SQL Tuning El Kitabı

SQL OPTIMIZER NEDİR ?

Herhangi bir SQL sorgusu çalıştırıldığında, istenilen bilgiye nasıl ulaşılacağına "Optimizer" adı verilen veri tabanı optimizasyon birleşeni karar vermektedir. Oracle, kullanıcılarına tahminler üzerine çalışan "Rule-Based Optimizer" ve daha çok akıl yürütme yöntemi ile çalışan "Cost-Based Optimizer" olmak üzere iki adet optimizasyon seçeneği sunmaktadır.

Rule-Based Optimizer

Veri tabanına ulaşılırken, Rule-Based Optimizer (RBO) ile önceden tanımlanmış kurallar seti kullanılarak hangi yolun izleneceğine karar verilir. Burada bahsedilen kurallar "**`SELECT /*+ RULE */. . .`**" şeklinde kullanılmaktadır ve böylece veri tabanında hangi indeksin kullanılacağı gibi ek bilgiler verilmektedir. Eğer bu yöntem kullanılacaksa, RDBMS'de aşağıdaki tanımlamaların yapılması gerekmektedir:

- INIT.ORA ya da SPFILE dosyasında OPTIMIZER_MODE = RULE değişikliği yapılmalıdır.
- ALTER SESSION SET OPTIMIZER_MODE = RULE komutu sistemde çalıştırılmalıdır.

Cost-Based Optimizer

Cost-Based Optimizer'ın (CBO) Rule-Based Optimizer'a göre daha kapsamlı ve karışık bir çalışma prensibi bulunmaktadır. Kullanılacak olan en iyi yöntemi belirlenirken, çeşitli veri tabanı bilgileri (tablo boyutları, kayıt sayıları, verilerin dağılımı vs.) kullanılmaktadır.

Cost-Based Optimizer'ının ihtiyacı olan veriyi sağlamak için veri tabanı objelerinin DBMS_STATS prosedürü kullanılarak analiz edilmeleri ve istatistiklerinin toplatılması gerekmektedir. Eğer bir tablonun analizi yapılmamışsa, Rule-Based Optimizer'ın kuralları kullanılarak yolu belirlenir. Aynı sorguda bazı tablolar analiz edilmiş ve bazıları analiz edilmemiş ise, sistem öncelikli olarak Cost-Based Optimizer'ını kullanır. Eğer bu yöntem kullanılacaksa; RDBMS'de aşağıdaki tanımlamaların yapılması gerekmektedir:

- INIT.ORA/SPFILE dosyasında OPTIMIZER_MODE = CHOOSE değişikliği yapılmalıdır ve sorgudaki tablolardan en az bir tanesinin istatistik bilgilerinin mevcut olması gerekmektedir.
- ALTER SESSION SET OPTIMIZER_MODE = CHOOSE komutu sistemde çalıştırılmalıdır ve sorgudaki tablolardan en az bir tanesinin istatistik bilgilerinin mevcut olması gerekmektedir.
- ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS (veya ALL_ROWS) komutu sistemde çalıştırılmalıdır ve sorgudaki tablolardan en az bir tanesinin istatistik bilgilerinin hesaplanmış olması gerekmektedir.

Explain Plan Tablosu

Oracle'da bir sorgunun çalışmasının sisteme olan maliyet bilgileri, EXPLAIN PLAN sayesinde hesaplanabilmektedir. Kullanılan plan tablosunun COST kolonunda sorgunun sisteme olan yükünün hesaplanmış değeri tutulmaktadır. Kullanılan optimizerin çalışma yolunu değiştirerek (sorguya yardımcı ek kurallar koyarak, indeks ekleyerek, indeks kaldırarak, nesnelerin analizini yaparak vs.) hesaplanan yükteki yükselmeler ve azalmalar gözlemlenir. Böylece sorgunun en uygun maliyeti veren çalıştırma yöntemi seçilir.

FILTER	Korelasyon alt sorgusu gibi eşleşen kayıtları daha kaliteli bir hale getirmek için sorguda uygulanacak kriterdir.
FULL TABLE SCAN	Tablo ilk kayıttan son kayıda kadar taranmakta ve herhangi bir indeks kullanılmamaktadır.
INDEX (UNIQUE)	SQL sorgusu belirli bir değeri aramak için unique (her satır için ayrı tek kayıt) indeks kullanmaktadır.
INDEX (RANGE SCAN)	SQL sorgusunda eşitsizlik ya da BETWEEN kriteri kullanılmaktadır.
HASH JOIN	SQL sorgusundaki tablolar okunur ve hash-key olarak bilinen bir matematiksel hesaplama ile hafızaya alınırlar.
MERGE JOIN	SQL sorgusunda FROM cümleciğinde birden fazla tablo yer aldığı zaman bu birleştirme yöntemi kullanılır. Oracle, iki sonuç tablosunu birleşen sütunlar üzerinde biraraya getirerek sıralayacak ve sonra birleşen sütunlar yardımıyla sonuçları biraraya getirecektir.
NESTED LOOP	Bu işlem, tabloları birleştirmenin bir başka yöntemidir. İç içe kullanılan döngü anlamına gelen yöntemde sistem paralel olarak birleştirilen indeksler üzerinde döngü içinde ilerleyerek sonuca ulaşmaya çalışmaktır.

SQL OPTİMİZASYON ÖNERİLERİ

Birden fazla sorgu kullanılması

☒ *Önerilmez*

```
SELECT name
FROM   products
WHERE  product_id = 1;

SELECT type_name
FROM   product_type
WHERE  product_type_id = 1;
```

☑ *Önerilir*

```
SELECT p.name,
       pt.type_name
FROM   products p,
       product_type pt
WHERE  p.product_type_id = pt.product_type_id
   AND p.product_id = 1;
```

Bir yerine iki query çalıştırmak her zaman daha çok iş gücüdür.

Tablo kolon tanımları kullanımı

☒ *Önerilmez*

```
SELECT p.name,
       pt.type_name,
       description,
       price
FROM   products p,
       product_type pt
WHERE  p.product_type_id = pt. product_type_id
   AND p.product_id = 1;
```

☑ *Önerilir*

```
SELECT p.name,
       pt.type_name,
       p.description,
       p.price
FROM   products p,
       product_type pt
WHERE  p.product_type_id = pt. product_type_id
   AND p.product_id = 1;
```

Referans verilmezse veri tabanı tüm tablolarda bu alanlar için arama yapmakta ve sorgu daha yavaş çalışmaktadır.

HAVING yerine WHERE kullanımı

☒ *Önerilmez*

```
SELECT product_type_id,
       AVG(price)
FROM   products
GROUP BY product_type_id
HAVING product_type_id IN ( 1, 2 );
```

☑ *Önerilir*

```
SELECT product_type_id,
       AVG(price)
FROM   products
WHERE  product_type_id IN ( 1, 2 )
GROUP BY product_type_id;
```

İkincisi işlemin başlangıcında kayıtları sınırlarken, ilkinde tüm kayıtlar için AVG çalıştırılmaktadır.

UNION yerine UNION ALL kullanımı

☒ *Önerilmez*

```
SELECT product_id,
       product_type_id,
       name
FROM   products
UNION
SELECT product_id,
       product_type_id,
       name
FROM   more_products;
```

☑ *Önerilir*

```
SELECT product_id,
       product_type_id,
       name
FROM   products
UNION ALL
SELECT product_id,
       product_type_id,
       name
FROM   more_products;
```

UNION ALL her iki sorgu sonucunda tüm kayıtları getirirken, UNION tekrarlanan kayıtları elemektedir. Bu nedenle, gerçekleşen eleme işleminden dolayı UNION ALL daha hızlıdır ve sistemi yormaz.

IN yerine EXISTS kullanımı

☒ *Önerilmez*

```
SELECT product_id,
       name
FROM   products
WHERE  product_id IN (
     SELECT product_id
     FROM   purchases );
```

☑ *Önerilir*

```
SELECT product_id,
       name
FROM   products pr
WHERE EXISTS (
   SELECT 1
   FROM   purchases pu
   WHERE  pu.product_id = pr.product_id );
```

IN bir listede aranan verinin olup olmadığını kontrol eder. EXISTS sadece kayıtların varlığını kontrol ederken, IN ise gerçek verileri kontrol eder. Alt sorgularda EXISTS daha iyi sonuçlar verdiğinden tercih edilmelidir.

DISTINCT yerine EXISTS kullanımı

☒ *Önerilmez*

```
SELECT DISTINCT product_id,
       name
FROM   products pr,
       purchase pu
WHERE  pr.product_id = pu.product_id;
```

☑ *Önerilir*

```
SELECT product_id,
       name
FROM   products pr
WHERE EXISTS
   ( SELECT 1
      FROM  purchase pr
      WHERE pr.product_id = pu.product_id);
```

Sorguda gelen kayıtlarda tekrarlı olanları görüntülememek için DISTINCT kullanılır. EXISTS ise bir alt sorguda gelen kayıtlar içinde istenilenlerin olup olmadığını kontrol eder. DISTINCT, gelen sonuçlarda tekrarlı olanları belirlemeden önce sıralama yaptığıdan verimsizdir ve bu yüzden de EXISTS tercih edilmelidir.

OR yerine UNION kullanımı

☒ ***Önerilmez***

```
SELECT ...
FROM   ps_jrnl_header a
WHERE  jrnl_hdr_status = 'E'
OR EXISTS
     ( SELECT  'x'
       FROM   ps_jrnl_header
       WHERE  business_unit_js = a.business.unit_js
         AND  journal_id = a.journal_id
         AND  unpost_seq = a.unpost_seq
         AND  jrnl_hdr_status = 'E' );
```

☑ ***Önerilir***

```
SELECT    ...
FROM      ps_jrnl_header a
WHERE     jrnl_hdr_status = 'E'
UNION
SELECT    ...
FROM      ps_jrnl_header a,
          ps_jrnl_header b
WHERE     a.business_unit_js = b.business.unit_js
     AND  a.journal_id = b.journal_id
     AND  a.unpost_seq = b.unpost_seq
     AND  a.jrnl_hdr_status = 'E'
     AND  b.jrnl_hdr_status != 'E';
```

UNION kullanıldığında, optimizer kayıtları getirmek için iki benzer işlem gerçekleştirir. OR yapısında ise optimizer karar verirken daha karmaşık işlemler yapar ve daha az verimli sonuçlara ulaşır.

Düzensiz SQL sorguları

☒ ***Önerilmez***

İlk sorguda product_id için 1 değeri verilirken, ikincisi için 2 değeri verilmektedir.

```
SELECT * FROM products WHERE product_id = 1;
SELECT * FROM products WHERE product_id = 2;
```

Sorgular aynı olmalarına rağmen farklı yerlerde boşluk karakterleri ile karşılaşılmaktadır.

```
SELECT * FROM products WHERE product_id = 1;
SELECT *   FROM products   WHERE product_id = 1;
```

Sorgularda karakterlerin büyük harf ve küçük harfle yazılmasına önem verilmemiştir.

```
select * FROM products WHERE product_id = 1;
SELECT * from products WHERE product_id = 1;
```

Indeks kullanımını engelleyen durumlar

☒ *Önerilmez*

```
...
WHERE SUBSTR(account_name,1,7) = 'CAPITAL'
```

☑ *Önerilir*

```
...
WHERE account_name LIKE 'CAPITAL%'
```

SUBSTR indeks kullanımını iptal eder.

☒ *Önerilmez*

```
...
WHERE account != 0
```

☑ *Önerilir*

```
...
WHERE account > 0
```

NOT, !=, <> indeks kullanımını iptal eder.

☒ *Önerilmez*

```
...
WHERE TRUNC(trans_date) = TRUNC(sysdate)
```

☑ *Önerilir*

```
...
WHERE trans_date BETWEEN TRUNC(sysdate) AND
                         TRUNC(sysdate) + 0.99999
```

TRUNC indeks kullanımını iptal eder.

☒ ***Önerilmez***

```
...
WHERE account_name || account_type = 'AMEXA'
```

☑ ***Önerilir***

```
...
WHERE account_name = 'AMEX'
  AND account_type = 'A'
```

"||" (concatenate) indeks kullanımını iptal eder.

☒ ***Önerilmez***

```
...
WHERE account_name = NVL ( :acc_name, account_name )
```

☑ ***Önerilir***

```
...
WHERE account_name LIKE NVL ( :acc_name, '%' )
```

Account_name her iki tarafta da kullanılırsa indeks iptal edilir.

☒ ***Önerilmez***

```
...
WHERE emp_type = 123
```

☑ ***Önerilir***

```
...
WHERE TO_NUMBER(emp_type) = 123
```

Emp_type VARCHAR2 tipinde olduğu için farklı tipteki değişken ile karşılaştırıldığında indeks iptal edilir.

Sıralama için Indeks kullanımı

Indeksler, sıralama için fazladan işlem yapılamasını önlemek amacıyla da kullanılabilir. Genel olarak indeksler artan bir sırayla oluşturulurlar. Eğer sorgudaki ORDER BY kalıbında yer alan kolonlar indeksteki sıralarıyla aynıysa, bu sorguya ilgili indeksi kullandırmak verinin istenilen sırada ve hızlı bir şekilde getirilmesini sağlar.

☒ ***Önerilmez***

```
SELECT acc_name,
       acc_surname,
FROM   account_acct
ORDER BY account_name;
```

Bu sorgunun yardım cümleciği yazılarak indeks kullanması aşağıdaki şekilde sağlanabilir:

☑ ***Önerilir***

```
SELECT /* INDEX_ASC(acct acc_ndx1) */ acc_name,
       acc_surname,
FROM   account_acct
ORDER BY account_name;
```

Bazı durumlarda, ORDER BY yerine anlamsız bir WHERE koşulu eklenmesi daha hızlı çalışmayı sağlayabilir. Aşağıdaki örnekte WHERE acc_name > chr(1) kullanılarak indeks kullanılması ve böylece sıralama yapılması sağlanmıştır.

☑ ***Önerilir***

```
SELECT /* INDEX_ASC(acct acc_ndx1) */ acc_name,
       acc_surname,
FROM   account_acct
WHERE  acc_name > chr(1) ;
```

SQL YARDIM CÜMLECİKLERİ

Aşağıda genel olarak Oracle9i'de bulunan yardım cümlecikleri görülmektedir. Bu yardım cümleciklerinin çoğu daha önceki Oracle sürümlerinde de desteklenmektedir. Aşağıdaki açıklamalarda sadece söz dizilim yapısı değil, aynı zamanda bu yardım cümleciklerinin en sık kullanılım şekilleri de bulunmaktadır.

STAR

Sorgudaki en büyük tablonun birleştirmede en sona alınmasını temin eder. Veri ambarı uygulamalarında bu yardım cümleciği sıkça kullanılır. En az üç tablo kullanıldığında STAR daha verimlidir.

```
SELECT /*+ STAR */ h.horse_name, o.owner,
       r.position, r.location, r.race_date
FROM   results r, horses h, owners o
WHERE  h.horse_name like 'WI%'
   AND h.horse_name = r.horse_name
   AND r.owner = o.owner;
```

STAR_TRANSFORMATION

Boyut tablolarında ve ana tablolarda STAR yardım cümleciğine benzer şekilde kullanılır. En belirgin fark ise, Cost-Based Optimizer'ın sorguyu dönüştürmeye değip değmeyeceğine karar vermesidir. Dönüştürmede, SQL kalıbı alt sorgulara bölünerek bitmap indekslerden yararlanması sağlanır. Bu yardım cümleciğinin kullanılabilmesi için Oracle parametresi STAR_TRANSFORMATION_ENABLED=TRUE olarak ayarlanmalıdır. STAR_ TRANSFORMATION genellikle ana tablo kolonları üzerinde bitmap indeksleri kullanmaya çalışır ve bunu cümleyi alt sorgulara bölerek yapar.

```
SELECT /*+ STAR_TRANSFORMATION */
```

PARALLEL(table [,integer] [,integer])

Sorguyu çalıştırmak için kullanılacak eş zamanlı proses sayısını belirler. İlk parametre, tablo için paralelliğin seviyesini belirler. Her bir sorgu için koşacak proses sayısına denktir. İkinci seçimli parametre ise, sorgunun kaç parçaya bölünüp koşturulacağını belirler. Eğer PARALLEL(EMP,4,2) gibi bir tanımlama yapılırsa, iki ayrı sorgu parçası üzerinde 4 tane proses koşacak anlamına gelmektedir. Eğer hiçbir değer girilmezse, INIT.ORA ya da SPFILE'da tanımlı olan genel parametrelere uyulur. Bu yardım cümleciği select, update, insert ve delete cümlelerinde kullanılabilir. Paralel çalışma özelliğinin faydalı olması için veri tabanı dosyaları birçok disk üzerine yayılmış olmalıdır. Dolayısıyla PARALLEL için girilen paralellik değeri, dosyanın yayıldığı disk sayısını geçmemelidir. Ancak birçok disk kullanılmışsa, çok sayıda CPU'ya sahip olmak işlemleri daha da hızlandıracaktır.

```
SELECT /*+ PARALLEL (x 4) */ COUNT(*)
FROM   x;
SELECT /*+ PARALLEL (x 4 2) */ COUNT(*)
FROM   x;
UPDATE /*+ PARALLEL (x 4) */ x
SET position = position+1;

DELETE /*+ parallel(x 4) */ from x;
INSERT INTO x
SELECT /*+ PARALLEL(winners 4) */ *
FROM winners;
```

PARALLEL_INDEX(table, index, degree of parallelism, cluster split)

Bölünmüş indeksler için indeks taramalarının paralelleştirilmesini sağlar. Aşağıdaki örnekte de görüldüğü gibi, optimizerın EMP tablosu üzerinde paralel indeks taraması yapılması sağlanır.

```
SELECT /*+ PARALLEL_INDEX(emp, emp_ndx, 4, 2) */
```

FIRST_ROWS or FIRST_ROWS(n)

Çağırılan sorgunun ilk n kaydını en kısa sürede dönecek şekilde optimizasyon sağlar. SQL cümlesindeki herhangi bir tablo için istatistik bilgisinin olması şart değildir, optimizer tarafından bu istatistikler hesaplanır. Diğer erişim ("access path") yardım cümlecikleri, FIRST_ROWS cümleciğiyle birlikte kullanılabilir ve hatta FIRST_ROWS'u ezebilirler. Eğer dönmesi istenen satır sayısına dair (n) parametresi belirtilirse, Oracle daha doğru tahminde bulunur ve böylece daha doğru bir plan uygular. (n) seçeneği sadece Oracle9i ve üst sürümleri için geçerlidir.

- Tam tablo arama (Full table scan) yerine, her zaman indeks kullanır.
- Mümkünse sort/merge birleşimleri yerine, iç içe döngü birleşimlerini kullanır.
- Mümkünse, ORDER BY kalıbı için indeks kullanır.

DELETE ve UPDATE cümleleri, gruplama işlemi (UNION, INTERSECT, MINUS, GROUP BY, DISTINCT, MAX, MIN, SUM gibi) ya da FOR UPDATE kalıbı içeren sorgularda optimizer FIRST_ROWS yardım cümleciğini işleme almaz. Çünkü bu işlemlerde mutlaka tüm kayıtlara erişilmesi gerekeceğinden en hızlı cevap optimizasyonu yapılamaz.

```
SELECT /*+ FIRST_ROWS(100) */
```

FULL(table)

Belirtilen tablo üzerinde tam tarama yapılmasını sağlar.

```
SELECT /*+ FULL(emp) */ ename
FROM   emp
WHERE  commencement_date > sysdate - 7
```

Eğer tablo için kısa ad (alias) tanımlanmışsa, yardım cümleciğinde de bu kısa ad kullanılmalıdır:

```
SELECT /*+ FULL(a) */ ename
FROM   emp a
WHERE  a.commencement_date > sysdate - 7
```

INDEX(table [index [index...]])

Belirtilen tablo için indeksi üzerinden tam tarama yapılmasını sağlar. Yardım cümleciğinde bir ya da birden fazla indeksin adı da doğrudan tanımlanabilir. Eğer indeks adı verilmemişse, optimizer tüm indeksleri kontrol eder ve en uygun olanı ya da olanları plana dahil eder. Eğer doğrudan isimleri verilmiş ise, optimizer bu indeksler içinde uygun olan ya da olanları seçer. Sadece tek bir indeks verilmişse optimizer bu indeksi kullanır.

```
SELECT /*+ INDEX(EMP EMP_NDX1) */
SELECT /*+ INDEX(EMP) */
```

INDEX_ASC(table [index])

Belirtilen tablo üzerinde artan indeks taraması yapılmasını sağlar.

```
CREATE INDEX POST ON OWNERS (ZIPCODE) REVERSE.
SELECT /*+ INDEX_ASC(EMP EMP_NDX1) */...
```

INDEX_COMBINE(table [index [index...]])

Tablo bilgilerine ulaşmak için bitmap indekslere erişimi sağlar.

```
SELECT /*+ INDEX_COMBINE(ACCT_TRAN AT_STATE_BMI AT_TYPE_BMI) */
```

INDEX_DESC(table [index])

Belirtilen tablo üzerinde azalan indeks taraması yapılmasını sağlar. Oracle normalde varsayılan olarak artan indeks taraması gerçekleştirir. Bu yardım cümleciği ise bunun tam tersinin gerçekleşmesini sağlar. Tipik bir kullanım örneği olarak, banka hesabınızdaki işlemlerin tarihe göre azalan bir şekilde listelenmesi gösterilebilir. Dağıtılmış sorgularda bu yardım cümleciği büyük fayda sağlar.

```
SELECT /*+ INDEX_DESC(ACCT_TRANS ACCT_TRANS_DATE_NDX) */...
```

INDEX_FFS(table [index])

Optimizer'ın tam tablo taraması yerine tam indeks taraması yapmasını sağlar. WHERE kalıbında belirtilen kolonların hepsi indeks içinde mevcutsa bu yardım cümleciği verimli çalışır.

```
SELECT /*+ INDEX_FFS(ACCT_TRAN AT_STATE_NDX1) */
```

INDEX_JOIN(table [index] table [index2)]

Bu yardım cümleciği erişim yolu olarak belirtilen iki indeksin birleşiminin kullanılmasını sağlar. Aşağıdaki örnekte, tablonun HORSE_NAME ve OWNER adlı iki birincil anahtar indeksli kolonları ve IDENTIFIER adlı birincil olmayan anahtar indeksli kolon WHERE koşuluna dahildir.

```
SELECT /*+ INDEX_JOIN(HORSE_OWNERS HO_NDX2 HO_PK) */ Horse_name,
       owner
FROM   HORSE_OWNERS
WHERE  horse_name = 'WILD CHARM'
   AND owner = 'Mr M A Gurry'
   AND identifier = 10;
```

INDEX_JOIN yardım cümleciği kullanılmadığında, optimizer sadece tek kolonlu indeksleri bir araya getirir.

REWRITE

Belirtilen tablo üzerindeki sorguda eğer varsa materialized view kullanılmasını sağlar. Aşağıdaki örnekte, at yarışı sonuçlarını tutan bir tablo kullanılmaktadır. OWNER, HORSE_NAME, POSITION ve COUNT(*) bilgileri için materialized view şu şekilde oluşturulur :

```
CREATE MATERIALIZE VIEW LOG ON RESULTS
WITH ROWID,
PRIMARY KEY (HORSE_NAME, OWNER, RACE_DATE)
INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW winning_horse_owners_vw
USING INDEX
REFRESH ON COMMIT
ENABLE QUERY REWRITE AS
      SELECT horse_name, owner, position, COUNT(*)
      FROM   results
      GROUP BY horse_name, owner, position;
```

Bu materialized view'ın doğrudan erişim dışında da faydalı olması için, Oracle parametresi QUERY_REWRITE_ENABLED=TRUE olmalı ve ilgili şema QUERY REWRITE yetkisine sahip olmalıdır. Örneğin ;

```
GRANT QUERY REWRITE TO HROA;
```

Aşağıda görülen SQL sorgusu istediği tüm verileri materialized view'dan alabileceğinden optimizer tablo yerine materialized view'ı kullanmayı tercih eder.

```
SELECT /*+ REWRITE */ horse_name, owner, position, COUNT(*)
FROM   results
GROUP BY horse_name, owner, position;
```

ROWID(table)

Belirtilen tablo için ROWID bazında tablo taraması yapar. "rowid", kayıdın fiziksel disk adresine karşılık düşmektedir.

```
SELECT /*+ ROWID(a) */ ename
FROM emp a
WHERE rowid > 'AAAGJ2AAIAAABn4AAA'
AND surname like 'GURR%'
```

ALL_ROWS

Çalıştırılan sorguda tüm kayıtları getirmek için en az kaynak tüketimini sağlar. "OPTIMIZER_MODE=CHOOSE" durumunda çalışılıyorsa ve SELECT, UPDATE veya DELETE komutlarına uygulanırsa, SORT MERGE veya HASH JOIN içeren iç içe kurulan döngülerin (NESTED LOOP) ezilmesine neden olabilir.

```
SELECT /*+ ALL_ROWS */ ...
```

AND_EQUAL (table index1 index2[... index5])

Tek kolonlu indekslerin birleştirilmesini sağlar. En az iki indeks ve en fazla beş indeks kullanılabilir. Eğer WHERE koşulundaki ilk indeksten dönen kayıt arttıkça bu indeks birleştirme işleminin verimi de azalır.

```
SELECT /*+ AND_EQUAL(horse_owners ho_ndx1 ho_ndx2 ho_ndx3) */
      count(*)
FROM  horse_owners
WHERE horse_name = 'WILD CHARM'
  AND owner = 'Mr M A Gurry'
  AND identifier = 14;
```

APPEND

Tabloya doğrudan erişip kayıt eklenmesini sağlar. Eklenecek kayıtlar normalde girilen "Buffer Cache" alanını atlayarak doğrudan tablonun sonuna eklenir. Yükleme sırasında bütünlük kısıtları göz ardı edilir, fakat yükleme bittiğinde bu kontroller gerçekleştirilir.

```
INSERT /*+ APPEND */ * INTO y
SELECT FROM winners;
```

CACHE (table)

Optimizer'ın tam tablo erişimiyle (full table scan) elde edilen veri bloklarının tümünün bellekteki "Buffer Cache" alanına yüklemesini sağlar. Bu yardım cümleciği genellikle küçük tablolar için kullanılsa da eğer veri tabanı sunucusunda çok fazla bellek varsa, çok nadiren değişen büyük tablolar da bu şekilde belleğe yüklenebilir.

```
SELECT /*+ FULL(winners) CACHE(winners)*/
count(*)
FROM  winners
```

CHOOSE

Sorgudaki tablolar için veri tabanı istatistikleri mevcutsa optimizerın "cost-based" çalışmasını, eğer istatistik verisi yoksa "rule-based" çalışmasını sağlar.

```
SELECT /*+ CHOOSE */
```

MERGE(table)

Asıl dış sorgu ile içeride oluşturulan sorgunun (view) birleştirilmesini sağlar. Aşağıdaki örnekte GROUP BY'lı iç sorgu dışarıdaki OWNERS tablosundaki seçimlerle birleştirilir.

```
SELECT /*+ MERGE(w) */ o.owner,
       w.num_wins, o.suburb
FROM  owners o,
      (SELECT owner, count(*) num_wins
       FROM winners
       WHERE position = 1
       GROUP BY owner) w
WHERE o.owner = w.owner
  AND w.num_wins > 15
ORDER BY w.num_wins desc
```

NO_EXPAND

Sorgunun ayrı parçalara bölünüp işlenmesini önler.

```
SELECT /*+ NO_EXPAND */ COUNT(*)
FROM   horse_owners
WHERE  identifier < 10 OR identifier > 20
```

NO_MERGE(table)

İç sorgunun birleştirilmesini engeller.

```
SELECT /*+ NO_MERGE(w) */ o.owner,
       w.num_wins, o.suburb
FROM  owners o,
      (SELECT owner, count(*) num_wins
       FROM winners
       WHERE position = 1
       GROUP BY owner) w
WHERE o.owner = w.owner
  AND w.num_wins > 15
ORDER BY w.num_wins desc
```

NOAPPEND

APPEND yardım cümleciğinin aksine tablonun içine geleneksel şekilde kayıt eklenmesini sağlar. Yeni kayıtların tablonun sonuna ekleneceğinin garantisi yoktur. İçeri atılan kayıtlar "buffer cache" bellek alanından geçerler ve bütünlük kısıtlarıyla da kontrol edilirler.

```
INSERT /*+ NOAPPEND */ * INTO y
SELECT FROM winners;
SELECT /*+ FULL(winners) NOCACHE(winners) */ count(*)
FROM   winners
```

NOINDEX(table [index [index...]])

Sorguda belirtilen indekslerin planda hesaba katılmamasını temin eder.

```
SELECT /*+ NOINDEX(EMP EMP_NDX1) */
```

Eğer sadece tablo adı verilmişse tabloda tanımlı hiçbir indeks kullanılmaz.

```
SELECT /*+ NOINDEX(EMP) */
```

NOPARALLEL(table)

Oracle'ın belirtilen tablonun taranmasında çoklu proses kullanmasını engeller. Örneğin, aşağıdaki şekilde paralellik kurulmuş olsun:

```
ALTER TABLE x PARALLEL 2;
```

Bu durumda, tablo taranacağında Oracle iki paralel proses kullanır. Takip eden sorgu ise NOPARALLEL yardım cümleciği ile paralel proses oluşturulmasını engeller.

```
SELECT /*+ NOPARALLEL(x) */ COUNT(*)
FROM
```

NOPARALLEL_INDEX(table, index)

Bölünmüş (partitioned) indeks için paralel indeks işlenmesine engel olur.

```
SELECT /*+ NOPARALLEL_INDEX(emp, emp_ndx) */
```

NOREWRITE

Seçilen tablo için yazılan sorguların Oracle tarafından varsa ilgili "Materialized View"'ları kullanacak şekilde değiştirilmesine engel olur. REWRITE yardım cümlesinin tam tersidir.

```
SELECT /*+ NOREWRITE */ horse_name, owner, position, COUNT(*)
FROM   results
GROUP BY horse_name, owner, position
```

ORDERED

Optimizerin FROM kalıbındaki tabloları belirtilen sırada (soldan sağa) birleştirmesini sağlar. Raporlama ortamlarında bu yardım cümleciği çok yardımcı olabilir. Ne kadar çok tablo belirtilmişse bu yardım cümleciğinin faydası da o kadar artar. Aşağıda bir örneği görülebilir:

```
SELECT /*+ ORDERED */ acct_name, trans_date, amount, dept, address
FROM   trans t, account a, category c, branch b, zip z
WHERE  t.trans_date > sysdate - 30
   AND a.zip = z.zip
   AND z.state = 'WA'
   AND t.account between 700000 and 799999
   AND t.account = a.account
   AND a.account = 'ACTIVE'
   AND a.category = c.category
   AND c.catgory = 'RETAIL'
   AND t.branch_id = b.branch_id
   AND b.branch = 'BELLEVUE'
```

Genel olarak bir sorguyu belirleyen asıl tablonun hangisi olacağı indeksin tipine, indeksteki kolonların sayısına ve indeksteki kayıt sayısına bağlıdır. Örneğin, WHERE kalıbında eşitlik bulunan bir kolon üzerinde UNIQUE indeksi bulunan tablo, NON-UNIQUE indeksli ve eşitlikli bir tabloya göre öncelik kazanır.

RULE

Belirtilen komut için “Rule-based” optimizasyon yapar. INIT.ORA’daki OPTIMIZER_MODE Oracle parametresinin CHOOSE olarak ayarlanması ve tablo istatistiklerinin toplatılmaması da aynı işi görür.

UNNEST

İç içe sorgularda, alt sorgunun ana sorguyla birleştirilmesini ve optimizerin daha doğru karar vermesini sağlar. UNNEST sadece UNNEST_SUBQUERY=TRUE olarak Oracle parametresinin ayarlanmasıyla kullanılabilir.

```
SELECT /*+ UNNEST */ count(*)
FROM   horses
WHERE  horse_name LIKE 'M%'
   AND horse_name NOT IN
      ( SELECT horse_name
        FROM horse_owners
        WHERE owner LIKE '%Lombardo%');
```

USE_CONCAT

Optimizerin WHERE koşulundaki OR şartlarını UNION ALL şekline dönüştürmesini sağlar. Aşağıdaki örnekte, indeks, OR koşulunun iki tarafı için iki kez taranmaktadır. Gelen veriler ise daha sonra tek sonuç çıktısı için birleştirilmektedir.

```
SELECT /*+ USE_CONCAT */ COUNT(*)
FROM   horse_owners
WHERE  identifier < 10 OR identifier > 20
```

SQL TUNING LİNKLERİ

Oracle® Database Performance Tuning Guide
http://download-uk.oracle.com/docs/cd/B19306_01/server.102/b14211/sql_1016.htm

Oracle Enterprise Manager Database Tuning with the Oracle Tuning Pack
http://download-east.oracle.com/docs/html/A86647_01/toc.htm

Oracle Documentation
http://www.stanford.edu/dept/itss/docs/oracle/10g/nav/portal_3.htm

Oracle Technologies Sites (Articles, sample codes, blogs, tutorials etc.)
http://www.oracle.com/technology/tech/index.html

Oracle Ask Tom Forum
http://asktom.oracle.com/

Practical SQL: Tuning Queries
http://www.informit.com/articles/article.asp?p=27015&rl=1

Oracle FAQ Support
http://www.orafaq.com/search/SQL+TUNE