

**SQL은 언제 SQL 텍스트가 이름
!!
SQL ID → 1:1 대응**

→ key가 SQL문 그 자체

→ Select * From customer
where id = " " + id + " "

→ 하드파싱 ↑↑ CPU 사용률 ↑↑

하지만 캐시공간 부족하면 버림
oracle은!
why? - 느려지기 때문

↓ 수정
Select from customer where id = ?

→ 라이브러리 캐시 조회하면
: Select from customer id = :1

하드파싱 1번, 공유 / 재사용

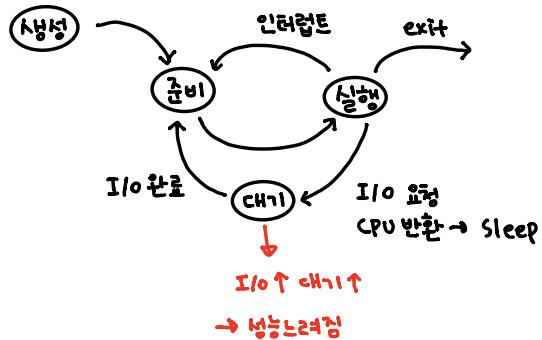
파라미터 사용

I/O 투닝 = SQL 투닝

프로세스 생명주기

: 노친이유: 디스크 I/O 때문

프로세스: 실행중인 프로그램



물리적

데이터파일

데이터파일

논리적

데이터스페이스

세그먼트 (레이블)

세그먼트 (인덱스)

액스텐트

액스텐트

액스텐트

블록

블록

블록

연속된공간

→ 세그먼트를 담는 컨테이너

오브젝트

→ 데이터를 읽고 쓰는 단위

몇번 데이터 파일의 몇번째 블록인지

나타내는 고유 주소값 가짐

⇒ DBA (데이터 블록 어드레스)

여러 데이터파일에 흩어져 저장 (디스크 경합 풀임)

⇒ 연속된공간XX

물리적 I/O ⇒ BCHR에 의해 결정 → 논리적 I/O × (100 - BCHR)

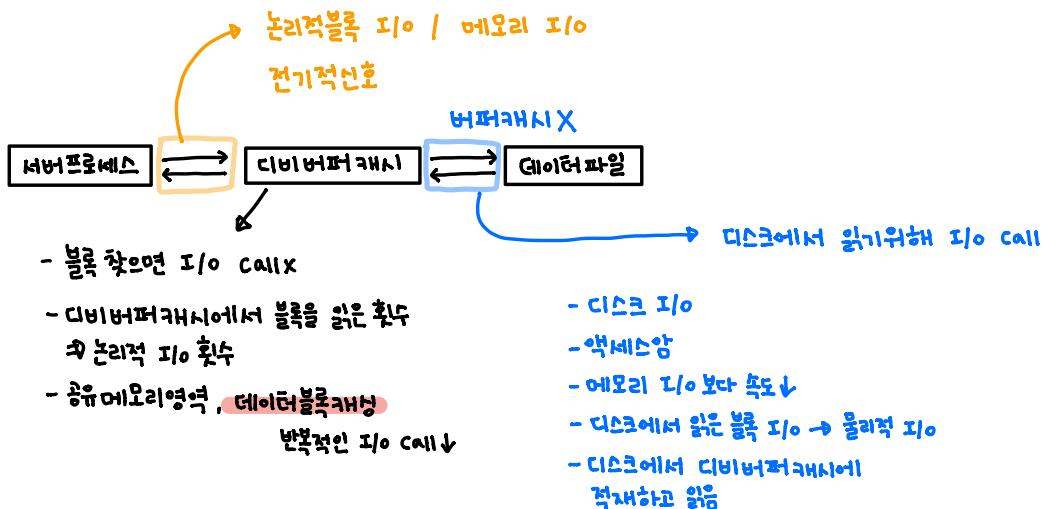
버퍼캐시 히트율

통제불가능

논리적 I/O ↓, 물리적 I/O ↓ ∵ 성능↑↑

↳ SQL을 투닝해서 읽는 총 블록개수 줄이면됨

즉 논리적 I/O ↓ 물리적 I/O ↓



싱글블록 I/O : 한번에 한 블록씩 요청

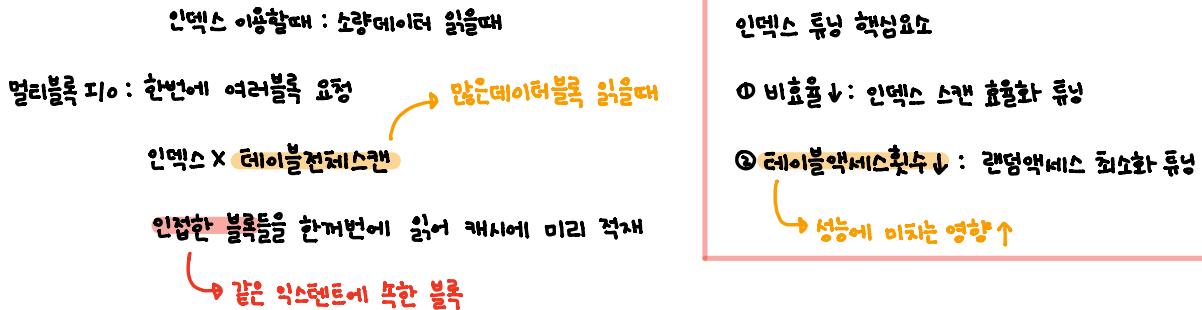


Table full Scan : 시퀀셜액세스 & 멀티블록 I/O 방식으로 디스크 블록 읽음

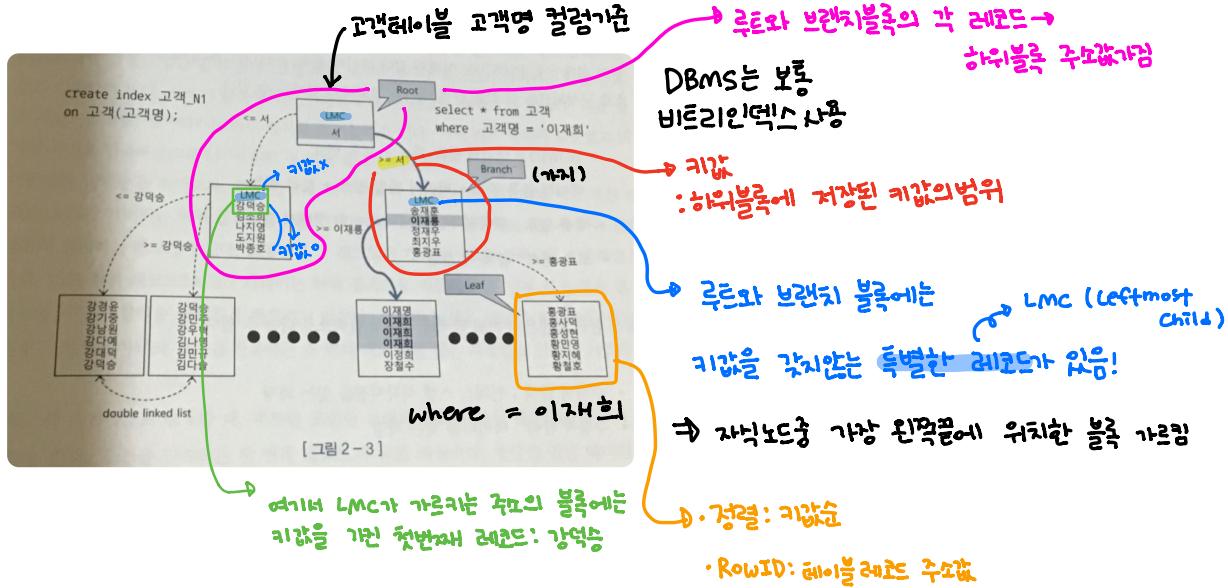
- 클레이블에서 소량데이터 검색 → 인덱스 사용 필수 / 읽을데이터가 일정량 ↑ → 레이블풀스캔이 유리

인덱스 레인지 스캔 : 랜덤액세스 & 싱글블록 I/O → 범위스캔

→ 캐시에 블록 X → I/O call → 캐시에 블록 X → I/O call

→ 읽었던 블록 반복해서 읽는 비효율

버퍼캐시 ← 캐싱된 버퍼블록 : 공유자원 대러니즘 필요
→ 버퍼 Lock
↳ 두개 이상의 프로세스가 동시에 접근할 수 있기 때문에 캐시버퍼 채인래치가 존재 → 캐시버퍼 채인래치
줄세우기



수직적 탐색: 인덱스 스캔 시작지점 찾는 과정 ex) 등산포탈

루트 ----- 루트부터 시작
↓
→ 표현을 만들하는 첫번째 레코드 찾는 과정

브랜치
↓
리프
→ 가능한 이유: 루트, 브랜치 각 인덱스
레코드는 하위 블록 주소값
갖고있기 때문

?) 브랜치블록 레코드는 랜덤액세스한 것?

수평적 탐색: 수직적 탐색으로 스캔시작점 찾은 후 찾고자 하는 레이터가 더 만나는지까지

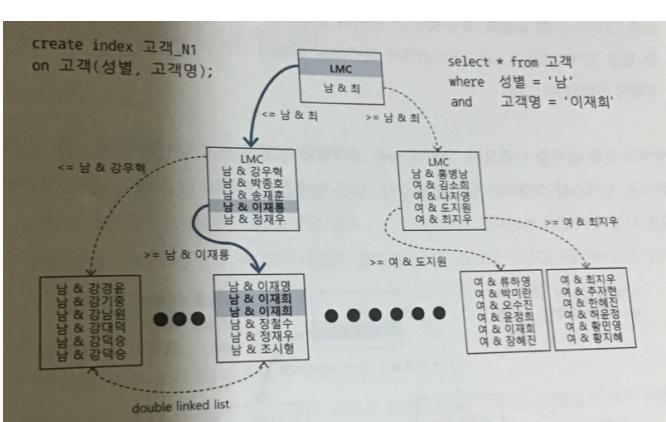
인덱스 리프블록 수평적 스캔
→ 데이터 찾는 과정

A ←→ B : 양방향 연결리스트

• 앞뒤블록 주도값가짐

1. 조건절 맞는 데이터 모두 찾기위해

2. RowID 얻기 위해



결합인덱스 구조

- 두 개 이상의 철검 결합 → 인덱스 생성가능

인덱스 스캔시작점: 성별=남이면서 고객명='이재희'인 레코드

- 인덱스컬럼을 가공하지 않아야 인덱스를 정상적으로 사용가능

인덱스가공하면 스캔시작점 못찾음, 멈출 수 없음 → 리프블록 전체스캔
 ↓
 가공하지 않은 인덱스인데 가공한값을 기준으로 검색하면
 스캔시작점 못찾음

정렬되어 있음

- 인덱스 Range Scan 첫번째조건 : 인덱스 선두 컬럼이 조건절에 있어야 가능
 - ④ 인덱스 선두컬럼이 가공되지 않은 상태로 조건절에 있으면 가능

ex) where 기준연도 = :기준연도 index : 기준연도 + 코드
 and substr(코드, 1, 4)
 ↪ 가공안했기 때문에 가능

인덱스선두컬럼 → 기본적으로 레이블 풀스캔 선택

스캔시작점 못찾음

where nvl(주문수량, 0) < 100

OR, Like, In

업데이트가
 IN-List Iterator 사용
 ⇒ Union All 효과

인덱스를 이용한 소트연산 생각

PK인덱스 : 장비번호 + 변경일자 + 변경순번

만약 인덱스가 장비번호 + 변경일자라면

Select 변경이력 where 장비번호 = 'C' and 변경일자 = '20190301' Sort (order by) 연산단계 추가

→ PK인덱스 스캔 출력 결과집합은 변경순번 순으로 정렬

order by, Select-list에서 커럼 가공해서 인덱스 사용할 수 없는 경우도 있음

ex) select ... order by 변경일자 || 변경순번

→ 가공한값 기준으로 정렬, 정렬연산 생각불가

```
select TO-CHAR(A.주문번호, FM0000) 주문번호, A.업체번호
from 주문 A
where A.주문일자 = '20190301'
and A.주문번호 > nvl(3, 0)
order by 주문번호
```

가공한 주문번호 ⇒ 변경전: 주문번호 → 변경후 A.주문번호

자동형변환

```
select * from 고액 where 생년월일 = 1994120
→ 업데이트가
  ↪ 주문번호 / 주문일자 / 거래일자 - 이전주제내적 표기
```

비교값: 날짜형

→ LIKE일때는 문자형 win!

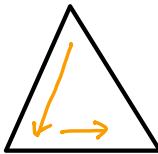
날짜형 > 문자형

날짜형 > 문자형

Index Range Scan

- B⁺Tree 인덱스의 가장 일반적이고 정상적 형태의 액세스방식

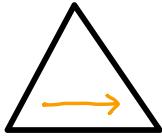
인덱스 루트
↓
리프블록



인덱스스캔 범위, 레이블 액세스 횟수 ↓↓ 성능 ↑

Index full Scan

- 수직적 탐색 X 인덱스 리프블록 수평탐색

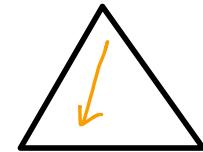


DBA (=데이터파일번호 + 블록번호)

디스크상에서 블록을 찾기 위한 주소정보

Index Unique Scan

- 수직적 탐색만함
- Unique 인덱스를 = 조건으로 탐색하는 경우



Index SKIP Scan

- 오라클 → 인덱스 선두질점이 조건절에 없어도 인덱스를 활용하는 새로운 스캔방식
- 인덱스 선두질점의 Distinct Value 개수가 적고 후행질점 Distinct Value 개수가 많을 때 유리



Ex) index : 업종유형코드 + 업종코드 + 기준일자

Select ... From 일별업종별거래 Where 업종유형코드 = '01' and 기준일자 between '2008051' and '20080531'

index range Scan 이면 업종유형코드='01'인 인덱스구간 모두 스캔

index skip scan → 업종유형코드='01' 구간에서 기준일자가 '2008051'보다 크거나 같고 '20080531'보다 작거나 같은
리코드를 포함할 가능성이 있는 리프블록만 플레이시 액세스 가능

Index Fast Full Scan

- 인덱스 세그먼트 전체를 블럭별로 I/O 방식으로 스캔

- 물리적으로 디스크에 저장된 순서대로 인덱스 리프 블록들을 읽어들임
- 루트와 브랜치블록 → 필요없는 블록이라 버림
- 인덱스가 순서대로 정렬X
- 쿼리에 사용한 절점이 모두 인덱스에 포함돼 있을 때만 사용가능

Select # from emp

↳ 인덱스 모두 포함X : 인덱스 스캔 후 반드시 테이블 액세스

↳ Table access by index rowid

· index rowid : 논리적주소

디스크상 → 테이블레코드

논리적인주소
(rowid)

↳ rowid가 가르키는 테이블 블록을 버퍼캐시에서 찾음
→ 버퍼캐시 적재 → 못찾으면 디스크에서 블록 읽음

rowid : 유전주소, 메인메모리 DB가 사용하는 포인터 : 전화번호

↳ 7번 데이터파일 123번 블록에 저장된 10번째 레코드

인덱스 클러스터링 팩터 (CF) : 균일성 계수

· 특정 절점 기준으로 같은 값을 갖는 데이터가 서로 모여있는 정도를 의미

CF 좋은 절점에 생성하는 인덱스 효율↑↑

Index range scan에 의한 테이블 액세스가 Table full Scan보다 느려지는 지점 : 인덱스 손익분기점

↳ 일반적으로 5~20% ←

비하인드 프로그램 튜닝

- Full Scan과 해시조인이 유리

```
insert into 고객_임시
select 고객번호, 고객명, 전화번호, 주소, 상태코드, 변경일시
from (select /*+ full(c) full(h) leading(c) use_hash(h) */
       c.고객번호, c.고객명, h.전화번호, h.주소, h.상태코드, h.변경일시
       , rank() over (partition by h.고객번호 order by h.변경일시 desc) no
  from 고객 c, 고객변경이력 h
 where c.고객구분코드 = 'A001'
 and h.변경일시 >= trunc(add_months(sysdate, -12), 'mm')
 and h.변경일시 < trunc(sysdate, 'mm')
 and h.고객번호 = c.고객번호)
where no = 1
```

→ 파티션 활용 전략이 매우 중요한 튜닝요소 + 병렬처리도 있으면 좋음

인덱스 철렁 추가

Call	Count	CPU Time	Elapsed Time	Disk	Query	Current	Rows
Parse	1	0.010	0.012	0	0	0	0
Execute	1	0.000	0.000	0	0	0	0
Fetch	78	10.150	49.199	27830	266968	0	1909
Total	80	10.160	49.211	27830	266968	0	1909

Rows	Row Source Operation
1909	TABLE ACCESS BY INDEX ROWID 로밍렌탈 (ct=266968 pr=27830 pw=0 time= ...)
266476	INDEX RANGE SCAN 로밍렌탈_N2 (ct=1011 pr=900 pw=0 time=1893462 us)

→ 인덱스를 스캔하고 있는 건수
 → 테이블 266476 봤음
 근데 최종결과합은 1909건
 ⇒ 사용여부 = 'Y' 조건
 체크과정에서 대부분 걸려짐

이만큼 테이블 랜덤액세스
 이단계에서
 $266968 - 1011 = 265,957$
 블록을 읽음

→ 테이블 액세스단계 필터 조건에 의해

버려지는 레코드가 많을때 인덱스에 철렁추가

인덱스 구조 헤이블

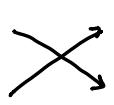
랜덤액세스가 아예 발생하지 않도록 헤이블을 인덱스 구조로 생성 가능

→ 오라클: IOT, MS-SQL: 클러스터형

- 인덱스 리프블록 = 데이터블록

↑ 일반테이블

total rowid
 자체리스트 rowid



자체리란데 가격 6500
 자체 가격 5000

↑ 인덱스 구조

아마 가격 5000
 자체리란데 가격 6500

- 일반레이블 = 쿠거즈 레이블, 순번없이 데이터 입력
- IOT = 인덱스구조, 정렬상태 유지하여 데이터 입력
 ↳ 시퀀셜 방식으로 데이터 액세스 / Between, 부등호 조건 넓은 범위 일떈 유리

클러스터 테이블

- 인덱스 클러스터 테이블

CLUSTER HEADER		TABLE COLUMNS				
DEPTNO	EMPNO	ENAME	JOB	MGR	SAL	
10	7839	KING	PRESIDENT		5000	
	7782	CLARK	MANAGER	7839	2450	
	7934	MILLER	CLERK	7782	1300	
CLUSTER						
20	7788	SCOTT	ANALYST	7566	3000	
	7902	FORD	ANALYST	7566	3000	
	7566	JONES	MANAGER	7839	2975	
	7876	ADAMS	CLERK	7788	1100	
	7869	SMITH	CLERK	7902	800	
CLUSTER						
30	7698	BLAKE	MANAGER	7839	2850	
	7499	ALLEN	SALESMAN	7698	1600	
	7844	TURNER	SALESMAN	7698	1500	
	7521	WARD	SALESMAN	7698	1250	
	7654	MARTIN	SALESMAN	7698	1250	
	7900	JAMES	CLERK	7698	950	
	CLUSTER					

· 같은 레코드를 한 블록에 모아서 저장

→ 클러스터 체인으로 연결



일반레이블 : 하나의 데이터블록
여러레이블을 공유 X

· 정렬은 안함

→ 클러스터에 도달해서는 시퀀셜방식으로 접근

해시클러스터 테이블

- 해시 알고리즘 사용해 클러스터 찾기

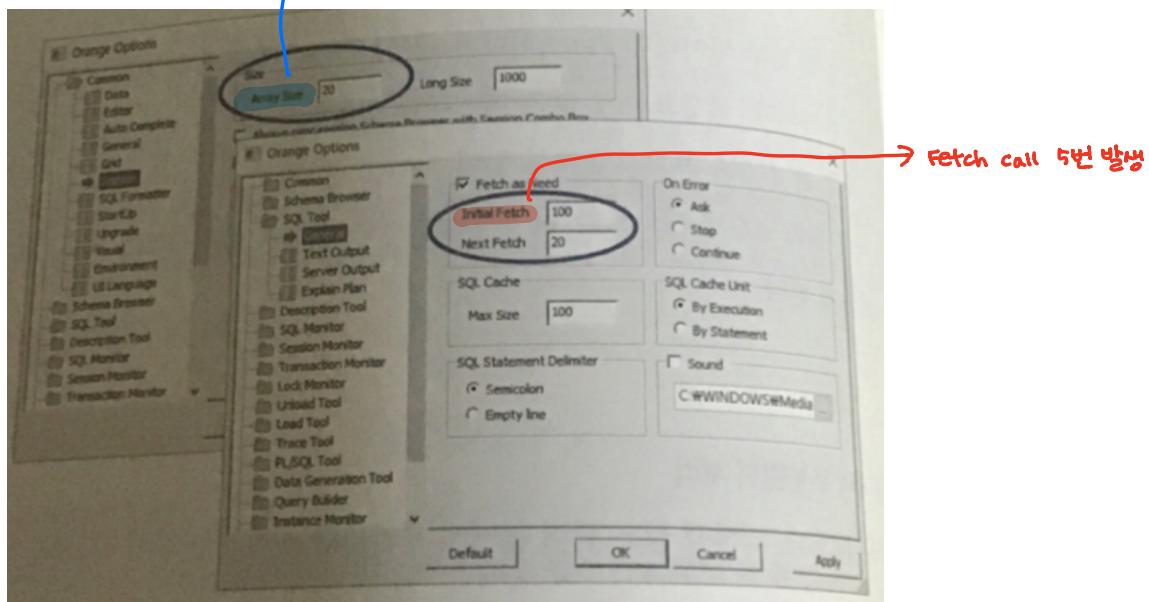
복분법위 처리

- 일정량을 전송하고 멈춤

데이터전송 → 서버프로세스는 CPU OS에 반환 → 대기큐에서 슬립 → Fetch call → 대기큐에서 나와 그다음 데이터

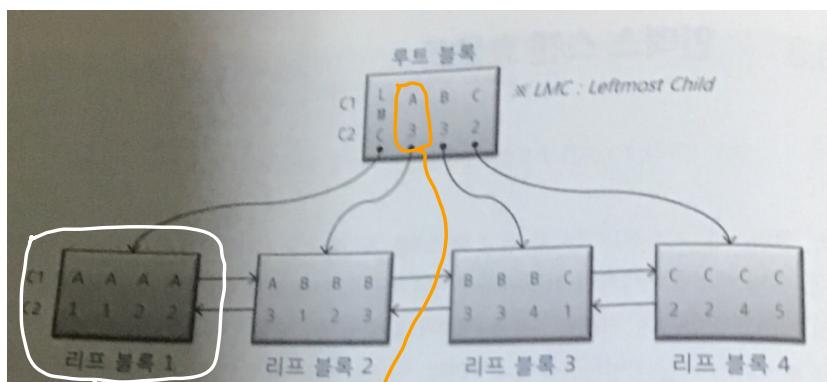
일정량 읽고 전송 → 잠

→ 오류가
Fetch call할 때 데이터 20개씩 요청하도록 설정



- 부분범위 처리학습: 앞쪽 일부만 출력하고 멈출 수 있는가

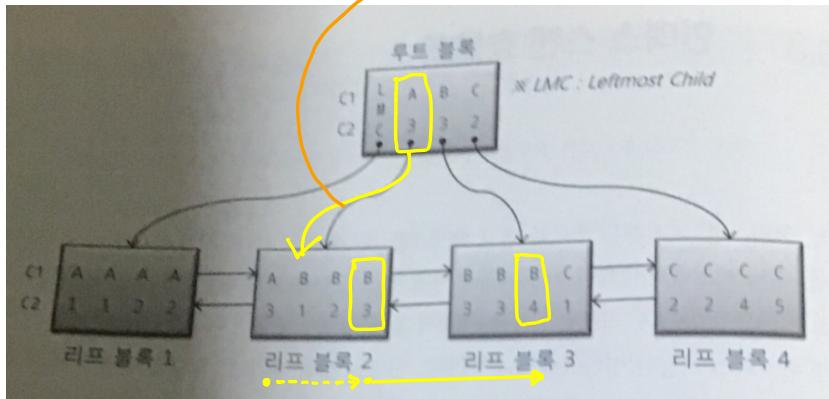
인덱스탐색



→ 여기에서는 C1=A AND C2=3인 레코드보다 작거나 같은 값 가지는 레코드가 저장되어 있음

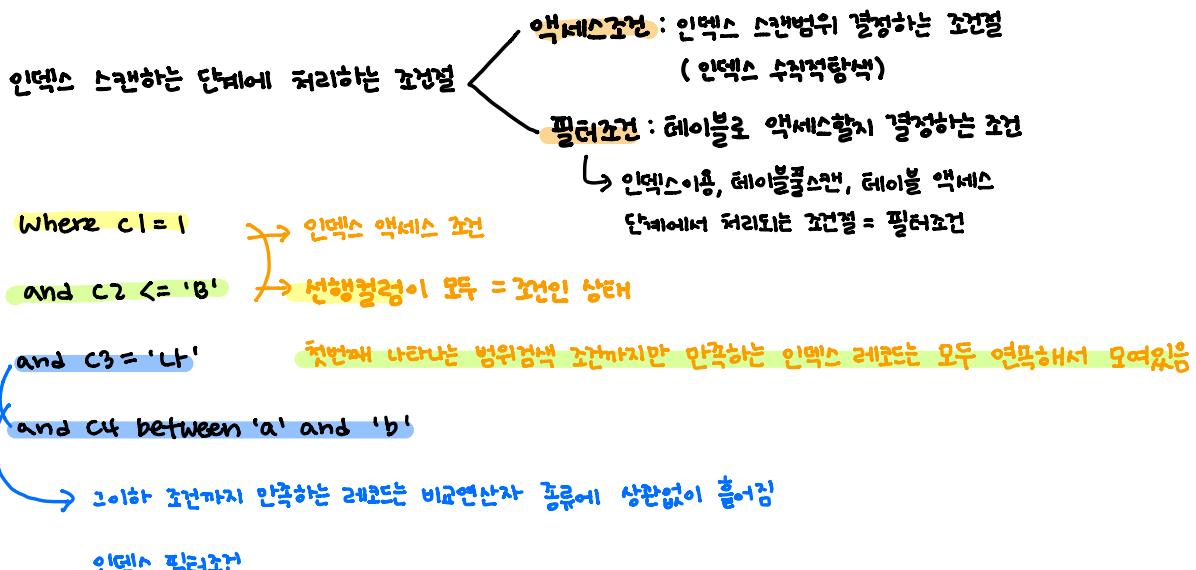
where $C1 = B$ AND $C2 = 3$

수직적 탐색: 스캔 시작점 찾는 과정



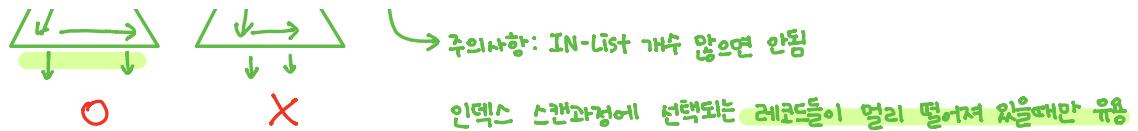
• 선두절점: 맨 앞쪽에 있는 절점 지정할 때

• 번행절점: 상대적으로 앞쪽에 놓인 절점



• 인덱스 스캔 효율성: 인덱스절점을 조건절에 모두 등록 (=) 조건으로 사용할 때 가장 좋음





between → IN-list

index Skip Scan도 가능 /*+ INDEX_SS (+ 월별고객판매집계_IDX2) */

num_index_keys

→ 인덱스: 고객번호 + 상품ID

Select /*+ num_index_keys (a 고객별가입상품_x1) */

from 고객별가입상품 a

where 고객번호 = :고객번호 → 액세스

인덱스 첫번째 컬럼까지만
액세스 조건으로 사용하라는 의미

and 상품ID in ('a01', 'a04', 'a09') → 필터

→ 상품ID까지 인덱스조건으로 사용하고 싶을 때

/*+ num_index_keys (a 고객별가입상품_x1) */

• Like보다 Between을 사용하는게 낫다.

where 판매월 between '201901' and '201912'

and 판매구분 = 'B'

판매월 판매구분	201812	201901	201912		
	A	B	A	B	A	B
			B	A	B	A

조건절 1 (BETWEEN)

조건절 2 (LIKE)

where 판매월 like '2019%'

and 판매구분 = 'B'

201900이 저장돼있다면 그 값도 ↗

읽어야 하므로 판매구분 = 'B'로 바로 못감

OR

• 인덱스 선두컬럼에 대한 음번조건에 or 조건 사용xx → where (:고객ID IS null or 고객ID=:고객ID)

- 인덱스 액세스 조건으로 사용불가 - 유일한 장점: 음번 조건컬럼이 null이 허용 컬럼이어도 사용가능

- 인덱스 필터 조건으로도 사용불가

- 헤이블 필터 조건으로만 사용가능

like / between

- 인덱스 번득질럼
 - null 허용 걸럼
 - 숫자형 걸럼
 - 가변길이 걸럼
- } between }
} like }

① where 고객id like :고객id || '%'
and 거래일자 between :dt1 and :dt2
→ 인덱스를 거래일자 + 고객id 순으로 구성

② null 허용걸럼에 쓰면안됨 → 조회할때 null 값은 데이터 누락됨

③ - and 고객id like :고객id || '%'
↳ 숫자형걸림이라면 to_char(고객id) ← 이렇게 자동형변환 일어남

union all

select * from 고객 where :cust_id is null and 거래일자 between :dt1 and :dt2

union all

↳ 응선조건걸럼

select * from 고객 where :cust_id is not null and 고객id = :cust_id
and 거래일자 between :dt1 and :dt2

→ 응선조건걸럼도 인덱스 액세스 조건으로 사용, null 허용걸럼도 사용가능

nvl / decode → 여러 개 사용하면 복별적이 가장 좋은 걸럼기준으로 한번만 OR Expansion 일어남
(선택X → 필터조건으로 처리)

→ 어느것을 사용하든 실행계획은 똑같음

· 함수인자로 사용했는데도 인덱스를 사용할 수 있는 이유: OR Expansion 큐리변환이 일어났기 때문

· Null 허용걸럼 사용불가

PL/SQL 사용자 정의 함수가 느린 이유

· 가상머신상에서 실행되는 인터프린터언어

· 호출시마다 컨택스트 스위칭 발생

↳ 함수 x번실행 + 함수내 SQL x번 실행

· 내장 SQL에 대한 Recursive Call 발생 : 가장 결정적인 이유

PL/SQL로 작성한 함수와 프로시저를 컴파일하면 바이트코드 생성 → 데이터 디버깅 불가

→ PL/SQL 엔진(가상머신)만 있으면 어디서든 실행

인덱스 ↑

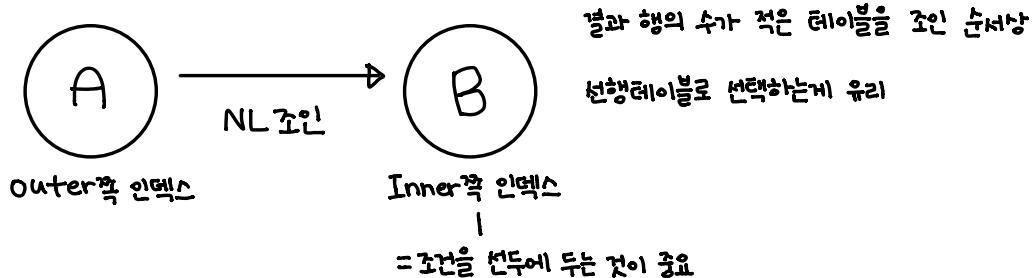
- DML 성능 ↓ (TPS 저하)
- 데이터베이스 사이즈 ↑ (디스크 공간낭비)
- 데이터베이스 관리 & 운영 비용 ↑

가장 정상적이고 일반적인 방식: Index Range Scan

인덱스 구성기준

1. 조건절에 항상 사용하거나 자주 사용하는 컬럼을 선정함
2. = 조건으로 자주 조회하는 컬럼을 앞쪽에 둔다

NL조인



소트연산 생략하기 위한 인덱스

```
select 계약ID, 청약일자, 입력자ID, 계약상태코드, 보험시작일자, 보험종료일자
from 계약
where 취급지점ID = :trt_brch_id
and 청약일자 between :sbcp_dt1 and :sbcp_dt2
and 입력일자 >= trunc(sysdate - 3)
and 계약상태코드 in ( :ctr_stat_cd1, :ctr_stat_cd2, :ctr_stat_cd3 )
order by 청약일자, 입력자ID
```

= 이 아닌 조건절 컬럼은 order by 컬럼보다 뒤쪽에 두어야 함

Ex) 정액일자 + 입력자ID + 입액일자 + 제약상태코드

최적의 인덱스: 취급지점ID + 정액일자 + 입력자ID

I/O 최소화, 소트연산 생략

1. = 연산자로 사용한 조건절컬럼 선정
2. order by 절에 기술한 컬럼추가 조건 만족하는 데이터 ↓ 인덱스추가
3. = 연산자가 아닌 조건절 컬럼은 데이터 분포를 고려해 추가여부 결정

Where 거주지역 = '서울'

and 혈액형 in ('A', 'O')

order by 연령

IN 조건은 =이 아님!

소트연산 생략인덱스: 거주지역 + 연령 + 혈액형

인덱스 병성여부 결정할 때 선택도가 충분히 낮은지가 중요한 판단기준

- 조건절에 의해 선택되는 레코드비율

- 선택도 × 흥레코드수 = 카디널리티

컬럼명	NDV
고소ID	736.000
상태구분코드	3

선택도 높음

NL조인: 양쪽테이블 모두 인덱스이용

Outer or Driving 사원 테이블			Inner 고객 테이블			
사원_X1 인덱스			고객_X1 인덱스			
입사일자	사원번호	사원명	관리사원번호	고객번호	고객명	전화번호
19960101	0001	홍길동	0001	1
19960712	0002	...	0001	2
19970208	0003	김철수	0002	3
19970223	0004	...	0003	4
	0005	이영희	0003	5
	0006	손수희	0004	6
	0007	19960101	0004	7
	0008	...	0005	8
			0006	9
			0006	10
			0007	11
			0008	12

?!) outer → inner ??

변행테이블

→ oo 맞음

- outer쪽 테이블은 사이즈가 크지 않으면 인덱스를 이용하지 않을 수 있다

- 사원 → 고객 (~from 사원 e, 고객 c)

Select /*+ ordered use_nl(B) use_nl(C) use_hash(D) */
 from A,B,C,D
 leading (C,A,D,B)

→ from절에 기술한 순서대로 조인
 ↓ NL방식
 NL방식
 → 해시방식

NL 조인 특징

- 랜덤액세스 위주의 조인방식
 - 한 레코드씩 순차적으로 진행
 - 인덱스 구성을 위해 특히 중요
- }
 → 노드데이터 주로 처리,
 부분범위 처리가 가능함
 온라인 트랜잭션 시스템(OLTP) 시스템에 적합

사원_X1: 입사일자(비트원), 고객_X1 : 관리사원번호

Rows	Row Source Operation
5	NESTED LOOPS
3	TABLE ACCESS BY INDEX ROWID OF 사원
2780	INDEX RANGE SCAN OF 사원_X1
5	TABLE ACCESS BY INDEX ROWID OF 고객
8	INDEX RANGE SCAN OF 고객_X1

→ 인덱스 스캔 후 사원테이블 액세스 횟수: 2780

→ 부서코드 = '2123' 조건 필터링한 결과: 3건

→ 입사일자 + 부서코드

테이블 액세스 후 필터링 되는 비율 ↑ 인덱스에 테이블 필터 조건 걸려 추가 고려

Cr: 처리적인 블록 요청 횟수, Pr: 디스크에서 읽은 블록 수, Pw: 디스크에 쓴 블록 수

Rows	Row Source Operation
5	NESTED LOOPS (cr=112 pr=34 pw=0 time=122 us)
3	TABLE ACCESS BY INDEX ROWID OF 사원 (cr=105 pr=32 pw=0 time=118 us)
3	INDEX RANGE SCAN OF 사원_X1 (cr=102 pr=31 pw=0 time=16)
5	TABLE ACCESS BY INDEX ROWID OF 고객 (cr=7 pr=2 pw=0 time=4 us)
8	INDEX RANGE SCAN OF 고객_X1 (cr=5 pr=1 pw=0 time=0 us)

→ 인덱스로부터 읽은 블록: 102 → 푸닝방법: 인덱스필터값에서 조정: 부서코드 + 입사일자

Rows	Row Source Operation
5	NESTED LOOPS (cr=2732 pr=386 pw=0 time=...)
2780	TABLE ACCESS BY INDEX ROWID 사원 (cr=166 pr=2 pw=0 time=...)
2780	INDEX RANGE SCAN 사원_X1 (cr=4 pr=0 pw=0 time=...)
5	TABLE ACCESS BY INDEX ROWID 고객 (cr=2566 pr=384 pw=0 time=...)
8	INDEX RANGE SCAN 고객_X1 (cr=2558 pr=383 pw=0 time=...)

→ 고객테이블과 조인하는 횟수: 2780번

→ 고객테이블 조인 후 5건으로 줄어든 이유: 사원테이블로부터 사원번호와 고객테이블의 최종주문금액 조건절

조합했기 때문

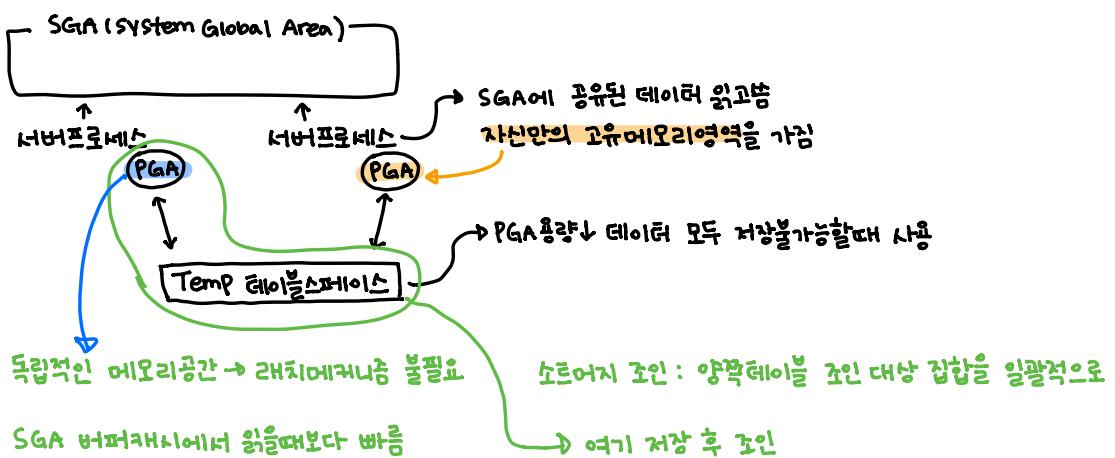
?) inner 테이블 alias 왼쪽에!

Select * from (사원 a, 고객 b)
 where a.사원아이디 = :id
 and b.사원아이디 = a.사원아이디
 and b.xx = a.xx

) 이유를 모르겠음 ㅠㅠ

소트어지 조인

→ 조인컬럼 인덱스 X, 대량데이터 조인이어서 인덱스가 효과적이지 않을 때 옵티마이저가 선택함



노트어지 조인은

1. 노트단계: 양쪽 집합을 조인 컬럼 기준으로 정렬
2. 머지단계: 정렬한 양쪽집합 서로 머지

→ 노트어지: 정렬된 사원기준으로 정렬된 고객화 조인

Select /*+ ordered use_merge(c) */

① e.사원번호, e.사원명, e.입사일자

② , c.고객번호, c.고객명, c.전화번호, c.회원증금액
from 사원 e, 고객 c

where c.관리사원번호 = e.사원번호

and e.입사일자 >= '19960101'

and e.부서코드 = '2123' + order by 사원번호

and c.회원증금액 >= 20000 + order by 관리사원번호

결과집합은 PGA 영역에
할당된 Sort Area에 저장

③ 머지: NL조인과 비슷! 다른점: 사원데이터를 기준으로 고객데이터 풀스캔 안함

→ 1, 2: 노트단계, 3: 머지단계

주용도

- 조인 조건식 등치 (=) 조건이 아닌 대량데이터 조인
- 조인 조건식이 아예 없는 조인 (크로스조인)

양쪽집합 정렬 → NL조인 PGA영역에 저장한 데이터이용 (빠름)

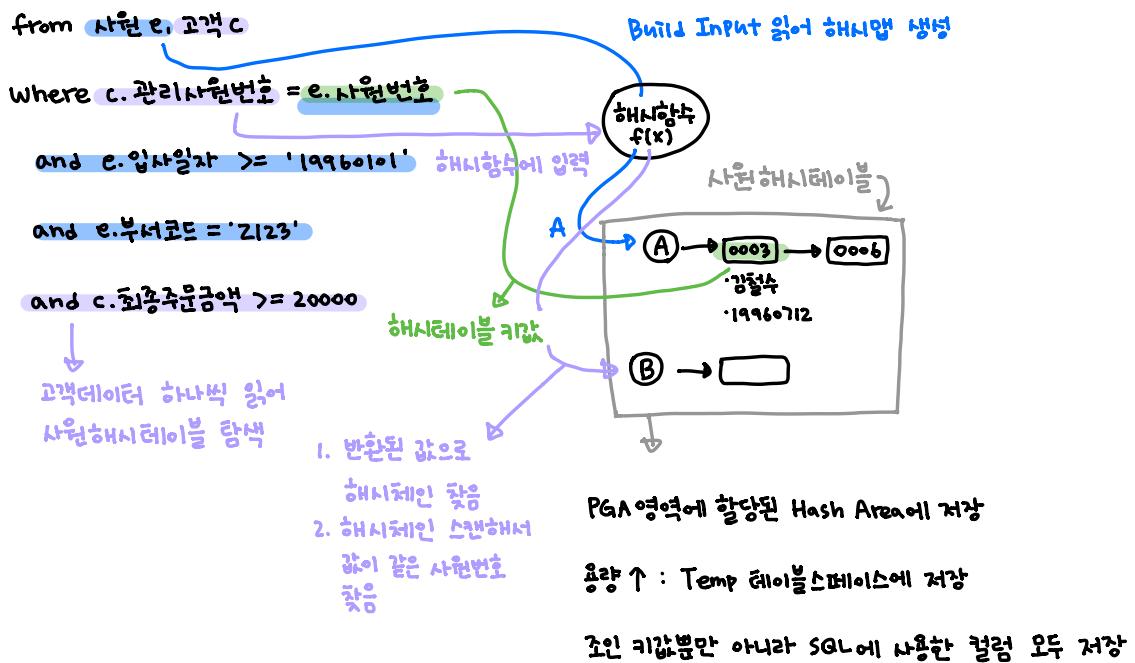
- 인덱스 유무 영향 X
- 스캔위주 액세스 방식 사용 → 모든 처리가 스캔방식은 아님!

해시 조인

1. Build 단계: 작은 쪽 테이블 (Build Input) 을 읽어 해시 테이블 (해시 맵) 생성
2. Prod 단계: 큰 쪽 테이블 (prod Input) 을 읽어 해시 테이블 탐색하면서 조인 = NL조인

Select /*+ordered use_hash(c) */

- ① e.사원번호, e.사원명, e.입사일자
- ② c.고객번호, c.고객명, c.전화번호, c.회원증금액



* PGA는 그리 큰 메모리 공간 X 어느하나 중대형 이상이면 Temp 테이블스페이스 작업 수행

디스크에 쓰는 작업

* 해시는 둘 중 작은 집합 → 해시맵 Build Input 선택 .. 디스크에 쓰는 작업 거의 XX

IN
 PGA

인메모리 해시조인일때 가장 효과적

Temp 테이블스페이스 써도 계일반증

HASH JOIN

TABLE ACCESS (BY INDEX ROWID) OF '사원' (TABLE) → 사원 데이터로 해시레이블 생성

INDEX (RANGE SCAN) OF '사원_X1' (INDEX)

TABLE ACCESS (BY INDEX ROWID) OF '고객' (TABLE) → 고객레이블 조인 키값으로 해시레이블 탐색하면서
조인

INDEX (RANGE SCAN) OF '고객_N1' (INDEX)

세 개 이상 레이블 해시 조인

경로 1: A ↔ B ↔ C

경로 2: A ↔ B C = B ↔ A ↔ C ∵ 조인 경로는 하나임!
 ↑ ↑ ↑
 T1 T2 T3

/* + leading (T1, T2, T3) use_hash(T2) use_hash(T3) */

↳ 첫번째 파라미터로 지정한 레이블은 무조건 Build Input으로 선택됨

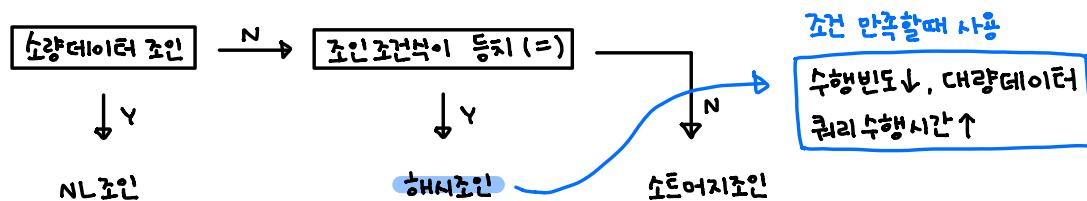
/* + leading (T1, T2, T3) swap_join_inputs (T2) */

↳ Build Input T2로 변경

T1과 T2 조인한 결과집합을 Build Input으로 선택하고 싶을 때

T3을 Probe Input으로 선택해 주는 방식

/* + leading (T1, T2, T3) no_swap_join_inputs (T3) */



- 인덱스 영구적 유지
- 해시레이블 → 단 하나의 쿼리를 위해 생성하고 조건이 끝나면 곧바로 소멸

및 공유 및 재사용
∴ 수행시간↓ 수행빈도↑ 해시조인으로 처리 → CPU, 메모리 사용률 ↑↑

서브쿼리와 조인

필터오퍼레이션: no_unnest 힌트 사용

↳ NL조인과 처리 루틴이 같음

차이점

1. 메인쿼리의 한 로우가 서브쿼리의

한 로우와 조인에 성공하는 순간 진행멈춤

→ 메인쿼리의 다음 로우를 계속 처리함

2. 필터는 캐싱기능 가짐 ↗ 쿼리단위로 이루어짐

쿼리시작 → PGA 메모리 공간 할당 → 쿼리수행

→ 공간채우 → 쿼리마침 → 공간반환

Select c.고객, c.고객명

from 고객 c → 메인: 드라이빙 집합

Where c.가입일시 >= trunc(add-month(sysdate,-1), 'mm')

and exists (

select /*+ no_unnest */ 'x'

from 거래

Where 고객번호 = c.고객번호

and 거래일시 >= trunc(sysdate, 'mm')

)

서브쿼리 Unnesting

• 메인과 서브쿼리 간의 계층구조를 풀어 서로 같은 레벨로 만들어준다는 의미

= 서브쿼리 Flattening

/ *+ unnest nl_sq */ 오라10g 세미조인이 캐싱기능도 갖게됨

↳ 일반조인문처럼 다양한 최적화 기법 사용 가능

Unnesting은 서브쿼리는 메인 쿼리 집합보다 먼저 처리될 수 있음

Select /*+ leading(거래@subq) use_nl(c) */ c.고객, c.고객명

from 고객 c

Where c.가입일시 >= trunc(add-month(sysdate,-1), 'mm')

★ rownum

and exists (

select /*+ qb_name(subq) unnest */ 'x'

서브쿼리 Unnesting을 방지하려는

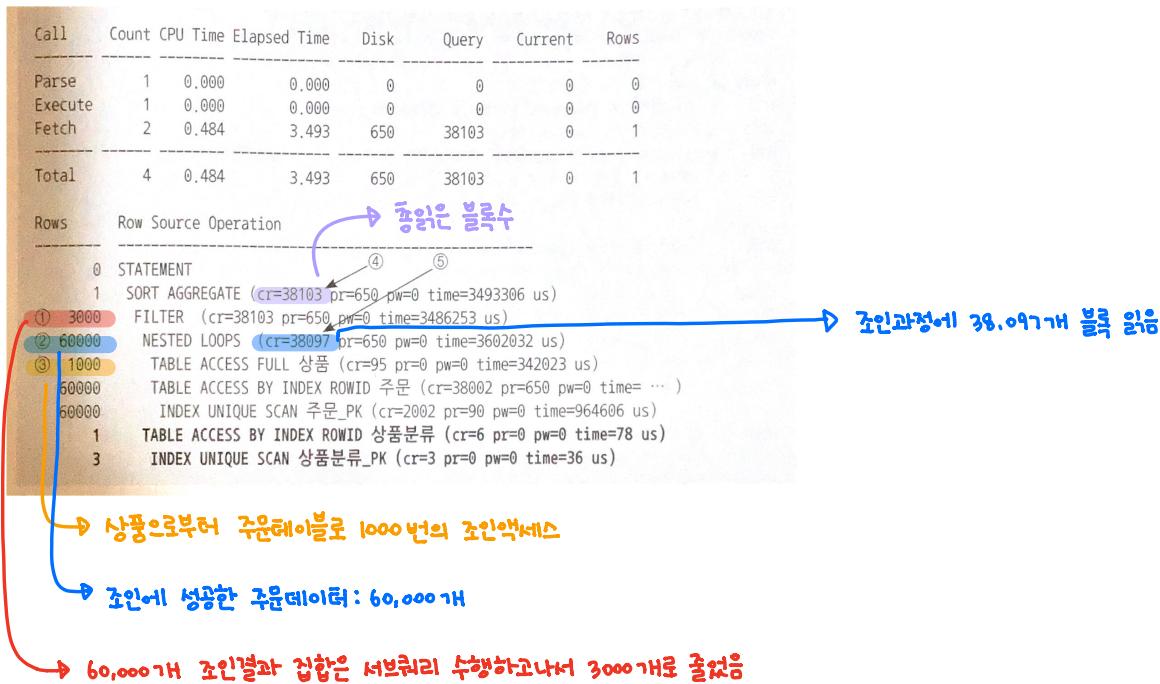
from 거래

목적이 아니면 서브쿼리에 항부로 쓰지 말것

서브쿼리 Pushing

→ 조인후 서브쿼리 필터링

```
select /*+ leading(p) use_nl(t) */ count(distinct p.상품번호), sum(t.주문금액)
from 상품 p, 주문 t
where p.상품번호 = t.상품번호
and p.등록일시 >= trunc(add_months(sysdate, -3), 'mm')
and t.주문일시 >= trunc(sysdate - 7)
and exists (select 'x' from 상품분류
            where 상품분류코드 = p.상품분류코드
            and 상위분류코드 = 'AK' )
```



```
select /*+ leading(p) use_nl(t) */ count(distinct p.상품번호), sum(t.주문금액)
from 상품 p, 주문 t
where p.상품번호 = t.상품번호
and p.등록일시 >= trunc(add_months(sysdate, -3), 'mm')
and t.주문일시 >= trunc(sysdate - 7)
and exists (select /*+ NO_UNNEST PUSH_SUBQ */ 'x' from 상품분류
            where 상품분류코드 = p.상품분류코드
            and 상위분류코드 = 'AK' )
```

Call	Count	CPU Time	Elapsed Time	Disk	Query	Current	Rows
Parse	1	0.000	0.000	0	0	0	0
Execute	1	0.000	0.000	0	0	0	0
Fetch	2	0.125	0.129	0	1903	0	1
Total	4	0.125	0.129	0	1903	0	1

```

select /*+ leading(p) use_nl(t) */ count(distinct p.상품번호), sum(t.주문금액)
from 상품 p, 주문 t
where p.상품번호 = t.상품번호
and p.등록일시 >= trunc(add_months(sysdate, -3), 'mm')
and t.주문일시 >= trunc(sysdate - 7)
and exists (select /*+ NO_UNNEST PUSH_SUBQ */ 'x' from 상품분류
            where 상품분류코드 = p.상품분류코드
            and 상위분류코드 = 'AK' )

```

Call Count CPU Time Elapsed Time Disk Query Current Rows

Parse	1	0.000	0.000	0	0	0	0
Execute	1	0.000	0.000	0	0	0	0
Fetch	2	0.125	0.129	0	1903	0	1
Total	4	0.125	0.129	0	1903	0	1

Rows Row Source Operation → 총 읽은 블록수

0 STATEMENT	1 SORT AGGREGATE (cr=1903 pr=0 pw=0 time=128934 us)	3000 NESTED LOOPS (cr=1903 pr=0 pw=0 time=153252 us)
① 150 TABLE ACCESS FULL 상품 (cr=101 pr=0 pw=0 time=18230 us)	1 TABLE ACCESS BY INDEX ROWID 상품분류 (cr=6 pr=0 pw=0 time=135 us)	3 INDEX UNIQUE SCAN 상품분류_PK (cr=3 pr=0 pw=0 time=63 us)
② 3000 TABLE ACCESS BY INDEX ROWID 주문 (cr=1802 pr=0 pw=0 time=100092 us)	3000 INDEX RANGE SCAN 주문_PK (cr=302 pr=0 pw=0 time=41733 us)	

→ 서브쿼리를 필터링한 결과 : 150건, 조인횟수 : 150건

→ 주문데이터 3000 건 읽음

뷰와 조인

최적화 단위: 쿼리블록 → 익스피리언서가 뷰 쿼리 반환 X → 뷰 쿼리 독립적으로 최적화

```

select c.고객번호, c.고객명, t.평균거래, t.최소거래, t.최대거래
from 고객 c
  ,(select 고객번호, avg(거래금액) 평균거래
        , min(거래금액) 최소거래, max(거래금액) 최대거래
     from 거래
    where 거래일시 >= trunc(sysdate, 'mm') -- 당월 발생한 거래
   group by 고객번호) t
where c.가입일시 >= trunc(add_months(sysdate, -1), 'mm') -- 전월 이후 가입 고객
and t.고객번호 = c.고객번호

```

Execution Plan

0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1M Card=1K Bytes=112K)	→ 전월 이후 가입한 고객을
1 0 NESTED LOOPS	필터링하는 조건이 인라인뷰
2 1 NESTED LOOPS (Cost=1M Card=1K Bytes=112K)	반환에 있음
3 2 VIEW (Cost=2K Card=427K Bytes=21M)	→ 인라인뷰안에서 당월에
4 3 HASH (GROUP BY) (Cost=2K Card=427K Bytes=14M)	거래한 모든 고객의 거래
5 4 TABLE ACCESS (BY INDEX ROWID) OF '거래' (TABLE) (Cost=2K ...)	데이터 가져옴
6 5 INDEX (RANGE SCAN) OF '거래_X01' (INDEX) (Cost=988 Card=427K)	
7 2 INDEX (RANGE SCAN) OF '고객_X01' (INDEX) (Cost=1 Card=190)	
8 1 TABLE ACCESS (BY INDEX ROWID) OF '고객' (TABLE) (Cost=3 Card=1 ...)	

→ merge 힌트를 이용해 뷰를 메인쿼리와 머징 ↔ no-merge

```
select c.고객번호, c.고객명, t.평균거래, t.최소거래, t.최대거래  
from 고객 c  
(select /*+ merge */ 고객번호, avg(거래금액) 평균거래  
     , min(거래금액) 최소거래, max(거래금액) 최대거래  
   from 거래  
  where 거래일시 >= trunc(sysdate, 'mm')  
 group by 고객번호) t  
where c.가입일시 >= trunc(add_months(sysdate, -1), 'mm')  
and t.고객번호 = c.고객번호
```

• 조인에 성공한 전제집합

을 그룹바이 해야됨

Execution Plan

```
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=1 Bytes=27)  
1      0  HASH (GROUP BY) (Cost=4 Card=1 Bytes=27)  
2      1  NESTED LOOPS (OUTER) (Cost=3 Card=5 Bytes=135)  
3      2    TABLE ACCESS (BY INDEX ROWID) OF '고객' (TABLE) (Cost=2 Card=1 ... )  
4      3    INDEX (RANGE SCAN) OF '고객_X01' (INDEX) (Cost=1 Card=1)  
5      2    TABLE ACCESS (BY INDEX ROWID) OF '거래' (TABLE) (Cost=1 Card=5 ... )  
6      5    INDEX (RANGE SCAN) OF '거래_X02' (INDEX) (Cost=0 Card=5)
```

→ 단점: 조인에 성공한 전제집합을 그룹바이 하고나서 데이터 출결가능

→ 부분처리 불가능

조인조건 Pushdown

메인쿼리를 실행하면서 조인 조건절 값을 건전이 뷰 안으로 밀어넣는 가능

Execution Plan

```
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=1 Bytes=61)  
1      0  NESTED LOOPS (OUTER) (Cost=4 Card=1 Bytes=61)  
2      1    TABLE ACCESS (BY INDEX ROWID BATCHED) OF '고객' (TABLE) (Cost=2 ... )  
3      2    INDEX (RANGE SCAN) OF '고객_X01' (INDEX) (Cost=1 Card=1)  
4      1    VIEW PUSHED PREDICATE (Cost=2 Card=1 Bytes=41)  
5      4    SORT (GROUP BY) (Cost=2 Card=1 Bytes=7)  
6      5    TABLE ACCESS (BY INDEX ROWID BATCHED) OF '거래' (TABLE) (Cost=2 ... )  
7      6    INDEX (RANGE SCAN) OF '거래_X02' (INDEX) (Cost=1 Card=5)
```

전월 이후 가입한 고객을 대상으로 건전이 당월거래 데이터만 읽어서 조인하고 그룹바이 실행

• 부분범위 처리가능

• no-merge + Push-Pred

스칼라 서브쿼리 조인

```
Select get_dname(e.deptno) dname
```

↳ 함수안에 있는 select 쿼리 → 메인쿼리 건수만큼 재귀적으로 반복실행

```
Select empno, (select d.name from dept d where d.deptno = e.deptno) dname
```

↳ 스칼라서브쿼리는 메인쿼리 레코드마다 하나의 값 반환, 재귀적으로 실행하는 구조X

캐싱작용 일어남

스칼라서브쿼리 캐싱효과 ↗ 필터서브쿼리 캐싱과 같은 기능

- 입력값, 출력값 → 내부캐시에 저장

캐시 → 입력값 찾음 ↗ 저장된값 반환

N ↗ 조인실행 (결과는 캐시에 저장)

캐싱: 쿼리단위로 이루어짐 / 쿼리시작할때 PGA 메모리에 공간 할당 → 쿼리수행 → 공간해제 → 쿼리마침 →

공간반환

```
Select (select get_dname(e.dept) from dual) dname
```

↳ 호출횟수 최소화

스칼라 서브쿼리 캐싱효과: 입력값의 종류 → 소수여서 해시충돌 가능성 ↓ 효과有

↑
반대 ↗ CPU 사용률 ↑ 메모리 사용 ↑

```
Select (select get_dname(e.dept) from dual) dname
```

↳ e.dept가 무수히 많으면 스칼라서브쿼리 캐싱효과 X 성능↓↓

- 메인쿼리 집합↑ 재사용성↑ 효과↑

- 메인쿼리 집합↓ 캐시 재사용성↓

스칼라서브쿼리 Unnesting ↗ NL방식 / 캐싱효과 크지않으면 랜덤 I/O 부담이 있음

/* + unnest */ → 이 힌트쓰면 해시조인으로 실행됨