Performanslı T-SQL Yazımı ve Indeks Kullanımı Üzerine Gerçek Hayat Tecrübeleri

Serap PARLAK[1], Dr. Deniz KILINÇ[1]

[1] Univera Bilgisayar Sistemleri Sanayi ve Ticaret A.Ş, İzmir

[2] Univera Bilgisayar Sistemleri Sanayi ve Ticaret A.Ş, İzmir

serap.parlak@univera.com.tr, deniz.kilinc@univera.com.tr

Özet: Performans kelimesi tek başına şirin görünse de çok göreceli ve değişken bir kavram olup, olumsuz sonuçları gece kâbusumuz haline gelebilir. Basit bir tanım yapacak olsak; "Herhangi bir işlemin beklenenden en az 1 birim fazla doğrulukta, kalitede ve hızda geçekleşmesi, o işin performanslı tamamlandığı anlamına gelir" diyebiliriz. Bu bildiride SQL Server' da T-SQL yazarken ve Indeks kullanırken performansı arttırmak (ya da düşürmemek) için nelere dikkat edilmesi gerektiği yaşanmış profesyonel tecrübelere dayanılarak anlatılmıştır.

Anahtar Sözcükler: Performans, Veritabanı, SQL Server, T-SQL, Indeks, Query Optimizer

Abstract: Although, the word "performance", may seem pretty much relative alone, it is a variable concept and negative results may become our nightmares. Suppose that we make a simple definition: "any process completed at least 1 unit more than expected accuracy, quality and speed", we can say, it is a good performance. In this paper, what should be considered is explained when writing T-SQL and using the index to increase the performance on the basis of professional experience.

1. Giriş

Performans kelimesi tek başına şirin görünse de çok göreceli ve değişken bir kavram olup, olumsuz sonuçları gece kâbusumuz haline gelebilir. Basit bir tanım yapacak olsak; "Herhangi bir işlemin beklenenden en az 1 birim fazla doğrulukta, kalitede ve hızda geçekleşmesi, o işin performanslı tamamlandığı anlamına gelir" diyebiliriz.

Bu konuda her türlü bilimsel metriğe sahip olsak da; olaya, yüke, işleme, kişiye ve beklentiye göre değişen sonuçlar hep karşımıza çıkmış ve fenomen haline gelmiş şu cümleyi söylemiş ya da işitmişizdir "*Duruma göre değişir – It depends*".

Konu göreceli ve kapsamlı olduğu için çalışma alanı daraltıp, bu bildiride SQL Server' da [1] T-SQL yazarken ve Indeks kullanırken performansı arttırmak (ya da düşürmemek) için nelere dikkat edilmesi gerektiği anlatılacaktır.

Transact-SQL [2], SQL Server ve istemci (client) arasında iletişimi sağlayan SQL sorgulama dilinin gelişmiş bir versiyonudur. *Transact Structured Query Language* kelimelerinin kısaltmasıdır. T-SQL kullanarak veri tabanına kayıt eklenebilir, silinebilir, güncellenebilir ya da sorgulama ve raporlama yapılabilir. T-SQL yazarken dikkat edilmesi gereken birçok husus vardır.

Indeksler ise mantık olarak kitapların arka sayfalarında bulunan dizinlere benzerler.

Aradığımız herhangi bir bilgiye normalden çok daha hızlı erişmemizi sağlayan nesnelerdir. Veritabanı dünyasında bir tablo veya view için indeks yaratmak mümkündür. Hem indeks yaratırken hem de sonraki bakım sürecinde uymamız gereken kuralları bilmezsek, performans artışı yerine tutarsız sistem yavaşlamalarıyla ve veritabanı boyutunun aşırı büyümesiyle karşılaşabiliriz.

2. T-SQL Yazarken Dikkat Edilmesi Gerekenler

2.1. T-SQL Sonucunda İhtiyaçtan Fazla Veri Döndürüyor mu?

Bu sorgu sonucunda ne kadar az sonuç kümesi döndürülürse, SQL Server bu veriyi işlemek için o kadar az kaynağa ihtiyaç duyar. Sonuç kümesinde gereksiz veri döndürmek performansı azaltır. Kodlama yapanlar tarafından en çok yapılan hatalar şöyledir:

- WHERE deyimi içermeyen sorgular.
- WHERE deyimi kullanılması gerekliliğine ilave olarak, WHERE deyimi mümkün olduğu kadar seçici olmalıdır. Örneğin belirli tarihteki kayıtlar isteniyorsa, ay veya yıl için tüm kayıtlar döndürülmemeli. Sadece ihtiyaç olan satırları döndürecek WHERE deyimleri tasarlanmalıdır.
- SELECT deyimi, tüm kolonları değil sadece gerekli kolonları içermelidir (Projection). "SELECT *" şeklinde kullanım yanlıştır.
- View'lardan SELECT yapmak yerine, ihtiyaç duyulan verinin direk tablolardan alınması önerilir. Çünkü çoğu view, SELECT deyiminde istenenden yani ihtiyaçtan fazla veri içerir.

Farkında olmadan döndürülen gereksiz veriler, Query Optimizer' ın indeks taraması yapması yerine tablo taraması yapmasına neden olabilir. Bu da veriyi okumak için fazladan IO gerektirir, SQL Server' ın diğer amaçlar için kullanabileceği buffer cash israf edilir, gereksiz network trafiği meydana gelir.

2.2. İhtiyaç Olmadığı Halde Cursor Kullanılıyor mu?

Cursor kullanımları hem SQL Server hem de uygulama performansını her anlamda yavaşlatır. Cursor'lere bulaşmamak, varsa kurtulmak en akıllıca yoldur. Satır satır işlem yapılması gerekliyse, cursor yerine şu seçeneklerin kullanılması düşünülmelidir;

- Geçici (temp) tablo kullanmak
- Türetilmiş tablolar kullanmak
- İlişkili alt sorgular kullanmak
- CASE deyimleri kullanmak
- Çoklu sorgular kullanmak

Bu maddelerin hepsi cursor yerine geçebilecek seçenekler olup, doğru kullanıldıklarında daha performanslı çalışacaklardır.

2.3. UNION / UNION ALL Kullanımları Uygun mu?

Çoğu kişi UNION ve UNION SELECT çalışma şeklini tam olarak anlamamıştır; bu yüzden SQL Server kaynakları boş yere harcanır. UNION kullanıldığı zaman oluşan sonuç kümesi üzerinde SELECT DISTINCT benzeri işlem gerçekleşir. Diğer bir deyişle UNION, iki benzer sonuç kümesini birleştirerek eş (duplicate) kayıtları tarayıp eleyecektir. Eğer amaç buysa, UNION kullanımı doğru sözdizimidir.

Fakat, eş kayıt içermeyecek iki sonuç kümesi birleştirilmek isteniyorsa UNION kullanımı gereksizdir, çünkü olmayacağı bilinen bir eş kayıt tarama işlemi fazladan sistemi yoracaktır. Sonuç olarak, eş kayıt olmayacağı biliniyorsa UNION yerine UNION ALL kullanılması gerekmektedir. UNION ALL iki sonuç kümesini birleştirir ancak eş kayıtları taramaz ve elemez.

2.3. DISTINCT Uygun Olarak Kullanılmış mı?

Bazı programcılar SELECT cümlelerine, gerek olsa da olmasa da otomatik olarak DISTINCT deyimini eklerler. Performans açısından DISTINCT deyimi, yalnızca sonuç kümesinde elenmesi gereken eş (duplicate) kayıtlar varsa kullanılmalıdır. Çünkü DISTINCT deyimi kullanımı sonuç kümesi üzerinde önce sıralama sonra da eş kayıtları eleme işlemi gerçekleştirir ve pahalı SQL Server kaynakları kullanır.

2.4. GROUP BY Deyimi İyileştirilebilir mi?

GROUP BY deyimi toplam fonksiyonu ile ve toplam fonksiyonsuz kullanılabilir. Eğer optimum performans isteniyorsa toplam fonksiyonsuz GROUP BY deyimi kullanılmamalıdır. Bu nedenle aynı sonucu elde edebilen DISTINCT seçeneği kullanılabilir ve daha hızlı olur.

Örneğin:

```
SELECT OrderID FROM Order_Details
WHERE UnitPrice > 10 GROUP BY OrderID
```

Yerine DISTINCT ile yeniden yazılabilir:

```
SELECT DISTINCT OrderID FROM
Order_Details WHERE UnitPrice > 10
```

Yukarıdaki iki sorgu aynı sonucu üretir ancak DISTINCT içeren daha az kaynak tüketir ve daha performanslı çalışır. Aşağıdaki adımlar takip edilerek GROUP BY deyimi hızlandırılabilir:

- Sorgudan dönen satır sayısı mümkün olduğunca küçük olmalıdır.
- Grup sayısı mümkün olduğunca az olmalıdır.
- Gereksiz kolonlar gruplanmamalıdır.
- GROUP BY içeren SELECT deyiminde JOIN varsa, JOIN kullanmak yerine alt sorgu kullanarak sorgu yeniden yazılmaya çalışılmalıdır. Eğer JOIN kullanılması gerekliyse, GROUP BY kolonunun kullanılan diğer kolonlarla aynı tablodan olmasına özen gösterilmelidir.
- SELECT deyimine GROUP BY ile aynı kolonları içeren ORDER BY deyimi ile de eklenmesi önerilir. Bu eklenti GROUP BY deyiminin daha hızlı çalışmasını sağlar.

2.5. Gereksiz Sıralama İşlemi Yapılıyor mu?

SQL Server sıralama işlemi gerçekleştirirken ekstra kaynak kullanır. Sıralama işlemi aşağıdaki T-SQL deyimlerinden biri ile gerçekleşir:

- ORDER BY
- GROUP BY
- SELECT DISTINCT
- UNION
- CREATE INDEX (Sık çalışmadığı için çok kritik değildir)

Diğer yandan sıralama yükünü azaltmak için dikkat edilebilecek bazı hususlar var:

- Sıralanacak satır sayısı en aza indirgenmelidir.
- Sıralanacak kolon sayısı en aza indirgenmelidir.
- Sıralanacak kolonların genişliği (fiziksel boyutu) en aza indirgenmelidir.
- Karakter veri tipleri yerine sayı veri tiplerine sahip kolonlar sıralanmalıdır.
- Sıkça sorgulanan belirli kolonlar varsa, bu kolonlara *clustered indeks* yaratılması önerilir.

2.6. Geçici (Temp) Tablolar Gerek Olmadığı Halde Kullanılıyor mu?

Geçici tablolar pratik bir kullanım sağlarken duruma göre fazla yük de getirebilirler. Yükten kurtarmak ve performansı iyileştirmek için geçici tablo kullanımını kaldıran farklı yöntemler kullanılabilir:

- İhtiyacı giderecek sorguyu standart sorgular veya stored procedure kullanarak yeniden yazmak
- Tablo veri tiplerini kullanmak
- İlişkili alt sorgular kullanmak
- Reel tablolar kullanmak.
- Geçici tabloya benzeyen UNION deyimleri kullanmak.

2.7. Stored Procedure Kullanılmış mı?

Stored procedure (SP)' ler yazılımcılara ve sisteme birçok fayda sağlarlar. Network trafiğini ve bekleme süresini azaltıp uygulama performansını arttırırlar. Örneğin network üzerinde 500 satır TSQL göndermek yerine, daha hızlı çalışacak ve daha az kaynak kullanacak bir SP çalıştırma isteği göndermek yeterlidir. Ayrıca, SQL Server ara bellekte tutulan SP çalışma planları yeniden kullanılabilir.

İstemci uygulama taleplerinde SP kullanımı daha verimlidir. Örneğin, uygulamanın image tipinde veri kolonuna, SP kullanmadan büyük bir binary değer INSERT edilmesi gerekiyorsa, bu binary değeri karakter stringe dönüştürüp SQL Server' a göndermesi gerekir. SQL Server bu değeri aldığında yeniden binary değere dönüştürerek sistem kaynaklarını verimsiz kullanmış olur. SP kullanımı bu sorunu elimine eder ve uygulamalardan parametre olarak binary değer göndermek mümkündür.

SP' ler T-SQL kodunun yeniden kullanımına yardımcı olurlar. Bu durum direk uygulama performansını arttırmaz, kod miktarını ve debug süresini azaltarak yazılımcıların verimliliğini arttırır.

2.8. T-SQL Dinamik Sorgu Çalıştırma Komutu Kullanılıyor mu?

Eğer dinamik bir T-SQL sorgu çalıştırılması gerekiyorsa, EXECUTE komutu yerine sp_executesql prosedürü kullanılmalıdır [6]. Sp_executesql komutu iki temel avantaja sahiptir. Birincisi, kod yazarken daha fazla seçenek yaratan parametre kullanım desteği olmasıdır. İkincisi, SQL Server tarafından yeniden kullanılmak üzere çalışma planı (execution plan) yaratıyor olmasıdır. Bu da performans arttırmaya yardımcı olur.

2.9. Referential Integrity Kullanımları

Veritabanında gereksiz referential integrity olmamalıdır. Örneğin, referential integrity uygulamak için primary key ve foreign key kullanılıyorsa, aynı amacı gerçekleştirecek gereksiz trigger eklenmemelidir. Yine aynı işi yapacak constraints,/defaults veya constraints /rules kullanımları gereksiz yük getirecektir.

2.10 WHERE Deyimi Sargable mı veya İyileştirilebilir mi?

Sargable Deyimi, WHERE deyiminde sabit bir değerin bir kolonla karşılaştırılmasını işaret eden, "Search ARGument" kelimelerinden oluşan SARG kısaltmasından gelir. Eğer bir WHERE deyimi sargable ise sorgunun hızlı tamamlanmasını sağlayacak indeks avantajı var demektir [3]. Eğer WHERE (veya bir parçası) non-sargable bir deyim ise indeks avantajından yararlanılmaz ve table scan (tüm tabloyu tarama) işlemi gerçekleşir, bu da sorgu performansının düşmesine neden olur.

WHERE deyimlerindeki "IS NULL", "<>", "!=", "!>", "!<", "NOT", "NOT EXISTS", "NOT IN", "NOT LIKE", ve "LIKE '%500'" gibi non-sargable arama argümanları genellikle (her zaman değil), arama gerçekleştirecek indeks kullanımına engel olurlar. Ek olarak, üzerinde bir *fonksiyon* bulunduran kolonların kullanıldığı deyimler, operatörün iki tarafında da aynı deyimi bulunduran deyimler, kolona dayalı karşılaştırmalar içeren deyimler non-sargable deyimlerdir.

Non-sargable deyimler içeren WHERE cümleleri her zaman tablo tarama

yapmayabilirler. Eğer WHERE cümlesi hem sargable hem non-sargable deyimler içeriyorsa, en azından sargable deyim için indeks kullanılır ve verinin hızlıca alınmasına yardımcı olur.

Çoğu koşulda, tablo üzerinde sorgunun SELECT, JOIN, WHERE deyimlerindeki tüm kolonları içeren covering indeks varsa, WHERE deyimi non-sargable olsa bile tablo tarama yerine covering indeks taraması kullanabilir. Unutulmamalıdır ki covering indeksin de, okuma yaparken disk IO arttırma gibi indeks üretim tarafında dezavantajları vardır. Eğer WHERE deyiminin sargable olup olmadığı bilinmiyorsa, sorgu çalışma planı ile indeks kullanıp kullanmadığı kontrol edilebilir.

Sargable operatörler	Non-sargable operatörler
=	IS NULL
>	<>
<	!=
>=	!>
<=	!<
EXIST	NOT
IS	NOT EXIST
IN	NOT LIKE
BETWEEN	LIKE '%abc'
LIKE 'abc%' (least best sargable operator)	LIKE '%ABC%'
	LIKE '%abc%'
	a function on a column
	column1 = column1
	column1 = column2

Tablo1. Sargable ve non-sargable operatörler

Eğer NOT IN içeren bir sorgu varsa performans düşüşüne neden olur çünkü Query Optimizer bu aktiviteyi yerine getirmek için tüm tabloyu tarar. NOT IN yerine şu seçeneklerden biri kullanılabilir:

- EXISTS veya NOT EXISTS kullanmak
- IN kullanmak
- LEFT OUTER JOIN gerçekleştirmek ve NULL durumunu kontrol etmek

Eğer T-SQL deyiminde IN ve EXISTS arasında seçim yapma şansı varsa EXISTS kullanılmalıdır. Çünkü EXISTS daha randımanlı ve hızlı gerçekleşir.

Eğer WHERE deyiminde IN veya OR kullandığımız zaman SQL Server indeks taraması yerine tablo tarama kullanıyorsa, indeks hint kullanarak Query Optimizer indeks kullanılacak duruma getirilebilir.

```
SELECT * FROM tblTaskProcesses WHERE
nextprocess = 1 AND processid IN(8,
32, 45)
```

Yerine

```
SELECT * FROM tblTaskProcesses (INDEX
=IX_ProcessID)WHERE nextprocess = 1
AND processid IN(8, 32, 45)
```

WHERE deyiminde IN veya BETWEEN kullanımı arasında seçme şansı varsa, BETWEEN kullanılması performans artışı sağlayacaktır.

Mümkünse T-SQL kodunda string concat (birleştirme) işlemlerinden kaçınılmalıdır çünkü çok hızlı işlemler değillerdir ve uygulama performansının genelini düşürürler.

Sorgunun WHERE deyimi OR operatörü içeriyorsa ve OR operatörünün referans aldığı kolonlardan biri indeksli değilse (veya kullanışlı bir indeksi yoksa) Query Optimizer bir tabloyu tamamen tarayabilir veya clustered indeks üzerinden tarama gerçekleştirebilir. Bu yüzden eğer birçok sorguda OR ifadesi kullanılıyorsa, referans gösterilen kolonların yararlı indeksleri olup olmadığından emin olunmalıdır.

Bir sorgu bir veya daha fazla OR deyimi içeriyorsa, performansı arttırmak için bazı koşullarda UNION ALL deyimleri birleşiminden oluşan seri ile yeniden yazılabilir.

Örneğin:

```
SELECT employeeID, firstname, lastname
FROM names WHERE dept = 'prod' OR city
= 'Orlando' OR division = 'food'
```

Üç koşul ile ayrılan bu sorgunun indeks kullanabilmesi için OR yerine UNION ALL ile yeniden yazılması gereklidir:

```
SELECT employeeID, firstname, lastname
FROM names WHERE dept = 'prod'
UNION ALL
SELECT employeeID, firstname, lastname
FROM names WHERE city = 'Orlando'
UNION ALL
SELECT employeeID, firstname, lastname
FROM names WHERE division = 'food'
```

Eğer uygulamada çok sayıda, CHAR ve VARCHAR kolon üzerinde genel arama karakterli (LIKE %) tekst arama yapılıyorsa, *SQL Server Full Text Search* seçeneğini kullanılması önerilir. Search Service, veritabanındaki aramayı önemli derecede hızlandırabilir.

3. SQL Server Indeks Kullanımı

İndeks kullanım denetimi kolay bir iş değildir ancak SQL Server performansı için çok kritiktir. Bir indeksle uğraşmak kolaydır ama birçok veritabanındaki yüzlerce indeksle uğraşmak zordur ve uzmanlık ister. Indeks denetimlerinin düzgün yapılması performans artırımı için çok önemlidir. İndeks denetimine başlamak için iki yaklaşım vardır.

Birinci yaklaşım, SQL Server performansına genel olarak etki edebilecek indekslere odaklanarak yani işi küçülterek daha yönetilebilir hale getirmektir. Örneğin çok tablo içeren yoğun bir veritabanında en fazla veri içeren tablolardan başlanıp daha az veri içerenlere doğru devam edilebilir. Bu yolla en çok güç sarf edilen alanlara odaklanılır ve bu da performansa pozitif etki eder.

Mecburen daha çok tercih edilen *ikinci yaklaşım da*, hatalardan yola çıkma yaklaşımıdır. Bu yöntemle, veritabanındaki tüm indekslerle değil, performans problemi görülen yerlere odaklanılabilir. Eğer veritabanındaki performans problemleri kesin olarak belirlenirse, iyileştirilebilecek indeksler belirlenir ve bunlarla daha fazla ilgilenilebilir. Eğer ilgilenilmesi gerekilen çok sayıda indeks varsa en büyüğünden başlanılabilir. Veritabanı indekslerini denetlemeye karar verildiğinde, bir plan çıkartılmalı ve sistemli bir şekilde uygulanmalıdır.

3.1. Veritabanındaki Tüm Tablolar Clustered Indeks İçeriyor mu?

Veritabanındaki tüm tabloların clustered indeks içermesi gerektiği genel bir kuraldır. Clustered indeks her zaman değil ancak genellikle, monoton artan (identity kolon veya arttırılan bir değerle oluşmuş kolon) bir kolon üzerinde olmalıdır ve benzersiz (unique) olmalıdır. Çoğunlukla Primary Key clustered indeks için ideal kolondur.

Clustered indeksi primary key olan bir tabloya bir satır ekleniyorsa, her INSERT fiziksel olarak disk üzerinde birer birer gerçekleşecektir. Bu nedenle page splitler gerçekleşmez [4].

Eğer bir çok satır bir heap' e (Clustered Indeks olmayan tablo) INSERT ediliyorsa, veri monoton olarak artsa da artmasa da data pagelerin üzerine belirli bir sırayla eklemez [4]. Buna göre SQL Server, diskten veriye ulaşırken özellikle okuma yaparken daha fazla çalışmak zorunda kalır. Eğer clustered indeks eklenmiş olsaydı veri data pagelere sırayla eklendiği için daha az disk IO gerçekleşecekti.

3.2. Tablodaki Kolonlar Birden Fazla Indekste Bulunuyor mu?

Tabloların güncel indekslerini gözden geçirirken kolonlara ait gereksiz eş (duplicate) indeksler olup olmadığına bakılmalıdır. Eş indeksleri silmek sadece diskte fazladan yer tutulmasını engellemez

aynı zamanda veri erişimi ve veri düzenleme işlemlerini (INERT, DELETE, UPDATE) de hızlandırır.

Örneğin Primary Key olduğuna dikkat edilmemiş kolon otomatik indekslenmiş farklı bir indeks adı altında tekrarlanmış olabilir.

3.3. Kullanılmayan Indeksler Var mı?

Kullanılmayan indeksler de eş indeksler gibi gereksiz disk alanı harcayıp, veri erişimi ve veri düzenleme işlemlerinde performansını azaltırlar.

3.4. JOIN Kullanan Tablolarda JOIN Kolonlara Uygun Indeks Var mı?

Tablolarda join olunan kolonların yüksek performansa ulaşması için indekslenmesi önerilir. Optimum JOIN performansı için oluşturulmuş indeksleri denetlemek kolay değildir, tam denetim için veritabanında gerçekleşen tüm JOIN' leri iyi biliyor olmak gerekir.

Programcılar genelde JOIN' lerde kullanılan Primary Key / Foreign Key ilişkilerini yaratırken, Primary Key kolonlara indeks yaratıldığını ancak foreign key kolonlara otomatik yaratılmadığını ve manuel oluşturmak gerektiğini unuturlar.

3.5. Kolondaki Verilerin Benzersizliği Indeks Kullanımı için Yeterli mi?

Tablonun bir veya daha fazla indeks içermesi, SQL Server Query Optimizer' ın indeks kullanacağı anlamına gelmez. Kullanmadan önce Query Optimizer yararlığını göz önünde bulundurur [3]. Eğer kolon en az %95 oranda benzersiz değilse, genellikle Query Optimizer bu kolondaki non-clustered indeksi kullanmaz. Bu nedenle %95 benzersiz olmayan kolonlara non-clustered indeks yaratılmamalıdır. Örneğin "evet" ve "hayır" bilgilerinden oluşan bir kolon %95 benzersiz olamayacağı için bu kolona yaratılmış indeks kullanılmaz ve performans için uğraşlarımız ters etki yapar.

3.6. Indeksler Hangi Sıklıkta Rebuild (yeniden derleme) Ediliyor?

Bir zaman sonra indeksler fragmante olur ve SQL Server bunlara erişmeye çalıştığında zorlanır, performans düşer. Bunun tek çözümü veritabanındaki tüm indeksleri düzenli olarak defragmante etmektir.

Amaç, indekslerin defragmante edilip edilmeyeceğine bakmaksızın hangi sıklıkta bu işlemi gerçekleştirilmesi gerektiğini belirlemek olmalıdır. Modifikasyonların hangi sıklıkta yapıldığına ve veritabanı boyutuna bağlı olarak günlük, haftalık, aylık yapılması gerekip gerekmediğine karar verilmelidir.

Eğer veritabanında günlük olarak çok fazla modifikasyon oluyorsa daha sık gerçekleştirilmelidir. Eğer veritabanı boyutu çok büyükse rebuild işlemi uzun sürecektir. Defragmante işlemi sırasında fazlaca kaynak kullanılacağı ve kullanıcıların etkileneceği de unutulmamalıdır.

3.7. Indekste İdeal Fillfactor Değeri Belirlenmiş mi?

Fillfactor değerinin nasıl belirleneceği uygulamanın SQL Server tablolarına yaptığı okuma yazma oranına bağlıdır. Genel kural şöyledir:

- Az güncellenen tablolar (100-1 okuma yazma oranı): %100 fillfactor
- Çok güncellenen tablolar (Yazma okumadan fazla) : %50- %70
- İkisinin arası : %80- %90

Uygulamanın optimum fillfactor' ünü bulmak için deneyler yapılmalıdır. Az fillfactor' ün çok fillfactor' den daha iyi olduğu doğru değildir [6]. Az fillfactor ile splitler azalırken, SQL Server sorgularıyla okunacak pagelerin sayısı artar, performans düşer. Çok az

fillfactor ile sadece IO artmaz, buffer cache de etkilenir. Data pageler diskten buffera alınırken boş alanlar dahil tüm pageler taşınır. Düşük fillfactor' lü data pageler SQL Server buffera taşınırken aynı anda alınan diğer önemli data pagelere daha az yer kalır.

Eğer fillfactor belirtilmezse varsayılan fillfactor 0 olur %100 ile aynı anlama gelir, indeks leaf page seviyesi %100 dolar..

Yeni indeksler yaratılırken ve var olan indeksler yeniden derlenirken gerekli fillfactor değerleri belirlenmelidir. Tüm koşullar göz önünde bulundurularak varsayılan değeri 0 verilmemelidir. Onun yerine, boş alan ayıran bir fillfactor değeri verilmelidir.

4. Sonuç

T-SQL ile sorgu yazılması basit gibi görünse de özellikle veritabanı boyutunun arttığı durumlarda, yapılan ufacık bir hata veya bilgi eksikliği tüm sistemin yavaşlamasına neden olabilir. Fazladan veri çekip çekmediğimizden sıralamaya, gruplamadan geçici tablo kullanımına, DISTINCT kullanımında sargable sorgular yazmaya kadar dikkat edilmesi gereken bir çok önemli nokta vardır. Yine indeks yaratılması ve yönetilmesi kolay bir iş olmayıp, SQL Server performansı için çok kritiktir. Bir indeksle uğraşmak kolaydır ama birçok veritabanındaki yüzlerce indeksle uğraşmak zordur ve uzmanlık ister. Uymamız gereken kuralları bilmezsek, performans artışı yerine tutarsız sistem yavaşlamalarıyla ve veritabanı boyutunun aşırı büyümesiyle karşılaşabiliriz.

Kaynaklar

[1] SQL Server: http://www.microsoft.com/sqlserver/en/us/default.aspx

[2] Transact-SQL Reference (Transact-SQL): http://msdn.microsoft.com/en-us/library/ms189826(v=sql.90).aspx

[3] JOIN Türüne Göre SQL Optimizer Çalışma Mantığı: http://univera-ng.blogspot.com/2009/12/join-turune-gore-sql-optimizer-calsma.html

[4] Heap, Clustered İndeks ve Nonclustered İndeks Data Yapıları - Bölüm 1 (Tablo ve İndeks Organizasyonu, Heap Mimarisi): http://univera-ng.blogspot.com/2010/12/heap-clustered-indeks-ve-nonclustered_28.html

[5] SQL Sorgu Optimizasyonuna Yardımcı Araçlar: http://univera-ng.blogspot.com/2009/12/sql-sorgu-optimizasyonuna-yardmc.html

[6] SQL Server Performance: http://www.sql-server-performance.com/