SQL SERVER MVP DEEP DIVES

Volume 2

MANNING

SAMPLE CHAPTER

EDITED BY

Kalen Delaney • Louis Davidson • Greg Low • Brad McGehee • Paul Nielsen • Paul Randal • Kimberly Tripp

MVP AUTHORS

Johan Åhlén • Gogula Aryalingam • Glenn Berry • Aaron Bertrand • Kevin G. Boles • Robert Cain • Tim Chapman • Denny Cherry • Michael Coles • Rod Colledge
John Paul Cook • Louis Davidson • Rob Farley • Grant Fritchey • Darren Gosbell • Sergio Govoni • Allan Hirt • Satya Jayanty • Tibor Karaszi • Jungsun Kim • Tobiasz Koprowski • Hugo Kornelis • Ted Krueger • Matija Lah • Greg Low • Rodney Landrum • Greg Larsen • Peter Larsson • Andy Leonard • Ami Levin • John Magnabosco
Jennifer McCown • Brad McGehee • Siddharth Mehta • Ben Miller • Allan Mitchell • Tim Mitchell • Luciano Moreira • Jessica Moss • Shahriar Nikkhah • Paul Nielsen
Robert Pearl • Boyan Penev • Pedro Perfeito • Pawel Potasinski • Mladen Prajdić • Abolfazl Radgoudarzi • Denis Reznik • Rafael Salas • Edwin Sarmiento
Chris Shaw • Gail Shaw • Linchi Shea • Jason Strate • Paul Turley • William Vaughn • Peter Ward • Joe Webb • John Welch • Allen White • Thiago Zavaschi

Operation Smile The authors of this book support the children of Operation Smile

SQL Server MVP Deep Dives
Volume 2

Edited by Kalen Delaney ▪ Louis Davidson ▪ Greg Low
Brad McGehee ▪ Paul Nielsen ▪ Paul Randal ▪ Kimberly Tripp

Chapter 2

brief contents

2 "Yes, we are all individuals" A look at uniqueness in the world of SQL

Rob Farley

This chapter looks at the idea of uniqueness, in both database design and query design. I explain the ways in which uniqueness can be enforced and compare the features of each. I then examine the idea of uniqueness within datasets, and challenge some basic methods that people use to create `GROUP BY` clauses. I hope you gain a new appreciation for uniqueness so that you can echo the Monty Python team in shouting "Yes, we are all individuals!"

NOTE For all my examples, I'll use the AdventureWorks sample database, running on a SQL Server 2005 instance, connecting with SQL Server 2008 R2 Management Studio. I prefer the way that the later versions of SSMS display execution plans, but want to demonstrate functionality that applies in earlier versions as well as the newer ones. You can download AdventureWorks by searching for it at codeplex.com.

Introducing uniqueness

Uniqueness is often taken for granted—we learned about it in our earliest days of database development. But I plan to show you that uniqueness is something that shouldn't be taken lightly at all. It's a powerful feature that you should consider carefully—not only when designing databases, but also when writing queries.

Constrained to uniqueness

With unconstrained data, anything goes. This isn't the way you like it—right from the word "Go," you define your tables as having a list of columns and constrain those columns to use particular types. You freely implement columns that are automatically populated with the next number in a list, or that use default constraints to

provide an initial value. You even use computed columns for those times when you want a column to be forced into a particular value. As much as the idea of constraining your systems sounds restricting, it's a design feature on which we all thrive.

Without uniqueness, you might not be able to tell the difference between two rows. Even though you might want to record the same data twice, you'd still like to be able to tell the difference between two rows, so having a way of making sure each row is uniquely identifiable is an incredibly useful feature—one that you often take for granted.

You have a few choices for constraining your tables to unique values.

Primary keys

Probably the most common cry of database architects is that every table must have a *primary key*—that is, a column (or collection of columns) whose values uniquely identify each row (and that don't allow NULL values). Many argue that without a primary key, that thing you've created isn't actually a table, despite what SQL Server calls it. I'm not going to try to argue one way or the other—I think everyone agrees that having primary keys on tables is a good idea.

In SQL Server Management Studio's Object Explorer pane, you see primary keys reflected using a gold key icon in the list of Columns, and listed again in the Keys section of the table properties (see figure 1).

By default, the primary key columns of a table are used as the keys of a unique clustered index on the table. You can see one listed in figure 1, called

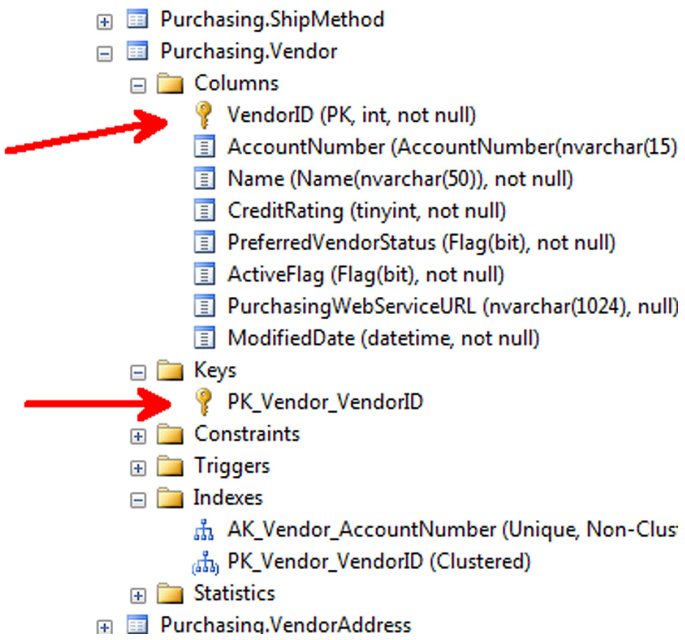

Figure 1 Gold key icons are used to indicate primary keys.

`PK_Vendor_VendorID`. If I try to drop this index, using the command `DROP INDEX Purchasing.Vendor.PK_Vendor_VendorID;`, I get an error, as shown here:

```
Msg 3723, Level 16, State 4, Line 3
An explicit DROP INDEX is not allowed on index 'Purchasing.Vendor.
PK_Vendor_VendorID'. It is being used for PRIMARY KEY constraint enforcement.
```

A primary key doesn't need to be enforced by the clustered index; it can be done with a nonclustered index instead. Attempting to remove a nonclustered index that enforces a primary key causes the same error to occur.

In your databases, you use the values stored in tables' primary key columns to identify the items represented in the table. In this example, the vendor is represented by their VendorID, and you'd refer to them as Vendors 1, 2, and 3, despite calling them by Name or Account Number in the nontechnical world.

Unique constraints

Another option for enforcing uniqueness is to use a unique constraint. This isn't designed to replace the primary key, but rather to provide an alternative to the primary key. This kind of constraint is often used for natural keys—a column (or set of columns) that you recognize as uniquely identifying each row in the real world but that hasn't been used as the primary key for practical or technical reasons (such as its ability to allow a NULL value—a unique constraint can have a single NULL value, whereas a primary key can't—or its size, particularly if they're referenced in many other tables). A set of columns that could potentially be used as a primary key is known as a candidate key.

Good examples of these can be found in the names of things. You can usually identify things by their names. Despite there being many Rob Farleys in the world, in many contexts my name is unique. On Twitter I'm @rob_farley, a handle that (hopefully quite obviously) no one else has. My profile also identifies me as user 14146019. In AdventureWorks, if I constrain the names of Product subcategories using a unique constraint, this is represented by a blue key icon in Object Explorer, as you can see in figure 2.

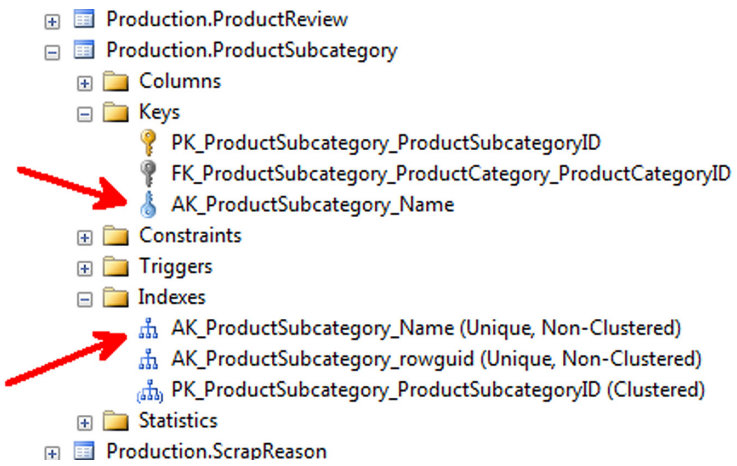

Figure 2 Blue key icons are used to indicate unique constraints.

Notice that the unique constraint is enforced by a unique index. The experience of attempting to drop this index is different from that of trying to drop one used by a primary key. This time the `DROP` command is successful, but the unique constraint is also dropped. Similarly, if the unique constraint is dropped, the unique index is also (and perhaps predictably) dropped.

Unique indexes

You've already seen that unique indexes are created to maintain both primary keys and unique constraints. But what if a unique index is created without an accompanying constraint? Is the effect the same, or is something lost?

Even without a primary key or unique constraint, a unique index will provide the same functionality as far as the ability to restrict the data. No entry in the Keys section of Object Explorer is seen, but a unique index can be seen in the Indexes section, as expected.

Unique constraint or unique index?

Having suggested that uniqueness can be maintained by either a unique constraint or a unique index, let's consider the differences between them, and I'll point out a couple of misconceptions about them, as well.

Advantages of the unique index

A unique index is created in the same way that any index is created, with the exception that it involves the word `UNIQUE`:

```
CREATE UNIQUE INDEX uix_Product_Name ON Production.Product(Name);
```

A number of additional options are available that are specific to indexes, as well as many that are also available when creating a unique constraint. The most common example I can think of is `FILLFACTOR`, but there are a couple of other options that aren't available to unique constraints—ones that I consider important and a significant factor in the decision about whether to use a unique constraint or a unique index:

- Included columns
- Filters

I'm sure all readers of this chapter will appreciate the significance of included columns as a performance-tuning tool. If an index `INCLUDE`s additional columns, then a copy of that data is stored at the leaf level of the index, potentially avoiding the need for an expensive lookup to find that data in the underlying table storage (clustered index or heap). Any time an index is used, whether it be a scan or a seek, there's a benefit to be seen from included columns. Included columns do add to the size of the index, and this data must be kept up-to-date whenever data in an included column is altered. But these downsides are often considered minor in comparison to the benefits of avoiding lookups.

Filtered indexes, new in SQL 2008, provide another significant tool in the performance tuner's utility belt. When I think of indexes, I often consider the analogy of the phonebook, and consider the Yellow Pages to be like a filtered index, listing phone numbers ordered by business type, but with a filter such as `WHERE IsBusiness = 'Y'` (or more strictly `WHERE HasPaidToAdvertiseInTheYellowPages = 'Y'`). We all use the Yellow Pages and realize its significance in helping us find the information we need, so I expect I don't need to describe the benefits of filtered indexes.

In the context of uniqueness, a filtered index presents an interesting situation. It means you can constrain your data so that Products that are colored Red must have different names from each other, but there can be duplicate names among other colors. A filtered index doesn't constrain all your data—just the data that satisfies the filter. Although this scenario is useful, you end up with a unique index that's only applicable to certain parts of the data.

Advantages of the unique constraint

In my experience, the benefits of using unique constraints are more about people than technology. Unique constraints are a logical feature used in database design, whereas indexes (of any kind) are a physical feature often seen primarily as a performance tool. Performance-tuning consultants may recommend that indexes be created, but database architects may recommend that constraints be created. I play both roles and find myself seeing things somewhere in between.

It's completely correct to have the database architect (designer, if you prefer) indicate when a field should be constrained to uniqueness. They're the ones who need to understand the business and the impact of such a decision. The performance-tuning expert should seek to understand the business, but ultimately is more concerned about the execution plans that are being created by queries and deciding whether a carefully constructed index would help. I'd like to suggest that the database architect consider the queries that will be needed and take a more active part in designing the indexes to be used, and that the performance-tuning expert consider the significance of unique constraints and appreciate the part that uniqueness (and all aspects of database design) have on performance.

Uniqueness in results

It's one thing to be able to constrain a table so that it's only populated with unique data, but to have unique data in a result set is a slightly different matter. The keyword `DISTINCT` is one of the first that you learn when beginning T-SQL, but it's also the first one to earn the stigma of "keyword to be avoided where possible."

The good and the bad of DISTINCT

`DISTINCT` has obvious uses. If there are duplicates in a dataset, then slipping the keyword `DISTINCT` in after `SELECT` will manage to de-duplicate the result set data. This is the good. But as most database professionals know, simply using `DISTINCT` in a `SELECT` query to filter out duplicates from the entire dataset can often hide bigger problems.

A better way of removing duplicate rows from a result set is to ask why they're there in the first place. Duplicates are often present because of mistakes (by query writers) in join predicates, or for a variety of other reasons. DISTINCT should never be used to "fix" a query, but only when you know ahead of time it'll be required, such as when you're querying to find the list of different Product colors sold:

```
SELECT DISTINCT p.Color
FROM Production.Product AS p
WHERE EXISTS (      SELECT *
                    FROM Sales.SalesOrderDetail AS s
                    WHERE s.ProductID = p.ProductID);
```

DISTINCT or GROUP BY

As anyone who's ever written a query like this knows, the next question that the client asks is "How many are there of each color?" At this point, you rewrite the query to use GROUP BY instead, which allows aggregate functions to be applied over the rows that share the same color:

```
SELECT p.Color, COUNT(*) AS NumProducts
FROM Production.Product AS p
WHERE EXISTS (      SELECT *
                    FROM Sales.SalesOrderDetail AS s
                    WHERE s.ProductID = p.ProductID)
GROUP BY p.Color;
```

The functionality provided here is similar to using the DISTINCT keyword, with the difference being slightly beyond the accessibility of aggregates. The most significant difference between the two is regarding the treatment of nonaggregate functions and subqueries. Figure 3 shows two queries (and their plans) that might seem similar in functionality but that are subtly and significantly different.

Notice the order of operations and widths of arrows. Using DISTINCT, the de-duplication happens on the dataset including the computed data. It applies the concatenation on every row before looking for duplicates. With GROUP BY, the query optimizer

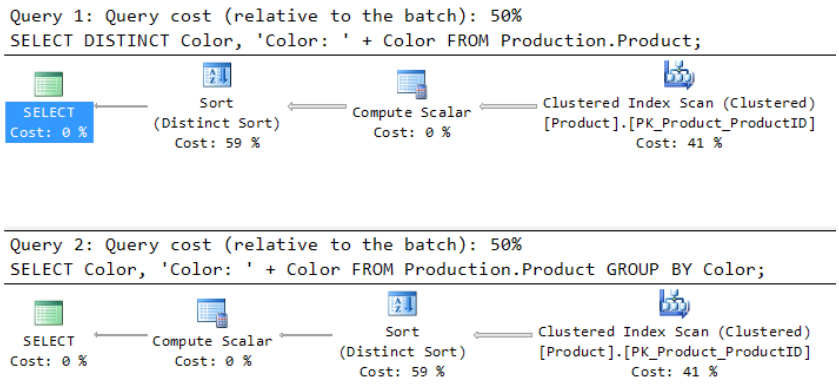

Figure 3 Comparing DISTINCT and GROUP BY

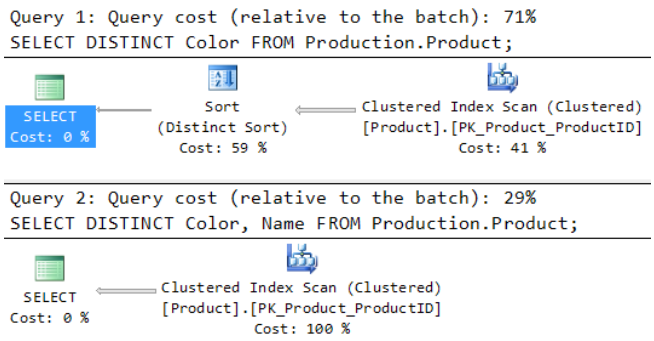

Figure 4 The sort operation disappears with an extra column.

does a little more work, realizing that the computation can be applied on the distinct values. In this scenario, it uses slightly less effort in comparing the values for making them unique, and then only needs to apply the computation 10 times, rather than more than 500 times.

I can't think of a single situation where `DISTINCT` is a better option than `GROUP BY`, except for readability and conciseness. The fact that `GROUP BY` requires that things be listed in the `GROUP BY` to allow their use in the `SELECT` clause—thereby forcing query writers to type more—encourages the lazy use of `DISTINCT` and frequently queries that perform more poorly. More on that soon.

Are they needed at all?

I ran a simple query, selecting the different colors in the Production.Product table, and I accomplished this using a sort (Distinct Sort) operator. If I add an extra column (the Name column), though, you see that the sort disappears, as shown in figure 4.

Of course, there's trickery going on here. I'm adding a column that's already known to be unique. Because the query optimizer knows that the Name column is already unique, it realizes that the combination of Name and Color must also be unique. Applying an operation couldn't make it anymore unique—there aren't degrees of uniqueness. Notice that the unique index isn't even being used here, but if you remove it temporarily, the impact is significant, as shown in figure 5.

Removing the constraint doesn't suddenly cause the data to be nonunique; it's simply that the query optimizer doesn't know it for certain. This is one of those scenarios that I described in Chapter 40 of the first *SQL Server MVP Deep Dives* book, titled "When is an unused index not an unused index?" If you don't have that book, I recommend you go and buy it immediately—it's an excellent resource, and none of the authors make a cent from it. Just like with this book, all the proceeds go to charity.

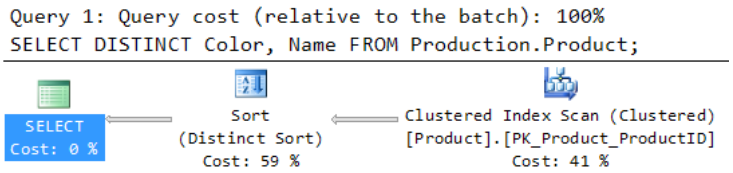

Figure 5 The Sort operation reappears when the uniqueness isn't already known.

```
SELECT p.Name, p.Color, SUM(o.OrderQty) AS NumSold
FROM Production.Product AS p
LEFT JOIN Sales.SalesOrderDetail AS o
ON o.ProductID = p.ProductID
GROUP BY p.Name, p.Color;
```

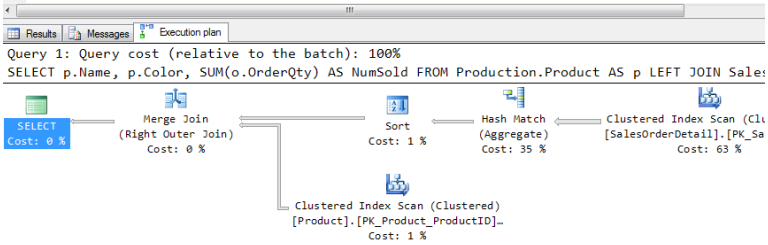

Figure 6 Counting the number of products sold by product name and color, using a unique constraint

Unnecessary grouping

The same effect as in figures 4 and 5 can be seen using the `GROUP BY` clause as well as `DISTINCT`, but you have more options available to you. Consider a `GROUP BY` clause in which you're grouping by the product name and color. Perhaps you're counting how much of a product has been sold. A query and plan can be seen in figure 6.

In this situation, you see that the aggregation is performed using only data from the table of order details. The product details are added to the mix later. You're grouping by the product itself, displaying the name and color. Because the product name is unique, there's no difference whether you group by the name or any other unique feature of the product.

If the product name weren't unique, you'd be in a situation similar to counting the population of towns by their name. You might not want the population of Boston to be listed as more than 4 million if you're thinking of Boston in Lincolnshire, UK (population approximately 60,000). Differentiating by name simply doesn't always cut it. Let's remove the unique constraint and look at the query again (see figure 7). The cost of this plan is significantly more, as you'd expect.

```
SELECT p.Name, p.Color, SUM(o.OrderQty) AS NumSold
FROM Production.Product AS p
LEFT JOIN Sales.SalesOrderDetail AS o
ON o.ProductID = p.ProductID
GROUP BY p.Name, p.Color;
```

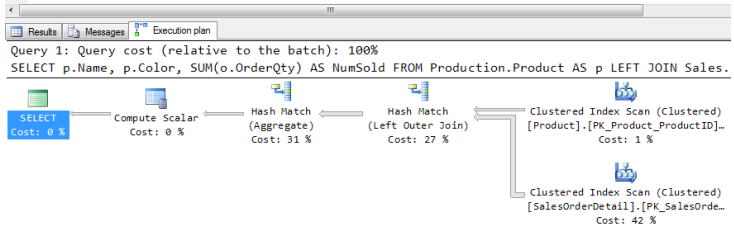

Figure 7 Counting the number of products sold by product name and color, without using a unique constraint

Being guided by "that" error

At this point, I want to break you of a common practice. When people write a `GROUP BY` clause, their behavior is driven by wanting to avoid that error we all know so well:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Production.Product.Name' is invalid in the select list because
it is not contained in either an aggregate function or the GROUP BY clause.
```

This is the error that occurs when the `GROUP BY` clause is ignored. When `GROUP BY p.Name` is included, the error complains about the Color column. Soon, all the required fields are added and the error disappears, leaving you with a valid query that probably works but that might not be ideal. Naturally, query writers don't go through this process every time, knowing how the construct works, but still the vast majority of query writers create their `GROUP BY` clause using only the nonaggregated fields from their `SELECT` clause.

But if you ignore the `SELECT` clause (and the `HAVING` clause and `ORDER BY` clause, which can also produce that error) and acknowledge that you want to group by the product itself, rather than just its name and color, then you can introduce a unique set of columns to the `GROUP BY` clause and see the old plan return. This unique set of columns would be the product's primary key. Grouping by the product's primary key means that every other column from `p` in the `GROUP BY` clause can be completely ignored—you can't make it any more unique (see figure 8). The extra columns are now only present in the `GROUP BY` clause to satisfy that error, giving you a far more ideal query. (Please don't use this as an excuse to remove other unique constraints and indexes—they're still important for data consistency.)

This change would mean that two products with the same name and color would appear with a corresponding row each. But in many situations, this would make sense. It could be fine to list both Bostons with their corresponding populations, rather than

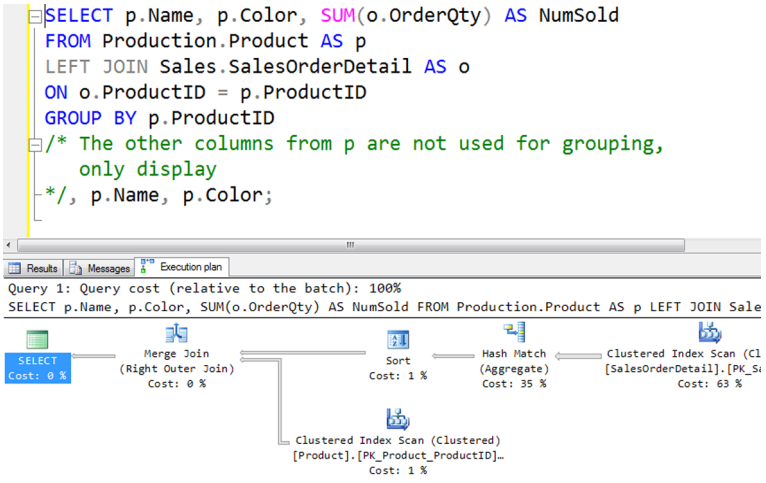

Figure 8 Counting the number of products sold, grouping by the primary key

combining them into a single row (even if only to highlight the need for further distinguishing information).

Summary

Uniqueness can be applied to data in a number of ways, to tables using primary keys, unique constraints and unique indexes, and to datasets using `DISTINCT` and `GROUP BY`. However it's done, it's important to consider the impact on data and the advantages to having data constrained like this. You should also consider the idea of grouping by the primary key on a table, even if that field doesn't appear in the `HAVING`, `SELECT`, or `ORDER BY` clauses.

About the author

Rob Farley is the owner and principal consultant of LobsterPot Solutions Pty Ltd., an Australian-based company specializing in SQL Server and business intelligence and the first Australian company to be a Gold Competency Partner in the Microsoft Partner Network. He's a Microsoft Certified Trainer and a regular conference presenter both around Australia and overseas. He heads up the Adelaide SQL Server User Group and has received the Microsoft MVP Award for SQL Server every year since 2006. Rob is a past branch executive committee member of the Australian Computer Society, and he's recently been appointed to the PASS Board of Directors. He's a dedicated husband and father, the author of two chapters in the first volume of *SQL Server MVP Deep Dives*, and is proud to be able to contribute again to this volume.

SQL SERVER/DATABASE

SQL SERVER MVP DEEP DIVES Volume 2

EDITORS: Kalen Delaney • Louis Davidson • Greg Low • Brad McGehee • Paul Nielsen • Paul Randal • Kimberly Tripp

To become an MVP requires deep knowledge and impressive skill. Together, the 64 MVPs who wrote this book bring about 1,000 years of experience in SQL Server administration, development, training, and design. This incredible book captures their expertise and passion in sixty concise, hand-picked chapters.

SQL Server MVP Deep Dives, Volume 2 picks up where the first volume leaves off, with completely new content on topics ranging from testing and policy management to integration services, reporting, and performance optimization. The chapters fall into five parts: Architecture and Design, Database Administration, Database Development, Performance Tuning and Optimization, and Business Intelligence.

What's Inside

- Discovering servers with PowerShell
- Using regular expressions in SSMS
- Tuning the Transaction Log for OLTP
- Optimizing SSIS for dimensional data
- Real-time BI
- Much more

This unique book is your chance to learn from the best in the business. It offers valuable insights for readers of all levels.

Written by 64 SQL Server MVPs, the chapters were selected and edited by Kalen Delaney and Section Editors Louis Davidson (Architecture and Design), Paul Randal and Kimberly Tripp (Database Administration), Paul Nielsen (Database Development), Brad McGehee (Performance Tuning and Optimization), and Greg Low (Business Intelligence).

Manning Publications and the authors of this book support the children of Operation Smile

For online access to the authors go to manning.com/SQLServerMVPDeepDivesVol2.
For a free ebook for owners of this book, see insert.

ISBN 13: 978-1-617290-47-3
ISBN 10: 1-617290-47-5
55999
9 781617 290473

MANNING $59.99 / Can $62.99 [INCLUDING eBOOK]