성능 향상을 위한 데이터 모델링 방법

김수연* · 이상호** · 서의호***

Data Modeling Methods for Performance Enhancement

Su-Yeon Kim* · Sang-Ho Lee** · Eui-Ho Suh***

요 약

현재의 시스템 개발 프로젝트에서 대부분의 기업은 관계형 데이터베이스를 목표 환경으로 채택하고 있지만 설계자들은 기존의 파일 시스템이나 계층형 데이터베이스의 설계 방식을 관계형 데이터베이스 설계 시에도 그대로 적용하려는 경향이 있다. 또한 모델링 시 시스템 특성과는 무관한 업무 중심의 엔티티관계도를 작성함으로써 구현 시에 모델을 상당 부분 변경하거나 추가해야 하는 오버헤드가 발생하기도 한다. 관계형 데이터베이스의 경우 구조를 어떻게 설계하느냐에 따라 효율의 차이가 크게 나타나므로 성능 향상을 위한 데이터 모델링 기법이 중요한 이슈로 등장하게 되었다.

본 논문에서는 관계형 데이터베이스 성능과 관련하여 이미 연구되고 실험된 내용을 여러 문헌과 자료를 참고하여 조사하였다. 성능 향상을 위한 모델링 기법들을 테이블, 릴레이션쉽, 인덱스 등의 객체별로 분류하여 정리하였고, 조사된 내용 중에서 몇 가지를 선택하여 Oracle DBMS 환경에서 실제로 실험을 실시하였다. 대용량 테이블에 대한 질의를 수행하여 소요되는 시간을 측정하고 그 결과를 분석하였다. 실험을 통해 검증된 결과를 토대로 제안되는 모델링 방법을 제시한다.

* 포항공과대학교 산업공학과
** 숭실대학교 컴퓨터학부
*** 포항공과대학교 산업공학과

1. 서 론

1990대 들어 기업의 시스템 개발 프로젝트에서 관계형 데이터베이스의 사용은 급속도로 확산되어 왔다. 기업의 정보시스템 수준에 따라 데이터베이스 설계와 관련된 몇 가지 경향이 존재한다. 관계형 데이터베이스를 처음 도입하는 기업의 경우에는 설계자들이 기존 파일 시스템이나 계층형 데이터베이스의 설계 방식을 관계형 데이터베이스 설계 시에 그대로 적용하는 경향이 있다. 데이터 모델링이 어느 정도 정착된 기업의 경우에는 비즈니스 중심의 모델링을 강조함으로써 구현 제약사항과는 독립적인, 즉 성능 측면이 충분히 반영되지 않은 모델을 산출하게 되어 구현 시에 모델을 상당 부분 변경해야 하는 오버헤드가 발생하는 경우가 많다. 거래처리에 대한 시스템이 이미 잘 구축되어 있는 기업의 경우에는 다양한 분석 업무에 활용하기 위한 목적으로 데이터웨어하우스를 구축하고 있다. 의사결정을 잘 지원하기 위해서는 대개 현시점의 데이터 뿐 아니라 과거 수십 년 동안의 데이터를 관리할 필요가 있다. 관계형 데이터베이스는 어떻게 설계하느냐에 따라 효율의 차이가 크게 나타나고 특히 관리해야 할 데이터의 용량이 점점 증가함에 따라 성능 향상을 위한 모델링은 중요한 이슈가 되고 있다.

본 논문에서는 관계형 데이터베이스 성능과 관련된 이슈들을 조사하였다. 성능 향상을 위한 모델링 방법들을 테이블, 릴레이션쉽, 인덱스 등의 객체별로 분류하여 정리하였고, Oracle DBMS를 이용하여 실험을 실시하였다. 실험에서는 대용량 테이블에 대한 질의를 수행하여 소요되는 시간을 측정하고 그 결과를 분석하였다. 실험을 통해 검증된 결과를 토대로 제안되는 모델링 방법을 제시한다.

2. 성능과 관련된 모델링 이슈

데이터 모델링 시의 튜닝 작업은 이후 가동 단계에서의 성능을 결정하는 중요한 작업이다. 성능 리뷰는 시스템 개발 생명주기의 여러 시점에 수행될 수 있지만 초기 단계의 리뷰에서 문제를 해결하는 것이 이후 단계에서 해결하는 것보다 용이하고 비용이 적게 든다. 구현된 시스템의 성능이 기대에 이르지 못하면 시스템에 대한 신뢰성이 저하되므로 시스템이 적정한 성능을 낼 수 있도록 데이터베이스를 설계하는 것은 아주 중요하다.

2.1. 테이블 관련 이슈

1) 집계 테이블(Aggregate Table)

시스템을 이용하는 현업의 요구는 점차 다양해지고 있다. 단순한 거래 처리를 지원하는 시스템보다는 분석 업무를 위하여 대용량의 데이터에 대한 집계성 또는 통계성 정보를 필요로 하는 조회 요건이 늘어나고 있는 추세이다. 이 경우 대용량 데이터에 대한 COUNT와 같은 질의(Query)가 많이 사용된다.

온라인으로 합계, 총건수 등의 집계성 정보를 조회하는 경우에는 통상적으로 많은 테이블과 대량의 레코드를 액세스해야 하는 시스템적인 부담이 발생하며 이는 곧바로 거래 처리에 영향을 미치게 된다. 따라서, 조회 대상이 되는 값을 미리 계산하여 별도의 테이블을 만들어 두는 것이 효율적이다.

예를 들어 은행의 지점별 총수신 현황에 대한

조회 요구가 자주 발생한다면, 조회 요청이 발생할 때마다 모든 수신 금액을 더하는 것보다는 수신 거래가 발생할 때 지점별 수신 금액을 누적하는 것이 보다 효율적일 수 있다. 수신 금액을 누적하는 것이 거래 처리시에는 부하를 줄일 수 있으나, 한 건의 질의에 대해 매번 수백만에서 수천만건의 데이터를 읽는 것에 비한다면, 수신 거래가 발생할 때 수신 금액을 합계 레코드에 기록해 두는 것이 보다 효율적일 것이다.

집계 테이블은 다음과 같은 경우에 추가하는 것이 좋다. 즉, 집계 데이터에 대한 조회 빈도가 높은 경우, 조회가 정형화되어 있는 경우, 데이터를 여러 부문에서 이용하는 경우, 데이터를 추출하는데 노력이 많이 드는 경우에는 조회 대상이 되는 값을 미리 계산하여 집계 테이블로 만들어 두는 것이 효율적이다. 그렇지 않은 업무에 대해서는 조회 요청 시에 온라인으로 모든 레코드를 읽어 처리할 수도 있다.

집계 테이블의 데이터를 생성하는 방법에는 온라인(On-line), 일괄처리(Batch), 지연처리(Deferred)가 있다. 먼저, 현 시점의 정확한 정보를 제공해야 하는 경우에는 원시 데이터가 만들어질 때 집계 데이터를 온라인으로 같이 생성해야 하는데, 이 경우 원시 거래의 처리 시간은 더 소요된다. 둘째, 거래의 부하가 문제가 되거나 현 시점의 정확한 데이터를 제공하지 않아도 되는 경우에는 트랜잭션의 부담을 줄이기 위해 일괄처리를 고려할 수 있는데, 이 경우에는 주기적으로 일괄처리 작업을 수행시켜야 하는 부담이 있다. 세째는, 온라인과 일괄처리의 중간 형태로서 원시 데이터가 생성된 후 시차를 두고 집계 데이터를 만드는 지연처리 방법이다. 지연처리는 원시 거래의 처리 시간을 단축시킬 수 있는 반면, 원시 데이터에서 집계 데이터로의 변환 프로그램을 작성해야 하는 부담이 있다.

2) 채번 테이블(Numbering Table)

채번(採番)이란 신규 레코드 삽입 시에 시스템에서 자동으로 키(Key) 값을 부여함으로써 사용자가 키 값을 입력하는 번거로움을 제거하여 시스템 사용의 편리성을 제고하기 위한 것이다. 예를 들어, 은행의 특정 지점에서 거래가 발생하면 지점별로 관리되는 최종 거래 번호에 1을 더한 번호가 거래 번호로 시스템에서 자동 부여된다.

채번을 구현하기 위한 방법으로는 로우(Row) 수가 적은 경우에는 간단히 테이블 전체 로우의 최종 값(Max)을 읽어 거기에 1을 더하는 알고리즘을 이용하면 되지만, 자동 채번의 빈도가 높고 로우 수가 많을 경우에는 레코드 삽입 시마다 테이블의 최종 값을 읽는 것이 부담이 되므로 별도의 채번 테이블을 갖는 것이 효율적이다.

채번 테이블은 동일한 채번 규칙을 갖는 일련번호를 기준이 되는 필드와 함께 하나의 테이블로 정의하는 것이다. 예를 들어 거래의 키가 지점별 일자별 일련번호인 경우 채번 테이블의 컬럼(Column)은 지점, 일자, 최종 일련번호로 구성된다.

실제로 거래가 발생하는 시점에는 채번 테이블을 먼저 액세스하여 최종 일련번호에 1을 더한 값으로 거래의 키를 부여하고 해당 레코드를 삽입한 후, 채번 테이블의 최종 일련번호를 그 값으로 변경한다.

3) 테이블 분할(Partitioning)

로우가 지나치게 많은 테이블은 조회 시 성능이 떨어질 뿐 아니라 신규 레코드 삽입 시에도 일정 데이터 용량 이후로는 성능이 급격히 저하

되며 테이블 백업 및 복구 시에도 관리 부담이 커지게 되므로 테이블을 관리 가능한 단위로 분할하는 것이 좋다. Oracle DBMS의 경우 통상 한 테이블에 수용 가능한 최대 로우 수가 3,000만건 정도로 알려져 있지만 500만건에서 1,000만건 이상이 되면 신규 레코드 삽입 시의 성능이 급격히 떨어지므로 유의한다.

테이블의 논리적 분할은 수평 분할(Split horizontally) 방법과 수직 분할(Split Vertically) 방법이 있다. 수평 분할은 전체 테이블을 로우 단위로 분할하는 것으로 대개 기간별, 부서별 등 동시 조회 빈도가 높은 단위로 분할한다. 수직 분할은 테이블 내의 컬럼을 구분하여 분할하는 것으로 대개 액세스가 빈번하게 발생하는 것과 아닌 것을 구분하여 분할하거나 컬럼의 성격을 구분하여 분할한다.

물리적 분할 시에는 DB2의 경우 테이블스페이스 분할(Tablespace Partitioning) 기능을, Oracle의 경우 분할 뷰(Partitioned View) 기능을 이용할 수 있다. DB2의 테이블스페이스 분할은 하나의 테이블을 물리적으로 다른 저장소(Data Set)에 보관하는 것을 의미하고, Oracle의 분할 뷰는 실제로는 여러 개의 다른 테이블을 정의하지만 액세스 시에는 하나의 테이블인 것처럼 인식하게 하는 것이다.

4) 코드 테이블 통합

논리 모델은 많은 코드 테이블을 포함하고 있으며 업무를 정확하게 나타내기 위해서는 코드 테이블 뿐 아니라 다른 테이블과의 릴레이션쉽까지 정확하게 표현하는 것이 바람직하다. 그러나 코드 테이블을 별도로 관리하는 경우에는 다음과 같은 성능상의 문제점이 예상된다. 즉, 관리대상 테이블이 증가하고, 레코드 삽입 시 관련된 모든 코드 테이블을 조회해야 하는 부담이 있으며, 코드값을 체크하는 로직과 코드 테이블 자체를 유지 보수하기 위한 프로그램이 다수 존재한다.

반면에, 다수의 코드 테이블을 통합하여 관리하게 되면 각 테이블마다 유지해야 하는 버퍼의 낭비를 줄일 수 있고, 코드가 메모리에 상주하게 될 가능성이 커지므로 코드값 조회 시의 성능이 향상되며, 코드값을 체크하는 로직은 공통 루틴을 만들어 사용할 수 있고, 코드 테이블 유지 보수 프로그램도 통합하여 하나만 작성하고 관리하면 된다.

대개 코드의 길이가 3 바이트 이하로 짧고 속성이 코드, 코드명 뿐인 테이블을 통합 대상으로 한다. 코드 통합 시에는 각각의 코드 테이블과 릴레이션쉽들을 제거하고 '코드유형' 테이블과 '코드' 테이블로 모델링한다. 릴레이션쉽을 연결하지 않는 대신, 해당 업무 테이블에 코드를 속성으로 추가하고 데이터의 정합성을 보장하기 위한 코드값 체크 알고리즘을 공통으로 정의하여 사용한다.

5) 서브타입(Subtype)

테이블 내의 대다수 레코드의 특성이 동일하고 부분적으로 다른 경우 서브타입을 이용하게 되는데, 서브타입은 업무를 이해하기 쉽게 표현해 줄 뿐 아니라 선택적인 속성 또는 릴레이션쉽을 필수적으로 변환해 주므로 업무 규칙을 보다 명확하게 표현해 준다. 서브타입의 구현 방법으로는 다음과 같은 것들이 있다.

첫째, 수퍼타입(Supertype)과 서브타입 전체를 하나의 테이블로 구현한다. 이 경우는 각 서브타입의 필드가 수퍼타입의 선택적 필드로 변환되므로 낭비 공간이 발생하게 된다.

둘째, 수퍼타입과 각각의 서브타입을 모두 분리한다. 즉, 수퍼타입과 각각의 서브타입을 별도의 테이블로 만들고, 수퍼타입과 서브타입간 일대일(1:1) 선택적 릴레이션쉽을 맺는다. 이때 서브타입의 키는 대개 수퍼타입의 키를 그대로 사용한다. 이 경우는 선택적 필드로 인한 낭비 공간은 최소화되지만 추가 테이블과 인덱스에 대한 유지 보수 부담이 발생하게 되고, 조회 시에 여러 테이블을 읽어야 하므로 성능이 저하된다.

세째, 위 두 가지의 중간 형태로서 각 서브타입을 수퍼타입의 공통 부분을 포함하여 분리하는 방법이 있다.

테이블의 로우 수가 많고 서브타입의 필드 수가 확장될 가능성이 있으며 서브타입 로우 수가 전체 로우 수의 10% 미만으로 적고, 디스크 공간이 문제가 될 때는 서브타입 분리를 고려할 수 있는데 이 경우 수퍼타입과 서브타입을 같이 조회할 때는 성능이 떨어지므로 유의한다. 디스크의 낭비 공간은 다음과 같이 계산할 수 있다.

[∑(number of supertype row) - ∑(number of subtype row)] * ∑(length of subtype field)

2.2. 릴레이션쉽 관련 이슈

1) 릴레이션쉽 제거

릴레이션쉽이 많은 테이블의 경우 그 테이블이 참조하고 있는 다른 테이블의 레코드들이 빈번하게 변경될 때 참조 무결성(Referential Integrity)을 보장하기 위하여 시스템적인 부담이 발생하게 된다. 또한 여러 업무 부문간 릴레이션쉽이 있는 경우 관련된 테이블에 대해서는 일괄적으로 데이터베이스 백업 및 복구가 이루어져야 하고 기존 데이터 이행 시에도 데이터 적재(Load)의 어려움이 발생하게 된다.

릴레이션쉽 제거 대상으로는 먼저, 여러 업무에서 공통으로 사용되는 테이블과의 릴레이션쉽이 있다. 예를 들어, 고객, 상품, 직원 등의 테이블과의 릴레이션쉽이 이에 해당한다. 또한, 업무별로 별도의 데이터베이스를 구성하는 경우 업무간 교차하는 릴레이션쉽도 제거하는 것이 좋다. 예를 들어, 여신, 수신, 회계 업무 영역간 교차하여 존재하는 릴레이션쉽은 제거한다.

릴레이션쉽을 제거할 경우에는 페어런트(Parent) 테이블의 키를 해당 테이블의 속성으로 추가하고 관련 프로그램에는 반드시 데이터의 정합성 체크 로직을 추가하여야 한다. 필요하다면 추가된 속성에 대한 인덱스를 추가할 수도 있다. 릴레이션쉽을 제거하고 추가하는 속성은 페어런트 테이블의 키의 이름, 데이터 타입, 길이 등 모든 특성을 동일하게 정의해야 한다.

2) 중복 릴레이션쉽 추가

릴레이션쉽 사슬(Chain)이 너무 길어 조회 시에 액세스 경로가 복잡해지는 경우에는 여러 개의 테이블을 읽어야 하는 부담이 발생하므로 자주 사용되는 경로에 대해서는 중복 릴레이션쉽을 추가하는 것이 좋다. 예를 들어, '고객' 테이블이 '거래내역' 테이블과 릴레이션쉽을 맺고 있고 '거래내역'은 다시 '미결제내역'과 릴레이션쉽을 맺고 있는 경우, 고객 이름별로 미결제내역을 조회하는 요건이 많다면 이를 위해 고객과 거래내역, 미결제내역 테이블을 액세스해야 한다. 이때 거래내역 테이블은 불필요하게 액세스하게 되는데, 고객 테이블과 미결제내역 테이블간 릴레이션쉽을 추가하게 되면 중복이 발생하지만 대신 액세스 테이블 수를 줄일 수 있다.

이 접근법은 실제로 중복을 허용함으로써 데이터의 정규화 규칙을 위배하는 것이다. 고객번

호가 모든 미결제내역 레코드에도 중복 관리됨으로써 불필요한 공간의 낭비를 초래할 뿐만 아니라 그보다 더 심각한 문제는 데이터 무결성의 유지가 어렵다는 것이다. 즉, 고객번호가 변경되면 미결제내역 테이블의 관련된 모든 레코드 상의 고객번호를 같이 변경해야 한다. 중복 릴레이션쉽을 추가할 때에는 데이터 정합성을 유지하기 위해 모든 데이터 변경 프로그램에 대해 지켜져야 하는 무결성 규칙을 정의하는 등 유의하여 사용한다.

3) 일대일(1:1) 릴레이션쉽

테이블의 선택적인 필드를 분리하여 일대일 릴레이션쉽을 연결하면 대개 공간의 낭비는 줄여주지만 조회 시에는 두 테이블을 액세스해야 하므로 성능이 떨어진다. 일대일 릴레이션쉽을 갖는 테이블의 통합은 디스크 공간의 낭비와 조회 시의 성능을 잘 고려하여 결정해야 한다.

다음과 같은 경우에는 테이블의 통합을 고려한다. 먼저, 일대일 릴레이션쉽을 갖는 두 테이블이 동시에 액세스되는 빈도가 높고 공간 낭비가 심각하지 않은 경우에는 두 테이블을 하나로 합하는 것을 고려한다. 다음으로, 일대일이면서 필수적인 릴레이션쉽을 갖는 테이블은 대개 두 테이블의 키가 같고 항상 두 테이블이 동시에 생성되므로 반드시 합하는 것을 검토한다.

일대일 선택적인 릴레이션쉽을 통합하기 위해서는 차일드-릴레이션쉽이 선택적인 쪽의 테이블-의 속성을 페어런트 테이블에 정의하고 속성을 모두 선택적으로 변경한다. 일대일 필수적인 릴레이션쉽을 갖는 테이블은 원래 정의한 속성의 선택성(Optionality)을 그대로 사용하면 된다.

4) 일대다(1:M) 릴레이션쉽

모델링 시에 제1정규화를 수행하여 반복 그룹을 별도의 테이블로 분리하면 대개 공간이 절약되고 차일드의 로우 값이 추가될 때 쉽게 대처할 수 있는 반면 조회 시에는 두 테이블을 액세스해야 하는 부담이 발생한다.

다음과 같은 경우에는 테이블의 통합을 고려한다. 즉, 차일드 테이블의 로우 수가 4개 이내로 제한되는 경우, 로우 수가 많더라도 항상 값이 필수적으로 들어오는 경우, 로우 수가 고정적이라 증가할 여지가 없는 경우, 두 테이블이 동시에 액세스되는 빈도가 높은 경우에는 페어런트와 차일드 테이블의 통합을 고려할 수 있다. 또한, 페어런트 테이블의 로우 수가 과다하게 많은 경우에도 정규화를 하게 되면 차일드 로우 수는 관리 불가능한 수치까지 증가하게 되어 입력 및 조회 시의 성능 저하를 가져올 수 있으므로 통합하는 것이 좋다.

일대다 릴레이션쉽을 통합하기 위해서는 차일드 테이블에서 여러 개의 레코드로 관리되던 항목들을 페어런트 테이블의 컬럼으로 정의하여 하나의 레코드로 관리한다. 이 경우 향후 관리해야 할 항목이 늘어나면 테이블에 해당 컬럼을 추가해야 하므로 데이터 구조를 변경해야 하는 부담이 발생한다.

2.3. 인덱스 관련 이슈

로우 수가 많은 테이블일수록 인덱스가 성능에 도움이 많이 되고, 로우 수가 적은 경우라도 조인(Join)이 자주 발생할 때에는 인덱스를 사용하는 것이 도움이 된다. 또한 SQL의 WHERE 조건문에서 자주 사용되는 컬럼에 대해서는 인덱스를 추가하는 것이 좋다.

인덱스를 추가할 때에는 다음 사항들을 염두에 두어야 한다. 첫째, 인덱스를 관리하기 위한 추가 공간이 필요하다. 둘째, 로우가 입력, 삭제

될 때나 인덱스에 대한 정보가 변경될 때 인덱스 유지 보수가 발생하게 되므로 인덱스가 없는 경우보다 속도가 늦어진다. 세째, 인덱스가 있는 테이블의 레코드 삽입은 인덱스가 없는 경우보다 더 많은 시간을 필요로 한다.

인덱스를 사용하여 조회하는 것이 항상 빠른 것은 아니다. 어떤 경우에는 인덱스를 사용하는 것이 더 느릴 수도 있다. 예를 들어, 데이터의 총건수를 조회하는 경우에는 전체 테이블을 스캔(Scan)하는 것보다 인덱스를 이용하는 것이 더 느릴 때가 있는데 이것은 테이블 데이터의 운반 단위가 인덱스 데이터의 운반 단위보다 크기 때문이다. 합성(Composite) 인덱스와 같이 인덱스 크기가 큰 경우에는 하나의 운반 단위에 포함될 수 있는 인덱스 수가 상대적으로 적어지므로 특히 더 느려지는 것에 유의한다.

2.4. 추출(Derived) 정보의 관리

추출 정보는 다른 속성값으로부터 도출되거나 계산되는 것이므로 원칙적으로는 데이터베이스에 관리할 필요가 없다. 그러나 그 정보를 추출하기 위한 루틴을 자주 이용하거나 정보를 추출하기 위해 여러 테이블의 조인이 발생하는 경우에는 검색 성능이 저하되므로 추출 정보를 관리하는 것이 좋다.

추출 정보의 관리 여부는 먼저 데이터의 변경 요건과 조회 요건 중에서 어떤 것이 더 많은지에 따라 결정될 수 있다. 추출 정보를 관리하지 않을 경우 데이터 정합성은 보장할 수 있지만 대개 조회 시의 성능 목표를 만족시킬 수 없게 되고, 조회 성능을 위하여 데이터를 중복시킨다면 데이터를 변경할 경우 여러 곳에 있는 데이터를 모두 수정해야 하는 부담이 발생한다. 또한 로우 수가 적을 때는 조인의 발생이 성능에 크게 영향을 주지 않지만 대용량 테이블인 경우에는 성능 상의 문제를 일으킬 수 있으므로 데이터의 용량도 추출 정보의 관리 여부를 결정하는 중요한 요인이 된다.

추출 정보를 관리하는 방법으로는 다음과 같은 것이 있다. 우선, 차일드 테이블에서 발생하는 정보의 합계나 잔액 등을 페어런트 테이블에 중복으로 갖는 방법이 있다. 예를 들어, 거래 테이블의 항목인 잔액을 계좌 테이블에서도 관리한다면 계좌만 읽어도 잔액을 알 수 있으므로 거래 테이블을 추가로 액세스할 필요가 없어 조회 성능이 좋아진다. 다음으로는, 차일드에서 검색 기준으로 자주 사용되는 속성이 페어런트에 있을 경우 기준이 되는 속성을 차일드에 중복으로 갖는 방법이 있다. 예를 들어 신용카드의 사용내역은 대부분 회원의 연령별 또는 카드 등급별(일반/우량/골드), 카드 종류별(BC/VISA/Master)로 조회하기 원한다. 이 경우 검색 조건이 되는 각 속성들은 회원 또는 신용카드 테이블에 존재하지만 카드사용내역 테이블의 속성으로 추가하면 조인을 발생시키지 않으므로 성능을 향상시킬 수 있다.

3. 실험 환경

3.1. 실험 개요

2장에서 우리는 성능 향상을 위해 데이터 모델링 시에 고려해야 하는 이슈들을 살펴 보았다. 3장에서는 조사된 내용 중 몇 가지 이슈를 선정하여 실제로 실험을 통하여 검증해 보고자 한다. 실험은 대용량 데이터에 대한 특정 질의

를 수행하여 성능이 급격히 저하되는 기준점을 찾아내는 방식으로 실시하였다.

실험은 COUNT, INDEX, MAX, JOIN, INSERT의 5가지에 대하여 실시하였다. 첫번째 실험(COUNT)에서 네번째 실험(JOIN)은 O/S가 Dynix/Ptx V2.1.7, 메인 메모리가 64 MB, 디스크 용량이 8.4 GB인 Sequent S2000/200 환경에서 실시하였다. DBMS는 Oracle 7.1.4을 사용하였고, 데이터베이스 버퍼 크기는 9,216,000 바이트이다. 선택되는 결과 로우 수를 변화시켜 가면서 조회 응답 시간을 측정하였는데 총 10회의 실험에서 첫번째와 여섯번째의 2회는 Cold Start-메모리를 Clear한 상태에서 질의 수행-를, 나머지 8회는 Warm Start-검색 결과가 메모리에 올라와 있는 상태에서 질의 수행-를 실시하여 각각의 평균을 산출하였다. 실제 업무에서는 검색 조건에 맞는 레코드가 항상 메모리에 올라와 있는 경우, 즉 Warm Start를 이용하게 되는 경우는 거의 없을 것이므로 Cold Start를 기준으로 결론을 도출하였다.

다섯번째 실험(INSERT)은 O/S가 HP/UX 10.20, 메인 메모리가 512 MB, 디스크 용량이 80 GB인 HP K460 환경에서 실시하였다. DBMS는 Oracle 7.2를 사용하였고, DB 버퍼 크기는 40,960,000 바이트이다. 레코드는 텍스트 파일로부터 읽어 테이블에 삽입하였으며 각 로우가 삽입될 때마다 Commit을 실행하였다.

3.2. 대상 테이블

실험1(COUNT)에서 실험4(JOIN)의 경우 신용카드 업무에서 사용되는 기본적인 데이터인 <회원>, <신용카드>, <카드사용내역>의 세 테이블을 대상으로 하였다. <표 3-1>은 각 테이블을 구성하는 인덱스 및 테이블과 인덱스의 크기를 나타내고 있다.

〈표 3-1〉 테이블 및 인덱스 크기

Table/Index	인덱스 구성	크기(Byte)
BC_BASE		31,457,280
BCBASEPK	BCH_CIDNO	6,297,600
BCBASE1	BCH_MBRNO+BCH_HOIKND +BCH_CIDNO+BCH_JIKNO +BCH_EOBCD	9,840,640
BCBASE2	BCH_OPNDT+BCH_CIDNO	6,574,080
BC_CARD		52,428,800
BCCARDPK	BCC_CIDNO+BCC_CDNO	11,571,200
BCCARD1	BCC_MBRNO+BCC_JIKNO	6,563,840
BCCARD2	BCC_CDNO	6,563,840
BC_USE		209,715,200
BCUSEPK	BCI_CIDNO+BCI_MONTH	36,700,160

표에서 BC_BASE는 신용카드를 소지하고 있는 회원에 관한 정보를 관리하는 테이블로서 크기는 31,457,280 바이트이고 BCBASEPK, BCBASE1, BCBASE2라는 세 개의 인덱스를 가지고 있다. BCBASEPK는 기본 키(Primary Key)로서 주민등록번호를 의미하는 BCH_CIDNO라는 컬럼으로 구성되며 크기는 6,297,600 바이트이다. BCBASE1은 BCH_MBRNO(회원관리점), BCH_HOIKND(회원종류), BCH_CIDNO(주민번호), BCH_JIKNO(가입권유자), BCH_EOBCD(업종코드)라는 5개의 컬럼으로 구성되는 인덱스로서, 크기는 9,840,640 바이트이다. BCBASE2는 BCH_OPNDT(가입일자)와 BCH_EOBCD(업종코드)의 두 컬럼으로 구성되는 인덱스이며, 크기는 6,574,080 바이트이다.

BC_CARD는 신용카드에 대한 정보를 관리하는 테이블로서 BCCARDPK, BCCARD1, BCCARD2라는 세 개의 인덱스로 구성되어 있다. BC_USE는 카드사용내역에 관한 정보를 관리하고 있으며 BCUSEPK라는 하나의 인덱스를 가지고 있다. 세 테이블을 합한 전체 크기는 293,601,280 바이트이고 인덱스의 크기는 84,111,360 바이트로서 테이블의 전체 크기는 인덱스의 약 3.5배이다.

〈표 3-2〉 BC_BASE DDL

```
CREATE TABLE  BC_BASE
  (BCH_CIDNO     CHAR(13)      NOT NULL,
   BCH_HOIKND    NUMBER(1)     ,
   BCH_HOINO     NUMBER(9)     NOT NULL,
   BCH_MBRNO     NUMBER(3)     NOT NULL,
   BCH_ACBRNO    NUMBER(3)     NOT NULL,
   BCH_ACKND     CHAR(2)       NOT NULL,
   BCH_ACSEQ     CHAR(7)       NOT NULL,
   BCH_STATUS    NUMBER(1)     NOT NULL,
   BCH_SETDT     CHAR(2)       NOT NULL,
   BCH_JIKNO     NUMBER(6)     ,
   BCH_JUMSU     NUMBER(3)     ,
   BCH_OPNDT     CHAR(8)       ,
   BCH_LSTDT     CHAR(8)       ,
   BCH_JUNO1     CHAR(13)      ,
   BCH_BIRGU     NUMBER(1)     ,
   BCH_BIRDT     CHAR(8)       ,
   BCH_SETGU     NUMBER(1)     ,
   BCH_EOBCD     CHAR(2)       ,
   BCH_BUBCD     NUMBER(1)     ,
   BCH_JIKCD     NUMBER(1)     ,
   BCH_CHICD     CHAR(2)       ,
   BCH_GPGU      NUMBER(1)     ,
   BCH_CARBT     NUMBER(1)     ,
   BCH_PCBT      NUMBER(1)     ,
   BCH_MARDT     CHAR(8)       ,
   BCH_FAMILBT   NUMBER(1)     ,
   BCH_BOKSUBT   NUMBER(1)     );
CREATE UNIQUE INDEX BCBASEPK ON BC_BASE
  (BCH_CIDNO                   ASC);
CREATE INDEX  BCBASE1 ON  BC_BASE
  (BCH_MBRNO                   ASC,
   BCH_HOIKND                  ASC,
   BCH_CIDNO                   ASC,
   BCH_JIKNO                   ASC,
   BCH_EOBCD                   ASC);
CREATE INDEX  BCBASE2 ON  BC_BASE
  (BCH_OPNDT                   ASC,
   BCH_CIDNO                   ASC);
```

BC_BASE(회원) 테이블은 총 25개의 컬럼으로 구성되어 있으며 전체 레코드 길이는 106 바이트이다. BC_BASE 테이블의 레이아웃과 구성 컬럼들의 상세한 내용을 나타내기 위하여 DDL(Data Definition Language)문을 출력하였다. BC_BASE 테이블에 대한 DDL문은 <표 3-2>와 같다.

BC_CARD(신용카드) 테이블은 총 24개의 컬럼으로 구성되어 있으며 전체 레코드 길이는 152 바이트이다. <표 3-3>은 BC_CARD 테이블의 DDL문이다.

〈표 3-3〉 BC_CARD DDL

```
CREATE TABLE  BC_CARD
  (BCC_CDNO     NUMBER(16)    NOT NULL,
   BCC_CIDNO    CHAR(13)      NOT NULL,
   BCC_MBRNO    NUMBER(3)     NOT NULL,
   BCC_HOIKND   NUMBER(1)     ,
   BCC_EOBCD    CHAR(2)       ,
   BCC_CDSTUS   NUMBER(1)     NOT NULL,
   BCC_CDKND1   NUMBER(1)     NOT NULL,
   BCC_CBKND2   NUMBER(1)     NOT NULL,
   BCC_CDKND3   NUMBER(1)     NOT NULL,
   BCC_CDKND4   NUMBER(1)     ,
   BCC_LNKCD    NUMBER(6)     ,
   BCC_DBLBT    NUMBER(1)     ,
   BCC_JIKNO    NUMBER(6)     ,
   BCC_CBRNO    NUMBER(3)     ,
   BCC_BALDT    CHAR(8)       ,
   BCC_ENDDT    CHAR(8)       ,
   BCC_CLODT    CHAR(8)       ,
   BCC_BCFDT    CHAR(8)       ,
   BCC_BCLDT    CHAR(8)       ,
   BCC_UJUNO    CHAR(13)      ,
   BCC_UJIKCD   NUMBER(1)     ,
   BCC_UCHICD   CHAR(2)       ,
   BCC_UNAME    VARCHAR2(20)  ,
   BCC_MBUSE    VARCHAR2(30)  );
CREATE UNIQUE INDEX BCCARDPK ON BC_CARD
  (BCC_CIDNO                   ASC,
   BCC_CDNO                    ASC);
CREATE INDEX  BCCARD1 ON  BC_CARD
  (BCC_MBRNO                   ASC,
   BCC_JIKNO                   ASC);
CREATE INDEX  BCCARD2 ON  BC_CARD
  (BCC_CDNO                    ASC);
```

BC_USE(카드사용내역) 테이블은 총 10개의 컬럼으로 구성되어 있으며 전체 레코드 길이는 85 바이트이다. BC_USE 테이블에 대한 DDL문은 <표 3-4>와 같다.

〈표 3-4〉 BC_USE DDL

```
CREATE TABLE  BC_USE
  (BCI_CIDNO      CHAR(13)       NOT NULL,
   BCI_MONTH      NUMBER(6)      NOT NULL,
   BCI_MBRNO      NUMBER(3)      ,
   BCI_HOIKND     NUMBER(1)      ,
   BCI_EOBCD      CHAR(2)        ,
   BCI_GAMT       NUMBER(12)     ,
   BCI_HAMT       NUMBER(12)     ,
   BCI_SAMT       NUMBER(12)     ,
   BCI_CAMT       NUMBER(12)     ,
   BCI_TOTAMT     NUMBER(12)     );
CREATE UNIQUE INDEX  BCUSEPK ON  BC_USE
  (BCI_CIDNO                    ASC,
   BCI_MONTH                    ASC);
```

실험5(INSERT)는 계좌 테이블을 대상으로 하였다. ACCOUNT(계좌) 테이블은 ACCOUNTPK, AOOUNT1, ACCOUNT2라는 세 개의 인덱스를 가지고 있다. ACCOUNTPK는 기본 키로서 계좌번호를 의미하는 ACNO라는 컬럼으로 구성된다. ACCOUNT1은 BRNO(관리점), ACKND(예금종류), SEQNO(일련번호)라는 세 컬럼으로 구성된다. ACCOUNT2는 BRNO(관리점)과 OPNDT(개설일자)의 두 컬럼으로 구성된다. 실험은 기본 키만 있는 테이블과 세 개의 인덱스를 가진 테이블에 대하여 각각 실시하였다.

ACCOUNT(계좌) 테이블은 총 12개의 컬럼으로 구성되어 있으며 전체 레코드 길이는 83 바이트이다. ACCOUNT 테이블의 레이아웃과 구성 컬럼들의 상세한 내용을 나타내기 위하여 DDL문을 출력하였다. ACCOUNT 테이블에 대한 DDL문은 <표 3-5>와 같다.

〈표 3-5〉 ACCOUNT DDL

```
CREATE TABLE  ACCOUNT
  (ACNO       CHAR(12)       NOT NULL,
   BRNO       CHAR(3)        NOT NULL,
   ACKND      CHAR(2)        NOT NULL,
   SEQNO      NUMBER(7)      NOT NULL,
   CIDNO      CHAR(13)       ,
   STATUS     CHAR(1)        ,
   OPNDT      CHAR(8)        ,
   LSTDT      CHAR(8)        ,
   LANDT      CHAR(8)        ,
   CURAMT     NUMBER(15)     ,
   DPYAMT     NUMBER(15)     ,
   LPYAMT     NUMBER(15)     );
CREATE UNIQUE INDEX ACCOUNTPK ON
ACCOUNT
  (ACNO                       ASC);
```

4. 실험결과

4.1. COUNT

2장에서 조사된 내용에 따르면 대량의 데이터에 대한 합계, 건수 등 집계성 정보를 자주 조회하는 경우에는 별도의 집계 테이블을 관리하는 것이 효율적이다. 이 실험은 집계 테이블이 필요한 기준점을 찾기 위한 것이다.

카드 소지자의 연령별로 카드사용 실적을 파악하는 것은 신용카드 업무의 중요한 조회 요건이다. 이 실험에서는 카드사용내역 데이터의 결과치를 증가시켜 가면서 데이터량이 어느 정도 되는 시점부터 집계용 테이블이 필요한지 찾아내었다. 전체 테이블의 크기는 209,715,200 바이트이고, 메인 메모리는 64 MB, 데이터베이스 버퍼 크기는9,216,000 바이트이다. 실행 SQL문은 다음과 같다.

```
SELECT COUNT(BCI_CIDNO) FROM BC_USE
WHERE BCI_CIDNO BETWEEN '7000001234567'
AND '7099991234567';
```

밑줄 부분을 변경시켜 가면서 데이터량을 조정하였고 모든 조건에 대해 같은 SQL을 이용하기 위하여 위의 경우에도 LIKE를 사용하지 않고 BETWEEN ... AND 문을 이용하였다. 실험 결과는 <표 4-1>과 같다.

〈표 4-1〉 COUNT에 대한 실험 결과

Selected Rows(%)	*Cold(초)	**Warm(초)
10.05 MB (4.79)	19.75	1.11
20.74 MB (9.89)	41.54	2.34
25.35 MB (12.09)	50.42	2.71
26.03 MB (12.41)	52.02	3.27
26.70 MB (12.73)	55.20	7.29
29.84 MB (14.23)	58.26	59.97
49.45 MB (23.58)	1:42.95	1:38.12

*Cold : Cold start

**Warm : Warm start

4.2. INDEX

<표 4-1>의 실험에서 대량의 데이터에 대한 건수를 조회할 때 집계 테이블을 추가할 필요없이 인덱스를 사용하면 더 빨라질 것이라고 생각할 수 있다. 이 실험은 인덱스를 사용하더라도 COUNT에 대한 성능 목표를 만족시키지 못한다는 것을 증명하기 위한 것이다.

신용카드를 소지하고 있는 전체 회원 수와 카드 수를 파악하는 것은 신용카드 업무의 기본적인 조회 요건이다. 이 실험에서는 전체 테이블을 스캔하는 경우와 인덱스를 사용하는 경우간 응답 시간의 차이를 파악하기 위하여 회원 수와 카드 수에 대한 COUNT를 여러 가지 방식으로 수행하였다. 먼저 BC_BASE(회원) 테이블에 대하여 전체를 스캔하는 경우와 기본 키인 BCBASEPK를 사용하는 경우, BCBASEPK를 사용하는 경우에도 세 가지 다른 SQL문을 이용하여 각각을 비교하였다.

전체 테이블의 크기는 31,457,280 바이트이고, 메인 메모리는 64 MB, 데이터베이스 버퍼의 크기는 9,216,000 바이트이다. 인덱스의 크기는 6,297,600 바이트로 테이블의 1/4.99배이다. 다음은 테이블 전체를 스캔하기 위한 SQL문이다.

```
SELECT COUNT(*) FROM BC_BASE;
```

두번째로는 다음의 SQL을 수행하였는데 이 SQL의 실행 결과는 첫번째와 거의 다르지 않았고 Query Optimizer의 실행 계획을 보여주는 Oracle의 EXPLAIN 기능을 이용하여 확인해 본 결과 첫번째 SQL과 마찬가지로 테이블 전체를 스캔하는 것으로 나타났으므로 실험 결과에서는 제외하였다.

```
SELECT COUNT(BCH_CIDNO) FROM BC_BASE;
```

다음은 Query Optimizer가 인덱스를 이용하도록 만들기 위하여 WHERE절에 의미없는 조건을 추가한 SQL문이다. WHERE절에 사용된 BCH_CIDNO(주민번호)는 BC_BASE 테이블의 기본 키로서 필수(NOT NULL) 필드이므로 BC_BASE 테이블의 모든 데이터는 자연히 WHERE 조건을 만족하게 된다. 따라서 다음 SQL은 위의 두 SQL과 동일한 결과를 산출하지만 전체 테이블을 스캔하지 않고 인덱스를 사용하게 된다.

```
SELECT COUNT(BCH_CIDNO) FROM BC_BASE
WHERE BCH_CIDNO > ' ';
```

이와 유사하게 인덱스를 이용하지만 COUNT

의 대상을 다르게 지정한 다음 SQL문에 대해서도 질의 결과를 측정하였다.

```
SELECT COUNT(*) FROM BC_BASE
WHERE BCH_CIDNO > ' ';
```

실험 결과는 <표 4-2>와 같다.

〈표 4-2〉 INDEX에 대한 실험 결과(1)

실행 SQL문	*Cold	**Warm
select count(*) from bc_base;	39.36초	38.61초
select count(bch_cidno) from bc_base where bch_cidno > ' ';	47.19초	5.05초
select count(*) from bc_base where bch_cidno > ' ';	40.08초	4.01초

두번째 실험에서는 카드 수를 조회하기 위하여 BC_CARD(신용카드) 테이블을 사용하였다. 여기서는 BC_BASE(회원) 테이블과는 달리 두 개의 인덱스를 대상으로 실험을 실시하였다. 하나는 기본 키인 BCCCARDPK로서 BCC_ CIDNO와 BCC_CDNO의 두 컬럼으로 구성된 합성 인덱스이다. 인덱스의 크기는 11,571,200 바이트로 테이블 크기의 1/4.53배이다. 다른 하나는 BCC_CDNO로만 구성된 BCCARD2 인덱스를 사용하였는데 크기는 6,563,840 바이트로 테이블 크기의 1/7.99배이다. 합성 인덱스인 BCCARDPK의 크기가 BCCARD2보다 크다. 전체 테이블의 크기는 52,428,800 바이트이고, 메인 메모리는 64 MB, 데이터베이스 버퍼 크기는 9,216,000 바이트이다.

먼저, 테이블 전체를 스캔하기 위한 SQL문이다.

```
SELECT COUNT(*) FROM BC_CARD;
```

다음은 기본 키인 합성 인덱스를 이용하게 하기 위한 SQL문이다.

```
SELECT COUNT(BCC_CIDNO) FROM BC_CARD
WHERE BCC_CIDNO > ' ';
```

세번째는 BCCARD2를 이용하게 하기 위한 SQL문이다.

```
SELECT COUNT(BCC_CDNO) FROM BC_CARD
WHERE BCC_CDNO > ' ';
```

실험 결과는 <표 4-3>과 같다.

〈표 4-3〉 INDEX에 대한 실험 결과(2)

실행 SQL문	*Cold	**Warm
select count(*) from bc_card;	26.89초	26.61초
select count(bcc_cidno) from bc_card where bcc_cidno > ' ';	1분 23.40초	1분 19.57초
select count(bcc_cdno) from bc_card where bcc_cdno > ' ';	10.18초	6.04초

4.3. MAX

2장의 내용을 보면 키(Key)로 이용하는 일련번호를 시스템에서 자동 부여하는 경우 대량의 데이터에 대해서는 채번 알고리즘을 이용하는 것보다 채번 테이블을 관리하는 것이 더 효율적이라고 조사되었는데 이 실험에서는 데이터량을 변화시켜 가면서 최종 값을 조회하는데 소요되는 시간을 측정함으로써 채번 테이블을 관리해야 하는 기준점을 파악하였다.

채번 테이블을 이용하지 않을 경우에는 레코드 삽입 시 전체 테이블 데이터 중에서 키 체계에 따른 최종 값(MAX)을 읽어 거기에 1을 더한

값을 키로 부여하게 되므로 레코드 삽입에 소요되는 시간은 최종 값을 조회하는 시간과 레코드를 실제로 삽입하는데 필요한 시간을 합한 시간이 된다. 이때 레코드 삽입 시간은 채번 알고리즘을 이용하는 경우와 채번 테이블을 이용하는 경우에 동일하게 적용되므로 최종 값을 조회하는 시간이 실제 레코드 삽입 시의 성능을 결정하는 주요 요인이 된다.

실험 대상이 되는 테이블의 전체 크기는 31,457,280 바이트이고, 메인 메모리는 64 MB, 데이터베이스 버퍼 크기는 9,216,000 바이트이다. 실행 SQL문은 다음과 같고 밑줄친 지점번호를 '505'부터 '560'까지 점차 증가시켜 가면서 실험을 실시하였다.

```
SELECT MAX(BCH_CIDNO) FROM BC_BASE
WHERE BCH_MBRNO BETWEEN 500 AND 505;
```

실험 결과는 <표 4-4>와 같다.

〈표 4-4〉 MAX에 대한 실험 결과

Selected Rows(%)	*Cold(초)	**Warm(초)
2.58 MB (8.21)	8.54	0.48
4.94 MB (15.69)	15.01	0.88
9.46 MB (30.06)	28.61	1.60
12.54 MB (39.86)	37.29	2.37
16.56 MB (52.64)	48.31	2.98
18.23 MB (57.94)	50.49	14.98
19.74 MB (62.74)	58.61	54.06
21.22 MB (67.46)	58.97	57.67

4.4. JOIN

2장에서 조사된 내용에 따르면 조회 시에 조인이 발생하는 것을 방지하기 위해 같은 데이터를 중복으로 여러 장소에 보관하는 것이 효율적일 수 있다. 이 실험은 추출 정보를 중복으로 관리해야 하는 기준점을 찾기 위한 것이다.

테이블 크기는 각각 6,297,600 바이트 및 36,700,160 바이트이고, 메인 메모리는 64 MB, 데이터베이스 버퍼 크기는 9,216,000 바이트이다. 같은 결과에 대해 조인이 발생하는 경우와 발생하지 않는 경우를 비교하기 위하여 두 가지 SQL문을 각각 실행하였는데 다음은 조인을 발생시키지 않는 SQL문이다.

```
SELECT COUNT(BCI_CIDNO) FROM BC_USE
WHERE BCI_CIDNO BETWEEN '7000001234567'
AND '7099991234567'
```

다음은 위와 동일한 결과를 산출하는 SQL문으로서 조인을 발생시키는 경우이다.

```
SELECT COUNT(BCI_CIDNO)
FROM BC_USE, BC_BASE
WHERE BCI_CIDNO = BCH_CIDNO AND
BCH_CIDNO BETWEEN '7000001234567'
AND '7099991234567'
```

이전 실험과 마찬가지로 밑줄 부분을 변경하여 데이터량을 변화시켜 가면서 결과를 측정하였다. 실험 결과는 <표 4-5>와 같다.

〈표 4-5〉 JOIN에 대한 실험 결과

Join Selectivity(%)	JOIN	*Cold(초)	**Warm(초)
4408*11010	X	0.39	0.06
(0.07*0.03)	O	0.44	0.09
44083*176160	X	1.84	0.43
(0.70*0.48)	O	4.69	0.70
124692*770703	X	1.90	1.84
(1.98*2.10)	O	13.89	2.22
294728*1820328	X	24.67	2.24
(4.68*4.96)	O	30.75	4.40
585047*3613131	X	40.86	24.01
(9.29*9.85)	O	1:09.28	50.07
875366*5405934	X	57.04	45.77
(13.90*14.73)	O	1:47.81	1:35.74
1319977*8154776	X	1:04.38	51.40
(20.96*22.22)	O	1:56.53	1:48.27
1421368*8837398	X	1:04.53	51.42
(22.57*24.08)	O	2:07.17	1:51.42

4.5. INSERT

2장에서 조사된 내용에 따르면 로우가 지나치게 많은 테이블은 조회 성능이 떨어질 뿐 아니라 신규 레코드 삽입 시에도 일정 용량 이후로는 성능이 급격히 저하되므로 테이블을 분할하는 것이 효율적이다. 이 실험은 대용량 테이블을 분할해야 하는 기준점을 찾기 위한 것이다. 이 실험에서는 신규로 생성된 테이블에 레코드를 125만건씩 누적해 가면서 삽입하는데 소요되는 시간을 측정하여 레코드 수가 어느 정도 되는 시점부터 테이블을 분할해야 하는지를 찾아내었다.

이 실험은 O/S가 HP/UX 10.20, 메인 메모리가 512 MB, 디스크 용량이 80 GB인 HP K460 환경에서 실시하였다. DBMS는 Oracle 7.2를 사용하였고, DB 버퍼 크기는 40,960,000 바이트이다. ACCOUNT(계좌) 테이블을 대상으로 하였는데 인덱스의 수에 따라 삽입 시간에 차이가 있을 수 있으므로 컬럼 구성은 동일하면서 인덱스가 하나인 테이블과 세 개인 테이블에 대하여 각각 실험을 실시하였다. 레코드는 텍스트 파일로부터 읽어 테이블에 삽입하였으며 각 로우가 삽입될 때마다 Commit을 실행하였다. 다음의 <표 4-6>은 ACCOUNTPK라는 기본 키만 가지고 있는 테이블을 대상으로 레코드 삽입 시간을 측정한 결과이다. ACCOUNTPK는 ACNO(계좌번호)라는 하나의 컬럼으로 구성되어 있다.

<표 4-6>에서 소요시간은 Insert Row수만큼의 레코드를 테이블에 차례대로 삽입하면서 각각에 소요된 시간을 측정한 결과이며, 누적 Row수는 레코드가 삽입됨에 따라 테이블에 누적되는 레코드 건수를 의미한다. 시간당 Insert건수는 삽입된 레코드의 분당 평균 건수와 초당 평균 건수를 각각 나타내고 있다.

〈표 4-6〉 INSERT에 대한 실험 결과(1)

Row수 (누적 Row수)	소요시간	시간당 Insert건수
1,250,055 (1,250,055)	90분	13,890건/분 (321건/초)
1,250,055 (2,500,110)	104분	12,019건/분 (200건/초)
1,250,055 (3,750,165)	102분	12,255건/분 (204건/초)
1,250,055 (5,000,220)	120분	10,417건/분 (173건/초)
1,250,055 (6,250,275)	121분	10,331건/분 (172건/초)
1,060,000 (7,310,275)	91분	11,648건/분 (194건/초)
1,250,055 (8,560,330)	116분	10,819건/분 (180건/초)
1,250,055 (9,810,385)	107분	7,757건/분 (129건/초)

〈표 4.7〉 INSERT에 대한 실험 결과(2)

Insert Row수 (누적Row수)	소요시간	시간당 Insert건수
1,250,055 (1,250,055)	54분	23,149건/분 (385건/초)
1,250,055 (2,500,110)	59분	21,187건/분 (353건/초)
1,250,055 (3,750,165)	56분	22,322건/분 (372건/초)
1,250,055 (5,000,220)	80분	15,625건/분 (260건/초)
1,250,055 (6,250,275)	64분	19,352건/분 (322건/초)
1,060,000 (7,310,275)	46분	23,043건/분 (384건/초)
1,250,055 (8,560,330)	55분	22,728건/분 (378건/초)
1,250,055 (9,810,385)	76분	16,448건/분 (274건/초)
780,000 (10,590,385)	47분	16,595건/분 (276건/초)

<표 4-7>은 <표 4-6>에서 사용된 동일한 테이블이 ACCOUNTPK, ACCOUNT1, ACCOUNT2라는 세 개의 인덱스를 가지고 있을 경우 레코드 삽입 시간을 측정한 결과이다. ACCOUNT1과 ACCOUNT2는 둘 다 합성 인덱스로서 ACCOUNT1은 BRNO(관리점), ACKND(예금종류), SEQNO(일련번호)의 세 컬럼으로 구성되어 있고 ACCOUNT2는 BRNO(관리점), OPNDT(개설일자)의 두 컬럼으로 구성된다.

5. 실험결과 분석

5.1. COUNT

4.1절의 실험 결과로부터 알 수 있는 사실은 첫째, COUNT에 대한 응답 시간은 선택된 로우 크기의 약 2배가 된다는 것이다. 이것을 식으로 표현하면 다음과 같다.

*Selected Row Size(MB)*2=Response Time(sec)*

두번째는, 선택된 로우 크기가 DB 버퍼 크기의 3배 이상이 되면, Cold Start와 Warm Start의 응답 시간 차이가 거의 없어진다는 것이다. 선택된 데이터량이 버퍼 크기의 2배가 될 때 Cold Start와 Warm Start의 차이가 없어질 것으로 예상되나 실험 결과는 3배로 나타났는데 이것은 인덱스를 이용하여 조회하였기 때문인 것으로 파악된다.

이 실험 결과에 따르면, 대용량 테이블에 대한 허용 가능한 조회 응답시간을 1분이라고 가정한다면, 메인 메모리가 64 MB이고 DB 버퍼 크기가 9 MB인 경우 선택된 로우 크기가 30 MB 이상이 되는 시점부터는 별도의 집계용 테이블을 추가하는 것이 좋다는 결론을 도출할 수 있다.

5.2. INDEX

실험 결과로부터 알 수 있는 사실은 첫째, 합성 인덱스가 A, B의 두 컬럼으로 구성되어 있는 경우 A와 A+B에 대해서는 인덱스를 사용할 수 있지만, B 컬럼만을 인덱스로 사용할 수는 없다는 것이다. 이것은 Query Optimizer의 실행 계획을 보여 주는 Oracle의 EXPLAIN 기능을 이용하여 확인하였다. 즉, 4.2절의 세번째 SQL문을 보면 BCC_CDNO가 WHERE 조건에서 사용되고 있지만 BCCARDPK 인덱스는 사용하지 않고 항상 BCCARD2 인덱스를 사용하게 된다.

둘째, <표 4-2>와 <표 4-3>의 실험 결과를 보면 대량의 데이터를 COUNT할 경우에는 인덱스를 사용해도 성능을 향상시키기 어렵다는 것을 알 수 있다. 특히 인덱스 크기가 큰 경우에는 전체 테이블 스캔보다 오히려 성능이 떨어지는 경우가 있다는 것이 실험적으로 증명되었다.

세째, <표 4-2>의 실험 결과를 보면 테이블의 크기가 인덱스의 약 5배일 때 전체 테이블을 스캔하는 경우와 인덱스를 사용하는 경우의 COUNT에 대한 응답 시간이 거의 동일하게 나타나고 있다. 이 결과로부터 테이블 데이터를 한번에 디스크로부터 읽어오는 운반 단위가 인덱스 운반 단위의 1/5 정도라는 것을 알 수 있다.

네째, <표 4-2>의 결과를 보면 동일한 인덱스를 사용하는 경우라도 COUNT 내의 변수를 다르게 했을 때 응답 시간이 크게 차이가 나는 것으로 나타난다. 그 이유를 파악하기 위하여 Oracle의 EXPLAIN 명령을 실행하였다. 결과는 다음 <표 5-1>에 나타나 있다.

〈표 5-1〉 표 4-2의 SQL에 대한 EXPLAIN 결과

```
select count(*) from bc_base

call   count    cpu    elapsed   disk    query    current    rows
-----  -----  -----  ---------  -----  -------  ---------  ------
Parse      1   0.01       0.01      0        0          0       0
Execute    1   0.00       0.00      0        0          0       0
Fetch      1   4.62      21.28   8091     8090          3       1
-----  -----  -----  ---------  -----  -------  ---------  ------
total      3   4.63      21.29   8091     8090          3       1

Misses in library cache during parse: 1
Optimizer hint: CHOOSE
Parsing user id: 10  (KNBCMS)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   OPTIMIZER HINT: CHOOSE
      0    SORT (AGGREGATE)
 147466      TABLE ACCESS (FULL) OF 'BC_BASE'

********************************************************************************
select count(bch_cidno) from bc_base where bch_cidno>' '

call   count    cpu    elapsed   disk    query    current    rows
-----  -----  -----  ---------  -----  -------  ---------  ------
Parse      1   0.00       0.00      0        0          0       0
Execute    1   0.00       0.01      0        0          0       0
Fetch      1   7.41      85.09   2743     2743          0       1
-----  -----  -----  ---------  -----  -------  ---------  ------
total      3   7.41      85.10   2743     2743          0       1

Misses in library cache during parse: 0
Optimizer hint: CHOOSE
Parsing user id: 10  (KNBCMS)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   OPTIMIZER HINT: CHOOSE
 147466    SORT (AGGREGATE)
 147467      INDEX (RANGE SCAN) OF 'BCBASEPK'
(UNIQUE)

********************************************************************************
select count(*) from bc_base where bch_cidno>' '

call   count    cpu    elapsed   disk    query    current    rows
-----  -----  -----  ---------  -----  -------  ---------  ------
Parse      1   0.00       0.00      0        0          0       0
Execute    1   0.00       0.00      0        0          0       0
Fetch      1   3.97       4.05      1     2743          0       1
-----  -----  -----  ---------  -----  -------  ---------  ------
total      3   3.97       4.05      1     2743          0       1

Misses in library cache during parse: 0
Optimizer hint: CHOOSE
Parsing user id: 10  (KNBCMS)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   OPTIMIZER HINT: CHOOSE
      0    SORT (AGGREGATE)
 147467      INDEX (RANGE SCAN) OF 'BCBASEPK' (UNIQUE)
```

위의 EXPLAIN 결과를 보면, 첫번째 SQL에 대해서는 Query Optimizer가 전체 테이블을 스캔한 결과를 COUNT하고 두번째 SQL에 대해서는 일단 인덱스를 범위 스캔(Range Scan)하고 그 결과를 COUNT 하기 위해 다시 한번 결과 로우 수만큼 액세스한다. 세번째 SQL에서는 두번째 SQL과 마찬가지로 인덱스를 범위 스캔하지만 COUNT 시에는 별도의 로우 액세스가 필요없다는 것을 알 수 있다.

따라서 우리는 다음과 같은 결론을 내릴 수 있다. 첫째, 합성 인덱스의 경우 두번째 컬럼만을 인덱스로 사용할 수는 없으므로 자주 쓰이는 컬럼의 순으로 인덱스를 구성해야 한다. 둘째, COUNT 수행 시에 검색 조건에 맞는 테이블 로우의 크기가 인덱스 크기의 5배 이하인 경우에는 인덱스를 사용하지 않는 편이 성능이 더 좋아진다. 세째, COUNT 내의 변수를 특정 컬럼으로 주지 않고 전체(*)로 하게 되면 결과 로우를 다시 액세스하지 않으므로 성능이 좋아진다.

5.3. MAX

실험 결과로부터 알 수 있는 사실은 첫째, MAX에 대한 응답 시간은 선택된 로우 크기의 약 3배가 된다는 것이다. 이것을 식으로 표현하면 다음과 같다.

*Selected Row Size(MB)*3=Response Time(sec)*

두번째는, 선택된 로우 크기가 DB 버퍼 크기의 2배 이상이 되면, Cold Start와 Warm Start의 응답 시간 차이가 거의 없어진다는 것이다.

따라서, 대용량 데이터에 대한 허용 가능한 신규 데이터 삽입 소요시간을 30초라고 가정한다면, 메인 메모리가 64 MB이고 DB 버퍼 크기가 약 9 MB인 경우 채번의 기준 대상이 되는

선택된 로우 크기가 10 MB 이상이 되는 시점부터는 별도의 채번용 테이블을 추가하는 것이 좋다는 결론을 도출할 수 있다.

5.4. JOIN

실험 결과로부터 알 수 있는 사실은 먼저, 조인이 발생하는 경우와 발생하지 않는 경우간 응답 시간의 차이가 존재한다는 것이다. 그리고 조인 선택율(Join Selectivity)이 5% 이하인 경우에는 조인이 발생하는 경우와 발생하지 않는 경우가 10초 이하의 차이를 보이고 있다. 특히 1% 이하인 경우에는 응답 시간이 둘 다 5초 이내, 2-5%인 경우는 30초 이내로 나타났다. 조인 선택율이 10-15%인 경우에는 30초 이상의 차이를 보이면서 조인이 발생하지 않는 경우는 응답 시간이 1분 이내이지만 조인이 발생하는 경우는 1분을 초과하고 있다. 조인 선택율이 20% 이상인 경우에는 1분 정도의 차이를 보이면서 조인이 발생하는 경우의 응답 시간은 2분을 초과하고 있다.

따라서, 대용량 데이터에 대한 허용 가능한 조회 응답 시간을 1분 정도라고 가정한다면, 메인 메모리가 64 MB이고 DB 버퍼 크기가 약 9 MB인 경우 크기가 각각 6 MB와 36 MB 정도 되는 두 테이블의 조인 시에는 조인 선택율이 10%가 되는 시점부터는 추출 정보를 중복으로 관리하는 것이 좋다는 결론을 도출할 수 있다.

5.5. INSERT

4.5절의 두 가지 실험 결과로부터 알 수 있는 사실은 첫째, 메인 메모리가 512 MB인 경우 1-3개의 인덱스를 가진 테이블에 레코드를 삽입할 때 건수가 1,000만건 이상이 되면 성능이 급격히 저하된다는 것이다. <표 4-6>에서 시간당 레코드 삽입 건수를 보면 7번째까지는 대부분 초당 350건을 상회하다가 980만건 이상부터 270건대로 급격히 떨어지는 것을 확인할 수 있다. 마찬가지로 <표 4-7>의 실험 결과에서 시간당 레코드 삽입 건수를 보면 7번째까지는 초당 170건을 상회하다가 980만건 이상부터 130건 이하로 급격히 떨어지는 것을 확인할 수 있다.

둘째, 인덱스가 세 개인 테이블의 레코드 삽입 시간은 인덱스가 하나인 테이블의 60% 정도가 된다는 것을 알 수 있다. <표 4-6>에서는 초당 삽입 건수가 평균 334건인데 <표 4-7>에서는 197건으로 인덱스가 하나인 경우의 60% 이하로 나타났다.

따라서, 메인 메모리가 512 MB인 경우 1-3개의 인덱스를 가진 테이블의 로우 수가 1,000만건 이상일 때는 테이블을 분할하여 관리하는 것이 더 효율적이라 할 수 있다.

6. 결 론

본 논문에서는 대용량 데이터베이스를 위한 데이터 모델링 시 고려해야 하는 성능 관련 이슈들을 조사하였고 몇 가지 실험을 통해 검증된 모델링 방법을 다음과 같이 제시하고 있다.

첫째, 대량의 데이터를 가진 테이블에 대하여 요약 정보를 자주 액세스하는 경우에 대해서는 별도의 집계용 테이블로 모델링하는 편이 좋다.

둘째, 요약 테이블을 두지 않고 검색 성능을 향상시키기 위하여 인덱스를 이용할 수도 있지만 때에 따라서는 인덱스를 사용할 수 없는 경우도 있으므로 세심한 주의가 필요하다. 인덱스를 이용할 경우에는 SQL문장 내의 변수라든가

컬럼의 순서 등이 성능에 큰 영향을 미칠 수 있으므로 유의하여 사용한다.

세째, 대용량 테이블의 키(Key)에 대해 시스템에서 자동 채번(Numbering)을 해야 하는 경우에는 로직으로 처리하지 말고 별도의 채번 테이블을 두는 것이 바람직하다.

네째, 여러 테이블을 조인해서 결과치를 돌려줘야 하는 경우에는 조인 선택율이 10% 이상이 되는 시점부터는 추출(Derived) 정보를 별도로 관리하는 편이 좋다.

마지막으로, 전체 로우 수가 1,000만건 이상인 테이블에 대해서는 신규 로우 삽입을 고려하여 1,000만건 정도의 단위로 테이블을 분할하여 관리하는 것이 좋다.

본 연구에서는 관계형 데이터베이스, 특히 대용량 데이터베이스를 설계하면서 직면할 수 있는 성능 관련 이슈들을 정리하였고 그에 대한 일반적인 모델링 방법을 제시하고 있다. 뿐만 아니라 각각의 방법에 대하여 실험을 통하여 검증된 구체적인 모델링 기준을 제시하고 있으므로 실제로 관계형 데이터베이스를 구축하고자 하는 설계자에게 도움이 될 것으로 기대된다.

참 고 문 헌

[1] 이화식, 대용량 데이터베이스 솔루션, 대청정보시스템(주), 1996.

[2] Advanced Data Modeling, Texas Instruments Inc., 1992.

[3] DB2 Performance Guide, 포항종합제철주식회사, 1992.

[4] Designing for High Performance, Texas Instruments Inc., 1995.

[5] Joyce Bischoff, Ted Alexander, Data Warehouse: Practical Advice from the Experts, Prentice Hall, 1997.

[6] Laura M. Haas, et al, Seeking the truth about ad hoc join costs, The VLDB Journal, 6: 241-256, 1997.

[7] Mark Gurry and Peter Corrigan, Oracle Performance Tuning, O'Reilly & Associates, Inc., 1996.

[8] Michael C. Reingruber and William W. Gregory, The Data Modeling Handbook, A Wiely-QED Publication, 1994.

[9] Oracle Korea Customer Support Technical Bulletins, Oracle Korea, 1994.

[10] POS-IEM(VOL.2-2, 2-3, 2-4, 3), 포스데이타 주식회사, 1994.

[11] Reviewing Your BAA Model for Performance, Texas Instruments Inc., 1993.

[12] Sam Anahory, Dennis Murray, Data Warehousing in the Real World: A practical Guide for Building Decision Support Systems, Addison-Wesley, 1997.

[13] Sean Kelly, Data Warehousing in Action, John Wiley & Sons Ltd., 1997.

[14] Vidette Poe, Building a Data Warehouse for Decision Support, Prentice-Hall, Inc., 1996.

[15] W.H. Inmon, Building the Data Warehouse, John Wiley & Sons, Inc., 1996.