White Paper

embarcadero

성능 좋은 SQL 작성법

작성자 : Jonathan Lewis (Embarcadero Technologies)

2009 년 9 월

이 기술 백서는 2010 년 6 월 19 일에 진행된 엠바카데로의 라이브 웹케스트 세미나의 부록입니다. 세미나 슬라이드는 모든 참석자에게 행사 후에 전달될 것이며 이 문서 뒷부분에도 첨부되어 있습니다. 이 기술 백서 내용의 대부분은 이미 2009 년 UKOUG 컨퍼런스에서 발표된 내용이며 조나단 루이스 (Jonathan Lewis)가 지난 5 년간 1 일 과정으로 진행하는 "최적의 SQL 문 작성법(Writing Optimal SQL)" 과정의 교재를 근간으로 하였습니다.

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California
94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Devgear
서울특별시 반포 1 동 743-14
4 층 ㈜데브기어
(T) 02.595. 4288

도입

성능 좋은 SQL문을 작성하려면, 다음과 같은 사전 지식이 있어야 한다. 먼저 "내 데이터가 어떤 모습인가", 그리고 "해당 데이터베이스의 스토리지 메커니즘을 통해 내가 직접 효율적인 SQL의 경로를 예측 또는 판단할 수 있는가", 마지막으로 "옵티마이저가 액세스 경로를 실제로 선택할 때 내가 생각했던 최적의 경로를 올바르게 선택할 수 있는가" 하는 것이다.

무엇보다 가장 중요한 지식은 "데이터베이스로부터 얼마나 많은 데이터를 가져와야 하고 이것들을 어디에서 찾아올 것인가" 이다. 다시 말해, 우리가 사용할 데이터의 규모와 분산 정도를 분석할 수 있어야 한다. – 이것은 옵티마이저가 파악하고자 하는 분석 요소이기도 하며, ***dbms_stats*** 패키지를 통해 수집할 수 있는 통계 정보이다.

하지만 불행히도, 옵티마이저가 판단하는 "데이터에 대한 그림"이 항상 올바른 것은 아니다. 옵티마이저는 영업시스템의 "평균적인"고객사가 연간 40개의 주문을 한다고 (거의 정확하게) 판단한다; 하지만 우리는 이런 "순수 평균"에 대해 잘 모르고 있을 수도 있다. 왜냐하면, 고객사 중 20개가 매출의 80%를 차지하고, 다른 1,000개의 고객사가 매출의 15%를, 그리고 나머지 80%의 고객사가 마지막 5%를 차지하기 때문이다.

Oracle의 쿼리 옵티마이저가 나쁜 계획을 만드는 많은 경우는 주로 "데이터가 어떤 모습인가"에 대한 Oracle의 의견이 우리의 지식과 다를 때이다. (여기에서는 이 문제를 자세히 다루지는 않을 것이다. 하지만, 좋은 실행 계획을 만드는 최고의 방편 즉, 우리 데이터가 진짜로 어떻게 생겼는지를 옵티마이저에게 알려줄 수 있는 작은 프로그램을 만들어서 "인공적이지만, 진실한"통계를 주어주는 것이 ***dbms_stats*** 패키지를 통해 통계를 가져오는 것보다 더 나은 경우가 종종 있다.)

인덱스 구성을 잘하려면 (GOOD INDEXING)

우리가 가져오려는 데이터를 잘 알고 있고, 해당 쿼리가 어떤 경로로 가져와야 하는 지를 안다면, 우리가 원하는 효율적인 경로를 옵티마이저가 선택할 것이라는 점을 확신할 수 있어야 한다. 이를 위해 우리는 전형적으로 인덱스들이 적절하게 구성되어 있는 지를 확인한다. 하지만 각 테이블의 인덱스들을 알맞게 설계한다는 것은 사소한 문제가 아니라, 데이터가 어떻게 사용되는 지에 대한 방대한 지식을 요구하는 작업이다. (물론, 인덱스 구성만이 데이터의 효율적인 접근을 확보하는 유일한 방법은 아니다. 하지만, 일단 실제 운영환경으로 가고 난 후라면 이것이 성능 문제를 해결하기 위한 유일한 가변 옵션이 된다.)

슬라이드 9페이지에는 (*슬라이드는 이 문서 뒤에 있음*) 데이터베이스의 모든 인덱스 정의에 대한 리스트를 가져오는 간단한 쿼리가 있다. 물론 독자는 이 쿼리를 확장하여 훨씬 많은 정보를 가져올 수 있을 것이다. 예를 들면, 테이블 크기(로우와 블록), 인덱스, 각 컬럼별 고유 값 데이터와 NULL의 갯수, "함수 기반"인덱스가 사용하는 식(expression) 등이 될 것이다.
(https://jonathanlewis.wordpress.com/scripts 의 블로그에서는 보다 정제된 복잡한 예문들을 찾을 수

있다). 이런 간단한 쿼리만으로도 테이블의 인덱스 문제는 확연히 밝혀진다 – 특히 (a)리소스 낭비나 (b)옵티마이저가 잘못된 인덱스를 선택할 가능성과 같은 문제의 경우이다.
포린키: 상식적으로 "포린키 락킹(locking)" 위험이 있다면 포린키 제약 조건에 상응하는 인덱스 생성에 대해 **생각해볼 필요**가 있다 – 하지만, 모든 포린키가 인덱스로 되어야 한다거나 각각의 포린키 제약조건에 정확히 일치해야 한다는 의미는 아니다. 만약 부모 로우 (parent row)를 삭제하거나 부모 키 (parent key)를 업데이트 하지 않는다면 락킹(locking) 관련 이유로 인덱스를 만들 필요가 없다 – 물론 몇몇 포린키 인덱스는 정교한 액세스라는 장점을 잘 활용하는 좋은 인덱스가 될 수도 있다.

포린키와 관련된 락킹 문제를 방지할 목적이라면, 인덱스가 단지 원하는 컬럼으로 ***시작되기만*** 하면 된다; 인덱스의 선행 컬럼(들)이 매우 빈번하게 반복적으로 사용될 가능성이 상당히 높다면, "***compress***"옵션을 사용하여 사이즈를 줄이는 것도 고려할 수 있다. (인덱스 압축(compression)은 CPU 사용을 다소 증가시킬 수 있지만, 인덱스의 사이즈를 줄임으로써 해당 인덱스와 관련된 I/O의 양을 줄여주는 효과가 있으므로 CPU사용과 맞바꿀만한 상당한 가치가 있다).

슬라이드의 9페이지에 있는 사례는 명명 규칙에 따라 포린키 인덱스 명에서는 FK를 포함하고 있다 (PK는 프라이머리키, UK는 유니크 키); 여기에서 "(grp_id) 컬럼 만으로 된 포린키 인덱스는 기술적으로는 불필요한 중복임을 알 수 있다. 왜냐하면 이미 (grp_id, role_id) 컬럼으로 구성된 인덱스가 grp_id 컬럼으로 시작되기 때문이다; 유사한 인덱스로 각 (org_id)와 (per_id) 또한 불필요한 중복이다. 역시 마찬가지로 (org_id, ap_id) 와 (per_id, ap_id)가 존재하기 때문이다. 다만, 이런 불필요한 인덱스를 제거하기 전에는 혹시라도 이 컬럼이 어마어마하게 빈번히 사용되어서 성능 문제를 일으키지는 않는가에 대해 한번쯤 주의를 기울여서 보아야 한다. 한편 (per_id)의 경우에는 옵티마이저가 이 인덱스를 대신하여 (per_id, ap_id) 인덱스를 사용하지 않을 확률이 있다. 왜냐하면 리프-블록의 개수가 더 많거나 ***clustusing_factor*** 가 훨씬 비싸다고 판단할 수도 있기 때문이다. 따라서 인덱스를 제거하기 전에는 이러한 통계값들의 상대적인 사이즈 비교를 할 필요가 있다. 그리고 나서 통계자료를 수집할 때면 간단한 스크립트를 작성하여 (특히) 더 큰 인덱스에 대해 ***clustusing_factor*** 를 조정해 주는 것이 좋다.

(role_id) 인덱스는 불필요한 중복이라고 할 수 없다. 왜냐하면 대신할 수 있는 다른 인덱스가 없고, 락킹 문제를 걱정할 필요가 없다는 점을 확실히 알지 못하기 때문이다; 하지만, ***role_id*** 가 매우 반복적으로 사용되는 것을 알게 된다면, 압축된 인덱스로 새로 만들 수도 있다. 다시 슬라이드로 돌아가서 불필요하게 중복된 포린키 인덱스를 대신해줄 수 있는 인덱스들을 좀더 자세히 보면 이들 중 (유니크한 ap_id를 포함한) 2개에서는 첫 번째 컬럼 만을 그리고 (grp_id, role_id)는 두 개의 컬럼 모두를 압축하기로 결정할 수 있다.

마지막으로 살펴볼 인덱스는 함수 기반 (function-based)인덱스로써 "last update date"에 대한 인덱스 이다 – 날짜 이하는 잘라버린다. 이 인덱스에 대해서는 몇 가지 고려 사항이 있다. 첫째로, 특정 날짜는 이 커다란 테이블에 넓게 흩어져있는 어마어마한 데이터를 식별하는 기준이 된다 – 잘 사용되는 유용한 인덱스이다;

두 번째로, 모든 날짜는 각자 많은 로우를 식별하게 되므로 압축의 효과를 볼 수 있는 인덱스 유형이다;
세 번째로 데이터가 업데이트 되면 로우는 인덱스의 "낮은(low)"쪽 끝에서 삭제되고, "높은(high)"쪽 끝에 삽입된다 – 이 인덱스는 아마도 상당부분을 비워둠으로써 해당 범위에서 크게 발생되는 조각화(fraction)를 대비하고 최근 몇 일간에 대해서 집중적으로 채워진 상태를 유지할 수 있어야 한다. 따라서 이 인덱스에는 아마도 ***coalesce*** 명령을 사용할 필요가 있다. (이 특별한 방식의 인덱스와 이와 관련된 문제에 대한 보다 자세한 생각은 내 블로그의 "Index Explosion" 항목을 참고하기 바란다).
현실을 직시해 보자. 애플리케이션 생명주기(lifecycle) 상에서 데이터베이스 개발자와 DBA 의 역할은 점점 커지고 있다. 경영진, QA, 개발팀, 파트너들 모두가 성공적으로 업무를 수행하려면, 이렇게 연결된 환경 내에서 서로 협력하고 커뮤니케이션 할 수 있는 도구가 매우 유용하다. 예를 들어, 어떤 문제가 발생했을 때 신속하게 문제점를 찝어 내고 경영진, 개발팀, QA 와 동일한 페이지를 보면서 커뮤니케이션 할 수 있는가? SLA(서비스 수준 계약) 준수를 증명하기 위한 리포트를 얼마나 빠르게 작성할 수 있는가? 몇번의 클릭만으로 프로젝트들을 중앙 버전 관리 시스템 안에 넣을 수 있는가? 개발 환경과 실제 운영 환경 사이에 진행되는 스키마 변경을 얼마나 빠르게 비교하고 커뮤니케이션 할 수 있는가? 갑자기 맡게 된 업무에 관련된 애플리케이션을 어떻게 리버스 엔지니어링 할 것이가? 데이터베이스 도구를 사전 검증할 때는 각 도구들이 어떻게 연관되고 대내외 관계자들과의 협업과 커뮤니케이션을 어떻게 지원하는 지를 고려하자.

데이터 파악 (KNOWING THE DATA)

인덱스 들에 대한 정확한 그림을 만들고 나면, (우리가 원하는 것은) 내 쿼리가 효율적으로 작동할 수 있게 이끌어 줄 인덱스 몇 개를 찾아서 검토하게 된다. 이때 우리는 데이터를 잘 보아야 한다 – 데이터 양이 얼마나 많은가? 이 인덱스를 사용하여 가져온다면 얼마나 많은 작업이 필요한가? 만약 우리가 우리의 데이터가 어떤 모양인지를 사전에 정확히 알지 못한다면 단순한 몇 가지 쿼리를 활용할 수 있을 것이다. "단순"하다는 것이 빠르다거나 수행비용이 싸다는 것을 의미하는 것은 물론 아니다.

슬라이드 10에 있는 예제 쿼리는 특정 컬럼의 데이터 중 고유한 값 별로 얼마나 많은 로우가 있는지를 찾아낸다 – 다양한 ***sample*** 절의 예에서 큰 테이블에서 사용할 수 있다. 이것은 적은 수의 고유한 값을 (254개까지, 만약 히스토그램을 생각하고 있다면) 가진 컬럼의 경우라면 즉시 도움이 된다. 그러나, 10,000개의 고유한 값을 가지는 경우라면 이 작은 데모의 경우가 많이 사용되지는 않는다.

하지만, 우리는 이 쿼리를 다듬어서 사용할 수 있다 – 예를 들어 값2와 3은 각각 12개의 로우에 들어 있기 때문에 우리는 12개를 가진 또 다른 고유한 값이 얼마나 많은 지가 궁금할 것이다. 이것은 슬라이드 11에서 볼 수 있다. 집계된 값을 살펴보면 해당 컬럼에서 동일한 값 하나가 22개의 로우에 들어 있는 이 예제 중에서 최악의 경우를 찾을 수 있다. 아마도 우리는 이 22개 로우를 바탕으로 쿼리를 최적화할 것이다 (또는 적어도 이 값을 가진 로우를 찾아와야 하는 잠재적 위협을 방지할 수 있는 쿼리를 작성할 것이다)

하지만, 우리가 알아야 하는 것은 데이터 양 뿐만이 아니다 – 우리는 쿼리를 다시 다듬어서 데이터가 얼마나 흩어져 있는지를 볼 수도 있다. 고유한 값 별로 로우의 수를 세고 나서 합계를 구하는 것 대신, 값 별 **블록**의 수를 세어서 합계를 낸다. 이렇게 하면 우리는 가장 나쁘게 "분산되어 있는 (scattering)" 것들을 찾을 수 있다. 여기에는 2개의 고유한 값을 가져오기 위해 19개의 서로 다른 블록을 액세스하여야 하는 경우가 보인다. (***index()*** 힌트는 이 쿼리를 최적화할 때 유효한 방법이다. 왜냐하면 ***colX***는 프라이머리 키 ***t1_pk***의 일부이기 때문이다)

우리는 동일한 분석을 인덱스에 적용할 수 있다. 슬라이드 13에 있는 쿼리는 ***dbms_stats*** 패키지에서 간단한 B-트리 인덱스 통계를 수집할 때 사용되는 것이다. 그리고 여기에서 ***sys_op_lbid()*** 함수를 (lbid = leaf block id) 꺼내어 동일한 "double aggregation" 메소드를 사용하는 쿼리를 컬럼 데이터에 적용하였다. – 첫 째 우리는 각 리프 블록에 있는 인덱스 엔트리의 개수를 셀 수 있다. 그리고 나서 엔트리 개수가 동일한 블록의 합계를 내어 인덱스의 내부적인 스토리지 효율성에 대한 그림을 얻었다 – 이를 통해 우리가 이 인덱스를 사용한다면 얼마나 많은 리프 블록을 읽어야 하는지를 알 수 있다. (우리가 인덱스 키가 얼마나 커서 블록당 얼마나 많은 로우가 있는지를 알 수 있다면, "*랜덤 액세스(random arrival)*"로부터 가져오는 이항 분산의 기준으로 70%가 전형적이라고 볼 때, 해당 인덱스의 "*건강(health)*"에 대한 단서 또한 파악할 수 있다. 슬라이드 15의 "smashed" 인덱스의 사례의 경우, 인덱스에 많은 빈 공간을 남겨두는 프로세스를 생각해 볼 필요가 있다는 단서를 찾을 수 있고 왜 이런 일이 일어나는지, 이에 대해 대응이 필요한지 아닌지에 대해 생각해보게 될 것이다.

그림 작성 (DRAWING THE PICTURE)

데이터에 대한 정보를 수집하는 것이 첫 번째 단계이다. 이 정보는 쿼리의 현명한 경로를 만들어 내려고 할 때 필요하다. 하지만 이 정보를 적용할 수 있으려면 먼저 쿼리와 수집한 정보를 맞춰볼 수 있는 방법이 있어야 한다. 파워포인트 한 장으로 실질적인 의미들을 모두 담는 것은 아무래도 무리가 있으므로, 몇 장으로 나누어 펼쳐서 내용을 현격히 제한적인 내용을 표현하였다 – 실제로는 아마도 큰 종이를 가지고 시작하여 쿼리에 대한 스케치를 두세 번 시도해야 적합한 레이아웃을 얻을 수 있을 것이다.

여기의 데모 쿼리는 5개 테이블을 조인하고 3개 테이블에 대한 서브쿼리를 가지고 있다. 이 쿼리는 실제로 흔하면서도 곤란할 수 있는 쿼리이다. 이 쿼리를 한 장의 그림으로 바꿔놓기 위해 각 ***from*** 절에 있는 각 테이블을 하나하나를 ***where*** 절에서 눈을 떼지 않으면서 쫓아가면서 네모 상자로 표시했다. 서브쿼리는 점선 테두리 상자 안에 그렸고, 서브 쿼리와 관련되어 조인 테이블들을 연결했다. "까마귀발(crow's feet)을 사용하여 관계 차수 (cardinality, 1:1, 1:M)를 표시했다. 테이블에 제공되는 각 상수는 화살표에 값을 붙여 표시하였다.

이 다이어그램이 완성되면, (슬라이드 18 과 같이) 모든 관련 인덱스, 그리고 (슬라이드 19와 같이) 데이터 양과 분산, (캐시 활용 가능성 등의) 알고 있는 수행 관련 정보들이 자세히 표시될 것이다 – 한 장에 모두 표현하기에는 너무 많다. 대부분의 경우, 특히 우리가 해당 데이터를 이미 매우 잘 알고 있는 경우라면 굳이 표시할 필요 이상으로 많은 정보일 것이다.

일단 이 모든 정보를 말끔한 그래픽으로 그려놓고 나면, 테이블을 (하나 또는 그 이상 무작위로, 원하는 만큼) 찾아내어 다음과 같은 질문을 던져볼 수 있다.
• 여기에서 시작한다면 얼마나 많은 데이터를 가져오게 될까,
• 데이터는 어떻게 수집되는가 (인덱스 또는 테이블스캔); 이 질문에 이어서 아래의 3가지 동일한 질문을 반복하게 된다:
• 다음 단계에서는 어느 테이블을 액세스 할 것인가
• 어떻게 액세스 할 것인가 (지금까지의 정보를 바탕으로)
• 결국 끝날 때의 데이터 양은 어느 정도인가

이상적이라면, 가장 싼 비용으로 가장 적은 양의 데이터를 가져오는 지점에서 시작하여, 역시 상대적으로 비용은 싸고, 가져오는 데이터를 작게 유지할 수 있는 순서로 나머지 테이블을 액세스한다. 슬라이드의 쿼리 그림을 가지고 좀더 자세히 살펴보자. (슬라이드 21에 있는 경로는 일단 무시하고) 메인 쿼리가 ***suppliers*** 테이블부터 시작한다고 해보자.

suppliers 에서 시작하는 이유는, 리즈(Leeds)지역의 공급사는 몇 개에 불과할 것이고 테이블에 적절한 인덱스가 있으므로 효율적으로 가져올 수 있기 때문이다. 분명히 ***suppliers*** 에서 이어지는 다음 단계는 공급사가 제공하는 ***products*** 이다. – 이에 맞는 인덱스가 있고, 리즈의 공급사들이 너무나 다양한 제품을 공급하지는 않기 때문에 중간 결과 집합 또한 여전히 크기가 작다.

이제 선택의 상황이 온다, ***order_lines*** 테이블로 갈 때 포린키 인덱스를 탈 수도 있고, 해쉬 조인과 테이블 스캔을 이용할 수도 있다 – 하지만 커다란 order_lines 테이블에 극도로 랜덤하게 흩어져있는 많은 로우들을 가져와야 한다. 따라서 우리는 order_lines로 가기 전에 products의 숫자를 (결국) 줄여줄 수 있는 서브쿼리로 먼저 옮겨가기로 한다. (전략 참고 – 이것을 선택한다면, 이 단계에서 작동하는 조인의 비용이 크게 비싸지 않다면, 데이터 집합의 양을 줄여줄 수 있는 조인이 양을 늘리는 조인 보다 더 좋다).

따라서 우리는 ***product_match*** 테이블로 간다 (이 결과 집합은 늘어날 수도 줄어들 수도 있다 – 대체 상품이 있는 상품이 몇 개 안될 수도 있고, 거의 모든 상품이 각각 몇 개씩의 대체 상품을 가지고 있을 수도 있다). 그리고 나서 서브 쿼리에 표시된 ***products*** 테이블로 간다. 이 중간 결과로 로우의 길이는 늘어나지만 로우의 숫자가 늘어나지는 않는다. 그리고 나서 ***suppliers*** 테이블로 간다. (아마도) 리즈 지역이 아닌 공급사의 로우 전체를 가져올 것이다 – 아마도 수집하는 모든 데이터에서 그럴 것이다.

우리가 ***order_lines*** 테이블을 액세스하려던 지점으로 다시 돌아가 보자. 물론 우리는 데이터에 대한 지식을 통해, 여기에서 리즈 지역/이외 지역을 구분할 수 있는 상품은 사실상 거의 없고, 구입하는 사람 또한 거의 없다는 것을 알고 있을 수 있다. 따라서 ***order_lines*** 테이블에 대해 랜덤 액세스를 하는 것이 충분히 쌀 수도 있다 – 하지만 일반적인 영업 시스템에서는 아마도 이런 조인을 달가워하지 않을 것이다.
지금까지 (제한적이지만) ***suppliers*** 테이블에서 시작하는 효과에 대해 검토해 보았다.

이제 슬라이드 21에 표시된 경로를 살펴보자. 우리는 지난 주 런던 지역 고객의 전체 주문 데이터를 원한다. 우리 시스템 상에서 주문 정보는 캐시에 머물러 있을 가능성이 가장 높은 데이터이고, 이미 ***orders*** 는 날짜를 인덱스로 가지고 있다. 비록 필요로 하는 것보다 훨씬 많은 데이터를 가져와야 하더라도, ***orders*** 에서 시작하는 것은 아마 꽤 효율적일 것이다. 가져오자 마자 불필요한 데이터를 가능한 빨리 버려버리기 위해 ***customers*** 와 조인하여 런던 이외 지역의 고객을 모두 제거한다. 그리고 나서 이제 ***order_lines*** 로 간다. 하지만 우리는 "최근 주문"을 가지고 왔기 때문에 원하는 order lines 또한 캐시가 잘 되어 있고, 클러스터도 잘 되어 있을 가능성이 높다. 비록 꽤 많은 데이터일 수 있지만, 여전히 효율적으로 획득할 수 있는 것이다. 이어지는 나머지 액세스 순서에 대한 논쟁은 독자가 실습해 볼 수 있도록 남겨두겠다.

여기에서는 두가지 변주를 주목할 필요가 있다. 먼저 ***orders*** 와 ***order_lines*** 테이블의 캐싱과 클러스터링에 대해서는 우리의 지식이 Oracle 보다 더 좋다는 점이다. Oracle은 이 차이로 인해 우리가 식별해낸 정교한 경로가 아닌 억지 경로를 선택하게 된다; 따라서 ***orders*** 와 ***order_lines*** 인덱스의 ***clustering_factor*** 를 정정하는 것으로 충분할 (일반적으로 더 일리가 있을) 수 있겠지만, 우리는 아마 코드에 힌트(hint)를 넣을 수 있을 것이다. 두번째로, ***orders*** 테이블에 (***id_customer, date_ordered***)인덱스가 있다면, 이 인덱스의 정밀성을 활용할 수 있으므로, ***customers*** 테이블에서 시작하여 ***orders*** 테이블로 갈 수도 있다; 이것은 인덱스 전략 검토 시 고려 항목으로 기록해 두어야 한다.

사례 연구 (CASE STUDY)

슬라이드 22 의 쿼리(위장이 포함됨)는 OLTP시스템 상에서 구조적 또는 코드 상의 변경을 최소한으로 하여 성능을 향상해야 하는 경우였다. 이 쿼리가 이상하게 보일 텐데, 알고 보면 사용자가 선택할 수 있는 계약을 드롭-다운 메뉴로 제공하기 위한 쿼리이다.

이 시스템의 AWR 리포트를 체크한 결과, 나는 옵티마이저가 지난 7일간 이 쿼리에 대해서 서로 다른 11개의 실행계획을 생성한 것을 발견했다 – 이것 중 어느 것도 특별히 효율적이지 않았다. (거의 모든 경우에 있어서 주요 문제는 ***transactions*** 테이블에 대한 테이블 스캔이었다)

쿼리에 대한 스케치를 그린 후에, 몇가지 크리티컬한 정보를 채워 넣었다. 우선 240개 사무소가 다양한 사이즈를 가지고 있었다 – 이것은 사무소 당 계약 개수의 차이를 통해 반영된다 – 따라서 Oracle의 자동 통계 수집 작업은 ***contracts*** 테이블의 ***id_office*** 컬럼에 대해 빈도 히스토그램 (frequency histogram)을 생성하였다; 이것은 바인딩 변수 보기 (bind variable peeking) 덕분이다. 즉 실행 계획은 매일 아침마다 쿼리가 누구를 대상으로 가장 먼저 수행되는가에 따라 급격하게 달라진다.

두번째로 크리티컬한 정보는 (물론, 옵티마이저에서는 사용될 수 없었던 정보) 이 표준 쿼리는 "어제 아침 이후에 일어난 것"에 대한 쿼리라는 점이었다 – 이것은 우리가 ***transactions*** 테이블에서 원하는 데이터는 오직 가장 최근의 내용에 대한 것뿐이라는 것을 의미한다. 이런 가장 최근의 정보는

캐쉬되어 사용될 확률이 매우 높은 데이터이다. 즉 비록 불필요한 많은 거래 데이터를 액세스하는 것부터 시작하더라고, 원하는 데이터를 찾아내기까지 I/O가 그리 많이 않을 것이라는 의미이다.

또 하나의 포인트는 (역시 마찬가지로 옵티마이저가 사용할 수 없는 정보) 사무소가 취급하는 계약의 개수는 매번 항상 늘어난다는 점이다 – 하지만 일자별 거래의 개수는 (사실상) 변화가 없다. 이는 과거의 계약들이 더 이상 거래를 만들어 내지 않기 때문이다.

따라서 첫번째 솔루션은: 시스템을 불안정하게 만드는 히스토그램을 제거하는 것이다. 그리고 나서 경로를 ***transactions -> contracts -> transaction_types***순서로 잡는 것이다. 이것은 대부분의 사용자가 필요로 하는 데이터보다 훨씬 많은 데이터를 가지고 시작할 것이고 터무니없이 많은 논리(logical) I/O를 일으킬 것이다. 하지만, 이것은 충분히 빠르고, 모든 사람이 행복하게 지속시킬 수 있을 정도로 일정할 것이다.

인덱스 변경에 대해서는 많은 반대가 있을 수 있지만, 인덱스에 대한 다양한 변경을 통해 경로를 보다 효율적으로 만들 수 있고, 결국 보다 나은 경로를 가질 수 있게 된다.

옵션 1은 ***office_id*** 컬럼을 *contracts* 테이블의 (***id***)컬럼에 있는 (프라이머리 키) 인덱스에 추가하는 것이다. 이를 통해 ***contracts*** 정보를 가져올 때 인덱스 만으로 충분하게 된다. 이런 변경은 위험성이 상대적으로 낮다 (대부분의 인덱스 변경에 비해). 왜냐하면 (***id***)인덱스는 이미 유니크하고, 단일 컬럼이기 때문이다. – 하지만 여전히 다른 쿼리에 대해 예상치 못한 부작용이 없는 지를 조심스럽게 테스트할 필요가 있다.

옵션 2는 ***transactions*** 테이블의 (***created***) 인덱스를 확장하여 필요한 컬럼을 모두 추가하여 테이블을 읽을 필요를 없애는 것이다 – 그결과 버퍼(buffer) 액세스에 대해 현격한 차이를 만들 것이다 – 하지만, 다른 쿼리에 끼치는 영향은 더욱 클 수도 있다.

이 옵션들을 고려하면서, 우리는 contracts 테이블에서 시작하여 transactions 테이블을 액세스하고, 인덱스를 강화하여 가능한 효율적으로 사용될 수 있도록 하는 또 다른 경로를 고려했지만 즉시 거부했다. 왜냐하면 이 솔루션은 확장성이 결여되었기 때문이다. 시간이 지날 수록 사무소는 새 계약을 만들 것이고, 절대로 제거하지는 않는다. 따라서 계약을 먼저 식별하고 나서 해당 계약에 의해 발생한 최근 24시간 내의 거래 (대개에 경우 존재하지 않는) 를 찾아가는 솔루션은 시간이 갈수록 더 많은 수행과 시간을 필요로 하게 된다. 이 솔루션을 확장성 있도록 바꾸려면, ***contracts*** 테이블에 "*contract terminated*" 나 "*last transaction date*" 컬럼을 추가한 후 이것을 쿼리의 드라이빙 조건이 되는 인덱스에 포함시켜야 한다.

그날, 고객은 안전한 코스를 선택했다 – 인덱스 변경은 하지 않고 코드에 힌트를 주어서 히스토그램을 제거하기로 한 것이다. 가장 빠른 옵션은 아니었지만, 충분히 빠르고 안정적이었다. 왜냐하면, 이 쿼리가 종속되는 것은 (아주 일정한) 일자별 거래 개수이기 때문이다. 가변적이고 증가하는 수량 즉 사무소 별 거래에 더 이상 기대고 있지 않다.

결론

옵티마이저가 쿼리에 대해 좋은 실행 계획을 만드는 데 실패했다면, 그 이유는 주로 데이터에 대한 모델이 실제와 일치하지 못하기 때문이다. 이런 문제를 해결하려면 옵티마이저가 하려고 하는 작업을 우리가 직접 해볼 필요가 있고, 이런 작업을 쉽게 할 수 있는 장치 또한 필요하다. 그림(그래픽)을 활용하는 접근법을 잘 활용하면 쿼리의 "형태" 파악과 해당 쿼리에 대한 보다 많은 정보를 이해하기 쉽게 표현할 수 있다.

만약 쿼리, 통계, 인덱스를 이렇게 이해하기 쉽게 표현할 수 있다면, 최적의 실행 경로를 파악하거나, 구조 변경(인덱스 생성이나 변경)을 통해 더 나은 경로를 만드는 것을 결정하는 것이 쉬워질 것이다.

저자 소개

조나단 루이스 (Jonathan Lewis) 는 IT 분야에서 25년 간 종사하면서 Oracle을 20년 이상 사용해 오고 있다. 최근 16년 간은 프리랜서 컨설턴트로서 다양한 고객의 심각한 성능 문제를 주로 1~2일에 해결하고 있다. 조나단의 교육과 세미나는 세계적으로 널리 알려져 있기도 하다 (전세계 42개국에서 방문). Oak Table 네트워크의 창립멤버이며, Oracle의 "Celebrity Seminar"이벤트를 위해 Oracle University와 최초로 개인 자격으로 계약하기도 하였다. 조나단은 UKOUG 메거진에 정기적으로 기고를 하고 있으며 전세계의 기타 출판 매체에도 기고하기도 한다. 여유가 생길 때면 Oracle 관련 고급 아티클을 jonathanlewis.wordpress.com 에 올리고 있다

How to ...
.. design efficient SQL

Jonathan Lewis
www.jlcomp.demon.co.uk
Jonathanlewis.wordpress.com

Who am I ?

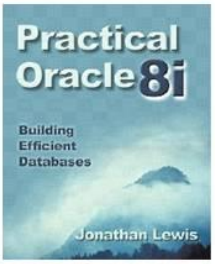
Independent Consultant

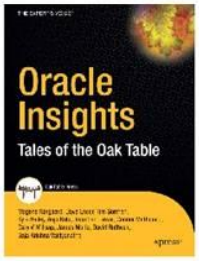
25+ years in IT
22+ using Oracle

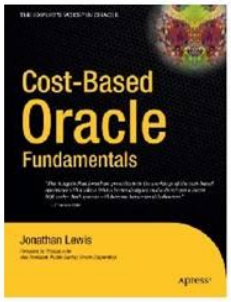
Strategy, Design, Review
Briefings, Seminars
Trouble-shooting

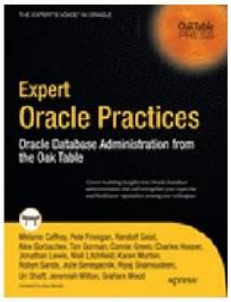
Jonathanlewis.wordpress.com
www.jlcomp.demon.co.uk

Member of the Oak Table Network.
Oracle author of the year 2006
***Select* Editor's choice 2007**
Oracle *ACE Director*
O-1 visa for USA









Highlights

Know the data

Does a good execution path exist?

Can the optimizer find that path?

Jonathan Lewis
© 2006 - 2010

Efficient SQL
3 / 28

Knowing the data

How much data?

Where is it?

Jonathan Lewis
© 2006 - 2010

Efficient SQL
4 / 28

Knowing the data - conflict

Your knowledge of the data

The optimizer's model of the data

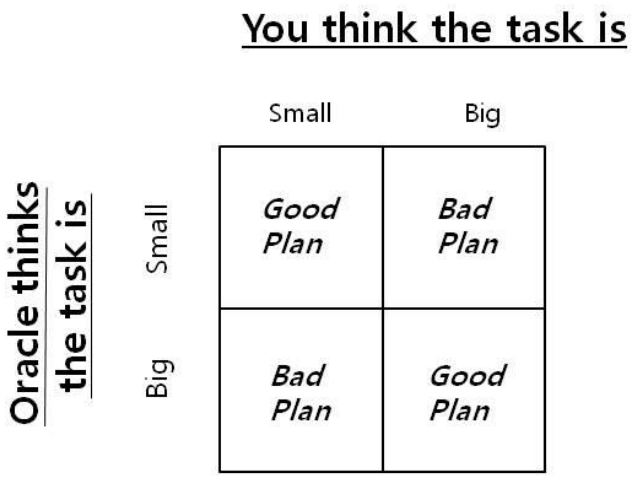
Strategy Choice

Lots of little jobs

How many
How little (how precise)

One big job

Common Outcomes



Jonathan Lewis
© 2006 - 2010

Efficient SQL
7 / 28

Optimizer problems

Correlated columns

Uneven data distribution

Aggregate subqueries

Non-equality joins

Bind variables

Jonathan Lewis
© 2006 - 2010

Efficient SQL
8 / 28

Know the metadata

```
select
        table_owner,
        table_name,
        index_name,
        column_name
from
        dba_ind_columns

order by
        table_owner,
        table_name,
        index_name,
        column_position
;
```

~~AP_GRP_FK_I~~	~~GRP_ID~~	
AP_GRP_ROLE_I	GRP_ID ROLE_ID	(compress)
AP_ORG_AP_I	ORG_ID AP_ID	(compress 1)
~~AP_ORG_FK_I~~	~~ORG_ID~~	
AP_PER_AP_I	PER_ID AP_ID	(compress 1)
~~AP_PER_FK_I~~	~~PER_ID~~	
AP_PK	AP_ID	
AP_ROLE_FK_I	ROLE_ID	(compress)
AP_UD_I (UPD_DATE)	TRUNC	(drop, compress, coalesce)

Jonathan Lewis
® 2006 - 2010

Efficient SQL
9 / 28

Know the data (a)

```
select
        colX,
        count(*) ct
from    t1
group   by  colX
order   by  colX
```

COLX	CT
1	9
2	12
3	12
4	8
5	7
6	9
…	
…	
9997	1
9998	1
9999	1
10000	1

```
sample (5)
sample block (5)
sample block (5, 2)             -- 9i
sample block (5, 2) seed (N)    -- 10g
```

Jonathan Lewis
® 2006 - 2010

Efficient SQL
10 / 28

Know the data (b)

```
select
        ct, count(*)
from    (
        select
             colX,
             count(*)  ct
        from  t1
        group by  colX
        )
group by ct
order by ct
;
```

CT	Count(*)
1	9001
…	
6	37
7	78
8	94
9	117
10	112
11	126
12	99
13	97
14	86
15	49
16	32
…	
22	1

There are 99 values that appear 12 times

Jonathan Lewis
© 2006 - 2010

Efficient SQL
11 / 28

Know the data (c)

```
select
        blocks, count(*)
from    (
        select
             /*+ index(t1  t1_pk)  */
             colX,
             count (
                 distinct  substr(rowid,1,15)
             )      blocks
        from  t1
        group by  colX
   )
group by blocks
order by blocks
;
```

BLOCKS	Count(*)
1	9001
…	
6	43
7	83
8	107
9	126
10	120
11	125
12	119
13	90
14	69
15	42
16	28
…	
19	2

There are 90 values that are scattered across 13 blocks

Jonathan Lewis
© 2006 - 2010

Efficient SQL
12 / 28

Know the data (d)

```
select   /*+ index(t, "T1_T1") */
        count(*)                                                    nrw,
        count(distinct sys_op_lbid(49721,'L',t.rowid))              nlb,
        count(distinct hextoraw(
              sys_op_descent("DATE_ORD")||
              sys_op_descend("SEQ_ORD")
        ))                                                          ndk,
        sys_op_countchg(substrb(t.rowid,1,15),1)                    clf
from
        "TEST_USER"."T1"  t
where
        "DATE_ORD" is not null
    or "SEQ_ORD" is not null
;
```

Jonathan Lewis
® 2006 - 2010

Efficient SQL
13 / 28

Know the data (e)

```
select
        keys_per_leaf,  count(*)  blocks
from    (
        select
               sys_op_lbid(49721,'l',t.rowid)       block_id,
                 count(*)                           keys_per_leaf
        from
                 t1  t
        where
                 {index_columns are not all null}
        group by
                 sys_op_lbid(49721,'l',t.rowid)
        )
group by         keys_per_leaf
order by         keys_per_leaf
;
```

Jonathan Lewis
® 2006 - 2010

Efficient SQL
14 / 28

Know the data (f)

Smashed Index

KEYS PER LEAF	BLOCKS
3	114
4	39
6	38
7	39
13	37
14	1
21	1
27	15
28	3
39	1
54	6
55	3
244	1
281	1
326	8

Normal Index

KEYS PER LEAF	BLOCKS
17	206
18	373
19	516
20	678
21	830
22	979
23	1,094
24	1,178
25	1,201
26	1,274
27	1,252
28	1,120
29	1,077
30	980
31	934
32	893
33	809
34	751
35	640
36	738
37	625
38	570
39	539
40	489

Jonathan Lewis
® 2006 - 2010

Efficient SQL
15 / 28

Draw the query - requirement

지난 주 의 주문 중
고객이 **런던**(***London***)*에 거주하고*
공급사가 **리즈**(***Leeds***)에 있고
다른 지역에도 공급사가 있는 경우

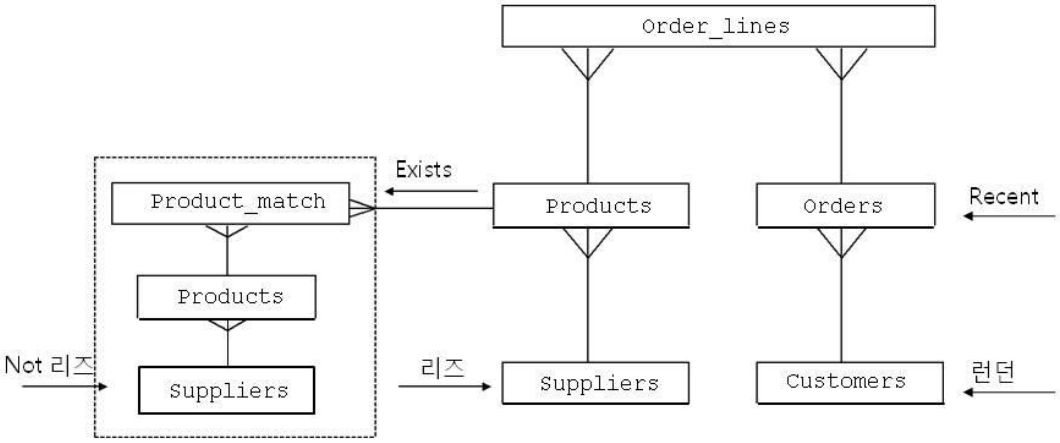
```
select   {columns}
From     customers          cus,
         orders             ord
         order_lines        orl
         products           prd1
         suppliers                  sup1
where    cus.location       = 'LONDON'
and      ord.id_customer    = cus.id
and      ord.date_placed    between sysdate - 7 and sysdate
and      orl.id_order       = ord.id
and      prd1.id            = orl.id_product
and      sup1.id            = prd1.id_supplier
and      sup1.location      = 'LEEDS'
and      exists  (
                 select   null
                 from     product_match     mch,
                          products          prd2,
                          suppliers                 sup2
                 where    mch.id_product    = prd1.id
                 and      prd2.id           = mch.id_product_sub
                 and      sup2.id           = prd2.id_supplier
                 and      sup2.location     != 'LEEDS'
                )
```

Jonathan Lewis
® 2006 - 2010

Efficient SQL
16 / 28

Draw the query - outline

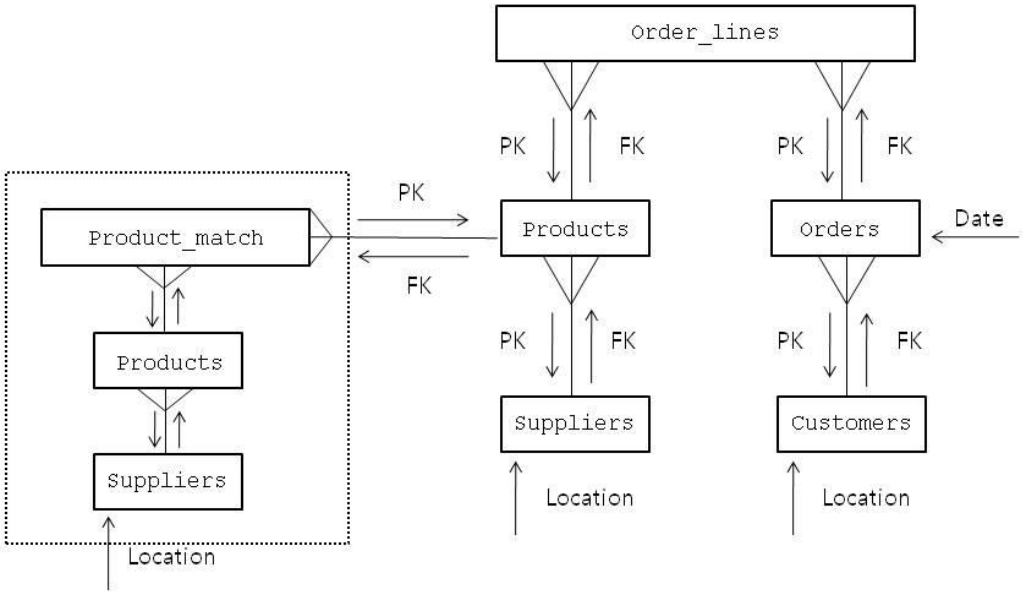
지난 주 의 주문 중
고객이 **런던**(***London***)*에 거주하고*
공급사가 **리즈**(***Leeds***)에 있고
다른 지역에도 공급사가 있는 경우



Jonathan Lewis
® 2006 - 2010

Efficient SQL
17 / 28

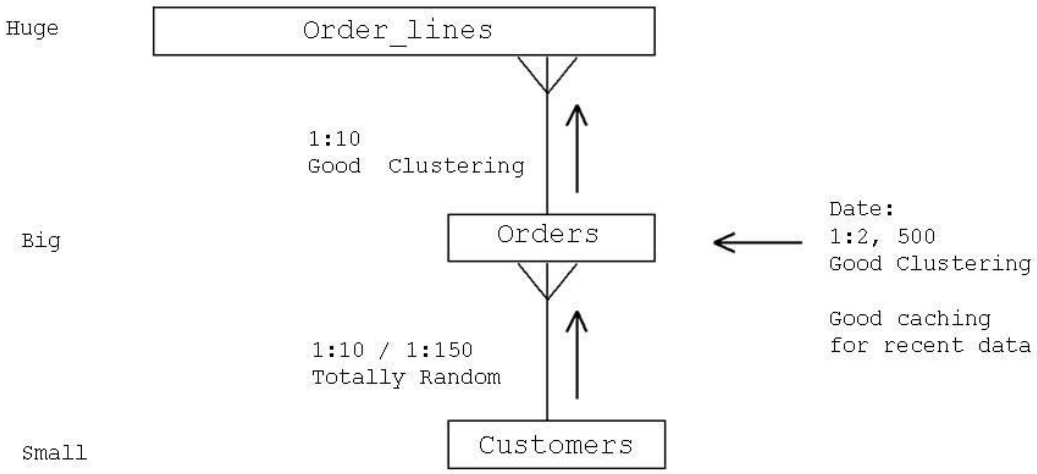
Draw the query - indexes



Jonathan Lewis
® 2006 - 2010

Efficient SQL
18 / 28

Draw the query - statistics



Jonathan Lewis
© 2006 - 2010

Efficient SQL
19 / 28

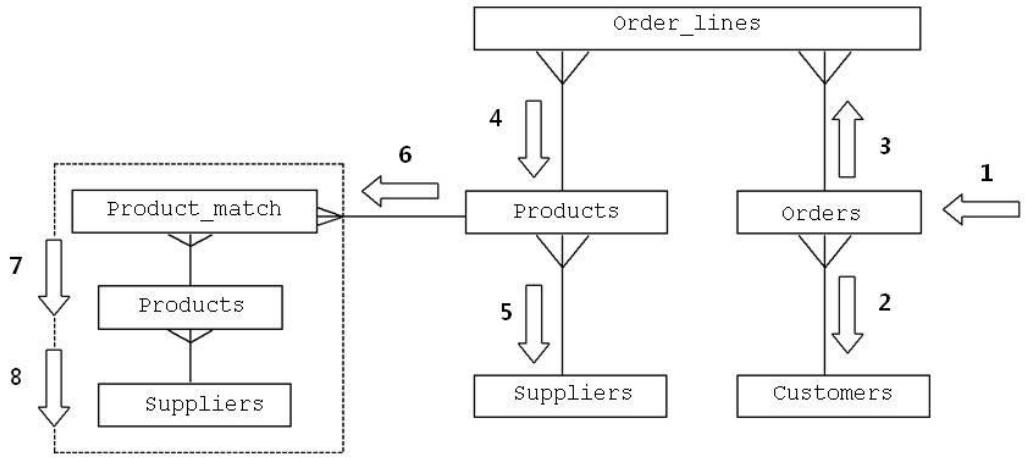
Sketch the paths - strategy

- **Pick a starting point**
  - How many rows will I start with
  - How efficiently can I get them
  - the first step may be inefficient (it only happens once)
- **How do I get to next table**
  - How many times do I make the step
  - How precise is the access path
  - How much data do I now have

Jonathan Lewis
© 2006 - 2010

Efficient SQL
20 / 28

Sketch in paths - analysis



Jonathan Lewis
© 2006 - 2010

Efficient SQL
21 / 28

Case Study (a)

```
select
        distinct  trx.id_contract

from
        transactions        trx,
        contracts           con,
        transaction_types   tty

where
        trx.id_ttype        =  tty.id
and     trx.id_contract     =  con.id
and     con.id_office       =  :b1
and     tty.qxt             <>  'NONE'
and     trx.created         between  :b2  and  :b3
and     trx.error           =  0
;
```

Jonathan Lewis
© 2006 - 2010

Efficient SQL
22 / 28

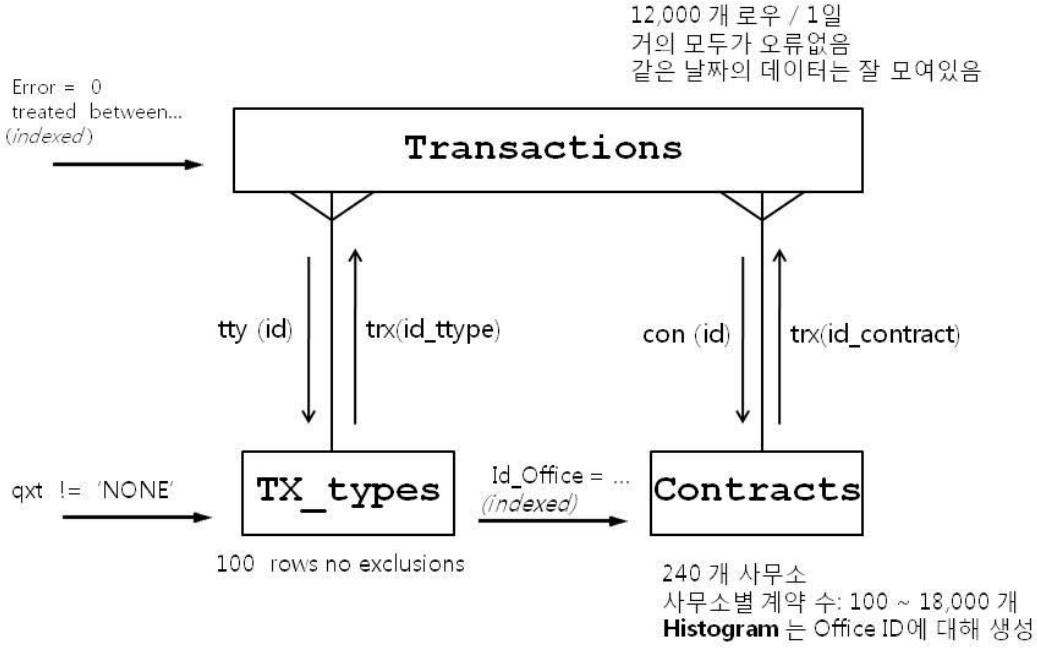
Case Study (b)

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT				14976
1	SORT UNIQUE		7791	304K	14976
2	FILTER				
3	HASH JOIN		7798	304K	14974
4	VIEW		9819	98190	1599
5	HASH JOIN				
6	INDEX RANGE SCAN	CON_OFF_FK	9819	98190	35
7	INDEX FAST FULL SCAN	CON_PK	9819	98190	1558
8	HASH JOIN		7798	228K	13374
9	TABLE ACCESS FULL	TRANS_TYPES	105	945	3
10	**TABLE ACCESS FULL**	**TRANSACTIONS**	**7856**	**161K**	**13370**

Jonathan Lewis
® 2006 - 2010

Efficient SQL
23 / 28

Case Study (c)



Jonathan Lewis
® 2006 - 2010

Efficient SQL
24 / 28

Case Study (d)

Get rid of the histogram on office_id
Use hints to force the only "constant volume" plan

Modified Indexes

```
contracts(id, office_id)
```

Expected plan:

```
hash join

    table access full
    transaction_types nested loop

     table access by rowid transactions
       index range scan transactions_idx
     index range scan contracts_idx
```

Jonathan Lewis
© 2006 - 2010

Efficient SQL
25 / 28

Case Study (e)

Modified Indexes

```
transactions(created, con_id, error, id_ttype)
contracts(id, office_id)
```

Expected plan:

```
hash join
  table access full transaction_types
  nested loop
    index range scan transactions_idx
    index range scan contracts_idx
```

Jonathan Lewis
© 2006 - 2010

Efficient SQL
26 / 28

Case Study (f)

Modified Indexes

```
contracts(id_office, id)
transactions(id_contract, created)
```

Expected plan:

```
hash join
  table access full
  transaction_types nested loop
    index range scan contracts_idx
    table access by rowid transactions
      index range scan transactions_idx
```

Jonathan Lewis
© 2006 - 2010

Efficient SQL
27 / 28

Summary

Know the data

Draw the picture

Identify the problems

- Bad indexing
- Bad statistics

Optimizer deficiencies

- Structure the query with hints

Jonathan Lewis
© 2006 - 2010

Efficient SQL
28 / 28