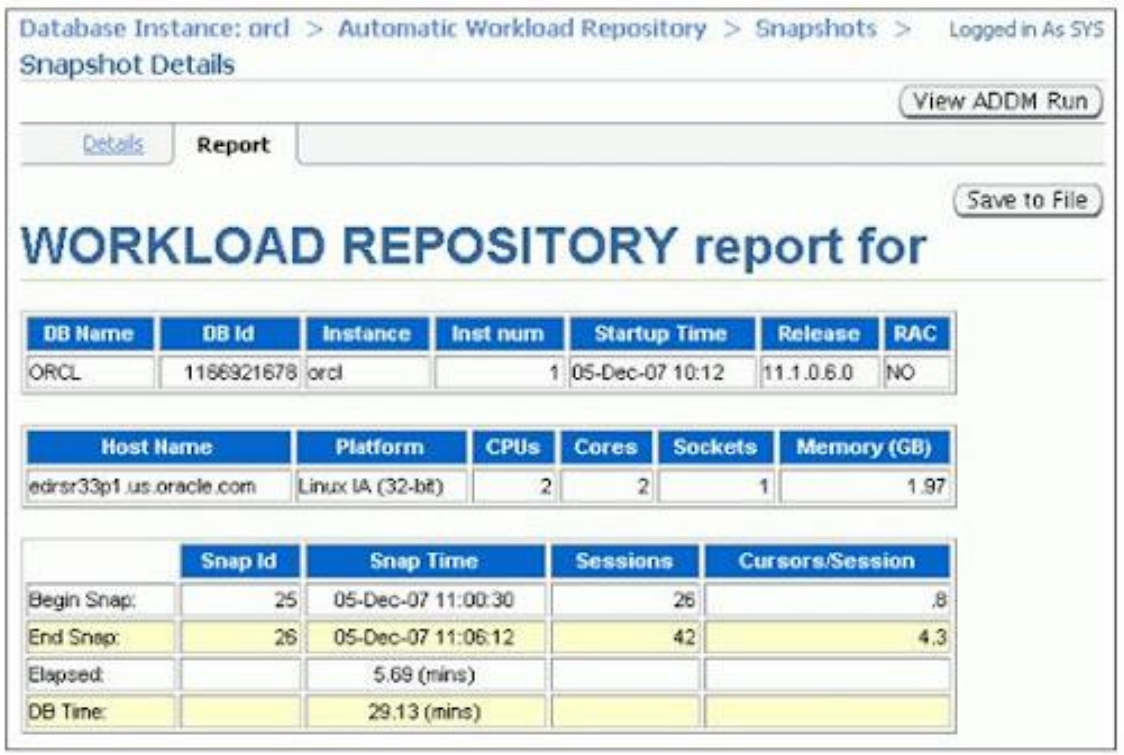
13. SQL 튜닝 실습

(1) 튜닝 대상 SQL 식별

AWR 리포트를 통한 주요 SQL 식별

- AWR(Auto Workload Repository) 리포트는 일정 기간 동안의 데이터베이스 활동을 수집하여 제공하는 보고서입니다.
- 이 보고서를 통해 CPU 사용량, 논리적 읽기, 디스크 읽기, 수행 시간 기준으로 가장 리소스를 많이 소비한 SQL을 식별할 수 있습니다.
- SQL ordered by Elapsed Time, Buffer Gets, Executions 등의 섹션을 분석하는 것이 중요합니다.



Database Instance: orcl > Automatic Workload Repository > Snapshots > Logged in As SYS

Snapshot Details

View ADDM Run

Details | Report

Save to File

WORKLOAD REPOSITORY report for

DB Name	DB Id	Instance	Inst num	Startup Time	Release	RAC
ORCL	1166921678	orcl	1	05-Dec-07 10:12	11.1.0.6.0	NO

Host Name	Platform	CPUs	Cores	Sockets	Memory (GB)
edrsr33p1.us.oracle.com	Linux IA (32-bit)	2	2	1	1.97

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	25	05-Dec-07 11:00:30	26	.8
End Snap:	26	05-Dec-07 11:06:12	42	4.3
Elapsed:		5.69 (mins)		
DB Time:		29.13 (mins)		

V$SQL 뷰와 Active Session History(ASH)

- V$SQL 뷰를 이용한 실시간 SQL 탐색
  - V$SQL 뷰는 공유 풀에 캐시된 SQL 실행 정보들을 실시간으로 제공합니다.
  - BUFFER_GETS, DISK_READS, ELAPSED_TIME, EXECUTIONS 등의 컬럼을 분석하여 많은 리소스를 사용하는 SQL 문장을 확인할 수 있습니다.
  - 필요 시 SQL_TEXT를 조회하여 문제 SQL을 직접 식별할 수 있습니다.
- Active Session History(ASH) 분석
  - ASH 뷰(V$ACTIVE_SESSION_HISTORY)는 특정 시간대에 세션이 어떤 SQL을 수행 중이었는지를 보여줍니다.
  - 과거 특정 시간 동안 병목이 발생한 SQL을 추적하고, SQL_ID 기준으로 성능 저하 원인을 파악할 수 있습니다.
  - AWR보다 더 세밀한 시간대 분석이 가능합니다.

실행계획 확인 및 SQL Monitor

- EXPLAIN PLAN을 통한 실행 계획 확인
  - EXPLAIN PLAN FOR 구문과 DBMS_XPLAN.DISPLAY를 사용하면 SQL 실행 전 예상 실행 계획을 확인할 수 있습니다.
  - 인덱스 미사용, TABLE FULL SCAN, NESTED LOOP 과다 등 비효율적인 실행 계획을 통해 튜닝 대상을 사전 판단할 수 있습니다.
- SQL Monitor를 활용한 장시간 SQL 추적
  - DBMS_SQLTUNE.REPORT_SQL_MONITOR 함수를 사용하면 오래 실행된 SQL의 상세 실행 정보를 시각적으로 확인할 수 있습니다.
  - 각 단계별 수행 시간, IO량, 병렬 처리 정보 등을 통해 어떤 단계에서 병목이 발생하는지 정확히 확인할 수 있습니다.

(2) 조인과 서브쿼리 튜닝

조인 유형 개요

- 데이터베이스에서 조인은 두 개 이상의 테이블을 연결하여 원하는 데이터를 추출하는 핵심 연산입니다.
- 옵티마이저는 조인 대상 테이블의 크기, 인덱스, 필터 조건 등을 바탕으로 최적의 조인 방식을 선택합니다.
  - NL 조인(Nested Loop Join): 중첩 루프 방식으로, 한 테이블(Outer)의 각 행마다 다른 테이블(Inner)에서 조건에 맞는 행을 탐색해 조인한다. 인덱스를 활용할 수 있어 소량 데이터나 OLTP 환경에 적합합니다.
  - 소트 머지 조인(Sort Merge Join): 조인 대상 테이블을 모두 정렬한 후, 정렬된 데이터를 병합하는 방식. 대량 데이터 처리에 유리하며, 등치 조건 외의 조인에도 사용됩니다.
  - 해시 조인(Hash Join): 한 테이블을 해시 테이블로 만든 뒤, 다른 테이블의 데이터를 해시 테이블과 비교해 조인한다. 대량 데이터, 특히 DW/OLAP 환경에서 효과적입니다.
- 조인 튜닝은 이러한 방식의 특성을 이해하고, 상황에 맞게 조인 방법을 유도하거나 비효율을 줄이는 데 목적이 있습니다.

NL (Nested Loop) 조인

- 기본 매커니즘
  - NL 조인은 Outer 테이블의 각 행마다 Inner 테이블에서 조인 조건에 맞는 행을 검색하는 방식입니다.
  - 일반적으로 양쪽 테이블 모두 인덱스를 사용하며, 특히 Inner 테이블의 인덱스 유무가 성능에 큰 영향을 줍니다.
  - 프로그래밍의 2중 for문 구조와 유사합니다.
- NL 조인 실행계획 제어
  - USE_NL 힌트로 NL 조인을 강제할 수 있습니다.
  - 또한 ORDERED, LEADING 힌트로 조인 순서를 제어할 수 있습니다.
- NL 조인은 랜덤 액세스 위주로 한 레코드씩 순차적으로 진행됩니다.
- 인덱스 전략이 매우 중요하며, 소량 데이터, OLTP, 부분범위 처리에 적합합니다.

NL 조인 수행과정과 튜닝 포인트

- NL 조인 수행과정 분석

  1) Outer 테이블에서 조건에 맞는 첫 번째 레코드를 선택합니다.

  2) 인덱스를 통해 해당 레코드의 ROWID로 실제 데이터를 접근합니다.

  3) 해당 값으로 Inner 테이블의 인덱스를 탐색하고 ROWID로 실제 데이터를 접근합니다.

  4) 조건에 맞는 경우 결과를 반환하며, Outer의 모든 레코드에 대해 반복합니다.

- NL 조인 튜닝 포인트

  - Inner 테이블의 조인 컬럼에 인덱스가 있는지 확인하고 최적화해야 합니다.
  - Outer 테이블의 필터링 효율이 낮으면 인덱스 컬럼 추가를 고려해야 합니다.
  - NL 조인은 랜덤 액세스가 많으므로 블록 I/O를 최소화하는 설계가 필요합니다.

NL 조인 튜닝 사례

- 드라이빙 테이블 최적화
  - 작은 결과 집합을 가진 테이블을 드라이빙 테이블로 선택해야 합니다.

```
/*+ ORDERED USE_NL(B) */
SELECT A.col1, B.col2
FROM (SELECT * FROM large_table WHERE key = 100) A  -- 결과 10건
JOIN detail_table B ON A.id = B.parent_id           -- 결과 10,000건
```

  - LEADING 힌트로 드라이빙 테이블을 명시합니다.

```
/*+ LEADING(A B) USE_NL(B) */
SELECT A.col1, B.col2
FROM (SELECT * FROM large_table WHERE key = 100) A   -- 결과 10건
JOIN detail_table B ON A.id = B.parent_id            -- 결과 10,000건
```

NL 조인 튜닝 사례

- 인덱스 전략
  - Inner 테이블의 조인 컬럼 인덱스는 필수이며, “인덱스 컬럼 순서 = 조인 컬럼 순서”로 구성해야 합니다.
  - position 조건이 인덱스 뒤쪽에 위치해 효율성 저하되는 문제상황 입니다.

```
/* 인덱스: (dept_id, position) */
SELECT e.name, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id AND e.position = 'Manager';
```

  - 필터 조건은 WHERE절에 명시하는 것이 일반적으로 인덱스 활용 측면에서 바람직합니다.

```
SELECT e.name, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.position = 'Manager';
```

소트 머지 조인

- 기본 매커니즘
  - 소트 머지 조인은 조인 대상 테이블을 조인 컬럼 기준으로 모두 정렬한 후, 정렬된 데이터를 병합하여 조인 결과를 생성하는 방식입니다.
- 주요 용도
  - 대량 데이터 조인 시 효율적이며, 등치(=) 외의 조건(>, <, BETWEEN 등)에도 적용할 수 있습니다.
  - 인덱스가 없거나 인덱스를 활용하기 어려운 경우에 적합합니다.
- 소트 머지 조인 제어
  - 옵티마이저가 자동으로 선택하거나, 힌트(USE_MERGE)로 강제할 수 있습니다.
- 정렬 비용이 발생하지만, 정렬 후에는 순차적 병합으로 효율적입니다.
- 인덱스 미사용 시에도 대량 데이터 조인에 강점이 있으며, 등치 외 조인 조건도 지원합니다.

소트 머지조인의 사용

- 대용량 정렬 최적화
  - 메모리 사이즈 조정으로 PGA 메모리 활용도 향상 가능
- 비등가 조인 사례

```
select e.employee_id, e.salary, j.grade
FROM employees e JOIN job_grades j
ON (e.salary BETWEEN j.lowest_sal AND Highest_sal);
```

해시 조인

- 기본 매커니즘
  - 해시 조인은 한 테이블(일반적으로 작은 쪽)을 메모리에 해시 테이블로 생성(Build Phase)하고, 다른 테이블의 각 행을 해시 함수로 변환하여 해시 테이블에서 매칭(Probe Phase)하는 방식입니다.
- 해시 조인이 빠른 이유
  - 대량 데이터 조인 시 Full Scan과 해시 탐색으로 랜덤 액세스를 최소화하기 때문입니다.
  - 한 번의 스캔으로 조인 결과를 생성할 수 있습니다.
- 해시 조인 실행계획 제어
  - USE_HASH 힌트로 해시 조인을 강제할 수 있습니다.
  - 일반적으로 대량 데이터, 인덱스 미존재, OLAP/배치 환경에서 사용됩니다.

해시 조인으로 대용량 배치 예시

- 성능 이점:
  - 대량 데이터에서 NL 조인 대비 10~30배 빠른 처리.
  - 인덱스 의존성 없이 풀 스캔 최소화

```
/*+ FULL(A) FULL(B) USE_HASH(B) */
SELECT A.customer_id, SUM(B.amount)
FROM customers A
JOIN transactions B ON A.id = B.cust_id  -- 100만 건 이상
GROUP BY A.customer_id
```

  - Build Phase: customers 테이블(소량)을 메모리에 해시 테이블로 로드합니다.
  - Probe Phase: transactions 테이블(대량)을 풀 스캔하며 해시 매칭을 수행합니다.

조인 메소드 선택 기준

상황	추천 조인 방식	비고
소량 데이터, OLTP	NL 조인	인덱스 활용, 빠른 응답
대량 데이터, DW/OLAP/배치	해시 조인	Full Scan + 해시, 랜덤 액세스 최소화
등치 외 조인 조건	소트 머지 조인	정렬 후 병합, 인덱스 미활용 가능
빈도 높은 쿼리, 성능 유사	NL 조인	해시 조인보다 NL 조인 선호
해시 조인이 월등히 빠른 경우	해시 조인	메모리 사용량 주의, 동시성 고려 필요

- 해시 조인은 대량 데이터, 수행 빈도 낮은 쿼리에만 사용하는 것이 권장됩니다.
- NL 조인은 인덱스 구성과 소량 데이터에 최적화되어 있습니다.
- 소트 머지 조인은 등치 외 조건, 인덱스 미존재, 대량 데이터에 적합합니다.

조인 메소드 혼용 사례

- 다중 테이블 조인 전략
  - NL 조인 적용: orders(1만 건) → products(소량) 조인 시 인덱스 활용 효율화.
  - 해시 조인 전환: suppliers(100만 건)와 조인 시 메모리 기반 고속 처리

```
SELECT /*+ LEADING(orders products)
           USE_NL(products)
           USE_HASH(suppliers) */
  orders.order_id,
  products.name,
  suppliers.address
FROM orders
JOIN products ON orders.prod_id = products.id      -- 소량: NL 조인
JOIN suppliers ON products.sup_id = suppliers.id   -- 대량: 해시 조인
WHERE orders.region = 'APAC';
```

조인튜닝의 체크포인트

- 조인 튜닝은 데이터 규모·분포·인덱스를 종합적으로 평가해야 합니다.
- NL 조인은 인덱스 최적화가, 해시 조인은 메모리 관리가, 소트 머지 조인은 정렬 효율화가 핵심 포인트입니다.

1) 드라이빙 테이블
   - 결과 행이 가장 적은 테이블 선정
2) 조인 순서
   - ORDERED/LEADING 힌트로 순서 강제
3) 인덱스 점검
   - 조인 컬럼 인덱스 존재 여부 확인
4) 조인 방식 선택
   - 소량: NL, 대량: 해시, 비등가: 소트 머지
5) 실행 계획 검증
   - EXPLAIN PLAN으로 랜덤 액세스 발생 지점 분석

서브쿼리 변환의 필요성

- 옵티마이저 한계:
  - 서브쿼리는 독립적인 쿼리 블록 단위로 최적화되므로 전체 쿼리 최적화에 한계가 있습니다.
- 성능 개선:
  - 서브쿼리를 조인으로 변환하면 다양한 최적화 기법(해시 조인, 머지 조인) 적용이 가능해집니다.
- 실행 계획 제어:
  - UNNEST 힌트로 서브쿼리를 조인 형태로 변환해 실행 계획을 통제할 수 있습니다.

서브쿼리와 조인

- 필터 오퍼레이션:
  - 쿼리를 NO_UNNEST로 처리하면 NL 조인 방식과 유사하게 동작하지만, 조인 성공 시 바로 중단됩니다.
  - 캐싱 기능으로 동일 입력값 재사용이 가능합니다.
- Unnesting:
  - 서브쿼리를 일반 조인으로 변환(USE_HASH, USE_MERGE 적용 가능)해 대량 데이터 처리에 효율적입니다.
  - 예시: SELECT /*+ UNNEST */ ....
- Pushing:
  - PUSH_SUBQ 힌트로 서브쿼리 필터링을 먼저 수행해 처리량을 줄입니다

스칼라 서브쿼리 조인

- 특징:
  - 단일 값 반환으로 함수처럼 동작하며, 입력값/출력값을 캐싱해 성능을 개선합니다.
  - 예시:

```
SELECT name, (SELECT dept_name FROM dept WHERE dept_id = e.dept_id) FROM emp e.
.....
```

- 성능 고려사항:
  - 장점: 소량 데이터 또는 입력값 종류가 적을 때 효율적입니다.
  - 단점: 입력값 종류가 많으면 캐싱 오버헤드가 발생하고, 다중 컬럼 반환 시 문자열 결합 등 우회 기법이 필요합니다.
  - Unnesting: _optimizer_unnest_scalar_sq 파라미터로 스칼라 서브쿼리를 일반 조인으로 변환할 수 있습니다.

서브쿼리 vs 조인 성능 비교

기준	서브쿼리	조인
가독성	복잡한 쿼리 구조화에 유리	간결한 구문
성능	소량 데이터 시 유리	대량 데이터 시 유리(I/O 최소화)
사용 시나리오	계층형 쿼리, 캐싱 필요 시	집계, 대량 데이터 병합

서브쿼리를 조인으로 변환 예시

- 서브쿼리 내 집약(SUM, COUNT) 후 조인으로 레코드 수를 줄이면 성능이 개선됩니다

```
-- 비효율적
SELECT d.department_name, SUM(e.salary)
FROM departments d
JOIN employees e ON d.department_id = e.department_id
WHERE e.job_id LIKE 'IT%'
GROUP BY d.department_name;

-- 효율적
SELECT d.department_name, agg.total_salary
FROM departments d
JOIN (
    SELECT department_id, SUM(salary) AS total_salary
    FROM employees
    WHERE job_id LIKE 'IT%'
    GROUP BY department_id
) agg ON d.department_id = agg.department_id;
```

서브쿼리 최적화 전략

- 불필요한 서브쿼리 제거:
  - 단순 필터링은 WHERE IN 대신 EXISTS나 조인으로 대체.
- 윈도우 함수 활용:
  - 서브쿼리 의존성을 ROW_NUMBER() 등으로 해결해 테이블 접근 횟수 감소
- 인라인 뷰 최소화:
  - 뷰 내부에서 불필요한 컬럼/조건을 제거해 데이터 처리량 감소

```
-- 서브쿼리 대신 윈도우 함수 사용
SELECT employee_id, department_id, first_name, hire_date
FROM (SELECT employee_id, department_id, first_name, hire_date,
             ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY hire_date) rn
      FROM employees)
WHERE rn = 1;
```

요약

- 조인 방식은 데이터 양, 인덱스 존재 여부, 조인 조건에 따라 신중히 선택해야 합니다.
- 서브쿼리와 조인 선택은 데이터 규모, 인덱스 여부, 결과 집합 특성을 종합적으로 평가해야 합니다.
- 서브쿼리는 구조화에 유리하나, 대량 데이터 시 조인 변환을 통해 I/O 비용을 절감할 수 있습니다.
- 튜닝 시에는 실행계획과 I/O, 메모리 사용량을 반드시 점검하는 것이 중요합니다.

(3) SQL 집계와 조건 분기를 활용한 쿼리 최적화

집계 함수와 그룹화의 원리

- SUM, COUNT, AVG, MAX, MIN 등 주요 집계 함수는 데이터 요약에 사용됩니다.
- GROUP BY는 특정 컬럼 기준으로 행을 그룹화하여 집계를 수행합니다.
- 인덱스 컬럼을 GROUP BY에 사용하면 정렬 비용을 절감할 수 있습니다.

조건 분기(CASE/DECODE) 활용

- CASE WHEN을 이용해 조건별 집계를 단일 패스로 처리합니다.
- DECODE는 오라클 전용 함수로 간결한 조건 분기에 유용합니다.

```
SELECT department_id,
       SUM(CASE WHEN job_id = 'IT_PROG' THEN salary ELSE 0 END) AS it_prog_salary,
       SUM(CASE WHEN job_id = 'ST_CLERK' THEN salary ELSE 0 END) AS st_clerk_salary
FROM employees
GROUP BY department_id;
```

WHERE vs HAVING 전략

- 처리 시점과 성능 영향 ➔ WHERE 사용 시 94% 성능 향상 사례
  - WHERE 절: 집계 전 데이터 필터링 → 집계 작업량 감소

```
SELECT dept, AVG(salary)
FROM employees
WHERE hire_date >= '2020-01-01'  -- 10,000건 → 500건으로 축소
GROUP BY dept;
```

  - HAVING 절: 집계 후 결과 필터링 → 전체 데이터 집계 필요

```
SELECT dept, AVG(salary)
FROM employees
GROUP BY dept
HAVING AVG(salary) > 5000;  -- 모든 10,000건 집계 후 필터링
```

실무 튜닝 사례

- 문제: 대량 거래 데이터의 지역별 상품별 매출 집계 지연
- 최적화:

```
SELECT region, product,
       SUM(sales) FILTER (WHERE quarter = 'Q1') AS q1_sales,  -- ANSI 표준
       SUM(sales) FILTER (WHERE quarter = 'Q2') AS q2_sales
FROM transactions
WHERE year = 2024  -- 먼저 필터링
GROUP BY region, product;
```

- 성능 개선 효과:
  - 실행 시간 120초 → 8초로 감소
  - Temp 테이블스페이스 사용량 90% 감소

조건 분기 성능 최적화

- 인덱스 활용: 조건 분기 컬럼에 인덱스 추가
- 불필요 연산 회피:

```
-- 비효율적
SUM(CASE WHEN status = 'Y' THEN price * 1.1 ELSE price * 0.9 END)


-- 효율적
SUM(price * CASE WHEN status = 'Y' THEN 1.1 ELSE 0.9 END)
```

복합 집계에 UNION ALL 대신 ROLLUP/CUBE 사용

- 여러 개의 조합별 합계를 구할 때, 전통적으로는 UNION ALL로 쿼리를 여러 번 작성하거나,
  서브쿼리 또는 애플리케이션 단에서 집계를 반복했습니다.

```
-- 부서, 직무, 전체 별로 각각 집계
SELECT dept, job, SUM(salary) FROM employees GROUP BY dept, job
UNION ALL
SELECT dept, NULL, SUM(salary) FROM employees GROUP BY dept
UNION ALL
SELECT NULL, NULL, SUM(salary) FROM employees;
```

복합 집계에 UNION ALL 대신 ROLLUP/CUBE 사용

- ROLLUP은 계층적 소계를 생성합니다.
- CUBE는 모든 조합의 소계를 생성합니다. (2^n 경우의 수)

```
SELECT dept, job, SUM(salary)
FROM employees
GROUP BY CUTE(dept, job);
```

```
SELECT dept, job, SUM(salary)
FROM employees
GROUP BY ROLLUP(dept, job);
```

윈도우 함수와 집계 결합으로

- SUM OVER(): 한 번의 테이블 스캔으로 누적 집계

```
SELECT date, sales,
       SUM(sales) OVER (PARTITION BY region ORDER BY date) AS cum_sales
FROM daily_sales;
```

- ROW_NUMBER(): 한 번에 그룹별 순번을 매기고 필요한 행만 필터링하여 가독성과 성능 모두 향상

```
SELECT customer_id, SUM(price)
FROM (
  SELECT *,
         ROW_NUMBER() OVER (PARTITION BY customer_id, product ORDER BY date) AS rn
  FROM orders
)
WHERE rn = 1  -- 중복 주문 제거
GROUP BY customer_id;
```

NULL 처리와 조건 분기 최적화로 집계 함수의 정확성과 안정성 확보

- 집계 함수의 계산에서 NULL 값은 자동 제외되어 리포트 오류나 해석 불가 상황이 발생 가능합니다.
- 조건 분기에 CASE WHEN 대신 COALESCE함수를 사용합니다.

```
COALESCE(SUM(CASE WHEN status = 'active' THEN sales END), 0)
```

- NULLIF 함수 사용으로 특정 값을 제외 후 집계합니다.

```
AVG(NULLIF(score, -1))  -- -1을 NULL로 처리해 평균 계산
```

요약

- 조건 분기는 CASE/DECODE로 단일 패스 처리
- 집계 전 WHERE 필터링으로 데이터 축소
- 윈도우 함수로 중복 제거 및 누적 집계 최적화
- NULL은 COALESCE/NULLIF로 명시적 처리
- 실무에서는 집계 함수 필터링(ANSI) 으로 다중 조건 분기 효율화


Thank you