order quantities

```
SQL> select mo.product_id, mo.mth, mo.qty
  2  from monthly_orders mo
  3  where mo.product_id in (6520, 6600)
  4  order by mo.product_id, mo.mth;

PRODUCT_ID  MTH         QTY
6520        2019-01-01  260
6520        2019-02-01  40
6600        2019-01-01  16
6600        2019-02-01  40
```

The thing to note here is that in January product 6520 has been ordered much more than was expected.

Given these data, I'll now make some SQL to find out when we run out of beers for those two products.

Accumulating until zero

A use of analytic functions that is useful in many cases is the rolling (accumulated) sum that I've shown before. In Listing 16-6 I use it again:

Listing 16-6. Accumulating quantities

```
SQL> select
  2     mb.product_id as p_id, mb.mth
  3   , mb.qty b_qty, mo.qty o_qty
  4   , greatest(mb.qty, nvl(mo.qty, 0)) as qty
  5   , sum(greatest(mb.qty, nvl(mo.qty, 0))) over (
  6        partition by mb.product_id
  7        order by mb.mth
  8        rows between unbounded preceding and current row
  9     ) as acc_qty
 10  from monthly_budget mb
 11  left outer join monthly_orders mo
 12     on mo.product_id = mb.product_id
 13     and mo.mth = mb.mth
 14  where mb.product_id in (6520, 6600)
 15  and mb.mth >= date '2019-01-01'
 16  order by mb.product_id, mb.mth;
```

In line 4 I calculate the monthly quantity as whichever is the greatest of *either* the budgeted quantity *or* the ordered quantity. In the output below you see in the January entry for product 6520 that `o_qty` is the greatest, making `qty` 260, while January for product 6600 has `b_qty` as the greatest, making `qty` 20.

The idea is that if the ordered quantity is the smallest, there haven't yet been orders to match the budget, but it's still expected to rise until the budget is reached. But when the ordered quantity is the greatest, I know the budget has been surpassed, so I don't expect it to become greater yet.

So this quantity is then what I accumulate with the analytic `sum` in lines 5-9, so I end up with column `acc_qty`, which shows me, accumulated, how much I expect to pick from the inventory:

```
P_ID  MTH         B_QTY  O_QTY  QTY  ACC_QTY
6520  2019-01-01  45     260    260  260
6520  2019-02-01  45     40     45   305
6520  2019-03-01  50            50   355
...
6520  2019-11-01  40            40   775
6520  2019-12-01  40            40   815
6600  2019-01-01  20     16     20   20
6600  2019-02-01  20     40     40   60
6600  2019-03-01  20            20   80
...
6600  2019-11-01  20            20   240
6600  2019-12-01  20            20   260
```

In Listing 16-7 I use this accumulated quantity to calculate the expected inventory for each month (if I don't restock along the way).

Listing 16-7. Dwindling inventory

```
SQL> select
  2     mb.product_id as p_id, mb.mth
  3   , greatest(mb.qty, nvl(mo.qty, 0)) as qty
  4   , greatest(
  5        it.qty - nvl(sum(
  6            greatest(mb.qty, nvl(mo.qty, 0))
  7        ) over (
  8          partition by mb.product_id
  9          order by mb.mth
10           rows between unbounded preceding and 1 preceding
11        ), 0)
12      , 0
13     ) as inv_begin
14   , greatest(
15        it.qty - sum(
16            greatest(mb.qty, nvl(mo.qty, 0))
17        ) over (
18           partition by mb.product_id
19           order by mb.mth
20           rows between unbounded preceding and current row
21        )
22      , 0
23     ) as inv_end
24  from monthly_budget mb
25  left outer join monthly_orders mo
26     on mo.product_id = mb.product_id
27     and mo.mth = mb.mth
28  join inventory_totals it
29     on it.product_id = mb.product_id
30  where mb.product_id in (6520, 6600)
31  and mb.mth >= date '2019-01-01'
32  order by mb.product_id, mb.mth;
```

Lines 4-13 calculate the quantity in stock at the beginning of the month, while lines 14-23 calculate how much was available at the end of the month.

```
P_ID  MTH         QTY  INV_BEGIN  INV_END
6520  2019-01-01  260  400        140
6520  2019-02-01  45   140        95
6520  2019-03-01  50   95         45
6520  2019-04-01  50   45         0
6520  2019-05-01  55   0          0
...
6600  2019-01-01  20   100        80
6600  2019-02-01  40   80         40
6600  2019-03-01  20   40         20
6600  2019-04-01  20   20         0
6600  2019-05-01  20   0          0
...
```

You see how the inventory dwindles until it reaches zero. As I use month for time granularity, in principle I can only state that the inventory will reach zero at some point during that month. But if I assume that the budgeted sales will be evenly distributed throughout the month, I can also make a *guesstimation* in Listing 16-8 of which day that zero will be reached:

Listing 16-8. Estimating when zero is reached

```
SQL> select
  2     product_id as p_id, mth, inv_begin, inv_end
  3   , trunc(
  4        mth + numtodsinterval(
  5                 (add_months(mth, 1) - 1 - mth) * inv_begin / qty
  6               , 'day'
  7              )
  8     ) as zero_day
  9  from (
...
 41  )
 42  where inv_begin > 0 and inv_end = 0
 43  order by product_id;
```

I wrap Listing 16-7 in an inline view and use `inv_begin / qty` in line 5 to figure out how large a fraction of the estimated monthly sales can be fulfilled by the inventory at hand at the beginning of the month. When I assume evenly distributed sales, this is then the fraction of the number of days in the month that I have sufficient stock for.

Filtering in line 42 gives me as output just the rows where the inventory becomes zero:

```
P_ID  MTH         INV_BEGIN  INV_END  ZERO_DAY
6520  2019-04-01  45         0        2019-04-27
6600  2019-04-01  20         0        2019-04-30
```

In reality, however, I wouldn't let the inventory reach zero. I'd set up a minimum quantity that I mustn't go below as a buffer in case I underestimated sales, and every time I get to the minimum quantity I must buy more beer and restock the inventory.

Restocking when minimum reached

In table `product_minimums` I have parameters for the inventory handling of each product. Listing 16-9 shows the table content for the two beers I use for demonstration:

Listing 16-9. Product minimum restocking parameters

```
SQL> select product_id, qty_minimum, qty_purchase
  2  from product_minimums pm
  3  where pm.product_id in (6520, 6600)
  4  order by pm.product_id;
```

Column `qty_minimum` is my inventory buffer; I plan that the inventory should never get below this. Column `qty_purchase` is the number of beers I buy every time I restock the inventory.

```
PRODUCT_ID  QTY_MINIMUM  QTY_PURCHASE
6520        100          400
6600        30           100
```

With this I am ready to write SQL that can show me when I need to purchase more beer and restock throughout 2019.

This is not simply done with analytic functions, since I cannot use the result of an analytic function inside the analytic function itself to add more quantity. This would mean an unsupported type of recursive function call, so it cannot be done. But I can do it with *recursive subquery factoring* instead of analytic functions, as shown in Listing 16-10:

Listing 16-10. Restocking when a minimum is reached

```
SQL> with mb_recur(
  2     product_id, mth, qty, inv_begin, date_purch
  3   , p_qty, inv_end, qty_minimum, qty_purchase
  4  ) as (
  5     select
  6        it.product_id
  7      , date '2018-12-01' as mth
  8      , 0 as qty
  9      , 0 as inv_begin
 10      , cast(null as date) as date_purch
 11      , 0 as p_qty
```

"If you have a steady consumption rate it is easy to forecast how far you can go with that rate; for example, if you know your car on average drives 20 kilometers per liter of fuel and it has 30 liters left in the tank, you can simply multiply to know that you can drive 600 kilometers before you run out of fuel."

```
12       , it.qty as inv_end
13       , pm.qty_minimum
14       , pm.qty_purchase
15      from inventory_totals it
16      join product_minimums pm
17         on pm.product_id = it.product_id
18      where it.product_id in (6520, 6600)
19   union all
20      select
21         mb.product_id
22       , mb.mth
23       , greatest(mb.qty, nvl(mo.qty, 0)) as qty
24       , mbr.inv_end as inv_begin
25       , case
26            when mbr.inv_end - greatest(mb.qty, nvl(mo.qty, 0))
27                  < mbr.qty_minimum
28            then
29               trunc(
30                  mb.mth
31                + numtodsinterval(
32                     (add_months(mb.mth, 1) - 1 - mb.mth)
33                      * (mbr.inv_end - mbr.qty_minimum)
34                      / mb.qty
35                   , 'day'
36                  )
37               )
38         end as date_purch
39       , case
40            when mbr.inv_end - greatest(mb.qty, nvl(mo.qty, 0))
41                  < mbr.qty_minimum
42            then mbr.qty_purchase
43         end as p_qty
44       , mbr.inv_end - greatest(mb.qty, nvl(mo.qty, 0))
45          + case
46               when mbr.inv_end - greatest(mb.qty, nvl(mo.qty, 0))
47                     < mbr.qty_minimum
48               then mbr.qty_purchase
49               else 0
50            end as inv_end
51       , mbr.qty_minimum
52       , mbr.qty_purchase
53      from mb_recur mbr
54      join monthly_budget mb
55         on mb.product_id = mbr.product_id
56         and mb.mth = add_months(mbr.mth, 1)
57      left outer join monthly_orders mo
58         on mo.product_id = mb.product_id
59         and mo.mth = mb.mth
60   )
61   select
62      product_id as p_id, mth, qty, inv_begin
63    , date_purch, p_qty, inv_end
64   from mb_recur
65   where mth >= date '2019-01-01'
66   and p_qty is not null
67   order by product_id, mth;
```

I start in lines 5-18 by setting up one row per product containing the beginning inventory along with the parameters for minimum quantity and how much to purchase. I set this row as being in December 2018 with the inventory in the `inv_end` column—that way it will function as a "primer" row for the recursive part of the query in lines 20-59.

In the recursive part I do:

- Join to the monthly budget for the *next* month in line 56. The first iteration here will find January 2019 (since my "primer" row was December 2018); then each iteration will find the next month until there are no more budget rows.
- The `inv_begin` of this next month in the iteration is then equal to the `inv_end` of the previous month, so that's a simple assignment in line 24.
- Lines 44-50 calculate the `inv_end`, which is the beginning inventory (previous `inv_end`) *minus* the quantity picked that month *plus* a possible restocking. If the beginning inventory minus the quantity would become less than the minimum, I add the quantity I will be purchasing for restocking.
- To show on the output how much I need to purchase for restocking, I separate this `case` structure out in lines 39-43.
- In lines 25-28 I use the same `case` condition to calculate an estimated date of the month when restocking by purchasing more beer should take place.

"But if the consumption is not steady, you need something else. If the Good Beer Trading Co sells a particular seasonal Christmas beer, it is not simply a steady 100 beers sold per month—June will sell very few of those beers, while December sells hundreds. For such a case you estimate (perhaps using the techniques of the previous chapter) what you think you are going to sell and store it as a forecast or sales budget."

Line 65 removes the "primer" rows from the output (they are not interesting), and line 66 gives me just those months when I need to restock:

```
P_ID  MTH         QTY  INV_BEGIN  DATE_PURCH  P_QTY  INV_END
6520  2019-02-01  45   140        2019-02-25  400    495
6520  2019-10-01  50   115        2019-10-10  400    465
6600  2019-03-01  20   40         2019-03-16  100    120
6600  2019-08-01  20   40         2019-08-16  100    120
```

I am now able to plan when I need to purchase more beer to restock the inventory.

In Listing 16-10 I used recursive subquery factoring. The way I did it means that for the budget and orders there will be a series of repeated small lookups to the tables for each month. Depending on circumstances this might be perfectly fine, but in other cases it could be bad for performance.

Listing 16-11 shows an alternative method of recursion (or, rather, *iteration*) with the `model` clause instead, where a different access plan can be used by the optimizer:

Listing 16-11. Restocking with model clause

```
SQL> select
  2     product_id as p_id, mth, qty, inv_begin
  3   , date_purch, p_qty, inv_end
  4  from (
  5     select *
  6     from monthly_budget mb
  7     left outer join monthly_orders mo
```

“Once you have forecast that you are going to sell 150 in January, 100 in February, 250 in March, etc., you need to figure out that the 400 you have in stock in your inventory will dwindle to 250 by end of January and to 150 by end of February—and be sold out a little later than the middle of March. Figuring this out is the topic of this chapter.”

```
8        on mo.product_id = mb.product_id
9        and mo.mth = mb.mth
10    join inventory_totals it
11       on it.product_id = mb.product_id
12    join product_minimums pm
13       on pm.product_id = mb.product_id
14    where mb.product_id in (6520, 6600)
15    and mb.mth >= date '2019-01-01'
16    model
17    partition by (mb.product_id)
18    dimension by (
19       row_number() over (
20          partition by mb.product_id order by mb.mth
21       ) - 1 as rn
22    )
23    measures (
24       mb.mth
25     , greatest(mb.qty, nvl(mo.qty, 0)) as qty
26     , 0 as inv_begin
27     , cast(null as date) as date_purch
28     , 0 as p_qty
29     , 0 as inv_end
30     , it.qty as inv_orig
31     , pm.qty_minimum
32     , pm.qty_purchase
33    )
34    rules sequential order iterate (12) (
35       inv_begin[iteration_number]
36        = nvl(inv_end[iteration_number-1], inv_orig[cv()])
37     , p_qty[iteration_number]
38        = case
39             when inv_begin[cv()] - qty[cv()]
40                   < qty_minimum[cv()]
41             then qty_purchase[cv()]
42          end
43     , date_purch[iteration_number]
44        = case
45             when p_qty[cv()] is not null
46             then
47                trunc(
48                   mth[cv()]
49                 + numtodsinterval(
50                      (add_months(mth[cv()], 1) - 1 - mth[cv()])
51                       * (inv_begin[cv()] - qty_minimum[cv()])
52                       / qty[cv()]
53                     , 'day'
54                   )
55                )
56          end
57     , inv_end[iteration_number]
58        = inv_begin[cv()] + nvl(p_qty[cv()], 0) - qty[cv()]
59    )
60 )
61 where p_qty is not null
62 order by product_id, mth;
```

With this method I do not need “primer” rows and repeated monthly lookups. Instead I grab all the data I need in one go in lines 5-15, rather as if I were using analytic functions. And then I can use `model`:

- Lines 19-21 create a consecutive numbering that I can use as dimension (“index”) in my measures. I deliberately make it have the values 0 to 11 instead of 1 to 12, because that fits how `iteration_number` is filled when using iteration.
- In the `measures` in line 24-32 I set up the “variables” I need to work with.
- In the `rules` clause I can then perform all my calculations. In line 34 I specify that I want my calculations to be performed in the order I have typed them, and they should be performed 12 times. This means that within each of the 12 iterations I can use the pseudocolumn `iteration_number`, and it will increase from 0 to 11.
- The first rule to be executed is lines 35-36, where I set `inv_begin` to the `inv_end` of the previous month (in the first iteration this will be `null`, so with `nvl` I set it to the original inventory in the first month).
- If the inventory minus the quantity is less than the minimum, then in lines 37-42 I set `11.91 pt` to the quantity I need to purchase.
- If I *did* find a `p_qty` (line 45), the rule in lines 43-56 calculates the day I need to purchase and restock.
- Lines 57-68 calculate the `inv_end` by using the other measures.

The 12 iterations and calculations are quite similar to what I did in the recursive subquery factoring, except that I use measures indexed by a dimension where the data in those measures have all been filled initially before I start iterating and calculating.

This method will for some cases enable more efficient access of the tables—but at the cost of using more memory to keep all the data and work with it in the `model` clause (potentially needing to spill some to disk if you have huge amounts of data here). Whether Listing 16-10 or 16-11 is the best will depend on the case. You’ll need to test the methods yourself.

Lessons learned

Analytic functions are extremely useful and can solve a lot of things, including rolling sums to find when you reach some minimum. But they cannot do everything, so in this chapter I showed you a mix of:

- Subtracting a rolling sum from a starting figure to discover when a minimum (or zero) has been reached.
- Using recursive subquery to repeatedly replenish the dwindling figure whenever the minimum has been reached.
- Using model clause to accomplish the same with an alternative data access plan.

This highly functional mix of techniques should help you solve similar cases in the future. ▲

Oracle RAC on Amazon EC2 Enabled by FlashGrid SkyCluster

Artem Danielov

with Artem Danielov

The use of Amazon Elastic Compute Cloud (Amazon EC2) in the Amazon Web Services (AWS) Cloud provides IT organizations with the flexibility and elasticity that are not available in the traditional data center. With AWS it is possible to bring new enterprise applications online in hours instead of months.

Ensuring high availability of backend relational databases is a critical part of the cloud strategy—whether it is a lift-and-shift migration or a green-field deployment of mission-critical applications. FlashGrid SkyCluster is an engineered cloud system designed for database high availability. SkyCluster is delivered as a fully integrated infrastructure-as-code template that can be customized and deployed to an AWS EC2 account with a few mouse clicks. Key components of FlashGrid SkyCluster for AWS include:

- Amazon EC2 VM instances
- Amazon EBS and/or local SSD storage
- FlashGrid Storage Fabric software
- FlashGrid Cloud Area Network software
- Oracle Grid Infrastructure software
- Oracle RAC database engine

By leveraging the proven Oracle RAC database engine FlashGrid SkyCluster enables the following use cases:

- Lift-and-shift migration of existing Oracle RAC databases to AWS
- Migration of existing Oracle databases from high-end on-premises servers to AWS without reducing availability SLAs
- Design of new mission-critical applications for the cloud based on the industry-proven and widely supported database engine.

This paper provides an architectural overview of FlashGrid SkyCluster and can be used for planning and designing high-availability database deployments on Amazon EC2.

Why Oracle RAC Database Engine

Oracle RAC provides an advanced technology for database high availability. Many organizations use Oracle RAC for running their mission-critical applications, including most financial institutions and telecom operators where high availability and data integrity are of paramount importance.

Oracle RAC is an active-active distributed architecture with shared database storage. The shared storage plays a central role in enabling automatic failover, zero data loss, and 100% data consistency, and in preventing application downtime. These HA capabilities minimize outages due to unexpected failures, as well as during planned maintenance.

Oracle RAC technology is available for both large-scale and entry-level deployments. Oracle RAC Standard Edition 2 provides a very cost-efficient alternative to open-source databases while ensuring the same level of high availability that the Enterprise Edition customers enjoy.

Supported Cluster Configurations

FlashGrid SkyCluster enables a variety of RAC cluster configurations on Amazon EC2. Two or three node clusters are recommended in most cases. Clusters with four or more nodes can be used for extra HA or performance. It is possible to have clusters with more than four nodes containing several 2- or 3-node database sub-clusters. It is also possible to use SkyCluster for running single-instance databases with automatic failover. Nodes of a cluster can be in one availability zone or can be spread across availability zones.

Configurations with Two RAC Database Nodes

Configurations with two RAC database nodes have two-way data mirroring using normal redundancy ASM disk groups. An additional EC2 instance is required to host quorum disks. Such a cluster can tolerate the loss of any one node without database downtime. See Figure 1.

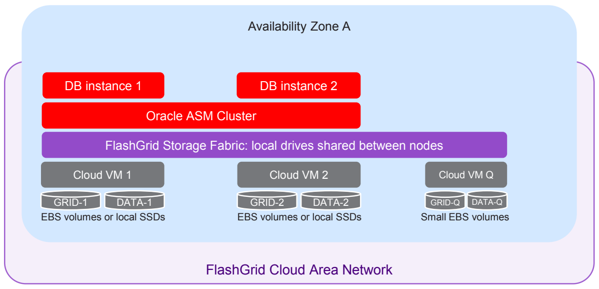


Figure 1. Two RAC database nodes in the same availability zone

(continued on page 16)

Solving the Unsolvable Performance Problem

How do you solve production performance problems? The method you use may be structurally incapable of helping you find certain types of performance problems. Changing how you look at your system makes all the difference.

by Cary Millsap

The way you look at your system can block you from understanding your performance problems.

Many Oracle technologists don't know how to answer the most fundamental question about performance: "How does a given program spend its time?"

Tracing explains how a program spends its time. This enables you to find wasted time, no matter where it occurs in your technology stack.

Method R Workbench makes it easy to work with trace data, even when you have thousands of trace files.

The ability to measure the response times of your business's more important programs helps you solve more problems, more efficiently.

Problem

We meet people all the time who have suffered a particular performance problem for months, or even years. The people we work with are plenty smart and plenty motivated. Most of them have spent loads of money on hardware upgrades and consulting assistance. How can their problems remain unsolved for so long?

Most technologists view a computer system from the *supply* perspective. They measure the system's internal resources and seek patterns that might imply bad behavior. Unfortunately, many performance problems are invisible to this method. However, the same problems are easy to find when you view the system from the *demand* perspective.

Plan

You can solve more problems and waste less time when you stop looking at a system as a collection of resources and start looking at it as a collection of *user experiences*. For example, imagine that your business tells you that BalanceInquiry is too slow. Then your success will be measured by how much you improve the response time of BalanceInquiry. No other metric matters. Your mission, then, begins with answering one critical question:

> *How does* BalanceInquiry *spend its time?*

The problem is, tools like AWR and Oracle Enterprise Manager don't answer that question. It's even worse than it sounds: those tools can actually weaken your understanding of what your programs are really doing. The performance ideas your tools inspire may actually be the cause of your suffering.

Analysis

You can answer the big question—"How does my program spend its time?"—by representing a program's duration in the same quantity-and-price format you would expect on an invoice or a restaurant receipt. This type of report is called a *profile*.

A profile's bottom line shows the duration that the person who ran the program actually felt, and the durations in the table sum to the bottom line. The format makes it clear that there are only two possible root causes for any performance problem: (a) some call count (quantity) is too high, or (b) some call duration (price) is too high.

Subroutine	Duration (seconds)	Calls	Mean duration per call (seconds)	Max duration per call (seconds)
db file parallel read	15.624	3	5.208	5.407
waiting for CPU	13.844	21	0.659	13.657
CPU: EXEC dbcalls	0.261	8	0.032	0.245
11 others	0.274	4,170	0.000	0.000
Total (14)	30.003	4,202	0.003	13.657

A *profile* represents a program's duration as a table of quantities (calls) and prices (durations). It's an invoice for response time.

Solution

You can't create a profile like this for an Oracle application program using AWR (or even ASH) data. But Oracle does provide the information you need to create a profile with its *extended SQL trace* feature.

Tracing is easy, but it can generate a lot of data. With our Method R Workbench, you can profile individual user experiences even if you have thousands of trace files. The result: a method for solving problems you wouldn't have solved any other way.

Example

A wealth management application has had performance problems for a year. The BalanceInquiry feature is normally almost instantaneous, but several times each day, it consumes 20 s (seconds) or more. The behavior is eroding customer faith in the application.

The application isn't designed for easy tracing, so we trace the whole system for an hour. We use Method R Workbench to find the 20 s BalanceInquiry execution. Its profile reveals that 19.8 s is consumed by a single log file sync call.

Further investigation using information that is available only in trace files reveals that this specific log file sync call was blocked by a log file parallel write call being executed at the same time by the Oracle LGWR process. Other users were similarly blocked by the same write call.

We found this problem within 30 minutes of receiving our first batch of trace files. How could it have evaded detection for a year? It's because the "average" log file sync call isn't a problem. AWR shows that the average log file sync duration on this system is only 0.025 s.

In hindsight, of course you can see in v$event_histogram that log file sync calls do take 19+ s sometimes. But the team didn't act on that information because log file sync was unremarkable—v$event_histogram shows fifty other call types that behave the same way. Even if someone *had* proposed log file sync as BalanceInquiry's problem, the organization probably wouldn't have had the resolve to fix it, because there was no cause-effect link to justify the cost.

…And so the intermittent BalanceInquiry problem persisted for *a year*. Before tracing, there were hundreds of possible root causes, none provable or disprovable with aggregated data. But after tracing, there is no doubt: log file sync (and thus log file parallel write) is the definite cause of the problem.

Method

This is a story we repeat over and over: intractable problem; trace it (trace *everything* if we have to, and sift through tens of thousands of trace files in just a few minutes with Method R Workbench); profile the interesting user experience; now we know where the wasted time goes.

» In an airport management system experiencing application timeouts, trace files revealed a row lock held for ten seconds because of an intermittent disk latency problem. Nothing in the system-wide statistics implicated either locking or disk I/O as a diagnostic priority.

» In a global convenience store's test kitchen where a simple ingredient search took two minutes on a $6,000,000 computer, trace files revealed a badly chosen Oracle parameter value and an application design mistake. Neither problem presented symptoms visible in system-wide statistics.

» In a professional society that had just moved to the cloud, trace files revealed response times dominated by now-longer network latencies. Luckily, a bad but easily fixable coding habit was causing a lot of unnecessary network round trips. Their Oracle monitoring tools, like most, discard network I/O call data.

» In a GPS location monitoring application that intermittently overwhelmed its CPU capacity, trace files revealed a bad coding habit of dynamically generating and parsing thousands of unique SQL statements per hour. Monitoring tools showed the bad practice but undervalued the impact.

» In a finance management system that overran its nightly batch window, trace files revealed the problem to be not an Oracle Database problem at all, but a suboptimally programmed third-party application running between specific pairs of SQL statements. Monitoring tools did report the database as mostly idle but provided insufficient information to help solve the problem.

Everywhere we go, we find problems hiding behind averages. The symptoms vary from one system to the next, but the same method works for all of them: you answer the question, "How does my program spend its time?" The answer begins with tracing.

Technology

Method R Workbench is easy-to-use, high-precision *Oracle time measurement software* for software development, code reviews, performance tests, concept proofs, hardware and software evaluations, upgrades, troubleshooting, and more—for Oracle developers, DBAs, and decision-makers in every phase of the software life cycle. The sifting and profiling operations described in this monograph are standard product features.

Method R Workbench 9

◇ Method R™
method-r.com
info@method-r.com

◇ Method R™ *Oracle® Performance Monograph Series*

(continued from page 13)

In configurations where local NVMe SSDs are used instead of EBS volumes, high-redundancy ASM disk groups may be used to provide an extra layer of data protection. In such cases the third node is configured as a *storage* node with NVMe SSDs or EBS volumes instead of the quorum node.

Configurations with Three RAC Database Nodes

Configurations with three RAC database nodes have three-way data mirroring using high-redundancy ASM disk groups. Two additional EC2 instances are required to host quorum disks. Such a cluster can tolerate the loss of any two nodes without database downtime. See Figure 2.

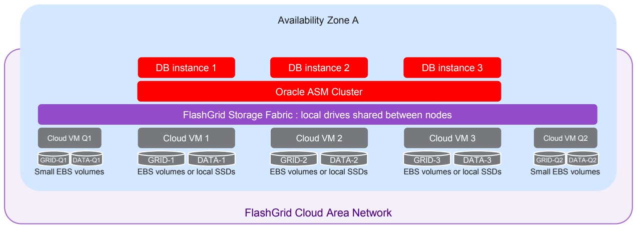


Figure 2. Three RAC database nodes in the same availability zone

Same Availability Zone vs. Separate Availability Zones

Amazon Web Services consists of multiple independent Regions. Each Region is partitioned into several Availability Zones. Availability Zones consist of one or more discrete data centers, each with redundant power, networking, and connectivity, housed in separate facilities. Availability Zones are physically separate, such that even extremely uncommon disasters such as fires, tornados, or flooding would only affect a single Availability Zone.

Although Availability Zones within a Region are geographically isolated from each other, they have direct low-latency network connectivity between them. The network latency between Availability Zones is generally lower than 1 ms. This makes the inter-AZ deployments compliant with the extended-distance RAC guidelines.

Placing all nodes in one Availability Zone provides the best performance for write-intensive applications by ensuring network proximity between the nodes. However, in the unlikely event of an entire Availability Zone failure, the cluster will experience downtime.

Placing each node in a separate Availability Zone helps avoid downtime, even when an entire Availability Zone experiences a failure. The trade-off is a somewhat higher network latency, which may reduce write performance. Note that the read performance is not affected because all reads are served locally.

If placing nodes in separate Availability Zones, using a Region with at least three Availability Zones is generally required. See Figure 3. The current number of Availability Zones for each Region can be found at **https://aws.amazon.com/about-aws/global-infrastructure/**. It is possible to deploy a 2-node RAC cluster in a Region with only two Availability Zones. However, in such a case the quorum server must be located in a different Region or in a data center with VPN connection to AWS, in order to prevent network partitioning scenarios. This configuration is beyond the scope of this article; to learn more, contact FlashGrid.

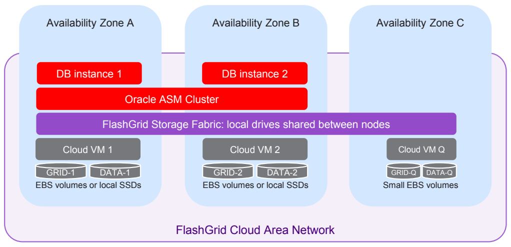


Figure 3. Two RAC database nodes in separate availability zones

Three RAC Database Nodes Across Availability Zones

Most of the AWS regions are limited to three Availability Zones. Because of this, placing the additional quorum nodes in separate Availability Zones may not be possible. However, with three RAC nodes, placing the quorum nodes in the same Availability Zones as the RAC nodes can still achieve most of the expected HA capabilities. Such a cluster can tolerate the loss of any two nodes or any one Availability Zone without database downtime. Note, however, that the simultaneous loss of two Availability Zones will cause database downtime. See Figure 4.

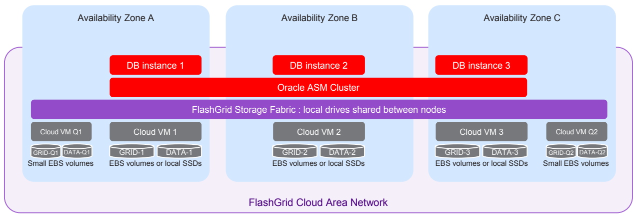


Figure 4. Three RAC database nodes in separate availability zones

Four or More RAC Database Nodes Across Availability Zones

It is possible to configure clusters with four or more nodes across availability zones, with two or more database nodes per Availability Zone. The database nodes are spread across two Availability Zones. The third Availability Zone is used for the quorum node. Such a cluster can tolerate the loss of an entire Availability Zone. In addition, it allows HA within each Availability Zone, which helps with application HA design. See Figure 5.

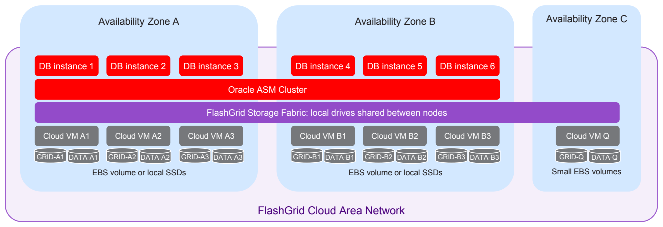


Figure 5. Example of a 6-node RAC database cluster across availability zones

FlashGrid SkyCluster Architecture Highlights

- Database clusters are delivered as Infrastructure-as-Code templates for automated and repeatable deployments.
- FlashGrid Cloud Area Network™ (CLAN) software enables high-speed overlay networks with advanced capabilities for HA and performance management.

- ➤ FlashGrid Storage Fabric software turns locally attached disks (elastic block storage or local instance-store SSDs) into shared disks accessible from all nodes in the cluster.
- ➤ FlashGrid Read-Local™ Technology minimizes network overhead by serving reads from locally attached disks.
- ➤ The architecture allows two-way or three-way mirroring of data across separate nodes or Availability Zones.
- ➤ Oracle ASM and Clusterware provide data protection and availability.

Network

FlashGrid Cloud Area Network enables running high-speed clustered applications in public clouds or multi-datacenter environments with the efficiency and control of a local area network.

The network connecting Amazon EC2 instances is effectively a single-IP network with a fixed amount of network bandwidth allocated per instance for all types of network traffic (except for Amazon Elastic Block Storage (EBS) storage traffic on EBS-optimized instances). However, the Oracle RAC architecture requires separate networks for client connectivity and for the private cluster interconnect between the cluster nodes. There are two main reasons for this requirement: 1) the cluster interconnect must have low latency and sufficient bandwidth to ensure adequate performance of the inter-node locking and Cache Fusion, and 2) the cluster interconnect is used for transmitting raw data and for security reasons must be accessible by the database nodes only. Also, Oracle RAC requires networks with multicast capability, which is not available in Amazon EC2.

FlashGrid CLAN addresses the limitations described above by creating a set of high-speed virtual LAN networks and ensuring QoS between them. See Figure 6.

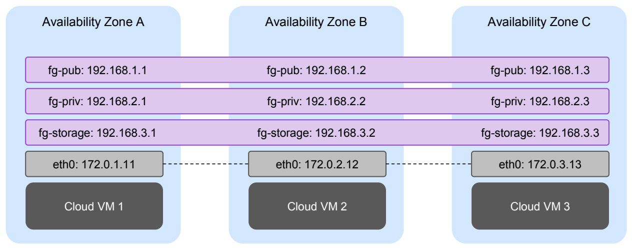


Figure 6. FlashGrid Cloud Area Network

Network capabilities enabled by FlashGrid CLAN for Oracle RAC in Amazon EC2:

- ➤ Each type of traffic has its own virtual LAN with a separate virtual NIC, e.g., fg-pub, fg-priv, fg-storage
- ➤ Negligible performance overhead compared to the raw network
- ➤ Minimum guaranteed bandwidth allocation for each traffic type while accommodating traffic bursts
- ➤ Low latency of the cluster interconnect in the presence of large volumes of traffic of other types
- ➤ Transparent connectivity across Availability Zones
- ➤ Multicast support
- ➤ Up to 100 Gb/s bandwidth per node

Shared Storage

FlashGrid Storage Fabric turns local disks into shared disks accessible from all nodes in the cluster. The local disks shared with FlashGrid Storage Fabric can be block devices of any type, including Amazon EBS volumes or local SSDs. The sharing is done at the block level with concurrent access from all nodes. See Figure 7.

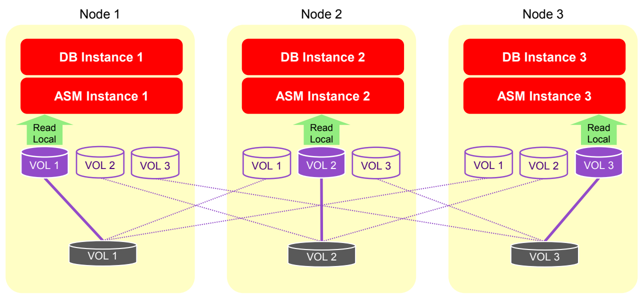


Figure 7. FlashGrid Storage Fabric with FlashGrid Read-Local Technology

In 2-node or 3-node clusters each database node has a full copy of user data stored on Amazon EBS volume(s) attached to that database node. The FlashGrid Read-Local Technology allows serving all read I/Os from the locally attached disks and increases both read and write I/O performance. Read requests avoid the extra network hop, thus reducing the latency and the amount of the network traffic. As a result, more network bandwidth is available for the write I/O traffic.

"FlashGrid SkyCluster enables a variety of RAC cluster configurations on Amazon EC2. Two or three node clusters are recommended in most cases. Clusters with four or more nodes can be used for extra HA or performance. It is possible to have clusters with more than four nodes containing several 2- or 3-node database sub-clusters. It is also possible to use SkyCluster for running single-instance databases with automatic failover. Nodes of a cluster can be in one availability zone or can be spread across availability zones."

ASM Disk Group Structure and Data Mirroring

FlashGrid Storage Fabric leverages proven Oracle ASM capabilities for disk group management, data mirroring, and high availability. In Normal Redundancy mode each block of data has two mirrored copies. In High Redundancy mode each block of data has three mirrored copies. Each ASM disk group is divided into failure groups—one failure group per node. Each disk is configured to be a part of a failure group that corresponds to the node where the disk is located. ASM stores mirrored copies of each block in different failure groups. See Figure 8.

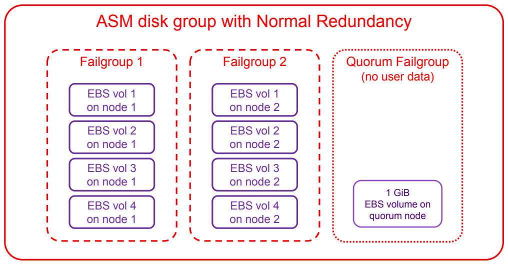


Figure 8. Example of a Normal Redundancy disk group in a 2-node RAC cluster

A typical Oracle RAC setup in Amazon EC2 will have three Oracle ASM disk groups: GRID, DATA, and FRA.

In a 2-node RAC cluster all disk groups must have Normal Redundancy. The GRID disk group containing voting files is required to have a quorum disk for storing a third copy of the voting files. Other disk groups also benefit from having the quorum disks as they store a third copy of ASM metadata for better failure handling.

In a 3-node cluster all disk groups must have High Redundancy in order to enable full Read-Local capability. The GRID disk group containing voting files is required to have two additional quorum disks, so it can have five copies of the voting files. Other disk groups also benefit from having the quorum disks as they store additional copies of ASM metadata for better failure handling. See Figure 9.

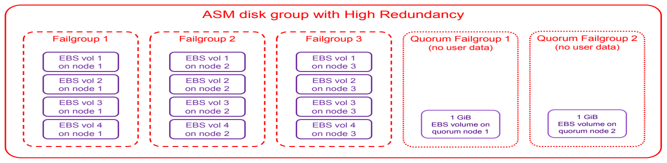


Figure 9. Example of a High Redundancy disk group in a 3-node RAC cluster

High Availability Considerations

FlashGrid Storage Fabric and FlashGrid Cloud Area Network have a fully distributed architecture with no single point of failure. The architecture leverages HA capabilities built in Oracle Clusterware, ASM, and Database.

Node Availability

Because all instances are virtual, failure of a physical host causes only a short outage for the affected node. The node instance will automatically restart on another physical host. This significantly reduces the risk of double failures.

A single Availability Zone configuration provides protection against loss of a database node. It is an efficient way to accommodate planned maintenance (e.g., patching database or OS) without causing database downtime. However, a potential failure of a resource shared by multiple instances in the same Availability Zone, such as network, power, or cooling, may cause database downtime.

Placing instances in different Availability Zones virtually eliminates the risk of simultaneous node failures, except for the unlikely event of a disaster affecting multiple data center facilities in a region. The trade-off is higher network latency. However, the network latency between AZs is less than 1 ms in most cases and will not have critical impact on the performance of many workloads.

Data Availability with EBS Storage

An Amazon EBS volume provides persistent storage that survives the failure of the node instance that the volume is attached to. After the failed instance restarts on a new physical node all its volumes are attached with no data loss.

Amazon EBS volumes have built-in redundancy that protects data from failures of the underlying physical media. The mirroring by ASM is done on top of the built-in protection of Amazon EBS. Together Amazon EBS plus ASM mirroring provide durable storage with two layers of data protection, which exceeds the typical level of data protection in on-premises deployments.

Data Availability with Local NVMe SSDs

Local NVMe SSDs are ephemeral (non-persistent), which means that in case an instance is stopped or fails over to a different physical host, the data on the SSDs attached to that instance is not retained. FlashGrid Storage Fabric provides mechanisms for ensuring persistency of the data stored on local NVMe SSDs. Mirroring data across two or three instances ensures that there is a copy of the data still available in the event of one instance losing its data. Placing the instances in different Availability Zones prevents the possibility of simultaneous failures of more than one instance. Placing one or two copies of data on NVMe SSDs and one copy on EBS provides high read bandwidth of NVMe and an additional layer of persistency of EBS. See Figure 10.

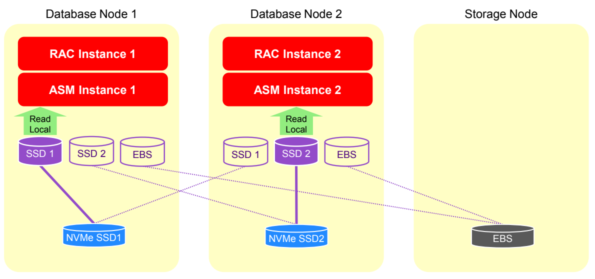


Figure 10. Two mirrors on NVMe SSDs plus one mirror on EBS

In the event of a loss of data on one of the instances with NVMe SSDs, FlashGrid Storage Fabric automatically reconstructs the affected disk groups and starts the data re-synchronization process after the failed instance is back online. No manual intervention is required.

Recommended Instance Types

An instance type must meet the following criteria:

- ➤ At least two vCPUs
- ➤ Enhanced Networking—direct access to the physical network adapter
- ➤ EBS optimized—dedicated I/O path for Amazon EBS, not shared with the main network

The following instance type families satisfy the above criteria and are optimal for database workloads:

- ➤ M4, M5: optimal memory-to-CPU ratio
- ➤ R4, R5: high memory-to-CPU ratio
- ➤ C5, Z1d: small memory-to-CPU ratio, for CPU-intensive workloads
- ➤ i3: high memory-to-CPU ratio, up to 15 TB of local NVMe SSDs
- ➤ i3en: high memory-to-CPU ratio, up to 100 Gb/s network, up to 60 TB of local NVMe SSDs
- ➤ X1, X1E: large memory size, large number of CPU cores

Quorum servers require fewer resources than database nodes. However, the above criteria are still important to ensure stable cluster operation. For example, m5.large or c5.large instances can be used as quorum servers. Using a T2 instance family for quorum servers is not supported. Note that there is no Oracle Database software installed on the quorum servers; hence, the quorum servers do not increase the number of licensed CPUs.

Single vs. Multiple Availability Zones

Using multiple Availability Zones provides substantial availability advantages. However, it does affect network latency. In the US-West-2 region for 8 KB transfers we measured 0.3 ms, 0.6 ms, and 1.0 ms between different pairs of Availability Zones, compared to 0.1 ms within a single Availability Zone.

Note that the different latency between different pairs of AZs provides an opportunity to optimize the selection of which AZs to use for database nodes. In a 2-node RAC cluster, it is optimal to place database nodes in the pair of AZs that has the minimal latency between them.

The latency impact in multi-AZ configurations may be significant for the applications that have high ratios of data updates. However, read-heavy applications will experience little impact because all read traffic is served locally and does not use the network.

EBS Volumes

Use of General Purpose SSD (gp2) volumes is recommended in most cases. However, use of gp2 volumes smaller than 1000 GB is not recommended due to expected variability in performance. Volumes of 1000 GB size and larger provide a guaranteed level of performance of 3 IOPS/GB up to 10,000 IOPS per volume. In most cases the following number of volumes and volume sizes are recommended:

- ➤ Usable disk group capacities below 8 TB: up to 8 gp2 volumes, 1 TB each
- ➤ Usable disk group capacities over 8 TB: 8 gp2 volumes, 1 TB to 16 TB each

All volumes in the same disk group must be of equal size.

Use of Provisioned IOPS SSD (io1) volumes may be cost efficient for configurations with very small capacities and small, but guaranteed, performance requirements below 1,000 IOPS.

Local NVMe SSDs

Use of local NVMe SSDs as the primary storage offers higher bandwidth and lower cost compared to Amazon EBS volumes. The i3 instance family includes NVMe SSDs up to 8 x 1900 GB with up to 16 GB/s of bandwidth and up to 3.3 mln IOPS. The new i3en instance family increases the local SSD capacity up to 8 x 7500 GB.

Reference Performance Results

The main performance-related concern when moving database workloads to the cloud tends to be around storage and network I/O performance. There is a very small to zero overhead related to the CPU performance between bare metal and VMs. Therefore, in this paper we focus on the storage I/O and RAC interconnect I/O.

CALIBRATE_IO

The CALIBRATE_IO procedure provides an easy way to measure storage performance, including maximum bandwidth, random IOPS, and latency. The CALIBRATE_IO procedure generates I/O through the database stack on actual database files. The test is read-only and it is safe to run it on any existing database. It is also a good tool for directly comparing the performance of two storage systems because the CALIBRATE_IO results do not depend on any non-storage factors, such as memory size or the number of CPU cores.

Test script:

```
SET SERVEROUTPUT ON;
DECLARE
  lat INTEGER;
  iops INTEGER;
  mbps INTEGER;
BEGIN DBMS_RESOURCE_MANAGER.CALIBRATE_IO (16, 10, iops, mbps, lat);
DBMS_OUTPUT.PUT_LINE ('Max_IOPS = ' || iops);
DBMS_OUTPUT.PUT_LINE ('Latency = ' || lat);
DBMS_OUTPUT.PUT_LINE ('Max_MB/s = ' || mbps);
end;
/
```

Results with two database nodes:

Cluster configuration	Max_IOPS	Latency	Max_MB/s
EBS storage	154,864	0	2,219
NVMe storage	1,375,694	0	27,338

Note that the CALIBRATE_IO results do not depend on whether the database nodes are in the same or different Availability Zones.

SLOB

SLOB is a popular tool for generating I/O-intensive Oracle workloads. SLOB generates database SELECTs and UPDATEs with minimal computational overhead. It complements CALIBRATE_IO by generating mixed (read+write) I/O load. AWR reports generated during the SLOB test runs provide various performance metrics. For the purposes of this paper we focus on the I/O performance numbers.

Results with two database nodes:

Cluster Configuration	Physical Write Database Requests	Physical Read Database Requests	Physical Read+Write Database Requests
EBS storage, same AZ	20,697 IOPS	100,539 IOPS	121,237 IOPS
EBS storage, different AZs	19,465 IOPS	92,081 IOPS	111,546 IOPS
NVMe storage, different AZs	23,913 IOPS	429,660 IOPS	453,573 IOPS

Test configuration details

EBS storage, same AZ, and different AZs

- ➤ Two database nodes, M4.16xlarge
- ➤ Four io1 20000 IOPS 400 GB volumes per node
- ➤ SGA size: 2.6 GB (small size selected to minimize caching effects and maximize physical I/O)

"FlashGrid SkyCluster offers a wide range of highly available database cluster configurations in AWS, ranging from cost-efficient 2-node clusters to large high-performance clusters. The combination of the proven Oracle RAC database engine, AWS availability zones, and fully automated Infrastructure-as-Code deployment provides high-availability characteristics exceeding those of the traditional on-premises deployments."

- 8KB database block size
- Schemas: 30 x 240 MB
- UPDATE_PCT= 20

NVMe storage, different AZs

- Two db nodes + storage node
- Instance type: i3.16xlarge
- (8) 1900GB NVMe SSDs per node
- SGA size: 4 GB (small size selected to minimize caching effects and maximize physical I/O)
- 8 KB database block size
- Schemas: 200 x 240 MB
- UPDATE_PCT= 5

Performance vs. On-Premise Solutions

Both EBS and NVMe SSD storage options are flash based and provide an order of magnitude improvement in IOPS and latency compared to traditional spinning hard drive–based storage arrays. With over 100K IOPS in both cases, the performance is comparable to having a dedicated all-flash storage array. It is important to note that the storage performance is not shared with other clusters or databases. Every cluster has its own dedicated set of EBS volumes or NVMe SSDs, which ensures stable and predictable performance with no interference from noisy neighbors.

NVMe SSDs enable speeds that are difficult or impossible to achieve even with dedicated all-flash arrays. Each NVMe SSD provides read bandwidth comparable to an entry-level flash array. The 27 GB/s bandwidth measured with 16 NVMe SSDs in a 2-node cluster is equivalent to a large flash array connected with 16 Fibre Channel links. Read-heavy analytics and data warehouse workloads can benefit the most from using the NVMe SSDs.

Compatibility

The following versions of software are supported with FlashGrid SkyCluster:

- Oracle Database: ver. 19*c*, 18*c*, 12.2, 12.1, or 11.2
- Oracle Grid Infrastructure: ver. 19*c*
- Operating System: Oracle Linux 7, Red Hat Enterprise Linux 7

Automated Infrastructure-as-Code Deployment

The FlashGrid SkyCluster Launcher tool automates the process of deploying a cluster. The tool provides a flexible web interface for defining cluster configuration and generating an Amazon CloudFormation template for it. The following tasks are performed automatically using the CloudFormation template:

- Creating cloud infrastructure: VMs, storage, and—optionally—network
- Installing and configuring FlashGrid Cloud Area Network
- Installing and configuring FlashGrid Storage Fabric
- Installing, configuring, and patching Oracle Grid Infrastructure
- Installing and patching Oracle Database software
- Creating ASM disk groups

The entire deployment process takes approximately 90 minutes. After the process is complete the cluster is ready for creating a database. Use of automatically generated standardized IaC templates prevents human errors that could lead to costly reliability problems and compromised availability.

Conclusion

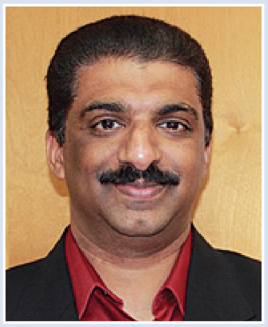
FlashGrid SkyCluster offers a wide range of highly available database cluster configurations in AWS, ranging from cost-efficient 2-node clusters to large high-performance clusters. The combination of the proven Oracle RAC database engine, AWS availability zones, and fully automated Infrastructure-as-Code deployment provides high-availability characteristics exceeding those of the traditional on-premises deployments. ▲

Artem Danielov is CTO at FlashGrid.

"NVMe SSDs enable speeds that are difficult or impossible to achieve even with dedicated all-flash arrays. Read-heavy analytics and data warehouse workloads can benefit the most from using the NVMe SSDs."

Oracle License Changes

by Biju Thomas


Biju Thomas

There were many licensing changes on the Oracle database product in 2019, especially with Oracle Database 19*c*. Here are a few major ones that every Oracle database customer must be aware of to plan for 2020 and beyond.

Three PDBs Included

Oracle introduced the multitenant architecture in Oracle Database 12*c*. Oracle 12*c* (12.1 and 12.2) allowed one user PDB as part of the Enterprise Edition license (single-tenant). With the multitenant license option, you can create up to 252 PDBs. In 12*c*, the Standard Edition did not have the multitenant option—only one PDB was allowed. If you are not licensed for Oracle multitenant, the container database architecture can still be in use (single-tenant mode)—with one user-created PDB, one user-created application root, and one user-created proxy PDB (no changes in 18*c*).

In Oracle Database 19*c*, Oracle allows you to create three user-created PDBs without any additional multitenant license; this is applicable for both the Standard Edition (SE2) and Enterprise Edition (EE). Multitenant architecture brings huge administrative advantages, especially to minimize patching, upgrade, and cloning. This license change will help accelerate the adoption of the container database architecture.

Multitenant, the Sooner the Better

The multitenant architecture was introduced in 2013 with Oracle Database 12*c*, but the adoption rate has been minimal—mainly, in my opinion, due to the cost involved. When you cannot create more than one PDB, why bother using the container architecture? And, for EBS customers, the database support lagged way behind and was stuck with 12.1 databases until recently.

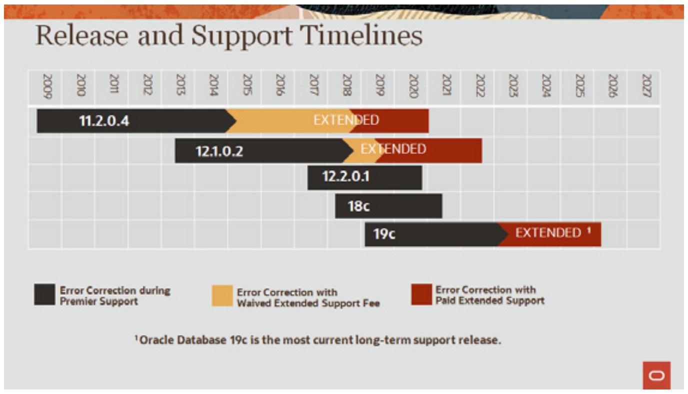
Though the non-CDB architecture had been "deprecated" since 12.1, it was available and supported to date, but things are changing. In 20*c*, the non-CDB architecture is "desupported," and I believe there will not be an option to create non-CDB databases.

Table 1 contains reminders of the support end-dates of database versions.

Pay attention to the patching end date. 12.2.0.1 and 18*c* are "annual releases," meaning you get no extended support. 19*c* is "long-term release," and it has premier support for four years and paid extended support for an additional three years.

Release	Patching End Date	Premier Support (PS) Ends	Extended Support (ES) Ends
19*c*	31-Mar-2023 without ES/ULA	31-Mar-2023	31-Mar-2026
Long-term support release	31-Mar-2026 without ES/ULA	31-Mar-2023	31-Mar-2026
18*c* **Annual Release**	08-June-2021	31-Mar-2023	Not Available
12.2.0.1 **Annual Release**	20-Nov-2020	31-Mar-2023	Not Available
12.1.0.2	31-Jul-2022 with paid ES, ULA, or EBS waiver	31-Jul-2018	31-Jul-2022
12.1.0.1 **for 12.1 Family**	Ended: 31-Aug-2016	31-Aug-2016	Not Available
11.2.0.4 **Terminal Release for 11.2 Family**	31-Dec-2020 with paid ES, ULA, or EBS waiver	31-Jan-2015	31-Dec-2020
11.2.0.3	Ended:	27-Jan-2015	27-Aug-2015
Patchset Release for 11.2 Family	27-Aug-2015	27-Jan-2015	27-Aug-2015

Table 1: Oracle Database Support End Dates



Picture from MOS Note: Release Schedule of Current Database Releases Doc ID 742060.1

Though not officially documented, Oracle product managers shared at OOW19, that 20*c* and 21*c* will be "annual release" and 22*c* may be the "long-term release." This means 20*c* will have a shorter life than 19*c*. Get your budgets and projects ready to upgrade all Oracle databases (including those that support Oracle E-Business Suite) to 19*c* multitenant architecture.

No RAC with SE2

Oracle Database Standard Edition 2 (SE2) was released in 2015. The major difference between SE and SE2 was the number of sockets and CPU cores allowed. SE allowed the use of 4 sockets (2 sockets for SE1) without any limitation on the number of CPU threads. In SE2, the number of sockets was limited to 2. SE2 also limits the maximum number of CPU threads per database instance to 16. You have to migrate SE licenses to SE2 licenses to upgrade a 12.1.0.1 or 11.2.x SE database to 12.1.0.2 or higher.

"Oracle RAC is no longer available with Standard Edition anymore."

Oracle Real Application Clusters are part of the SE and SE2 family. Even with the limitation of two sockets in SE2, if you can find two single-socket machines, you are allowed to create a two-node RAC instance with an SE2 license. Well, it changed in 19*c*. Oracle RAC is no longer available with Standard Edition anymore.

If you are using Oracle RAC with SE2 today, maybe it's time to evaluate alternatives. The primary alternatives are Oracle RAC (additional-cost EE option) on top of Oracle Enterprise Edition (licensed by core, not by socket) or cloud migration. The first thing you need is an evaluation of the business requirement to find the appropriate solution.

Oracle Analytics, Spatial and Graph

Oracle Analytics and Oracle Spatial and Graph used to be extra-cost options in the Oracle database.

Oracle Advanced Analytics empowers data and business analysts to extract knowledge, discover new insights, and make predictions—working directly with large data volumes in the Oracle database. Oracle Advanced Analytics provides a combination of powerful in-database algorithms and open-source R algorithms. Analytic capabilities are accessible via SQL and R languages, and through the SQL Developer extension or open-source R clients.

Graphs let you model data based on relationships in a more natural, intuitive way. They let you explore and discover connections and patterns in social networks, IoT, big data, data warehouses, and complex transaction data for applications such as fraud detection in banking, customer 360, and smart manufacturing. All graph features are now included as part of the Oracle database without any additional cost.

The spatial analysis enables a better understanding of complex interactions based on geographic relationships. All editions of Oracle Database now include comprehensive spatial analytics and data models.

As of December 5, 2019, the Machine Learning (formerly known as Advanced Analytics), Spatial and Graph features of Oracle database may be used for development and deployment purposes with all on-prem editions and Oracle Cloud Database Services (version 12.2 and above). Oracle Machine Learning is supported on SE2 as well.

EBS Customers: Time to Upgrade That Database!

E-Business Suite (EBS) now supports Oracle container architecture (finally!). An upgrade to Oracle Database 19*c* should be in the plans, not only to take advantage of the multitenant architecture but also to be on a supported database platform.

As you can see in Table 1, with the EBS waiver Oracle EBS customers can enjoy the benefits of Oracle Extended support on database versions 11.2.0.4 and 12.1.0.2. The extended support for 11.2.0.4 is ending in 10 months (Dec 2020), and 12.1.0.2 is ending in July 2022 (previously, July 2021). EBS customers have only a couple of years to plan and upgrade all their databases to 19*c* with container database architecture. EBS versions 12.1.3 and 12.2 support the 19*c* database.

As part of the upgrade to Database 19*c*, you will convert the EBS database to the CDB architecture with a single pluggable database (PDB). A CDB with one PDB (single-tenant) is currently the only certified deployment for Oracle E-Business Suite with Database 19*c*. We are not aware of a non-CDB architecture to be certified or supported for EBS with Database 19*c*. ▲

Biju Thomas is an Oracle ACE director and a frequent speaker at many Oracle conferences. Biju blogs at **http://www.bijoos.com/oraclenotes** *and you can follow him on Twitter (***@biju_thomas***) or Facebook (***@oraclenotes***). Also check out his Oracle Tidbits.*

First International NoCOUG SQL Challenge

We Have a Winner!

Reprinted from the August 2009 issue

The *First International NoCOUG SQL Challenge* was a great success; nine solutions were found by participants in seven countries and three continents. Alberto Dell'Era wins the contest for his wonderful solution using Discrete Fourier Transforms; the runner-up is André Araujo from Australia, who used binary arithmetic and common table expressions in his solution. The *August Order of the Wooden Pretzel*[1] will be bestowed on Alberto but the real prize is six books of his choice from the Apress catalog. André will receive a prize of six e-books of his choice. Thanks to Chen Shapira for publicizing the event in her blog, Dan Tow for helping to judge the contest, and Apress for donating the books.

The Challenge

An ancient 20-sided die (icosahedron) was discovered in the secret chamber of mystery at Hogwash School of Es-Cue-El. A mysterious symbol was inscribed on each face of the die. The great Wizard of Odds discovered that each symbol represents a number. The great wizard discovered that the die was biased: that is, it was more probable that certain numbers would be displayed than others if the die were used in a game of chance. The great wizard recorded this information in tabular fashion as described below.

Name	Null?	Type
FACE_ID	NOT NULL	INT
FACE_VALUE	NOT NULL	INT
PROBABILITY	NOT NULL	REAL

The great wizard then invited all practitioners of the ancient arts of Es-Cue-El to create an Es-Cue-El spell that displays the probabilities of obtaining various sums when the die is thrown N times in succession in a game of chance, N being a *substitution variable* or *bind variable*. The rules of the competition can be found in the May 2009 issue of the *NoCOUG Journal* and on the Web at **www.nocoug.org/SQLchallenge/FirstSQLchallenge.pdf**.

[1] Steven Feuerstein was asked the following question in an interview published in the May 2006 issue of the *NoCOUG Journal*: *"SQL is a set-oriented non-procedural language; i.e., it works on sets and does not specify access paths. PL/SQL on the other hand is a record-oriented procedural language, as is very clear from the name. What is the place of a record-oriented procedural language in the relational world?"* Steven replied: *"Its place is proven: SQL is not a complete language. Some people can perform seeming miracles with straight SQL, but the statements can end up looking like pretzels created by someone who is experimenting with hallucinogens. We need more than SQL to build our applications, whether it is the implementation of business rules or application logic. PL/SQL remains the fastest and easiest way to access and manipulate data in an Oracle RDBMS, and I am certain it is going to stay that way for decades."*

The Solutions

The modus operandi was for each participant to post their solutions on their blog or website; you can find them all with a simple Google search. The first solution—by **Laurent Schneider** from Switzerland—used the *CONNECT BY* clause to join the table to itself N times and the *SYS_CONNECT_BY_PATH* and *XMLQUERY* functions to perform the necessary additions and multiplications. The number of records generated by CONNECT BY grows exponentially and hurts performance.

The second solution—by **Craig Martin** from the USA—used the CONNECT BY clause to join the table to itself N times and logarithms to perform the necessary additions and multiplications. The number of records grows exponentially in this case too.

The third solution—by **Rob van Wijk** from the Netherlands—used the *Model* clause to generate records. The number of records grows exponentially in this case too.

The fourth solution—by **Vadim Tropashko** from the USA —used *recursive common table expressions* to generate records. The number of records grows exponentially in this case too. Recursive common table expressions are available in Microsoft SQL Server and DB2 but are not presently available in Oracle 11*g*R1. Rumor has that they will be available in Oracle Database 11*g*R2.

The fifth and sixth solutions—by **Alberto Dell'Era** from Italy—used advanced mathematical techniques such as *convolutions*, *Discrete Fourier Transforms*, and *Fast Fourier Transforms*. The Fast Fourier Transform method is an efficient way of calculating Discrete Fourier Transforms and was implemented using the Model clause.

The seventh solution—by **Fabien Contaminard** from France—was based on the *multinomial* probability distribution, an extension of the binomial distribution.

The eighth solution—by a blogger named Cd-MaN from Romania—used *pipelined table functions* and recursion. The solution was demonstrated in a Postgres database but can easily be adapted for use in an Oracle Database. The use of recursion means that this is not a pure SQL solution.

The ninth solution—by **André Araujo** from Australia—used binary arithmetic and common table expressions.

Judges' Decision

Dan Tow authored the following statement on behalf of the judging committee:

To begin, we'd like to congratulate the contestants on finding so many different and clever ways to solve this problem, and on making the problem of picking a winner so difficult for us, personally. Each of the solutions had major advantages to recommend it. We were also very pleasantly surprised at the global extent of the entries, with entries from seven nations and three continents!

The criteria for the judging were stated in advance at **www.nocoug.org/SQLchallenge/FirstSQLchallenge.pdf**. The main criteria that separated the top scores from the rest, given that they were all quite good as technical solutions from one perspective or another, were the inclusion of commentary and test results, which were minimal or altogether lacking from most entries. (Hey, we understand that you're busy, so we're not surprised to see these missing or minimal, but they were important to getting a win here!)

There were elegant solutions using the Model clause, including one by Alberto Dell'Era that implemented a solution using Fast Fourier Transforms that was technically amazing, well-documented and tested, and scaled better than any other solution, with order N * Log(N) scaling, almost linear up to enormous numbers of throws of the dice. However, these used "iterate" loops that we believe violated the contest requirement that "Solutions that use procedural loops to multiply probabilities are not eligible" stated at the top of the "Judges' Statement." (We know that there are wonderful, super-efficient ways to solve this problem in procedural code like C, but the point behind that unbendable requirement was to get contestants "thinking in SQL," doing the job in a set-wise manner, not just finding ways to bend SQL into doing what we'd be better off doing in C, procedurally.) The "procedural loop" component in these solutions was really minimal and easy to miss, even, in a casual examination of the code, but we think we have to stick with the prestated rules here and disqualify those solutions, even while we admire them.

Scaling almost as well (at order N * N), and also very well documented and tested, both from the perspective of performance and functionality, was the amazing Fourier-Transform-based solution, **www.adellera.it/investigations/nocoug_challenge/index.html** also by Alberto Dell'Era from Italy, that we think has to be declared the winner here, with no procedural loops and good-to-excellent scores in every stated judging criteria.

The runner-up choice is difficult, too, but we'd probably have to go with André Araujo of Australia, **www.pythian.com/news/2385/nocoug-sql-challenge-entry**. Of all the solutions, his probably best combined straightforward (very clever, but still straightforward to the reader!), portable SQL that could be easily understood and maintained by a developer without an advanced degree in mathematics, with fairly scalable and well-tested SQL that ran well up to quite high numbers of throws of the die ("N"). It is true that the SQL had a hard-coded limit of N = 511 (a limit that André documented well, to the credit of the solution), and that functional limit lost a few points, but we should keep in mind that this high value of N is one that most of the implementations (other than Alberto Dell'Era's) would never reach in our lifetime, anyway, owing to their comparative lack of scalability—being logically correct at high N is worth nothing if the program never finishes! If we had to actually maintain one of these in a production environment, and we didn't anticipate needing results at very high values of N, we'd probably go with André's solution, just because we'd be frightened of long-term maintenance on the high mathematics of Alberto Dell'Era's brilliant but more complex and technically harder-to-follow solution.

Analysis of the Winning Solution

Alberto recognized that the contents of the die table define a mathematical function and that the process of joining the table with itself and grouping the results is the so-called *convolution* of this function with itself. Throwing the die N times is therefore equivalent to performing N – 1 convolutions. For example, for N = 3, we have to perform two convolutions. This is best expressed using *common table expressions* as follows. Notice that the definition of the second convolution references the first convolution.

```
WITH

first_convolution AS
(
  SELECT face_value, SUM (probability) AS probability
  FROM
  (
    SELECT
      d1.face_value + d2.face_value AS face_value,
      d1.probability * d2.probability AS probability
    FROM die d1 CROSS JOIN die d2
  )
  GROUP BY face_value
),

second_convolution AS
(
  SELECT face_value, SUM (probability) AS probability
  FROM
  (
    SELECT
      d1.face_value + d2.face_value AS face_value,
      d1.probability * d2.probability AS probability
    FROM first_convolution d1 CROSS JOIN die d2
  )
  GROUP BY face_value
)

SELECT face_value, probability
FROM second_convolution
ORDER BY face_value;
```

The most common solution of the problem requires an N-way cross join. Convolutions have obvious advantages over an N-way cross join because they keep the size of intermediate results in check. The question is how to compute the required N – 1 convolutions with a single SQL statement if the value of N is not known in advance. One solution is to recursively invoke a *table function* as was done by the Romanian contestant; that is, we have to resort to procedural programming. Alberto was able to avoid procedural programming using a *Fourier transform;* that is, a certain function whose definition is derived from the original function. The Fourier transform has the interesting property that the transform of the convolution of functions is the simple product of the individual transforms. Therefore, the Fourier transform of the N-way convolution of our function with itself is the Nth power of the Fourier transform of the function. The Fourier transform is straightforward to compute and—with a little mathematical trick involving a conversion from the Cartesian coordinate system to the polar coordinate system—the Nth power of the Fourier transform is also straightforward to compute. At this point, Alberto has the Fourier transform of the N-way convolution. To obtain the result that he really needs, all that is left for him to do is to compute the *Inverse Fourier Transform* of the Fourier transform. An explanation of Fourier Transforms can be found on the Web; efficient C-language implementations can also be readily found.

For readability and maintainability, Alberto uses a sequence of *common table expressions*. The following is a simplified version of his solution; the full solution handles more general cases and uses some tricks to reduce the number of scientific computations. Hints to guide the optimizer and improve efficiency are included in the version shown below.

First Alberto creates a one-column table of sequence numbers using the CONNECT BY method; this table is used several times in the rest of the solution. The number of elements in the table is one more than the product of the number of sides of the die and the number of throws.

```
sequence AS
(
  SELECT /*+ NO_MERGE */
    LEVEL - 1 AS n
  FROM dual
  CONNECT BY LEVEL <= (:N * :sides + 1)
)
```

Alberto then constructs a *discrete function* whose domain is the numbers in the sequence table. The function is called *discrete* because it is only defined for certain discrete values—not for all values in a range as in the case of a *continuous* function. Whenever possible, the function uses the values in the die table. If a value is not found, the value of the function is set to zero. This is done using an *outer join*.

```
function AS
(
  SELECT /*+ NO_MERGE */
    n,
    COALESCE(probability, 0) AS x
  FROM sequence LEFT OUTER JOIN die ON (n = face_value)
)
```

Alberto then computes the *Fourier transform* of the discrete function. There's some advanced math going on here but it's easy to take in small doses; you'll recognize Pi as the well-known mathematical constant. A *cross join* with the Sequence table is required to calculate sums.

```
transform AS
(
  SELECT /*+ NO_MERGE LEADING(function) */
    sequence.n,
    SUM(x * COS(-2 * :Pi * sequence.n * function.n / (:N * :sides + 1))) AS x,
    SUM(x * SIN(-2 * :Pi * sequence.n * function.n / (:N * :sides + 1))) AS y
  FROM function CROSS JOIN sequence
  GROUP BY sequence.n
)
```

In another little dose of math, Alberto switches to *polar form* for ease of further computation.

```
polar AS
(
  SELECT /*+ NO_MERGE */
    n,
    SQRT((x * x) + (y * y)) AS r,
    CASE
      WHEN ABS(y) < 0.000001 AND ABS(x) < 0.000001 THEN 0
      ELSE ATAN2(y, x)
    END AS theta
  FROM transform
)
```

Computing the Nth power of the Fourier transform is then very easy.

```
power AS
(
  SELECT /*+ NO_MERGE */
    n,
    POWER(r, :N) AS r,
    theta * :N AS theta
  FROM polar
)
```

Alberto has no more use for the polar form, so he converts back to Cartesian form.

```
cartesian AS
(
  SELECT /*+ NO_MERGE */
    n,
    r * COS(theta) AS x,
    r * SIN(theta) AS y
  FROM power
)
```

So far, Alberto has calculated the Fourier transform of the discrete function and computed its Nth power. As explained earlier, this is the Fourier transform of the N-way convolution of the discrete function. To obtain the convolution itself, Alberto computes the *Inverse Fourier Transform* using another *cross join* with the Sequence table.

```
convolution AS
(
  SELECT /*+ NO_MERGE LEADING(cartesian) */
    sequence.n,
    SUM
    (
      x * COS(+2 * :Pi * cartesian.n * sequence.n / (:N * :sides + 1)) -
      y * SIN(+2 * :Pi * cartesian.n * sequence.n / (:N * :sides + 1))
    ) / (:N * :sides + 1) AS x
  FROM cartesian CROSS JOIN sequence
  GROUP BY sequence.n
)
```

Finally, Alberto can display the results. The result has more than 30 digits of decimal precision; only 30 are displayed in the interests of accuracy.

```
SELECT
  n AS face_value,
  ROUND(x, 30) AS probability
FROM convolution
WHERE n >= :N
ORDER BY n;
```

As you can see, Alberto's solution used advanced mathematical techniques, but it is not very long and the use of *common table expressions* makes it quite readable. We have a winner! ▲

NoCOUG Conference #132

The Chinese dim sum was fabulous. The speakers and presentations were awesome. The T-shirts and raffle prizes were cool. The post-conference wine-and-cheese reception was over the top. But the attendance at the fall conference was disappointing—the lowest in 33 years.

SkyCluster
by flashgrid

Database HA in the Cloud

- Proven **Oracle RAC** engine
- AWS, Azure, GCP
- HA clustering with 2+ nodes
- Infrastructure-as-Code
- 24/7 support

Launch in **1 hour**
in **your cloud** account!

www.flashgrid.io

NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED

memSQL

The No-Limits Database™

The cloud-native, operational database
built for speed and scale

SPEED
Accelerate time to insight with a database built for ultra fast ingest and high performance query

SCALE
Build on a cloud-native data platform designed for today's most demanding applications and analytical systems

SQL
Get the familiarity & ease of integration of a traditional RDBMS and SQL, but with a groundbreaking, modern architecture

Learn more at **memsql.com**