Ch 8. SQL & DB 설계

이 장의 구성

SQL 기본 명령어 / JOIN / 윈도우 함수 / CTE
DB 설계 (정규화, ACID, CAP) / NoSQL / MapReduce
면접 문제 35개 (Easy ~ Hard) + 전체 해설

Part 1

SQL 기초와 면접 전략

SQL 면접이 중요한 이유

- DS 업무 시간의 상당 부분을 **SQL로 데이터를 추출/분석**하는 데 사용
- 거의 모든 회사에서 SQL 문제를 출제 (Product/Analytics 팀 필수)
- 초기 면접 라운드에서 **자동 코딩 평가 도구**로 스크리닝
- DB별 문법 차이보다 **비즈니스 요구 → SQL 변환 능력**을 평가
- 일부 회사는 쿼리 **성능 최적화**까지 질문

SQL 문제 풀이 전략

1. **문제를 완전히 이해**한 뒤 시작 — 요구사항을 면접관에게 되풀이하여 확인
2. **역방향으로 작업**— 최종 결과 테이블이 단일 SELECT라고 상상하고 거꾸로 분해
3. JOIN, CTE, 서브쿼리가 필요하면**한 번에 하나씩**단계적으로 구축
4. 결과 스키마(컬럼명, 타입)를 먼저 정의하면 방향을 잃지 않음
5. 막히면 반드시 면접관에게 질문 — 코딩 면접과 동일

기본 SQL 명령어 요약

명령어	설명
`SELECT`	특정 컬럼 선택 (대부분의 쿼리 핵심)
`CREATE TABLE`	테이블 생성 + 스키마 정의
`INSERT / UPDATE / DELETE`	행 추가 / 수정 / 삭제
`GROUP BY`	특정 컬럼 기준 행 그룹화
`WHERE`	그룹화 **전** 필터링
`HAVING`	그룹화 **후** 필터링
`ORDER BY / DISTINCT / UNION`	정렬 / 중복제거 / 결과 합치기

Part 2

JOIN 패턴

4가지 JOIN 유형

INNER JOIN

매칭되는 행만 보존
기본 JOIN = INNER
가장 일반적

LEFT JOIN

왼쪽 테이블 전체 보존
오른쪽에 매칭 없으면 NULL
사용자 중 게시글 작성 비율 계산에 활용

RIGHT JOIN

오른쪽 테이블 전체 보존
LEFT의 반대
실무에서는 LEFT로 순서를 바꿔 사용하는 경우가 많음

OUTER JOIN

양쪽 모두 보존
매칭 없는 행도 NULL로 포함
비활성 사용자+비활성 게시글 동시 조회

JOIN 예시 — Reddit users & posts

```sql
-- INNER JOIN: 게시글 작성 사용자 수
SELECT COUNT(DISTINCT u.user_id)
FROM users u
JOIN posts p ON u.user_id = p.user_id;

-- LEFT JOIN: 게시글 작성 비율
SELECT
  COUNT(DISTINCT CASE WHEN p.post_id IS NOT NULL
        THEN u.user_id END)
  / COUNT(*) AS pct_users
FROM users u
LEFT JOIN posts p ON u.user_id = p.user_id;
```

- Self JOIN: 같은 테이블 내 행 쌍 분석 (자주 함께 구매되는 상품 등)
- JOIN 키에 **인덱스** 설정 여부가 성능에 큰 영향

Part 3

고급 SQL — CTE & 윈도우 함수

CTE vs 서브쿼리

CTE (Common Table Expression)

- `WITH alias AS (...)` 형태로 정의
- 큰 쿼리를 **관리 가능한 단위**로 분할
- **재귀 CTE** 가능 (계층 구조 탐색)
- 가독성이 높아 면접에서 선호

서브쿼리 (Subquery)

- 쿼리 **인라인**으로 삽입
- 반드시 **고유 별칭** 부여 필요
- CTE와 기능적으로 유사
- 단순한 경우 서브쿼리가 간결

CTE 실전 예시 — 사용자별 게시글 분포

```sql
WITH user_post_count AS (
  SELECT users.user_id,
         COUNT(post_id) AS num_posts
  FROM users
  LEFT JOIN posts ON users.user_id = posts.user_id
  GROUP BY users.user_id
)
SELECT num_posts,
       COUNT(*) AS num_users
FROM user_post_count
GROUP BY num_posts;
```

- 1단계: 사용자별 게시글 수 집계
- 2단계: 게시글 수 기준 히스토그램 생성

윈도우 함수 — 개념과 구문

- 집계 함수처럼 **행 집합에 대해 계산**하되, 행을 그룹으로 합치지 않음
- 개별 행의 정체성을 유지하면서 집계값을 함께 표시 가능

```
함수() OVER (
  PARTITION BY 분할기준     -- GROUP BY와 유사
  ORDER BY 정렬기준         -- 행 처리 순서
  ROWS BETWEEN start AND end  -- 슬라이딩 윈도우 크기
)
```

- GROUP BY + JOIN 대비 코드가 **훨씬 간결**

핵심 윈도우 함수 3가지

LAG / LEAD

이전/이후 행 참조
시계열 분석 필수
예: 게시글 간 시간 차이
YoY/MoM 비교에 활용

RANK / ROW_NUMBER

순위 매기기
RANK: 동점 시 같은 순위
ROW_NUMBER: 항상 고유 번호
"N번째 행 추출"의 핵심

누적 / 이동 집계

SUM OVER (ORDER BY) = 누적합
AVG OVER (ROWS BETWEEN) = 이동평균
윈도우 프레임 크기로 제어

LAG와 RANK 실전 예시

```sql
-- LAG: 같은 subreddit 내 이전 게시 시간과의 차이
SELECT p.*,
  LAG(post_time, 1) OVER (
    PARTITION BY user_id, subreddit_id
    ORDER BY post_time ASC
  ) AS prev_post_time
FROM posts p;
```

```sql
-- RANK: 사용자별 게시글 본문 길이 순위
SELECT *,
  RANK() OVER (PARTITION BY user_id
               ORDER BY LENGTH(body) DESC) AS rank
FROM users u LEFT JOIN posts p ON u.user_id = p.user_id;
```

함수	동점 처리	결과 예시
`RANK()`	같은 순위, 건너뜀	1, 2, 2, 4

Part 4

DB 설계 — 키, 정규화, ACID, CAP

Primary Key vs Foreign Key

Primary Key (기본키)

- 각 행을 **고유하게 식별**
- 중복값/NULL **불가**
- 테이블당 **1개**만 허용
- 자동 인덱싱됨
- 좋은 PK 조건: 안정성, 유일성, 비축소성

Foreign Key (외래키)

- 다른 테이블의 PK를 **참조**
- 중복값/NULL **허용**
- 테이블당 **여러 개** 가능
- 자동 인덱싱 안 됨
- 데이터 무결성 보장 (참조 무결성)

정규화 vs 비정규화

항목	정규화 (Normalization)	비정규화 (Denormalization)
목적	데이터 중복 제거, 무결성 향상	읽기 성능 최적화
방법	테이블 분리 + FK 연결	중복 데이터 허용, JOIN 제거
장점	저장 공간 절약, 일관성	빠른 읽기, 단순한 쿼리
단점	JOIN 비용 증가	쓰기 시 중복 갱신 필요
적합 환경	OLTP (트랜잭션 중심)	OLAP (분석 중심), 대규모 시스템

- 면접 포인트: **왜 FK를 설정하는가?, 1:1, 1:N, N:M 관계 설계**

인덱스 — 클러스터드 vs 논클러스터드

항목	클러스터드 인덱스	논클러스터드 인덱스
물리적 정렬	인덱스 순서 = 디스크 저장 순서	별도 저장, 포인터로 연결
테이블당 개수	**1개**만 허용	**여러 개** 가능
범위 검색	매우 빠름	간접 참조로 느릴 수 있음
INSERT/UPDATE	물리 재배치 필요 → 느림	상대적으로 빠름

- **OLTP** (UPDATE/INSERT 빈번): 인덱스 많으면 성능 저하
- **OLAP** (SELECT/JOIN 주로): 인덱스가 성능을 크게 향상

ACID — 트랜잭션의 4가지 보장

❶ Atomicity(원자성) — 트랜잭션은 전체 성공 또는 전체 실패, 부분 실행 없음

❷ Consistency(일관성) — 트랜잭션 전후로 무결성 제약 조건 유지

❸ Isolation(격리성) — 동시 트랜잭션이 서로 간섭 없이 독립 실행

❹ Durability(지속성) — 완료된 트랜잭션은 시스템 장애에도 보존

- OLTP 시스템에서 특히 중요 (실시간 다수 사용자 트랜잭션 처리)

CAP 정리 — 분산 DB의 트레이드오프

세 가지 중 두 가지만 동시에 만족 가능:

속성	설명	우선 사례
Consistency	모든 노드가 같은 데이터를 봄	WhatsApp 결제 (CP)
Availability	항상 응답 보장	Instagram 피드 (AP)
Partition Tolerance	노드 간 통신 단절에도 동작	분산 시스템 기본 전제

- AP: 좋아요 수가 약간 달라도 빠른 응답이 중요
- CP: 잔액 불일치 방지가 속도보다 중요

Part 5

확장, NoSQL, MapReduce

수직 확장 vs 수평 확장

수직 확장 (Scaling Up)

- 기존 서버에 CPU/RAM 추가
- 아키텍처 변경 불필요
- 비용이 급격히 증가
- 물리적 한계 존재

수평 확장 (Scaling Out)

- 서버(노드)를 추가
- 비용 효율적, 내결함성 우수
- 노드 간 데이터 일관성 관리 필요
- **샤딩**: 행을 노드별로 분산 저장 (해시/범위/일관된 해싱)

관계형 DB vs NoSQL

관계형 DB (MySQL, Postgres)

- 고정 스키마, 테이블 기반
- ACID 보장 → 데이터 무결성
- 표준 SQL → 학습 전환 용이
- 수직 확장에 적합

NoSQL (MongoDB, Neo4J, Cassandra)

- 유연한 스키마 (문서/그래프/키-값)
- BASE 모델 → 가용성 우선
- 수평 확장에 유리
- 스타트업의 빠른 반복 개발에 적합

BASE 모델과 MapReduce

BASE (NoSQL의 일관성 철학)

- Basically Available — 항상 응답하나 최신 아닐 수 있음
- Soft State — 입력 없이도 시스템 상태 변경 가능
- Eventual Consistency — 결국 일관된 상태로 수렴

MapReduce 4단계

단계	동작
Split	데이터를 청크로 분할, 노드에 분배
Map	각 노드에서 (key, value) 쌍 생성
Shuffle	같은 key를 같은 노드로 이동 (핵심!)
Reduce	key별 값 집계 → 최종 출력

Hadoop vs Spark

항목	Hadoop MapReduce	Spark
처리 방식	디스크 기반	메모리(RAM) 기반
속도	상대적으로 느림	빠름 (인메모리)
리소스 관리	YARN 필요	자체 내장
비용	저렴	RAM 비용 높음

- Hadoop: HDFS(저장) + MapReduce(처리) + YARN(스케줄링)
- Spark: 배치 처리 + 스트리밍 + ML 라이브러리 통합

Part 6

면접 문제 — 집계 & 필터링 패턴

Q8.1 CTR 계산 / Q8.3 디바이스별 조회수 — 조건부 집계

```sql
-- Q8.1 Facebook: 2019년 앱별 클릭률
SELECT app_id,
  SUM(IF(event_id = 'click', 1, 0))
  / SUM(IF(event_id = 'impression', 1, 0)) AS ctr
FROM events
WHERE timestamp >= '2019-01-01' AND timestamp < '2020-01-01'
GROUP BY app_id;
```

```sql
-- Q8.3 NYT: 노트북 vs 모바일 조회수
SELECT
  SUM(IF(device_type = 'laptop', 1, 0)) AS laptop_views,
  SUM(IF(device_type IN ('phone','tablet'), 1, 0)) AS mobile_views
FROM viewership;
```

- **패턴**: `SUM(IF(...))` — 행 데이터를 열로 피벗하는 가장 기본 형태

Q8.2 Top 3 도시 / Q8.5 HAVING 필터

```sql
-- Q8.2 Robinhood: 완료 주문이 가장 많은 상위 3개 도시
SELECT u.city, COUNT(DISTINCT t.order_id) AS num_orders
FROM trades t JOIN users u ON t.user_id = u.user_id
WHERE t.status = 'complete'
GROUP BY u.city ORDER BY num_orders DESC LIMIT 3;
```

```sql
-- Q8.5 eBay: $1000 이상 소비 고객 중 주문 상품 수 Top 10
SELECT user_id, COUNT(product_id) AS num_products
FROM user_transactions
GROUP BY user_id
HAVING SUM(spend) > 1000
ORDER BY num_products DESC LIMIT 10;
```

- **패턴**: JOIN + GROUP BY + LIMIT / **HAVING** = 그룹화 후 필터

Q8.6 히스토그램 — 서브쿼리 + 이중 GROUP BY

> Twitter: 2020년 사용자별 트윗 수의 분포(히스토그램)

```sql
SELECT num_tweets AS tweet_bucket, COUNT(*) AS num_users
FROM (
  SELECT user_id, COUNT(*) AS num_tweets
  FROM tweets
  WHERE tweet_date BETWEEN '2020-01-01' AND '2020-12-31'
  GROUP BY user_id
) total_tweets
GROUP BY num_tweets
ORDER BY num_tweets;
```

- **패턴**: 안쪽 쿼리에서 사용자별 집계 → 바깥 쿼리에서 분포 집계
- 2단계 GROUP BY = 히스토그램 공식

Part 7

면접 문제 — 윈도우 함수 패턴

Q8.4 누적 합계 / Q8.10 7일 이동 평균

```sql
-- Q8.4 Amazon: 제품별 날짜순 누적 지출
SELECT trans_date, product_id,
  SUM(spend) OVER (
    PARTITION BY product_id ORDER BY trans_date
  ) AS cum_spend
FROM total_trans;
```

```sql
-- Q8.10 Twitter: 사용자별 7일 롤링 평균
WITH tweet_counts AS (
  SELECT user_id, CAST(tweet_date AS date) AS tweet_date,
         COUNT(*) AS num_tweets
  FROM tweets GROUP BY user_id, CAST(tweet_date AS date)
)
SELECT user_id, tweet_date,
  AVG(num_tweets) OVER (
    PARTITION BY user_id ORDER BY tweet_date
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) AS rolling_avg_7d
FROM tweet_counts;
```

Q8.11 N번째 거래 / Q8.9 첫 거래 필터 — ROW_NUMBER

```sql
-- Q8.11 Uber: 모든 사용자의 3번째 거래 추출
WITH nums AS (
  SELECT *, ROW_NUMBER() OVER (
    PARTITION BY user_id ORDER BY transaction_date
  ) AS trans_num
  FROM transactions
)
SELECT user_id, spend, transaction_date
FROM nums WHERE trans_num = 3;
```

```sql
-- Q8.9 Etsy: 첫 거래 $50 이상인 고객
WITH purchase_num AS (
  SELECT user_id, spend, ROW_NUMBER() OVER (
    PARTITION BY user_id ORDER BY transaction_date
  ) AS rownum
  FROM user_transactions
)
```

Q8.7 동일 상품 재구매 / Q8.8 중복 채용공고 — RANK

```sql
-- Q8.7 Stitch Fix: 같은 상품을 다른 날 구매한 고객 수
SELECT COUNT(DISTINCT user_id) FROM (
  SELECT user_id, RANK() OVER (
    PARTITION BY user_id, product_id
    ORDER BY CAST(purchase_time AS DATE)
  ) AS purchase_no
  FROM purchases
) t WHERE purchase_no = 2;
```

```sql
-- Q8.8 LinkedIn: 중복 채용공고를 올린 회사 수
WITH ranked AS (
  SELECT company_id, ROW_NUMBER() OVER (
    PARTITION BY company_id, title, description
    ORDER BY post_date
  ) AS rn FROM job_listings
)
SELECT COUNT(DISTINCT company_id) FROM ranked WHERE rn > 1;
```

Q8.12 카테고리별 Top 3 / Q8.32 YoY 성장률

```sql
-- Q8.12 Amazon: 2020년 카테고리별 매출 상위 3개 상품
WITH spend AS (
  SELECT product_id, category_id, SUM(spend) AS total
  FROM product_spend
  WHERE transaction_date BETWEEN '2020-01-01' AND '2020-12-31'
  GROUP BY product_id, category_id
),
ranked AS (
  SELECT *, RANK() OVER (
    PARTITION BY category_id ORDER BY total DESC) AS rnk
  FROM spend
)
SELECT * FROM ranked WHERE rnk <= 3;
```

- **"그룹별 Top N" 공식**: 집계 CTE → RANK CTE → 외부 필터

```sql
-- Q8.32 Wayfair: LAG(col, 52) = 52주 전 값으로 YoY 성장률 계산
LAG(total_spend, 52) OVER (PARTITION BY product_id ORDER BY week)
```

Part 8

면접 문제 — JOIN 응용 & 관계 분석

Q8.22 인기 토픽 미팔로우 사용자 — MINUS

> Twitter: Top 100 토픽을 팔로우하지 않는 기존 사용자

```sql
WITH top_topics AS (
  SELECT * FROM topic_rankings
  WHERE ranking_date = '2021-01-01' AND rank <= 100
)
SELECT DISTINCT user_id
FROM user_topics WHERE follow_date <= '2021-01-01'
MINUS
SELECT u.user_id
FROM user_topics u
JOIN top_topics t ON u.topic_id = t.topic_id;
```

- **패턴**: 전체 집합 - 조건 집합 = `MINUS` (또는 `EXCEPT`)
- 대안: `WHERE NOT EXISTS` 서브쿼리

Q8.25 연령대별 활동 비율 — 조건부 집계 + JOIN

> Snapchat: 연령대별 send/open 시간 비율

```sql
WITH time_stats AS (
  SELECT ab.age_bucket,
    SUM(IF(type='send', time_spent, 0)) AS send_time,
    SUM(IF(type='open', time_spent, 0)) AS open_time,
    SUM(time_spent) AS total_time
  FROM age_breakdown ab
  JOIN activities a ON ab.user_id = a.user_id
  WHERE a.type IN ('send','open')
  GROUP BY ab.age_bucket
)
SELECT age_bucket,
  send_time / total_time AS pct_send,
  open_time / total_time AS pct_open
FROM time_stats;
```

- **패턴**: 조건부 SUM + 전체 SUM → 비율 계산

Q8.26 동시 세션 / Q8.30 함께 구매된 상품 — Self JOIN

```sql
-- Q8.26 Pinterest: 가장 많은 세션과 겹치는 세션
SELECT s1.session_id, COUNT(s2.session_id) AS concurrents
FROM sessions s1
JOIN sessions s2
  ON s1.session_id != s2.session_id
  AND s2.start_time BETWEEN s1.start_time AND s1.end_time
GROUP BY s1.session_id
ORDER BY concurrents DESC LIMIT 1;
```

```sql
-- Q8.30 Walmart: 같은 트랜잭션에서 함께 구매된 상품 쌍 Top 10
-- Self JOIN + p1.product_id < p2.product_id (중복 쌍 방지)
SELECT p1.product_name, p2.product_name, COUNT(*) AS cnt
FROM purchase_info p1
JOIN purchase_info p2
  ON p1.transaction_id = p2.transaction_id
  AND p1.product_id < p2.product_id
GROUP BY 1, 2 ORDER BY cnt DESC LIMIT 10;
```

Part 9

면접 문제 — 리텐션 & 시계열

Q8.23 월간 리텐션 / Q8.31 재활성화 — EXISTS

```sql
-- Q8.23 Facebook: 이번 달 + 지난 달 모두 활동한 MAU
SELECT DATE_TRUNC('month', curr.timestamp) AS month,
       COUNT(DISTINCT curr.user_id) AS mau
FROM user_actions curr
WHERE EXISTS (
  SELECT 1 FROM user_actions prev
  WHERE prev.user_id = curr.user_id
    AND DATE_TRUNC('month', prev.timestamp)
      = DATE_TRUNC('month', curr.timestamp) - INTERVAL '1 month'
)
GROUP BY month ORDER BY month;
```

```sql
-- Q8.31 Facebook: 전월 미접속 → 당월 접속 (재활성화)
-- 동일 구조에서 WHERE NOT EXISTS로 변경
```

- EXISTS = 리텐션 / NOT EXISTS = 재활성화 (반대 조건)

Q8.29 전환율 / Q8.33 롤링 수익

```sql
-- Q8.29 Etsy: 최근 1주 가입자 중 구매 비율
SELECT COUNT(DISTINCT p.user_id)
       / COUNT(DISTINCT s.user_id) * 100 AS pct
FROM signups s
LEFT JOIN user_purchases p ON p.user_id = s.user_id
WHERE s.signup_date > NOW() - INTERVAL '7 DAY';
```

```sql
-- Q8.33 Stripe: 7일 롤링 수익 (Self JOIN 방식)
SELECT t2.txn_date, SUM(t1.total_amount) AS weekly_total
FROM daily_transactions t1
INNER JOIN daily_transactions t2
  ON t1.txn_date > t2.txn_date - INTERVAL '7 DAY'
  AND t1.txn_date <= t2.txn_date
GROUP BY t2.txn_date ORDER BY t2.txn_date;
```

- LEFT JOIN 비율 = 전환율 공식 / Self JOIN + **날짜 범위** = 롤링 집계

Part 10

면접 문제 — 개념 & 설계

개념 문제 핵심 요약 (Q8.14-21)

Q8.14 DB 뷰란?

쿼리 결과의 가상 테이블. 물리 스키마 없음. 장점: 복잡성 추상화, 보안(부분 노출), 메모리 절약

Q8.15 인덱스와 OLTP/OLAP

UPDATE/INSERT 많으면 인덱스 = 부담 (OLTP). SELECT/JOIN 많으면 인덱스 = 성능 향상 (OLAP)

Q8.16 좋은 PK의 조건

안정성(변하지 않음), 유일성(중복 불가), 비축소성(컬럼 제거 시 유일성 위반)

Q8.19 WHERE vs HAVING

WHERE = 그룹화 전 행 단위 필터. HAVING = 그룹화 후 그룹 단위 필터. 집계 함수는 HAVING에서만

Q8.17 관계형 vs NoSQL

관계형: ACID, 스키마 고정, 수직 확장. NoSQL: BASE, 유연한 스키마, 수평 확장

Q8.21 클러스터드 vs 논클러스터드

클러스터드: 물리 정렬 = 인덱스 순서, 1개. 논클러스터드: 별도 저장 + 포인터, 여러 개

Q8.18 MapReduce 셔플 / Q8.34 상호 친구 수

Q8.18 (Capital One): MapReduce로 랜덤 셔플

- Map: 각 행에 랜덤 키 (0~k) 할당 → Shuffle → Reduce: 그대로 출력

Q8.34 (Facebook): 모든 사용자 쌍의 상호 친구 수

- X의 친구: [W,Y,Z], Y의 친구: [X,Z]
- **Map**: X → `((X,Y), [W,Y,Z])` / Y → `((X,Y), [X,Z])` *(키 정렬)*
- **Shuffle**: 같은 키 (X,Y) → 같은 노드
- **Reduce**: `[W,Y,Z] ∩ [X,Z] = [Z]` → 상호 친구 1명

Q8.35 검색 빈도 추적 시스템 설계

> Google: 검색 쿼리 문자열과 빈도를 대규모로 추적하는 시스템

단계	설계
기본 구조	key-value store: (쿼리 문자열, 빈도)
확장 전략	**수평 확장** — 노드 추가 (Google 규모에서 수직은 한계)
샤딩 옵션 1	알파벳 범위 — 단순하나 데이터 분포 불균형
샤딩 옵션 2	해시 함수 — 균등 분산, 노드 추가 시 재해싱 문제
최적 해법	Consistent Hashing — 노드 변경 시 최소 재배치

Part 11

SQL 면접 패턴 총정리

SQL 면접 4대 패턴

1. 집계 & 필터

GROUP BY + HAVING
SUM(IF(...)) 조건부 집계
COUNT(DISTINCT ...)
기본이지만 가장 빈출

2. 윈도우 함수

ROW_NUMBER / RANK
LAG / LEAD
SUM/AVG OVER
"그룹별 Top N"의 공식

3. JOIN 응용

Self JOIN (쌍 분석)
LEFT JOIN (전환율)
MINUS/EXCEPT
EXISTS / NOT EXISTS

4. 시계열 & CTE

DATE_TRUNC (기간 버킷)
롤링 평균/합계
리텐션/재활성화
다중 CTE 체이닝

면접 문제 유형별 매핑

패턴	대표 문제	핵심 키워드
조건부 집계	8.1 CTR, 8.3 디바이스, 8.25 연령대	`SUM(IF(...))`, 피벗
HAVING 필터	8.5 상위 고객, 8.6 히스토그램	그룹 후 필터링
누적/롤링	8.4 누적합, 8.10 이동평균, 8.33 롤링	`OVER(ORDER BY)`, `ROWS BETWEEN`
Top N	8.2 도시, 8.12 카테고리	`RANK()` + 외부 필터
N번째 행	8.9 첫 거래, 8.11 3번째	`ROW_NUMBER()` = N
리텐션	8.23 MAU, 8.31 재활성화	`EXISTS` / `NOT EXISTS`
쌍 분석	8.26 동시세션, 8.30 장바구니	Self JOIN + 부등식
YoY 비교	8.32 주간 성장률	`LAG(col, 52)`

핵심 포인트

- SQL은 DS 면접에서 **가장 확실하게 출제**되는 영역 — 준비 ROI가 높음
- **윈도우 함수**를 능숙하게 쓰면 대부분의 중급 문제 해결 가능
- 문제 풀이 순서: 결과 스키마 정의 → 역방향 분해 → CTE로 단계별 구축
- 코딩이 아니라 **비즈니스 요구를 SQL로 번역하는 능력**이 핵심
- DB 설계 개념(ACID, CAP, 정규화)은 스타트업/엔지니어링 포지션에서 추가 출제
- 실전 연습: DataLemur.com (각 문제별 온라인 실습 가능)