

Explore performance-based database design

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/explore-performance-based-design/1-introduction>

Introduction

Completed

Database design plays a crucial role in determining the performance of a database, even though it may not always be within the control of the database administrator. Often, you might find yourself working with third-party vendor applications that you didn't develop. In such cases, it becomes essential to understand the underlying design principles and make necessary adjustments to optimize performance. Proper database design ensures that the system can handle the expected workload efficiently, whether it's an online transaction processing (OLTP) system or a data warehouse. By aligning the design with the specific needs of the workload, you can significantly enhance the overall performance and responsiveness of the database.

Whenever possible, it's important to design your database with the workload in mind. For OLTP systems, the focus should be on minimizing transaction times and ensuring data integrity. This involves designing tables and indexes that support quick lookups and updates. On the other hand, data warehouse workloads require a design that facilitates complex queries and large-scale data analysis. This might involve denormalizing tables to reduce the number of joins required for queries or using partitioning to manage large datasets effectively. By tailoring the design to the specific workload, you can ensure that the database performs optimally under various conditions.

Many design decisions can have a profound impact on database performance. One such decision is the choice of datatypes for columns. Selecting the appropriate datatype can reduce storage requirements and improve query performance. For example, using integer types for numeric data instead of character types can lead to faster processing and reduced storage space. Additionally, proper indexing strategies can greatly enhance query performance by allowing the database engine to quickly locate the required data. By carefully considering these and other design aspects, you can create a database that not only meets the functional requirements but also performs efficiently and reliably.

2. Describe normalization

<https://learn.microsoft.com/en-us/training/modules/explore-performance-based-design/2-describe-normalization>

Describe normalization

Completed

Database normalization is a design process used to organize data into tables and columns within a database. Each table should contain data related to a specific entity and only include information that supports that entity. The primary goal of normalization is to minimize duplicate data within the database, which helps prevent performance degradation during inserts and updates. For example, if a customer's address needs to be updated, it's simpler to implement the change if the address is stored in a single location, such as the `Customers` table.

The most common forms of normalization are first, second, and third normal forms.

First normal form

First normal form has the following specifications:

- Create a separate table for each set of related data
- Eliminate repeating groups in individual tables
- Identify each set of related data with a primary key

In this model, you should avoid using multiple columns in a single table to store similar data. For example, if a product can come in multiple colors, you shouldn't have multiple columns in a single row containing the different color values. The first following table, `ProductColors`, isn't in first normal form because it has repeating values for color. For products with only one color, there's wasted space. Additionally, if a product comes in more than three colors, it becomes impractical to set a maximum number of columns. Instead, we can recreate the table as shown in the second table, `ProductColor`.

First normal form also requires that there's a unique key for the table, which is a column (or columns) whose value uniquely identifies each row. In the second table, neither of the columns is unique on its own, but together, the combination of `ProductID` and `Color` forms a unique key. When multiple columns are needed to create a unique key, it's referred to as a composite key.

- `ProductColors` table:

ProductID	Color1	Color2	Color3
1	Red	Green	Yellow
2	Yellow		
3	Blue	Red	
4	Blue		
5	Red		

- ProductColor table:

ProductID	Color
1	Red
1	Green
1	Yellow
2	Yellow
3	Blue
3	Red
4	Blue
5	Red

The third table, **ProductInfo**, is in first normal form because each row refers to a particular product, there are no repeating groups, and we have the column ProductID to use as a Primary Key.

ProductID	ProductName	Price	ProductionCountry	ShortLocation
1	Widget	15.95	United States	US
2	Foop	41.95	United Kingdom	UK
3	Glombit	49.95	United Kingdom	UK
4	Sorfin	99.99	Republic of the Philippines	RepPhil
5	Stem Bolt	29.95	United States	US

Second normal form

Second normal form has the following specification, in addition to those required by first normal form:

- If the table has a composite key, all attributes must depend on the complete key and not just part of it.

Second normal form is only relevant to tables with composite keys, like in the table `ProductColor`, which is the second table. Consider the case where the `ProductColor` table also includes the product's price. This table has a composite key on `ProductID` and `Color`, because only using both column values can we uniquely identify a row. If a product's price doesn't change with the color, we might see data as shown in this table.

ProductID	Color	Price
1	Red	15.95
1	Green	15.95
1	Yellow	15.95
2	Yellow	41.95
3	Blue	49.95
3	Red	49.95
4	Blue	99.95
5	Red	29.95

This table isn't in second normal form. The price value is dependent on the `ProductID` but not on the `Color`. There are three rows for `ProductID 1`, so the price for that product is repeated three times. The issue with violating second normal form is that if we need to update the price, we must ensure it's updated everywhere. If we update the price in the first row but not in the second or third, we would encounter an *update anomaly*. After the update, we wouldn't be able to determine the actual price for `ProductID 1`. The solution is to move the `Price` column to a table that has `ProductID` as a single column key, because that is the only column that `Price` depends on. For example, we could use Table 3 to store the `Price`.

If the price for a product was different based on its color, the fourth table would be in the second normal form, since the price would depend on both parts of the key: the `ProductID` and the `Color`.

Third normal form

Third normal form is typically the aim for most OLTP databases. Third normal form has the following specification, in addition to those required by second normal form:

- All nonkey columns are nontransitively dependent on the primary key.

A transitive relationship implies that one column in a table is related to other columns through a second column. Dependency means that a column can derive its value from another as a result of this relationship. For example, your age can be determined from your date of birth, making your age dependent on your date of birth. Refer back to the third table, `ProductInfo`. This table is in second normal form but not in third. The `ShortLocation` column is dependent on the `ProductionCountry` column, which isn't the key. Like second normal form, violating third normal

form can lead to update anomalies. We would end up with inconsistent data if we updated the `ShortLocation` in one row but didn't update it in all the rows where that location occurred. To prevent this, we could create a separate table to store country/region names and their shortened forms.

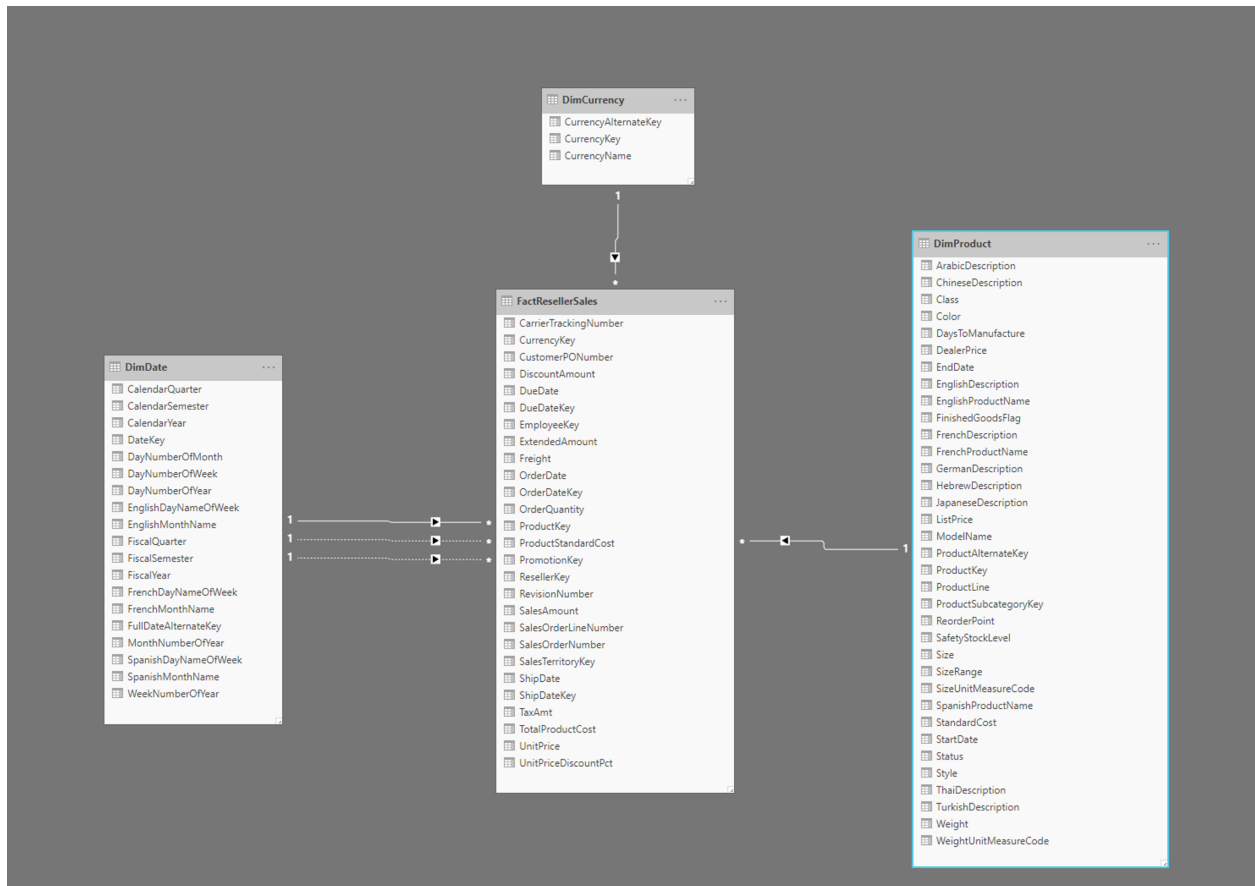
Denormalization

While the third normal form is theoretically desirable, it isn't always possible for all data. In addition, a normalized database doesn't always give you the best performance. Normalized data frequently requires multiple join operations to get all the necessary data returned in a single query. There's a tradeoff between normalizing data when the number of joins required to return query results has high CPU utilization, and denormalized data that has fewer joins and less CPU required, but opens up the possibility of update anomalies.

Denormalized data can be more efficient to query, especially for read heavy workloads like a data warehouse. In those cases, having extra columns may offer better query patterns and/or more simplistic queries.

Star schema

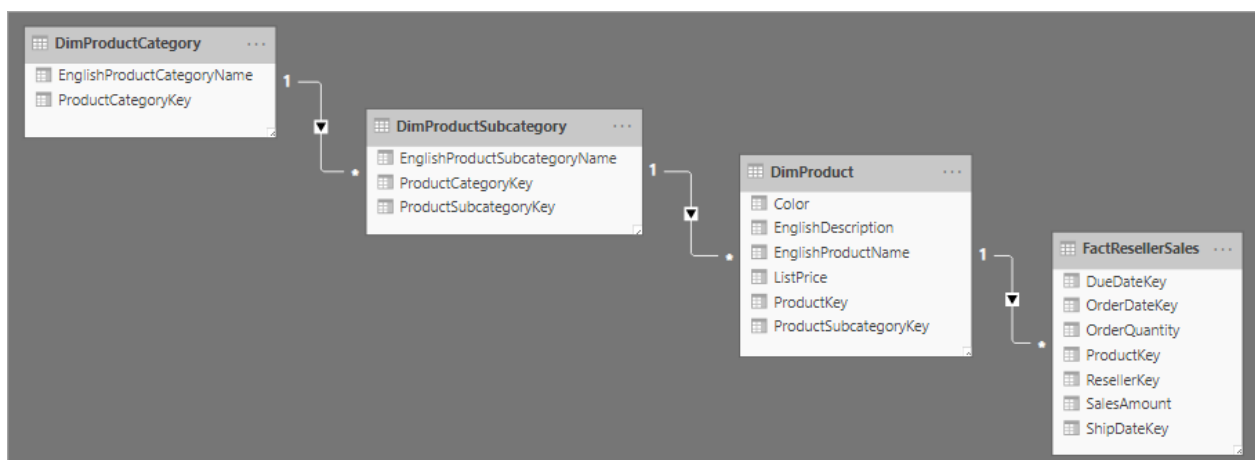
While most normalization is aimed at OLTP workloads, data warehouses have their own modeling structure, which is typically a **denormalized** model. This design uses fact tables to record measurements or metrics for specific events, such as sales, and joins them to dimension tables. Dimension tables are smaller in terms of row count but may have a large number of columns to describe the fact data. Examples of dimensions include inventory, time, and geography. This design pattern makes the database easier to query and offers performance gains for read workloads.



The image illustrates an example of a star schema, featuring a **FactResellerSales** fact table and dimensions for date, currency, and products. The fact table contains data related to sales transactions, while the dimensions only contain data related to specific elements of the sales data. For instance, the **FactResellerSales** table includes only a **ProductKey** to indicate which product was sold. All the details about each product are stored in the **DimProduct** table and are related back to the fact table using the **ProductKey** column.

Related to star schema design is the snowflake schema, which uses a set of more normalized tables for a single business entity. The following image illustrates an example of a single dimension in a snowflake schema. The Products dimension is normalized and stored across three tables:

DimProductCategory, **DimProductSubcategory**, and **DimProduct**.



The main difference between star and snowflake schemas is that the dimensions in a snowflake schema are normalized to reduce redundancy, which saves storage space. The tradeoff is that your queries require more joins, which can increase your complexity and decrease performance.

3. Choose appropriate data types

<https://learn.microsoft.com/en-us/training/modules/explore-performance-based-design/3-choose-appropriate-data-types>

Choose appropriate data types

Completed

SQL Server offers a wide variety of data types, and your choice can significantly impact performance. While SQL Server can automatically convert some data types (known as 'implicit conversion'), this process can be costly and negatively affect query plans. The alternative is explicit conversion, where you use the `CAST` or `CONVERT` function in your code to force a data type conversion.

Also, choosing data types that are larger than necessary can lead to wasted space and require more pages to be read. It's crucial to select the appropriate data types for your data, as this will reduce the total storage required for the database and improve query performance.

Note

In some cases, conversions aren't possible at all. For example, a date can't be converted to a bit. Conversions can negatively impact query performance by causing index scans where seeks would have been possible, and extra CPU overhead from the conversion itself.

The following image indicates in which cases SQL Server can do an implicit conversion and in which cases you must explicitly convert data types in your code.

From \ To	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	date	time	datetimeoffset	datetime2	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml	CLR UDT	hierarchyid
binary		●	●	●	●	●	●	●	■	■	■	■	●	●	✗	✗	●	●	●	●	●	●	●	●	●	✗	✗	●	●	●	●	●
varbinary	●		●	●	●	●	●	●	■	■	■	■	●	●	✗	✗	●	●	●	●	●	●	●	●	●	●	✗	✗	●	●	●	●
char	■	■		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	■	●	●	●	●	●	●	●	●	●
varchar	■	■	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	■	●	●	●	●	●	●	●	●	●
nchar	■	■	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	■	✗	●	●	●	●	●	●	●	●
nvarchar	■	■	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	■	●	✗	●	●	●	●	●	●	●
datetime	■	■	●	●	●	●		●	●	●	●	●	■	■	■	■	■	■	■	■	■	■	■	■	✗	✗	✗	✗	●	✗	✗	✗
smalldatetime	■	■	●	●	●	●	●		●	●	●	●	■	■	■	■	■	■	■	■	■	■	■	■	✗	✗	✗	✗	●	✗	✗	✗
date	■	■	●	●	●	●	●	●		✗	●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	✗	✗
time	■	■	●	●	●	●	●	●	✗		●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	✗	✗
datetimeoffset	■	■	●	●	●	●	●	●	●	●		●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	✗	✗
datetime2	■	■	●	●	●	●	●	●	●	●	●		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	✗	✗
decimal	●	●	●	●	●	●	●	●	✗	✗	✗	✗	◆	◆	●	●	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	✗	✗	✗
numeric	●	●	●	●	●	●	●	●	✗	✗	✗	✗	◆	◆	●	●	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	✗	✗	✗
float	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●		●	●	●	●	●	●	●	●	✗	✗	✗	✗	✗	●	✗	✗	✗
real	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●		●	●	●	●	●	●	●	✗	✗	✗	✗	✗	●	✗	✗	✗
bigint	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●		●	●	●	●	●	●	●	✗	✗	✗	✗	✗	●	✗	✗
int(INT4)	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●	●		●	●	●	●	●	●	✗	✗	✗	✗	✗	●	✗	✗
smallint(INT2)	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●	●	●		●	●	●	●	●	✗	✗	✗	✗	✗	●	✗	✗
tinyint(INT1)	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●	●	●	●		●	●	●	●	✗	✗	✗	✗	✗	●	✗	✗
money	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●	●	●	●	●		●	●	●	✗	✗	✗	✗	✗	●	✗	✗
smallmoney	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●	●	●	●	●	●		●	●	✗	✗	✗	✗	✗	●	✗	✗
bit	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	✗	✗	✗	✗	●	✗	✗	✗
timestamp	●	●	●	●	✗	✗	●	●	✗	✗	✗	✗	●	●	✗	✗	●	●	●	●	●	●	●	●	✗	●	✗	✗	✗	✗	✗	✗
uniqueidentifier	●	●	●	●	●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	●	✗	✗	✗	●	✗	✗	✗
image	●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	●	✗	✗	✗	✗	✗	✗	✗
ntext	✗	✗	●	●	●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	●	✗	●	✗	✗	✗
text	✗	✗	●	●	●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	●	✗	●	✗	✗	✗
sql_variant	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	✗	✗	✗	✗	●	✗	✗	✗
xml	■	■	■	■	■	■	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	○	●	✗	✗
CLR UDT	■	■	■	■	■	■	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	✗	✗
hierarchyid	■	■	■	■	■	■	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

■ Explicit conversion

● Implicit conversion

✗ Conversion not allowed

◆ Requires explicit CAST to prevent the loss of precision or scale that might occur in an implicit conversion.

○ Implicit conversions between xml data types are supported only if the source or target is untyped xml. Otherwise, the conversion must be explicit.

SQL Server provides various system-supplied data types that can be used in your tables and queries. Also, SQL Server allows the creation of user-defined data types using either T-SQL or the .NET framework.

4. Design indexes

Design indexes

Completed

SQL Server offers several index types to support different workloads. At a high level, an index can be thought of as an on-disk structure associated with a table or view, enabling SQL Server to more easily find the row or rows associated with the index key (which consists of one or more columns in the table or view), compared to scanning the entire table.

Clustered indexes

A common DBA job interview question is to ask the candidate the difference between a clustered and nonclustered index, as indexes are fundamental data storage technologies in SQL Server. A clustered index is the underlying table, stored in sorted order based on the key value. There can only be one clustered index on a given table because the rows can be stored in only one order. A table without a clustered index is called a heap, and heaps are typically used only as staging tables. An important performance design principle is to keep your clustered index key as narrow as possible. When considering one or more key columns for your clustered index, you should choose columns that are unique or contain many distinct values. Another property of a good clustered index key is for records that are accessed sequentially and are used frequently to sort the data retrieved from the table. Having the clustered index on the column used for sorting can prevent the cost of sorting every time that query executes because the data will already be stored in the desired order.

Note

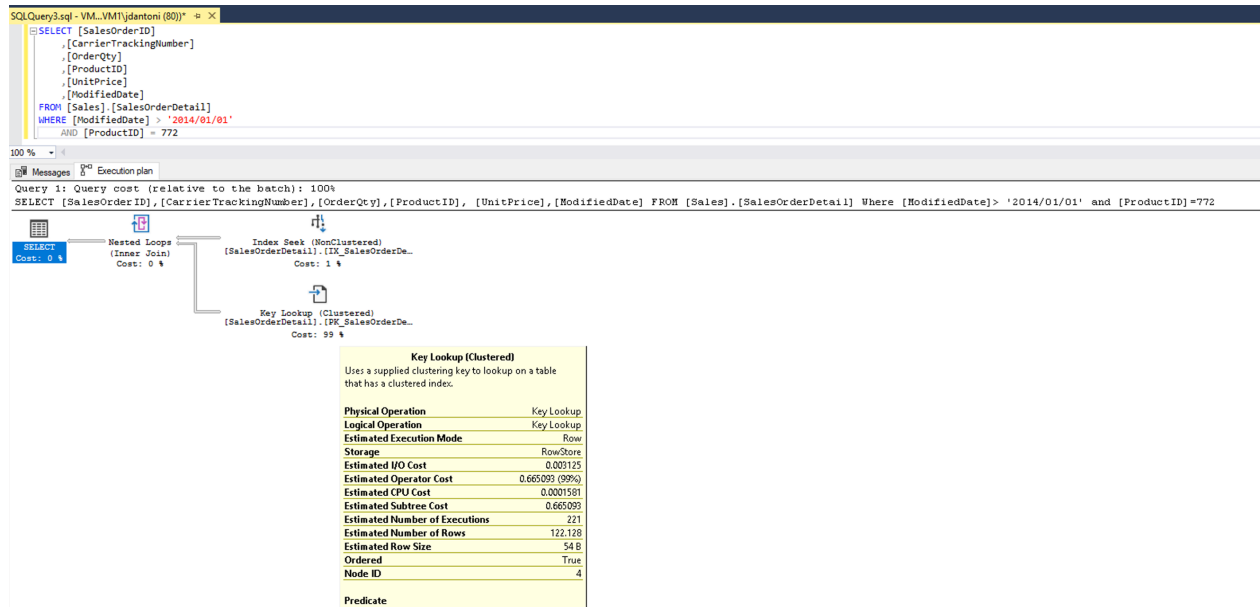
When we say that the table is 'stored' in a particular order, we're referring to the logical order, not the physical, on-disk order. Indexes have pointers between pages, and the pointers help create the logical order. When scanning an index *in order*, SQL Server follows the pointers from page to page. Immediately after creating an index, it's most likely also stored in physical order on the disk, but after you start making modifications to the data and new pages need to be added to the index, the pointers will still give us the correct logical order, but the new pages will most likely not be in physical disk order.

Nonclustered indexes

Nonclustered indexes are separate structures from the data rows. A nonclustered index contains the key values defined for the index and a pointer to the data row that contains the key value. You can add extra nonkey columns to the leaf level of the nonclustered index using the included columns

feature in SQL Server, allowing you to cover more columns. You can create multiple nonclustered indexes on a table.

The following example shows when you need to add an index or add columns to an existing nonclustered index.



The query plan indicates that for each row retrieved using the index seek, more data need to be retrieved from the clustered index (the table itself). There's a nonclustered index, but it only includes the product column. If you add the other columns in the query to a nonclustered index, you can see the execution plan change to eliminate the key lookup.

SQLQuery3.sql - VM...VM1\jdantoni (80) * 

```

CREATE NONCLUSTERED INDEX [IX_SalesOrderDetail_ProductID] ON [Sales].[SalesOrderDetail] ([ProductID] ASC) INCLUDE (
    [CarrierTrackingNumber]
    , [UnitPrice]
    , [ModifiedDate]
    , [OrderQty]
)
with (DROP_EXISTING=ON)

SELECT [SalesOrderID]
    , [CarrierTrackingNumber]
    , [OrderQty]
    , [ProductID]
    , [UnitPrice]
    , [ModifiedDate]
FROM [Sales].[SalesOrderDetail]
WHERE [ModifiedDate] > '2014/01/01'
AND [ProductID] = 772

```

100 %  Messages  Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT [SalesOrderID] , [CarrierTrackingNumber] , [OrderQty] , [ProductID] , [UnitPrice] , [ModifiedDate] FROM [Sales].[SalesOrderDetail] WHERE [ModifiedDate]

 Index Seek (NonClustered)
Cost: 0 %

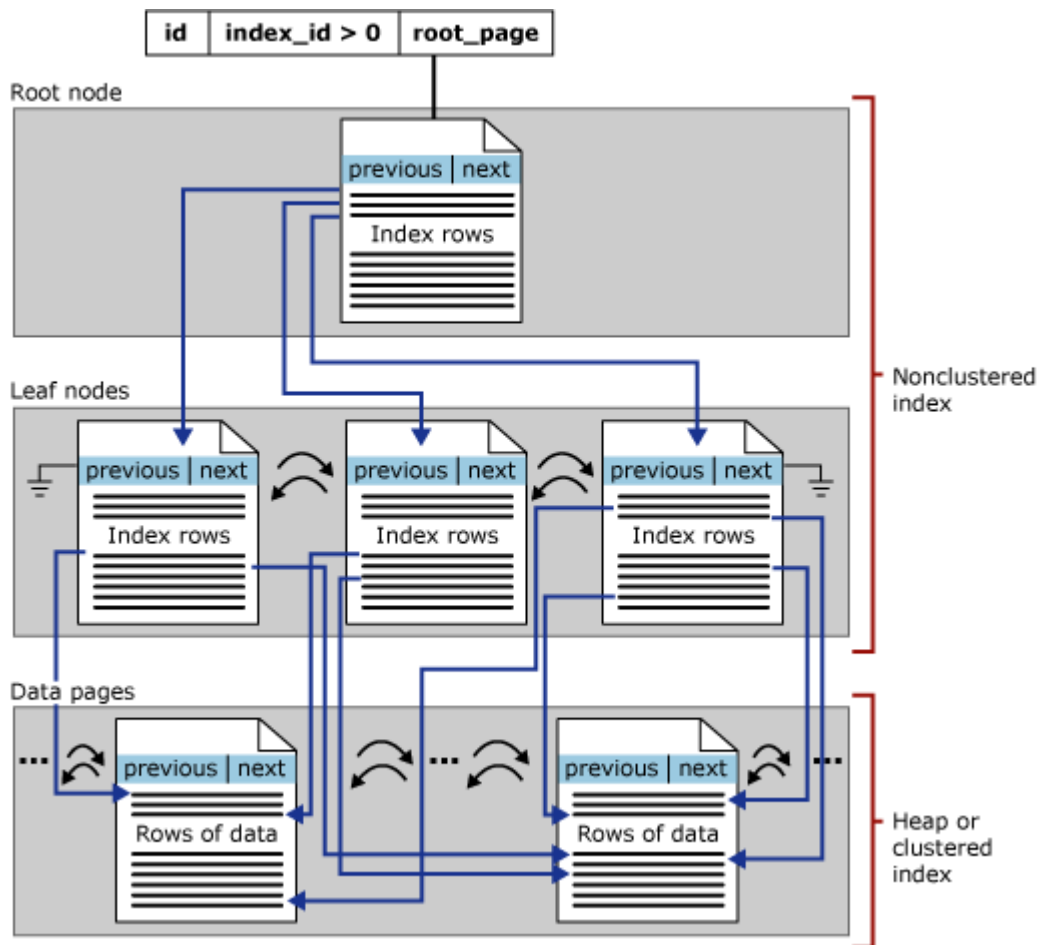
(SalesOrderDetail].[IX_SalesOrderDe...
Cost: 100 %

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0038018 (100%)
Estimated I/O Cost	0.0034017
Estimated Subtree Cost	0.0038018
Estimated CPU Cost	0.0004001
Estimated Number of Executions	1
Estimated Number of Rows	122,128
Estimated Number of Rows to be Read	221
Estimated Row Size	62 B
Ordered	True
Node ID	0
Predicate	
[AdventureWorks2017].[Sales].[SalesOrderDetail].	
[ModifiedDate]>CONVERT_IMPLICIT(datetime,[@1],0)	
Object	
[AdventureWorks2017].[Sales].[SalesOrderDetail].	
[IX_SalesOrderDetail_ProductID]	
Output List	
[AdventureWorks2017].[Sales].[SalesOrderDetail].SalesOrderID,	
[AdventureWorks2017].[Sales].	
[SalesOrderDetail].CarrierTrackingNumber,	
[AdventureWorks2017].[Sales].[SalesOrderDetail].OrderQty,	
[AdventureWorks2017].[Sales].[SalesOrderDetail].ProductID,	
[AdventureWorks2017].[Sales].[SalesOrderDetail].UnitPrice,	
[AdventureWorks2017].[Sales].[SalesOrderDetail].ModifiedDate	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2017].[Sales].	
[SalesOrderDetail].ProductID = Scalar Operator	
(CONVERT_IMPLICIT(int,[@2],0))	

The index created above is an example of a covering index. In addition to the key column, you're including extra columns to cover the query and eliminate the need to access the table itself.

Both nonclustered and clustered indexes can be defined as unique, meaning there can be no duplication of the key values. Unique indexes are automatically created when you create a PRIMARY KEY or UNIQUE constraint on a table.

This section focuses on b-tree indexes in SQL Server, also known as row store indexes. The following image represents the general structure of a b-tree:



Each page in an index b-tree is called an index node, and the top node of the b-tree is called the root node. The bottom nodes in an index are called leaf nodes, and the collection of leaf nodes is the leaf level.

Index design is a blend of art and science. A narrow index with few columns in its key requires less time to update and has lower maintenance overhead; however, it may not be useful for as many queries as a wider index that includes more columns. You may need to experiment with several indexing approaches based on the columns selected by your application's queries. The query optimizer will generally choose what it considers to be the best existing index for a query; however, that doesn't mean there isn't a better index that could be built.

Properly indexing a database can be a complex task. When planning your indexes for a table, you should keep a few basic principles in mind:

- Understand the system's workloads. Tables primarily used for insert operations benefit less from extra indexes compared to tables used for data warehouse operations with high read activity.
- Optimize indexes around the most frequently run queries.
- Choose appropriate data types for columns in your queries. Indexes work best with integer data types, unique, or non-null columns.
- Create nonclustered indexes on columns frequently used in predicates and join clauses, keeping them as narrow as possible to minimize overhead.

- Consider data size/volume. Table scans on small tables are relatively cheap, while scans on large tables are costly.

Another option provided by SQL Server is the creation of filtered indexes. Filtered indexes are ideal for columns in large tables where a significant percentage of rows share the same value in that column. The following example is an employee table that stores records of all employees, including those who have left or retired.

```
CREATE TABLE [HumanResources].[Employee](
    [BusinessEntityID] [int] NOT NULL,
    [NationalIDNumber] [nvarchar](15) NOT NULL,
    [LoginID] [nvarchar](256) NOT NULL,
    [OrganizationNode] [hierarchyid] NULL,
    [OrganizationLevel] AS ([OrganizationNode].[GetLevel]()),
    [JobTitle] [nvarchar](50) NOT NULL,
    [BirthDate] [date] NOT NULL,
    [MaritalStatus] [nchar](1) NOT NULL,
    [Gender] [nchar](1) NOT NULL,
    [HireDate] [date] NOT NULL,
    [SalariedFlag] [bit] NOT NULL,
    [VacationHours] [smallint] NOT NULL,
    [SickLeaveHours] [smallint] NOT NULL,
    [CurrentFlag] [bit] NOT NULL,
    [rowguid] [uniqueidentifier] ROWGUIDCOL NOT NULL,
    [ModifiedDate] [datetime] NOT NULL)
```

In this table, there's a column called `CurrentFlag`, which indicates if an employee is currently employed. This example uses the `bit` datatype, representing two values: one for currently employed and zero for not currently employed. Creating a filtered index with `WHERE CurrentFlag = 1` on the `CurrentFlag` column allows for efficient queries of current employees.

Also, you can create indexes on views, which can provide significant performance gains when views contain query elements like aggregations and/or table joins.

Columnstore indexes

Columnstore indexes offer enhanced performance for queries involving large aggregation workloads. Initially targeted at data warehouses, columnstore indexes have since been adopted for various other workloads to address query performance issues on large tables. Like b-tree indexes, a clustered columnstore index represents the table itself stored in a special way, while nonclustered columnstore indexes are stored independently of the table. Clustered columnstore indexes inherently include all columns in a table but aren't sorted.

Nonclustered columnstore indexes are typically used in two scenarios. The first is when a column's data type isn't supported in a columnstore index (for example, XML, CLR, `sql_variant`, `ntext`, `text`, and `image`). Since a clustered columnstore index always contains all columns of the table, a nonclustered index is the only option. The second scenario involves filtered indexes, used in hybrid transactional analytic processing (HTAP) architectures, where data is loaded into the table while reports are

simultaneously run. Filtering the index (typically on a date field) allows for efficient insert and reporting performance.

Columnstore indexes store each column independently, offering two benefits: reduced IO by scanning only necessary columns and greater compression due to similar data within columns. They perform best on analytic queries scanning large data sets, such as fact tables in data warehouses. You can augment a columnstore index with a b-tree nonclustered index for singleton value lookups.

These indexes also benefit from batch execution mode, processing sets of rows (typically around 900) at a time instead of one by one. This approach reduces CPU instructions significantly.

```
SELECT SUM(Sales) FROM SalesAmount;
```

Batch mode can provide performance increase over traditional row processing. While batch mode for rowstore doesn't have the same level of read performance as a columnstore index, analytical queries may see up to a 5x performance improvement.

Another advantage of columnstore indexes for data warehouse workloads is the optimized load path for bulk insert operations of 102,400 rows or more. While 102,400 is the minimum value to load directly into the columnstore, each collection of rows, called a rowgroup, can be up to approximately 1,024,000 rows. Having fewer, but fuller, rowgroups makes your `SELECT` queries more efficient because fewer rowgroups need to be scanned to retrieve the requested records. These loads occur in memory and are directly loaded into the index. For smaller volumes, data is written to a b-tree structure called a delta store and asynchronously loaded into the index.

```
SQLQuery3.sql - VM...VM1\jdantoni (52))*  X
SET STATISTICS TIME ON
INSERT INTO FactResellerSales_CCI_Demo
SELECT TOP 1024000 *
FROM FactResellerSalesXL_CCI
INSERT INTO FactResellerSales_Page_Demo
SELECT TOP 1024000 *
FROM FactResellerSalesXL_CCI

100 %
Messages
SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 6 ms.

SQL Server Execution Times:
    CPU time = 7609 ms,  elapsed time = 7902 ms.

(1024000 rows affected)

SQL Server Execution Times:
    CPU time = 17031 ms,  elapsed time = 19685 ms.

(1024000 rows affected)

Completion time: 2020-04-22T14:02:07.5526154+00:00
```

In this example, the same data is being loaded into two tables, *FactResellerSales_CCI_Demo* and *FactResellerSales_Page_Demo*. The *FactResellerSales_CCI_Demo* has a clustered columnstore index, and the *FactResellerSales_Page_Demo* has a clustered b-tree index with two columns and is page compressed. As you can see each table is loading 1,024,000 rows from the *FactResellerSalesXL_CCI* table. When `SET STATISTICS TIME` is `ON`, SQL Server keeps track of the elapsed time of the query execution. Loading the data into the columnstore table took roughly 8 seconds, where loading into the page compressed table took nearly 20 seconds. In this example, all the rows going into the columnstore index are loaded into a single rowgroup.

If you load less than 102,400 rows of data into a columnstore index in a single operation, it's loaded in a b-tree structure known as a delta store. The database engine moves this data into the columnstore index using an asynchronous process called the tuple mover. Having open delta stores

can affect the performance of your queries, because reading those records is less efficient than reading from the columnstore. You can also reorganize the index with the `COMPRESS_ALL_ROW_GROUPS` option in order to force the delta stores to be added and compressed into the columnstore indexes.

5. Exercise: Identify database design issues

<https://learn.microsoft.com/en-us/training/modules/explore-performance-based-design/5-exercise-identify-issue-s-database-design>

Exercise: Identify database design issues

Completed

In this exercise, you'll learn how to identify database design issues using SSMS.

Note

To complete this exercise, you'll need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

6. Module assessment

<https://learn.microsoft.com/en-us/training/modules/explore-performance-based-design/6-knowledge-check>

Module assessment

Completed

7. Summary

Summary

Completed

In this module, you learned about database normalization and its three main forms: first, second, and third normal forms. You've also explored the concept of denormalization and its application in data warehouses through models like star schema and snowflake schema. The module also covered SQL Server's range of data types and the impact of their selection on performance. You learned about implicit and explicit conversion of data types and the use of `CAST` or `CONVERT` functions to enforce data type conversion. Lastly, you've been introduced to SQL Server's system-supplied data types and the creation of user-defined data types using T-SQL or the .NET framework.

The main takeaways from this module include understanding the importance of database normalization and denormalization, and how they can influence database performance. You learned how to optimize storage and enhance query performance by choosing appropriate data types and using explicit conversion when necessary. You also gained knowledge about SQL Server's index types, including clustered and nonclustered indexes, and their role in optimizing different workloads. Furthermore, you learned about the considerations for index design, such as understanding system workloads, optimizing indexes around frequently run queries, and choosing appropriate data types for columns.

Additional reading

- [Educational SQL resources](#)
- [SQL Server Data Types](#)
- [Index Design Basics](#)