2019年度性能トラブル対処法 インデックス設計追記版

PostgreSQLエンタープライズ・コンソーシアム　技術部会
ヤマトシステム開発株式会社

アジェンダ

- 1. インデックス設計において考慮すべきポイント
  - 1.1. 考慮すべきポイント
  - 1.2. 考慮漏れにより起こりうる性能トラブルの例

- 2. テーブルフルスキャンとインデックススキャンの処理時間調査
  - 2.1.目的
  - 2.2.実験手順・環境
  - 2.3.実験結果
  - 2.4.読み取れた傾向
  - 2.5.想定と異なる傾向

1. インデックス設計において考慮すべきポイント

1.1.考慮すべきポイント

- テーブルのデータ型やパーティショニングを検討・設計した後、想定される検索や結合の条件を精査し、対象となる列にインデックスを定義します。
- PostgreSQL ではデフォルトでは B-Tree インデックスが作成されますが、特殊な用途がない限りはこれで問題ありません。
- 闇雲なインデックスの付与は更新処理の性能低下につながります。定義すべき列の選択は、システムの用途により、以降の観点で精査する必要があります。精査の結果インデックススキャンが有利でない場合はテーブル設計とSQL設計も含めた検討が必要です。

1.1.1.オンライントランザクション系①

- 設計においては、以下の実行計画が採用されるようなSQL とテーブル構造を目標とします。
  - テーブルの結合方法: ネステッドループジョイン
  - テーブルへのアクセス方法: インデックススキャン

※ オンライントランザクション系では参照するデータが少量であり上記実行計画がメモリの使用量/参照するブロック数の観点から最も効率がよいと想定できる。

- インデックスは検索SQLの内、ネステッドループジョインの以下の列に作成します。
  - WHERE 句の条件に記述されている駆動表の列
  - テーブル結合条件に記述されている参照表の列

※ 駆動表は外部表、参照表は内部表とも呼ばれる。

1.1.2.オンライントランザクション系②

- 駆動表に対する条件のうちインデックスを作成する項目は
  データが累積しても抽出件数が少量かつ変化しにくいものを
  選択します。
  - 例：「処理済みフラグ」が「未処理」のもの、処理日付など
- 検索条件が適切なインデックスを定義できない場合
  テーブルの非正規化を検討します。
  - 例： 未入金の売り上げを調査したい
    - × 売上テーブルのうち入金テーブルに紐付かないレコードを検索する
      ⇒ 売上テーブルのテーブルフルスキャンになってしまう
    - ○ 売上テーブルに未入金フラグを用意しインデックスを定義する
- ORDER BY や GROUP BY に指定される列にインデックスを
  設定することで、ソート処理が回避されパフォーマンスが
  向上する場合があります。

1.1.3.データウェアハウス系

- 検索条件のうち、カーディナリティの高い列に
  インデックスを作成します。
  - オンライントランザクション系の場合は要件が明確であり、インデックス設計に
    「この業務ではどの項目で検索しなければならないか」
    というアプローチが可能です。
    一方データウェアハウス系ではアドホックな検索が中心になり
    同じアプローチはとれません。
  - カーディナリティの推測値は以下のビューのカラムから確認できます。
    pg_stats.n_distinct
- ただし、インデックスによる抽出件数が多くなると
  テーブルフルスキャンの方が性能が良い場合があるため
  注意が必要です。
  本資料では実際に性能が逆転する事例を検証しています。
  詳細は「2. テーブルフルスキャンとインデックススキャンの
  処理時間調査」を参照ください。

1.1.4.その他 考慮すべきポイント

- B-Treeインデックスにおいては更に性能を向上させられる応用的な使い方もあり（参考文献1）、例として部分インデックスの利用で大幅に性能が向上するケースもあります（参考文献2）。

- 定義したインデックスの使用頻度が低いと更新性能の足かせとなります。

- 統計情報を参照し不要なインデックスを削除することも全体的なパフォーマンス向上のために必要です（参考文献1）。

1.2.考慮漏れにより起こり得る性能トラブルの例

- 意外と現場で起こりうるのがレコード数の多いテーブルに対するインデックスの定義漏れであり、テーブルフルスキャンが実行されディスクI/Oが上昇してしまうなどのトラブルに繋がることがあります。

- 一定規模のテーブルに対するインデックスの定義漏れ、検討漏れに注意が必要です。

2. テーブルフルスキャンとインデックススキャンの処理時間調査

2.1.目的

- パフォーマンスに優れたSQLを実行するためには、
  要件によってテーブルフルスキャンかインデックススキャンの
  どちらかを選択する必要があります。
  - 当然ながら、インデックススキャンとするならば
    インデックスを作成する必要があります。
- しかし、テーブルフルスキャンとインデックススキャンの
  処理時間が逆転するレコードの参照件数は「全件数の10％」
  といった固定の割合で発生するわけではありません。
- 処理時間が逆転する件数は、「データの格納状態」と
  「インデックス順」の関係により大きく変化します。
- そこで、「データの格納状態」が「インデックス順である」または
  「インデックス順でない」環境を用い、データ参照方式と
  レコードの参照件数を変化させた時の処理時間を計測し
  傾向を分析しました。

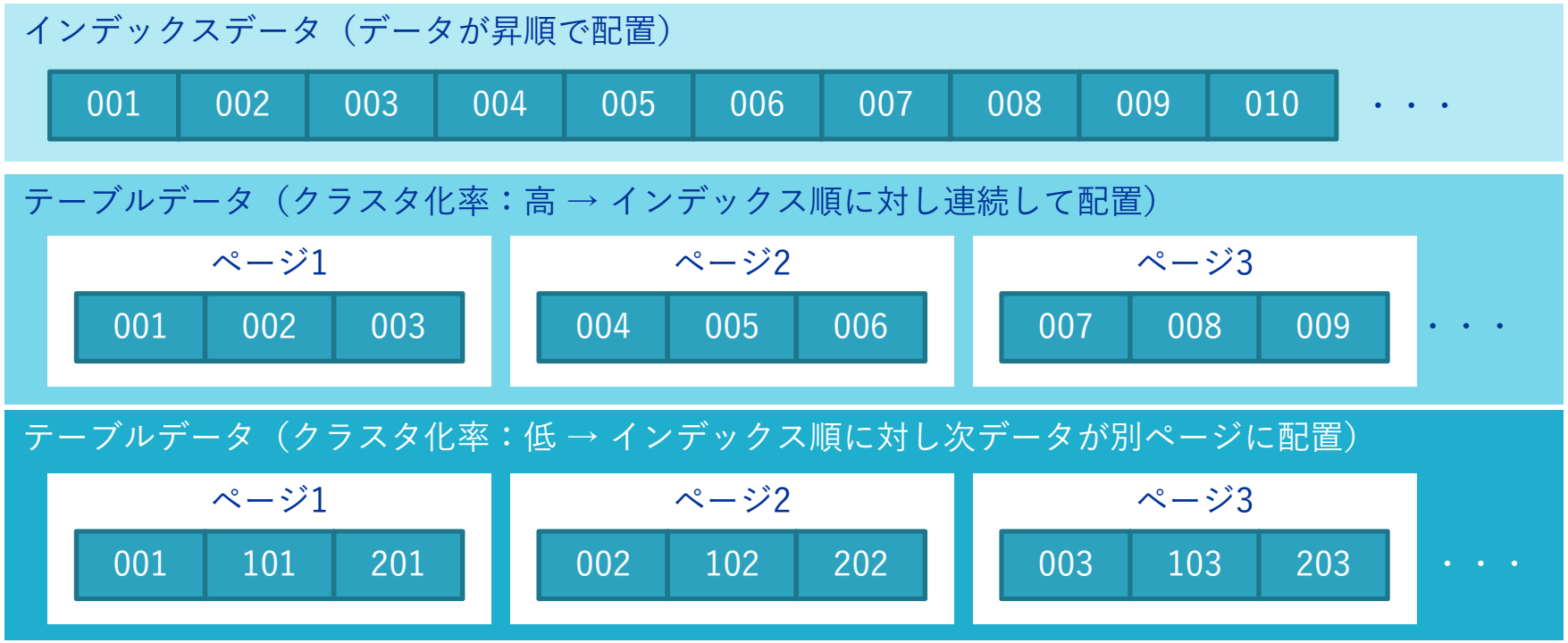
2.2.実験手順・環境

- 1000万件のテーブルを用意※し、レコードの抽出件数を
  1%から100%まで変化させ処理時間を計測します。
  計測は下記3つの条件を組み合わせ、8パターン実施しました。
  a. 実行計画は インデックススキャンとテーブルフルスキャン
  b. HDD の他、HDD と比較しランダムアクセス性能が高い
     SSD にテーブルデータを配置
  c. インデックススキャンによる範囲検索に対して
     効率の良い/悪いデータの配置を考慮したテーブルを用意
- shared_buffers はテーブルデータが充分載るサイズを
  設定しました。
- データファイルからの読み取り性能を比較するため
  クエリごとに OS ファイルキャッシュのクリア と PostgreSQL の
  再起動(shared_buffers のクリア)を実施しました。

※テーブルレイアウト、データサイズ、クエリは巻末の参考情報をご確認ください

2.2.1.条件cについて補足

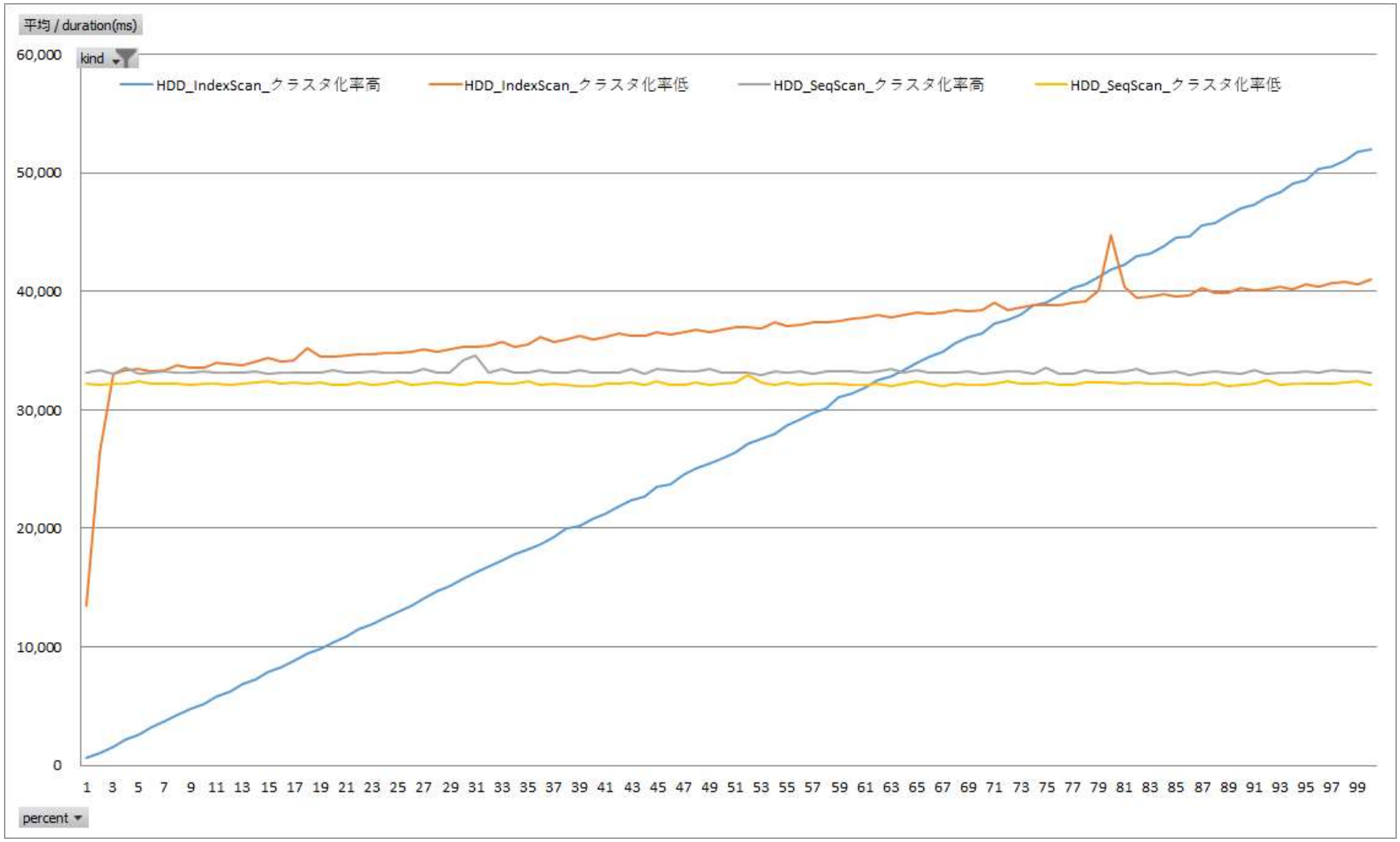
インデックススキャンによる範囲検索	インデックス順に対するテーブルデータの物理配置	インデックスに対するテーブルのクラスタ化率
効率が良い	連続して配置される	クラスタ化率：高
効率が悪い	次データが別ページに配置される	クラスタ化率：低



2.2.2.shared_buffers 設定について補足

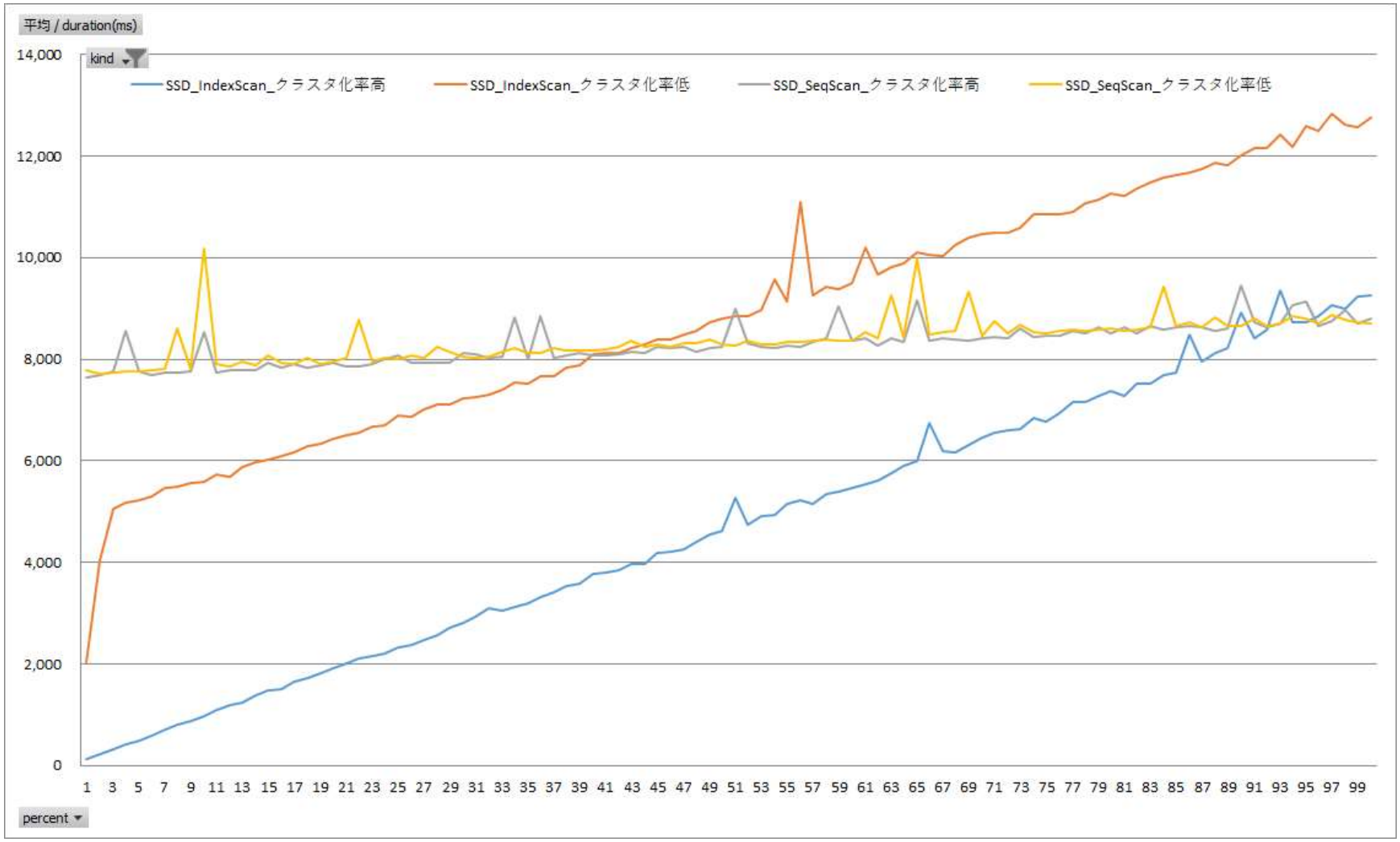
- テーブル全体が shared_buffers(以降共有メモリとします)に載らない場合、クラスタ化率低のデータに対するインデックススキャンでは一度読んだページを再度読む際に、該当ページは既に共有メモリから追い出されており、常にファイルからの読み込みが発生します。
  そのため処理時間の傾向としては、共有メモリに載る場合のグラフのテーブルデータの全ページが参照されるまで(今回では2.5%)の傾きが100%まで続くと容易に推測されます。
- 一方クラスタ化率高の場合、一度読んだページを再度読むことはないため、テーブル全体が共有メモリに載る場合と変わりません。
- そのため、今回の実験ではテーブル全体が載らない設定での実験は省略しています。

2.3.実験結果(HDDのみ)



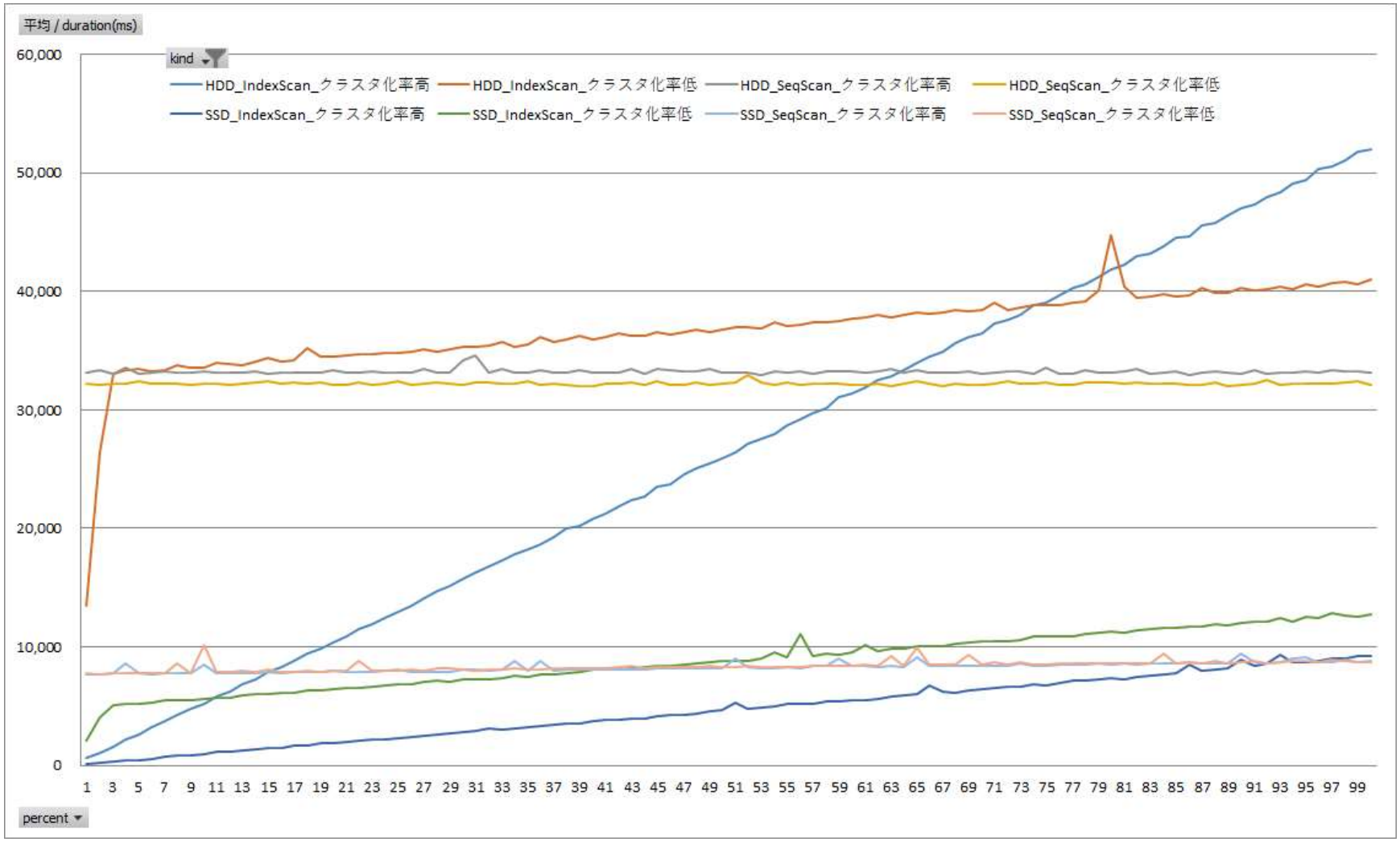
※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間(ms)を示す

2.3.実験結果(SSDのみ)



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間(ms)を示す

2.3.実験結果(全パターン表示)



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間(ms)を示す

2.4.読み取れた傾向

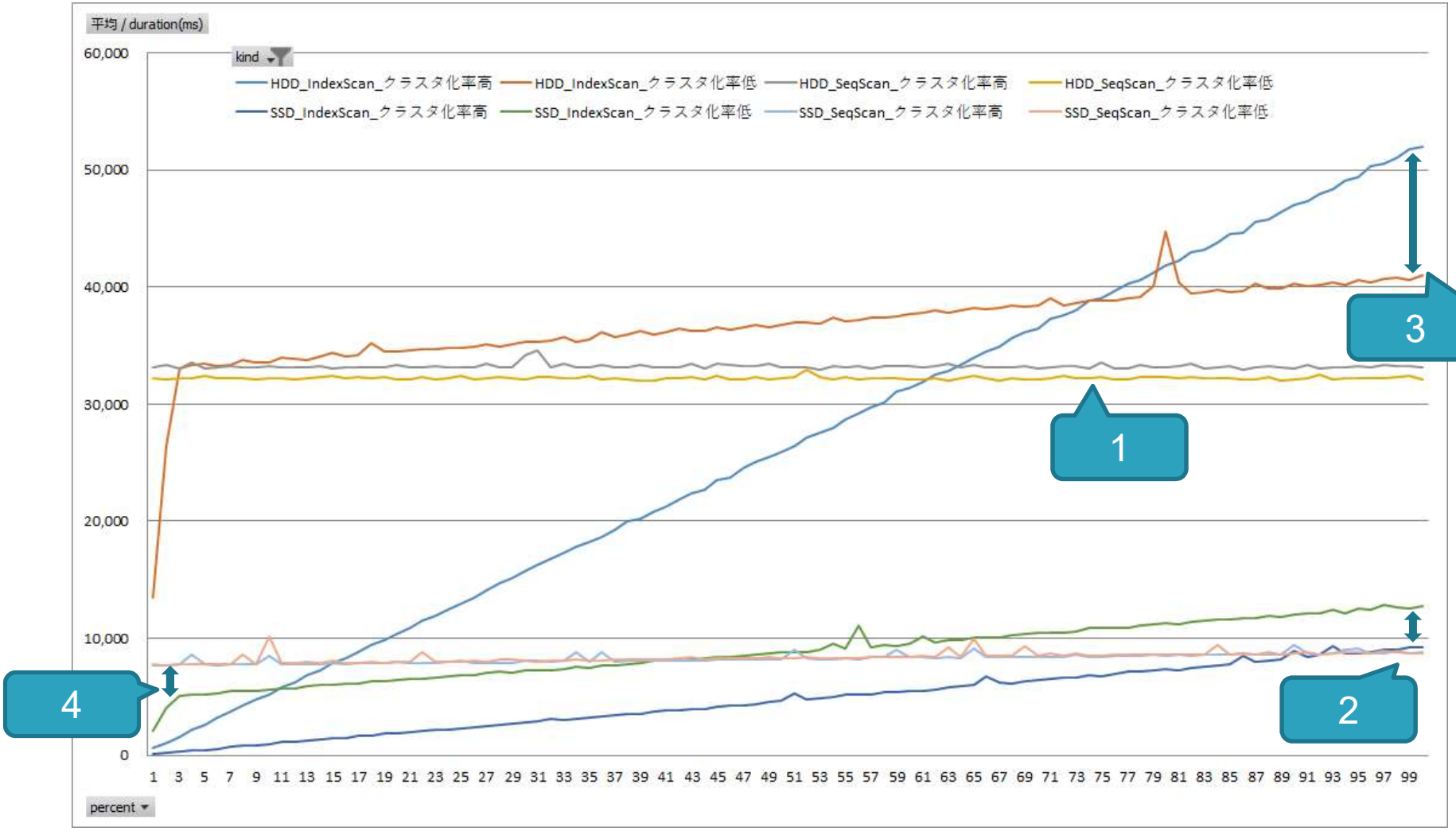
1. 非常に選択率が少ない場合(単一行の選択
   1%未満の範囲検索)、インデックススキャンが有利
   ⇒オンライントランザクション系はこの使い方で充分と考えられる。
2. テーブル全体が共有メモリに載り かつ 選択率が大きい場合
   クラスタ化度合いが処理時間に影響し
   特に SSD よりも HDD 環境では顕著に表れる
   a. SSD の場合、インデックススキャンの方が
      テーブルフルスキャンよりも有利な傾向が強い
   b. HDD の場合、テーブルフルスキャンの方が
      インデックススキャンよりも有利な傾向が強い
3. テーブル全体が共有メモリに載らない場合
   大量データ抽出にインデックススキャンは避けたほうが良い

※ あくまで傾向であり、実際のデータ格納順、ハードウェア構成により変化する

2.5.想定と異なる傾向

- 実験結果に想定と異なる傾向が見つかりました。
  追加実験により、１～３の原因は確認済みです。
  1. HDD のテーブルフルスキャン２種類で
     クラスタ化率低の方がわずかに速い
  2. SSD のインデックススキャン２種類で
     100% のデータ読出し時、クラスタ化率低の方が遅い
  3. HDD のインデックススキャン２種類で
     100% のデータ読出し時、クラスタ化率低の方が速い
  4. SSD のデータ読み出しが 2.5% の時、クラスタ化率低の
     インデクススキャンの方がテーブルフルスキャンよりも速い
     （原因未確認）

2.5.1.想定外の傾向箇所



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間 (ms) を示す

2.5.2.傾向に対する考察1

1. HDD のテーブルフルスキャン2種類でクラスタ化率低の方がわずかに速い

想定

・参照するデータ量が同じであるため、処理時間が同じ

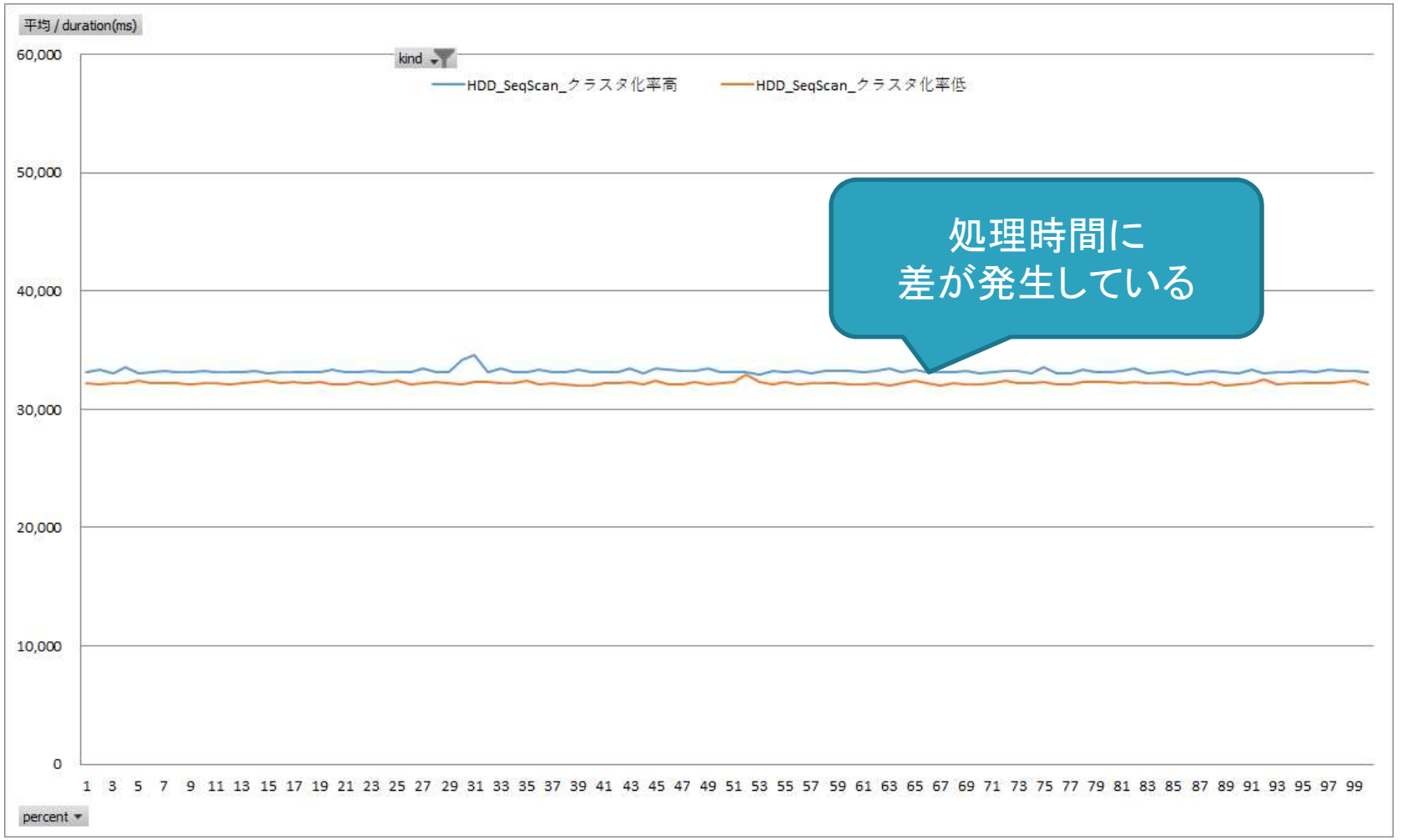
推測

・各データファイルのファイルサイスは同一（1.95GB）だが、DDで読出し時間を計測すると0.7秒差がある（５回計測の平均値）
そのため、ファイルのフラグメンテーションによる差と考えられる

確認

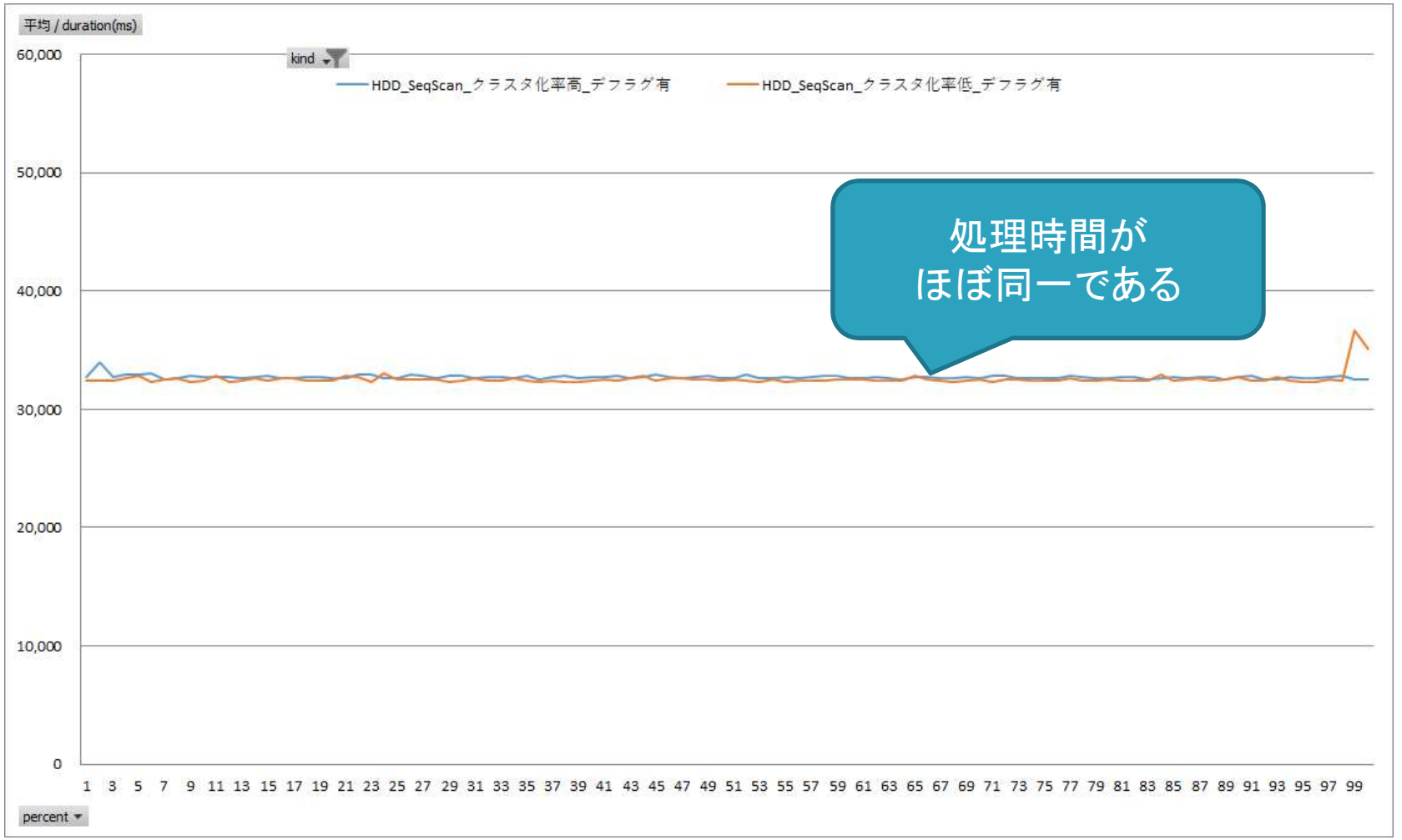
・デフラグ後再度計測した結果ほぼ同じ処理時間となることを確認

2.5.2.1.グラフによる比較 デフラグ前



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間(ms)を示す

2.5.2.2.グラフによる比較 デフラグ後



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間(ms)を示す

2.5.3.傾向に対する考察2

2. SSD のインデックススキャン2種類で 100% のデータ読出し時、クラスタ化率低の方が遅い

想定

・100%では参照するデータ量が同じであるため、処理時間が同じ

推測

・ SSDからの総読出し時間は同じだが、クラスタ化率高と比較しクラスタ化率低にてページイン/ページアウトの発生頻度が高い(詳細は後述)

確認

・ パフォーマンス計測ツール(perf)にて CPU キャッシュ状況を計測しクラスタ化率低の方が頻繁な書き換えが発生していることを確認

2.5.3.1.傾向が発生した原因の推測①

- SSD からの総読出し時間は同じだが、インデックススキャンによる範囲検索時にアクセスされるテーブルページの順番は、クラスタ化状態により異なります。

インデックス順	アクセスされるテーブルページ番号※	
	クラスタ化率高	クラスタ化率低
1	1	1
2	1	2
3	1	3
中略		
39	1	39
40	1	40
41	2	41

※ ページ当たり40レコード格納されているとする

2.5.3.1.傾向が発生した原因の推測②

- クラスタ化率高の場合は同一ページ内のデータが連続して参照されるため、対象ページが CPU の1次キャッシュ・2次キャッシュにのり、2行目から40行目までは DRAM より高速な CPU キャッシュ内で処理できます。
- しかしクラスタ化率低は、レコードごとに参照するページが異なるため、参照したページが CPU の1次キャッシュ、2次キャッシュから追い出され、全ての行が DRAM からの処理になるため時間を要したと考えられます。

2.5.3.2.パフォーマンス計測ツール 計測結果

- パフォーマンス計測ツール(perf)にて CPU キャッシュ状況を計測しました。クラスタ化率高と比べてクラスタ化率低の場合に頻繁なデータロードが発生していることを確認できました。

- perf 結果

データ格納パターン	L1キャッシュのロード回数	L2キャッシュのロード回数
クラスタ化率高	5,855,051,505	67,515,092
クラスタ化率低	8,531,979,022	147,054,646

2.5.4.傾向に対する考察3

3. HDD のインデックススキャン2種類で 100% のデータ読出し時、クラスタ化率低の方が速い

想定

・SSDのインデックススキャン2種類と傾向が同じ

推測

・インデックススキャンの際、クラスタ化率低と比較してクラスタ化率高はヘッドシークの移動発生頻度が高い(詳細は後述)

確認

・インデックスをテーブルとは物理的に別のディスク(SSD)に配置し再度計測した結果、SSDと同じ傾向となったことを確認

2.5.4.1.傾向が発生した原因の推測①

- インデックススキャンによるテーブルデータの参照は、
  以下の動作となります。
  1. インデックスのリーフノードからテーブルアドレスを取得
  2. 取得したテーブルアドレスを用いてテーブルデータを参照

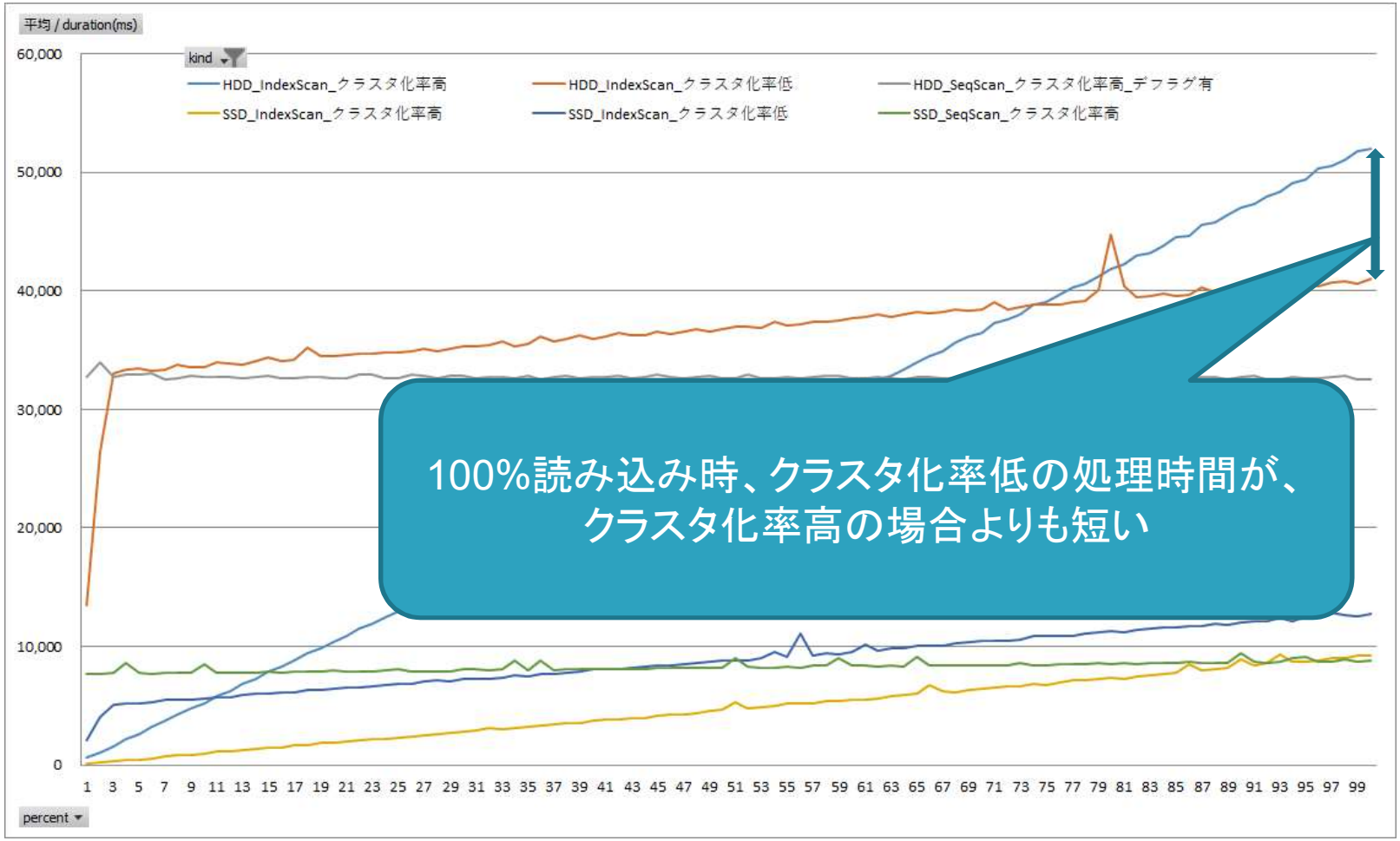
- インデックススキャンによる範囲検索の場合
  1 と 2 の処理を繰り返すこととなります。
  そのため、物理的に同一の HDD にテーブルとインデックスが
  格納されている場合、1 から 2、2 から 1 の動作の度に
  ヘッドシークが発生します。

2.5.4.1.傾向が発生した原因の推測②

- ここでインデックススキャンによる範囲検索時に、
  ヘッドシークが発生するデータ参照割合を考えます。
  - クラスタ化率低: 2.5%まで
    全体の2.5%の参照を行うとテーブルデータファイルの
    全ページを参照することになります。
    2.5%以降の参照はテーブルデータは共有メモリから
    参照されるのでヘッドシークは発生しません。
  - クラスタ化率高: 100%まで
    100%の件数を参照しないとテーブルデータファイルの
    全ページを参照せず、そのため100%までヘッドシークが
    発生します。
- 以上から、ヘッドシーク発生回数がクラスタ化率低の方が
  少なくなるため、処理時間が少なくなると考えられます。

2.5.4.2.グラフによる比較 ①

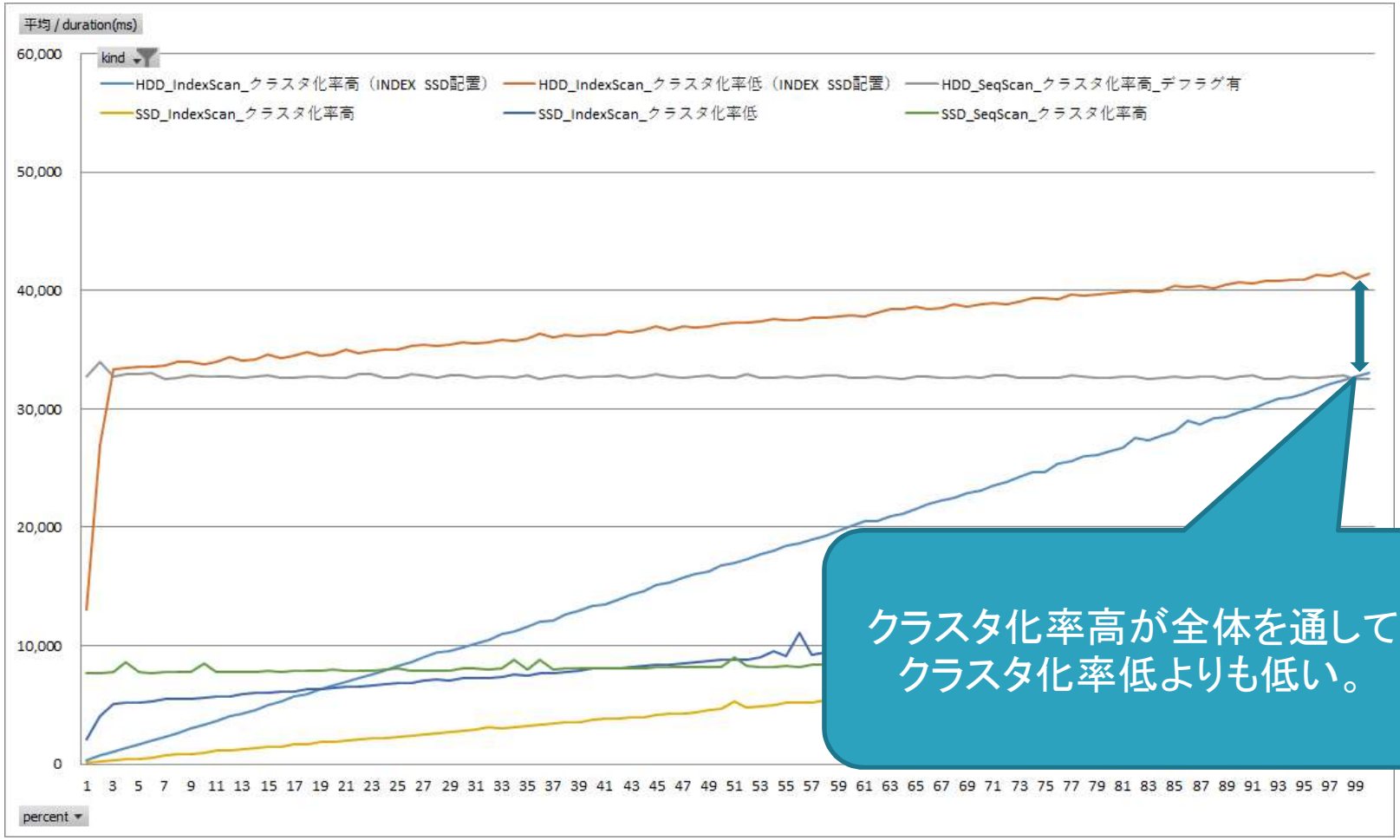
- 同一の HDD にテーブルとインデックスを格納した状態



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間 (ms) を示す

2.5.4.2.グラフによる比較 ②

- インデックスをテーブルとは別のディスク(SSD)に格納した状態



※ 横軸はデータの取得割合（1～100%）、縦軸は処理時間(ms)を示す

2.5.5.傾向に対する考察4

4. SSD の データ読み出しが 2.5% の時、クラスタ化率高の インデクススキャンの方がテーブルフルスキャンよりも速い

想定

・ファイルからの読込ブロック数は同じであり、処理時間も同じ

推測

・SSDの場合、PostgreSQL のレコード処理中は先読み実行(Read Ahead)が並列に動けないため、SSDの読み込みとレコード処理が直列化される
・HDDの場合は PostgreSQL のレコード処理と先読み処理にが並列して行われるため、全体の処理時間はディスクからの読込時間となり、処理時間はほぼ同じになる

確認

・力不足により確認方法が分からず、今後の課題とする
（情報提供をお待ちしております）

参考情報: 実行環境

- OS: CentOS Linux release 7.7.1908
- HDD と SSD の読み取り性能:
  - HDD:　60.5MB/秒
  - SSD: 432.0MB/秒
  - 確認コマンド
    dd if=<HDD または SSD に作成したファイル>
    of=/dev/null bs=1M count=1024
- PotgreSQL: 11.6

参考情報: テーブルレイアウト

```
CREATE TABLE
   test
   (   col01      char(30),
         -- 主キー。
         -- 1 から 10,000,000 の数値を左ゼロ埋めした値。
      col02      char(139),
         -- ダミーデータ
      CONSTRAINT test_pk PRIMARY KEY( col01 )
   )
```

参考情報: テーブルサイズ

項目	値
レコード数	1000 万件
レコード長	169 byte
ページサイズ	8,192 byte
ページ内格納レコード数	40
テーブルサイズ	1.95 GB
インデックスサイズ※	0.47 GB

※ インデックスはデータ投入後に作成しています。
データ投入前に作成するとクラスタ化率低の場合インデックスサイズが大きくなり
クラスタ化率高の場合と条件が異なるためです。

参考情報: データ投入①

- CSV ファイルを用意し COPY コマンドにより投入
- クラスタ化率高の CSV ファイルデータ
  - test.col01 の値を、単純に増加
  - イメージ

```
00000000000000000000000000000001,aaaaaaa....
00000000000000000000000000000002,aaaaaaa....
00000000000000000000000000000003,aaaaaaa....
00000000000000000000000000000004,aaaaaaa....
00000000000000000000000000000005,aaaaaaa....

中略

00000000000000000000009999996,aaaaaaa....
00000000000000000000009999997,aaaaaaa....
00000000000000000000009999998,aaaaaaa....
00000000000000000000009999999,aaaaaaa....
00000000000000000000010000000,aaaaaaa....
```

参考情報: データ投入②

- クラスタ化率低の CSV ファイルデータ
  - test.col01の値を 25万間隔で増加 かつ 40レコードごとに40レコード前の値+1に戻る ようにデータを作成する
  - イメージ

```
00000000000000000000000000001,aaaaaaa....  <-1レコード目 1ページの最初
00000000000000000000000250001,aaaaaaa....
00000000000000000000000500001,aaaaaaa....

中略

00000000000000000000009750001,aaaaaaa....  <-40レコード目 1ページの最後
00000000000000000000000000002,aaaaaaa....  <-41レコード目 2ページの最初
00000000000000000000000250002,aaaaaaa....

中略

00000000000000000000010000000,aaaaaaa....  <-1000万レコード目 25万ページの最後
```

参考情報: インデックススキャン時のクエリ

```
SELECT
   *
FROM
   test
WHERE
   col01   >=  '00000000000000000000000000000001'
AND col01   <=  '00000000000000000000000000250000'
   -- 1%から100%の割合で変化させる
```

参考情報: テーブルフルスキャン時のクエリ

```
SELECT
   *
FROM
   test
WHERE
   col01 || '_' >=  '00000000000000000000000000000001' || '_'
AND col01 || '_' <=  '00000000000000000000000000250000' || '_'
   -- 1%から100%の割合で変化させる
   -- カラムに文字列連結することでインデックススキャンを選択させない
```

参考情報: PostgreSQL 実行計画関係の設定

enable_bitmapscan	off
enable_gathermerge	on
enable_hashagg	off
enable_hashjoin	off
enable_indexonlyscan	off
enable_indexscan	on
enable_material	off
enable_mergejoin	off
enable_nestloop	on
enable_parallel_append	off
enable_parallel_hash	off
enable_partition_pruning	off
enable_partitionwise_aggregate	off
enable_partitionwise_join	off
enable_seqscan	**off**
enable_sort	on
enable_tidscan	on

※ 大量データ参照時にインデックス
スキャンが選択されるよう
enable_seqscan を off に設定

参考情報: キャッシュクリア手順とコマンド

1. PostgreSQL 停止

```
pg_ctl stop -w
```

2. OS キャッシュクリア

```
echo 3 > /proc/sys/vm/drop_caches
```

3. PostgreSQL 開始

```
pg_ctl start -w
```

参考文献

- 文献1. PostgreSQLのインデックス・チューニング by Tomonari Katsumata

  https://www.slideshare.net/InsightTechnology/dbts-osaka-2014-b23-postgresql-tomonari-katsumata

- 文献2. PostgreSQL SQLチューニング入門　実践編

  https://www.slideshare.net/satoshiyamada71697/postgresql-sql

PostgreSQL Enterprise Consortium

POSTGRESQL ENTERPRISE CONSORTIUM