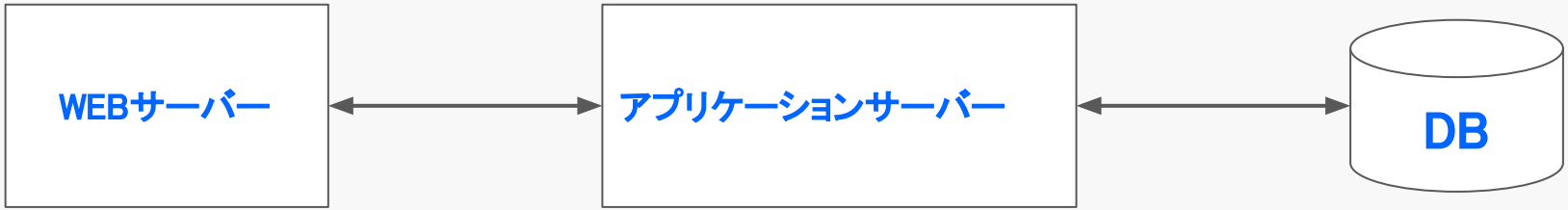
統計情報・実行計画とSQLチューニング

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

DBとパフォーマンスについて

システム開発において、パフォーマンス(処理時間)を速くすることは重要だが、このパフォーマンス向上の鍵を握るのがDBである。

- 運用を続けていくと**DBのデータ量が増加する**。SQLが悪いと、**DBのデータ量の増大に比例して処理時間が増える**
- **アプリケーションサーバー**はスケールアウトが容易だが、DBサーバはスケールアウトが難しい。処理の重いSQLが流れると、それだけ**リソースを使ってしまう。**



オンライン処理で実行されるSQLは、**1秒以下の処理速度**を目指す

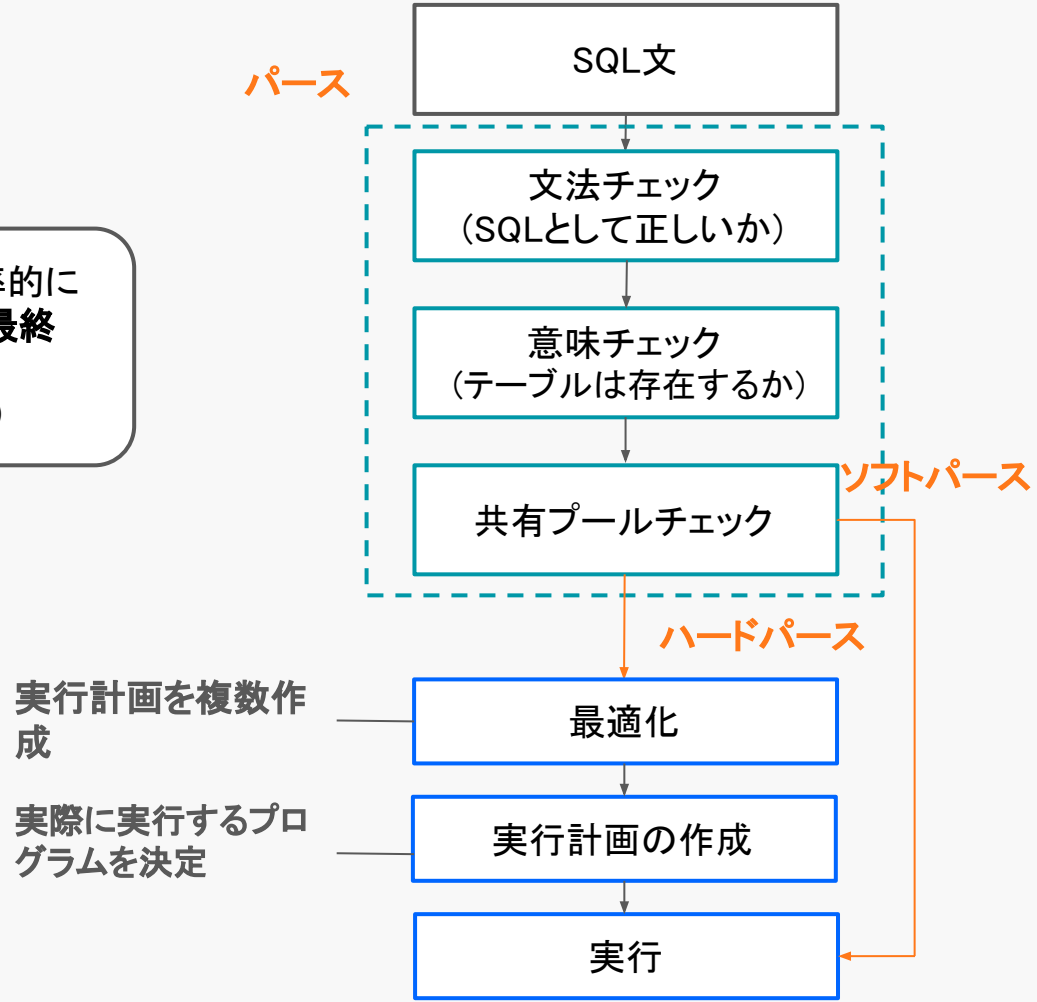
オプティマイザーと実行計画

SQLのチューニングを行う前に、SQLがどのように実行されるのかを知るのは大事です。まずは、SQL実行までの流れについて学習しましょう

SQL実行の流れ

SQLは右の順で実行される

SQLはそのまま実行されるのでなく、最も効率的に実行する**処理内容(バイナリのプログラム)を最終的に作成して実行される**
(喩え: 東京→大阪に行く、最短経路を決める)

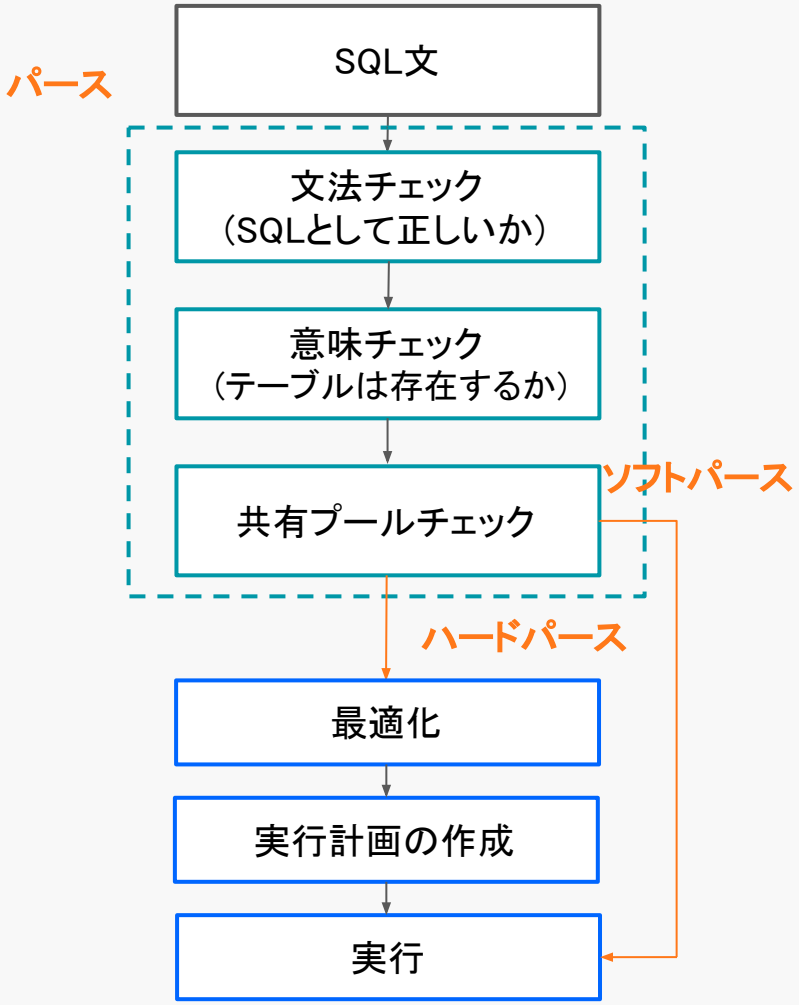


ソフトパース

パースの際に、**共有プールをチェックして** 以前に実行された実行計画が存在しないか確認する。もし、実行計画が以前に作成されており存在する場合は、**最適化、実行計画の作成を飛ばして** 、そのまま実行される。これを**ソフトパース**と言う。

ハードパース

SQLを初めて実行する場合は、**ハードパース**となり、実行計画作成分だけ時間がかかる



意図しないハードパースを避ける

同じSQLでも、DBを立ち上げて初めての実行では、**共有プール上に実行計画がなく**ハードパースされる
以下のようなSQLは、**同じSQLとして認識されないため**、似たSQLでもハードパースされてしまう

①SQL1
```
SELECT * FROM users WHERE name = ”Taro”;
```
②SQL2
```
SELECT * FROM users WHERE name = ”Jiro”;
```

バインド変数を使って実行

以下のように実行すると①,②のSQLは同じものとされて②はソフトパースされる

①SQL1
```
SET @name=”Taro”
SELECT * FROM users WHERE name = @name;
```
②SQL2
```
SET @name=”Jiro”
SELECT * FROM users WHERE name = @name;
```

アプリケーションのORマッパーのメソッドを使うと**自動的にバインド変数が活用される**。アプリケーションからSQLを直書きして実行すると、バインド変数を使わずに実行される

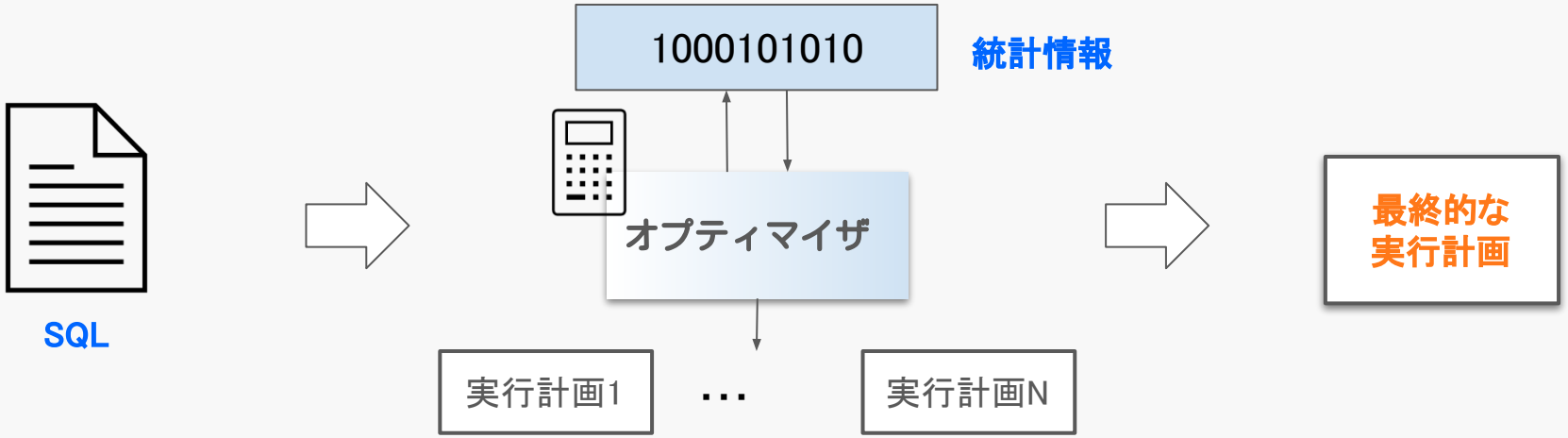
コストベースオプティマイザと統計情報

コストベースオプティマイザとは

DBの**オプティマイザ**が、SQLから最も効率的な処理の実行方法（**実行計画**）を作成する。効率的な処理方法を作成するには、**最新の統計情報が必要となる**。
コストベースオプティマイザでは、インデックスや、データ量などの情報を元に、複数の**実行計画のコスト**（効率のよさ）を計算し、**最もコストの低い実行計画が実行される**

対象となる統計情報の例

カラム・データタイプ、データ量、レコード件数、テーブルに存在するインデックス・制約、カーディナリティ（どれだけ値にばらつきがあるか） など



統計情報を集める

テーブルの統計情報は自動的に収集されるが、コマンドを用いて手動的に収集することもできる。
急なテーブルのデータ量の増加があった場合 は、手動で統計情報を更新すればよい。

統計情報を収集するコマンド

ANALYZE TABLE テーブル名

テーブル統計情報の一覧を確認するコマンド

SELECT * FROM mysql.innodb_table_stats;

インデックス統計情報の一覧を確認するコマンド

SELECT * FROM mysql.innodb_index_stats;

実行計画とは

SQLが結果を得るための、処理の具体的な内容と実行順序を記述したもの
EXPLAIN SQL文: SQLを実行せずに、実行計画だけを表示する
EXPLAIN ANALYZE SQL文: SQLを実行して、実行時間も含めた実行計画を表示する(Actual)

EXPLAIN SQL文

```
id|select_type|table        |partitions|type|possible_keys|key|key_len|ref|rows|filtered|Extra                 |
--+-----------+-----------+----------+----+-------------+---+-------+---+----+--------+----------------------
----------+
 1|SIMPLE     |employees|           |ALL|                |   |       |   |  15|    10.0|Using where          |
```

table: 出力行のテーブル名
type: 結合する型
rows: 調査される行の見積もり
filtered: フィルターされる行の割合

出力フォーマット一覧: https://dev.mysql.com/doc/refman/8.0/ja/explain-output.html
結合する型一覧: https://dev.mysql.com/doc/refman/8.0/ja/explain-output.html#explain-join-types

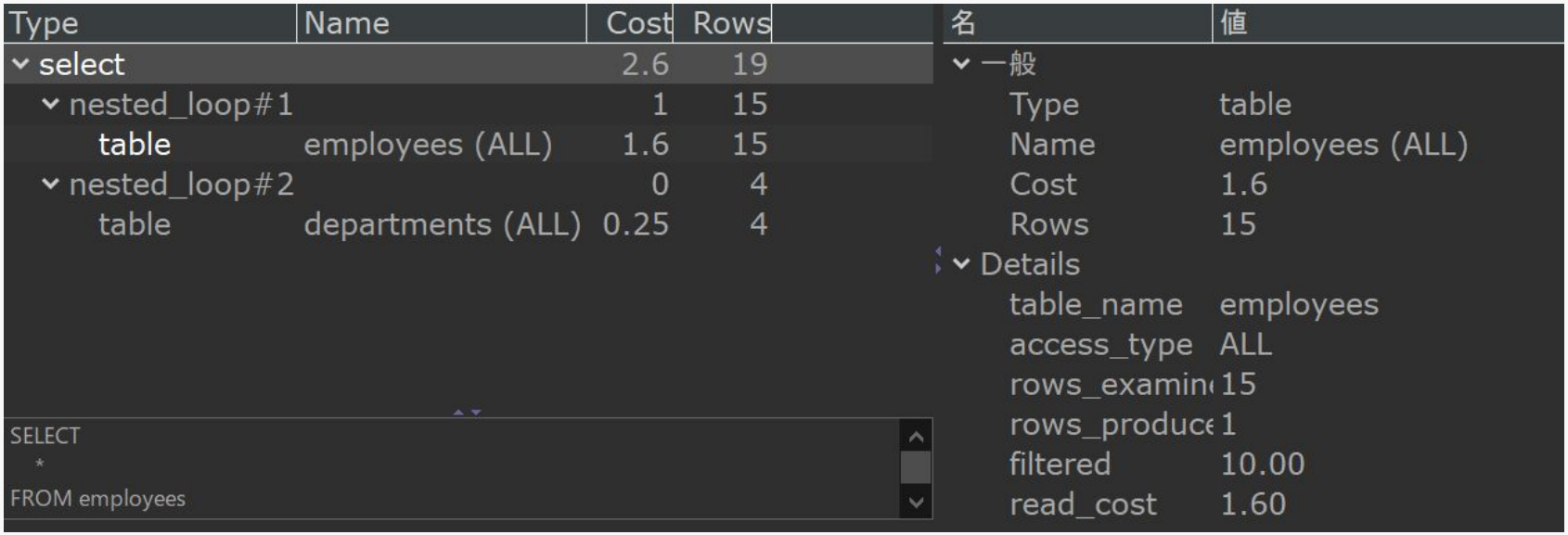
EXPLAIN ANALYZE SQL文

EXPLAIN ANALYZEは、SQLが実行されて、実行計画とともに**どれくらいの行数が取得されるのか、どれくらい時間に時間がかかったか**表示される。

```
-> Inner hash join (no condition)  (cost=2.60 rows=6) (actual time=0.189..0.303 rows=60 loops=1)
    -> Table scan on departments  (cost=0.43 rows=4) (actual time=0.020..0.032 rows=4 loops=1)
    -> Hash
        -> Filter: (employees.department_id = employees.department_id)  (cost=1.75 rows=2) (actual time=0.062..0.090 rows=15 loops=1)
            -> Table scan on employees  (cost=1.75 rows=15) (actual time=0.059..0.081 rows=15 loops=1)
```

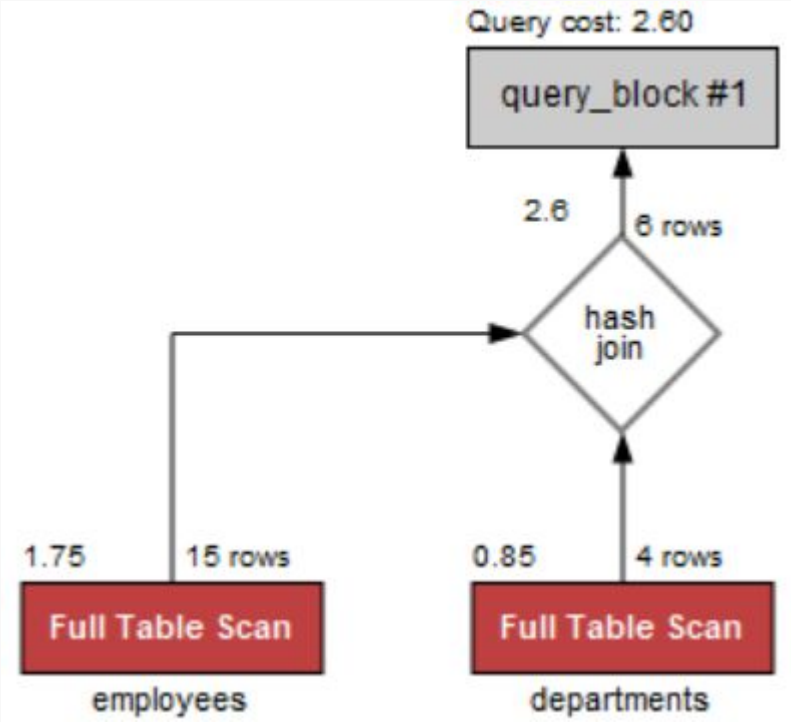
SQLクライアントを用いて実行計画をわかりやすく表示する

1. DBeaverで実行計画を表示する。「SQLエディタ→実行計画を説明する」(Ctrl+Shift+E)



SQLクライアントを用いて実行計画をわかりやすく表示する

2.　MySQL Workbenchで実行計画をグラフ表示する
(Query→Explain Current Statement(Ctrl+Alt+x))



実行計画で見るところ

-> **Nested loop inner join** (cost=147227.00 rows=326655) (**actual time**=0.095..742.538 **rows**=327068 loops=1)
    -> **Filter**: (od.item_id is not null) (cost=32897.75 rows=326655) (**actual time**=0.080..220.336 **rows**=327068 loops=1)
        -> **Table scan** on od (cost=32897.75 rows=326655) (**actual time**=0.079..168.919 **rows**=327068 loops=1)
    -> **Single-row index lookup on it using PRIMARY** (id=od.item_id) (cost=0.25 rows=1) (**actual time**=0.001..0.001 **rows**=1 loops=327068)

カーディナリティ: どれだけの行が各処理で取得されているか
実行時間(actual time): Actualの実行計画の場合実行時間が表示される
アクセスの方法: フルスキャンなのか、インデックススキャンなのか
JOINの方法: hash、sort-merge、Nested Loopなどのテーブル間紐づけの方法
JOINの順番: どの順番でテーブルが結合されるのか
パーティションが利用されているか: テーブルパーティションを利用すると絞り込む対象が少なくなる。

代表的なテーブルアクセスの方法（フルスキャン、インデックススキャン、インデックスレンジスキャン）

テーブルフルスキャン

- テーブルから**各行を全て確認して**データを取り出す
- 大量(約100万以上)のデータがテーブルに存在する場合、**非常に時間がかかる**

テーブルフルスキャンが利用される例

・テーブルから全件選択する

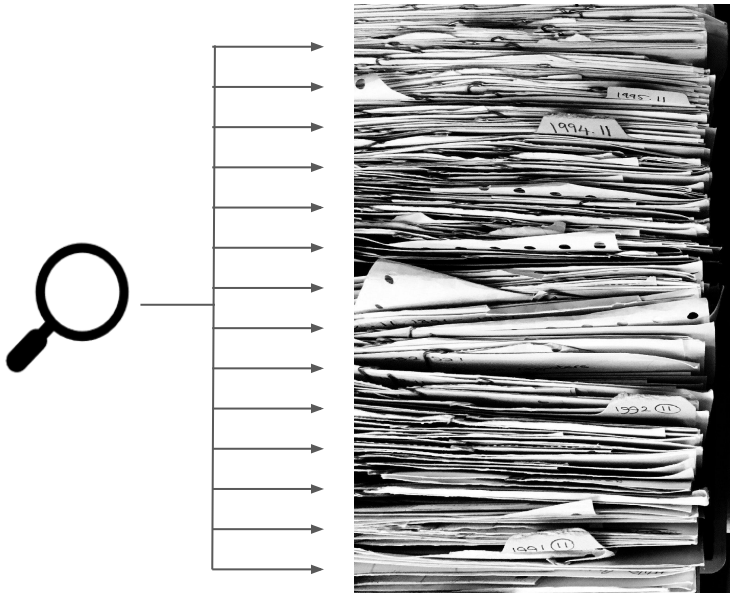
SELECT * FROM users

・インデックスのない列で絞り込む

SELECT * FROM users WHERE name=”〇〇”

・インデックスが上手く利用できていない場合

SELECT * FROM users WHERE name LIKE ”%〇〇%”



テーブル

インデックスユニークスキャン

- テーブル内のユニークな1件を **インデックスを用いて検索する**
- 1件見つかれば、探索が終了するため、**高速に処理が終わる**

インデックスユニークスキャンが利用される例

・主キー、ユニークキーで絞り込みをする

SELECT * FROM users WHERE id=10;

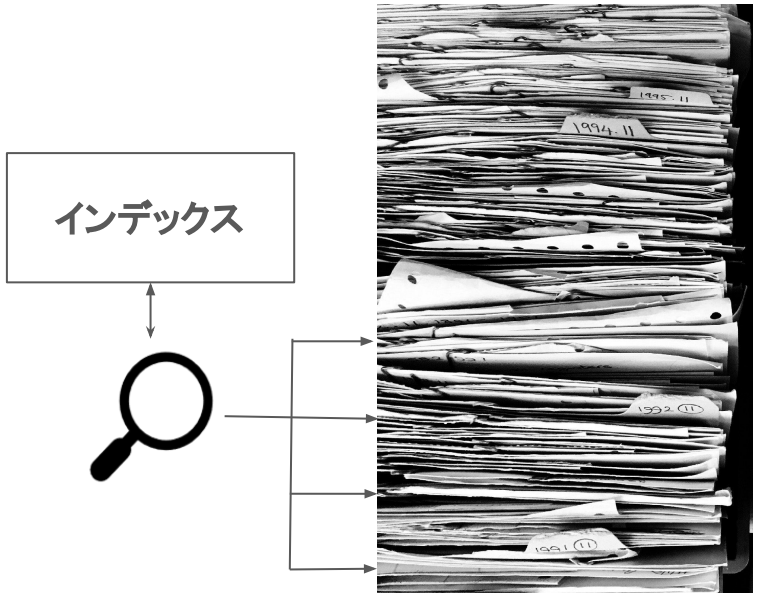
インデックスレンジスキャン

- インデックスを用いて、**複数の行を取得する際の検索方法**
- 対象となるレコード件数に応じて、処理時間が変わる

インデックスレンジスキャンが利用される例

・ユニークでないキーで絞り込みをする

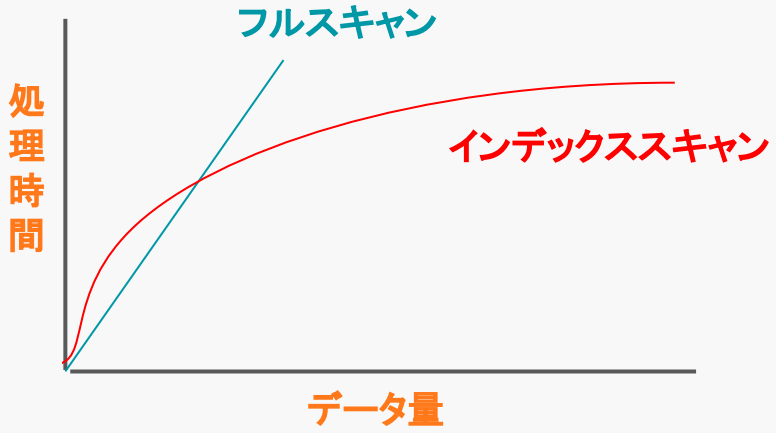
SELECT * FROM users WHERE age=50;



フルスキャンとインデックススキャンのどちらを使うべきか

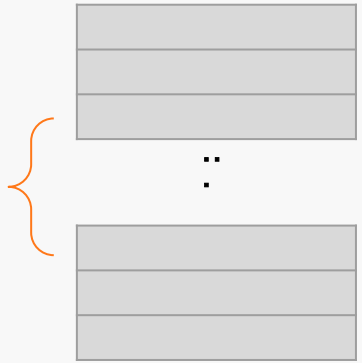
- インデックススキャンは、**データの一部しか読み込まないため処理時間を短くできる**
- フルスキャンは**データ量が増えれば処理時間が長くなるが** 、インデックススキャンはデータ量が増えても**一定のパフォーマンスで処理ができる**
- ただし絞りこめるデータの割合が、**全データ量と比較して少ない（あまり絞り込めない）場合** は、フルスキャンの方が速いことがある

1. データ量が少ない（約1万件以下）の場合、インデックスを使っても使わなくても処理時間はかわらない



2. 絞り込めるデータの割合に応じてフルスキャンとインデックススキャンのどちらが有効か決まる

全データの中から、どれだけのデータを取り出すのか



フルスキャンとインデックススキャンのどちらを使うのかの目安

絞り込まれる行の全体に占める割合	どちらのスキャンを使うと良いか
20%以上	フルスキャンを用いる
1-20%	ケースバイケース(一説では、15%以下はインデックススキャンの方が高速)
1%未満	インデックススキャンを使う

テーブル結合の方法
（ネステッドループ、ハッシュジョイン、ソートマージ）

テーブル結合について

INNER JOINなどで複数のテーブルを結合にする場合にも、複数のパターンがある。
テーブル結合の方法は、オ**プティマイザーが統計情報を元に適切なものを選択する** 。テーブル結合を強制的に変更するには、**ヒント句**[*)]を用いたりする

*) ただし、**記述通りの実行計画にならないことも多い**
(https://dev.mysql.com/doc/refman/8.0/ja/optimizer-hints.html)

ヒント句(itemsテーブルのインデックスをあえて使わないヒント)

```
SELECT  /*+ NO_INDEX(it) */ * FROM orders AS od
INNER JOIN items AS it
ON od.item_id = it.id;
```

代表的な結合方法

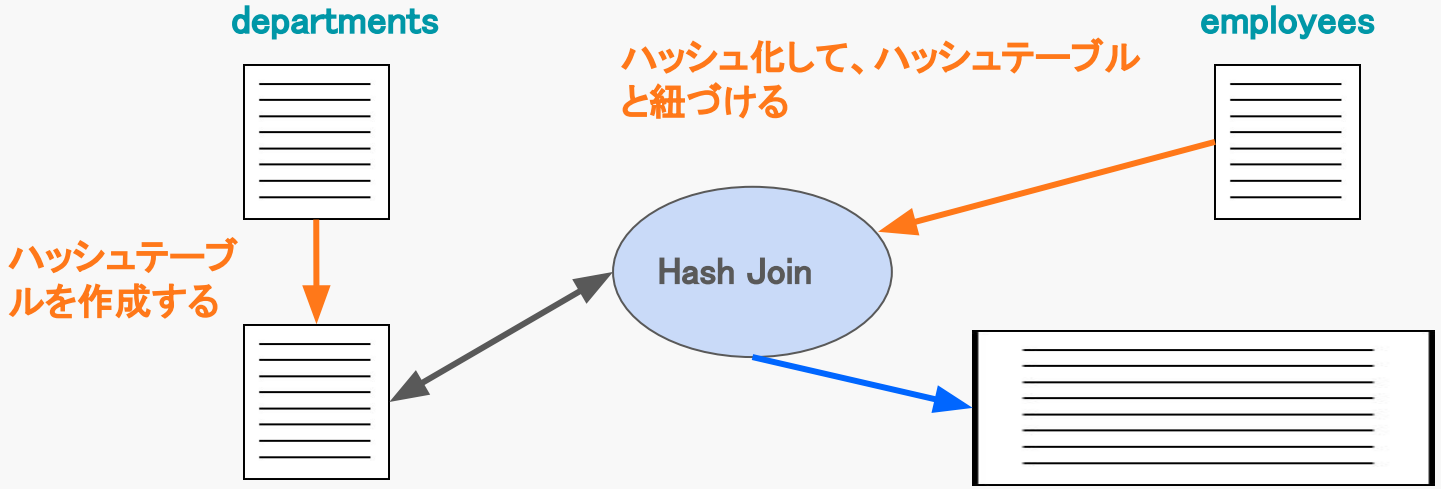
- **ネステッドループジョイン(Nested Loop JOIN)**
- **ハッシュジョイン(Hash JOIN)**
- **ソートマージジョイン(Sort Merge JOIN)**

ネステッドループジョイン(Nested Loop JOIN)

一方のテーブル(上の例では名前)を**ループさせて一行ずつデータを読み込み**、各行に結合する行を**他方のテーブル**(上の例では電話帳)**から探索して結合する**。
ループして探索するテーブルを**外部表(駆動表)**、突合するテーブルを**内部表**という

- 外部表から一行一行読み込んで、突合し、**外部表の最後までループする**
- **外部表のデータ量が少ない方が**ループ回数が減るため、パフォーマンスは良い
- 内部表の突合対象のカラムには、**インデックスがある方が**処理速度が速い可能性が高い

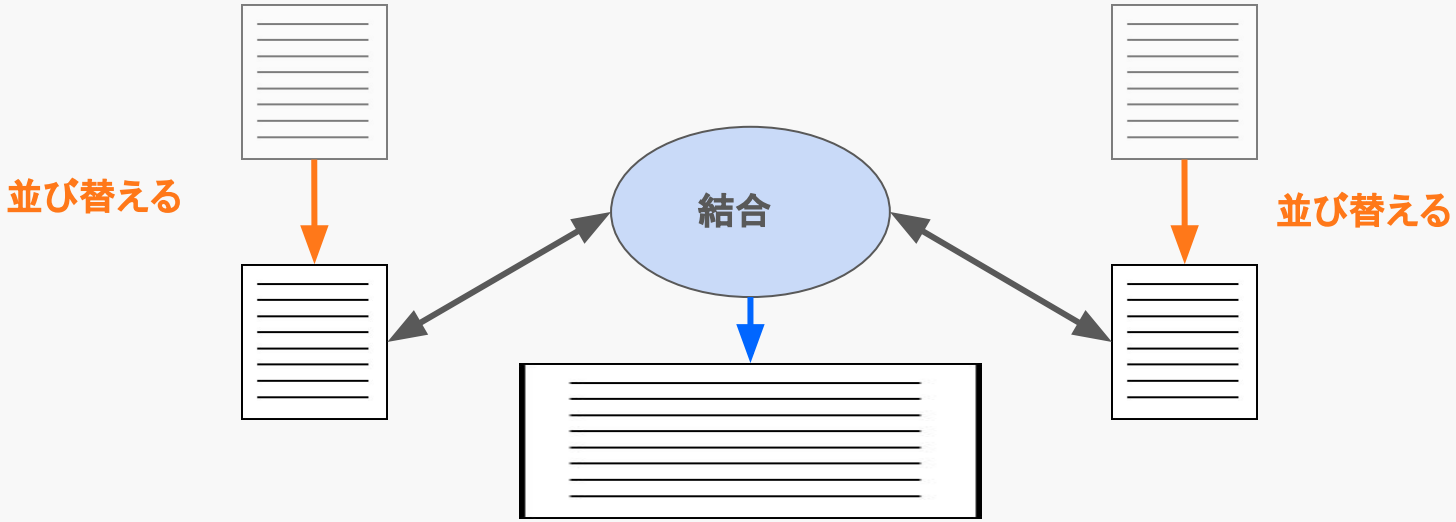
ハッシュジョイン(Hash JOIN)



小さい方のテーブルの結合対象カラムを**ハッシュ値に変換し**、テーブルとしてメモリ上に配置する。
大きい方のテーブルを**ハッシュ化していき**、テーブル同士を突合する

- **インデックスを必要としない**テーブル結合である
- 一時的にメモリ上にデータを配置するため、**メモリが必要**。サーバーのメモリが少ない場合、処理速度が大きく劣化することがある
- **= での比較でしか利用できない**

ソートマージジョイン(SORT MERGE JOIN)



=以外の比較で使用されることが多い(<, <=, >=, >など)

以下の2つのステップで実行される

1. **2つのテーブルを結合対象のカラムで並び替える**
2. **並び替えられたもの同士を比較して連結していく**

実行計画の見方

実行計画はより**深くに記述された処理から順に実行され 、同じ深さの処理は上から順に実行される**

実行順序

④ 下の2つ(②, ③)のネステッドループ(上のテーブル(od)が**外部表**、下のテーブル(it)が**内部表**)

② -> Nested loop inner join (cost=147227.00 rows=326655) (**actual time**=0.095..742.538 **rows**=327068 loops=1)

-> **Filter**: (od.item_id is not null) (cost=32897.75 rows=326655) (**actual time**=0.080..220.336 **rows**=327068 loops=1)

① -> Table scan on **od** (cost=32897.75 rows=326655) (**actual time**=0.079..168.919 **rows**=327068 loops=1)

-> Single-row index lookup on **it** using PRIMARY (id=od.item_id) (cost=0.25 rows=1) (**actual time**=0.001..0.001 **rows**=1

③ **loops=327068**)

1行あたりの実行時間 全実行時間

フルスキャン

取得された行数

主キーを用いたインデックススキャン

実行された回数

SQLチューニングの最も基本的な内容

1.　SELECTで取得するカラムは必要なものだけにする

カラムを余分に取得するとデータ量が増えその分 **データの転送に時間がかかる** ため、必要なカラムだけを記述するようにする

```
SELECT
    *
FROM
    employees;
```

```
SELECT
    id, name, age, company_name
FROM
    employees;
```

2. テーブル結合時には、別名を省略せずにつける

テーブルに別名をつけずに実行しても、エラーは発生せずに実行できる。ただし、パース（SQLの解釈）に時間がかかるため、**別名をつけて**SQLを解釈しやすくするとよい。

```
SELECT
    *
FROM employees
INNER JOIN departments
ON department_id = id;
```

```
SELECT
    *
FROM employees AS emp
INNER JOIN departments AS dt
ON emp.department_id = dt.id;
```

チューニングの流れ

チューニングの流れ

SQLチューニングの際に、むやみにインデックスを追加すると **かえって処理速度が遅くなることがある。** 実際の現場では、以下の流れでチューニングを行いましょう

1. 実際に運用をしていて **処理速度の遅いSQLを出力して** 、なぜ遅いのか実行計画を調べる
2. 実行計画を見て、**SQLの改善案を作成する**
3. **改善案に効果があるのか、他のSQLに影響はないか** を確認する
4. 改善案を適用して、実際に改善されたか確認する

リリースしていないシステムの場合

リリースしていないシステムの場合は実際に処理速度の遅いSQLがどれかわからない。この場合、DBに **十分なデータ量を積んで(想定されるMAXのデータ量)** システムの処理を実行し、遅いSQLを見つけるとよい。