MySQL SQLオプティマイザのコスト計算アルゴリズム

InnoDB Deep Talk #1
2012/03/10 平塚 貞夫

自己紹介

- DBエンジニアやってます。専門はOracleとMySQL。
  - システムインテグレータで主にRDBMSのトラブル対応をしています。
  - 仕事の割合はOracle：MySQL＝8：2ぐらいです。

- Twitter：@sh2nd
- はてな：id:sh2

- 写真は実家で飼っているミニチュアダックスのオス、アトムです。

本日のお題

- MySQLのステータス変数にLast_query_costというものがありまして、これを参照すると直前に実行したSQLのコストを確認することができるようになっています。この値は実績値ではなく、SQLオプティマイザがSQL実行計画を選択するために算出した推定値を表しています。

```
mysql> SELECT * FROM item WHERE i_name = 'NFOHP7ywvB';
+------+-----------+---------+
| i_id | i_name    | i_price |
+------+-----------+---------+
|   50 | NFOHP7ywvB |   10161 |
+------+-----------+---------+

mysql> SHOW LOCAL STATUS like 'Last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 20365.399000 |
+-----------------+-------------+
```

- 本日は、みなさんにこれを計算できるようになっていただきます。

- MySQL 5.6.4-m7を用いて説明していきます。

実践ハイパフォーマンスMySQLについて

- 実践ハイパフォーマンスMySQL 第2版 170ページに以下の記述があります。

> MySQLはコストベースのオプティマイザを使用するため、さまざまな実行プランのコストを見積もり、最も安価なものを選択しようとする。コストの単位は、ランダムな4KBデータページの読み取りである。オプティマイザが見積もったクエリのコストを確認するには、クエリを実行して、Last_query_costセッション変数を調べればよい。

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
|     5462 |
+----------+

mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 1040.599000 |
+-----------------+-------------+
```

> この結果から、オプティマイザがクエリを実行するために1,040ページのランダムデータを読み取る必要があると見積もっていることがわかる。

- 残念ながら、この説明は誤りです。忘れてください。

テストデータ

- 以下の商品テーブルを使って調べていきます。

```
CREATE TABLE `item` (
  `i_id` int(11) NOT NULL,
  `i_name` varchar(100) DEFAULT NULL,
  `i_price` int(11) DEFAULT NULL,
  PRIMARY KEY (`i_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

- 10万レコード用意しました。商品名と商品価格はランダムな値になっています。

```
+--------+--------------------+---------+
| i_id   | i_name             | i_price |
+--------+--------------------+---------+
|      1 | Y7I8zlqJ3ZuYnBasy  |   57231 |
|      2 | wGDEP8MC3Gs        |   89587 |
|      3 | DvEFlHVcWLP3Zp     |   67530 |
|      4 | uoVnfCFOOmtg       |   18176 |
|      5 | tNTodSMwd3F13Np1   |   98690 |
～
| 100000 | cA7dnrmM32         |   14062 |
+--------+--------------------+---------+
```

単一テーブルのフルスキャン

単一テーブルのフルスキャン (1/4)

- まずテーブルをフルスキャンしたときのコストを確認していきましょう。

- 商品名カラムにインデックスがないので、商品名を指定したSQLはテーブルフルスキャンになります。

```
mysql> EXPLAIN SELECT * FROM item WHERE i_name = 'NFOHP7ywvB';
+----+-------------+-------+------+---------------+------+---------+------+--------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows   | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+--------+-------------+
|  1 | SIMPLE      | item  | ALL  | NULL          | NULL | NULL    | NULL | 100382 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+--------+-------------+
```

- このときのコストは20,365になりました。

```
mysql> SHOW LOCAL STATUS like 'Last_query_cost';
+-----------------+--------------+
| Variable_name   | Value        |
+-----------------+--------------+
| Last_query_cost | 20365.399000 |
+-----------------+--------------+
```

単一テーブルのフルスキャン (2/4)

- SQLのコストは以下の二つのパラメータから計算されます。
  - found_records
    読み取られるレコード数の推定値。
  - read_time
    ディスクアクセス回数の推定値。

- フルスキャンの場合、found_recordsにはテーブルの統計情報が用いられます。

```
mysql> SHOW TABLE STATUS LIKE 'item'¥G
*************************** 1. row ***************************
           Name: item
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 100382 ★これがそのままfound_recordsになります
 Avg_row_length: 47
    Data_length: 4734976
```

※InnoDBのテーブル統計情報はサンプリングで求めているため、正確に10万レコードにはなりません。

単一テーブルのフルスキャン (3/4)

- read_timeはフルスキャンの場合、scan_time()という関数を通してストレージエンジンに問い合わせた結果が用いられます。

- InnoDBの場合、scan_time()はテーブルのページ数を返します。

```
UNIV_INTERN double ha_innobase::scan_time() {
    return((double) (prebuilt->table->stat_clustered_index_size));
}
```

- これはテーブルステータスのData_lengthを16KBで割り算した値と同じです。

```
mysql> SHOW TABLE STATUS LIKE 'item'¥G
*************************** 1. row ***************************
           Name: item
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 100131
 Avg_row_length: 47
    Data_length: 4734976 ★4,734,976÷16,384＝289がread_timeになります。
```

単一テーブルのフルスキャン (4/4)

- ここまででfound_recordsとread_timeが求められました。
  - found_records：100,382
  - read_time：289

- SQLのコストは、found_recordsとread_timeから以下の式で算出されます。

```
コスト＝read_time＋found_records×ROW_EVALUATE_COST
　　＝289＋100,382×0.20
　　＝20,365.4
```

- ROW_EVALUATE_COSTは0.20で固定です。

```
#define ROW_EVALUATE_COST 0.20
```

- フルスキャンにおけるコスト計算は以上です。

- 計算アルゴリズムはそれほど難しくありませんが、何の根拠があって単位の異なるfound_recordsとread_timeを足しているのか、0.20という定数にどんな意図があるのかなど疑問点がいくつか出てくると思います。ちなみに私もさっぱり分かりませんので聞かれても困ります。とりあえず先へ進みましょう。

単一テーブルのユニークスキャン

単一テーブルのユニークスキャン (1/2)

- 次は、ユニークインデックスを用いて1レコードを引き当てたときのコストです。

```
mysql> EXPLAIN SELECT * FROM item WHERE i_id = 20000;
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | item  | const | PRIMARY       | PRIMARY | 4       | const |    1 |       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
```

- このときfound_recordsとread_timeは以下の値になります。
  - found_records：1
  - read_time：1

- SQLオプティマイザは、検索条件にユニークインデックスの等価条件が含まれていることが分かると必ずこのインデックスを用いるSQL実行計画を選択します。他の選択肢については探索自体が行われません。このときfound_records、read_timeには統計情報から得られた値ではなく、固定値として1が入るようになっています。

- つまり、主キー検索の場合SQLオプティマイザは何も考えないので、速いです。

単一テーブルのユニークスキャン (2/2)

- SQLのコストは以下のように表示されます。

```
mysql> SHOW LOCAL STATUS like 'Last_query_cost';
+-----------------+----------+
| Variable_name   | Value    |
+-----------------+----------+
| Last_query_cost | 0.000000 |
+-----------------+----------+
```

- なぜかゼロになっていますが、調べたところこれはMySQLの不具合であることが分かりました。開発元にバグ報告済みです。
  - MySQL Bugs: #64567:
    Last_query_cost is not updated when executing an unique key lookup
    http://bugs.mysql.com/bug.php?id=64567

- どうやら、内部処理をショートカットしすぎたためにステータス変数の更新までスキップしてしまったようです。

- 実際のコストは1.0となります。前述の式に当てはめると1.2になりますが、ユニークスキャンのコストは現状1.0でハードコーディングされています。

単一テーブルのレンジスキャン

範囲検索における読み取りレコード数の推定

- 次は、インデックスの張られたカラムに対して範囲検索をしたときのコストです。いくつか例を見てみましょう。

```
mysql> EXPLAIN SELECT * FROM item WHERE i_id BETWEEN 10001 AND 10100;
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra    |
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------+
|  1 | SIMPLE      | item  | range | PRIMARY       | PRIMARY | 4       | NULL |  100 | Using ... |
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------+

mysql> EXPLAIN SELECT * FROM item WHERE i_id BETWEEN 10001 AND 12000;
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra    |
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------+
|  1 | SIMPLE      | item  | range | PRIMARY       | PRIMARY | 4       | NULL | 1999 | Using ... |
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------+

mysql> EXPLAIN SELECT * FROM item WHERE i_id BETWEEN 10001 AND 20000;
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows  | Extra    |
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------+
|  1 | SIMPLE      | item  | range | PRIMARY       | PRIMARY | 4       | NULL | 20776 | Using ... |
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------+
```

- 最後の例を除き、rowsに出力される読み取りレコード数の推定値がかなり正確であることが分かると思います。まずはこの仕組みを確認していきます。

他のRDBMSの場合

- Oracle Databaseの場合、このような範囲検索に対してはヒストグラム統計を用いて読み取りレコード数を推定していきます。Oracle Databaseでは二種類のヒストグラムが使い分けられています。
  - 頻度ヒストグラム
    カラム値と該当レコード数を組にして格納します。カラム値の種類数が少ない場合に用いられます。
  - 高さ調整済みヒストグラム
    バケット番号とそのバケットに格納されるカラム値の範囲を組にして格納します。
    カラム値の種類数が多い場合に用いられます。
    例：No.1【1～100】No.2【101～101】No.3【102～500】No.4【501～1000】
    ⇒ カラム値＝101で検索するとバケット一つ、全体の25%にヒットすると推定。
    ⇒ カラム値＞150で検索するとバケット二つ、全体の50%にヒットすると推定。

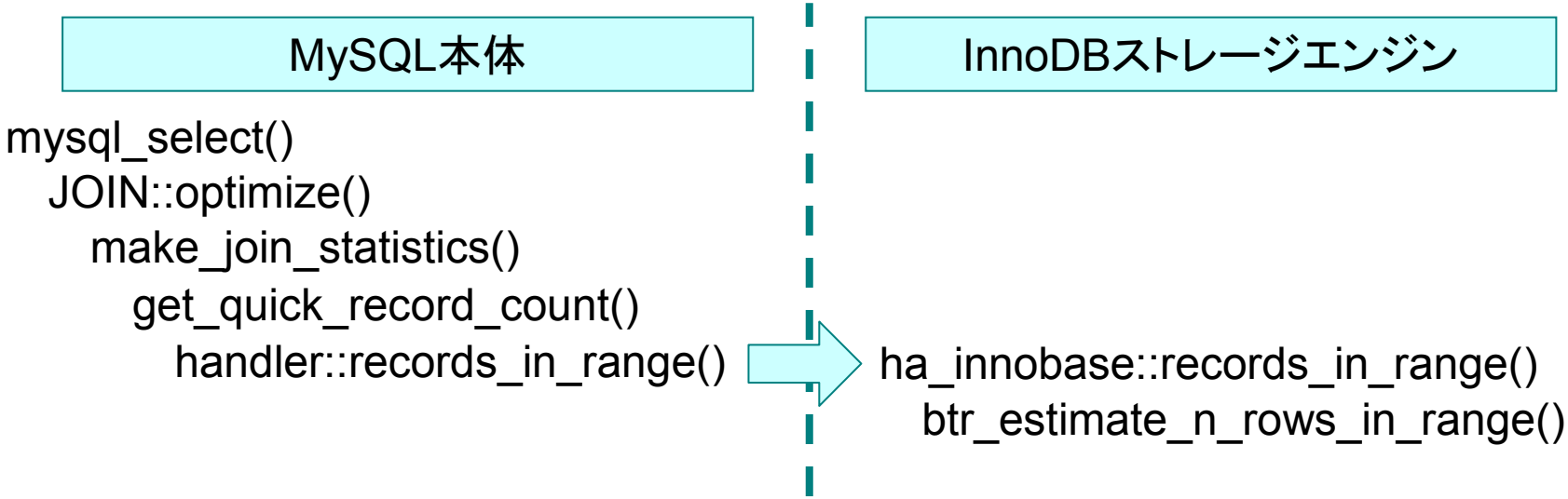
- PostgreSQLもOracle Databaseと同様にヒストグラム統計を用いています。
  PostgreSQLの場合はCompressed Histogramという、Oracle Databaseにおける頻度ヒストグラムと高さ調整済みヒストグラムを組み合わせた形のヒストグラムを利用しています。

- MySQLにはヒストグラムはありません。

レンジ分析

- 範囲検索における読み取りレコード数の推定は、MySQL本体ではなく各ストレージエンジンに任されています。この処理のことをレンジ分析(Range Analysis)といいます。

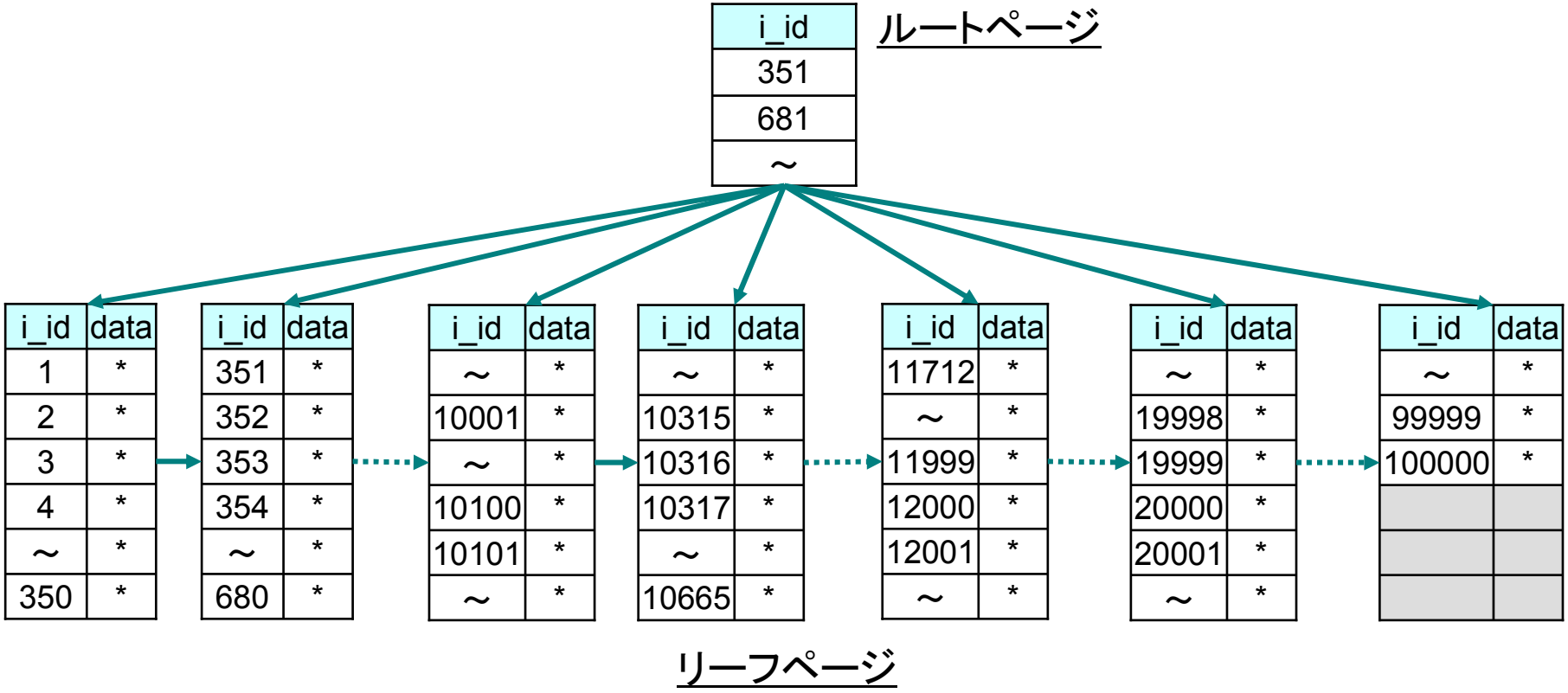
- MySQLはストレージエンジンAPIのrecords_in_range()を呼び出すことで、ストレージエンジンに読み取りレコード数の見積もりを依頼します。



- 次のページから、InnoDBのレンジ分析アルゴリズムを見ていきます。

InnoDBのレンジ分析 (1/9)

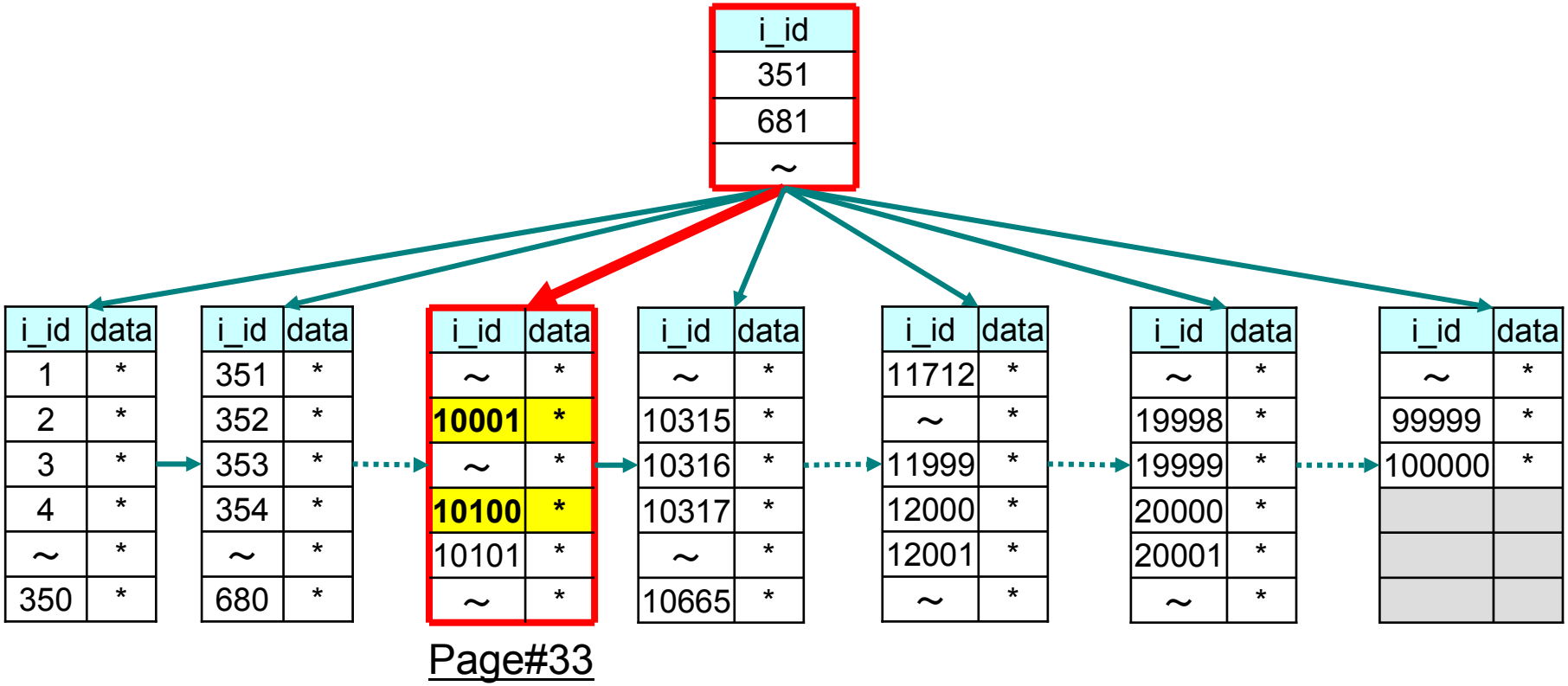
- インデックスのB+ツリー構造を以下に示します。それぞれの箱はInnoDBがデータを取り扱う単位である、16KBのInnoDBページを表しています。ツリー構造の頂点のページをルートページ、末端のページをリーフページと呼びます。実際には3段以上の構造になることもあります。



InnoDBのレンジ分析 (2/9)

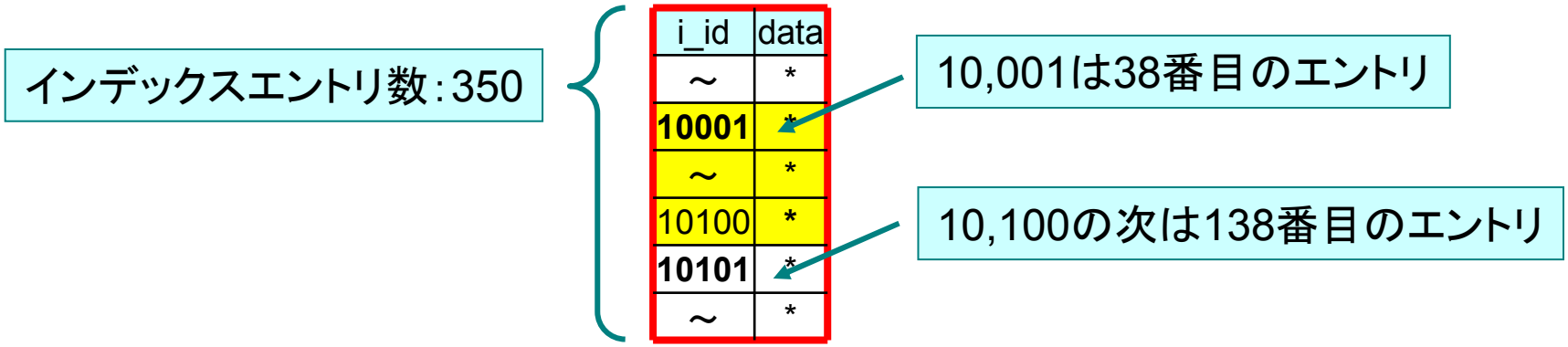
- InnoDBは範囲検索の下限値と上限値を受け取ると、最初にそれらが格納されているリーフページを読み取ってしまいます。つまり、見積もるぐらいだったら実際に数えてしまえという…。

- 最初のi_id BETWEEN 10001 AND 10100という例では、以下のようにどちらの値も33番のリーフページに格納されていました。



InnoDBのレンジ分析 (3/9)

- リーフページを読み取ることで、以下の情報を取得することができます。
  - そのページに含まれるインデックスエントリの数
  - それぞれのインデックスエントリが、ページ内で何番目の位置にあるのか
    (上限値がその値を含む条件のときは、実際には次のエントリを参照します)



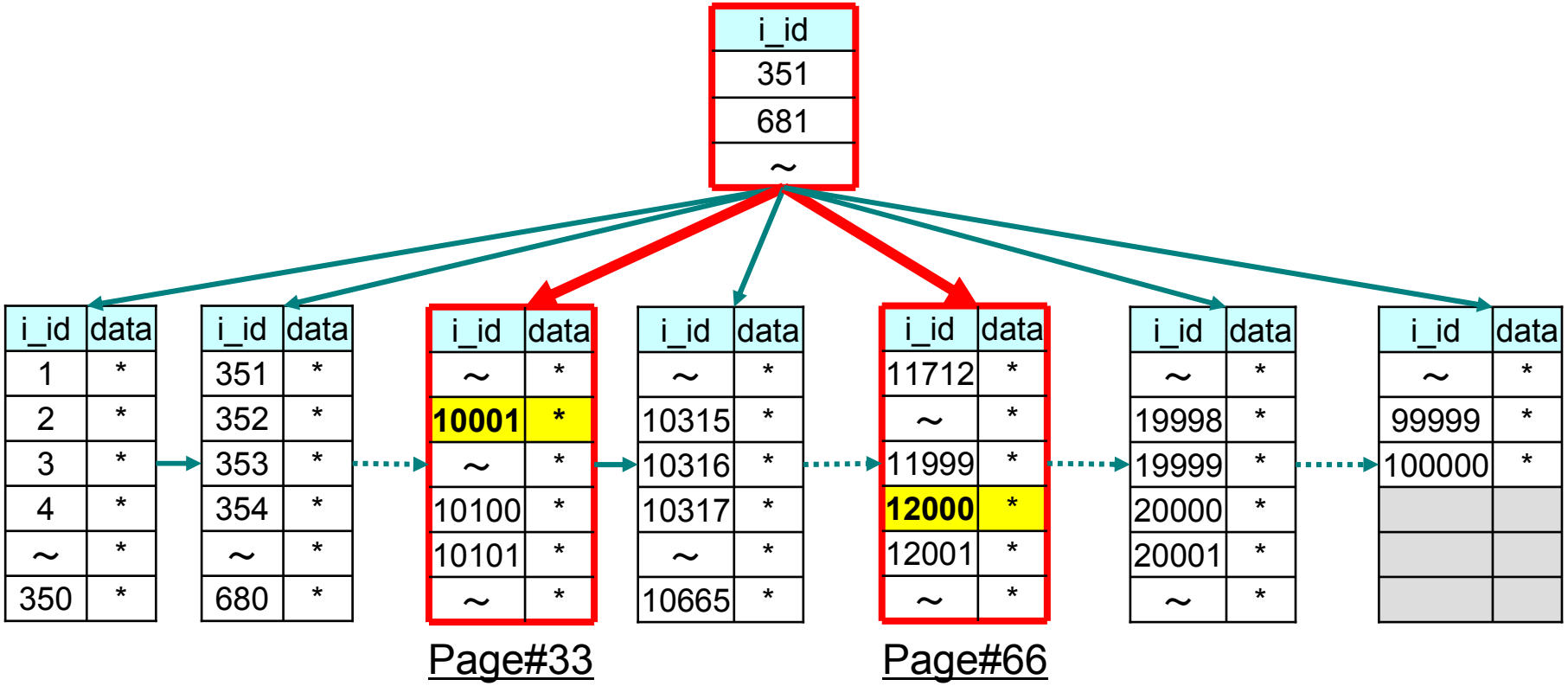
- 下限値、上限値におけるインデックスエントリの番号をnth_rec_1、nth_rec_2とすると、読み取りレコード数の推定値は以下の式で表されます。前ページの例のように下限値、上限値の双方が同じリーフページに格納されている場合、この式は正確なレコード数を返します。

読み取りレコード数の推定値＝nth_rec_2－nth_rec_1
＝138－38
＝100

InnoDBのレンジ分析 (4/9)

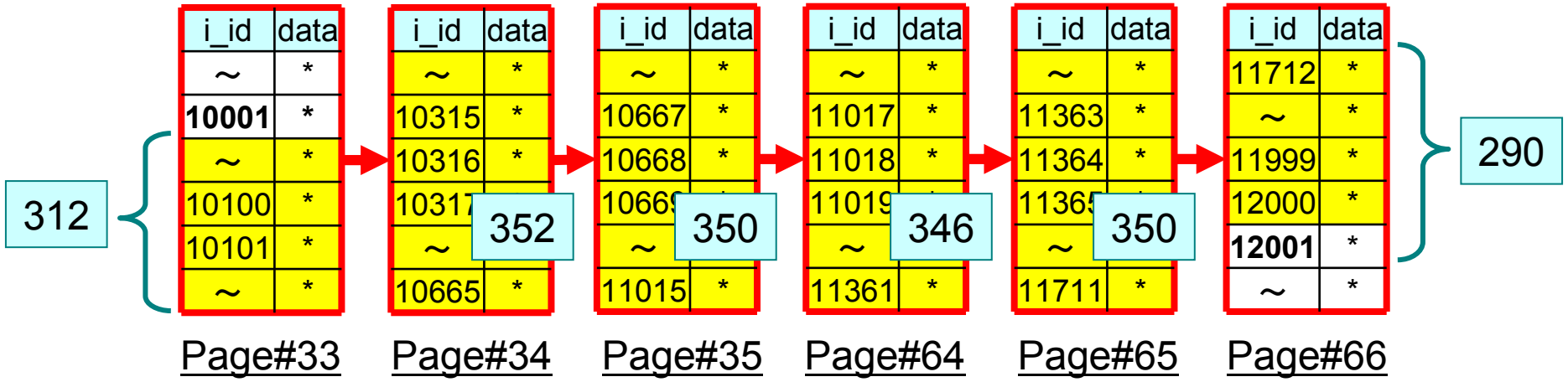
- 次に、i_id BETWEEN 10001 AND 12000の例を見てみましょう。

- この例では上限値のインデックスエントリが下限値とは別の、少し離れたリーフページに格納されています。



InnoDBのレンジ分析 (5/9)

- 下限値と上限値が異なるリーフページに格納されている場合、読み取りレコード数を推定するにはその間に挟まっているリーフページを考慮する必要があります。InnoDBはINDEX RANGE SCANを行ってこれらを実際に読み取っていきます。



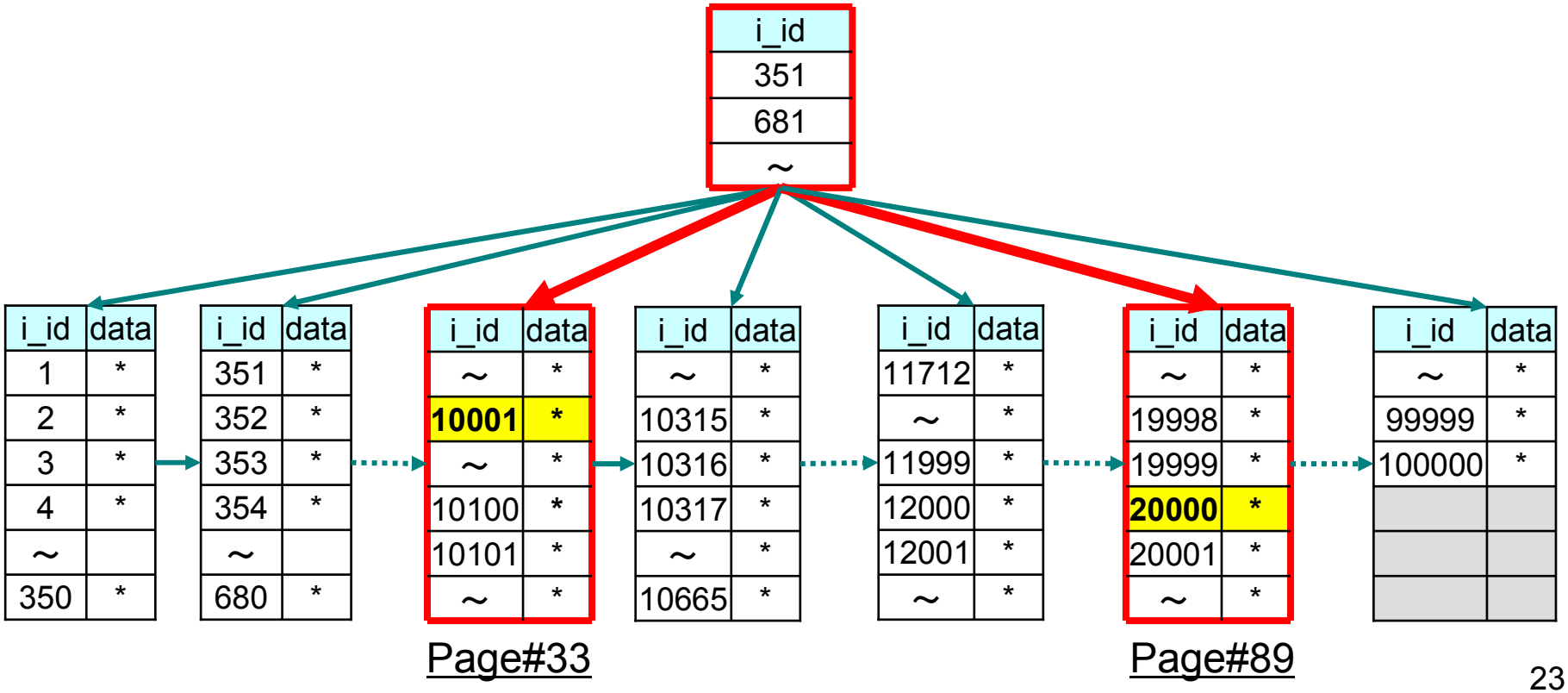
- リーフページに含まれるインデックスエントリの数をn_recs_Nとすると、読み取りレコード数の推定値は以下の式で表されます。この式も先ほどの例と同様に正確なレコード数を返すとされています。実際に試してみると1足りないのですが、下限値が格納されているリーフページについて計算誤りがあるのではないかと私は考えています。

読み取りレコード数の推定値＝(n_recs_1－nth_rec1)＋Σ(n_recs_middle)＋(nth_rec_2 - 1)
＝312＋(352＋350＋346＋350)＋(290－1)
＝1,999

InnoDBのレンジ分析 (6/9)

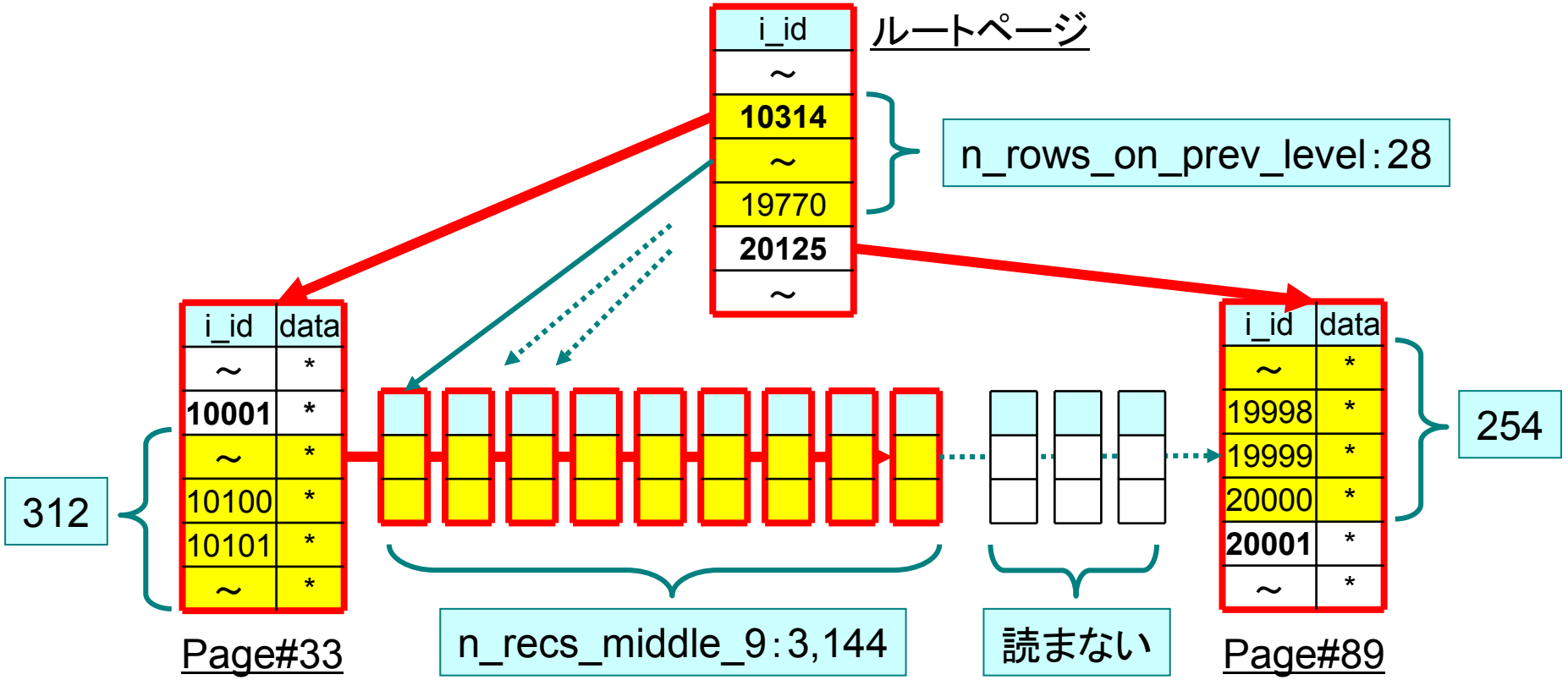
- 次はi_id BETWEEN 10001 AND 20000という例です。この例では読み取りレコード数の推定値が実際の値から大きくずれていることが分かります。

```
mysql> EXPLAIN SELECT * FROM item WHERE i_id BETWEEN 10001 AND 20000;
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows  | Extra    |
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------+
|  1 | SIMPLE      | item  | range | PRIMARY       | PRIMARY | 4       | NULL | 20776 | Using ... |
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------+
```



InnoDBのレンジ分析 (7/9)

- このときもINDEX RANGE SCANを行ってリーフページを実際に読み取るという方針は変わりません。しかし、リーフページが大量にある場合InnoDBは読み取りを途中で打ち切るようになっている点が異なります。現在の実装では、InnoDBは中間のリーフページを9ページ目まで読み取るようになっています。



InnoDBのレンジ分析 (8/9)

- 読み取りを途中で打ち切った場合は、ルートページの情報も見積もりに用いられます。ルートページにおいて下限値へのポインタと上限値へのポインタの間にあるエントリ数をn_rows_on_prev_levelとすると、これは検索範囲内にあるリーフページの数を表していることになります。

- 中間のリーフページに含まれるインデックスエントリの数をn_recs_middle_9として、読み取りレコード数の推定値は以下の式で表されます。

読み取りレコード数の推定値
＝(検索範囲内のリーフページ数)×(リーフページあたりの推定レコード数)×2
＝n_rows_on_prev_level×((n_recs_1－nth_rec1)＋n_recs_middle_9＋(nth_rec_2 - 1))÷10×2
＝28×(312＋3,144＋254)÷10×2
＝20,776

- この式には定数が二つあります。
  - 10:読み込んだリーフページの数です。実際には11ですが(1＋9＋1)、下限と上限はそれぞれ0.5ページ扱いになっていると思われます。
  - 2:補正係数です。「このアルゴリズムでは推定値が実際の値より少なくなることが多いため、2倍にした」とソースコードのコメントに書いてありました。

InnoDBのレンジ分析 (9/9)

- 最後にもう一つ補正が入ります。ここまでの計算で得られた読み取りレコード数の推定値がテーブルのレコード数の半分を超えた場合、推定値はテーブルのレコード数の半分に補正されます。

```
mysql> EXPLAIN SELECT * FROM item WHERE i_id BETWEEN 1 AND 100000;
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows  | Extra
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------
|  1 | SIMPLE      | item  | range | PRIMARY       | PRIMARY | 4       | NULL | 50137 | Using ...
+----+-------------+-------+-------+---------------+---------+---------+------+-------+----------
```

- そろそろ疲れてきたと思いますが、あと少しなので頑張りましょう。

単一テーブルのレンジスキャン (1/2)

- ここまででfound_recordsが求められました。最後の例を用いて説明を続けます。
  - found_records：20,776

- 次はread_timeを求めます。フルスキャンの場合、このパラメータにはscan_time()という関数を通してテーブルのページ数289が格納されていました。レンジスキャンの場合は、read_time()という関数を通して以下の値が格納されます。

read_time
＝1＋(読み取りレコード数の推定値)÷(レコード数の上限値)×(テーブルのページ数) … (A)
＋(読み取りレコード数の推定値)×ROW_EVALUATE_COST＋0.01 … (B)
＝1＋20,776÷324,290×289＋20,776×0.20＋0.01
＝19.5＋4,155.2
＝4,174.7

- (A)の部分はフルスキャンのときと同じ考え方で、レコードの選択率を加味した上で読み取りページ数を求めています。レコード数の上限値というのは、平均レコード長ではなく最小レコード長でデータを敷き詰めたときに格納できる最大のレコード数のことを表しています。最初の定数1はMySQL 5.6のMulti-Range Readという新機能を加味した値のようですが、MySQL 5.6.4-m7の時点では特に意味のある数値にはなっていないように見受けられます。

単一テーブルのレンジスキャン (2/2)

- (B)の部分ですが、これはフルスキャンの場合にSQLのコストを算出するところで使われていた式と同じです。この式がread_timeを求める時点で登場しているのは、本来は誤りではないかと私は考えています。結果としてread_timeの値はフルスキャンにおける289に対し4,155.2と大きく増えてしまっています。

- 最終的なSQLのコストは以下の式で算出されます。この式はフルスキャンの場合と同じです。

```
コスト＝read_time＋found_records×ROW_EVALUATE_COST
      ＝4,174.7＋20,776×0.20
      ＝8,329.9
```

```
mysql> SHOW LOCAL STATUS LIKE 'Last_query_cost';
+-----------------+------------+
| Variable_name   | Value      |
+-----------------+------------+
| Last_query_cost | 8329.924107 |
+-----------------+------------+
```

まとめ

単一テーブルアクセスにおけるコスト計算

- 「レンジスキャンにおけるコスト計算アルゴリズムはそもそも誤っているのではないか」という懸念はいったん横に置いておいて、得られたコストが結局のところ何を表しているのかを考えてみましょう。

- read_timeから算出されるページ数由来のコストは実際の値が小さいこともあり、InnoDBの場合ほとんど意味を持ちません。元々read_timeは、テーブルデータをキャッシュしないMyISAMのために用意されたパラメータだと考えられます。

- InnoDBにおいては、found_recordsで示される読み取りレコード数の推定値がコストの大半を占めています。大雑把にまとめると、Last_query_costは以下のように概算することが可能です。
  - フルスキャン：テーブルのレコード数 × 0.2
  - ユニークスキャン：1.0
  - 10ページ以下のレンジスキャン：読み取りレコード数 × 0.4
  - 11ページ以上のレンジスキャン：読み取りレコード数 × 0.8
    (ただし、テーブルのレコード数 × 0.2を超えた場合はその値に補正される)

- またここから、InnoDBがインデックスによるデータアクセスをフルスキャンに比べて4倍、検索範囲が狭い場合は2倍高コストだとみなしていることが分かります。

もっと詳しく調べるには

MySQL 5.6 Optimizer Tracing

- MySQL 5.6から、Optimizer Tracingという新機能によってSQLオプティマイザの挙動を確認できるようになります。

```
mysql> SET LOCAL optimizer_trace = 'enabled=on,end_marker=on';
mysql> EXPLAIN SELECT * FROM item WHERE i_name = 'NFOHP7ywvB';
mysql> SELECT * FROM information_schema.OPTIMIZER_TRACE¥G

*************************** 1. row ***************************
QUERY: EXPLAIN SELECT * FROM item WHERE i_name = 'NFOHP7ywvB'
TRACE: {
～
            "rows_estimation": [
              {
                "database": "scott",
                "table": "item",
                "table_scan": {
                  "rows": 100274,
                  "cost": 289
                } /* table_scan */
              }
            ] /* rows_estimation */
```

- トレース結果はinformation_schema.OPTIMIZER_TRACEテーブルにJSON形式で出力されます。

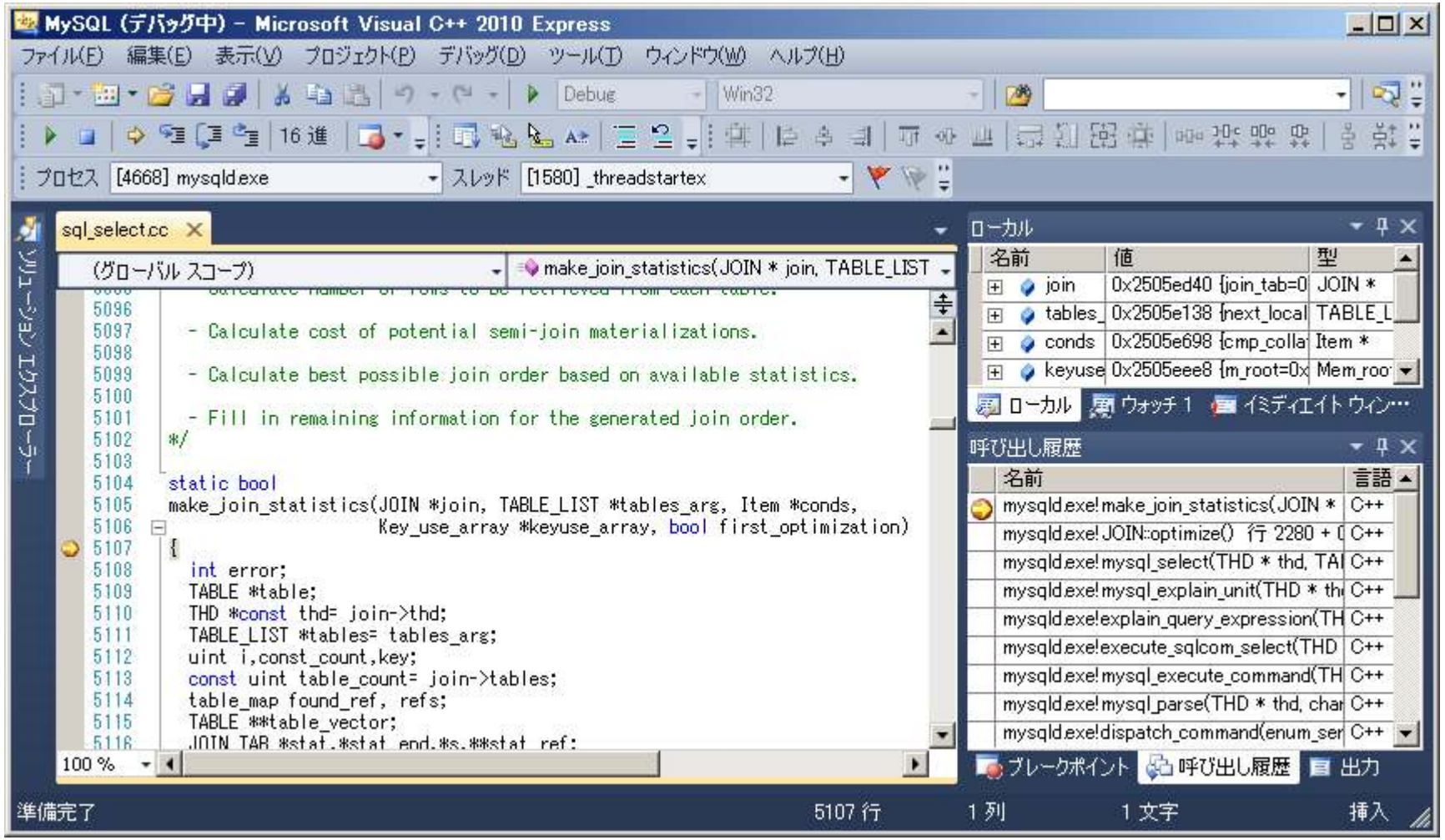
デバッグログ

- mysqldのデバッグ版バイナリにdebugオプションをつけて起動すると、mysqld.traceというファイルにデバッグログが出力されます。

```
T@5    : | | | | | | | >JOIN::optimize
T@5    : | | | | | | | | >make_join_statistics
T@5    : | | | | | | | | | opt: rows: 100274
T@5    : | | | | | | | | | opt: cost: 289
T@5    : | | | | | | | | | >Optimize_table_order::choose_table_order
T@5    : | | | | | | | | | | >Optimize_table_order::greedy_search
T@5    : | | | | | | | | | | | >Optimize_table_order::best_extension_by_limited_search
T@5    : | | | | | | | | | | | | >Optimize_table_order::best_access_path
T@5    : | | | | | | | | | | | | | opt: access_type: "scan"
T@5    : | | | | | | | | | | | | | opt: rows: 100274
T@5    : | | | | | | | | | | | | | opt: cost: 289
T@5    : | | | | | | | | | | | | | <Optimize_table_order::best_access_path 8276
T@5    : | | | | | | | | | | | | | opt: cost_for_plan: 20344
T@5    : | | | | | | | | | | | | | opt: rows_for_plan: 100274
full_plan; idx :1  best: 20343.8  accumulated: 20343.8  increment: 20343.8  count: 100274
     POSITIONS: item
BEST_POSITIONS: item
      BEST_REF: item(100274,100274,289)
T@5    : | | | | | | | | | | <Optimize_table_order::best_extension_by_limited_search 9350
optimal; idx :1  best: 20343.8  accumulated: 0  increment: 0  count: 1
     POSITIONS: item
BEST_POSITIONS: item
      BEST_REF: item(100274,100274,289)
T@5    : | | | | | | | | | <Optimize_table_order::greedy_search 8852
T@5    : | | | | | | | | | <Optimize_table_order::choose_table_order 8395
T@5    : | | | | | | | | <make_join_statistics 5723
T@5    : | | | | | | | <JOIN::optimize 2779
```

デバッグ

- sql/sql_select.ccのmake_join_statistics()にブレークポイントを仕掛けてステップ実行していきます。10時間ぐらい粘れば見えてくるはずです。



参考文献

- Sergey Petrunia氏のMySQL Conferenceプレゼンテーション資料
  Sergey氏は旧MySQL ABのSQLオプティマイザ担当で、現在はMonty Program ABでSQLオプティマイザの開発を行っています。
  - Understanding and Control of MySQL Query Optimizer: Traditional and Novel Tools and Techniques (2009)
    http://www.mysqlconf.com/mysql2009/public/schedule/detail/6864
  - New Query Engine Features in MariaDB (2010)
    http://en.oreilly.com/mysql2010/public/schedule/detail/13509
  - Why Are The New Optimizer Features Important and How Can I Benefit From Them? (2011)
    http://en.oreilly.com/mysql2011/public/schedule/detail/19899
    ⇒ MariaDB 5.3でついに待望のHASH JOINが実装されました。

- 奥野 幹也氏(@nippondanji)のEnterpriseZine連載記事
  - MySQLチューニング虎の巻
    Block Nested Loop Join／Batched Key Access Join
    http://enterprisezine.jp/dbonline/detail/3606
    ⇒ MySQL 5.6で追加されたテーブル結合アルゴリズムの解説記事です。

おわりに

おわりに

- MySQL SQLオプティマイザのコスト計算アルゴリズムについては、これまで解説記事がどこにもありませんでした。MySQL Forgeにもありませんし、私の知る限り書籍もないです。というわけで、この資料がみなさまのお役に立てば幸いです。

- 今回はさわりの部分しか紹介できなかったので、ブログなどで続きを解説していければと考えています。
  - テーブルを結合した場合のコスト計算と探索アルゴリズム
  - サブクエリを用いた場合のコスト計算
  - 他のRDBMSとの違い

宿題

- 以下のSQLについて調べてください。

```
SELECT o.o_id, ol.ol_id, ol.i_id, ol.ol_quantity, o.o_date
FROM orders o INNER JOIN order_line ol ON o.o_id = ol.o_id
WHERE o_date BETWEEN '2011-12-01 00:00:00' AND '2011-12-02 00:00:00';
```

1. テストデータをロードして、SQL実行計画とコストを確認してください。

2. ordersテーブルのo_dateカラムにインデックスを作成するとSQL実行計画とコストがどのように変化するか、確認してください。

3. なぜorder_lineテーブルを駆動表とするSQL実行計画が選ばれなかったのか、order_lineテーブルが駆動表になった場合のコストとあわせて説明してください。

4. これらのコストはそれぞれどのような計算アルゴリズムで求められたのか、説明してください。

5. Oracle DatabaseとPostgreSQLでSQL実行計画を確認し、MySQLと比較してください。