SQL Server에서 SQL 튜닝 시 알아야 할 힌트 와 사용 방법

㈜엑셈 컨설팅본부/DB컨설팅팀 박 성호

Optimizer 가 SQL 을 해석할 때 항상 최적의 실행계획을 생성하지는 못한다. 복잡한 SQL 일수록 최적의 실행계획을 생성하기 위해 고려해야 할 대상 (Table, Index 가 많은 경우)이 많기 때문에, 실행계획 생성 시 SQL 의 Cost 를 잘못 계산하여 최적의 실행계획을 세우지 못하는 경우가 발생한다. SQL 작성자가 SQL 을 아무리 효율적으로 작성하더라도 Optimizer 는 가끔 비효율적인 실행계획으로 작성자를 당황하게 하는 경우들이 있다. 그런데 이러한 성능문제를 해결하기 위해서는 Optimizer 가 SQL 의 올바른 Access Path (조인순서, 조인방법, 데이터 액세스 등)를 가진 가장 효율적인 실행계획을 수립할 수 있도록 SQL 작성자의 개입이 필요하다. 왜냐하면, Optimizer 가 비효율 실행계획을 수립하였지만, 실행계획을 수립하는 것은 SQL 이 가진 오브젝트나 기타 관련 정보를 토대로 SQL 의 Cost 를 계산하는 것이므로 비효율 실행계획을 수립하는 원인을 SQL 작성자가 찾아내기란 쉽지 않다. 그러므로, Optimizer 에게 SQL 작성자의 의도를 전달하여 효율적인 Access Path 를 가진 실행계획을 수립할 수 있도록 도와줄 필요가 있다.

위와 같은 경우에 해당 SQL 에 힌트를 사용하는 것을 고려할 수 있다. 왜냐 하면, 힌트는 SQL 작성자가 의도한 대로 Optimizer 가 SQL 을 해석할 수 있도록 유도할 수 있는 키워드이기 때문이다. 또한, 힌트는 SQL 에 적용하더라도 결과 (추출 데이터)에 영향을 전혀 주지 않기 때문에 데이터 무결성에 대한 부담은 가지지 않아도 된다.

그러나 SQL Server 에서의 힌트 사용은 타 DBMS 보다도 더 주의해야 한다. 잘못된 힌트 사용으로 인해 SQL 구문 오류가 발생하여 Application 이 동작하지 않는 위험한 상황을 연출할 수 있기 때문이다. 그리고 잘못 사용된 힌트로 인해 SQL 의 성능문제를 개선하지 못하고, 성능문제를 더욱 가중 시킬 수 있으므로 주의해서 사용해야 한다.

이렇게 힌트는 Optimizer 가 제대로 효율적인 실행계획을 수립하지 못한 경우에 사용하면 SQL 의 성능 개선하는데 효과적이다. 그러나 잘못 사용된 힌트는 SQL 의 성능문제 외의 또 다른 문

제를 야기할 수 있기 때문에, 힌트를 제대로 사용하기 위해 반드시 올바른 사용법을 알고 있어야 한다.

언제 힌트를 사용하는 게 효율적인가?

Optimizer가 비효율 실행계획을 수립하는 경우

SQL 에 힌트를 사용하는 가장 대표적인 경우는 SQL 작성자가 의도하지 않은 비효율 실행계획으로 수행되어 성능문제를 일으키는 경우일 것이다. 대개 이런 경우는 Optimizer 가 SQL 이 가진 정보를 토대로 Cost 계산하여 비효율 실행계획을 수립하는 경우로, 이럴 때 효율적인 실행계획으로 수행될 수 있도록 개입이 필요하다. 물론, SQL 의 성능문제를 SQL 을 재 작성하여 해결할 수 있다면 가장 바람직한 경우이겠지만, SQL 을 재 작성한다고 해서 항상 효율적인 실행계획으로 수행된다고 보장할 수는 없다. 또한, SQL 의 성능문제를 해결하는데 소요시간 과다로 인해 성능문제 해결이 지연됨으로써 더욱 심각한 문제를 초래할 수 있으므로, 이럴 때는 SQL 의 실행계획을 제어하는 힌트를 적용하여 빠른 조치를 하는 것이 바람직할 수 있다.

인덱스 구성의 변경에 의한 실행계획 이상

Optimizer 가 SQL 의 Cost 계산 시 중요한 정보 중 하나가 인덱스 구성정보이다. 그런데 기존 테이블에 인덱스가 추가 생성되거나, 기존 인덱스가 삭제되는 등 인덱스 구성에 변경이 되는 경우 SQL 의 실행계획에 이상이 발생할 소지가 있고, 이로 인해 성능문제를 유발할 수 있다. 이런 경우 SQL 의 성능문제를 해결하는 방안으로 힌트를 사용할 수 있다.

그러면 이제부터 SQL Server 에서 SQL 튜닝 시 알아야 할 기본 힌트에 대해 알아보고, 그 사용방법을 간단한 테스트를 통해 알아 보도록 하자. 테스트를 위해 아래의 스크립트를 사용하여 테스트 데이터를 생성하자.

Script. 테스트에 사용될 스크립트

```
create database pshdb;
```

```
go
use pshdb;
go
if object_id('sql_t1') is not null
   drop table sql_t1
go
if object_id('sql_t2') is not null
   drop table sql_t2
go
if object_id('sql_t3') is not null
   drop table sql_t3
go
create table sql_t1 (id varchar(10), name varchar(5), regdate varchar(8))
go
create table sql_t2 (id varchar(10), telseq int, telno varchar(14), teldiv
varchar(1))
go
create table sql_t3 (id varchar(10), addrseq int, addr varchar(100), addrdiv
varchar(1))
go
insert into sql_t1 values ('cust1', 'aman', '20120101')
go
insert into sql_t1 values ('cust2', 'bman', '20110101')
go
insert into sql_t1 values ('cust3', 'cgirl', '20130101')
go
insert into sql_t2 values ('cust1', 3, '010-2222-2222','b')
go
insert into sql_t2 values ('cust2', 1, '010-1111-1111','a')
go
insert into sql_t2 values ('cust2', 2, '010-1111-1004','b')
go
insert into sql_t2 values ('cust3', 4, '010-3333-3333','c')
go
insert into sql_t3 values ('cust1', 2, '서울시 송파구', '2')
go
insert into sql_t3 values ('cust2', 1, '서울시 서초구','1')
go
insert into sql_t3 values ('cust3', 3, '서울시 강서구','3')
go
insert into sql_t3 values ('cust3', 4, '서울시 양천구','4')
```

```
go
create index idx01_sql_t1 on sql_t1 (id)
go
create index idx02_sql_t1 on sql_t1 (name)
go
create index idx01_sql_t2 on sql_t2 (id, telno)
go
create index idx02_sql_t2 on sql_t2 (telno)
go
create index idx01_sql_t3 on sql_t3 (id, addr)
go
create index idx02_sql_t3 on sql_t3 (addr)
go
```

힌트의 종류와 사용방법

힌트의 사용목적은 SQL 의 실행계획을 효율적인 Access Path 로 유도하고, 힌트를 적용한 시점의 실행계획을 유지하는 것이다. 이러한 힌트의 사용목적을 충족하면서 힌트 구문을 잘 적용하기 위해서는 먼저 SQL 의 효율적인 Access Path 를 판단하기 위한 필수 3 요소를 알아야 한다. 왜냐하면, 성능문제를 가진 모든 SQL 을 개선하는데 이 3 요소만 잘 점검하면 거의 대부분의 성능문제는 개선할 수 있고, SQL 의 실행계획을 고정하는데 반드시 필요하기 때문이다.

[SQL 튜닝 시 알아야 할 Access Path 의 구성요소]

- 조인순서: SQL 수행 시 먼저 수행되는 테이블(Driving Table)과 FROM 절의 테이블간의 조인순서는 항상 존재한다. 이러한 조인순서는 SQL 을 수행할 때 효율적인 Access Path 로 수행되기 위한 가장 중요한 요소이다.

- 조인방법: 선행 테이블과 후행 테이블 간의 조인방법, 그리고 선행 결과 셋과 후행 테이블의 조인방법을 결정하는 것은 SQL 의 효율적인 Access Path 유도하는데 필수 요소이다.

- 데이터 액세스: 데이터를 액세스 할 때 어떤 방법 (Index Scan OR Full Table Scan 등)으로 수행할 지를 결정하는 것으로, SQL 의 효율적인 Access Path 를 결정하는데 반드시 필요한 요소이다. 또한, 조인연결 칼럼에 대해서도 고려해야 한다.

Optimizer 가 SQL 에 대한 최적의 실행계획을 세울 때 항상 존재하고, 반드시 결정되어야 하는 요소들이 바로 앞에서 알아본 조인순서, 조인방법, 데이터 액세스이다. Optimizer 가 최적의 실

행계획을 세우기 위해 반드시 결정하여야 하는 이 요소들이 바로 SQL 에 힌트를 적용할 때 반드시 적용해야 하는 구성요소가 된다. 왜냐하면, 힌트의 사용목적은 SQL 을 효율적인 실행계획으로 수립하고 수립된 실행계획은 변하지 않도록 하는데 있기 때문이다. 최소한 이 세가지 구성요소들에 대해 정의가 되어야 기본 형태의 SQL 에 대한 효율적인 실행계획을 수립하고, 수립된 실행계획을 고정시켜, 실행계획 변경에 의한 성능문제를 피할 수 있다. 물론, 힌트를 적용할 때 기본 형태의 SQL 에 뷰, 서브쿼리 등에 대한 부분도 고려해야 하지만, 가장 중요한 힌트 적용의 기본 형태는 조인순서, 조인방법, 데이터 액세스를 효율적으로 결정하는 것이다. 그러면 테스트를 통해 조인순서, 조인방법, 데이터 액세스 관련 힌트와 그 사용방법을 알아 보도록 하자.

조인순서 힌트

- 힌트: force order
- 힌트 의미: From 절에 나열된 순서대로 조인순서를 유도하는 힌트

From 절에 나열된 순서가 조인순서가 되기 때문에, 최초 SQL 에 FORCE ORDER 힌트를 적용하기 전 From 절의 순서를 먼저 조정해야 한다. 또한, SQL 을 수정할 때 (테이블 순서가 조정될 때)는 SQL 에 적용된 힌트가 효율적인지를 다시 점검해야 한다.

조인순서[1]. SQL_T1, SQL_T2 순서로 수행

```
select *
  from sql_t1 t1 inner join sql_t2 t2
       on t1.id = t2.id
option (force order)

|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1003]))
     |--Nested Loops(Inner Join, OUTER REFERENCES:([t1].[id]))
     |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]))
     |    |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t2].[idx01_sql_t2] AS [t2]),...
     |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]),...
```

조인순서[1]은 SQL_T1 테이블을 먼저 수행 후 SQL_T2 테이블과 Nested Loops Join 으로 수해되었다. SQL 을 확인해 보면, SQL 의 맨 마지막 라인에 option (force order) 구문을 적용하

여 조인순서가 From 절 순서대로 수행되도록 유도한 것을 알 수 있다. From 절 순서는 SQL_T1, SQL_T2 순이다.

조인순서[2]. SQL_T2, SQL_T1 순서로 수행

```
select *
  from sql_t2 t2 inner join sql_t1 t1
       on t2.id = t1.id
option (force order)

|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1003]))
     |--Nested Loops(Inner Join, OUTER REFERENCES:([t2].[id]))
     |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]))
     |    |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t1].[idx01_sql_t1] AS [t1]),...
     |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]), ...
```

앞의 조인순서[1]과 같이 힌트 구문을 동일하며, From 절의 테이블 순서를 SQL_T2, SQL_T2 순서대로 나열하여 조인순서를 조정하였다.

조인방법 힌트

- 힌트: hash, loop, merge
- 힌트 의미: 조인방법을 유도하는 힌트

조인방법[1]. 일괄적으로 특정 조인방법을 지정하는 경우

조인방법[1]은 테이블 조인은 SQL_T1, SQL_T2, SQL_T3 순서로 수행되고, SQL_T2 와 SQL_T3 테이블을 Hash Join 으로 수행하도록 힌트를 적용한 것이다. SQL(1)은 SQL 의 OPTION 절에 조인순서와 조인방법을 유도하는 힌트를 추가하는 것이고, SQL(2)는 ANSI SQL 의 조인 구문에 각 힌트를 지정한 것이다.

```
SQL(1)

select *
  from sql_t1 t1
       inner join sql_t2 t2 on t1.id = t2.id
```

```
       inner join sql_t3 t3 on t1.id = t3.id
option (force order, hash join)

|--Hash Match(Inner Join, HASH:([t2].[id])=([t3].[id]), ...
     |--Hash Match(Inner Join, HASH:([t1].[id])=([t2].[id]),...
     |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]))
     |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]))
     |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t3] AS [t3]))

SQL(2)

select *
  from sql_t1 t1
       inner hash join sql_t2 t2 on t1.id = t2.id
       inner hash join sql_t3 t3 on t1.id = t3.id
option (force order)

|--Hash Match(Inner Join, HASH:([t2].[id])=([t3].[id]), ...
     |--Hash Match(Inner Join, HASH:([t1].[id])=([t2].[id]), ...
     |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]))
     |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]))
     |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t3] AS [t3]))
```

조인방법[1]의 SQL(1)과 SQL(2)는 동일한 조인순서와 조인방법으로 수행된다. 그런데 SQL(1)과 SQL(2)의 테이블 중 SQL_T2 는 Hash Join 으로 수행하고, SQL_T3 는 Nested Loops Join 으로 수행해야 하는 경우와 같이 Hash Join 이나 Nested Loops Join 등 한가지 조인방법만 가지지 않는다면 항상 SQL 을 SQL(2)와 같이 작성(ANSI SQL) 후 각 테이블의 조인방법을 별도로 지정해야 한다. 각 테이블의 조인방법 지정은 다음의 조인방법[2]에서 확인해 보자.

조인방법[2]. 각 테이블 별 조인방법을 지정하는 경우

앞에서 언급했듯이 From 절의 테이블들에 각각 조인방법을 달리 적용해야 한다면 다음의 SQL(1), SQL(2)와 같이 ANSI SQL 의 조인 구문에 힌트를 추가하면 된다.

```
SQL(1). SQL_T2 는 Hash Join, SQL_T3 는 Nested Loops Join

select *
  from sql_t1 t1
       inner hash join sql_t2 t2 on t1.id = t2.id
```

```
       inner loop join sql_t3 t3 on t1.id = t3.id
option (force order)

|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1006]))
     |--Nested Loops(Inner Join, OUTER REFERENCES:([t2].[id]))
     |    |--Hash Match(Inner Join, HASH:([t1].[id])=([t2].[id]), ...
     |    |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]))
     |    |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]))
     |    |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t3].[idx01_sql_t3] AS [t3]),...
     |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t3] AS [t3]),...
```

SQL(2). SQL_T2 는 Nested Loops Join, SQL_T3 는 Hash Join

```
select *
  from sql_t1 t1
       inner loop join sql_t2 t2 on t1.id = t2.id
       inner hash join sql_t3 t3 on t1.id = t3.id
option (force order)

|--Hash Match(Inner Join, HASH:([t2].[id])=([t3].[id]),...
     |--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1003]))
     |    |--Nested Loops(Inner Join, OUTER REFERENCES:([t1].[id]))
     |    |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]))
     |    |    |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t2].[idx01_sql_t2]...
     |    |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]),...
     |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t3] AS [t3]))
```

SQL(3)

```
select *
  from sql_t1 t1,
       sql_t2 t2,
       sql_t3 t3
 where t1.id = t2.id
   and t1.id = t3.id

|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1003]))
     |--Nested Loops(Inner Join, OUTER REFERENCES:([t3].[id]))
     |    |--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1006]))
     |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([t1].[id]))
     |    |    |    |--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]))
```

```
|    |    |    |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t3].[idx01_sql_t3]...
|    |    |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t3] AS [t3]),...
|    |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t2].[idx01_sql_t2] AS [t2]),...
|--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t2] AS [t2]),...
```

각 테이블의 조인방법을 다르게 적용해야 하는 경우에 성능문제를 발생시키는 SQL 이 SQL(3)과 같이 작성되어 있다면, SQL Server 가 제공하는 조인방법 힌트를 적용할 수 없다. (단, OPTION 절을 이용하여 모든 테이블의 조인방법을 지정하는 것은 가능하다.) 이런 경우 SQL 을 조인방법[2]의 SQL(2)와 같이 재 작성하여 각 테이블에 조인방법 힌트를 적용하면 된다.

데이터 액세스 힌트

- 힌트: index
- 힌트 의미: 데이터 액세스 시 인덱스를 사용하여 수행하도록 유도하는 힌트

SQL_T1 테이블의 ID 칼럼에는 인덱스가 생성되어 있다. 그러나 SQL_T1 테이블에 입력된 데이터가 3 건으로 많지 않아, Optimizer 는 SQL(1)과 같이 Full Table Scan 으로 수행하였다. 그런데 SQL_T1 테이블의 데이터에 대한 액세스 방식이 Full Table Scan 보다 인덱스 스캔이 유리하다고 가정한다면 SQL(2), SQL(3)과 같이 테이블 명 뒤에 인덱스 힌트를 적용하여 원하는 인덱스를 사용할 수 있다.

```
SQL(1)

declare @p0 varchar(10) = 'cust1'
select *
  from sql_t1 t1
 where id = @p0

|--Table Scan(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]), ...

SQL(2)

declare @p0 varchar(10) = 'cust1'
select *
  from sql_t1 t1 with(index(idx01_sql_t1))
 where id = @p0
```

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]))
     |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t1].[idx01_sql_t1] AS [t1])...
     |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]),...

SQL(3)

declare @p0 varchar(10) = 'cust1'
select *
  from sql_t1 t1 with(index=idx01_sql_t1)
 where id = @p0

|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]))
     |--Index Seek(OBJECT:([pshdb].[dbo].[sql_t1].[idx01_sql_t1] AS [t1]), ...
     |--RID Lookup(OBJECT:([pshdb].[dbo].[sql_t1] AS [t1]), ...
```

힌트 사용 시 주의할 점

잘못된 인덱스 명 사용에 의한 구문 오류

운영환경에서 SQL 의 성능문제를 해결하기 위해서 인덱스 힌트를 적용해야 하는 경우가 있겠지만, SQL Server 에서 일반적으로 Optimizer 가 효율적인 실행계획을 수립하기 때문에, 인덱스 힌트를 남발해서는 안 된다. 왜냐하면, 인덱스 힌트 구문 안에 기술된 인덱스가 삭제되면 해당 SQL 은 구문 오류가 발생하여 수행되지 않기 때문이다. 이러한 SQL 이 중요한 업무 처리를 수행하는 Application 내에 존재한다면 성능문제 외로 심각한 문제를 일으킬 수 있으므로 특히 주의해야 한다. 그리고, SI 프로젝트 시 (개발환경) SQL 에 인덱스 힌트를 적용할 때는 DBA 와 협의 하에 운영환경의 Naminig Rule 을 반드시 지켜야 한다. 그렇지 않으면 개발 시 인덱스 명과 운영환경의 인덱스 명이 달라 문제를 일으킬 수 있기 때문이다.

Hint를 적용한 SQL이 자주 변경되는 경우

SQL 이 자주 변경되는 것은 업무가 자주 변경된다는 것이다. 이런 SQL 은 힌트를 적용하는 것이 부적절하다. 특히, SQL 이 변경된 후 조인순서, 조인방법, 데이터 액세스를 다시 점검하고 힌트를 재 조정해야 하는 경우라면 적용된 힌트에 의해 성능문제가 발생할 소지가 높으므로 힌트 적용 외로 SQL 의 성능문제를 개선할 수 있는지 확인해야 한다.

Dynamic SQL에 적용된 Global Hint

여기서 Dynamic SQL 은 조회 조건에 따라 SQL 이 변경되는 SQL 을 의미한다. 이런 Dynamic SQL 은 하나의 SQL 로 보이지만, 하나의 SQL 이 아니다. 하나의 SQL 은 하나의 실행계획만 가지지만, 이런 Dynamic SQL 은 조회 조건에 따라 각기 다른 실행계획으로 수립되어 수행되어야 효율적인 수행이 될 수 있기 때문에 하나의 SQL 이 아니고, 각 조회 조건에 따라 다른 SQL 이 된다. 그런데 이런 Dynamic SQL 의 모든 조회 조건에 일괄 적용되도록 힌트를 부여하면, 특정 조회 조건에만 성능이 효율적일 뿐 다른 조회 조건에는 도리어 성능을 악화 시키는 원인이 될 수 있다. 그러므로 이런 경우에는 힌트를 조회 조건에 맞게 적용될 수 있도록 힌트 부여도 Dynamic 하게 적용해야 한다.

이제까지 SQL Server 에서 SQL 튜닝 시 사용되는 힌트와 그 사용방법에 대해서 알아보았다. SQL Server 의 기본적인 가이드는 SQL 에 대해 Optimizer 가 효율적인 실행계획을 수립하므로, 웬만하면 힌트를 적용하지 말라는 것이다. 필자가 생각하기에도 그 말이 맞다. 그런데, 특정 Application 의 SQL 성능문제로 인해 운영중인 DB 의 성능이 위급한 상황에 놓이는 경우, Optimizer 만 믿고 기다릴 수는 없다. 그럴 경우 불가피하게 힌트를 적용해야 할 경우가 있다. 그러므로 앞에서 알아 본 힌트 구문에 대한 이해와 향후 활용할 수 있도록 여러 테스트를 수행하여 그 사용방법을 익혀두길 바란다.