SQLチューニング事例

1日目	2日目	3日目	4日目	5日目	6日目	7日目
8日目	9日目	10日目	11日目	12日目	13日目	14日目
15日目	16日目	17日目	18日目	19日目	20日目	21日目

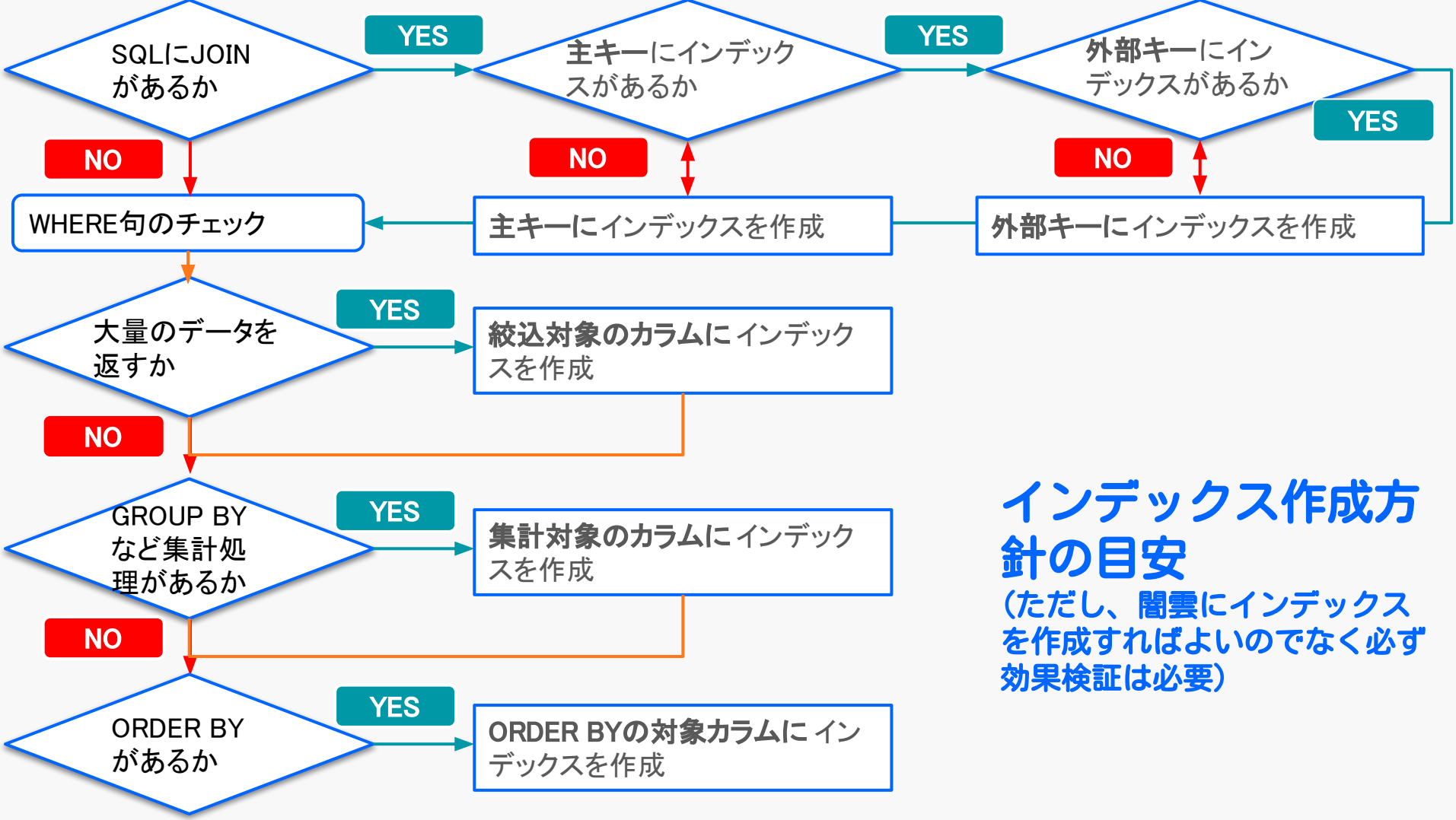
SQLチューニングの基本的な方法

1. **インデックスを活用する**
   インデックスを用いたチューニング(**チューニング事例1**)、インデックスが利用されない(**チューニング事例2**)

2. **無駄な処理をなくす**
   意味のない並びかえを避ける(**チューニング事例3**)、同じ処理を何度も行わない(**チューニング事例4**)

3. **効率の良いSQLの使い方をする**
   INかEXISTSか(**チューニング事例5**)、その他のありがちなチューニング事例(**チューニング事例6**)

SQLチューニング事例１
（インデックスを利用する）



インデックス作成方針の目安

（ただし、闇雲にインデックスを作成すればよいのでなく必ず効果検証は必要）

1-1. インデックスを作成することでSQLの処理速度を向上

絞込処理を行った場合に、その対象のカラムに対してインデックスを作成するとフルスキャンからインデックススキャンに変わり、処理速度が向上する
(ただし、絞り込める件数が全体の15%以下であること)

BAD

```
SELECT * FROM employees
WHERE name="Taro";
```

GOOD

```
CREATE INDEX idx_employees_name ON employees(name)

SELECT * FROM employees
WHERE name="Taro";
```

1-2. 複合インデックスを作成する

絞込み条件で複合インデックスを利用する場合には、インデックスは特殊な使われ方をする

```
# nameとageにインデックスを作成
CREATE INDEX idx_employees_name_age ON employees(name, age)

SELECT * FROM employees
WHERE name="Taro" AND age=21;# ANDの場合、インデックスを利用できる

SELECT * FROM employees
WHERE name="Taro" OR age=21;# ORの場合、インデックスを利用できない

SELECT * FROM employees
WHERE name="Taro";# nameだけでも、インデックスを利用できる

SELECT * FROM employees
WHERE age=19;# ageだけでは、インデックスを利用できない
```

1-2. 複合インデックスでなく各カラムにインデックスを作成する

複合インデックスでなく、**各カラムにインデックスを作成する**

```
# nameとageにインデックスを作成
CREATE INDEX idx_employees_name ON employees(name)
CREATE INDEX idx_employees_name ON employees(age)

SELECT * FROM employees
WHERE name="Taro" AND age=21;# ANDの場合、インデックスを利用できる

SELECT * FROM employees
WHERE name="Taro" OR age=21;# ORの場合、インデックスを利用できる

SELECT * FROM employees
WHERE name="Taro";# nameだけでも、インデックスを利用できる

SELECT * FROM employees
WHERE age=19;# ageだけでも、インデックスを利用できる
```

1-3. ORDER BYとGROUP BYの対象カラムにインデックスを作成

ORDER BYとGROUP BYに使用される対象カラムにインデックスを付与すると、処理速度を向上できる。対象カラムが複数ある場合は、複合インデックスを作成する。

```
SELECT * FROM employees ORDER BY name
SELECT name, COUNT(*) FROM employees GROUP BY name

# nameにインデックスを作成してORDER BY, GROUP BYを高速化する
CREATE INDEX idx_employees_name ON employees(name)


SELECT * FROM employees ORDER BY name, age
SELECT * FROM employees GROUP BY name, age
# name,ageにインデックスを作成してORDER BY, GROUP BYを高速化する
CREATE INDEX idx_employees_name ON employees(name, age)
```

1-4. 外部キーにインデックスを追加する

外部キーにインデックスを追加することでネステッドループの外部表と内部表をオプティマイザが **適切に選択するようになり** 処理速度が向上することがある

```
SELECT * FROM orders AS od
INNER JOIN items AS it
ON od.item_id =it.id;


# 外部キーにインデックスを追加
CREATE INDEX idx_orders_item_id ON orders(item_id);
```

***) MySQLでは、外部キー制約をつけると自動的にインデックスも付与される**

SQLチューニング事例2
（インデックスが利用されない例）

2-1. インデックスの対象カラムに関数を使う

インデックスは、カラムの値に対して作成される。ただし、カラムの値を関数で変換した場合には、 **インデックスが利用できずにフルスキャンが実行される**

BAD(nameにインデックスがあるが、関数をつけているため利用できない)

```
SELECT * FROM employees
WHERE LOWER(name)="taro";
```

GOOD(関数インデックスを作成する)

***) ただし、関数インデックスは万能ではないので、使用は注意が必要**

```
CREATE INDEX idx_employees_name_lower ON employees((LOWER(name)))

SELECT * FROM employees
WHERE LOWER(name)="taro";
```

2-2. インデックスの対象カラムが数値の場合に計算をする

インデックスのカラムでも数値演算をしたものを比較に使うと、 **インデックスが利用されない**

BAD(ageにインデックスがあるが、計算をしているため利用できない)

```
SELECT * FROM employees
WHERE age + 2 = 20;
```

GOOD(右辺の計算式を左辺に移して、左辺はカラムだけにする)

```
SELECT * FROM employees
WHERE age = 18;
```

2-3. 文字列のカラムの比較対象にシングルクォート，ダブルクォートをつけない

インデックスをCHAR型やVARCHAR型に張っている場合に、比較対象には、シングルクォートかダブルクォートをつけない場合、**左辺の数値への変換が行わ** れインデックスを利用できない

BAD(prefecture_code(CHAR型)にインデックスがあるが、利用できない)

```
SELECT * FROM employees
WHERE prefecture_code = 21;
```

GOOD(クォートをつける)

```
SELECT * FROM employees
WHERE prefecture_code = '21';
```

2-4. 後方一致検索、中間一致検索を利用している

LIKE句を用いて検索の場合、前方一致はインデックスを使用されるが、 **中間一致、後方一致はインデックスを利用できない。**

```
# 前方一致は、インデックスが利用される
SELECT * FROM employees
WHERE name LIKE "田中%";

# 後方一致は、インデックスが利用されない
SELECT * FROM employees
WHERE name LIKE "%太郎";
```

後方一致の改善案

```
# 名前を逆順に格納した、name_reverseカラムを用意して、前方一致検索をする
SELECT * FROM employees
WHERE name_reverse LIKE "郎太%";

# 中間一致は、インデックスが利用されない
SELECT * FROM employees
WHERE name LIKE "%田%";
```

SQLチューニング事例3
（意味のない並びかえを避ける）

意味のない並び替えを避ける

並び替えは処理時間がかかるため、極力避けることで処理時間を向上できる。
以下の処理は、並び替えが発生する、不必要な場合は極力並び替えを避けると処理を高速化できる

- ORDER BY
- GROUP BY
- 集計関数(SUM, COUNT, AVG, MIN, MAX)
- DISTINCT
- UNION, INTERSECT, EXCEPT
- ウィンドウ関数

3-1. 無駄なORDER BY, GROUP BYを避ける

ORDER BY, GROUP BYは処理時間がかかるため、必要な場合だけ利用するようにしましょう、
また、利用する際は、極力 **レコードの絞り込みができてから** 行いましょう

無駄なORDER BY(下のORDER BYは何の役割もない)

```
SELECT * FROM employees AS emp
INNER JOIN (SELECT * FROM departmentsORDER BY name) AS dp
ON emp.department_id = dp.id;
```

絞込みのできていないGROUP BY(HAVINGで絞り込むのでなく、WHEREで絞り込む)

```
SELECT
    department_name, COUNT(*)
FROM
    employees
GROUP BY department_name
HAVING department_name IN("経営企画部", "営業部");
```

3-2. MAX, MINの計算にはインデックスで高速化できる

MAX, MINの処理は、インデックスを利用するとすぐに値を取得できる

```
SELECT
  MAX(age), MIN(age)
FROM users

インデックスを作成する
CREATE INDEX idx_users_age ON users(age);

SELECT
  MAX(age), MIN(age)# インデックスで高速化
FROM users
```

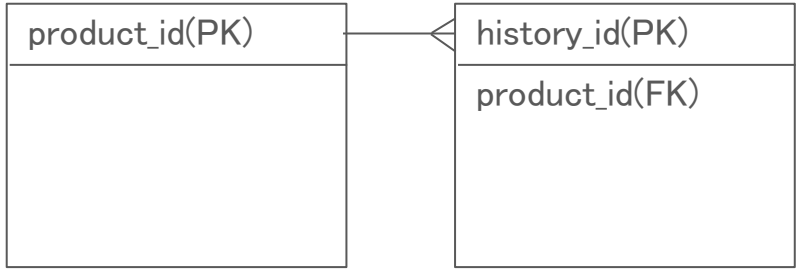
3-3. DISTINCTの代わりにEXISTSを利用する

重複を削除するDISTINCTは重複削除の前に並び替えがされるため、処理時間がかかる。並び替えは必要なく重複のない値のみが必要なパターンでは、EXISTSで代替できることがある

historiesにproduct_idが存在するレコードをproductsテーブルから取り出し、DISTINCTを使ってproduct_idの重複を削除

```
SELECT DISTINCT p.product_id
FROM products AS p
INNER JOIN histories AS h
ON p.product_id = h.product_id
```

productsテーブル



EXISTSでproduct_idを取り出す

```
SELECT
  p.product_id
FROM products AS p
WHERE EXISTS
(SELECT 1 FROM histories AS hWHERE h.product_id = s.product_id)
```

3-4. UNIONの代わりにUNION ALLを利用する

UNIONは結合の前に**順番の並び替えと重複削除が行われる**ため処理時間がかかる。重複のない行同士を紐づける場合は、UNION ALLを利用すればよい

UNIONは処理時間がかかるため、重複削除の必要がない（重複のない要素）場合は利用しない

```
SELECT GRADE, PERCENTAGE
FROM   STUDENT_GRADE_A
UNION
SELECT GRADE, PERCENTAGE
FROM   STUDENT_GRADE_B
```

UNION ALLで結合する

```
SELECT GRADE, PERCENTAGE
FROM   STUDENT_GRADE_A
UNION ALL
SELECT GRADE, PERCENTAGE
FROM   STUDENT_GRADE_B
```

SQLチューニング事例4
（同じ処理を何度も行わない）

4-1. 副問い合わせで同じテーブルを見る回数は、1回にする

SQLチューニングの基本として、**極力テーブルへのアクセスの回数を減らす** ことがある。1回でテーブルの中身を確認できる場合は1回で確認を済ませる

似たような副問い合わせが存在して、無駄に2度SELECTが実行される

```
SELECT
  *
FROM employees
WHERE
department_id IN (SELECT id FROM departments WHERE name="営業部")
OR
department_id IN (SELECT id FROM departments WHERE name="総務部") ;
```

同じ意味合いの副問い合わせ1つにする

```
SELECT
  *
FROM employees
WHERE
department_id IN (SELECT id FROM departments WHERE name IN("営業部", "総務部"));
```

4-2. SELECT内の副問い合わせをやめて、JOINを使う

SELECTの対象カラム内に、副問い合わせを使うと行ごとに副問い合わせのSQLが実行されて処理時間がかかるため、**JOINで1回の紐づけだけにする**

SELECTの対象カラムに副問い合わせを使うと行ごとにSELECTが実行されて、処理コストが余計にかかる

```
SELECT
  emp.*, (SELECT name FROM departments AS dp WHERE dp.id = emp.department_id)
FROM employees AS emp;
```

LEFT JOINで紐づけると同じ結果を得られる

```
SELECT
  emp.*, dp.name
FROM employees AS emp
LEFT JOIN departments AS dp
ON emp.department_id = dp.id;
```

4-3. 何度も無駄に絞込みを行わない

WHEREの絞込みも処理時間がかかる。なるべく回数を減らすと **処理の無駄をなくす** 事ができる

WHEREで2回絞り込んで、紐づけを行う

```
SELECT o1.*, o2.daily_summary
FROM orders AS o1
LEFT JOIN(
    SELECT order_date, SUM(order_amount * order_price) AS "daily_summary" FROM orders
    WHERE order_date BETWEEN "2020-01-01" AND "2020-01-31"
    GROUP BY order_date
) AS o2
ON o1.order_date = o2.order_date
WHERE o1.order_date BETWEEN "2020-01-01" AND "2020-01-31";
```

ウィンドウ関数を用いて、絞込1回と集計1回にする

```
SELECT
    *, SUM(o1.order_amount * o1.order_price) OVER(PARTITION BY o1.order_date) AS daily_summary
FROM orders AS o1
WHERE o1.order_date BETWEEN "2020-01-01" AND "2020-01-31";
```

SQLチューニング事例5
（INかEXISTSか）

INを使う場合

絞り込む対象のテーブルの行数が、副問い合わせで絞り込まれるレコードの行数よりも **大きい** 場合

```
T1 > T2
SELECT * FROM T1 WHERE id IN (SELECT id FROM T2)
# T2からレコードを取得して、T1と紐づけを行い、紐づけができたレコードのみをT1から取り出す
```

EXISTSを使う場合

絞り込む対象のテーブルの行数が、相関副問い合わせの対象レコード行数よりも **小さい** 場合

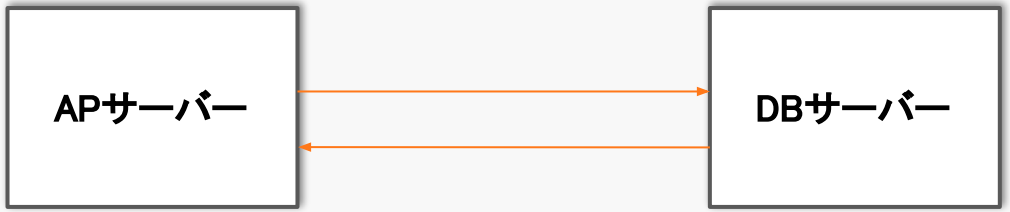
```
T1 < T2
SELECT * FROM T1 WHERE EXISTS (SELECT id FROM T2 WHERE T2.id=T2.id)
# T1からレコードを取得して、T2と紐づけを行い、紐づけができたレコードのみをT1から取り出す
```

***) 最近のDBでは処理時間が改善されているため、INでもEXISTSでも処理時間は変わらない場合もある**

SQLチューニング事例6
（その他のありがちなチューニング例）

6-1. 実行されるSQLの回数を減らす

アプリケーションでは、SQLの実行毎にアプリケーションサーバーとDBサーバー間で通信が発生し時間がかかる。SQLを**複数回実行するよりも1回の実行で済ました方が効率よく処理できる**



6-2. SELECTの実行回数を減らす

以下の通りWHEREで毎回絞り込むのでなく、INで一回実行する

×同じSQLを複数回実行する

```
SELECT * FROM products  WHERE id = 100
SELECT * FROM products  WHERE id = 200
SELECT * FROM products  WHERE id = 300
```

○同じSQLを複数回実行する

```
SELECT * FROM products  WHERE id IN (100, 200, 300)
```

6-3. INSERTの実行回数を減らす

マルチインサートを用いると、1回のSQLで複数のレコードを格納することができる

×2回INSERT処理を行う

```
INSERT INTO projects(name, start_date, end_date) VALUES ('AI for
Marketing','2019-08-01','2019-12-31');

INSERT INTO projects(name, start_date, end_date) VALUES ('ML for Sales','2019-05-15','2019-11-20');
```

○1回のINSERTで2つのレコードを追加する

```
INSERT INTO
        projects(name, start_date, end_date)
VALUES
        ('AI for Marketing','2019-08-01','2019-12-31'),
        ('ML for Sales','2019-05-15','2019-11-20');
```

6-4. 大量のINSERTの際に、インデックスを削除してINSERT後にインデックスを追加する

インデックスが存在する場合にはINSERTを行う際、**インデックスを構築する処理が走り** 時間がかかる。何万件以上のINSERT処理の実行の場合、**いったんインデックスを削除して** からINSERTをして、**INSERT完了後にインデックス再作成すること** を検討する

インデックス削除
（DROP INDEX）

大量INSERT

インデックス再作成
（CREATE INDEX）

6-5. 大量INSERTの際、主キーやインデックスのない一時テーブルを作成してINSERTし、その後SELECT INSERTで本テーブルにINSERTする

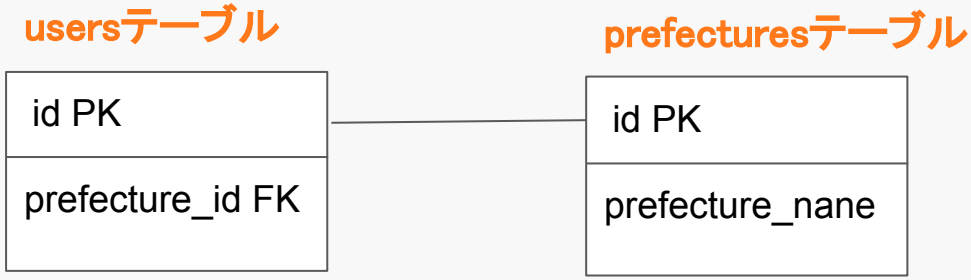
一時テーブルにINSERT

一時テーブルから
SELECTしてINSERT

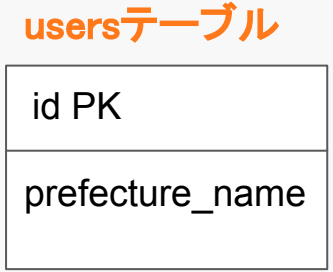
一時テーブルの中身を消す

6-6. あえてカラムを増やす（非正規化して）

結合処理は、処理時間がかかるため **あえて正規化を外してカラムを追加し** 、結合が発生しないようにする

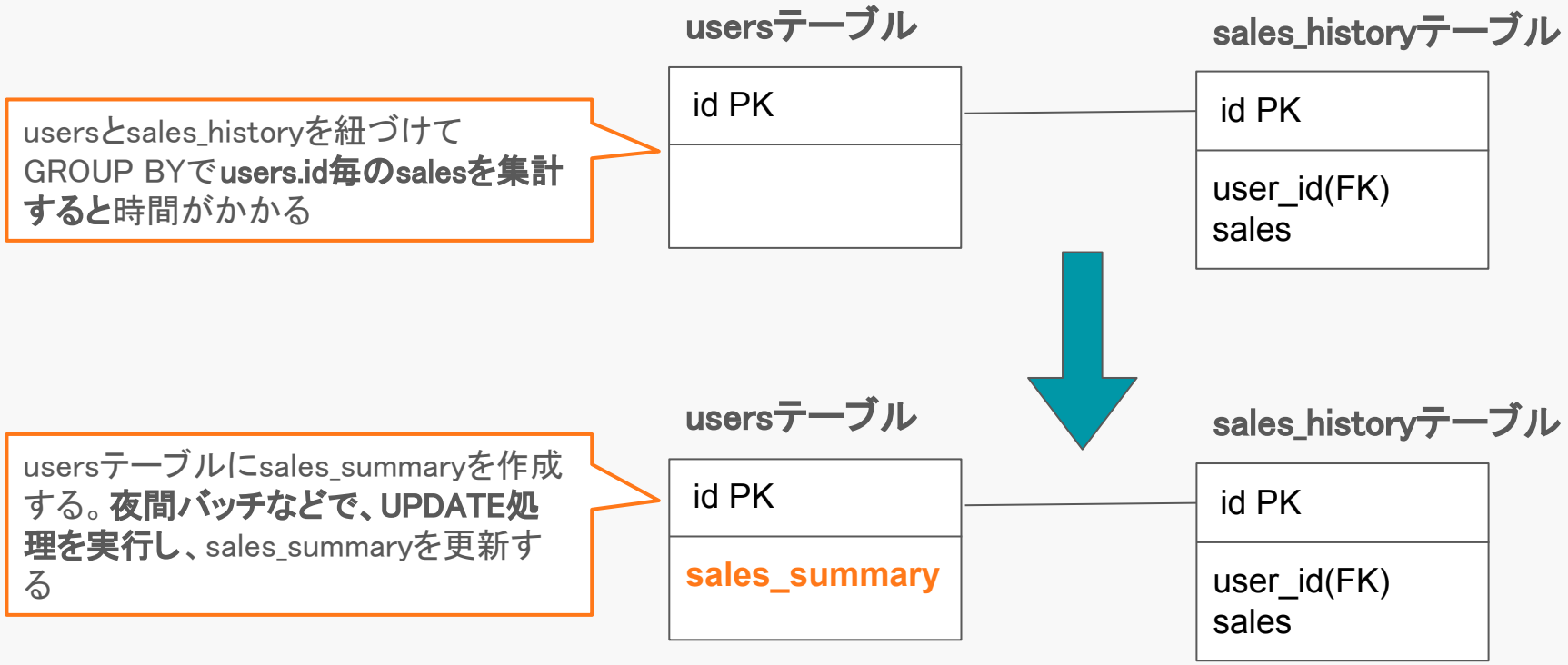


上のテーブル設計では、usersテーブルからprefecture_nameを取り出すには **usersテーブルとprefecturesテーブルを結合しなければならない**
usersテーブルにprefecture_nameを入れると、結合の必要がなくなる



6-7. 夜間バッチで実行結果を格納する

リアルタイムで実行結果が必要ない場合 、夜間バッチなどで実行結果をテーブルに格納して、確認できるようにする。集計処理などは、**処理コストが大きいため夜間バッチで実行結果を格納する**



6-8. ピーク時間をさける

SQLの実行にはDBへの負荷がかかるため、**ピーク時間を避けて実行する**とよい。例えば、ユーザーが利用する日中帯は避けて、夜間帯に重いSQLを実行するなどを行う

6-9. LIMITをつけて実行する

テーブルの中身を数件確認したい場合は、**LIMITをつけて実行すると**処理時間がかからない。ただしCOUNT(*)などの集計関数は、LIMITをつけても処理時間は短くならない

6-10. レコード全削除は、DELETEでなくTRUNCATEを用いる

DELETE処理は時間がかかるため、レコードを全削除する場合は **TRUNCATEを用いるとよい**

SQLチューニング例題

例題1

以下のSQLの性能上の問題点を上げて修正案を提示してください

```
SELECT
  *
FROM sales_summary
WHERE YEAR(sales_date) = “2021” AND MONTH(sales_date) = “12”;
```

問題点: カラムに対して関数を用いているためインデックスを利用できない
改善案: 関数を使わずに、同じ意味の構文にする

```
SELECT
  *
FROM sales_summary
WHERE
    sales_date BETWEEN “2021-12-01”
    AND “2021-12-31”;
```

例題2

以下のSQLの性能上の問題点を上げて修正案を提示してください(nameにインデックスが張られているとします)

```
SELECT
  *
FROM sales_history
WHERE RTRIM(name) = “product_a”
```

問題点: カラムに対して関数RTRIMを用いているためインデックスを利用できない
RTRIMは文字列の右側の空白を削除する関数
改善案: LIKEの**前方一致を用いた**検索を行いて絞込みを行い、そのあとRTRIMの検索をする

```
SELECT
    *
FROM
(SELECT * FROM products WHERE name LIKE ″product_a%″) AS tmp
WHERE RTRIM(name) = ″product_a″
```

例題3

以下のSQLの性能上の問題点を上げて修正案を提示してください

```
SELECT
  *
FROM users
WHERE first_name IN (SELECT first_name FROM employees WHERE age < 30)
AND
last_name IN (SELECT last_name FROM employees WHERE age < 30)
```

問題点: 副問い合わせを2回実行していて、絞込みも同じになっている
改善案: 副問い合わせを1回にして複数カラムに対する絞込みをする

```
SELECT
  *
FROM users
WHERE (first_name, last_name) IN (SELECT first_name, last_name FROM employees WHERE age < 30)
```

例題4

以下のSQLの性能を改善するためのインデックス作成案を提案してください

```
SELECT ct.first_name, ct.last_name, st.name, SUM(sh.sales_amount)
FROM
    customers AS ct
    INNER JOIN sales_history AS sh ON  ct.id = sh.customer_id
    INNER JOIN products AS pr ON  sh.product_id = pr.id
    INNER JOIN stores AS st ON  pr.store_id = st.id
WHERE
    ct.first_name = ″Olivia″ AND ct.last_name = ″Roach″
GROUP BY
    ct.first_name, ct.last_name, st.name;
```

改善案:

①: WHEREの絞込みに条件を追加（customersテーブルのfirst_nameとlast_nameにそれぞれ、または複合インデックス）
②: 各外部キー、主キーになければインデックスを作成