Tech Notes(vol.7)

Informix Dynamic Server 7.3의 새로운 기능 : 관련 하위 질의(Correlated Subquery) 확장 및 옵티마이저 지시문

소개

formix Dynamic Server 버전 7.3 데이터베이스 엔진은 Informix OnLine Dynamic Server 가 소개된 이래로 가장 기능이 풍부하고 많은 사람의 주목을 받은 엔진 중의 하나로서 신뢰성(Reliability), 가용성(Availability), 서비스(Serviceability), 즉 RAS 에 중점을 두고 있습니다. 이러한 세 가지 요소도 훌륭하지만 가장 관심을 끄는 것은 새로운 기능입니다. 이 새로운 기능에는 성능 향상과 다른 데이터베이스 플랫폼으로부터의 이식성을 개선하는 내용이 포함되어 있습니다. 향상된 기능 중 일부를 소개하면 다음과 같습니다.

- 확장 SQL
- 상위, 하위 및 향상된 부분 문자열 기능 등의 문자열 처리 루틴
- NVL(Null Value) 기능
- Decode/Case 문
- 키 우선 스캔(Keyfirst scans)
- 처음의 N 행 선택
- 관련 하위 질의(Correlated Subquery) 확장
- 옵티마이저 지시문

성능을 중시하는 사람이라면 아마도 마지막 두 가지 개선점, 즉 관련 하위 질의(Correlated Subquery) 확장과 옵티마이저 지시문에 가장 흥미를 가질 것입니다.

관련 하위 질의(Correlated Subquery; CSQ) 확장

과거에는 성능에 관한 글을 쓰거나 프리젠테이션을 할 경우 관련 하위 질의(CSQ)의 성능이 그다지 좋지 않았기 때문에 사용을 반대하는 입장이었으며 지금도 CSQ 사용을 별로 권장하지는 않습니다. 그러나 값이 다른 테이블에 존재하는 어떤 테이블에서 모든 행을 업데이트하는 다음과 같은 경우를 가정해 봅시다.

```
update customers set stat = "A"
where exists (
    select "X"
     from orders o
     where o.custid=customer.custid
       and o.cmpny = customer.custid
       and o.stat = "OPEN" )
```

이 예는 “OPEN” 주문을 가진 고객의 상태를 모두 업데이트하고자 하는 경우입니다. 이것을 실행하기 위한 유일한 방법은 위의 SQL 문을 사용하는 것입니다. 그러나 이 SQL 문을 사용하는 경우에는 고객 테이블의 모든 레코드에 대해 주문에 대한 하위 질의를 실행한다는 데 문제가 있습니다. 즉, 고객 테이블의 행이 백만 개인 경우에는 하위 질의를 백만 번 실행하게 되는데 백만 명의 고객 중에 몇 명의 고객만이 “OPEN” 주문을 한 경우라면 이러한 SQL 문은 매우 비효율적인 방법이 될 것입니다.

이 구문을 실행하는 가장 효율적인 방법은 가능하면 색인으로 주문 테이블을 검색하여 “OPEN” 상태의 주문을 모두 찾은 다음, 고객 테이블을 검색하여 관련된 레코드를 찾고 그 레코드를 업데이트하는 것입니다. 버전 7.3 이 출시되기 전에는 이런 방법으로 하위 질의를 실행하지 않아서 여러 가지 불편한 점이 많았지만 버전 7.3 이 출시되면서 이러한 점이 수정되었으며 보다 효율적인 방법으로 하위 질의를 실행할 수 있게 되었습니다.

버전 7.2 에서 질의에 대한 “Set Explain” 출력을 살펴보면 다음과 같습니다.

```
QUERY:

update orders set ship_charge = 0
where exists (
   select "X" from customer c
   where c.customer_num = orders.customer_num
     and c.state = "MD" )
```

```
1) Informix.orders: SEQUENTIAL SCAN

    Filters: EXISTS <subquery>


      Subquery:

      ---------

      Estimated Cost: 1

      Estimated # of Rows Returned: 1


      1) Informix.c: INDEX PATH

      Filters: Informix.c.state = 'MD'

      (1) Index Keys: customer_num

            LowerIndexFilter:

         c.customer_num = orders.customer_num
```

주문 테이블에 “SEQUENTIAL SCAN”이 있으며 하위 질의의 고객 테이블에 “INDEX PATH”가 있음에 유의하십시오. 이 경우 데이터베이스 엔진은 모든 주문 레코드를 읽어 각 주문에 대한 고객 테이블에 대해 하위 질의를 수행하게 되는데, 이 방법이 매우 비효율적인 경우도 있습니다.

Informix 에서는 이러한 문제를 해결하기 위해서 “하위 질의 평면화(Subquery Flattening)”라는 기능을 구현하였는데 그 개념은 질의가 조인(join)으로 변환되어 조인으로서 최적화되고 실행된다는 것입니다. 위에서 기술한 Set Explain 출력의 질의에서, Maryland(MD)에 사는 고객들의 주문을 선택하고자 한다면 다음과 같은 SQL 문으로 할 수 있습니다.

```
select orders.*
   from orders, customers c
   where c.customer_num = orders.customer_num
```

이 경우에 Informix 옵티마이저에서는 “MD”라는 주(state)에 대한 주문 테이블을 검색하고, 그런 후에 고객 테이블을 검색할 것입니다. 이제 버전 7.3 의 경우 같은 질의에 대해 “Set Explain”의 출력을 살펴봅시다.

```
QUERY:
```

```
update orders set ship_charge = 0
   where exists (
      select "X" from customer c
      where c.customer_num = orders.customer_num
        and c.state = "MD" )

1) Informix.c: SEQUENTIAL SCAN
Filters: Informix.c.state = 'MD'

2) Informix.orders: INDEX PATH
(1) Index Keys: customer_num
    Lower Index Filter:
       orders.customer_num = c.customer_num

NESTED LOOP JOIN
```

Set Explain 출력에는 하위 질의가 없으며 옵티마이저에서는 이제 “MD”라는 주에 기반을 둔 고객을 연속적으로 스캔하기 시작함을 알 수 있습니다. 주에 기반을 둔 고객에 대한 색인을 추가함으로써 성능을 더욱 향상시킬 수 있습니다. 이 출력을 통해서 볼 때, 옵티마이저에서는 Maryland 에 사는 고객을 찾아내어 그러한 고객의 주문을 검색하는, 가장 효과적인 경로를 선택하는 것을 알 수 있습니다. 이제 모든 주문 레코드를 읽고 고객 테이블에 하위 질의를 하는 일을 더 이상 하지 않습니다.

“하위 질의 평면화”를 이용하는 또 다른 새로운 기능은 중복 건너뛰기(Skip Duplicate) 색인 스캔입니다. 이 스캔 기능은 동일한 값에 대한 두 번째 테이블을 여러 번 검색하는 것을 방지하는 것이 그 목적입니다. 다음과 같은 set explain 출력을 가정해 봅시다.

```
QUERY:

update orders set backlog = "Y"
   where exists (
```

```
      select "X" from items
      where orders.order_num = items.order_num
        and stock_num = 6 and manu_code = "SMT" )

1) Informix.items: INDEX PATH (Skip Duplicate)
   Filters:(items.stock_num=6 AND
         items.manu_code='SMT')
   (1) Index Keys: order_num

2) Informix.orders: INDEX PATH
   (1) Index Keys: order_num
      Lower Index Filter:
         orders.order_num = items.order_num

NESTED LOOP JOIN
```

이 경우에 데이터베이스 엔진은 조건에 맞는 레코드에 대한 항목 테이블을 검색할 것이며 order_num 에 기반을 둔 연관 레코드에 대한 주문 테이블을 검색할 것입니다. 여기서 똑같은 order_num 이 검색 기준을 만족시켜 똑같은 주문 테이블이 검색되고 똑같은 레코드가 여러 번 업데이트될 가능성이 있습니다. 이런 경우에 “Skip Duplicate” 스캔을 수행하여 모든 중복을 건너뜀으로써, 단 하나의 고유한 값만을 반환하고 동일한 order_num 에 대해 여러 번 스캔하는 것을 피할 수 있습니다.

다음에 나오는 CSQ 확장은 첫 행/세미 조인(First Row/Semi Join)의 개념입니다. 이것은 첫 번째 행만을 찾기 위해 조인의 두 번째 테이블을 스캔하는 것과 관련이 있습니다. 이것은 CSQ 가 평면화(flatten)되고 조인된 두 번째 테이블이 EXISTS 절의 일부일 때 가능합니다. 데이터베이스 엔진은 테이블을 검색하고 한 행을 발견하면 검색을 중단할 수 있습니다. 한 행 또는 수 천 행이 발견된 경우에 EXISTS 조건이 참이면 엔진이 첫 번째 행 이후에 검색을 중단할 수 있습니다. 이러한 예는 다음과 같은 set explain 출력에서 볼 수 있습니다.

```
QUERY:

UPDATE PS_JRNL_LN SET jrnl_line_status = '3'
   WHERE BUSINESS_UNIT='ABC'
     AND PROCESS_INSTANCE=5960
     AND EXISTS (
         SELECT 'X'
         FROM PS_COMBO_SEL_06 A
         WHERE A.SETID='ABC'
           AND A.COMBINATION='OVERHEAD'
           AND A.CHARTFIELD='ACCOUNT'
           AND PS_JRNL_LN.ACCOUNT BETWEEN A.RANGE_FROM_06
           AND A.RANGE_TO_06)

Estimated Cost: 79
Estimated # of Rows Returned: 1

1) sysadm.ps_jrnl_ln: INDEX PATH
   (1) Index Keys: process_instance business_unit

2) Informix.a: INDEX PATH (First Row)
   Filters: (Informix.a.range_to_06 =
          ps_jrnl_ln.account AND
          a.tree_effdt = <subquery> )
   (1) Index Keys: setid chartfield combination
          range_from_06 range_to_06
      Lower Index Filter: (a.setid = 'ABC'
            AND (a.combination = 'OVERHEAD'
            AND a.chartfield = 'ACCOUNT' ) )
      Upper Index Filter:
```

```
        a.range_from_06 <= ps_jrnl_ln.account

NESTED LOOP JOIN (Semi Join)
```

이러한 경우에 데이터베이스 엔진은 첫 번째 테이블을 읽은 다음, 첫 번째 행만을 찾기 위해 두 번째 테이블인 ps_combo_sel_06 을 읽습니다. 이러한 기능은 “세미 조인”이라고 할 수 있는데, 그 이유는 두 번째 테이블의 모든 행을 실제로 조인하는 것은 아니기 때문입니다.

소프트웨어 패키지를 위해 특별히 개선한 사항으로 “제어 블록에서 프로모션 예견(Predicate Promotion Across Control Blocks)”이라는 것이 있습니다. 이것은 매우 간단한 프로세스에 대해 긴 기술적 명칭을 붙인 것입니다. 기본적으로 이것은 가능한 경우 CSQ 를 상수로 대체하여 CSQ 사이에 서로 관련이 없도록 만드는 것을 목적으로 합니다. 다음과 같은 예를 살펴봅시다.

```
select * from ps_jrnl_ln
where business_unit = 'ABC'
and process_instance = 5960
and not exists
( select "X"
    from PS_SP_BU_GL_NONVW P
    where P.business_unit
           =ps_jrnl_ln.business_unit)
```

이 질의는 CSQ 로, 다음과 같이 관련이 없는 하위 질의로 다시 작성할 수 있습니다.

```
select * from ps_jrnl_ln
```

```
    where P.business_unit = 'ABC')
```

대부분의 사람들이 처음에 이러한 방법으로 질의를 작성하지만, 때때로 소프트웨어 패키지에서도 별다른 생각 없이 질의를 작성하며 위에서 기술한 것과 같은 결과를 만들어 냅니다. 이 경우에는 다음의 set explain 출력에서 알 수 있듯이 Informix 에서 CSQ 를 비-CSQ 로 변경할 수 있습니다.

```
QUERY:

select * from ps_jrnl_ln
   where business_unit = 'ABC'
     and process_instance = 5960
     and not exists
     ( select "X"
        from PS_SP_BU_GL_NONVW P
         where P.business_unit =
```

```
        ps_bus_unit_tbl_gl.business_unit = 'ABC'

2) ps_bus_unit_tbl_fs: INDEX PATH
   (1) Index Keys: business_unit descr (Key-Only)
      Lower Index Filter:
         ps_bus_unit_tbl_fs.business_unit =
               Ps_bus_unit_tbl_gl.business_unit

NESTED LOOP JOIN
```

하위 질의에서 “Lower Index Filter”가 ps_jrnl_in.business_unit 이 아닌 ‘ABC’와 동일한 business_unit 에 있음을 주목하십시오.

또 다른 바람직한 추가 기능으로 “상수 하위 질의 최적화(Constant Subquery Optimization)”라는 것이 있는데, 이것은 실행 시에 데이터베이스 엔진이 다음 규칙에 따라 질의의 첫 번째 행에 있는 EXISTS 또는 NOT EXISTS 절의 프로세스를 중단할 수 있게 하는 것입니다.

질의가 NOT EXISTS 이며 행이 발견되었을 경우, 혹은 질의가 EXISTS 이며 행이 발견되지 않은 경우

앞의 예에서 하위 질의가 행을 반환하지 않는 경우에는 ps_jrnl_In 테이블의 첫 레코드에서 질의가 중단됩니다.

이러한 확장 기능은 뷰와 연관된 것을 제외하고 CSQ 를 포함한 모든 SQL 문에 적용됩니다. 뷰는 이후에 출시하는 데이터베이스에서 다룰 예정입니다. 물론 이 문제에 대한 간단한 해결 방법이 있습니다. 뷰를 사용하지 않는 것이 그것입니다. 무엇보다도 CSQ 에 대한 확장 기능은 매우 뛰어나며 특히 CSQ 를 매우 많이 사용하는 응용 프로그램에서 훌륭한 결과를 보여 주었습니다. 이러한 응용 프로그램으로는 PeopleSoft 가 있는데 이 프로그램에서는 CSQ 를 대단히 많이 사용합니다. 7.3 의 베타 테스트 중에서, 단순히 데이터베이스 엔진을 7.3 버전으로 업그레이드만 했는데 4 시간 30 분에서 23 분으로 단축되었습니다.

옵티마이저 지시문

옵티마이저 지시문(optimizer directives)이란, 구문을 선택할 때 추가할 수 있는 주석으로서 옵티마이저에서 고려해야 할 경로를 알려줍니다. 이 지시문은 다른 데이터베이스의 "힌트(Hints)"와 비슷하나 Informix 에서 사용하는 용어로 그 기능이 더 우수합니다. Informix 의 지시문은 보다 많은 기능을 가지고 있습니다. 또한 힌트는 무시해도 좋을 옵티마이저의 선택에 영향을 주는 반면, Informix 의 지시문은 반드시 따라야만 하는 경로를 지시합니다. 이러한 지시문을 제공하는 가장 주된 목적은 데이터베이스 엔진이 올바른 경로를 찾지 못해 개발자가 문제를 겪게 될 몇몇 상황을 해결하는 데에 도움을 주기 위한 것입니다. Informix 로서는 이 옵티마이저가 100 퍼센트 완벽한 것이기를 원하겠지만 이것은 사실상 불가능합니다. 개발자들이 지시문을 사용해야만 하는 상황을 발견할 경우 Informix 에서는 이를 파악하여 옵티마이저를 수정하고 계속 발전시켜 나가고자 할 것입니다.

지시문은 옵티마이저에서 선택해야 하는 경로의 범위를 제한합니다. 다음 원이 옵티마이저에서 선택할 수 있는 경로의 범위라고 가정하면 지시문은 그러한 선택을 작은 원의 내부로 제한하는 것입니다. 부정 지시문의 경우, 선택은 작은 원의 외부로 제한됩니다.

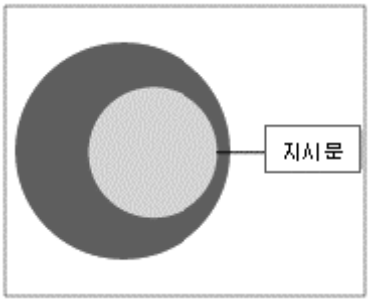


그림 1: 지시문이 옵티마이저의 선택을 제한하는 방법에 대한 시각적 표현.

지시문의 문법은 다음과 같습니다.

SELECT --+ directive text

SELECT {+ directive text }

UPDATE --+ directive text

UPDATE {+ directive text }

데이터베이스 응용 프로그램 성능: 성능의 요소

소개

Moore 의 법칙에 따르면 컴퓨터 성능은 매 18 개월마다 두 배로 향상된다고 합니다. PC 의 경우 Intel 8088, 4.77MHz, 최대 메모리 640KB 에서 출발하여 현재 복수(최근의 경우 4 개)의 450MHz Pentium II 프로세서와 메모리 용량이 최대 4GB 에 이르는 PC 로 발전하였으며 64 개의 프로세서와 64GB 의 메모리를 가진 컴퓨터도 있습니다. 또한 클러스터 방식으로 컴퓨터를 사용하여 다중 병렬 처리(MPP)를 제공할 수도 있습니다. 이 모든 강력한 기능으로 인해 이제 데이터베이스 응용 프로그램의 성능에 대해 걱정할 필요가 없어졌을까요?

데이터베이스 응용 프로그램은 더욱 복잡해지고 데이터의 양도 시간이 지남에 따라 증가하고 있습니다. Informix 의 경우 이미 테라바이트 단위의 데이터베이스를 처리하고 있습니다. 그렇다면 페타바이트(1000 TB)의 데이터베이스가 등장할 날은 언제쯤일까요? 사실은 이미 존재하고 있습니다. 결과적으로 데이터베이스의 처리는 보다 복잡해지고 있으며 사용자의 수 또한 증가하고 있습니다.

이 글에서는 성능에 영향을 미치는 요소를 개괄적으로 살펴보고 응용 프로그램 개발과 조정에 도움이 될 지침을 제공하고자 합니다. 여기에는 하드웨어, 운영 체제, 데이터베이스 엔진, 응용 프로그램 개발 및 데이터베이스 액세스가 포함됩니다.

하드웨어 구성 요소

하드웨어는 성능을 좌우하는 첫 번째 요소입니다. 하드웨어 이외의 모든 요소들이 일정하다고 할 때 하드웨어를 교체함으로써 조금이라도 성능을 향상할 수 있습니다. 어떤 경우에는 하드웨어가 가장 비용이 적게 드는 해결책이 될 수도 있습니다. 그렇다면 어떤 하드웨어를 추가해야 할까요? 여기서는 Sun Enterprise Server 10000 을 예로 들어 시스템의 주요 하드웨어 구성 요소를 살펴보겠습니다.

- CPU: CPU는 프로그램의 명령을 실행하는 중앙 처리 장치입니다. 80년대 후반 이후 Unix 업체들은 SMP(대칭형 다중 처리)를 지원해 왔습니다. 다수의 프로세서를 지원할 수 있도록 시스템의 확장성이 증가되었습니다. SMP는 기능 면에서 동일한 여러 개의 CPU를 포함하고 있는 컴퓨터를 의미합니다. 각각의 CPU는 인터럽트와 입출력을 비롯하여 시스템의 어떤 작업도 처리할 수 있습니다.

고성능 CPU 는 칩내장(on-chip) 메모리를 포함하고 있으며 종종 보조 캐시를 사용합니다. 예를 들어, Sun ES10000 의 UltraSPARC 프로세서에는 16KB 의 명령 캐시와 16KB 의 데이터 캐시 외에도 4MB 의 보조 캐시가 들어 있습니다.

왜 이렇게 많은 캐시를 포함하고 있을까요? Sun ES10000 을 다시 한 번 생각해 봅시다. 프로세서 속도는 336MHz 로 표시되어 있으며 이것은 128 비트(16 바이트)의 인터페이스를 사용합니다. 각 CPU 사이클에 대해 16 바이트를 가져올 수 있습니다.

336 x 1000000 cycles/sec x 16 bytes/cycle =

5376000000 bytes/sec @ 5.2GB/sec

따라서 성능이 최고 수준에 올랐을 때 하나의 UltraSPARC 프로세서는 초당 5.2GB 로 처리할 수 있습니다. 메모리 캐시는 CPU 가 사용 중인 상태에서도 명령과 데이터에 더 빠르게 액세스할 수 있습니다.

- 메모리: 메모리 하위 시스템은 인터리브의 개념을 사용하여 데이터 전송 성능을 증가시킵니다. Sun ES10000의 경우 메모리 모듈은 초당 1.3GB라는 전송 속도를 보여줍니다.
- 시스템 버스: 시스템 버스는 입출력 하위 시스템의 중요한 구성 요소로서 입출력 제어기와 시스템 사이의 통신을 가능하게 해줍니다. 표준 Sun SBus는 초당 100MB의 속도로 데이터를 전송합니다.
- 제어기: 여러 유형의 제어기를 사용할 수 있습니다. 제어기는 디스크 드라이브, 네트워크, 테이프 드라이브 등에 액세스하는 데 사용됩니다. 표준 SCSI 제어기의 경우 성능이 초당 20MB인 SCSI 버스를 수용할 수 있습니다. Sun은 광섬유 기술에 기반을 둔 디스크 어레이(disk array)용 제어기를 출시하였는데 그 속도는 초당 100MB라고 합니다. 이것은 시스템 버스의 속도와 일치하는 것입니다.
- 디스크: 고성능 시스템에서 사용되는 표준 디스크 드라이브는 SCSI 드라이브입니다. 사용자는 용량, 회전 속도, 대기 시간, 탐색 시간, 트랙간 탐색 등을 관찰하여 디스크의 전송 속도를 계산할 수 있습니다. 이러한 모든 요소와 요청한 전송의 크기가 실제 처리량을 결정합니다. 예를 들어, 디스크의 여기 저기에 분산되어 있는 몇 개의 디스크 블록을 전송해 줄 것을 요구하는 경우 탐색 시간과 대기 시간이 매우 길어질 것입니다. 그러나 인접한 디스크 블록 여러 개를 전송하는 경우에는 대부분

의 시간을 데이터를 전송하는데 사용하게 될 것입니다.

평균 탐색 시간, 평균 대기 시간 및 트랙간 탐색 시간은 밀리세컨드(ms)로 측정됩니다. 최고 수준의 라인 드라이브의 경우 평균 탐색 속도는 5.4 밀리세컨드입니다. 이 시간 동안, UltraSPARC 프로세서는 약 7 백만 개의 명령을 실행할 수 있습니다. 이러한 오버헤드의 영향을 줄이기 위해 대부분의 디스크 제조회사는 디스크 드라이브 안에 메모리 캐시를 포함시키고 있습니다.앞에서는 연속적인 공간을 할당하고 디스크 조각화를 피하도록 권하는 이유를 설명하였습니다. 그렇다면 디스크 드라이브에서 기대할 수 있는 전송 속도는 어느 정도나 될까요?

일부 디스크 드라이브의 최고 전송 속도는 초당 80MB 에 이릅니다. 물론 이것은 "최상의 경우"를 가정했을 때의 일입니다. 성능은 입출력의 유형과 대기 시간 및 탐색 시간의 영향에 의해 매우 달라질 수 있습니다. "최악의 경우"를 가정해 보면, 각 페이지에 대해 평균 탐색 시간과 평균 대기 시간이 필요하다면 초당 240KB (매 8.4 밀리세컨드 당 2KB)의 수준으로 성능이 떨어집니다. 이처럼 상태에 따라 큰 차이가 납니다. 실제 성능은 응용 프로그램을 벤치마크하여 계산할 수 있습니다.

Optional Cache	————	4096KB
Track-toTrack Seek	————	0.8/1.1msec
Avg Seek, Read/Write	————	5.4/6.2msec
Max Seek, Read/Write	————	12.2/13.2msec
Average Latency	————	2.99msec
Spindle Speed	————	10025 RPM

그림 1: 일부 디스크 드라이브의 특성

[1] http://www.seagate.com/disc/cheetah.shtml에서 발췌한 자료

- 네트워크: 가장 널리 사용되는 네트워크는 이더넷(ethernet)인데 그 전송 속도는 초당 10Mbit 정도입니다. 오류 발견을 위해 사용되는 제어 비트 및 데이터의 표준 오버헤드를 고려할 때 기대할 수 있는 최고 속도는 초당 1MB 정도입니다. 그러나 새로운 네트워크 기술이 빠르게 출현하고 있으며

초당 100Mbit(초당 10MB)의 전송 속도를 제공하고 있습니다.

이더넷 프로토콜은 정보 전송에 필요한 형식을 정의합니다. 정보의 각 패킷의 크기는 최대 1526 바이트입니다. 여기에는 26 바이트의 제어 정보[2]가 포함됩니다. 전송되는 데이터의 크기는 효과적인 전송 속도에 영향을 미치는 주요 요소입니다. 예를 들어 20 바이트의 데이터를 전송하는 경우에도 여전히 오버헤드가 50 퍼센트 이상인 26 바이트의 제어 데이터를 함께 전송해야 합니다.

[2]Douglas Comer, Internetworking with TCP/IP, 18-19 쪽.

- 모뎀: 모뎀의 속도는 다양하며 그 중에 가장 일반적인 모뎀 속도는 초당 14.4 Kbits, 초당 28.8Kbits, 초당 56Kbits 등입니다. 지금 이 글에서는 초당 MB의 단위를 사용하고 있으므로 단위를 바꿔 보면 각각 초당0.0014.MB, 초당 0.0028MB 비트, 초당 0.0056MB가 됩니다.
- 대화형 사용자: 드디어 사용자에 대해 언급할 차례가 되었습니다. 사용자는 응용 프로그램에 따라 상대적으로 높은 수신 속도를 얻을 수 있긴 하지만 일반적으로 짧은 순간으로 제한됩니다. 그러나 전송 속도는 대개 매우 느립니다. 인간은 생각하고 명령을 내리고 결과를 분석합니다. 일분에 영문 120 단어를 입력하는 점원이 있다고 가정해 봅시다. 단어 당 평균 글자 수가 8자라고 가정하면, 분당 960 바이트 또는 초당 16바이트의 속도를 내게 되며 이것은 초당 0.000016MB의 속도로 전송됩니다.

여기에서는 시스템의 성능을 어느 정도 일반화할 수 있도록 해주는 기본적인 숫자들을 사용하였습니다. 그러나 구성 요소 간의 상호 작용에 대해서는 전혀 고려하지 않았으므로 이러한 숫자들은 주의해서 사용해야 합니다.
예를 들어, Sun ES10000 에는 UltraSPARC 부속 보드(daughterboard)를 16 개까지 장착할 수 있습니다. 각 보드에는 프로세서 네 개, 메모리 모듈 한 개 또는 두 개, SBus 두 개 그리고 장치들이 연결된 다수의 제어기들이 포함되어 있습니다. 또한 각 부속 보드에서의 모든 상호 작용과 부속 보드간의 통신 및 기계에서 수행되는 프로세싱의 성격 또한 고려해야 합니다.

오늘날의 복잡한 컴퓨터에 대한 언급은 이쯤에서 마무리하고, 이제 데이터베이스 환경을 도울 수 있는 몇 가지 간단한 지침을 살펴보겠습니다. 일부 제어할 수 없는 것도 있지만 그러한 것들도 여전히 염두에 두어야 합니다.

하드웨어: 전반적인 고려 사항

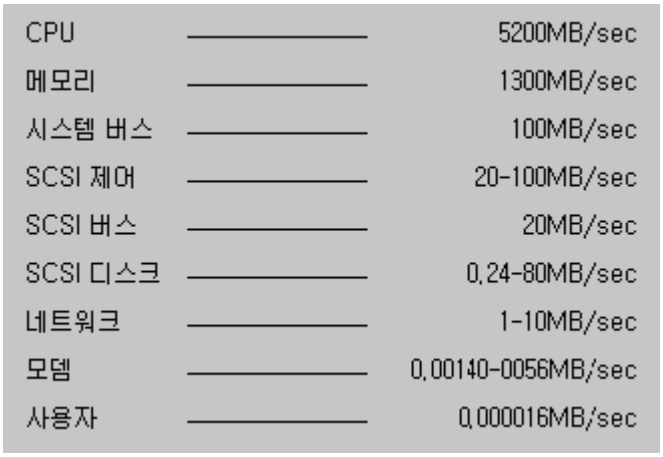


그림 2: 시스템 구성 요소의 최대 성능

그림 2 의 숫자를 보면 시스템이 제대로 균형을 이루지 못하고 있다고 느낄 수도 있을 것입니다. 수 많은 벤치마크를 통해 시스템이 잘 균형 잡혀 있음을 보여 주었는데, 이는 병목 현상을 일으키는 구성 요소가 없다는 것을 뜻합니다. 다음과 같은 몇 가지 간단한 측면에서 이것을 설명할 수 있습니다.

- CPU에는 캐시가 있어, CPU에서 아직 캐시에 저장되지 않은 것을 요청하기 전에 몇 번이고 재사용할 수 있는 데이터와 명령을 포함하고 있습니다.
- 성능의 균형을 맞추기 위해 구성 요소들을 추가할 수 있습니다. 처리 환경에 따라 CPU, 메모리, 제어기 및 디스크 드라이브의 최적 비율을 산출할 수 있습니다.
- 디스크 드라이브에는 대기 시간 및 탐색 시간의 영향을 줄여 줄 수 있는 로컬 캐시가 들어 있습니다. 또한 작성된 데이터를 다시 읽는 경우에는 드라이브에 액세스하지 않아도 캐시에서 해당 데이터를 제공합니다.

그림 2 의 숫자에서 다음과 같은 지침을 생각할 수 있습니다.

- CPU를 사용 중인 상태로 유지합니다.

  CPU 는 모든 처리의 핵심입니다. CPU 가 유휴 상태에 있게 되면 처리 사이클을 잃게 됩니다. 캐시가 필요한 데이터를 가지고 있지 않는 경우에 CPU 는 유휴 상태가 되며 메모리가 적절한

정보를 포함하고 있지 않는 경우에 캐시는 대기 상태가 됩니다. 또한 메모리는 입출력이 완료되기를 기다리게 됩니다. 이 글 후반부에서 이러한 것에 영향을 미치는 방법에 대해 살펴보겠습니다.

- 입출력의 균형을 맞춥니다.

  가장 흔히 볼 수 있는 성능상의 문제점은 입출력의 많은 부분이 한 디스크 드라이브로 집중되는 반면 다른 디스크들은 유휴 상태에 있게 되는 것입니다. 이러한 문제의 변형으로 모든 입출력이 동일한 제어기를 통과하는 경우를 들 수 있습니다. 이는 데이터베이스 테이블이 적절하게 분산되지 않았거나 모든 시스템 입출력이 한 드라이브에서 수행되는 경우에 발생합니다. 데이터베이스에서 이러한 핫스폿을 해결하려면 입출력을 분석하여 테이블을 적절히 분산시켜야 합니다. 여기에는 논리적 로그를 다른 DB 영역으로 이동하는 작업도 포함됩니다. 핫스폿이 한 테이블 내에서 발생하는 경우에는 Informix 테이블 분할 기능을 사용하여 테이블이 여러 디스크 드라이브에 나누어지도록 할 수 있습니다. 또한 하드웨어 디스크 스트라이핑(hardward disk striping)을 사용하여 분할 기능을 향상시킬 수 있습니다. 시스템 측면에서 볼 때는 시스템 디스크에서 핫스폿이 가장 많이 발생합니다. 스왑 영역과 임시 디렉토리를 다른 드라이브로 옮기는 것이 좋습니다.

- 디스크 조각화를 피합니다.

  앞에서 입출력의 조각화로 인해 디스크 처리 능력에 엄청난 악영향이 미치는 것을 보았습니다. 큰 청크들을 할당하여 테이블이 연속적으로 놓이도록 하십시오.

- 네트워크 트래픽을 제한합니다.

  네트워크는 상대적으로 대역폭이 좁으므로 트래픽이 증가하게 되면 충돌이 발생하는데, 이 충돌은 재전송을 유발하게 되며 결과적으로 네트워크 통신에 더 많은 오버헤드를 추가하게 됩니다. 응용 프로그램을 개발할 때에는 이 점을 꼭 명심하십시오. 네트워크는 중요한 리소스입니다. 따라서 데이터베이스 서버가 계산을 수행할 수 있을 때에는 대규모의 데이터를 전송하지 말고 단지 작은 결과만을 반환하도록 하십시오.

이러한 일반적인 지침은 계산 환경의 수준에 따라 달라질 수 있습니다. 결론은 언제나 시스템 작업의 균형을 유지해야 한다는 것입니다.

운영 체제

운영 체제의 주요 목적은 하드웨어 리소스의 사용을 최적화하는 것입니다. 이 말은 간단해 보이지만 매우 복잡한 결정을 요구합니다. 다시 Sun ES10000 을 예로 들어 보겠습니다. 완벽하게 구성된 기계에는 상호 교차 연결을 통해 서로 통신하는 16 개의 시스템 보드가 있습니다. 그리고 각 시스템 보드에는 보조 캐시, 최대 4GB 메모리, 두 개의 SBus 및 다수의 제어기가 장착된 UltraSPARC CPU 네 개가 있습니다.

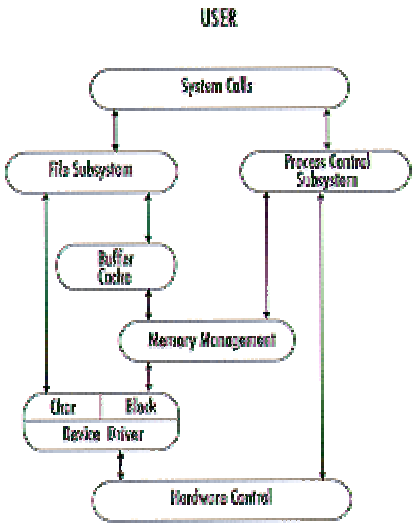


그림 3: OS Kernel[3]의 블록 도표.

[3]Maurice J. Bach의 The Design of the UNIX Operating System 개정판 20 쪽

그림 3 은 운영 체제의 주요 구성 요소를 보여줍니다. 사용자는 프로세스라고 하는 실행 프로그램입니다. 이것은 시스템 호출을 통해 시스템과 상호 작용합니다. 예를 들어, Solaris 운영 체제는 약 185 개의 시스템 호출을 제공합니다. 시스템 호출을 통해 프로세스는 장치, 파일 시스템을 액세스할 수 있으며 다른 로컬 또는 원격 프로세스와 통신할 수 있습니다.

운영 체제는 메모리의 사용을 최적화하여 가능한 한 많은 프로세스를 수용해야 합니다. 운영 체제와 그것의 시스템 프로세스, 장치 드라이브, 파일 시스템, 사용자 프로세스 등에서 메모리를 사용합니다.

시스템의 주요 구성 요소를 전반적으로 살펴보면 다음과 같습니다.

- 프로세스

  프로세스란 메모리를 연속적으로 어드레스할 수 있는 공간으로 여기는 실행 프로그램입니다. 32 비트 시스템에서 프로세스는 4GB 의 메모리를 어드레스할 수 있습니다. 운영 체제는 프로세스 주소 공간을 페이지로 분리합니다. 프로세스는 언제가 한 번은 실행에 필요한 최소한의 페이지를 요구하는데, 이를 일명 작업 세트 크기(working set size)라고 합니다.

  프로세스는 실행, 실행 준비 또는 이벤트 대기(휴면)와 같은 여러 상태의 실행을 거치게 됩니다. 이렇게 함으로써 운영 체제에서 실행 프로세스의 일정 관리 방법을 결정할 수 있습니다.

- 스레드(Thread)

  오늘날에는 UNIX 를 구현할 경우 대부분 프로세스 내에 “실행 스레드”라는 개념이 포함되어 있습니다. 가장 간단한 모델은 한 프로세스에 실행 스레드 한 개가 포함되어 있는 경우입니다. 한 프로세스 내에서 임의의 개수의 스레드를 시작할 수도 있습니다. 원시(native) 스레드 구현을 이용하는 경우 단일 프로세스를 여러 CPU 상에서 동시에 실행할 수 있습니다.

  스레드는 프로세스의 과정에서 실행되므로 보다 적은 리소스를 사용하며 스레드 간의 통신도 보다 간단합니다. 또한 스레드를 생성하는 것이 프로세스를 생성하는 것보다 비용이 훨씬 덜 듭니다.

- 프로세스 제어 및 일정 관리

  운영 체제는 일정 시간이 경과한 후에, 시스템 인터럽트가 발생했을 때, 혹은 현재 실행 중인 프로세스에서 시스템 호출을 실행하였을 때 프로세스가 실행되도록 일정을 작성할 수 있습니다. 프로세스가 여전히 실행 대기 중이면 스케줄러에서는 실행을 계속할지 혹은 또 다른 프로세스로 대신할지를 결정합니다.

  큰 시스템에서 프로세스 스케줄러는 어디에서 프로세스가 마지막으로 실행될지를 반드시 고려해야 합니다. 이렇게 하여 프로세스 메모리 페이지들이 프로세서 캐시에서 이용 가능한 상태가 되도록

할 수 있습니다. 그러면 프로세서의 생산성을 바로 높일 수 있습니다.

- 메모리 관리

  물리적 메모리에 대한 요구량이 너무 많은 경우에 운영 체제는 프로세스의 모든 작업 세트를 통합하여 디스크로 스왑하여 필요한 메모리를 수용하도록 합니다. 메모리 요구량이 너무 많아 운영 체제가 시스템을 관리하는 데에 CPU 시간의 많은 부분을 소비하게 되는 경우가 발생할 수도 있습니다. 이것은 종종 시스템 관리 도구들에서 시스템 시간을 높은 백분율로 보고하는 것으로 반영됩니다.

- 파일 시스템

  대부분의 UNIX 파일 시스템은 BSD Fast File System 에 그 기원을 두고 있습니다. 이 파일 시스템 구조는 원래의 UNIX 파일 시스템에 비해 몇 가지 개선된 점이 있습니다. 이러한 개선된 사항에는 블록 크기가 증가한 것과 파일 설명자(inode)와 그것이 설명하고 있는 데이터에 근접하여 실린더 그룹을 작성할 수 있다는 것이 포함됩니다.

  블록 크기를 원래의 512 바이트에서 두 배로 증가시키는 단순한 작업을 통해 성능이 두 배로 증가되었습니다. 현재 파일 시스템은 종종 8KB 를 기본 블록 크기로 사용하며 일부 UNIX 업체는 블록 크기를 조정할 수 있는 파일 시스템을 제공합니다.

  이 글의 맨 마지막에 있는 참고 문헌들에서 파일 시스템에 관한 자세한 정보를 얻을 수 있습니다. 특별히 관심을 끄는 두 가지 내부 구조가 있는데, inode 와 디렉토리가 그것입니다. inode 는 특정 파일을 설명하며 소유자, 그룹, 파일 유형, 액세스 허가 여부, 액세스 시간, 링크의 개수 및 파일 크기에 관한 정보를 갖고 있습니다. 또한 데이터의 위치를 지정하는 15 개 포인터 세트를 가지고 있습니다. 그림 4 는 inode 의 구조를 데이터 포인터와 함께 설명하고 있습니다. 처음 12 개의 데이터 포인터는 직접 데이터 블록을 참조하는데 이는 그림 4 에서 “Direct0”에서 “Direct11”까지 지정되어 있습니다. 후에 나오는 3 가지 포인터 PTR1, PTR2, PTR3 은 색인 블록에 대한 포인터입니다. 각각의 포인터는 간접 레벨을 추가합니다. PTR3 포인터는 색인 블록에 대한 포인터를 포함하는 색인 블록을 가리킵니다. 그림 4 에서 보여주듯이 PTR3 은 색인 블록의 3 개의

레벨을 가리킵니다.

파일을 읽기 위해서는 운영 체제에 inode, 요청한 색인 블록, 그리고 최종적으로 데이터 블록을 가져야 합니다. 처음으로 파일을 읽는 경우, inode 를 얻기 위해 디스크를 한번 액세스해야 하고, 색인 블록을 얻기 위한 또 한번의 디스크 액세스가 필요하며, 필요한 데이터 블록을 얻기 위한 또 한번의 디스크 액세스를 해야 할 것입니다. 파일 시스템은 메모리에 디스크 블록을 캐싱하는 것으로 디스크 액세스를 최적화합니다. 프로세스가 자주 액세스하는 파일로부터 데이터를 요구하는 경우, 데이터 캐시는 디스크 입출력 없이도 데이터를 반환할 수 있습니다.

Inode 는 파일을 설명하긴 하지만 그 설명에 이름이 포함되어 있지는 않습니다. 디렉토리 파일은 이름과 inode 사이의 맵핑을 제공합니다. 이러한 체계는 여러 이름들이 하나의 inode 를 참조하는 것을 가능하게 합니다. 디렉토리는 또한 파일 시스템의 계층구조를 제공합니다.

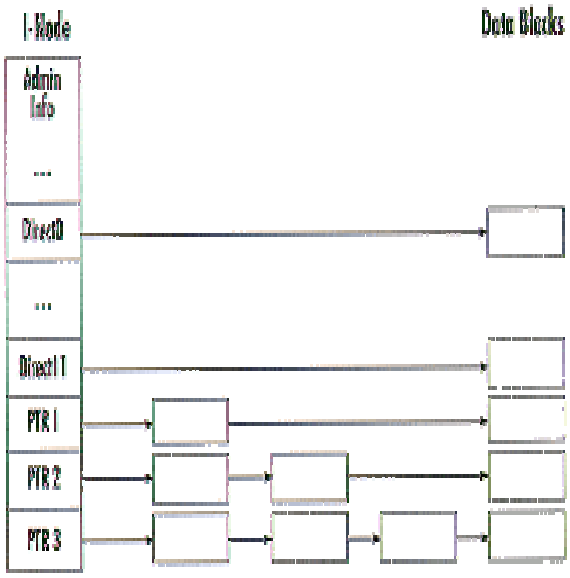


그림 4: I-node 데이터 포인터

그림 5 는 디렉토리의 형식을 보여주고 있습니다. 그 형식은 4 바이트 단위로 맞추어 정렬되며 각 엔트리들은 가변적인 길이를 갖습니다. 파일을 찾기 위하여 순차적인 탐색이 실행됩니다. 탐색의 평균 길이는 엔트리 수와 함께 증가합니다.

그림 5: 디렉토리 엔트리

- *네트워크 액세스*

UNIX 시스템에서는 라이브러리 기능을 통하여 네트워크에 액세스할 수 있습니다. 두 개의 표준 라이브러리로 소켓과 TLI(Transport Layer Interface)가 있습니다. 사용자 프로그램에서는 네트워크 라이브러리를 호출함으로써 네트워크 연결을 설정할 수 있으며, 이는 곧 시스템 호출을 발생시킵니다. 일단 네트워크 연결이 설정되면, 라이브러리 호출 또는 시스템 호출을 사용하여 네트워크를 통해 데이터를 전송할 수 있습니다.

TCP/IP 연결을 생각해 봅시다. TCP 프로토콜(전송 제어 프로토콜)에서는 전송을 위한 데이터를 준비하고 그 데이터를 IP 레이어로 전달하기 전에 24 바이트의 제어 정보를 추가합니다. IP(인터넷 프로토콜) 또한 여기에 24 바이트의 제어 정보를 추가하고 이 데이터를 물리적인 레이어로 전달합니다. 물리적인 레이어에서는 받아들인 데이터를 32 바이트의 제어 정보를 포함하는 패킷으로 나눠야 합니다.

만약 TCP 패킷이 물리적인 틀에 꼭 들어 맞는다면, 통신상에서 총 80 바이트의 오버헤드가 발생합니다. 그렇지 않으면, TCP, 좀더 정확하게 말하면 IP 패킷은 여러 개의 물리적인 패킷으로 나누어집니다. 물리적인 패킷의 최대 크기는 사용되는 네트워크 인터페이스에 따라 다릅니다. 이더넷 형식은 제어 정보를 포함하여 최대 크기를 1526 바이트로 정의하고 있습니다. FDDI 형식은 최대 크기를 4500 바이트로 정의하고 있습니다.

그림 6: TCP/IP 레이어 모델

- *시스템 호출*

  프로세스들은 시스템 호출을 통하여 운영 체제로부터 서비스를 요구합니다. 이러한 방법은 사용자에게 부가적인 사용 권한을 효율적으로 부여함으로써 누구나 시스템 리소스에 액세스할 수 있도록 합니다. 모든 리소스가 수 많은 사용자들 사이에 공유되기 때문에, 시스템 호출이 적절하게 실행될 수 있도록 보증할 수 있는 부가적인 점검이 반드시 필요합니다. 만약 그렇지 않다면 모든 사용자들이 혼란에 빠질 수 있습니다.

  시스템 호출을 실행하는 것은 상대적으로 비용이 많이 드는 작업이므로, 시스템 호출이 실행될 때 어떤 일들이 일어나야 하는지 고려해야 합니다.

  - 하드웨어가 커널 모드로 전환됩니다.
  - 프로세스 실행 과정이 저장됩니다. 여기에는 프로그램 카운터, 프로세서 상태, 프로세서 레지스터 등이 포함됩니다.
  - 매개변수의 수가 검증됩니다.
  - 매개변수의 위치를 점검되고 매개변수가 커널 공간으로 복사됩니다.
  - 시스템은 시스템 호출 중단 사태에 대비합니다.
  - 시스템 호출 작업이 실행됩니다.
  - 반환 값이 적절한 위치에 복사됩니다.
  - 프로세서는 사용자 프로세스 실행 과정을 복구합니다.
  - 하드웨어가 사용자 모드로 다시 전환됩니다.

어떤 시스템 호출은 다른 것들에 비하여 더 많은 것들을 처리합니다. 다음 문장을 참조하시기 바랍니다.

```
fd = open("/usr/Informix/online.log". O_RDONLY):
```

이 시스템 호출은 파일 경로명을 분석하여 “/” 을 찾습니다. 그런 다음 디렉토리를 열고 “usr”에 대한 엔트리를 찾습니다. “usr” 디렉토리가 열린 다음 “Informix”를 열어서 검색합니다. 마지막으로 “online.log”에 대하여 inode 번호를 찾게 됩니다. 이러한 모든 작업에는 디스크 입출력과 버퍼 캐시 작업이

포함됩니다.

운영 체제: 전반적인 고려 사항

하드웨어에서 고려했던 것과 마찬가지로 운영 체제에서도 같은 사항들을 고려해야 합니다. 운영 체제는 프로세스가 어떤 이벤트를 기다려야 할 때마다 CPU를 다른 프로세스에 넘겨줍니다. 이러한 방법으로 운영 체제는 CPU를 가능한 한 바쁘게 유지되도록 합니다. 스케줄링 방법이 실행 환경과 더 잘 조화를 이루도록 하기 위해 운영 체제를 어느 정도 조정할 수 있습니다. 예를 들어, 데이터베이스 서버로 지정한 컴퓨터는 대화형 사용자용으로 설정되어서는 안됩니다.

파일 시스템은 디스크 정보를 캐싱함으로써 성능을 최적화하고자 합니다. 또한 파일 시스템은 비교적 큰 입출력 크기를 사용하고 있습니다. 이는 하드웨어 부분에서 언급한 것과 일맥 상통합니다.

운영 체제 부분에서 찾아 낸 정보로부터, 다음과 같은 지침을 세울 수 있습니다.

- 시스템 호출의 사용을 제한합니다.

  앞에서 보았듯이 시스템 호출은 실행에 앞서 몇 가지의 작업을 요구니다. 시스템 호출 사용을 제한하는 쉬운 방법이 있습니다. 예를 들면, 시스템 호출을 루프에서 끄집어 내어도 여전히 같은 기능을 제공할 수 있습니다. 다음 코드를 살펴보십시오.

```
while (0 < read(fd, buf, 1) ) {
```

```
}
```

파일을 여는 함수에 대한 정의를 변경하고 그 대신에 파일 설명자를 전달할 수 있습니다. 어떤 상태를 점검하기 위하여 시스템 호출을 반하는 대신에, 특정 이벤트를 차단하는 것이 나을 것입니다. 마지막으로, 어떤 작업은 항상 시스템 호출을 실행하지 않고도 사용자 공간에서 행해질 수 있습니다. 그런 경우에는 상호 연관된 프로세스들을 사용하는 대신에 스레드를 사용하는 편이 훨씬 이로울 것입니다.

- 코드와 데이터를 조정합니다.

  프로세스를 실행하기 위해서는 어느 정도의 메모리가 필요합니다. 이 메모리에는 명령 페이지와 데이터 페이지가 포함됩니다. 프로세스에 필요한 최소 메모리 양을 작업 세트 크기(working set size)라고 합니다. 종종 함께 사용되는 함수들은 동일한 페이지가 아니라면 연속된 페이지에라도 두어 서로 가까이 있게 해야 합니다. 이러한 사항은 데이터 페이지에도 해당됩니다.

- 입출력 작업에 버퍼를 사용합니다.

  파일 시스템에서 힌트를 얻어 입출력 작업에도 버퍼를 사용하도록 합니다. 한번에 한 문자씩 처리하는 대신에 어떤 규모가 될 때까지 혹은 특정 조건을 만족할 때까지 데이터를 모은 후 입출력을 실행합니다. 이러한 것들은 “C” 언어의 fputc() 함수와 같은 프로그래밍 라이브러리를 사용하여 자동적으로 수행할 수 있습니다.

- 네트워크의 활동을 제한합니다.

  네트워크가 점점 빨라지고 있긴 하지만 다른 요소들도 역시 빠르게 발전하고 있기 때문에 개념적으로 볼 때 성능상 별로 변화가 없을 수도 있습니다. 네트워크상의 사용자 수는 경쟁 문제를 악화시킵니다. 큰 조직의 경우, 사용자당 사용량이 조금만 달라져도 큰 결과를 가져올 수 있습니다. 1000 명의 사용자로 이미 포화 상태에 이른 네트워크에서 네트워크 사용량이 10% 줄어든다면, 이것은 100 명의 사용자가 추가로 네트워크를 사용할 수 있음을 의미합니다.

  만약 성능상에 어떤 문제가 있다면, 먼저 어디에서 병목 현상이 발생하는지를 찾아 내고, 그 상황을 검토하여 어떤 하드웨어를 어디에 추가할 것인지를 결정하거나 혹은 소프트웨어로 구현하는 방법을 검토해야 합니다.

  또한 운영 체제에서 제공하는 기능들, 특히 조정 매개변수들, 스케줄링, 스레드 지원 등에 대해서

질의 처리

질의 실행은 다음과 같은 여러 요소의 영향을 받습니다.

- 데이터 분할(Partitioning): 병렬 처리는 데이터 분할에 의해 좌우됩니다. 엔진은 실행과 관련된 DB 영역 수량 만큼의 스캔 스레드를 시작합니다. 이 스캔 결과는 익스체인지를 통해 조인, 그룹, 정렬 및 병합 등을 실행하는 다른 스레드로 전달됩니다.

- 색인(Indexes): 색인을 사용하면 질의를 실행하는 데 필요한 입출력을 줄일 수 있습니다. 기본 색인 체계는 b+ 트리를 기본으로 합니다.

- 통계(Statistics): Informix Dynamic Server에서는 무엇보다도 데이터의 양과 테이블에서의 데이터 분포에 대한 통계를 유지합니다. 통계는 최상의 질의 계획을 세우기 위한 가장 중요한 요소입니다.

- 옵티마이저(Optimizer): 옵티마이저에서는 질의 처리를 위해 가능한 모든 방법을 찾는 철저한 검색 알고리즘을 사용합니다. 질의를 최적화를 하는 동안, 옵티마이저에서는 질의를 재작성할 수도 있습니다. 이렇게 재작성된 질의는 질의 원본에서 분명하지 않았던 질의 계획을 생성할 수도 있습니다. 각 계획의 비용은 각 연산의 비용과 처리할 데이터의 추정량을 파악하여 수립할 수 있습니다.

Informix Dynamic Server: 전반적인 고려 사항

앞서 확인했듯이, Informix Dynamic Server 에서는 CPU 사용량을 최적화하고 입출력 처리를 줄이려고 노력합니다. 이를 위해 CPU VP, 메모리 할당, 네트워크 연결 등을 조정할 수 있는 몇몇 매개변수를 사용할 수 있습니다. 이 밖에도, 데이터베이스 관리자는 응용 프로그램 시스템 설계자의 도움을 받아 성능을 향상시키기 위해 여러 가지 일들을 해볼 수 있습니다.

DB 영역에서 필요한 만큼 테이블을 분할해야 합니다. 또한, 응용 프로그램에 가장 큰 영향을 주는 질의를 분석하여 분할화 전략을 결정해야 합니다.

색인은 성능에 도움이 될 수 있지만 색인이 너무 많으면 삽입이나 업데이트 성능을 떨어지게 할 수 있습니다. 어떤 색인들은 분명히 필요한 것들입니다. 그러나 다른 색인들의 경우, 질의 계획을 검토하여

그것들이 정말로 사용된 적이 있는지의 여부를 확인해야 합니다. 질의 계획에 테이블 스캔이 있으면, 처리 속도를 향상시키는 색인을 생성할 수 있습니다. 따라서, 테이블 스캔이 반드시 불필요한 것은 아닙니다.

PHYSFILE	LOGFILES	LOGSIZE
NETTYPE	RESIDENT	NUMCPUVPS
MULTIPROCESSOR	_____	AFF_SPROC
SINGLE_CPU_VP	_____	AFF_NPROCS
LOCKS	BUFFERS	NUMAOIVPS
SHMVIRTSIZE	_____	SHMADD

표 1: 일부 onconfig 서버 매개변수.

질의를 처리하기 위한 최선의 방법을 찾고자 할 때, 옵티마이저에게 가장 중요한 사항은 업데이트된 통계를 얻는 것입니다. 통계를 얻기 위한 작업은 상당한 양의 행이 추가되었거나 수정되었을 때, 그리고 색인이 추가된 뒤에 이루어져야 합니다. Informix 에서는 다음과 같은 UPDATE STATISTICS 방법을 권장합니다.

- 중간 분포(Medium distribution only): 각 테이블이나 전체 데이터베이스에 대해 실행.
- 높음(High): 색인이 포함된 모든 열에 대해 실행
- 낮음(Low): 다중 열 색인의 모든 열에 대해 실행.

응용 프로그램 개발

응용 프로그램은 더욱 더 복잡해지고 있습니다. 오늘날에는 2-티어, 3-티어, N-티어 응용 프로그램, 다중 스레드 처리, 구성 요소 아키텍처(CORBA와 DCOM 포함) 등에서 선택을 해야 합니다. 이런 결정은 응용 프로그램 성능에 큰 영향을 줍니다. 이런 사항들은 여기에서 모두 다루기에는 너무 광범위하므로 여기에서는 응용 프로그램 성능을 향상시키는 데 도움이 될 수 있는 기본적인 개념만을 설명하고자 합니다.

알고리즘

문제를 해결하기 위해 어떤 방법을 선택하느냐가 성능에 가장 큰 영향을 줍니다. 데이터 유형을 표현하는 방법과 정렬 및 검색 실행 방법 등 알고리즘을 설명하는 많은 서적들이 있습니다. 무엇보다도 문제를 정확히 정의하는 것이 올바른 알고리즘을 찾기 위한 첫걸음입니다.

참고 서적 목록에서 제시한 “Programming Pearls”에 알고리즘 응용과 문제를 찾는 방법에 대한 좋은 예제가 많이 포함되어 있습니다. 이 책의 제 5 장에서 저자인 Jon Bentley 는 성능을 400 배나 개선할 수 있는 여러 방법에 대한 연구 사례를 제공합니다.

물론, 항상 극적인 성능 향상을 기대할 수는 없습니다. 중요한 것은 여러 가지 문제 해결 방법을 찾아 보고 그것들을 평가하여 가장 좋은 것을 수행하는 것입니다. Universal Data Option 과 함께 Informix Dynamic Server 를 사용할 때 새로운 많은 가능성이 열립니다. 서버에 업무 로직을 추가함으로써 데이터베이스 엔진의 고급 알고리즘 및 병렬 처리를 사용할 수 있습니다.

이것은 적당한 비교 연산자나 날짜 계산 함수를 제공하는 것만큼이나 간단할 수 있습니다. 그룹화를 하고, 응용 프로그램으로 모든 정보를 가져와 합계를 계산하는 대신 서버에서 합계를 계산하도록 하는 함수들을 제공할 수 있습니다. Universal Data Option 기능 사용 방법에 대한 자세한 사항은 참고 서적 목록에 제시된 “Best Practices”에서 찾을 수 있습니다.

컴파일러 사용법

때로는 가장 간단한 것을 쉽게 잊는 수가 있습니다. 제품 생산용으로 컴파일해야 할 코드를 디버깅을 위한 것으로 잘못 컴파일하는 경우도 있을 것입니다. 이런 일은 개발자가 컴파일러 옵션을 바꾸는 일을 잊어버려 생길 수 있습니다. “-O” 대신에 “-g”를 사용하는 것과 같은 아주 사소한 데에서 문제가 발생합니다.

컴파일 옵션은 어떤 영향을 줄까요? 컴파일 옵션은 전체 명령의 수와 함수 호출을 위해 준비해야 할 명령의 수에 영향을 미칩니다. 다음은 컴파일 옵션에 대한 간단한 예제 프로그램입니다.

```
#include
```

```
int fn1(int i) {
int j, k;
k = 0;

      for (j = 0; j < i; j++) {
      k += j;
      }
      return(k);

}

int main(int argc, char **argv) {
int ret;

      ret = fn1(1000000);
      printf("ret = %d\n", ret);
      return(0);

}
```

이 프로그램은 예제에 불과합니다. fn1(1000000)의 결과는 오버플로우되어 적절한 결과를 제공하지 않습니다.

이 프로그램을 컴파일할 때 최적화 옵션을 적용하거나 적용하지 않을 수 있으며 프로필 옵션과 디버깅 옵션을 설정할 수 있습니다. 다음 표는 특정 플랫폼에서 컴파일한 결과로 생성된 어셈블러 파일에 관한 통계의 일부입니다. 이런 결과는 플랫폼과 사용한 컴파일러에 따라 다를 수 있습니다.

컴파일러	명령 수	객체 파일 크기	실행 파일 크기	실행 경과 시간	실행 사용자 시간	실행 시스템 시간
최적화됨	21	1232	5564	0.07 초	0.04 초	0.02 초
최적화 안됨	55	1312	5696	0.13 초	0.11 초	0.02 초
디버그됨	59	3684	8008	0.13 초	0.12 초	0.01 초
디버그 및 최적화됨	26	3488	7680	0.07 초	0.04 초	0.02 초
프로필됨	51	1524	10012	0.15 초	0.12 초	0.01 초
프로필 및 최적화됨	28	1404	9840	0.09 초	0.05 초	0.01 초
디버그 및 프로필됨	69	3844	12320	0.16 초	0.11 초	0.02 초
디버그, 프로필 및 최적화됨	28	3596	11960	0.08 초	0.05 초	0.01 초

표 2: 어셈블러 파일 통계.

디버깅 옵션과 최적화 옵션은 상호 배타적으로 사용되곤 합니다. 이러한 이유로 디버깅용 파일을 만들기 위해 컴파일을 할 때 메이크(make) 파일에는 최적화 옵션이 없음을 볼 수 있습니다.

대부분의 “C” 컴파일러는 어셈블러 코드를 생성할 수 있는 옵션(-S)을 제공합니다. 이 옵션은 각 컴파일러 옵션으로 생성되는 명령 수를 결정할 때 사용합니다. 생성되는 어셈블러 소스는 주석이나 어셈블러 지시어도 포함하는데 이러한 것들은 명령 수에 합산되지는 않습니다. 오브젝트 파일 크기와 실행 파일 크기는 바이트로 표시됩니다.

실행 시간은 timex(1) 함수를 사용하여 얻을 수 있습니다. 경과 시간은 실행을 완료하는 데 소비한 시간을 나타냅니다. 이 시간은 “스톱 워치(stop watch)” 방식으로 측정됩니다. 시스템 사용량이 많을 경우, 수행을 위해 대기열에서 기다린 시간도 경과 시간에 모두 포함됩니다. 사용자 시간과 시스템 시간은 사용자 공간에서 소비된 CPU 사이클과 커널 모드에서 소비된 사이클을 나타냅니다.

대부분의 실행 시간은 fn1() 함수의 루프에서 소비됩니다. 전체 실행 시간은 1 초 이내이어서 그다지 문제가 되지 않는다고 생각할 수도 있을 것입니다. 그러나, 실행간의 비율을 살펴보면 표준 컴파일에 소비된 시간이 사용자 시간(0.11 초)의 경우 최적화 컴파일(0.04 초)의 거의 3 배나 되는 것을 알 수 있습니다. 다시 말해서, 최적화된 버전이 63%나 더 빠르다는 것입니다!

프로필 처리

응용 프로그램에서 성능을 향상시키려면, 먼저 시간이 소비되는 곳을 찾아야 할 필요가 있습니다. 프로그램을 컴파일할 때 함수 호출과 각 함수에 소비된 시간에 대한 정보를 생성할 수 있습니다. 다음은 앞에서 제시한 프로그램을 사용하는 예제입니다. 먼저, 다음 프로필 옵션을 사용하여 프로그램을 컴파일 할 필요가 있습니다.

```
cc -p -o prg1 prg1.c
```

다음으로 프로그램을 정상적으로 수행하십시오. 그러면, mon.out 이라는 파일이 생성됩니다. 다음 prof 명령을 수행하면 프로필 처리 통계를 얻을 수 있습니다.

```
prof -V prg1


%Time Seconds Cumsecs #Calls msec/call Name
100.0   0.11  0.11    1      110.       fn1
0.0     0.00  0.11    1       0.       main
```

이제, fn1()에서 소비된 시간을 알 수 있으므로 이 프로그램을 최적화할 수 있는지 여부를 확인할 수 있습니다. 사실 이것은 1 에서 N 사이의 모든 수를 더하는 고전적인 문제입니다. 루프는 다음 코드로 대체할 수 있습니다.

```
i * ( ( i - 1 ) / 2 ) ;
```

이것은 fn1() 수행 시간을 무의미하게 만듭니다. 물론, 오버플로우 문제는 아직도 남아 있습니다.

다음은 성능을 향상시킬 수 있는 몇 가지 간단한 방법입니다.

- 루프를 코드 밖으로 이동시키십시오: 그리고, 루프에 필요없는 코드가 포함되지 않도록 하십시오. 일부 코드를 빼도 결과에 영향을 주지 않는 경우도 있습니다.
- 루프 병합: 같은 범위에 대해 작업을 수행하는 두 루프가 있으면, 두 루프를 함께 수행할 수 있습니다.

- 테스트 순서 변경: 복잡한 테스트는 결과를 결정할 수 있는 즉시 종료되도록 합니다. “논리합” 조건일 경우, 첫 번째 참 값이 결과를 결정합니다. “논리곱” 조건일 경우, 첫 번째 거짓 값이 결과를 결정합니다. 데이터에 대한 지식을 바탕으로 최대한 빨리 결과를 찾을 수 있도록 테스트 순서를 바꿀 수 있습니다.
- Sentinel 값 사용: Sentinel 은 조건이 결국 일치하게 되도록 하는 값입니다. 텍스트 문자열이 있는데, 이를 단어 단위로 분리해야 할 경우 다음과 같은 조건을 사용한다고 가정해봅시다.

```
for ( ; string[pos] != ' ' && pos < maxlen; pos++ )
;
```

이 코드는 단어 구분자로서 공백을 찾습니다. 또한, 문자열 끝을 검사합니다. 문자열 끝의 NULL 을 공백으로 대체하면 루프당 1 회의 비교를 줄일 수 있습니다. 다음과 같이 바꿀 수 있습니다.

```
for ( ; string[pos] != ' '; pos++ ) ;
```

일단 작업을 완료하고 나면 NULL 값을 되돌려 놓았는지 반드시 확인해야 합니다.

- 테이블 조회: 테이블 조회를 할 때, 가장 일반적인 것이 먼저 나오게 하도록 정렬할 필요가 있을 수 있습니다. 이렇게 함으로써 한 번의 비교로 대부분의 검색을 종료할 수 있을 것입니다. 영문자순 정렬이나 이진 검색을 실행해야 하는 상황도 있을 것입니다. 같은 확률을 갖는 100 개 요소가 포함된 테이블의 경우, 평균 50 회의 비교를 하는 순차 검색과 달리 단 7 회의 비교로 결과를 얻을 수 있습니다.

데이터베이스 액세스

응용 프로그램에서 데이터베이스를 액세스하는 기능을 추가하게 되면, 데이터베이스 서버와의 통신과 관련된 몇 가지 이슈들을 야기합니다. 여기에서는 고려해야 할 몇 가지 중요한 사항에 대해 설명합니다.

데이터베이스 연결

데이터베이스 서버와의 연결을 설정하기 위해서는 여러 가지 작업을 필요로 합니다. 이 작업은 인증 절차를 거쳐 실행 스레드를 생성해야 하는 로그인 절차로 볼 수 있습니다. 응용 프로그램을 사용하는 동안 연결을 유지해야 합니다. 연결을 바꿀 필요가 있을 경우에도 다시 사용할 수 있도록 다른 연결들을 유지해야 합니다. 이 작업은 사용된 클라이언트 인터페이스에 따라 서로 다른 방법으로 수행됩니다. Informix ESQL/C 를 사용하면, 연결 이름을 지정할 수 있습니다.

```
EXEC SQL CONNECT TO 'database' AS 'connection1';
```

이 방법을 통해 연결에 이름을 부여할 수 있습니다. “connection1”과 “connection2”라는 이름의 두 연결을 열었을 경우, 다음과 같이 “connection1”을 현재 연결로 지정할 수 있습니다.

```
EXEC SQL SET CONNECTION 'connection1';
```

다중 스레드 환경일 경우, 스레드간에 연결을 이동시킬 수 있습니다. 다른 스레드에서 연결을 사용할 수 있도록 하려면 먼저 연결을 “유휴”로 설정해야 합니다. 다음 코드를 수행하면 “connection1”을 다른 스레드에서 사용할 수 있습니다.

```
EXEC SQL SET CONNECTION 'connection1' DORMANT;
```

“connection1”이 현재 연결이면 다음 코드를 사용할 수 있습니다.

```
EXEC SQL SET CONNECTION CURRENT DORMANT;
```

연결 이름은 호스트 변수를 통해 전달할 수 있습니다. 그럴 경우, 연결 풀을 보다 쉽게 관리할 수 있습니다. 자세한 사항에 대해서는 Informix 설명서를 참고하십시오.

구문 준비

SQL 문을 수행하려면, 먼저 SQL 문을 분석하고 최적화해야 합니다. 자주 수행하는 구문일 경우 이 작업은 큰 부담이 될 수 있을 것입니다. PREPARE 문을 사용하여 자주 수행되는 구문을 준비해 두면 이런 부담을

덜 수 있습니다.

```
EXEC SQL PREPARE stmt1 FROM

"UPDATE customer SET lname=? WHERE cus_num=?";
```

이렇게 하면, 물음표 위치에 대신 인수를 전달하여 구문을 수행합니다.

```
EXEC SQL EXECUTE stmt1 USING :lname, :cnum;
```

이렇게 여러 번 실행할 SQL 문을 한 번에 준비함으로써 큰 성능 향상 효과를 얻을 수 있습니다.

삽입 커서

삽입할 행이 많을 경우, 행을 버퍼 처리함으로써 데이터베이스와의 통신을 최적화할 수 있습니다.

FET_BUF_SIZE 환경 변수에 대한 자세한 사항은 Informix SQL 참조 설명서를 보십시오. 아울러, sqlhosts 파일에서 기본 설정 값을 지정할 수 있습니다.

서버 측의 프로그래밍

Universal Data Option 을 가진 Informix Dynamic Server 는 새로운 가능성들을 제공합니다. Michael Stonebraker (“Betting on ORDBMS,” Byte Magazine, April 1998)에 따르면, 현재 너무나도 많은 클라이언트, 미들웨어 및 데이터베이스들이 나와 있기 때문에 그 중에서 선택할 수 있습니다. 실제적으로는 처리 요구 사항에 따라 이 세 가지를 혼합하여 사용할 수 있습니다.

Universal Data Option 은 엄격한 지속성 메카니즘이라기 보다는 응용 프로그램의 일부가 될 수 있습니다. 대부분의 업체는 업무 데이터를 처리하는 특별한 방법을 가지고 있습니다. 따라서, 그 업체에서 사용할 수 있는 보고서 작성, 인쇄 등의 모든 추가 기능이 포함된 응용 프로그램을 개발할 필요가 있습니다. 데이터를

처리하기 위한 코드를 추가해야 할 것입니다. 이는 서버에서 함수를 추가하는 것보다 복잡해 보입니다. 또한, 데이터를 응용 프로그램에 전송하는 것은 서버에서 처리하는 것보다 부담이 더 큽니다. vehicleType() 함수의 결과를 색인화할 수 있을 것입니다. 그러한 색인이 유용하다면 사용될 것이며 성능도 함께 향상될 것입니다.

Universal Data Option 을 사용하면 그 밖에도 많은 작업을 할 수 있습니다. 새 유형, 새 연산자, 새 합계 함수 등을 만들 수 있습니다. 여기에 대해서는 참고 서적 목록에 표시된 “Informix Dynamic Server with Universal Data Option: Best Practices” 서적을 참고하십시오.

맺음말

성능은 복잡한 주제이며 성능 평가는 더욱 더 복잡합니다. 지금까지 살펴본 것과 같이 처리 비용은 프로세서, 즉, 캐시, 메모리, 디스크, 네트워크 등의 차이에 따라 크게 증가합니다. 또한, CPU 는 입출력

설계 및 구현 단계에서 이런 기본 개념을 명심함으로써 적절한 성능을 제공하는 응용 프로그램을 만들 수 있습니다.

참고 서적

Jon Bentley, Programming Pearls, ISBN 0-201-10331-1, Addison-Wesley, 1986

Jon Bentley, More Programming Pearls, ISBN 0-201-11889-0, Addison-Wesley, 1988

Maurice J. Bach, Design of the UNIX Operating System, ISBN 0-13-201799-5, Prentice-Hall, 1986

Samuel J., Leffler ...[et al.], Design and Implementation of the 4.3 BSD UNIX Operating System, ISBN 0-201-06196-1, Addison-Wesley, 1989

Andrew S. Tanenbaum, Operating Systems Design and Implementation, ISBN 0-13-637406-9, Prentice-Hall, 1987

Bil Lewis and Daniel J Berg, Threads Primer, ISBN 0-13-443698-9, SunSoft Press, 1996

Douglas Comer, Internetworking with TCP/IP, ISBN 0-13-470154-2, Prentice-Hall, 1988

Jacques Roy, Threaded Application Development Using Informix Client SDK: A Practical Guide, Informix Tech Notes Volume 8, Issue 3, 1998

Informix Press, Informix Dynamic Server with Universal Data Option: Best Practices, ISBN 0-13-911074-7, 1998

데이터 모델링의 개요

관계형 데이터베이스 구축은 크게 4 단계로 구분할 수 있습니다.

정보전략계획 단계는 가장 효율적으로 기업을 운영하는데 필요한 정보의 전략적 관점, 기업을 개선하는데 기술이 어떻게 이용되는지에 관한 전략적 관점, Data, Activity, Technology 의 최상위 계층으로 전략 계획을 수립하여 기업이 필요로 하는 정보에 대한 전략적 Vision 제시와 전략 계획 수행하는 단계입니다.

업무영역분석 단계는 완전히 정규화된 논리적 데이터 모델을 제시하며 기업을 운영하는 데 필요한 처리 과정과 통합 방법을 나타내는 단계로 기업운영에 필요한 논리 Model 의 구축과 Data 를 사용하는 활동, Data 를 저장, 유지 보수하며 조작하는 기법을 제시합니다. 이 단계는 상세한 부분이 아닌 무엇이 필요하고 어떻게 구축할 것인가의 방법을 고려합니다

설계단계는 특별한 순서에 의해 사용된 Record 의 설계, 특별한 과정을 처리, 수행하기위한 절차의 단계입니다. Data 의 상세 Design, Data 처리 System 과 Data 와의 직접 연결, H/W 와 S/W 의 관계 등을 나타냅니다.

구축단계는 Data 를 이용한 응용 Program 단계로서 Code 생성기에 대한 상세한 Program 논리 또는 입력에 관한 설계, Physical DB 의 구조, 응용 Program 의 접근, H/W 및 S/W 의 선정 등의 활동을 정의합니다. 이 단계에서 Files, DBMS, Program 의 구조, 기술적 설계, 상세구축 등을 고려합니다.

데이터 모델링은 이러한 구축 절차 중 업무영역분석(BAA) 단계에 데이터모형을 작성하는 기법이라 할 수 있습니다. 그럼 데이터 모델링 절차와 목적을 상세하게 살펴보도록 하겠습니다.

데이터 모델링이란 기업의 정보구조를 실체와 관계를 중심으로 정해진 기호와 규칙을 사용하여 명확하고 체계적으로 표현하고 문서화하는 기법을 말합니다. 이 작업은 업무영역분석(Business Area Analysis) 시 프로세스 모형 설계와 병행하여 데이터 모형을 완성하기 위한 기법이라 할 수 있습니다.

- 데이터 모델링을 수행하는 목적
    - 연관조직의 정보 요구에 대한 정확한 이해 제공
    - 분석자, 개발자, 사용자 간의 의사 소통 수단
    - DATA 중심의 분석 방법
    - 변경 및 영향에 대한 분석 제공

실체 정의

업무수행을 위하여 기업이 알아야 될 대상이 되는 사람, 장소, 사물, 사건 및 개념 등을 실체라고 정의하며 각 실체는 유일하게 식별 가능해야 합니다.또한 인스턴스라 불리는 개별적인 객체들의 집합이라 할 수 있습니다.

즉 경복궁, 덕수궁, 비원의 인스턴스들을 묶어 고궁이라는 실체를 정의할 수 있습니다.

실체의 조건으로서는 업무에 유용한 정보를 제공하며 명확한 속성 유형이 하나 이상 존재하고 최소한 하나 이상의 실체와의 관계를 가지고 있어야 합니다. 실체를 추출하는데 있어 대상이 되는 데이터로는 정보 전략 계획 단계의 전사 데이터, 현행 사용 장표 및 각종 서식, 현행 정보 시스템의 데이터 구조를 대상으로 합니다. 이러한 데이터를 가지고 실체를 추출 시 주의 사항으로는 경험된 Project 의 내용을 가지고 실제 회사 내에 존재하지도 않는 실체를 상상력으로 결정하지 말고 실제 존재여부를 확인한 후 정의하라는 것입니다. 상상력으로 만든 실체는 사용자에게 오히려 업무부담을 줄 뿐 아니라 정보시스템의 신뢰도까지 떨어뜨릴 수 있습니다. 실체명을 부여할 시는 현업용어(업무사용용어), 단수명사, 유일한 명칭을 사용하는 것이 좋습니다.

실체에 대한 정의항목은 실체명, 동의어, 발생건수, 성장율 등을 기술하여 분석가외에 개발자, 사용자들도 실체에 대해 쉽고 빠르게 이해할 수 있도록 하여야 하며 실체의 Volume 에 대한 예측이 가능하도록 하여야 합니다.

- 실체 정의의 예

  실 체 명: 고객

  정 의: 당사의 제품 및 상품을 구입한 개인 또는 법인

  종 류: 독립 실체

  동 의 어: 거래처

  발생건수: 5000 건 / 년

성 장 율: 20% / 년

관계 정의

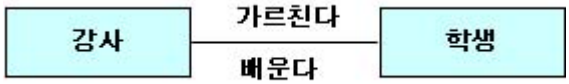
관계는 두 실체를 연결하여 보고자 하는 업무적인 이유, 업무규칙(Business Rule)이나 제약(Constraints)을 나타내는 두 실체사이에서의 논리적인 연결입니다.

관계를 추출하기 위한 대상 데이터로는 정보전략계획의 산출물인 정보 아키텍처, 정보수집 과정중의 업무 설명, 수집된 사용 장표, 실체 정의 내용이며 추출 기준은 현업 사용 동사형 어휘와 한 장표에 나타나는 실체들간의 업무규칙으로 적용합니다.

모든 관계는 관련되는 두 실체들 중 어떤 실체가 주체로 참여하는가에 따라 관계의 의미가 결정되며 참여 주체 실체를 기준으로 하는 각각의 관계의미를 관계명으로 구성합니다.

관계요소명(관계명)에 대한 명명은 하나의 관계에 두개의 관계명으로 표기하며 사용자 용어를 사용하고 현재시제 동사로 능동/수동형으로 시계방향으로 표현합니다.

예를 들어 강사와 학생간의 관계에서 '강사는 학생을 가르친다', '학생은 강사로부터 배운다'라는 양방향의 관계명을 기술할 수 있습니다.



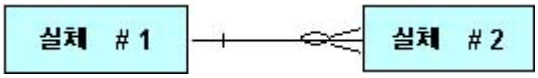
관계의 종류를 살펴보면 일반적 관계(Master, Detail 구조), 재귀관계, 병렬관계(Child, Parent 관계가 2 가지 형태로 접근), 상호 배타적 관계(사원 ? 정규직 사원, 계약직 사원)등이 있으며 이러한 관계를 통해 외부키(foreign Key)가 나타나게 된다. 즉 외부키는 부모실체의 주식별자가 자식실체로 이주된 속성이나 속성그룹입니다.

관계의 종류를 식별성으로 구분해 보면 식별관계와 비 식별관계로 구분할 수도 있습니다. 식별관계는 부모의 주식별자가 자식의 주식별자의 일부가 되어 부모의 주식별자 없이 자식의 각 인스턴스를 식별할 수 없는 관계이며 비 식별관계는 부모의 주식별자가 자식의 Non-Key 영역(주식별자가 아닌 일반속성영역)으로

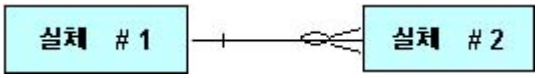
이주한 관계로 부모 주식별자 없이도 독립적으로 자신의 각 인스턴스를 식별할 수 있는 관계입니다.

두개의 실체간의 관계에서는 관계에 참여하는 각 실체가 얼마나 많이 참여할 수 있는가의 관계 비율을 표현하는 기수성(Cardinality), 관련되는 실체 존재 조건으로 관계연결의 여부가 미치는 영향을 표현하는 선택성(Optionality)을 정의합니다.

- 기수성(Cardinality)의 종류
    - 1 : 0,1,M - 1 대 0 또는 그 이상

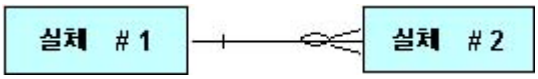


    - 1 : 1,M - 1 대 1 또는 그 이상



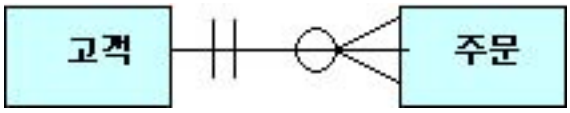
    - 1 : 0,1 - 1 대 0 또는 1

    - 1: N - 1 대 N



기수성의 관계문장은 '각 주 실체는 [오직 한 | 하나 또는 여러]대상 실체로 관계합니다'로 표현할 수 있으며 선택성은 '각 주 실체는 대상 실체와 [항상 | 때때로]로 관계합니다'고 표현할 수 있습니다. 이를 통합하여 관계명을 완성해 보면 '각 주실체는 [오직 한 | 여러] 대상실체와 [항상 | 때때로] 관계합니다.'

- 관계명 완성의 예 (고객과 주문간의 관계)



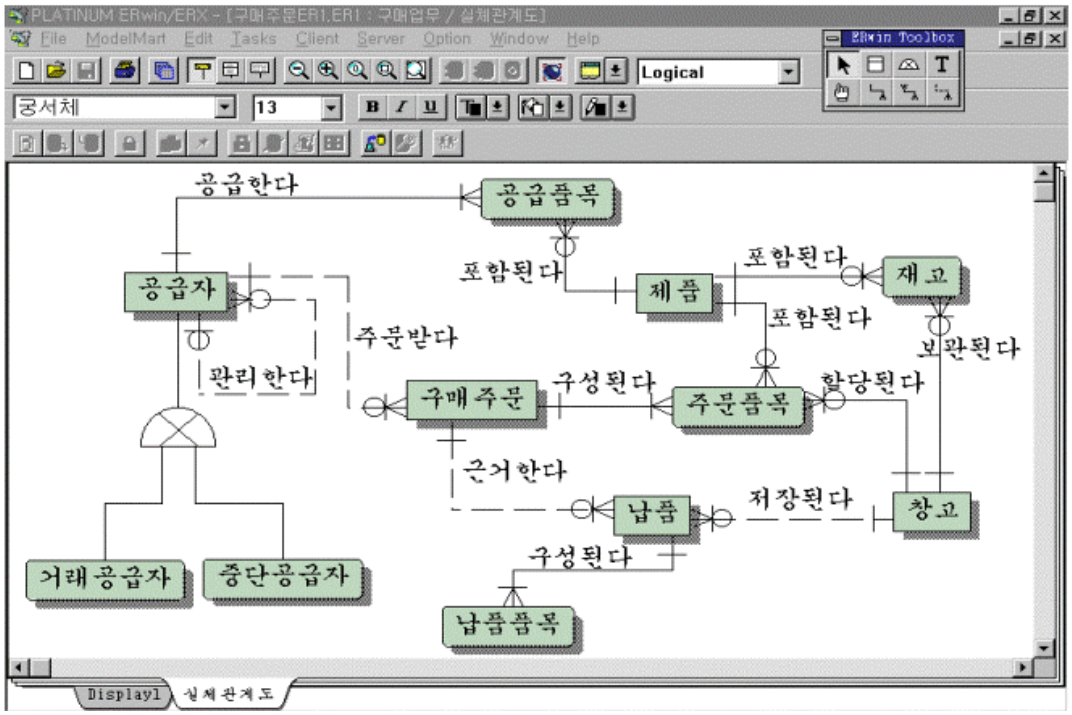
각 고객은 여러 주문을 때때로 발행합니다.

각 주문은 오직 한 고객으로부터 항상 발행됩니다.

관계의 상세화 작업은 N:N 의 관계를 제거합니다. 즉 교차실체를 추가하며 배타적 관계를 파악하고 중복된 관계를 없앱니다.

제품과 거래선과의 N : N 관계를 제거한 교차실체(주문)

- 실체와 실체간의 관계를 통한 실체관계도(ERD)를 작성하는 순서
    - 실체의 표기: 실체명을 기록
    - 실체의 배열: 프로세스 진행주기와 관련된 실체 중심부에 배열하고 2차적 실체를 가까운 곳에 배열하고 배열된 실체와 관련 핵심 실체를 외곽에 전개
    - 관계의 연결
    - 관계명 표기: 시계방향을 관계명 기록
    - 관계의 기수성 결합
    - 관계의 선택성 결합



실체관계도(ERD)의 예

식별자 정의

식별자(Key, Identifier)란 한 실체내의 특정 인스턴스를 구분할 수 있는 단일 속성 또는 속성그룹을 말합니다. 식별자 업무규칙은 한 실체내의 인스턴스를 유일하게 구분할 수 있어야 하며 모든 실체는 반드시 하나 이상의 식별자를 보유하여야 하며 복수개의 식별자도 보유할 수 있습니다.

- 식별자의 종류
    - 후보 식별자(Candidate Key)
    - 주 식별자(Primary Key)
    - 대리 식별자(Surrogate Key)
    - 부 식별자(Alternate Key)
    - 역 엔트리(Inversion Entry)

후보 식별자란 실체의 각 인스턴스를 유일하게 식별하기 위하여 제공되는 속성이나 속성의 그룹을 말합니다. 예를 들어 사원 실체의 후보 식별자는 사번, 주민등록번호가 될 수 있는 것입니다.

주 식별자란 실체의 각 인스턴스를 유일하게 식별하는데 가장 적합한 식별자로 후보 식별자에서 선택한 속성이나 속성의 그룹입니다. 주 식별자로 선정 시 고려사항은 업무적으로 활용도가 높은 것, Short & Simple 한 것, Null 을 포함하지 않는 것, 정적으로 유지되는 것을 선정하여야 합니다.

대리 식별자는 긴 복합 식별자를 대체하는데 사용되는 인위적이고, Non-Intelligent 한 단일 속성을 말합니다. 예를 들어 은행의 현금지급에 주식별자는 계정번호,고객번호, 거래일자이지만 이것을 하나의 속성인 거래번호로 정의할 수 있습니다. 이때 거래번호가 대리 식별자라 할 수 있습니다.

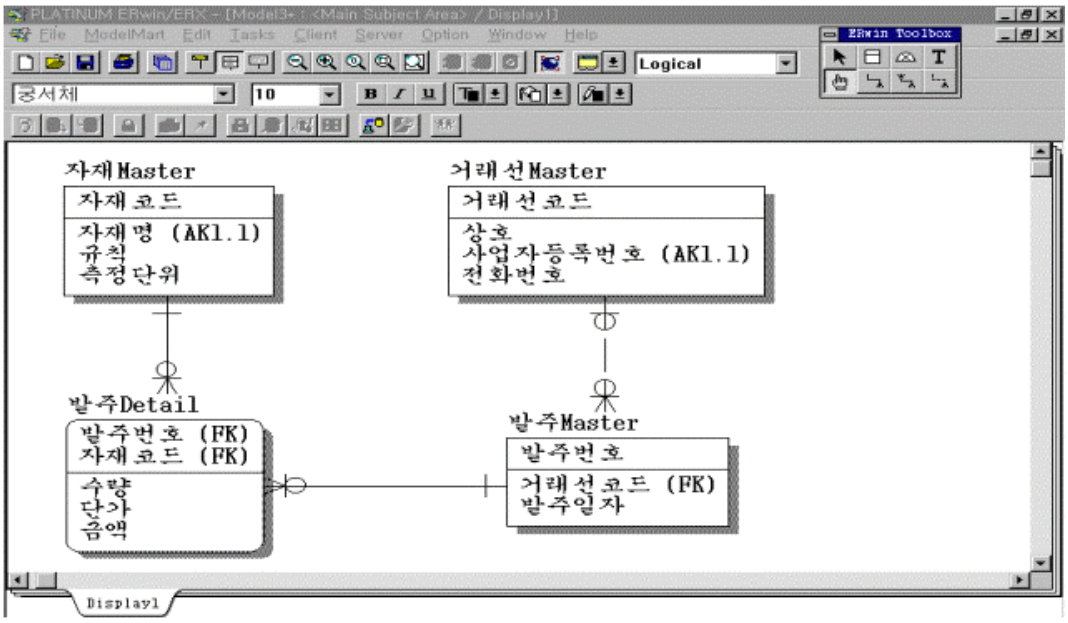
부 식별자란 후보 식별자에서 주 식별자로 선택되지 않은 식별자입니다.

즉 사원 실체에서 주민번호가 부 식별자가 될 수 있습니다.

또한 하나 또는 그 이상의 속성이 실체의 인스턴스를 접근(Access)하는데 자주 사용되어 Index 로 활용하고자 할 때 부 식별자로 선정합니다.

식별자 업무 규칙은 실체관계의 무결성을 강조한 개념으로 하나의 실체 내에서 인스턴스 하나 하나가 입력, 삭제되거나 외부식별자가 변경될 시 계속적인 관계유지 방법을 정의하는 것입니다.

입력규칙은 자식 실체에 한건의 인스턴스가 입력 시 행해지는 규칙을 말하며 삭제규칙은 부모실체에 한건의 인스턴스가 삭제 시 행해지는 규칙을 정의하는 것을 말합니다. 이러한 업무규칙은 현업의 업무처리 규칙에 근거하여 추출합니다.



식별자 예제

속성 정의

속성은 정보가 유지되어지는 실체에서 개개의 특성, 특질을 나타내는 것으로 더 이상 분해할 수 없는 최소한의 데이터 보관단위를 의미합니다

속성의 추출은 첫째 업무영역분석단계에서 수집된 서식, 장부, 양식 등 각종 자료에 있는 정보단위를 정의합니다. 둘째, 실체의 정의 시 파악된 속성을 정의합니다. 셋째, 프로세스 모델링을 하면서 필요한 정보들을 정의합니다. 넷째, 기존 정보시스템에 존재하는 필드(Field) 또는 속성을 정의합니다. 이 네가지 유형은 중복이 되더라도 각 업무분석 시 거쳐가는 단계마다 작업을 수행해야 합니다.

속성의 형태는 Key 와 Non-Key 로 구분됩니다.

속성의 명명은 현업의 표준용어를 사용하고 필요 시 수식어를 달도록 하며 장표나 서식의 명칭을 그대로 기술하는 것이 좋으며 소유격 사용을 배제하며 핵심단어들로 속성명을 기술하여 누구나 쉽게 속성명을 이해할 수 있도록 하는 것이 좋다. 또한 정보시스템실과 현업의 협조를 통해 속성에 대한 명명규칙(Naming Rule)의 기준안을 정하여 사용하는 것도 바람직한 방법이라 할 수 있습니다.

속성의 정의 사항은 명칭, 설명, 형식, 길이, 유효값, 기본값 등을 정의합니다.

- 속성 유형에는 다음과 같이 세 가지가 있습니다.

    - 기초 속성(Basic Attribute): 외부로부터 정보가 제공되어야만 유지되는 속성. 즉 현업에서 기본적으로 사용되는 속성
    - 추출 속성(Derived Attribute): 기존 속성으로부터의 가공처리를 통해 생성 및 유지되는 속성
    - 설계 속성(Designed Attribute): 실제로 존재하지는 않으나 시스템의 효율성을 도모하기 위해 설계자가 임의로 부여하는 속성

예를 들어, 주문실체에 주문번호, 주문일자, 주문총금액, 주문상태가 있고 주문상세실체에 주문번호, 주문품목, 주문수량, 주문단가, 주문금액이 있습니다. 여기서 주문상태는 설계자가 주문의 진행 상태(정상,지연,취소)를 확인하기 위해 부여한 설계속성이며 주문총금액, 주문금액은 주문수량과 단가를 계산을 통해 가공된 속성이기 때문에 추출속성에 해당합니다. 추출속성은 자료의 중복성 및 무결성 확보를 위해 최소화하는 것이 바람직합니다.

속성의 도메인(Domain)은 속성에 대한 세부적인 업무제약조건 및 특성을 전체적으로 정의해 주는 부분입니다. 도메인 추출은 추후 개발 및 실체를 데이터베이스로 생성할 때나 프로그램 구현 시 유용하게 사용하는 산출물입니다.

- 도메인 정의항목

    - 데이터 타입
    - 길이
    - Format Mask
    - Unique
    - Null 여부
    - 초기값
    - 유효값

속성의 업무 규칙은 도메인의 무결성에 부가하여 한 실체의 어떤 속성에 대해 입력, 수정, 삭제발생시 같은

실체나 다른 실체에 존재하는 다른 속성들에 미치는 영향을 관리하는 규칙입니다. 이는 한 속성에 대해 발생하는 사건(Event)에 대해 연쇄적으로 발생하는 작용으로 연쇄작용(Trigger Operation)이라 합니다. 이러한 속성업무 규칙은 일부는 프로그램 구현에서, 또 일부는 DB Trigger 를 이용하여 구현됩니다.

속성에 대한 각각의 이벤트 발생시 처리하거나 체크되어야 할 업무규칙에 대해서는 사용자와 지속적인 협의를 거쳐서 가능한 세부적으로 기술해 주는 것이 좋습니다. 이외에도 필요한 업무규칙은 설계자가 추가해서 정리해 두어야 합니다.

이러한 속성업무규칙은 각 단위 프로그램의 구현 시 참조가 되는 미니스펙(Mini Spec)에도 첨부가 되어서 개발자가 구현 시 참조할 수 있도록 해야합니다.

속성업무 규칙 정의 내용은 속성값에 대한 무결성, 일치성을 유지하는 모든 업무규칙의 연쇄작용 정의, 연쇄작용을 발생시키는 작업, 작업대상, 발생되는 조건, 연쇄작용의 결과로 수행되어질 작업의 정의, 동일실체내의 속성에 영향을 미치는 내용의 정의, 다른 실체내의 속성에 영향을 미치는 내용의 정의, 중복자료의 유지에 관한 정의, 추출속성에 관련된 원시속성의 Event 에 대한 연쇄작용 정의 등이 있습니다.

정규화(Normalization)

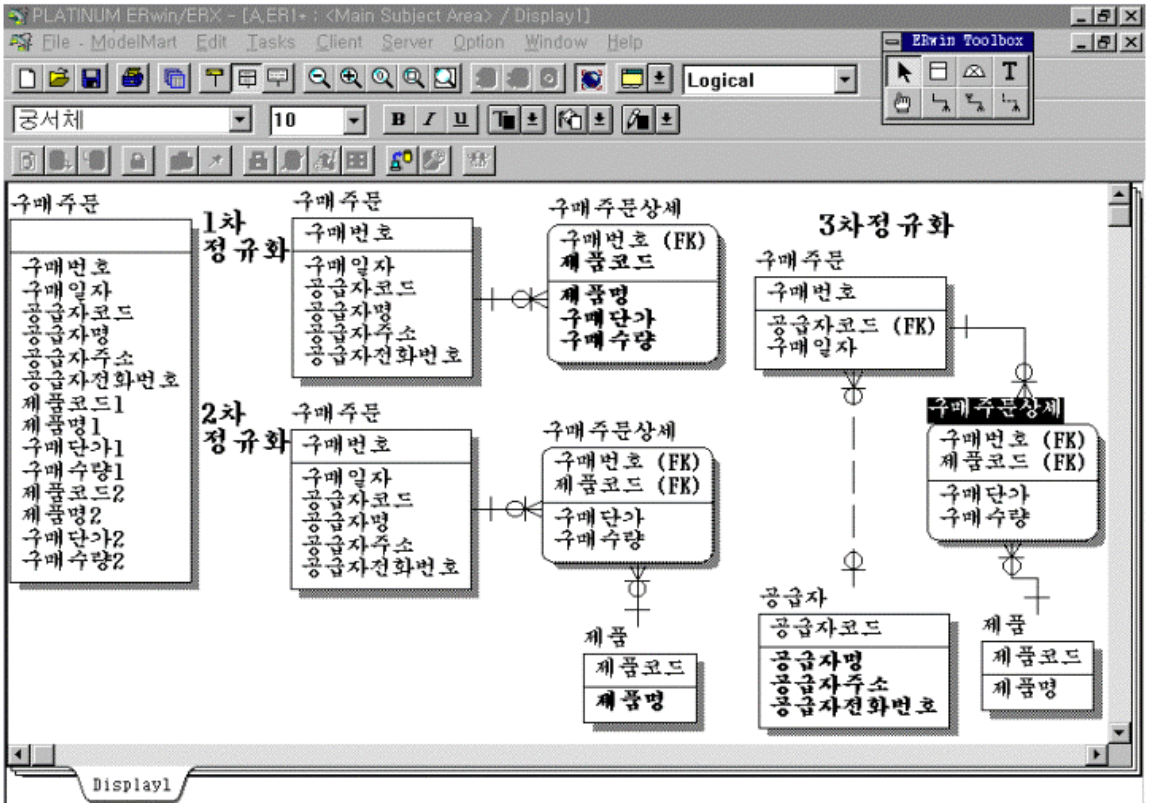
완전하게 정규화된 Data Model 은 정확성, 일치성, 단순성, 비 중복성, 안정성을 보장해 주는 최적의 데이터 모형입니다.

정규화의 목적은 정보의 중복을 최소화하고 정보모형의 단순화, 정보 공유도 증대, 정보의 일관성 확보, 정보신뢰도 증대, 속성의 배열 검증, 실체/하부유형실체/속성의 누락여부 검증 등입니다.

- 정규화의 수학적 표현
    - 1차 정규화: 반복되는 속성이나 Group 속성 제거
    - 2차 정규화: 주 식별자(주 식별자가 복합속성인 경우에 한하여) 전체에 종속되지 않는 속성의 분할 제거
    - 3차 정규화: 주 식별자가 아닌 속성(비식별자)에 종속하는 속성의 분할 제거
    - 4차 정규화: 특정 속성값에 따라서 선택적인 속성의 제거

- 비 정규화(DeNormalization): 데이터 모델링 규칙에 얽매이지 않고 수행 System 이 물리적으로 구현되었을 때 순전히 Performance향상을 목적으로 한 작업

- 정규화 예: 제품구매주문의 정규화 작업



일반화

일반화는 공통된 특성을 가진 일련의 실체들을 그룹으로 묶는 방법입니다. 일반화을 고려해야 할 대상은 특정실체를 좀더 세분하여 표현해야 할 필요가 있거나 논리적으로 모델에 대한 이해를 명확히 해야할 필요가 있는 실체에 대해 일반화 작업을 수행합니다.

일반화의 기본 요소는 상부실체(SuperType), 즉 카테고리 실체를 포함하는 총체적 실체와 하부실체(SubType), 즉 하나의 실체로부터 특정 업무논리 성격으로 세분화되어 관리되는 실체와 카테고리 판별자로 불리우는 구분속성, 즉 Subtype 의 구분을 결정하는 속성으로 구성된다.

일반화 정의 절차는 속성,관계의 선택성 검증 후 선택성을 상세화하여 규명하는 업무규칙 정의, 구분속성의

설정, 속성 재배열, 관계상세화 순서로 진행합니다.

사원과 차량에 대한 일반화 예

모형 통합 및 확장성 분석

데이터 모형의 통합은 각각의 특정 업무 영역별 관점에 의해 설정된 논리적 모형을 조정하여 기존의

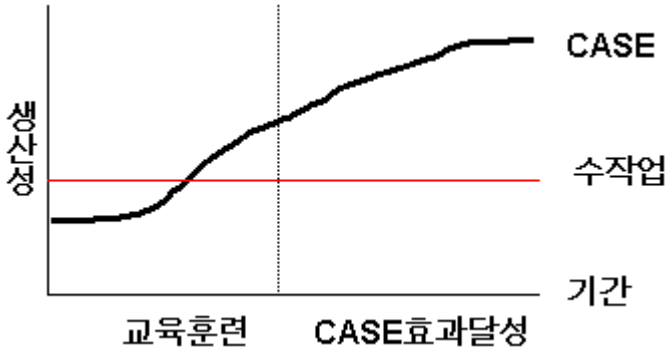
가능성 분석 작업을 하여 향후 확장에 대한 분석 작업을 하여야 합니다.

CASE 정의

기업에서 IT 기술은 점점 더 비중이 커져가고 있으며 IT 기술의 효과적 활용이 기업의 경쟁력을 좌우하는 상황이 되었습니다. 정보시스템이 기업활동에 미치는 영향을 들자면, 첫째로 새로운 상품, 서비스 도입은 정보시스템 지원 없이는 불가능하다는 것입니다. 둘째, 정보화 투자가 늘어남으로써 전산비용은 기업의 중요 비용항목으로 대두 되었습니다. 셋째, 정보기술의 활용은 실제로 기업의 수익성을 향상시켜주고 있습니다. 업무처리비용과 시간을 대폭 절감하였으며 고객의 요구를 신속하게 충족시켜주고 있습니다.

이러한 추세에 따라 새로운 개발환경 구축에 대한 인식이 확산되었으며 이에 대한 해결책의 한 부분으로 등장한 것이 CASE(Computer Aided Software Engineering)입니다. CASE 는 정보시스템 부서의 업무를 자동화, 단기간의 저렴한 비용으로 품질 높은 시스템을 공급하는 것이 목적입니다.

아래 도표는 CASE 의 학습곡선입니다



CASE 활용

최초의 CASE 의 활용은 소프트웨어의 개발공정을 지원했으며 그중에서도 주로 도표작성 과정에서 사용되었습니다. 업무 분석 단계나 시스템설계 단계에서 도표화 기법을 주로 사용함에 따라 모든 도표를 수작업으로 작성합니다는 것은 많은 노력이 소요되며 변경이 어려움으로 도표화 지원 CASE 가 등장하게 되었습니다.

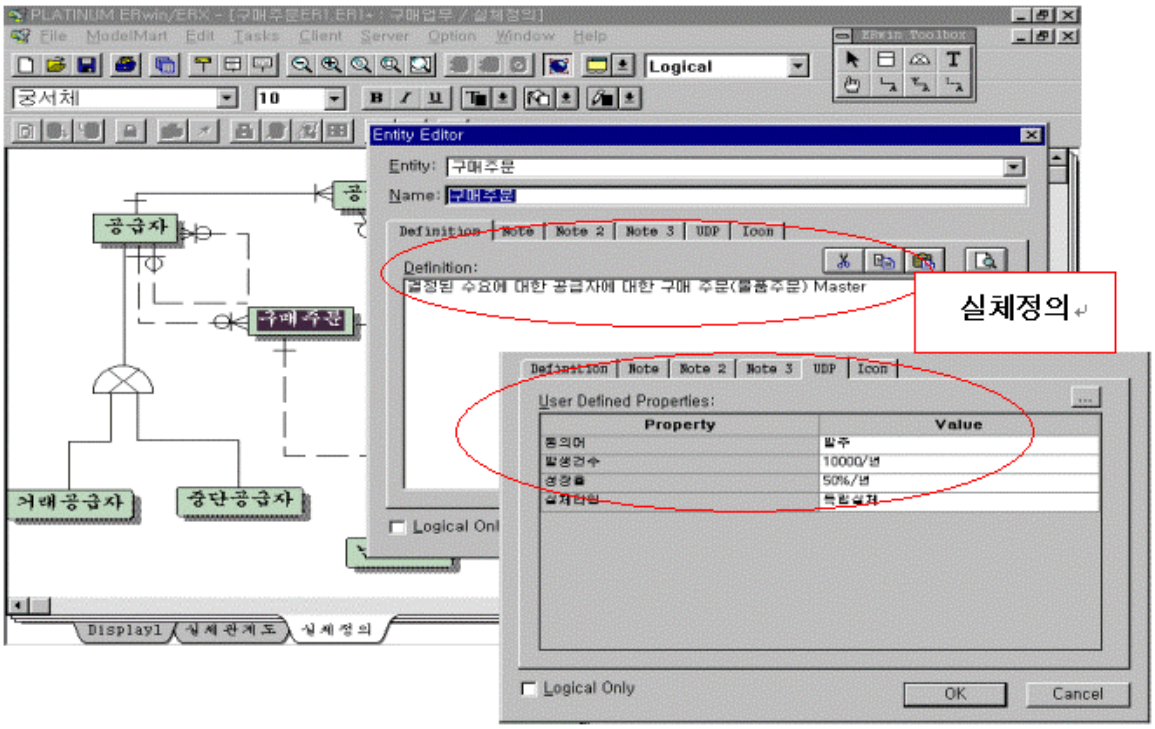
정보기술의 발전에 따라 CASE 의 지원범위도 확대 되었습니다. 단순화 도표작성에서 프로그램의 자동생성,

소프트웨어 개발관리, 유지보수, 정보자원의 효과적 관리등 여러분야에서 CASE 는 활동되어지고 있습니다.

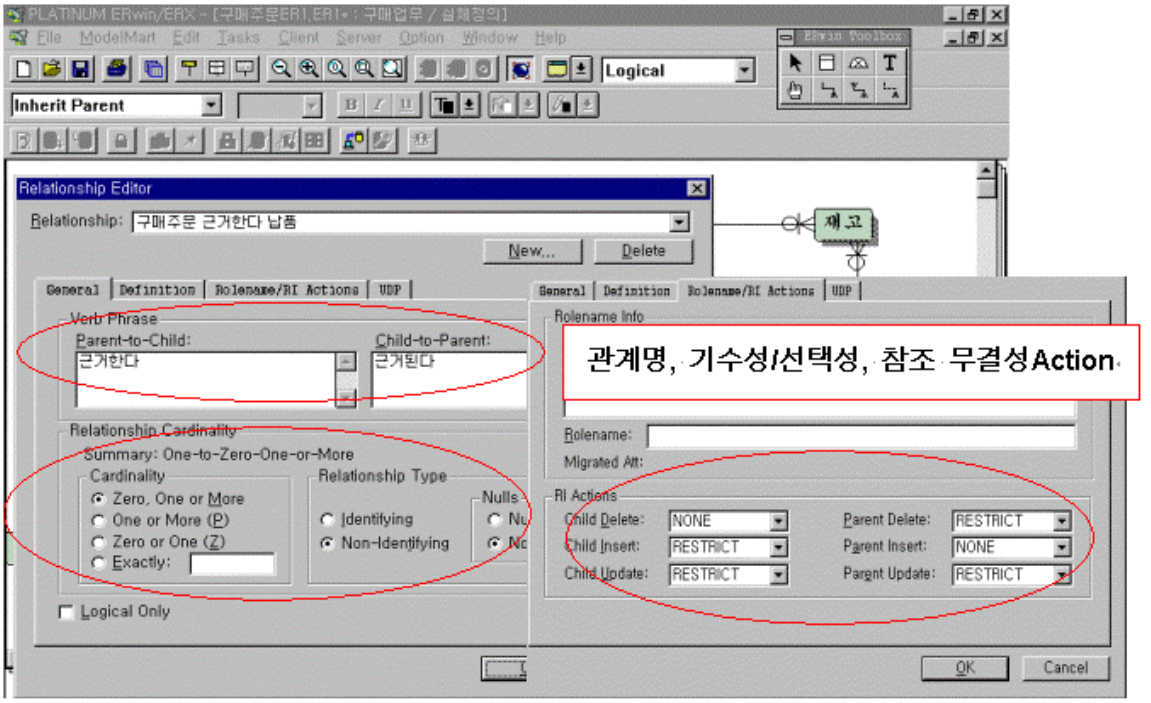
CASE 는 소프트웨어공학을 지원하는 도구를 의미합니다. 소프트웨어공학의 범위는 매우 커서 CASE 의 기능도 제품마다 다를수 있습니다. 실제로 소프트웨어공학의 모든 범위를 지원하는 CASE 는 존재하지 않습니다.

여기에서는 데이터 모델링과 데이터베이스 설계을 지원하는 CASE 의 활용방안을 살펴 보도록 하겠습니다.

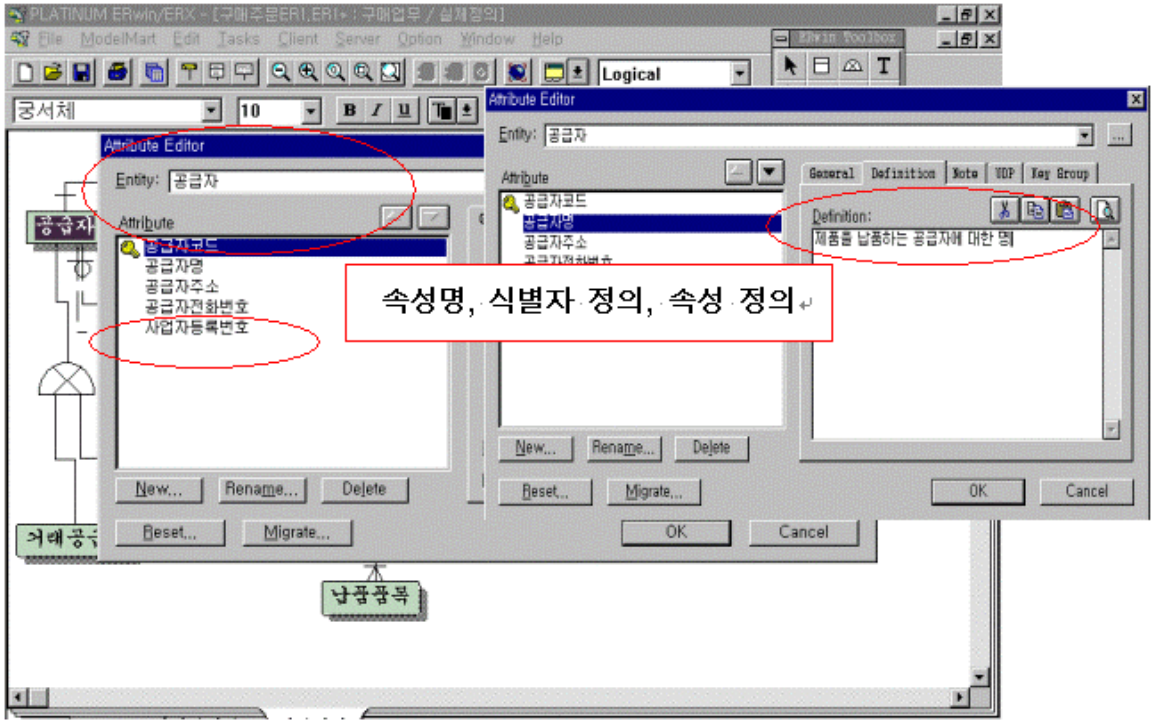
모델링 도구에서는 각 실체 정의에 대한 정의와 동의어, 발생건수, 성장율 등을 정의할 수 있어야 합니다.



또한 실체와 실체의 관계에 대한 기수성(Cardinality) 및 선택성 (Optionality) 표현, 관계명 기술, 데이터베이스 구축 시 업무 규칙으로서 삽입 규칙, 삭제 규칙, 수정 규칙등을 정의한 참조 무결성(Referential Integrity)을 기술할 수 있어야 합니다.



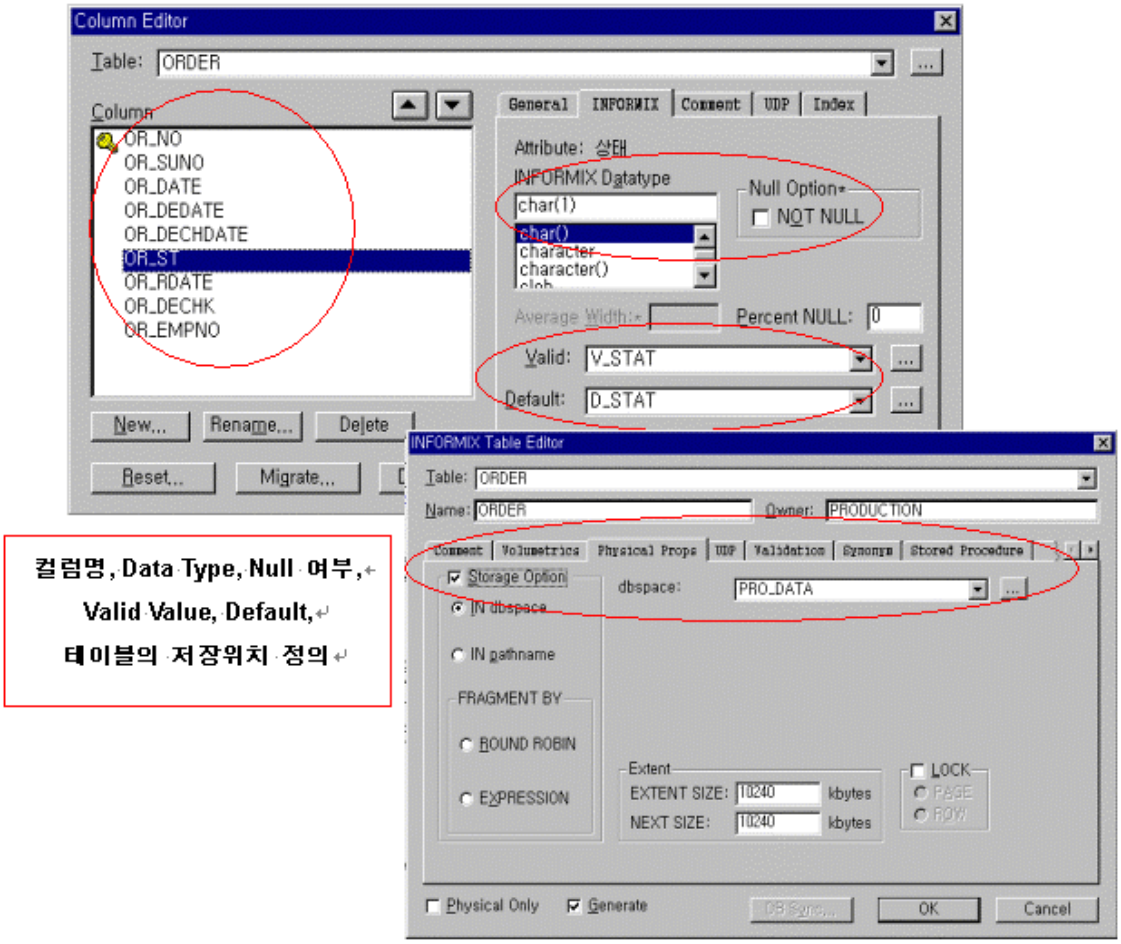
데이터모델링 도구로써의 CASE는 속성에 대하여서도 속성명, 속성 정의, 식별자 정의부분도 지원하고 있어야합니다.



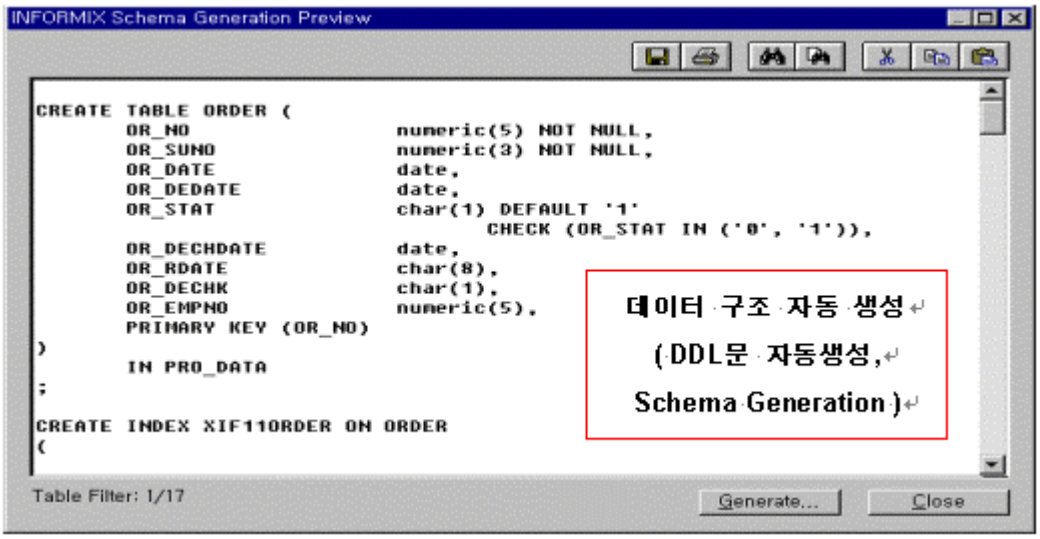
시스템설계단계에서의 CASE는 데이터베이스 설계부분 즉, 실체 관계도를 바탕으로 한 데이터베이스의 물리적 구조를 표현하는 데이터구조(Schema)를 자동으로 생성할 수 있어야 합니다. 데이터 구조도에

대해서는 물리적 성능을 감안한 조정작업을 수행하되, 이러한 조정작업이 실체 관계도에는 영향을 미치지 않아야 합니다.

데이터베이스의 물리적 구조의 표현부분은 Table 의 저장 위치와 속성의 Data Type, Null 여부, Valid Value(유효값), Default value(기본값)등을 표현할 수 있습니다.



또한 CASE 는 물리적 구조를 정의한 후 DDL(Data Definition)문을 자동으로 만들어 Database 에 데이터구조(Schema)를 생성해 주어야 합니다.

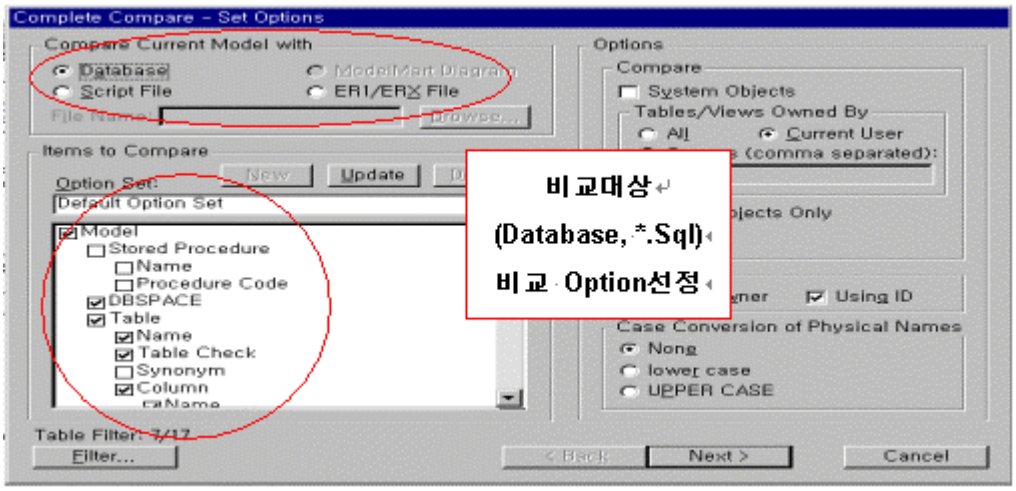


데이터 구조를 완성한 후 시스템 운영과정에서 CASE 의 역할은 모델 관리와 변경관리, 역공학, 도규먼트(Document) 관리 등을 수행합니다. 시스템 운영과정에서 발생하는 변경사항에 대해 CASE 는 변경 영향과 변경 범위에 대한 정보를 제공하고 변경 과정을 기록하여야 합니다.

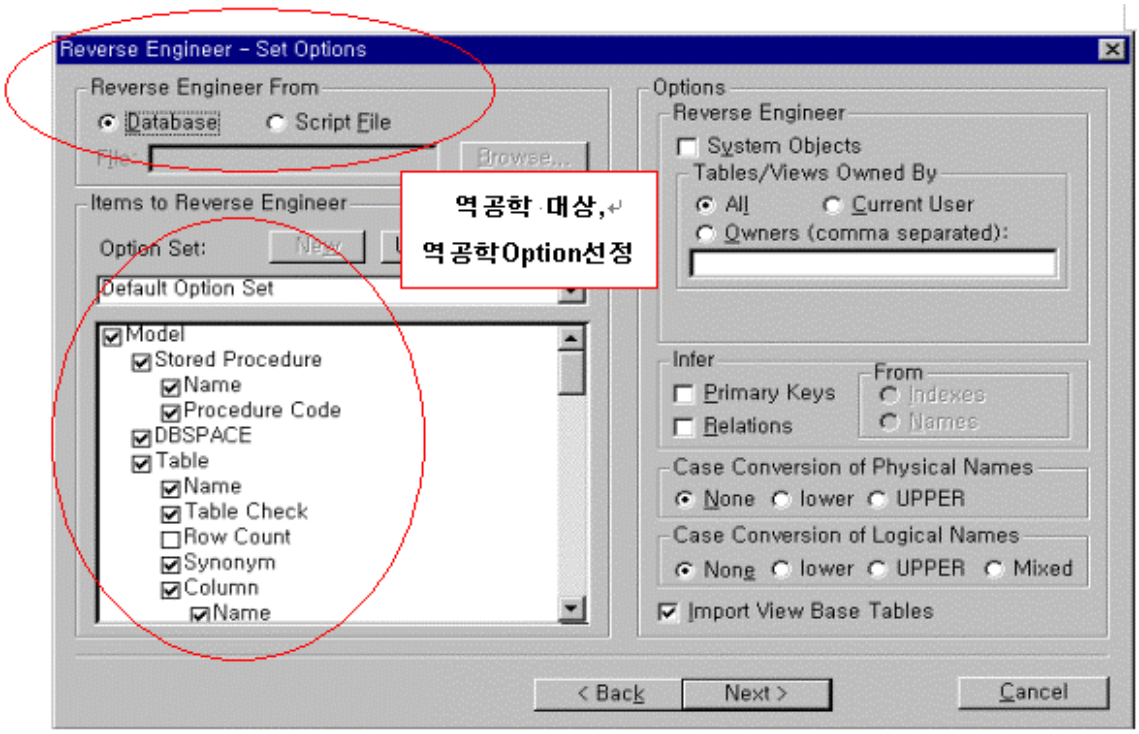
모델 관리는 CASE 로 개발된 시스템을 일관성 있게 운영할 수 있는 기능입니다. 전사적인 정보자원은 하나의 모델로 구축되어질 수 있고 몇 개의 모델로 분할할 수도 있습니다.

모델 관리에서의 CASE 는 전사 모델을 개인별 또는 업무별로 작업 범위를 정하여 프로젝트 수준의 CASE 로 체크인, 체크아웃을 통해 전송 또는 저장할 수 있어야 합니다. 또한 작업자에 대한 적당한 권한을 설정할 수 있어야 하며 작업 완료 후 상충되는 부분을 확인할 수 있도록 하여야 합니다.

변경 관리는 데이터 모델이 변경되었을 시 데이터 구조와 변경된 현재 데이터 모델 사이의 변화된 내용을 사용자에게 보여주어야 합니다.



역공학(Reverse Engineering)은 수작업으로 개발된 시스템을 CASE 로 재구성하는 작업입니다. 역공학이 필요한 이유로는 첫째, 변경에 대한 문서화가 되지 않아 현행 시스템의 유지보수 업무가 어렵기 때문입니다. 둘째, 기존시스템 환경 변경시 필요합니다. 즉 메인프레임 시스템을 다운사이징 시 역공학은 상당한 시간과 노력을 절감해 줍니다. 셋째, 신규업무를 CASE 로 개발한 경우 기존업무와의 인터페이스가 발생하여 유지보수가 양방향으로 발생됩니다. 이런 경우 CASE 로 개발된 업무와 관련성이 깊은 시스템에 대해 역공학을 적용할 수 있습니다.



ERwin CASE Tool 에 관한 문의

- Tel> 578-3528 ERwin 기술팀

- URL> www.genesis.co.kr