SQL 코딩 가이드

개정이력

개정번호	개정일자	구분	개정내역	작성자	비고
0.1	2025.08.12	신규	최초작성	이상훈	

목차

1. 일반 규칙

항목	규칙
대소문자	SQL 키워드는 대문자, 사용자 정의 객체는 소문자 또는 카멜표기법
들여쓰기	스페이스 4칸 권장 (탭 X)
문장 종료	모든 SQL 문은 `;` 세미콜론으로 종료 (권장)
주석 작성	한 줄: `--`, 여러 줄: `/* */`

1.1 대소문자 규칙 (Case Style)

항목	규칙	예시
SQL 키워드	대문자 사용	SELECT, FROM, WHERE
사용자 정의 객체 (테이블, 컬럼, 변수 등)	소문자 또는 카멜 표기법	user_id, orderDate
저장 프로시저/함수	소문자 + 접두어 (usp_, fn_)	usp_get_user_orders

🔷 이유: 일관성 유지 및 가독성 향상

1.2 들여쓰기 및 줄바꿈 (Indentation & Line Breaks)

- 들여쓰기: 스페이스 4칸 (Tab 지양)
- SELECT 문: 각 컬럼을 별도 줄에 작성
- 절 단위 줄바꿈: SELECT, FROM, JOIN, WHERE, ORDER BY 등은 줄바꿈
- JOIN 조건은 항상 ON 절과 함께 같은 블록에

예시:

```
SELECT u.user_id
      ,u.user_name
      ,o.order_id
  FROM users u
       INNER JOIN orders o
          ON u.user_id = o.user_id
 WHERE o.status = 'COMPLETE'
;
```

1.3 세미콜론(;) 사용

- 모든 SQL 문장 끝에 세미콜론을 붙일 것 (권장)
  SQL Server는 SQL 표준을 따르기 위해 점차 세미콜론 사용을 엄격히 요구

```
SELECT *
  FROM users
;

DELETE FROM logs
 WHERE log_date < GETDATE() - 30
;
```

🔶 THROW, WITH (CTE), MERGE, BEGIN...END 등에서 오류 예방

1.4 주석(Comment) 규칙

- 한 줄 주석: --
- 블록 주석: /* ... */
- 업무 목적 또는 쿼리 의도를 설명

예시:
-- 최근 30일 이내 완료된 주문 조회

```
SELECT order_id
      ,order_date
  FROM orders
 WHERE status = 'COMPLETE'
   AND order_date >= DATEADD(DAY, -30, GETDATE())
;
```

❗ "무엇을 하는가"보다 "왜 하는가"를 설명하는 주석이 더 중요

1.5 별칭(Alias) 사용 규칙

- 테이블에는 명확하고 짧은 별칭 사용 (u, o 등)
- 컬럼명과 충돌하지 않게, 의미 있는 접두어/약자 사용
- 컬럼에도 가능한 경우 AS를 사용해 명시

```
SELECT u.user_name AS name
      ,o.order_date AS orderDate
  FROM users u
       JOIN orders o
```

```
          ON u.user_id = o.user_id
;
```

1.6 SELECT * 지양

- 실제 필요한 컬럼만 명시
- 이유: 성능 저하, 스키마 변경 시 오류 가능성 증가

```
-- ❌ 나쁜 예:
SELECT *
  FROM users
;

-- ✅ 좋은 예:
SELECT user_id
      ,user_name
      ,created_at
  FROM users
;
```

1.7 SQL 문 순서(권장 스타일)

```
SELECT     -- 조회할 컬럼
FROM       -- 데이터 소스
JOIN       -- 필요한 테이블 연결
WHERE      -- 조건 필터링
GROUP BY   -- 그룹화
HAVING     -- 그룹화된 결과 필터링
ORDER BY   -- 정렬
```

1.8 코드 정렬 및 포맷 예시 (표준화)

```
SELECT u.user_id
      ,u.user_name
      ,o.order_id
      ,o.status
  FROM users u
       LEFT JOIN orders o
         ON u.user_id = o.user_id
 WHERE u.is_active = 1
   AND o.order_date >= '2025-01-01'
```

```
ORDER BY o.order_date DESC
;
```

1.9 기타 규칙

항목	규칙
논리 연산자	AND, OR는 줄 바꿈하여 명확히
비교 연산자	=, <>, IN, BETWEEN, LIKE 등은 명확히 띄어쓰기
CTE (공통 테이블 표현식)	반드시 세미콜론 앞에 사용
정수 상수	따옴표 없이 사용 (1, 0)
날짜 상수	ISO8601 형식 사용: '2025-08-11'
NULL 비교	IS NULL, IS NOT NULL 사용 (= NULL ❌)

📌 정리 요약표

구분	지켜야 할 규칙
대소문자	키워드는 대문자, 객체명은 소문자 또는 카멜
들여쓰기	스페이스 4칸
줄바꿈	SELECT, FROM, WHERE 등 절마다 줄바꿈
세미콜론	문장 끝마다 ; 사용
주석	의도 설명 중심
SELECT *	지양하고 명시적 컬럼 작성
테이블 별칭	명확한 약자 사용 (u, o 등)
조건절	명확하게 정렬, AND/OR 줄바꿈
포맷	항상 동일한 구조로 유지

2. 명명 규칙

2.1 일반 원칙

원칙	설명
일관성 유지	전사적으로 동일한 규칙 적용
의미 전달	역할과 목적이 명확하게 드러나야 함
축약어 최소화	가급적 전체 단어 사용 (단, 자주 쓰는 축약어는 예외)
영어 사용	객체 이름은 모두 영어로 작성 (로마자 사용)
구분자	단어는 snake_case로 구분 (예: user_name)

2.2 객체별 명명 규칙 표

객체 유형	접두어/패턴	예시	설명
테이블	복수형, snake_case	users, order_items	데이터 단위는 복수형
컬럼	소문자 + snake_case	user_id, created_at	타입보다 의미 중심
기본키	테이블명 + `_id`	user_id	외래키와 매칭 쉽게
외래키	참조하는 PK 이름 동일	user_id, product_id	참조 일관성 유지
인덱스	IX_테이블명_컬럼명	IX_users_email	읽기 최적화 기준 명확히
PK 제약조건	PK_테이블명	PK_users	고유 식별
FK 제약조건	FK_자식_부모	FK_orders_users	테이블 간 관계 명확히
UK 제약조건	UQ_테이블_컬럼	UQ_users_email	유니크 제약
CHECK 제약조건	CK_테이블_컬럼	CK_users_age	값 검증 규칙
DEFAULT 제약조건	DF_테이블_컬럼	DF_users_is_active	디폴트값 명시
VIEW	vw_설명	vw_active_users	`vw_` 접두어
프로시저	sp_기능명	sp_get_user_orders	`sp_` (Stored Procedure)
함수	fn_기능명	fn_get_tax_rate	`fn_` 접두어
트리거	trg_테이블_이벤트	trg_users_insert	trg_ 접두어
스키마	기능별 구분 가능	security.users, audit.logs	논리적 분리 가능

2.3 컬럼 명명 규칙 상세

유형	명명 규칙	예시
기본키	table_id 형태	user_id, product_id
외래키	참조 테이블 PK 동일하게	user_id, order_id
날짜/시간	_at, _date, _time 접미사	created_at, updated_at, login_time
상태/유무	is_, has_, status	is_active, has_email, order_status
코드 값	_code	country_code, status_code
금액/숫자	_amount, _count, _qty	total_amount, item_count, stock_qty
플래그	is_, flag_	is_deleted, flag_validated

2.4 예시: 하나의 테이블에 대한 명명 예

테이블명: orders
컬럼명:

컬럼명	설명
order_id	기본키
user_id	외래키 (users 참조)
order_date	주문 일자
total_amount	주문 총액
order_status	주문 상태
created_at	생성 시간
updated_at	수정 시간
is_paid	결제 여부

2.5 지양해야 할 명명 사례

잘못된 예	개선 예	설명
tblUsers	users	접두어 `tbl` 불필요
UID, CD, NM	user_id, code, name	축약어 지양
regdate, moddt	created_at, updated_at	의미 명확히

user1, user2	created_by, approved_by	역할 중심으로 명명

2.6 명명 규칙 자동 검사 팁

- 저장소 커밋 전 정적 분석 스크립트로 명명 규칙 점검
- SSMS(Management Studio) + SQL Prompt 같은 도구 활용
- DB 문서화 자동화 도구에 규칙 통합 (예: Redgate, ApexSQL)

2.7 명명 규칙 표준 템플릿 (요약)

객체	접두어/형식	예시
Table	복수형	users
Column	의미 기반 + snake_case	user_name
PK	PK_테이블명	PK_users
FK	FK_자식_부모	FK_orders_users
View	vw_	vw_monthly_sales
Procedure	usp_	usp_create_order
Function	fn_	fn_get_discount
Trigger	trg_	trg_users_insert

3. SELECT 쿼리 스타일

```
SELECT u.user_id
      ,u.user_name
      ,o.order_id
      ,o.order_date
  FROM users u
       INNER JOIN orders o
          ON u.user_id = o.user_id
 WHERE o.status = 'COMPLETE'
ORDER BY o.order_date DESC
;
```

규칙:

- 키워드는 대문자, 컬럼명/테이블명은 소문자
- 컬럼은 한 줄에 하나씩
- JOIN/WHERE/ORDER BY 등은 줄바꿈
- JOIN 시 별칭 필수

3.1 SELECT 문 기본 구조 (권장 스타일)

```
SELECT 컬럼1
      ,컬럼2
      ,컬럼3
  FROM 테이블명 [별칭]
       [JOIN 절]
 WHERE 조건
GROUP BY 컬럼
HAVING 그룹 조건
ORDER BY 정렬 컬럼
;
```

모든 절(SELECT, FROM, WHERE 등)은 새 줄에 배치
들여쓰기(4칸)를 유지하여 계층 구조 표현

3.2 절별 스타일 규칙

🎯 SELECT 절

규칙	설명

컬럼은 한 줄에 하나	가독성 향상
가능한 별칭(alias) 명시	UI 대응 또는 의미 명확화
함수 사용 시 띄어쓰기 명확히	COUNT(*), ISNULL(col, 0) 등
SELECT * 지양	명시적 컬럼 나열 권장

예시

```
SELECT u.user_id
      ,u.user_name AS name
      ,o.order_id
      ,o.order_date
      ,ISNULL(o.total_amount, 0) AS total_amount
```

🧱 FROM 절

- 테이블 별칭(Alias) 필수, 짧고 명확하게 (예: u, o)
- 별칭은 소문자 약자 권장

```
  FROM users u
```

🔗 JOIN 절

규칙	설명
JOIN 타입은 키워드 대문자로	INNER JOIN, LEFT JOIN
ON 조건은 같은 줄 또는 한 줄 아래	가독성 우선
여러 JOIN은 위에서 아래로 정렬	JOIN 순서 일관되게 유지

예시

```
  FROM users u
       INNER JOIN orders o
          ON u.user_id = o.user_id
       LEFT JOIN payments p
         ON o.order_id = p.order_id
```

🔍 WHERE 절

규칙	설명
조건문은 한 줄에 하나	AND, OR로 구분
논리 연산자(AND, OR)는 줄 맨 앞	조건 식별 용이

예시

```
 WHERE u.is_active = 1
   AND o.status = 'COMPLETE'
```

```
   AND o.order_date >= '2025-01-01'
```

🧱 GROUP BY / HAVING 절

항목	설명
그룹 컬럼은 SELECT에 포함되어야 함	GROUP BY 기준 명확히
HAVING은 집계 조건에만 사용	WHERE 절과 구분

```
GROUP BY u.user_id
     ,u.user_name
HAVING COUNT(o.order_id) > 5
```

📦 ORDER BY 절

항목	설명
명확한 정렬 기준 제시	인덱스 고려 가능
숫자 순서 대신 컬럼명 사용 권장	ORDER BY 1 지양

```
ORDER BY o.order_date DESC
;
```

3.3 전체 예시: 복합 쿼리 스타일 예

```
SELECT u.user_id
      ,u.user_name
      ,COUNT(o.order_id) AS order_count
      ,ISNULL(SUM(o.total_amount), 0) AS total_spent
  FROM users u
       LEFT JOIN orders o
         ON u.user_id = o.user_id
 WHERE u.is_active = 1
   AND o.order_date >= '2025-01-01'
GROUP BY u.user_id
        ,u.user_name
HAVING COUNT(o.order_id) >= 5
ORDER BY total_spent DESC
;
```

3.4 지양해야 할 스타일

잘못된 예	문제점
SELECT * FROM users;	불필요한 데이터 조회, 스키마 변경에 취약
JOIN 절에 별칭 없음	유지보수 시 가독성 저하
절이 한 줄에 모두 작성됨	협업 및 리뷰 시 불편
조건이 OR/AND 없이 나열됨	로직 오류 가능성

3.5 쿼리 스타일 자동화 도구 (권장 사용)

도구	설명
SQL Prompt (Redgate)	자동 포맷 + 정적 분석 가능
SSMS 플러그인	SQL Formatter 등
VS Code + SQL Formatter 확장	.sql 파일 작업 시 유용
Poor Man’s T-SQL Formatter	무료 웹 기반 SQL 포매터

3.6 스타일 리뷰 체크리스트

- SELECT * 대신 명시적 컬럼 사용
- 모든 절(SELECT/FROM/WHERE 등)은 줄바꿈
- JOIN 시 별칭 필수 사용
- 논리 조건은 줄바꿈하여 구분
- 정렬 기준 명확 (ORDER BY)
- 컬럼 및 별칭 명확하게 작성

4. 프로시저 예시

```
CREATE PROCEDURE sp_get_user_orders
    @user_id INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT order_id
          ,order_date
          ,status
      FROM orders
     WHERE user_id = @user_id;
END;
```

규칙:

- sp_ 접두어 사용
- 매개변수는 명확한 타입 지정
- SET NOCOUNT ON; 추가하여 성능 최적화
- BEGIN ~ END 블록 사용

4.1 프로시저 작성 규칙 요약

항목	권장 규칙
이름 규칙	sp_도메인_동작 형태 (예: sp_user_get_by_id)
설명 주석	프로시저 목적/파라미터/작성자 등 명시
트랜잭션 처리	필요한 경우 명시적으로 사용 (BEGIN TRAN, ROLLBACK, COMMIT)
에러 처리	TRY...CATCH 블록 사용
출력 방식	RETURN, OUTPUT, SELECT 중 상황에 맞게 선택
접근 최소화	꼭 필요한 객체만 접근 (보안, 성능 관점)

4.2 프로시저 기본 템플릿

```
CREATE OR ALTER PROCEDURE sp_user_get_by_id
    @user_id INT
AS
```

```
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        SELECT user_id
              ,user_name
              ,email
              ,created_at
          FROM users
         WHERE user_id = @user_id
        ;
    END TRY

    BEGIN CATCH
        -- 에러 처리
        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR(@ErrMsg, 16, 1);
    END CATCH
END;
```

4.3 상세 예시 - 사용자 주문 목록 조회 프로시저

📌 시나리오
사용자 ID를 입력받아 최근 주문 목록을 페이징으로 조회하는 프로시저

💡 요구사항

- 입력: @user_id, @page, @page_size
- 출력: 주문번호, 주문일, 금액, 상태
- 정렬: 주문일 DESC
- 에러 처리 포함
- 총 개수도 같이 리턴 (페이징 UI용)

✅ 전체 프로시저 예제

```
CREATE OR ALTER PROCEDURE usp_order_get_list_by_user
    @user_id     INT,
    @page        INT = 1,
    @page_size   INT = 10
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
```

```
        -- 변수 선언
        DECLARE @offset INT = (@page - 1) * @page_size;

        -- 결과 조회
        SELECT order_id
              ,order_date
              ,total_amount
              ,order_status
          FROM orders
         WHERE user_id = @user_id
        ORDER BY order_date DESC
        OFFSET @offset ROWS
        FETCH NEXT @page_size ROWS ONLY;

        -- 총 개수 반환
        SELECT COUNT(*) AS total_count
          FROM orders
         WHERE user_id = @user_id
    ;

    END TRY

    BEGIN CATCH
        -- 에러 발생 시 처리
        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @ErrSeverity INT = ERROR_SEVERITY();
        DECLARE @ErrState INT = ERROR_STATE();

        RAISERROR(@ErrMsg, @ErrSeverity, @ErrState);
    END CATCH

END;
```

4.4 예시 - 데이터 저장용 (INSERT) 프로시저

```
CREATE OR ALTER PROCEDURE usp_user_create
    @user_name NVARCHAR(100),
    @email     NVARCHAR(100),
    @created_by INT,
    @new_user_id INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
```

```
    BEGIN TRY
        INSERT INTO users (user_name, email, created_by,
created_at)
        VALUES (@user_name, @email, @created_by, GETDATE());

        SET @new_user_id = SCOPE_IDENTITY(); -- 방금 INSERT된
ID 반환
    END TRY
    BEGIN CATCH
        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR(@ErrMsg, 16, 1);
    END CATCH
END;
```

📌 OUTPUT 파라미터를 사용하여 INSERT 후 ID 반환

4.5 주석 템플릿 (프로시저 상단)

```
/*************************************************************
 * 프로시저명 : usp_order_get_list_by_user
 * 목적       : 특정 사용자의 주문 목록을 페이징으로 조회
 * 파라미터   :
 *   @user_id   - 사용자 ID
 *   @page      - 페이지 번호 (기본값: 1)
 *   @page_size - 페이지 크기 (기본값: 10)
 * 작성자     : 홍길동
 * 작성일     : 2025-08-11
 * 수정이력   :
 *   2025-08-12 - 조건 수정 (order_status 추가)
 *************************************************************/
```

4.6 Best Practice 요약

항목	권장 예시
프로시저 이름	usp_도메인_동작 (예: usp_product_update_price)
파라미터 기본값	가능하면 지정 (예: @page INT = 1)
에러 처리	TRY...CATCH로 감싸고 RAISERROR 사용
트랜잭션 처리	BEGIN TRAN, ROLLBACK, COMMIT 명시
출력 방식	SELECT, OUTPUT, RETURN 구분 사용

코드 포맷	절마다 들여쓰기, 대소문자 구분, 정렬

📎 참고: 트랜잭션 포함 예시 (UPDATE)

```
CREATE OR ALTER PROCEDURE usp_product_update_price
    @product_id INT,
    @new_price  DECIMAL(18,2)
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRAN;

        UPDATE products
           SET price = @new_price
              ,updated_at = GETDATE()
         WHERE product_id = @product_id;

        COMMIT;

    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK;

        RAISERROR(ERROR_MESSAGE(), 16, 1);
    END CATCH
END;
```

5. 조건문과 변수 사용

5.1 변수 사용 규칙

📌 선언 및 초기화

규칙	설명
@로 시작, 의미 있는 이름 사용	예: @user_id, @error_msg
DECLARE 후 SET 또는 SELECT로 값 할당	초기화는 명확하게 수행
한 줄에 여러 개 선언 가능하나, 가독성을 위해 줄바꿈 권장	

예시

```
DECLARE @user_id INT;
DECLARE @order_date DATETIME = GETDATE(); -- 선언과 동시에 초기화

-- 또는
DECLARE
    @name NVARCHAR(100),
    @email NVARCHAR(100);
```

🟡 SET vs SELECT 비교

항목	SET	SELECT
표준 SQL	✅ O	❌ X
단일 변수	✅ 명확	✅ 가능
다중 변수	❌ 반복 필요	✅ 한 번에 가능
NULL 처리	명확하게 1개만 설정	다수 행이면 마지막 값 설정됨 (주의!)

```
-- 단일 변수
SET @user_id = 123;

-- 다중 변수 초기화
SELECT
    @name = user_name,
    @email = email
FROM users
WHERE user_id = @user_id;
```

5.2 조건문 (IF / ELSE)

원칙	설명
블록 구조 사용 (BEGIN...END)	단일 라인도 블록으로 명확하게 작성
복잡한 조건은 변수로 분리	조건 로직 분리 → 가독성 향상
중첩 조건 최소화	논리 정리 또는 CASE로 대체 고려

기본 문법

```
IF @user_id IS NULL
BEGIN
    RAISERROR('User ID is required.', 16, 1);
    RETURN;
END
ELSE
BEGIN
    PRINT 'User ID is valid.';
END
```

5.3 CASE 문 사용 규칙

CASE 유형	설명
단일 값 비교	CASE 컬럼 WHEN 값 THEN ...
복합 조건 처리	CASE WHEN 조건 THEN ...
반드시 ELSE 처리	누락 시 NULL 반환될 수 있음

예시 1: SELECT 내 사용

```
SELECT order_id
      ,total_amount
      ,CASE order_status WHEN 'P' THEN '진행중'
                         WHEN 'C' THEN '완료'
                         WHEN 'F' THEN '실패'
                         ELSE '기타'
        END AS status_name
  FROM orders
;
```

예시 2: 조건에 따른 값 설정

```
DECLARE @discount_rate DECIMAL(5,2);

SELECT @discount_rate =
                  CASE WHEN @user_grade = 'VIP' THEN 0.2
                       WHEN @user_grade = 'REGULAR' THEN 0.1
                       ELSE 0
                   END;
```

5.4 예외 처리와 조건문 통합

```
IF @input_value IS NULL OR @input_value = ''
BEGIN
    RAISERROR('입력 값이 누락되었습니다.', 16, 1);
    RETURN;
END
```

5.5 실무 예제 - 사용자 활성 상태 변경

```
CREATE OR ALTER PROCEDURE usp_user_set_active
    @user_id INT,
    @is_active BIT
AS
BEGIN
    SET NOCOUNT ON;

    IF @user_id IS NULL
    BEGIN
        RAISERROR('User ID는 필수입니다.', 16, 1);
        RETURN;
    END

    BEGIN TRY
        UPDATE users
        SET
            is_active = @is_active,
            updated_at = GETDATE()
        WHERE user_id = @user_id;
```

```
        IF @@ROWCOUNT = 0
        BEGIN
            RAISERROR('해당 사용자가 존재하지 않습니다.', 16, 1);
        END
    END TRY
    BEGIN CATCH
        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR(@ErrMsg, 16, 1);
    END CATCH
END;
```

5.6 조건문 사용 시 Best Practice 요약

항목	권장 방안
변수명은 의미 있게	예: @total_count, @is_valid
IF 블록은 항상 BEGIN...END	단일 라인도 명시적으로 작성
CASE는 가능한 ELSE 포함	예외 처리 및 안정성 강화
SET과 SELECT의 차이를 이해하고 선택	SELECT는 다중 변수 시 유용, SET은 단일 변수 명확
논리 복잡한 조건은 변수로 나누기	가독성과 유지보수에 유리

📎 부록: 변수 + 조건 예시 패턴 모음

1. 값 존재 여부 확인

```
IF EXISTS (
    SELECT 1
      FROM users
     WHERE email = @email
)
BEGIN
    RAISERROR('이미 존재하는 이메일입니다.', 16, 1);
    RETURN;
END
```

2. 동적 메시지 처리

```
DECLARE @msg NVARCHAR(100);

IF @count = 0
    SET @msg = '처리할 데이터가 없습니다.';
```

```
ELSE
    SET @msg = CONCAT(@count, '건이 처리되었습니다.');

PRINT @msg;
```

3. NULL 방어 처리

```
IF @param IS NULL OR LTRIM(RTRIM(@param)) = ''
BEGIN
    RAISERROR('입력값이 비어 있습니다.', 16, 1);
    RETURN;
END
```

6. 성능 고려 사항

항목	권장사항
SELECT *	사용 지양, 필요한 컬럼만 명시
인덱스	WHERE 조건 및 JOIN 대상 컬럼에 적절히 사용
커서	가능한 사용 지양, 집합 기반 처리로 대체
뷰	필요한 경우에만 사용, 중첩된 뷰는 피하기
동적 SQL	SQL 인젝션 방지 필수 (예: `sp_executesql` 사용)

6.1 인덱스(Index) 활용

고려사항	설명
✅ 필요한 컬럼에 인덱스 생성	`WHERE`, `JOIN`, `ORDER BY`, `GROUP BY` 등에 자주 쓰
❗ 과도한 인덱스 생성 금지	DML 성능 저하 및 공간 낭비 발생
✅ 인덱스 커버리지 고려	쿼리에 필요한 모든 컬럼을 인덱스에 포함시켜 **Index**
✅ 클러스터드 vs 넌클러스터드 인덱스 구분 사용	PK는 클러스터드, 조회용은 넌클러스터드 사용 권장
✅ INCLUDE 컬럼 사용 고려	필터 조건은 아니지만 조회 대상인 컬럼은 `INCLUDE`

-- 예: 커버링 인덱스

```
CREATE NONCLUSTERED INDEX IX_orders_user_date
ON orders(user_id, order_date)
INCLUDE (order_status, total_amount);
```

6.2 형 변환 지양 (Implicit Conversion)

문제	예시	성능 영향
암시적 형 변환	WHERE col = '123' (col이 INT)	인덱스 사용 안 됨
문자열 날짜 비교	WHERE date_column = '2025-01-01' (date_column이 DATETIME)	인덱스 미사용 가능성
변환 함수 사용	WHERE CONVERT(VARCHAR, date_col) = '...'	인덱스 완전 무시

🔥 반드시 데이터 타입 일치하도록 작성!

6.3 SELECT * 지양

항목	설명
SELECT *는 항상 지양	불필요한 데이터 읽기 발생 → I/O 증가
컬럼 구조 변경 시 장애 위험	칼럼 추가되면 응용 프로그램 오류 발생 가능성
필요한 컬럼만 명시	인덱스 커버리지 확보에도 유리

```
-- ❌
SELECT *
  FROM users
;

-- ✅
SELECT user_id
      ,user_name
      ,created_at
  FROM users
;
```

6.4 LIMIT 조건 설정 (TOP / OFFSET)

항목	설명
대용량 테이블에서는 페이징 필수	UI 화면에서는 반드시 LIMIT or TOP 사용
OFFSET-FETCH 사용 시 인덱스 필요	정렬 컬럼에 인덱스 없으면 성능 급감

```
-- 예: 최근 주문 10건만 조회
SELECT TOP 10 *
  FROM orders
ORDER BY order_date DESC
;
```

6.5 JOIN 최적화

규칙	설명
ON 조건 명확히	불명확하면 CROSS JOIN으로 처리됨
양쪽 조인 키 인덱스 필요	외래키, 기본키 양쪽에 인덱스

필요 없는 테이블은 조인 지양	서브쿼리 또는 EXISTS로 대체 고려
JOIN 순서와 JOIN 방식 주의	SQL Server는 실행 계획 기반으로 자동 결정하나, 통제 필요시 힌트 사용 가능

6.6 서브쿼리 vs JOIN vs EXISTS

구문	특징	권장 상황
EXISTS	빠름, 조건만 체크	존재 여부 판단 (예: 필터링)
IN	적은 수 데이터 OK	많으면 성능 저하
JOIN	다량 데이터 통합 시 유리	많은 양의 결과 병합 필요 시
NOT IN	NULL 포함 시 예외 발생	NOT EXISTS로 대체 권장

-- 성능 좋은 EXISTS 예시

```
SELECT *
  FROM users u
 WHERE EXISTS ( SELECT 1
                  FROM orders o
                 WHERE o.user_id = u.user_id
                   AND o.order_date >= '2025-01-01')
;
```

6.7 트랜잭션 최소화

원칙	설명
트랜잭션은 가능한 한 짧게	트랜잭션 지속 시간 = 락 지속 시간
사용자 입력 대기 중 트랜잭션 금지	UI 처리 전에 트랜잭션 시작 지양
다량 DML은 배치 처리 고려	1,000건 단위로 나눠 처리 권장

6.8 집계 함수/정렬 최소화

- `ORDER BY`, `GROUP BY`, `DISTINCT` 등은 많은 리소스를 소모
  → 반드시 필요한 경우에만 사용

```
-- 성능 저하 요인
SELECT DISTINCT user_id
  FROM orders
;

-- 대체 예시
SELECT user_id
  FROM orders
GROUP BY user_id
;
```

6.9 통계(Statistics) 및 실행 계획 관리

항목	설명
통계는 최신 상태 유지	오래된 통계 → 비효율 실행 계획
자동 통계 업데이트 옵션 확인	기본값 AUTO_CREATE_STATISTICS, AUTO_UPDATE_STATISTICS 권장
실행 계획 확인	SSMS의 Actual Execution Plan 사용 권장

6.10 테이블 파티셔닝/아카이빙 고려 (대용량)

조건	적용 방안
1천만 건 이상 테이블	파티셔닝 or 아카이빙
오래된 데이터 조회 거의 없음	분리 테이블로 이전 후 인덱스 최적화
빈번한 삭제 작업 발생	파티션 스위칭으로 성능 개선 가능

6.11 잠금(Locking) 고려

항목	설명
SELECT 시 불필요한 잠금 방지	`WITH (NOLOCK)` 또는 READ COMMITTED SNAPSHOT 고려
대량 DML 작업 시 락 분산 필요	배치 처리 or 쿼리 조정
트랜잭션 간 교착 상태 감지	TRY-CATCH로 예외 처리 구성 권장

6.12 쿼리 리팩토링 고려 항목

지양	대안

중첩 SELECT	JOIN 또는 CTE로 풀기
조건문 안에서 함수 호출	함수 → 상위 쿼리로 분리
반복 쿼리 호출	WITH CTE or 임시 테이블로 처리

🔍 실무용 체크리스트

항목	체크
[ ] SELECT * 사용 지양	
[ ] 필요한 인덱스 생성	
[ ] WHERE 조건에 형 변환 없음	
[ ] LIMIT / 페이징 처리 있음	
[ ] 정렬 컬럼에 인덱스 존재	
[ ] 트랜잭션 범위 최소화	
[ ] 실행 계획 분석 완료	
[ ] 통계 최신 상태 유지	
[ ] 서브쿼리/EXISTS 적절 사용	
[ ] 대용량 테이블 아카이빙 고려	

7. 보안 및 트랜잭션

트랜잭션 예시

```
BEGIN TRY
    BEGIN TRANSACTION;

    -- 업무 로직 처리
    UPDATE accounts
       SET balance = balance - 1000
     WHERE user_id = 1;

    UPDATE accounts
       SET balance = balance + 1000
     WHERE user_id = 2;

    COMMIT;
END TRY

BEGIN CATCH
    ROLLBACK;

    -- 에러 처리
    DECLARE @error_message NVARCHAR(4000) = ERROR_MESSAGE();
    RAISERROR(@error_message, 16, 1);
END CATCH
;
```

보안 팁:

- 직접 GRANT/DENY 관리보다는 역할(Role) 기반 권한 제어
- EXECUTE 권한 최소화
- 로그 테이블 활용하여 감사 기록 저장

7.1 보안(Security)

SQL Server 보안은 크게 다음 5가지 영역에서 고려되어야 합니다:

1. SQL Injection 방지
2. 권한 최소화(Least Privilege)
3. 데이터 암호화
4. 감사 로그(Audit Logging)
5. 보안 개발 수칙 준수

7.1.1 SQL Injection 방지

🔶 원칙

- 사용자 입력을 직접 쿼리에 포함하지 않고, 반드시 **파라미터 바인딩** 또는 **Stored Procedure**를 사용할 것
-

❌ 잘못된 예

```
-- 취약한 코드 (SQL Injection 가능)
DECLARE @sql NVARCHAR(MAX)
SET @sql = 'SELECT * FROM Users WHERE username = ''' +
@username + ''''
EXEC(@sql)
```

✅ 안전한 예

```
-- sp_executesql 사용 (권장)
DECLARE @sql NVARCHAR(MAX)
SET @sql = N'SELECT * FROM Users WHERE username = @username'
EXEC sp_executesql @sql, N'@username NVARCHAR(50)', @username
= @inputUsername
```

✅ 또는 Stored Procedure 사용

```
-- 프로시저 정의
CREATE PROCEDURE GetUserByUsername
    @username NVARCHAR(50)
AS
BEGIN
    SELECT * FROM Users WHERE username = @username
END

-- 호출
EXEC GetUserByUsername @username = 'admin'
```

7.1.2 최소 권한 원칙 (Least Privilege)

🔶 원칙

- DB 계정은 필요한 권한만 갖도록 제한
- SELECT, INSERT, UPDATE, DELETE 권한을 개별적으로 할당
- DDL 권한은 개발자 또는 DBA 전용 계정에서만 수행

✅ 예시

```
-- 사용자에게 특정 테이블에 SELECT 권한만 부여
GRANT SELECT ON dbo.Orders TO reporting_user
```

🔐 팁
- **뷰(View)**로 민감 데이터 노출 최소화
- Role 기반 권한관리 권장 (DB 역할 분리)

7.1.3 민감 정보 암호화 (TDE, Always Encrypted 등)

🔶 저장 데이터 암호화 (At Rest)
- TDE (Transparent Data Encryption): 데이터베이스 전체 암호화
- **Always Encrypted**: 특정 열만 암호화 (개발자도 내용 볼 수 없음)

🔶 예시 - Always Encrypted
```
-- 열 암호화 마스터 키 생성 (한 번만)
CREATE COLUMN MASTER KEY CMK_Auto1
WITH (
    KEY_STORE_PROVIDER_NAME = N'MSSQL_CERTIFICATE_STORE',
    KEY_PATH = N'CurrentUser/My/CertificateName'
);

-- 열 암호화 키 생성
CREATE COLUMN ENCRYPTION KEY CEK_Auto1
WITH VALUES (
    COLUMN_MASTER_KEY = CMK_Auto1,
    ALGORITHM = 'RSA_OAEP',
    ENCRYPTED_VALUE = ...
);
```

🔐 팁
- 비밀번호, 주민등록번호 등은 반드시 암호화 저장
- 복호화 권한은 최소화

7.1.4 감사 로그 (Audit Logging)

🔶 로깅 대상
- 로그인/로그아웃
- 실패한 로그인 시도
- 중요 테이블의 INSERT/UPDATE/DELETE
- DB 권한 변경

✅ 예시 (사용자 정의 로그 테이블)

```
CREATE TABLE AuditLog (
    LogID INT IDENTITY PRIMARY KEY,
```

```
    UserName NVARCHAR(100),
    Action NVARCHAR(200),
    ActionTime DATETIME DEFAULT GETDATE(),
    TableName NVARCHAR(100),
    PrimaryKeyValue NVARCHAR(100)
)
```

⛔ 주의

- 로그 테이블은 일반 사용자 접근 불가 처리
- SQL Server Audit 기능 사용 권장 (Enterprise Edition)

7.2 트랜잭션(Transaction)

🔶 트랜잭션이란?
데이터 작업의 **원자성(Atomicity)**, **일관성(Consistency)**, **격리성(Isolation)**, **지속성(Durability)**를 보장하는 SQL Server의 핵심 기능입니다. (**ACID 원칙**)

7.2.1 트랜잭션 기본 구조

```
BEGIN TRANSACTION;

-- 작업 수행
UPDATE Accounts
   SET Balance = Balance - 100
 WHERE UserId = 1
;

UPDATE Accounts
   SET Balance = Balance + 100
 WHERE UserId = 2
;

COMMIT;
```

7.2.2 트랜잭션 예외 처리 (T-SQL)

```
BEGIN TRY
    BEGIN TRAN;

    -- 여러 DML 작업 수행
    DELETE FROM Orders WHERE OrderDate < '2023-01-01';
```

```
    UPDATE Inventory SET Quantity = Quantity - 1 WHERE ItemID
= 123;

    COMMIT;
END TRY
BEGIN CATCH
    ROLLBACK;

    INSERT INTO ErrorLog(ErrorMessage, ErrorDate)
    VALUES (ERROR_MESSAGE(), GETDATE());
END CATCH
```

✅ 권장

- 항상 BEGIN TRY...END TRY + ROLLBACK 구조 사용
- 커밋 전까지는 모든 변경이 DB에 반영되지 않음

7.2.3 격리 수준(Isolation Level)

SQL Server에서 설정 가능한 격리 수준은 다음과 같습니다:

격리 수준	특징	문제 방지
READ UNCOMMITTED	Dirty Read 허용	X
READ COMMITTED	기본값, 커밋된 데이터만 읽음	Dirty Read 방지
REPEATABLE READ	읽은 데이터 재확인 가능	Non-repeatable Read 방지
SERIALIZABLE	완전한 트랜잭션 격리	Phantom Read 방지
SNAPSHOT	버전 기반 격리 (TempDB 사용)	대부분의 문제 방지, 성능 우수

예시

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION;

-- 데이터 읽기
SELECT *
  FROM Orders
 WHERE CustomerID = 1001
;

-- 처리 로직...
COMMIT;
```

7.2.4 트랜잭션 관련 개발 원칙

원칙	설명
트랜잭션은 짧게	가능하면 짧은 시간 내 커밋
예외 발생 시 반드시 ROLLBACK	항상 예외 처리를 넣고 ROLLBACK 수행
순차적 락 획득	교착 상태 방지
외부 호출 포함 금지	트랜잭션 중 외부 API 호출 X

✅ 보안 및 트랜잭션 요약 체크리스트

항목	체크 여부
[ ] 파라미터 바인딩 또는 프로시저 사용	
[ ] 사용자 권한 최소화 적용	
[ ] 민감 정보 암호화 처리	
[ ] 감사 로그 기록 기능 구현	
[ ] 트랜잭션 BEGIN / COMMIT / ROLLBACK 명시	
[ ] TRY-CATCH로 예외 처리 구현	
[ ] 트랜잭션 격리 수준 설정 확인	
[ ] 트랜잭션 내 외부 호출 제거	

8. 코드 관리

항목	설명
형상관리	모든 SQL 스크립트는 Git 등 VCS에 저장
배포 관리	배포 시점/버전 기록 관리
파일 구조	`/procedures`, `/functions`, `/tables`, `/views` 등 디렉토리 구분 권장

📌 코드 관리란?
SQL 코드(Stored Procedure, View, Function, Trigger, DDL 등)를 **버전 관리, 표준화, 배포 및 유지보수** 가능한 형태로 **체계적으로 운영**하는 방법론입니다.

8.1 객체명 명명 규칙 (Naming Convention)

✅ 원칙

- 객체의 역할과 종류를 **일관된 규칙**으로 명확히 구분
- 접두어/접미어 사용 권장
- 팀/회사 내 표준 명명 규칙 문서화

📌 예시

객체 유형	접두어 예	명명 예시
Table	없음 또는 `tbl_`	`Customer`, `tbl_Order`
View	`vw_`	`vw_CustomerList`
Procedure	`usp_`	`usp_GetCustomerById`
Function	`ufn_`	`ufn_GetTotalAmount`
Trigger	`trg_`	`trg_Audit_OrderInsert`
Index	`IX_`	`IX_Order_OrderDate`
Constraint	`CK_`, `PK_`, `FK_`	`FK_Order_CustomerID`

8.2 소스 코드 버전 관리

✅ 권장 도구

- Git, GitHub / GitLab / Bitbucket
- Azure DevOps 또는 TFS
-

📂 디렉토리 구조 예시

```
/DatabaseProject
 │
 ├── /Tables
 │    └── Customer.sql
 ├── /Views
 │    └── vw_CustomerList.sql
 ├── /StoredProcedures
 │    └── usp_GetCustomerById.sql
 ├── /Functions
 │    └── ufn_CalculateDiscount.sql
 ├── /Triggers
 │    └── trg_Audit_OrderInsert.sql
 ├── /Migrations
 │    └── 2025-08-13_Add_New_Column_Order.sql
 ├── /SeedData
 │    └── init_data.sql
 └── README.md
```

✅ 관리 팁

- 모든 객체는 별도 `.sql` 파일로 저장
- 변경 이력은 Git 커밋 메시지로 추적
- 배포용 스크립트와 개발용 스크립트 구분

8.3 코드 배포 및 변경 관리

✅ 변경 이력 관리 (Schema Change History)

- DDL 변경 내역 관리 필수
- 버전 별로 배포 스크립트 생성 (YYYYMMDD_Description.sql)
- `CHANGE_LOG` 테이블로 수동 이력 관리 가능

예시

```
CREATE TABLE SchemaChangeLog (
    ChangeID INT IDENTITY PRIMARY KEY,
    ChangeDate DATETIME DEFAULT GETDATE(),
    ScriptName NVARCHAR(200),
```

```
    Description NVARCHAR(500),
    AppliedBy NVARCHAR(100)
);
```

배포 스크립트 예시

```
-- 20250813_AddStatusToOrder.sql
BEGIN TRANSACTION;

ALTER TABLE Order ADD Status NVARCHAR(50) NOT NULL DEFAULT
'Pending';

INSERT INTO SchemaChangeLog
(
 ScriptName
,Description
,AppliedBy
)
VALUES
(
 '20250813_AddStatusToOrder.sql'
,'주문 상태 컬럼 추가'
,SYSTEM_USER
)
;

COMMIT;
```

8.4 코드 표준화 (Style Guide)

✅ 코딩 스타일 가이드

항목	규칙
대소문자	SQL 키워드는 대문자, 식별자는 소문자 또는 PascalCase
들여쓰기	4칸 스페이스 또는 탭 일관 사용
구문 분리	SELECT, FROM, WHERE 구문은 줄바꿈
주석	블록 주석(/* */)과 라인 주석(--) 적극 사용

예시 (스타일 적용)

```
-- 고객 정보 조회 프로시저
CREATE PROCEDURE usp_GetCustomerById
    @CustomerId INT
```

```
AS
BEGIN

    SET NOCOUNT ON;

    SELECT c.CustomerId
          ,c.Name
          ,c.Email
      FROM Customer AS c
     WHERE c.CustomerId = @CustomerId
    ;

END
```

8.5 테스트 및 검증 코드 관리

✅ 유닛 테스트 SQL

- 각 프로시저/함수에는 테스트 SQL 별도 관리
- `Tests` 디렉토리에 저장

예시

```
-- Tests/test_usp_GetCustomerById.sql
EXEC usp_GetCustomerById @CustomerId = 1;
```

8.6 롤백 및 복구 전략

✅ 전략

상황	롤백 방식
DML 오류	트랜잭션 내 ROLLBACK 사용
DDL 오류	백업 또는 이전 버전 스크립트로 복구
배포 스크립트	항상 BEGIN TRAN / ROLLBACK 옵션 포함
긴급 상황	전체 DB 백업에서 복구 필요

8.7 코드 리뷰 및 승인 프로세스

- Pull Request(PR)로 코드 변경 사항 검토
- 최소 1인 이상 리뷰어 승인 필요
- 보안 및 성능 관련 검토 항목 포함

항목	체크
[ ] SQL Injection 방지 여부 확인	
[ ] 트랜잭션 정상 처리 여부	
[ ] 성능 저하 가능성 (인덱스 활용 등)	
[ ] Naming Rule 준수 여부	
[ ] 주석 및 설명 포함 여부	

8.8 문서화

- 모든 객체(Procedure, Function 등)는 기능 설명 주석 포함
- 스키마 설명서 자동화 도구 활용 가능 (예: Redgate SQL Doc, ApexSQL Doc)
- ERD 및 객체 의존성 문서 유지

✅ 코드 관리 요약 체크리스트

항목	필수 여부	완료 여부
명명 규칙 문서화 및 일관성	✅	[ ]
Git으로 SQL 코드 버전 관리	✅	[ ]
객체별 코드 분리 및 저장	✅	[ ]
배포 스크립트 별도 관리	✅	[ ]
트랜잭션 처리 포함 여부 확인	✅	[ ]
롤백/복구 계획 마련	✅	[ ]
코드 리뷰 프로세스 운영	✅	[ ]
테스트 스크립트 작성 및 검증	✅	[ ]
주석 및 문서화	✅	[ ]

9. 코드 리뷰 체크리스트

- SELECT * 사용 지양
- 명명 규칙 지켜졌는가?
- JOIN 조건 명확한가?
- 성능 저하 요인 있는가?
- 에러 및 예외 처리 포함되어 있는가?
- 주석 충분한가?
- 보안 고려되었는가?

9.1 기본 구성 및 스타일

항목	설명	체크
[ ] 명명 규칙 준수	객체명은 사내 정의된 네이밍 컨벤션을 따르는가? (`usp_`, `vw_`, `ufn_` 등)	
[ ] 코딩 스타일 통일	키워드는 대문자, 식별자는 소문자/PascalCase, 들여쓰기 유지	
[ ] 주석 작성 여부	프로시저/함수의 목적, 파라미터, 리턴값에 대한 주석이 있는가?	
[ ] 불필요한 코드 제거	테스트용 코드, 미사용 변수/임시 테이블이 제거되었는가?	
[ ] 코드 가독성	쿼리, 조건문, 조인 등이 읽기 쉽게 정렬되었는가?	

9.2 보안(Security)

항목	설명	체크
[ ] SQL Injection 방지	사용자 입력은 문자열 결합이 아닌 바인딩 변수 또는 프로시저로 처리했는가?	
[ ] 권한 최소화 고려	필요한 최소 권한만 부여되도록 설계되었는가? (예: SELECT만 허용)	
[ ] 민감정보 암호화 고려	주민등록번호, 계좌번호 등은 암호화 저장 또는 마스킹 처리되는가?	
[ ] 감사 로깅 포함 여부	중요한 INSERT/UPDATE/DELETE에 대해 Audit 로그 또는 기록이 있는가?	

9.3 트랜잭션 및 예외 처리

항목	설명	체크
[ ] 트랜잭션 사용 적절성	여러 DML 작업이 하나의 논리적 작업이면 BEGIN TRAN/COMMIT 사용 여부	
[ ] ROLLBACK 처리	오류 발생 시 트랜잭션을 ROLLBACK 하는 구조가 있는가?	
[ ] TRY-CATCH 블록 사용	예외 처리용 TRY-CATCH 구문이 작성되어 있는가?	
[ ] 에러 로그 기록	오류 발생 시 ErrorLog 테이블 또는 다른 방식으로 기록되는가?	
[ ] 트랜잭션 범위 최소화	트랜잭션 범위가 작고 빠르게 커밋되는가? (락 최소화)	

9.4 성능 최적화

항목	설명	체크
[ ] 인덱스 활용 여부	WHERE, JOIN 조건에 인덱스가 존재하는 컬럼이 사용되는가?	
[ ] 불필요한 SELECT 제거	`SELECT *` 대신 필요한 컬럼만 명시했는가?	
[ ] 적절한 조인 사용	JOIN 방식(Nested Loop, Hash, Merge)과 조건이 적절한가?	
[ ] 비효율적 서브쿼리 제거	반복적 실행되는 서브쿼리는 CTE나 JOIN으로 대체 가능한가?	
[ ] 쿼리 실행계획 확인 여부	실행계획을 통해 성능 병목이 없는지 검토했는가?	
[ ] NOLOCK 또는 READPAST 검토	조회 성능을 위해 NOLOCK 사용이 필요한가? 오용은 없는가?	

9.5 객체 간 의존성 및 영향도

항목	설명	체크
[ ] 테이블/프로시저 의존성 고려	수정한 객체가 다른 객체에 영향을 주는가?	
[ ] 타 시스템/서비스 연동 여부	API, 메시지큐, 외부 호출 등 외부 시스템에 미치는 영향이 고려되었는가?	
[ ] DDL 변경 영향도 검토	테이블 구조 변경이 기존 코드나 프로시저에 영향을 주는가?	

9.6 테스트 및 검증

항목	설명	체크
[ ] 테스트 코드 존재 여부	기능 테스트를 위한 실행 스크립트 또는 테스트 케이스가 존재하는가?	
[ ] 경계값 테스트 수행	NULL, 0, 음수, 특수문자 등의 입력값 테스트가 수행되었는가?	
[ ] 오류 재현 테스트	예외 상황이 정상적으로 처리되는지 테스트되었는가?	

9.7 배포 및 롤백 가능성

항목	설명	체크
[ ] 버전 기록 확인	변경된 스크립트에 변경일, 작성자, 버전 주석이 포함되어 있는가?	
[ ] 배포 시나리오 작성 여부	배포 순서, 의존 객체 반영 순서 등이 문서화되었는가?	
[ ] 롤백 스크립트 포함	긴급 상황 대비 롤백 쿼리 또는 백업 스크립트가 준비되었는가?	

📘 부록: 체크리스트 템플릿 (표 형태)

항목	상태(✅/❌)	비고
명명 규칙 준수		
주석 및 설명 포함 여부		
트랜잭션 처리		
예외 처리 TRY-CATCH		
SQL Injection 방지		
SELECT * 사용 금지		
실행계획 검토		
테스트 코드 확인		
롤백/백업 고려		

10. 인덱스와 조인 (Index & Join)

10.1 인덱스 (Index)

10.1.1 인덱스의 개념

- **인덱스(Index)**는 테이블 내 데이터를 빠르게 조회하기 위한 자료구조입니다.
- 책의 목차와 같은 역할을 하며, 성능 최적화의 핵심입니다.

10.1.2 인덱스 종류

종류	설명	예시
Clustered Index	테이블의 데이터를 정렬된 순서로 저장 (테이블 = 인덱스)	PK_OrderID (기본키 자동 생성)
Non-Clustered Index	별도 저장 구조, 필요한 데이터 위치만 참조	IX_Customer_Email
Unique Index	중복 허용 안 되는 인덱스	UQ_Customer_SSN
Filtered Index	WHERE 조건이 있는 인덱스	IX_Order_OnlyActive (WHERE Status = 'Active')
Columnstore Index	대용량 분석용 컬럼 기반 인덱스	CSIX_Sales

10.1.3 인덱스 작성 가이드

✅ 인덱스 생성 예시

```
-- 일반 인덱스 생성
CREATE NONCLUSTERED INDEX IX_Customer_Email
ON Customer(Email);

-- 다중 컬럼 인덱스
CREATE NONCLUSTERED INDEX IX_Order_CustomerID_OrderDate
ON Orders(CustomerID, OrderDate);

-- 필터 인덱스
CREATE NONCLUSTERED INDEX IX_Order_ActiveOnly
ON Orders(Status)
WHERE Status = 'Active';
```

10.1.4 인덱스 주의사항

- 너무 많은 인덱스는 **DML 성능 저하** 유발 (INSERT/UPDATE/DELETE)
- 선택도(Selectivity)가 낮은 컬럼엔 인덱스 부적합 (예: 성별, 상태)
- 인덱스 조합 컬럼 순서 중요 → 자주 사용하는 WHERE 조건 컬럼이 먼저
- 정기적으로 인덱스 조각모음(리빌드/리오르그) 필요

10.1.5 인덱스 활용 점검

- SET STATISTICS IO ON → 읽은 페이지 수 확인
- 실행 계획(Actual Execution Plan)에서 **Index Seek**인지 **Scan**인지 확인
- 쿼리 튜닝 시 인덱스 제안 활용 가능

10.2. 조인 (Join)

10.2.1 조인의 종류

종류	설명	사용 예
INNER JOIN	양쪽 테이블에 모두 존재하는 값만 반환	회원과 주문 정보
LEFT JOIN	왼쪽 테이블의 모든 값 + 오른쪽 일치하는 값	전체 고객 + 구매 여부
RIGHT JOIN	오른쪽 테이블의 모든 값 + 왼쪽 일치하는 값	미사용 권장 (가독성 낮음)
FULL OUTER JOIN	양쪽 테이블의 모든 값	드물게 사용
CROSS JOIN	데카르트 곱 (주의 필요)	조합 계산 등 특수 용도

10.2.2 조인 작성 예시

```
-- INNER JOIN 예시
SELECT o.OrderID
      ,c.Name
  FROM Orders o
       INNER JOIN Customer c
          ON o.CustomerID = c.CustomerID
;

-- LEFT JOIN 예시
SELECT c.CustomerID
```

```
      ,c.Name
      ,o.OrderID
  FROM Customer c
       LEFT JOIN Orders o
         ON c.CustomerID = o.CustomerID
;
```

10.2.3 조인 작성 가이드라인

- **ON 절에 명확한 조건**을 걸 것 (모호한 조인은 성능 저하)
- 조인 대상 컬럼에 **인덱스 존재 여부** 확인
- 조인 컬럼의 데이터 타입이 **동일한지 확인** (암시적 변환은 성능 이슈 발생)

10.2.4 성능 고려사항

항목	설명
조인 순서	SQL Server는 옵티마이저가 자동 결정하지만, 통계에 의존
인덱스 활용	조인 컬럼에 인덱스가 없으면 **Hash Join** 또는 **Nested Loop Scan** 유발
WHERE vs ON	조인 조건은 반드시 `ON` 절에 명시, `WHERE`로 조건 거는 건 결과 오류 가능
조인 수 제한	다중 조인 시 성능 급감 → 3~4개 이상 조인은 반드시 실행 계획 검토

10.2.5 조인 시 주의할 점

- 동일 테이블을 여러 번 조인할 경우 **테이블 별칭 필수**
- 외부 조인(LEFT/RIGHT) 결과 필터링 시 `NULL` 조건을 잘 고려할 것
- **CROSS JOIN**은 실수로 사용 시 폭발적 데이터 증가 유발

10.3 인덱스 + 조인 최적화 전략

전략	설명
조인 컬럼 인덱스 설정	외래키, 고객ID 등은 인덱스 필수
인덱스 힌트는 최후 수단	옵티마이저가 더 좋은 계획을 선택할 수도 있음
조회 범위 축소	WHERE 조건으로 스캔 범위를 제한
실행 계획 분석	실제 실행 계획에서 "Index Seek"가 나오는지 확인
통계 최신화	오래된 통계는 잘못된 실행계획을 유발할 수 있음 (`UPDATE STATISTICS`)

✅ 체크리스트: 인덱스 & 조인

항목	체크
[ ] 조인 조건에 인덱스가 설정되어 있는가?	
[ ] 조인 컬럼 간 데이터 타입이 일치하는가?	
[ ] 불필요한 CROSS JOIN 또는 FULL OUTER JOIN이 없는가?	
[ ] SELECT * 사용을 지양하고 필요한 컬럼만 조회하는가?	
[ ] WHERE 절에 적절한 조건이 있어 인덱스 사용을 유도하는가?	
[ ] 실행 계획 분석으로 성능 문제를 사전에 점검했는가?	
[ ] LEFT JOIN 결과에서 NULL 처리 고려가 되었는가?	

11. (NOLOCK) 사용에 대하여

11.1 NOLOCK이란?

- `WITH (NOLOCK)`은 **공유 잠금(Shared Lock)**을 무시하고 데이터를 읽는 **테이블 힌트(Table Hint)**입니다.
- 트랜잭션이 완료되지 않은 **커밋되지 않은 데이터(Uncommitted Data)**도 읽을 수 있습니다.

```
SELECT *
  FROM Orders WITH (NOLOCK)
;
```

📌 NOLOCK은 `READ UNCOMMITTED` 격리 수준과 동일한 동작을 합니다.

11.2 작동 방식 요약

항목	설명
잠금 회피	다른 트랜잭션이 잠금 중이어도 기다리지 않고 읽음
Dirty Read 허용	아직 커밋되지 않은 데이터도 읽음
트랜잭션과 독립적 읽기	실행 시점 기준으로 읽지만, 일관성 보장 없음

11.3 장점

장점	설명
🔓 잠금 회피	다른 사용자가 테이블에 잠금을 걸고 있어도 대기하지 않음
🚀 조회 성능 개선	대용량 테이블 조회 시 블로킹 없이 빠른 응답 가능
⚡ 리포트 쿼리용 유용	OLAP 성 보고용/모니터링 쿼리에 적절

11.4 심각한 단점

단점	설명
⚠️ Dirty Read	커밋되지 않은 데이터 읽기 (ROLLBACK 시 잘못된 데이터 노출)

❗ Phantom Read	데이터가 중간에 바뀌거나 추가되어 읽을 때마다 결과가 달라질 수 있음
❌ 잘못된 조인 결과	조인된 테이블 중 하나라도 트랜잭션 중이면 전체 조인 결과가 왜곡될 수 있음
🧟 데이터 일관성 깨짐	동일 쿼리를 여러 번 실행해도 결과가 달라질 수 있음
💥 비즈니스 로직 오류	재고, 금융, 결제, 계좌 등 정확성이 중요한 로직에는 절대 금지

11.5 예제 비교

🎯 정상 커밋 기반 쿼리

```
SELECT *
  FROM Orders
;
```

-- 다른 트랜잭션이 Commit될 때까지 기다림

⚠️ NOLOCK 사용

```
SELECT *
  FROM Orders WITH (NOLOCK)
;
```

-- Commit 되지 않은 INSERT/UPDATE/DELETE도 읽을 수 있음
결과적으로 "보고되지 말아야 할 중간 데이터"가 사용자에게 보여질 수 있음

11.6 사용이 적절한 경우

상황	설명
리포트용 실시간 데이터 조회	약간의 오차 허용 가능할 때
운영 DB의 대용량 조회시 블로킹 최소화 필요	단순 SELECT, 모니터링 쿼리 등
데이터 정합성보다 성능이 우선일 때	예: 대시보드 조회, 통계 페이지 등

❌ 사용하면 안 되는 경우

상황	설명
트랜잭션 중 사용	NOLOCK으로 읽은 데이터를 기반으로 UPDATE/DELETE 하는 경우
금융, 회계, 정산 시스템	금액, 수량, 상태 등 정밀한 데이터 판단이 필요한 경우
JOIN 또는 WHERE 절에 사용	하나라도 잘못 읽으면 전체 쿼리 결과가 오류 발생
데이터 무결성이 중요할 때	동일한 쿼리를 실행해도 결과가 달라지는 상황이 발생

✅ 안전한 대안

대안	설명
READ COMMITTED SNAPSHOT	버전 기반 읽기로 일관성 유지하면서 블로킹 방지
WITH (READCOMMITTED)	커밋된 데이터만 읽음, 기본값
REPEATABLE READ	조회 중 데이터 변경 방지
INDEX 최적화	성능 문제는 NOLOCK이 아닌 인덱스로 해결할 수 있는 경우가 많음

11.7 권장 정책

항목	정책
사용 기본값	NOLOCK 금지
예외 사용	리포트/모니터링용 쿼리에 한해 제한적 사용, 사전 리뷰 필수
코드 리뷰 기준	NOLOCK 사용 여부를 코드 리뷰 체크리스트에 포함
옵션 로그	NOLOCK 사용 쿼리 추적 및 로깅 권장

📋 사용 여부 체크리스트

체크 항목	확인
[ ] NOLOCK이 필요한 이유가 명확히 설명되었는가?	
[ ] 결과의 불일치가 비즈니스상 무해한가?	
[ ] 조회 쿼리로만 사용되며, 데이터 변경에는 연계되지 않는가?	
[ ] 테스트 환경에서 결과 왜곡 여부를 확인했는가?	
[ ] 운영환경 사용 시 반드시 운영 승인이 되었는가?	

✍️ 예시: 리포트 용도 제한적 허용

```
-- 주간 통계 리포트: 변경되지 않은 데이터 조회, 약간의 지연 허용
SELECT COUNT(*) AS TotalOrders
  FROM Orders WITH (NOLOCK)
 WHERE OrderDate BETWEEN '2025-08-01' AND '2025-08-07'
;
```

📌 결론

❗ (NOLOCK)은 읽기 속도를 높이기 위한 칼날이지만, 데이터 정합성이라는 심장을 찌를 수 있습니다.

- 정확성이 중요하다면 절대 사용하지 말 것
- 성능 최적화는 먼저 쿼리 튜닝 → 인덱스 → 통계 관리를 고려할 것
- 사용 시 반드시 **리스크를 알고 문서화**한 뒤 활용할 것