PGECons
PostgreSQL Enterprise Consortium

2023年度WG3活動成果報告
運用課題解決編
適切なVacuum設定の考察

PostgreSQL Enterprise Consortium

Contents

責任範囲

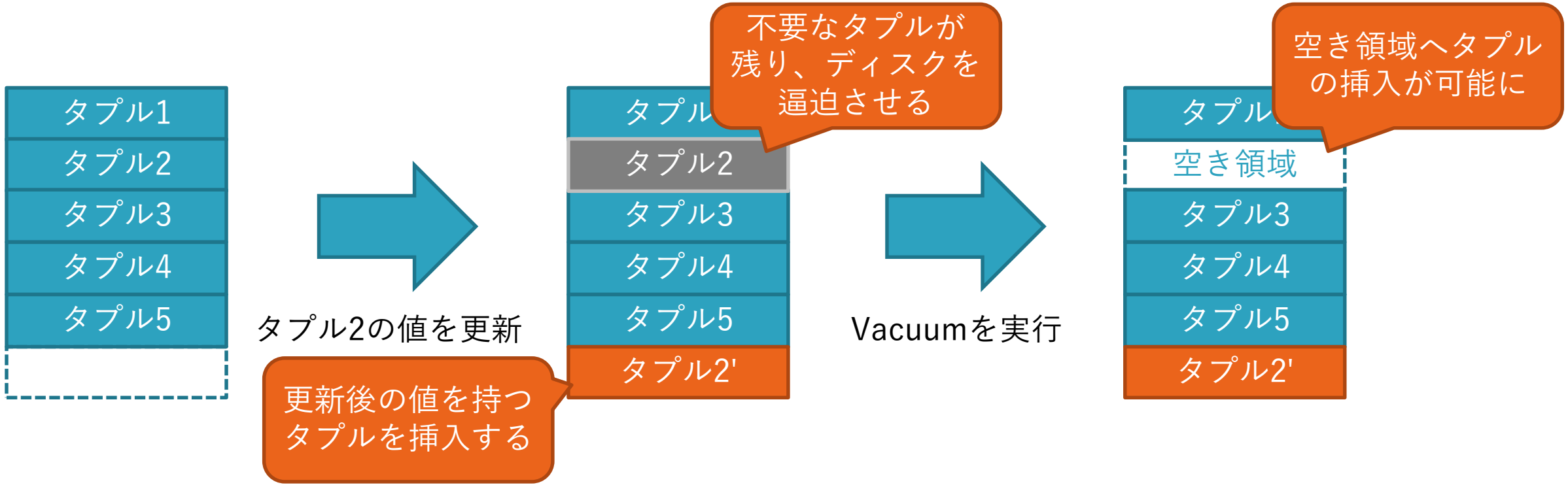
- 本資料は、PGEConsが独自に検証した結果であり
  結果はPGEConsの責任の元、公開しています。

Purpose of this verification

本検証の目的

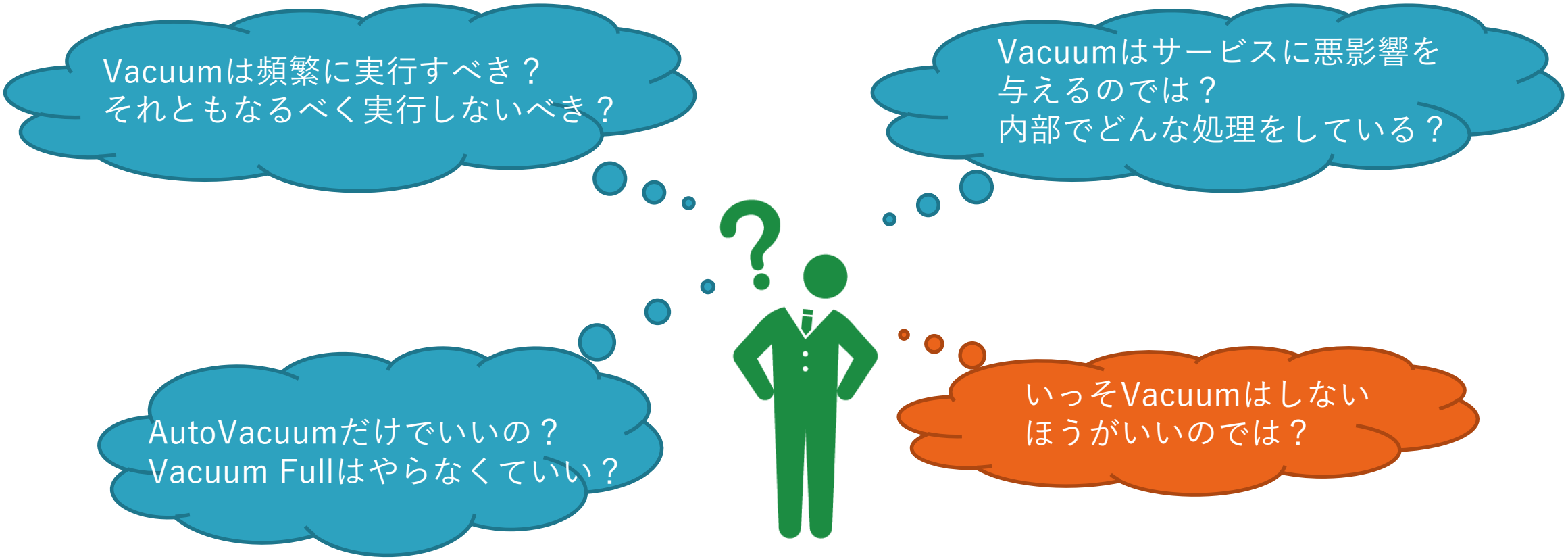
Vacuumとは

- **追記型アーキテクチャであるPostgreSQLにおいて特徴的な機能**
  - タプルの更新時、ディスク上のデータを書き換えるのではなく
    新規にタプルが追加する処理が行われている
  - これにより、不要となった更新前のタプルがディスクを逼迫させてしまう。
    それを解消させる機能がVacuumとなる



Vacuumの設計における課題

- Vacuumは重要な機能であるが、その詳細まではあまり知られていない

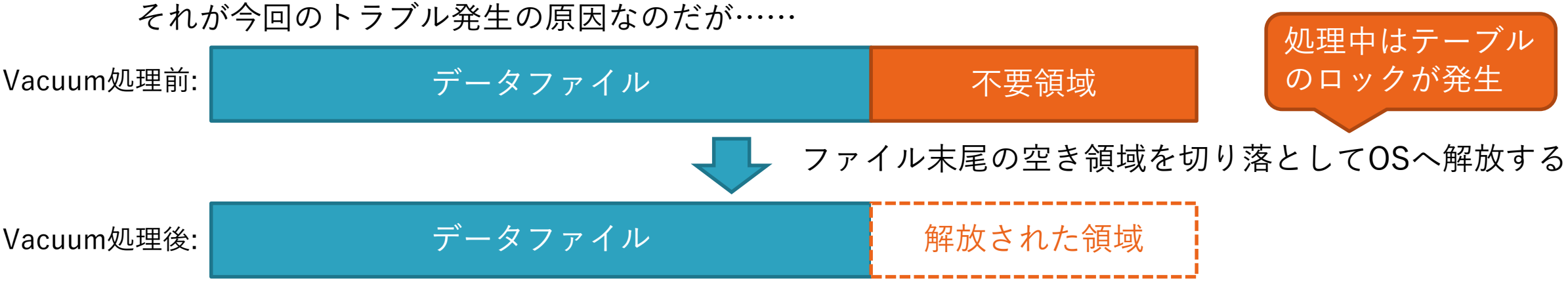


事例 – Vacuumによるトラブル

■ とある企業でVacuumによるトラブルが発生

- あるデータベースにてクエリのレスポンス遅延が発生。遅延は47秒間継続し、遅延発生中に合計55件ものクエリでタイムアウトが発生した
- レスポンス遅延が発生した47秒は、AutoVacuumによる**切り落とし処理**に要した時間。
  切り落とし処理中はテーブルがロックされるため、クエリがロック解放待ちとなってしまった
- Vacuum実行時の負荷は考慮していたが、切り落とし処理のロックまでは考慮されていなかった。
  それが今回のトラブル発生の原因なのだが……

切り落とし処理については本発表で詳しく解説予定



レスポンス遅延発生の“真因”はVacuumへの理解不足であった

目的

Vacuumを正しく運用するためには、
Vacuumそのものについて理解する必要がある

本発表を通じてVacuumを正しく理解し、
その上で最適な設計方式について考察する

Vacuum seen from its role

役割から見るVACUUM

Vacuumの持つ役割

- **Vacuumが持つ役割と機能を正しく理解する**
  - Vacuumは一般に「不要なタプルを削除して空き領域を確保する」ことが目的だと考えられているが、それだけではない
  - Vacuumの役割として次が挙げられる

① 更新前タプルの削除

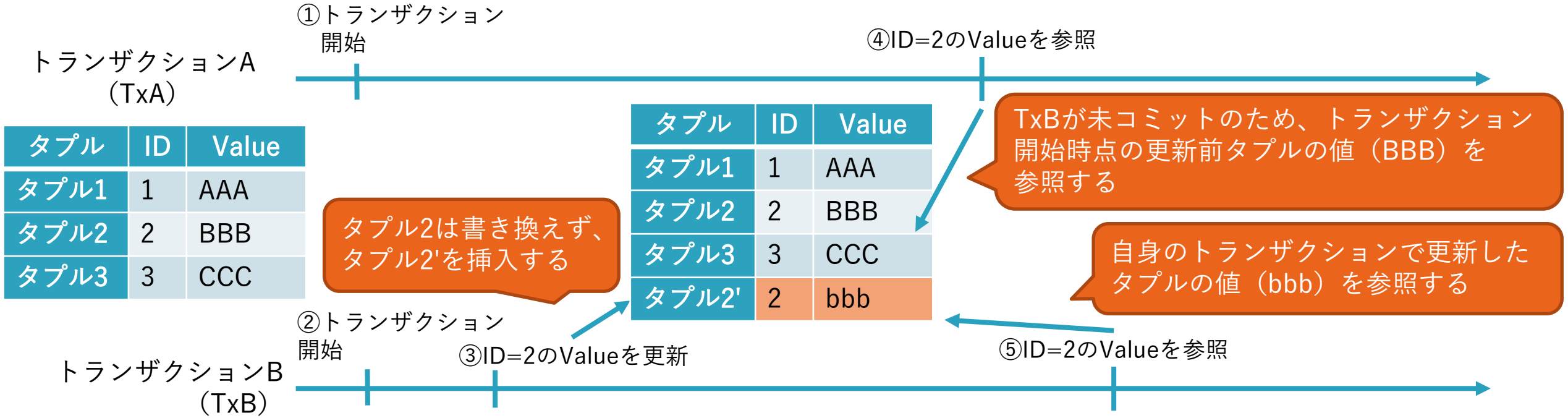
② 空き領域の詰め

③ インデックスの均一化

④ トランザクションIDの周回問題回避

①更新前タプルの削除 – PostgreSQLの多版型同時実行制御

- **データ整合性の維持には多版型同時実行制御（MVCC）が使用されている**
  - データ更新時、そのタプル行を直接書き換えるのではなく新たに更新後の情報を持つタプルを追記している
  - 更新前の情報をテーブル内に残すことで参照の整合性を維持しているが、参照される可能性がある限りそのタプルを削除することができない



タプル	ID	Value
タプル1	1	AAA
タプル2	2	BBB
タプル3	3	CCC

タプル	ID	Value
タプル1	1	AAA
タプル2	2	BBB
タプル3	3	CCC
タプル2'	2	bbb

①更新前タプルの削除 – Vacuumによるタプル削除

- **不必要になった更新前タプルを削除し空き領域を作る**
  - データの更新時は現在のタプルのXMAX（参照終了期間）に
    その処理を行ったトランザクションID（XID）が挿入される
  - Vacuumが実行されると、未完了のトランザクションのXIDのうち最も小さいものとそのタプルが持つXMAXの値を比較する。XMAXの値が未完了のトランザクションのXIDよりも小さい場合はそのタプルを削除し、空き領域として使用できるようにする

Vacuumプロセスで両者を比較

テーブル構成

ID	Value	XMAX
1	AAA	-
2	BBB	101
3	CCC	104
2'	bbb	-
3'	ccc	-

トランザクションの実行状況表

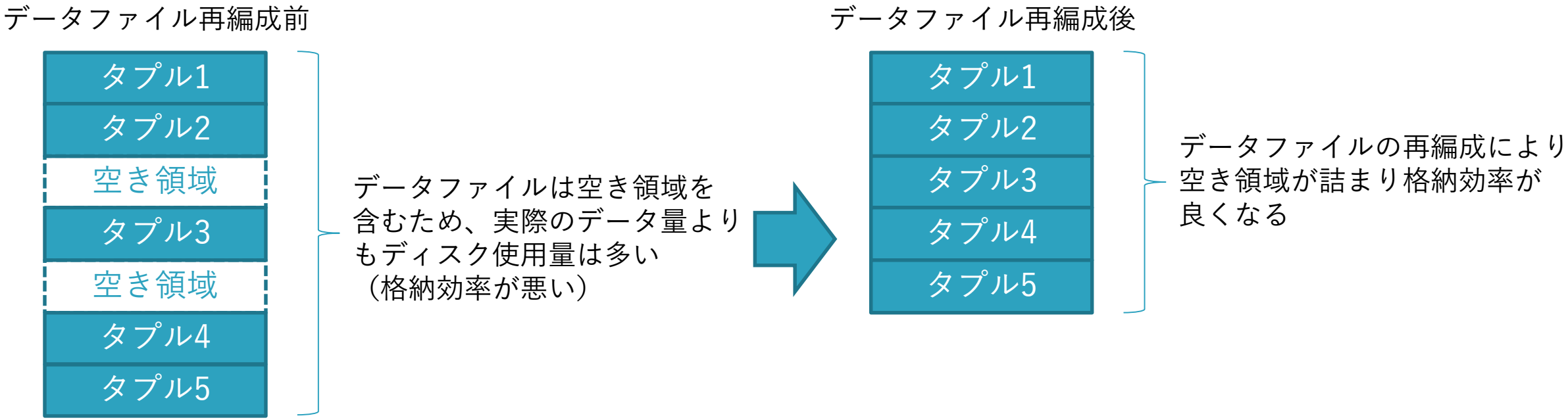
トランザクション名	XID	ステータス
TxA	100	完了
TxB	101	完了
TxC	102	未完了
TxD	103	未完了

未完了のトランザクションのXIDで一番小さいものは102

ID2のタプル: 未完了トランザクションのXIDよりも小さい（もう参照されることがない）ので削除可能
ID3のタプル: 未完了トランザクションのXIDよりも大きい（トランザクションCで参照される可能性がある）ので削除不可

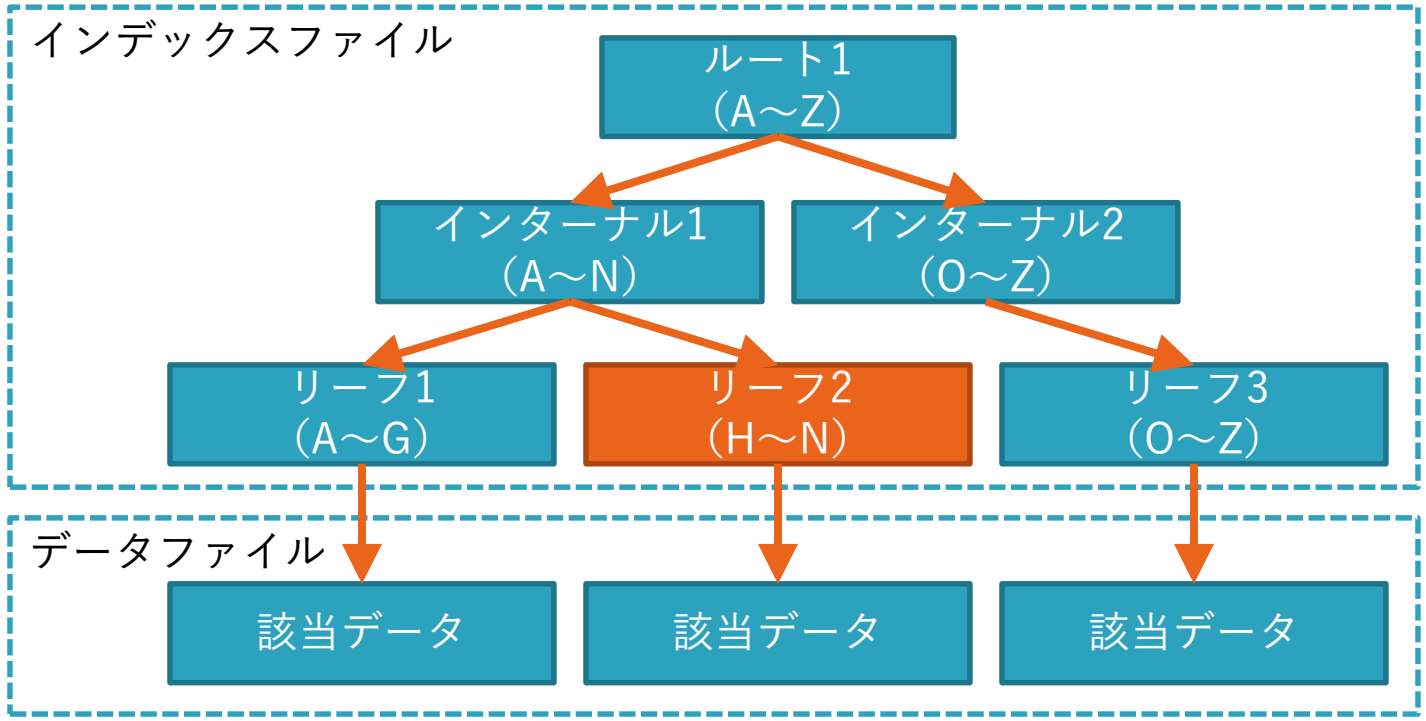
②空き領域の詰め

- **データファイルを再編成し、不要な空き領域を詰める**
  - Vacuumによって不要な領域が再利用できるが、これが繰り返されるとデータが歯抜けの状態となり、実際のデータ量よりもOS上のディスク使用量が大きくなってしまう
  - データファイルの再編成を行うことで、空き領域を書き込み可能なディスク領域としてOSへ返却する
  - 後述のVacuum Fullでのみ実行される



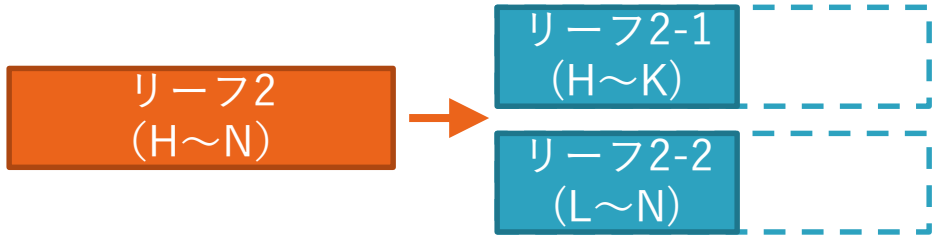
③インデックスの均一化

- **インデックスツリーの不均一化を解消し、参照速度への影響を抑える。**
  - タプルの挿入や更新が何度も行われることで、インデックスのツリー構造が不均一になる。ツリー構造が不均一になれば参照速度に影響を与える
  - 後述するVacuum Fullであればインデックスの再編成が行われる。
    Standard Vacuumだと完全な再編成ではなく可能な範囲での解消のみとなる



特定のリーフにのみ更新が集中すると、そのリーフだけが肥大化する

- 肥大化したリーフは、他のリーフと比べて参照時間が長くなる
- リーフに格納されるデータ量がページサイズを超えると、ページの分割（SPLIT）が発生する。これにより新しいページの獲得や親インターナルの更新が必要となる



④トランザクションIDの周回問題回避 – トランザクションIDとは

- **MVCCを行うためにトランザクション毎に払い出されるID**
  - PostgreSQLではトランザクションごとに（XID）が払い出される。XIDは32bit（約40億）で管理されるが、全て使い切った場合は3に戻り周回して再利用される
  - 実行中のトランザクションは自分よりも古いXIDのトランザクションは参照できるが、ロールバック済み・実行中・新しいトランザクションは参照できない

タプル	ID	Value	XMIN	XMAX
タプル1	1	AAA	90	-
タプル2	2	BBB	91	101
タプル3	3	CCC	92	103
タプル2'	2	bbb	101	-
タプル3'	3	ccc	103	-

トランザクション開始時点で存在するタプルのため、参照可能

トランザクション開始時点で存在しないタプルのため、参照不可

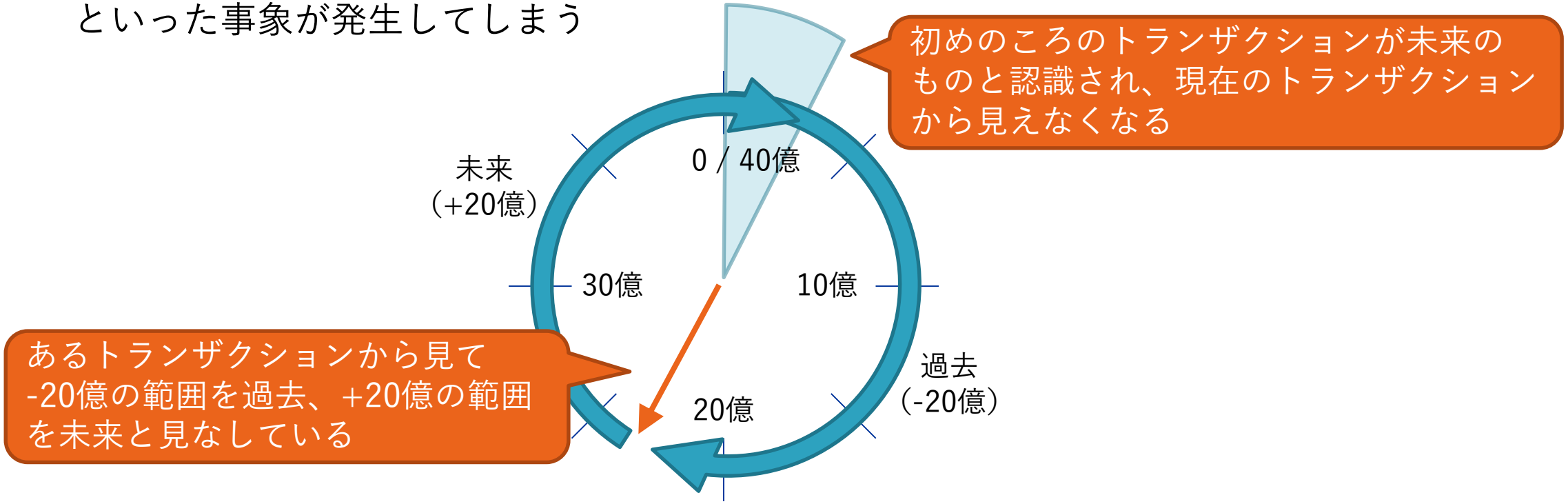
トランザクション名	XID	ステータス
トランザクションA	100	完了
トランザクションB	101	完了
トランザクションC	102	未完了
トランザクションD	103	未完了

IDが2のタプルを更新

IDが3のタプルを更新

④トランザクションIDの周回問題回避 – 周回問題とは

- **周回によりXIDの逆転が発生する**
  - 現在のXIDより、前20億を過去、後ろ20億を未来として取り扱う
  - XIDが20億を超えると、最初のころのXIDが未来のものとして認識されてしまう
  - これにより、存在するはずのタプルが新しいトランザクションで見えなくなる、といった事象が発生してしまう

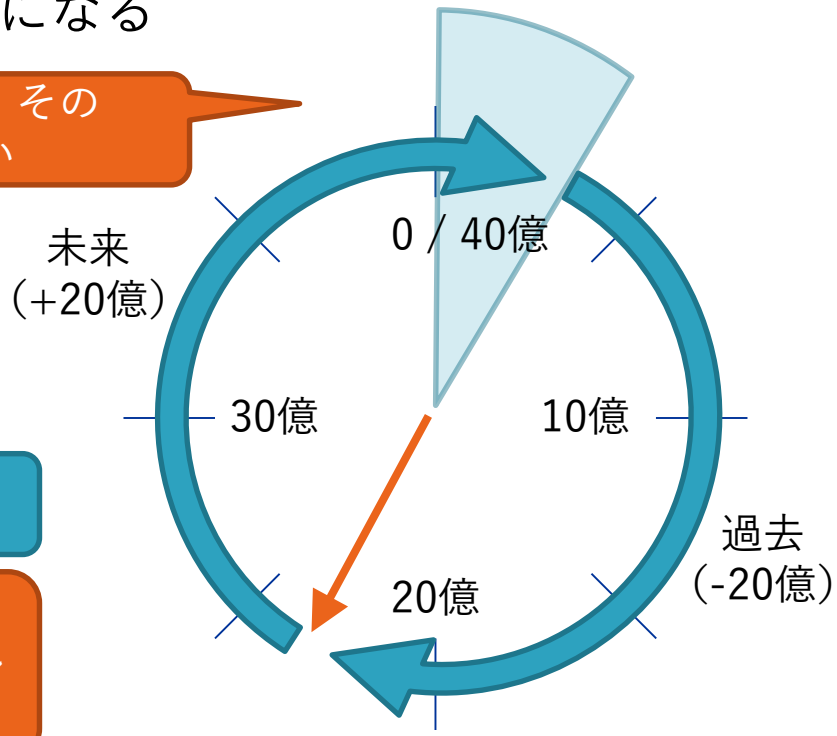


④トランザクションIDの周回問題回避 – Vacuumによる問題回避

■ FREEZE処理で古いXIDを凍結する

- FREEZE処理では、どのトランザクションよりも古いことを示す値（XMINに2）をデータに対して付与する
- FREEZE処理がなされたトランザクションは過去のトランザクションであると見なすことで、新しいトランザクションから見えるようになる

タプルから古いXIDが削除されれば、そのXIDを周回して再利用しても問題ない



タプル	ID	Value	XMIN	XMAX
タプル1	1	AAA	2	-
タプル2	2	BBB	91	101
タプル3	3	CCC	92	103
タプル2'	2	bbb	2	-
タプル3'	3	ccc	103	-

Vacuumで削除されたタプル

既に終了しているトランザクション（XIDが101以下）のXIDを持つタプルのXMINを2へ更新

Vacuumは必須なのか？

① 更新前タプルの削除

- ディスクの逼迫を回避するため、必要

② 空き領域の詰め

- 性能向上に繋がるため、推奨

③ インデックスの均一化

- 性能向上に繋がるため、推奨

④ トランザクションIDの周回問題回避

- FREEZE処理が実行されなければトランザクションがタプルを参照できなくなるため、必須

Vacuum by type

種類から見るVACUUM

Vacuumの種類

- Vacuumは大きく以下の2種類があり、処理内容や目的が異なる

① Vacuum Full

② Standard Vacuum

Vacuum Full

- **テーブルをロックし、再構成を実施する。**
  - 役割
    - 不必要になった（もう参照されない）更新前タプルを削除し空き領域を作る
    - テーブルとインデックスを再編成し、空き領域の解放とインデックスの均一化を行う
    - XIDの周回問題を回避するためFREEZE処理を行う
  - 制約
    - Vacuum実行中はテーブルを排他ロックするため、参照や更新ができなくなる
    - テーブルの再構築に伴い、ディスクにテーブルと同サイズの空き容量が必要となる

Standard Vacuum

Vacuum Fullに変わる、システム利用中に実行できるVacuum

- 役割
  - 不必要になった(もう参照されない)更新前タプルを空き領域にする
  - 空き領域がデータファイルの末尾にある場合のみ、その空き領域を削除しファイルを小さくする
  - インデックスの不均一化を部分的に解消する
  - XIDの周回問題を回避するためFREEZE処理を行う

- 制約
  - 空き領域がデータファイルの末尾にある場合の削除時に限り、テーブルの排他ロックを行う
  - OSから見たディスク使用量が改善されない

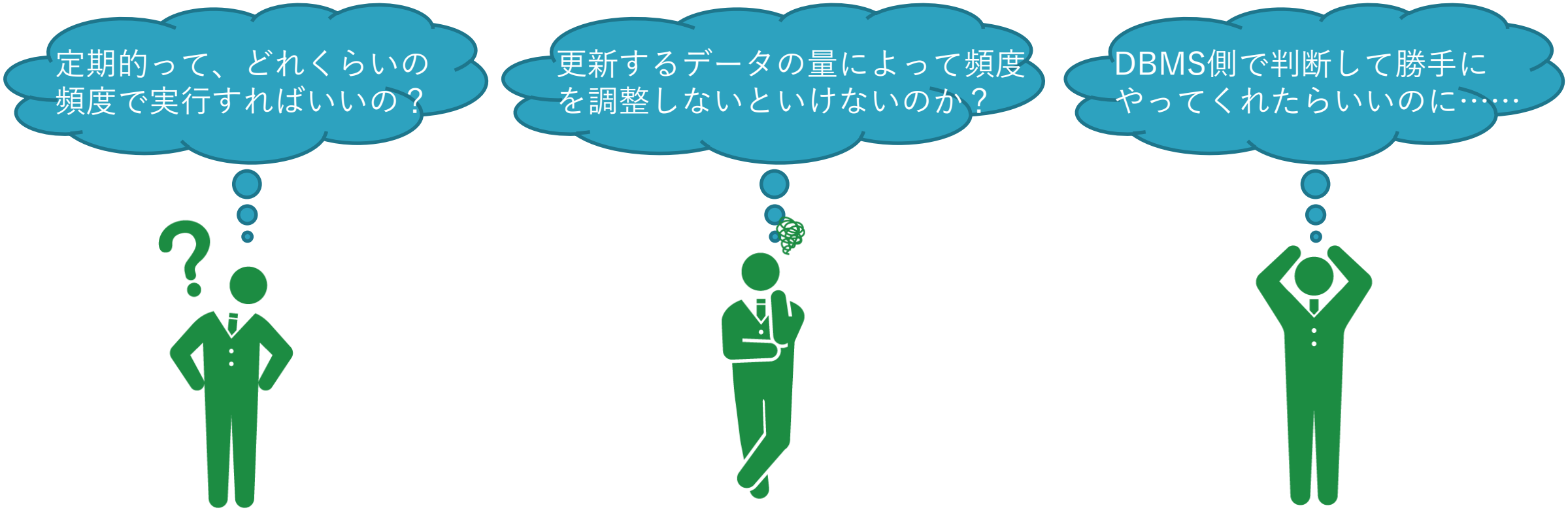
    ※空き領域がデータファイル末尾にある場合は改善される

違い

	Vacuum Full	Standard Vacuum
不要タプル削除後	OSへディスク領域を返却	空き領域として再使用可能
インデックス	再構成によって均一化	均一化は部分的なもの
FREEZE処理	実施される	
ディスクの空き領域	テーブルサイズと同等の空き領域が必要	空き領域は不要
排他ロック	処理完了までロックが発生	切り落とし処理時のみ排他ロックが発生

- **2つのVacuumは異なるものとして個別に実施計画を立てる**
  - Standard Vacuum: オンライン中でも定期的に実行させることが望ましい
  - Vacuum Full: テーブルの排他ロックが発生するため、サービスが停止する。
    必ずしも実行する必要はないが、実行する場合は空きディスク領域、
    業務停止許容時間を確保し計画的に行う必要がある

「Vacuumを定期的に実行するのが望ましい」とは言っても……

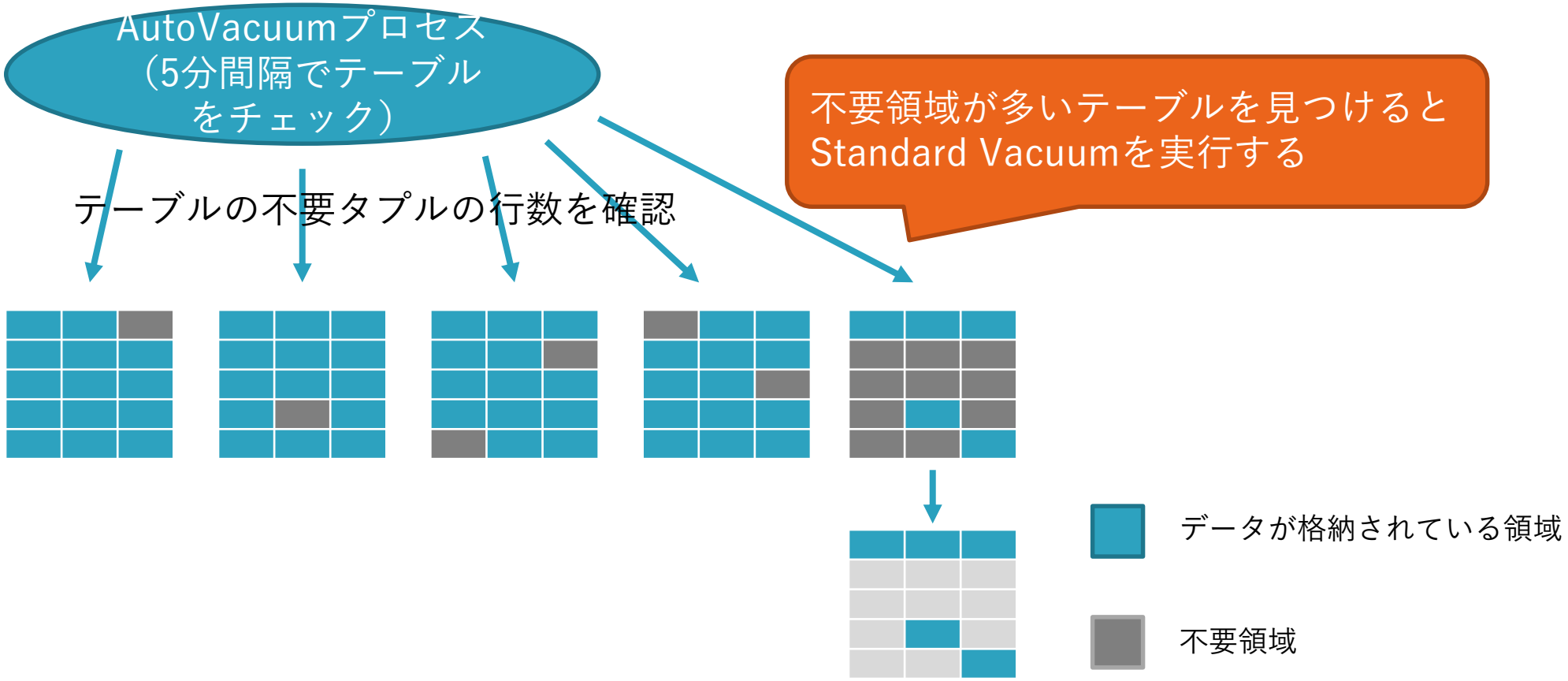


定期実行の課題を解決するのがAutoVacuum

AutoVacuum

■ DBMSが自動判断してVacuumを行う機能

- □ テーブルを監視し、不要領域が一定量を超えると自動でStandard Vacuumを実行する



AutoVacuumのチューニング

- **AutoVacuumがサービスに影響を与えないよう、チューニングが可能**
  - DBMSが自動判断して実行することで、一度に大量の不要領域を処理しようとして大きな負荷を与えかねない
  - パラメータをチューニングすることで、実行頻度やCPUへの負荷を変えることができる

Vacuumの実行頻度を変えたい

最小数と割合の値を小さくすると、より少ない不要領域のタプル数でもVacuumを実行させられる

- autovacuum_vacuum_threshold: Vacuumを実行するか判断する、不要タプルの最小数
- autovacuum_vacuum_scale_factor: Vacuumを実行するか判断する、テーブル全体に対する不要タプルの割合

AutoVacuumがCPUに与える負荷を調整したい

コスト制限は低く、休止時間は長く設定することでCPU負荷を減らすことができる（処理時間は長くなる）

- autovacuum_vacuum_cost_limit: AutoVacuumで一度に処理する処理量（コスト制限）
- autovacuum_vacuum_cost_delay: コスト制限に到達すると発生する休止時間
- autovacuum_max_workers: 同時に実行するAutoVacuumプロセスの最大数

Functions related to Vacuum

VACUUMと関係のある機能

Vacuumと関係のある機能

① コミットログ（Clog）
② 可視性マップ
③ 空き領域マップ
④ HOT（Heap Only Tuple）
⑤ インデックスの削除

①コミットログ（Clog）

- **トランザクションとそのステータスを保持したビットマップ**
  - トランザクションのステータスをClogに書き出し、現在の実行状況を保持する
  - トランザクション実行時、ClogからXIDを取得しどのタプルを参照するか決める
  - Vacuumが行われると、全てのテーブルのタプルで最も古い（FREEZE以外）XMIN以前の部分がClogから削除される

既にCOMMITEDとなった古いXIDは参照されない

XID	ステータス
100	COMMITED
101	COMMITED
102	COMMITED
103	COMMITED
104	IN_PROGRESS
105	IN_PROGRESS
106	IN_PROGRESS
107	IN_PROGRESS
108	IN_PROGRESS

トランザクションが発生するほど、Clogにデータが蓄積され、肥大化する

Vacuumの実行

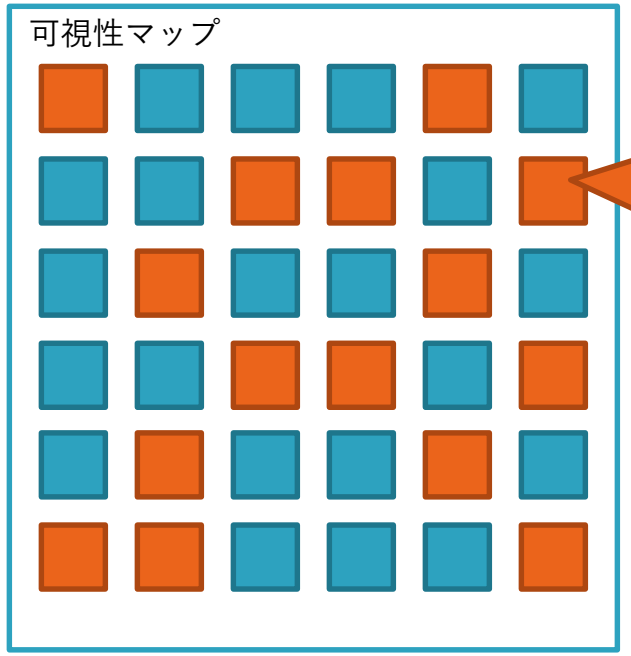
XID	ステータス
100	COMMITED
101	COMMITED
102	COMMITED
103	COMMITED
104	IN_PROGRESS
105	IN_PROGRESS
106	IN_PROGRESS
107	IN_PROGRESS
108	IN_PROGRESS

Vacuumによって参照されない古いデータがClogから削除される
⇒肥大化が解消される

IN_PROGRESS:
トランザクション処理中
COMMITED:
コミット済み
ABORTED:
ロールバック済み
SUB_COMMITED:
SAVEPOINT設定中

②可視性マップ

- **テーブル内のページ毎の参照状態を記録するビットマップテーブル**
  - テーブルのページ毎に1bitのデータ領域を確保し「不要なタプルが存在しない」かつ「どのトランザクションも更新していない」ページは0、そうでないページは1が入る
  - ビットが0の部分には不要タプルが存在しないため、Vacuumは不要。これにより、必要なページのみVacuumの処理対象とすることで、処理を効率化できる



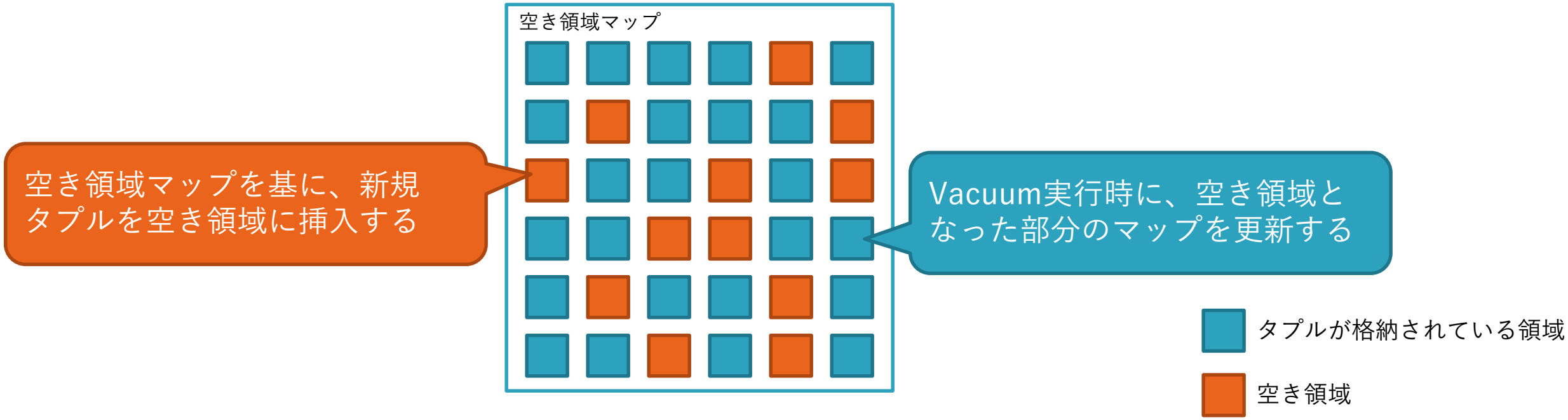
可視性マップをもとに、Vacuumが必要なブロックをピンポイントで処理できる
⇒効率的にVacuumができる

可視性マップで0となっている部分
不要タプルが存在せず、Vacuum処理が不要なページを示す

可視性マップで1となっている部分
更新したタプルが存在し、Vacuum処理が必要なページを示す

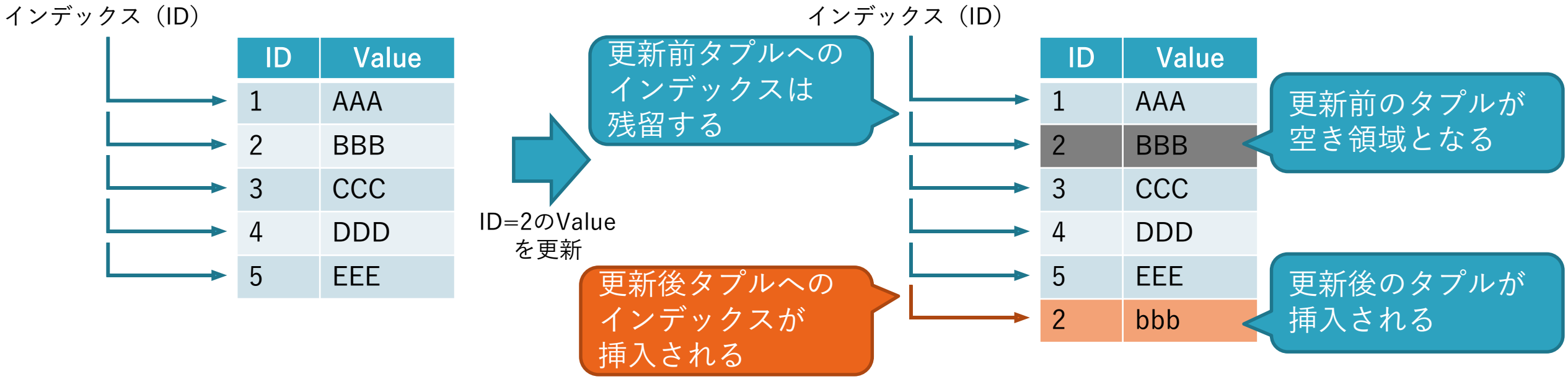
③空き領域マップ

- **インデックスとテーブルが持つ、空き領域を示すマップ**
  - PostgreSQLでタプルを追加する際は、空き領域のあるページを探して追記している
  - 空き領域マップはページ毎にどの程度の空き領域があるかを記録したもの
  - Vacuum実行時に空き領域マップの更新が行われ、タプル挿入の効率化に使用される



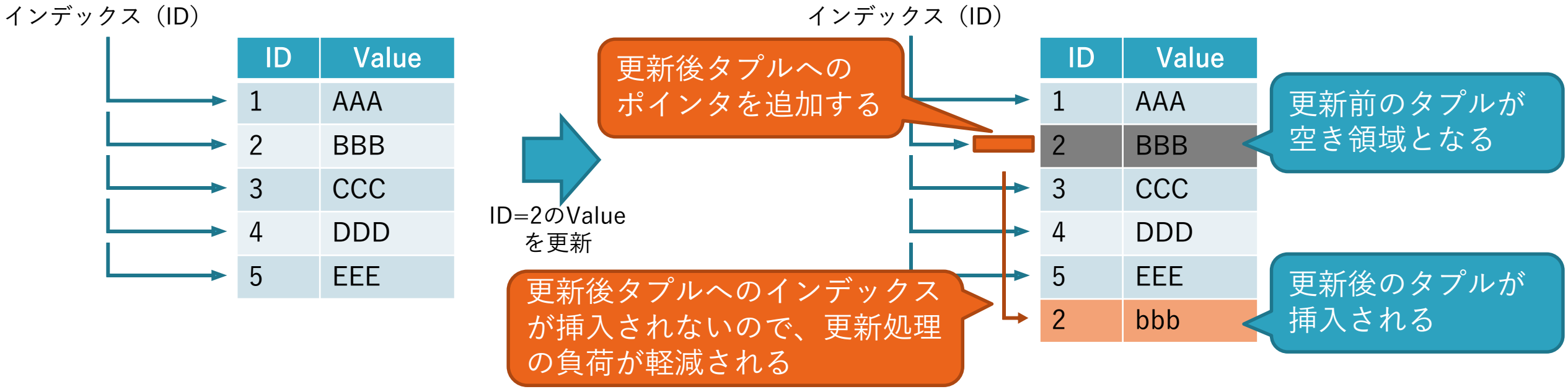
④HOT（Heap Only Tuple）- インデックス更新時の挙動

- **インデックス付きタプルが更新されると、更新先タプルへのポインタを持つ**
  - HOTの実装以前、インデックスを持つタプルが更新されるとデータの位置が変わるため、更新前タプルのインデックスは残したまま更新後タプルへのインデックスが新規に追加されていた
  - そのため、インデックスの値が変更されていない場合もインデックスのエントリが追加されるので更新処理のオーバーヘッドが大きい



④HOT（Heap Only Tuple）- HOTによるアクセス改善

- **ラインポインタで1つのインデックスから更新前後のタプルを参照できる**
  - HOTはラインポインタを変更し、新たなインデックスは追加せず更新前タプルに更新後タプルへのポインタを追加する
  - これによりインデックスを追加せずタプル更新を行うので、更新処理の速度が改善される
  - なお、HOTが機能するケースは更新前後のタプルが同じページに存在し、かつ更新される列がインデックスで参照されていない場合のみ。



⑤インデックスの削除

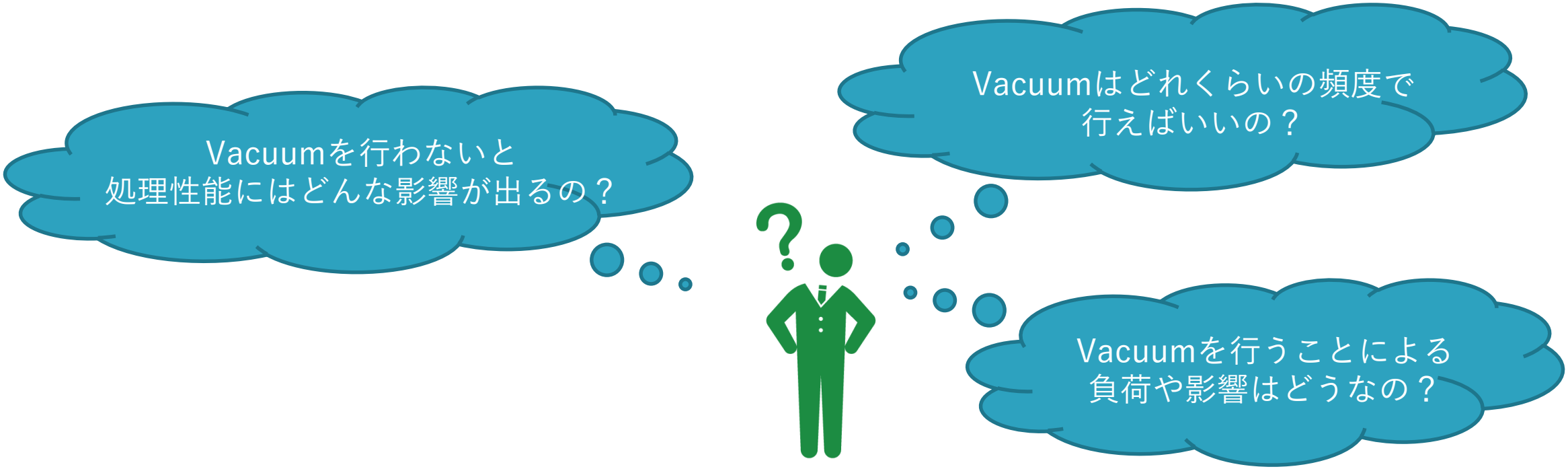
- **不要なインデックスを削除し、アクセス効率を改善する**
  - B-Treeインデックスは更新が続くことで、
    不要な更新前タプルのインデックスエントリが蓄積される
  - PostgreSQLには不要なインデックスを削除しアクセスを効率化させる機能がある
  - 削除のタイミングは複数存在する
    - 単純削除: インデックススキャンが実行されたタイミングで実施される
    - ボトムアップインデックス削除: UPDATEクエリ実行時に実施される
    - トップダウンのインデックス削除（再編成）: Vacuum Fullで実行される

Verification

検証

Vacuumのベストプラクティスを探る

- **定期的にVacuumが行う上で疑問がある**



**実機検証を行いながら、
Vacuumのベストプラクティスとなる設計を探っていく**

サマリ

- **検証①: Standard VacuumとAutoVacuumの処理時間の差を計測**
  - 目的: 手動で行うStandard VacuumとAutoVacuumの処理内容に違いはあるか
  - 結果: 処理時間に差異はなく、内容も同じと推測。ただしAutoVacuum実行時にはAnalyzeが伴う
- **検証②: インデックスの量による処理時間の変化を計測**
  - 目的: テーブルにインデックスが追加されると、Vacuumの処理性能に影響を与えるか
  - 結果: Vacuum対象のインデックスが多いほど、それに比例して処理時間が長くなる
- **検証③: 末尾の空白の切り落とし処理の負荷を計測**
  - 目的: テーブル末尾に意図的に空白領域を作成した際に、Vacuumの処理時間の変化を確認する
  - 結果: 空白領域が大きいほど、それに比例して処理負荷が大きくなる
- **検証④: 不要領域とクエリ処理時間の関係を計測(シーケンシャルスキャン)**
  - 目的: テーブル上に不要領域や空き領域が存在した場合に、シーケンシャルスキャンに与える影響を確認する
  - 結果: 不要領域が大きいほど処理時間が長くなる
- **検証⑤: 不要領域とクエリ処理時間の関係を計測(インデックススキャン)**
  - 目的: テーブルに対し更新を行った際の、インデックススキャンへの影響を確認する
  - 結果: インデックスの改善が機能するため、影響は部分的になる

検証環境①

検証機マシンスペック

- CPUコア数: 4
- メモリ: 16GB
- ディスク読み込み速度: 約75MB/sec
- ディスク書き込み速度: 約60MB/sec
- OS: Ubuntu 20.04.5 LTS
- PostgreSQL: 15.1

検証用テーブル構造

タプル名	データ型	注釈
id	integer	1～1,000,000の数値が格納される主キー列
txt	char(10)	ランダムな10桁の文字列
val	integer	1~65535のランダムな数値

- タプル数: 1,000,000
- テーブルサイズ: 約95MB

検証環境②

コンフィグ（検証に関連する部分のみ抜粋）

#AutoVacuumによる検証を行う場合のみONとする
autovacuum = off

#AutoVacuum・Analyze実行の閾値を最低とする
autovacuum_vacuum_threshold = 1
autovacuum_vacuum_insert_threshold = 1
autovacuum_analyze_threshold = 1
autovacuum_vacuum_scale_factor = 0.0001
autovacuum_vacuum_insert_scale_factor = 0.0001
autovacuum_analyze_scale_factor = 0.0001

#AutoVacuum実行時に待ちやディレイを発生させない
autovacuum_max_workers = 10
autovacuum_vacuum_cost_delay = 0ms

#テーブルとインデックスを共有メモリに十分格納可能な容量を指定
shared_buffers = 1024MB

#クエリ実行時、シーケンシャルスキャンおよびインデックススキャンのみを使用する
enable_async_append = off
enable_bitmapscan = off
enable_gathermerge = off
enable_hashagg = off
enable_hashjoin = off
enable_incremental_sort = off
enable_material = off
enable_memoize = off
enable_mergejoin = off
enable_parallel_append = off
enable_parallel_hash = off
enable_partition_pruning = off
enable_seqscan = off

補足

検証について

- 検証前にはCLUSTERコマンドを使用してテーブルの再構築を行う
  （空き領域の詰めに加えて、主キーのインデックス順にタプルを並べ替える）
- UPDATEコマンドを使用して不要領域を作成する際は、不要領域がテーブル内で均一になるよう更新タプルを指定する
- 計測は5回行い、その平均値を採取する

検証①: Standard VacuumとAutoVacuumでの処理時間 – 概要

目的: Standard VacuumとAutoVacuumで処理内容に差はあるのか、
処理時間からその違いを確認する

手順: UPDATEコマンドで意図的に不要領域を作成し、Standard VacuumとAutoVacuumを実行。
それぞれ処理時間を計測する

id	txt	val
1	AAA	24565
2	BBB	6576
3	CCC	34287
4	DDD	345
…	…	…

UPDATE
による更新

id	txt	val
1	AAA	24565
2	BBB	6576
3	CCC	34287
4	DDD	345
…	…	…
1	aaa	24565
2	bbb	6576
3	ccc	34287
4	ddd	345
…	…	…

処理時間の差を計測する

Standard Vacuum/
AutoVacuum
を実行

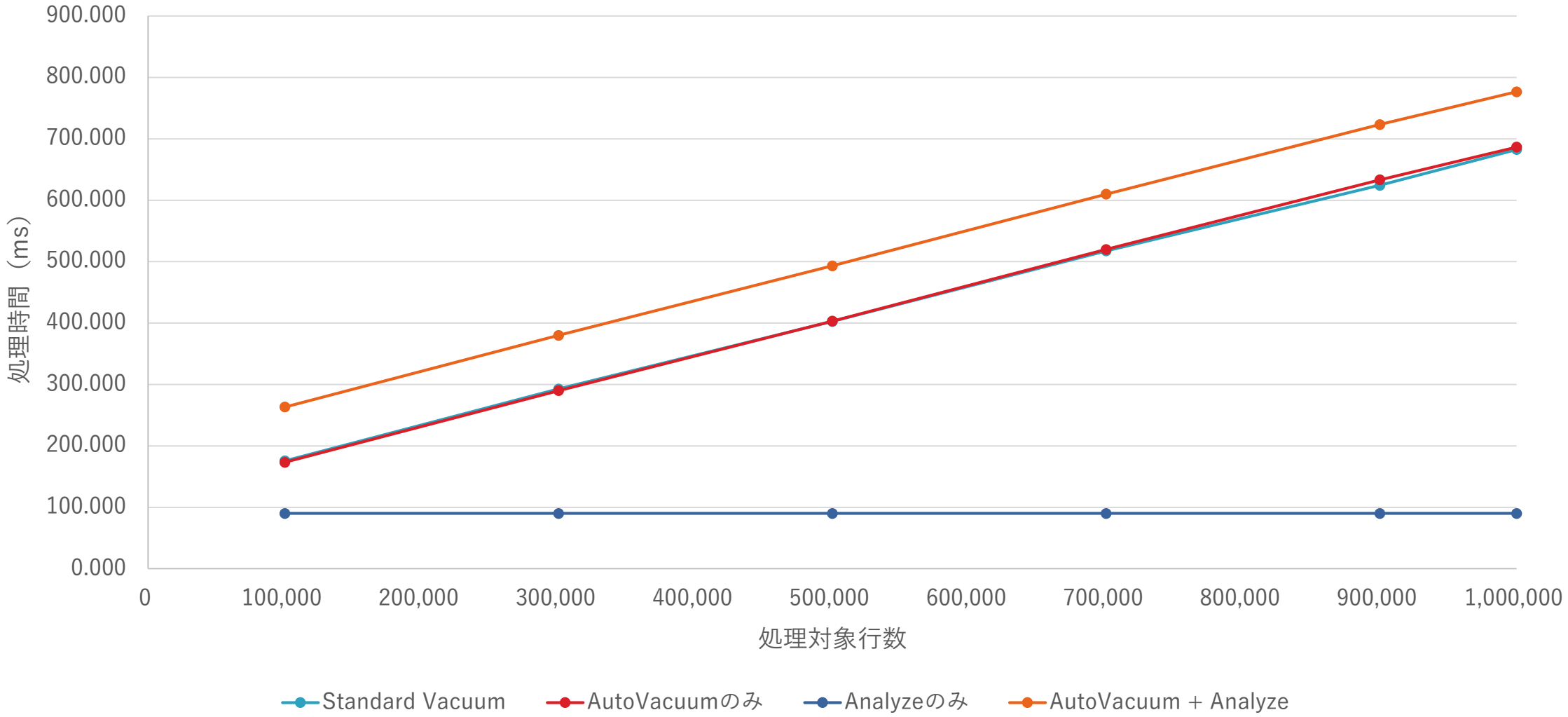
id	txt	val
1	AAA	24565
2	BBB	6576
3	CCC	34287
4	DDD	345
…	…	…
1	aaa	24565
2	bbb	6576
3	ccc	34287
4	ddd	345
…	…	…

空き領域

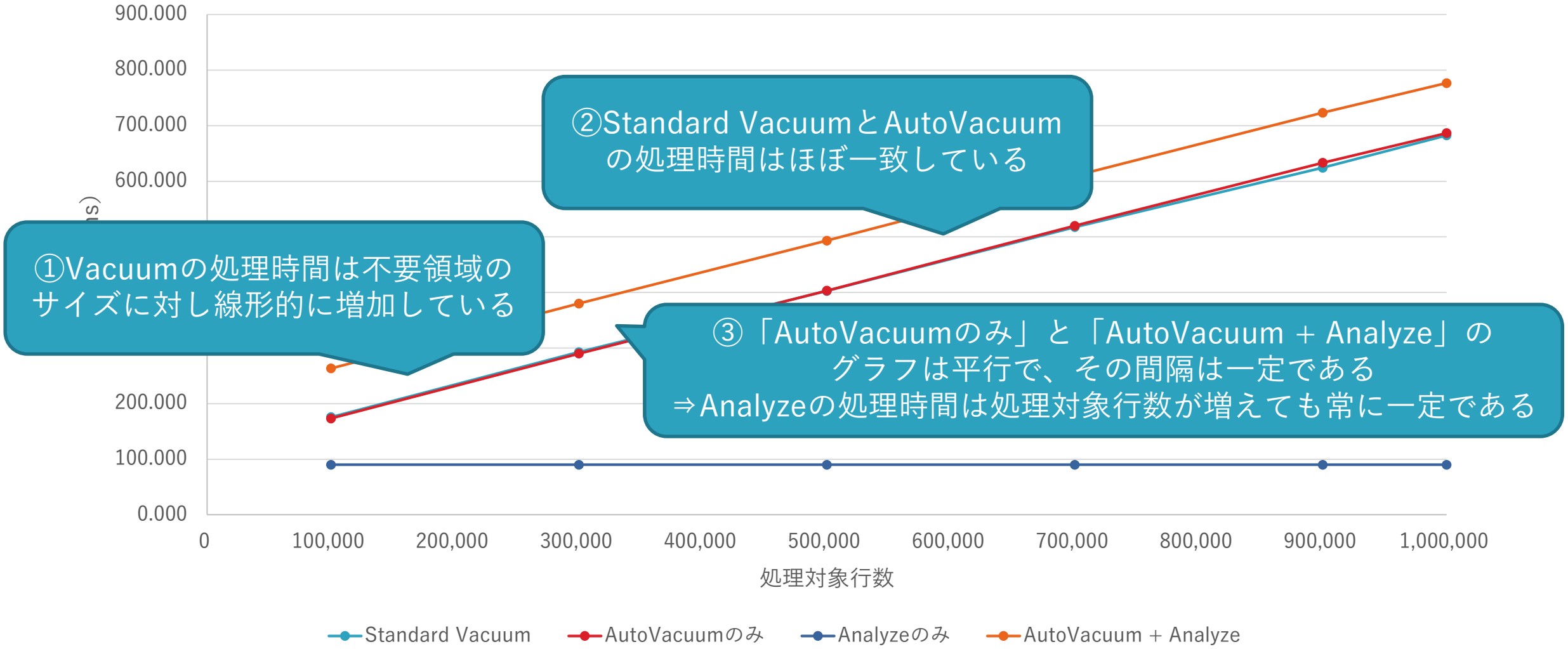
検証①: Standard VacuumとAutoVacuumでの処理時間 - クエリ

```
--テーブルの再編成を行う
CLUSTER 検証テーブル;

--タプルの更新を行う。以下は10万行のタプルを更新する際のもので、テーブル全体を均一に更新する
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN      1 AND   10000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 100001 AND  110000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 200001 AND  210000;
/*中略*/
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 800001 AND  810000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 900001 AND  910000;

--Standard Vacuumを実行し処理時間を計測。AutoVacuumの計測時は自動実行されるまで待機する
VACUUM VERBOSE 検証テーブル ;
```

検証①: Standard VacuumとAutoVacuumでの処理時間 - 結果



検証①: Standard VacuumとAutoVacuumでの処理時間 - 考察



検証①: Standard VacuumとAutoVacuumでの処理時間 - 結論

- Standard VacuumとAutoVacuumの処理時間はほぼ一致しており、
  処理内容の実態に差はないものと思われる

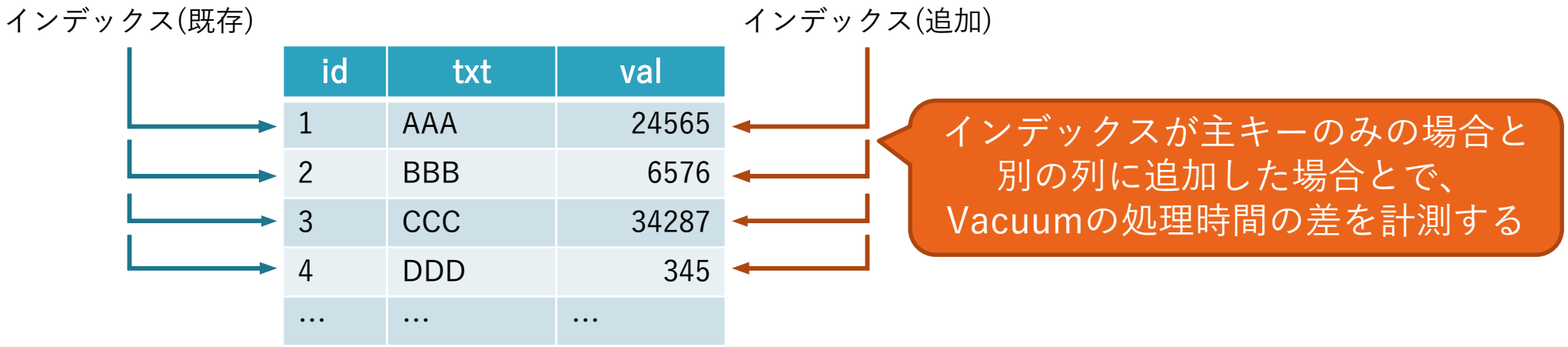
- ただし、AutoVacuumが実行されるとAnalyzeが実行される。
  Analyzeは不要領域の量に関係なく処理時間は一定のため、AutoVacuumの
  処理回数が増えるとAnalyzeによるオーバーヘッドも大きくなってしまう

- なお、検証環境ではAutoVacuumの休止時間が発生しない設定としている。
  負荷が大きくならないよう、AutoVacuumに休止時間を掛けることも可能

検証②: インデックスの量による処理時間変化 – 概要

目的: インデックスが存在することでVacuumの処理時間に変化が生まれるか確認する

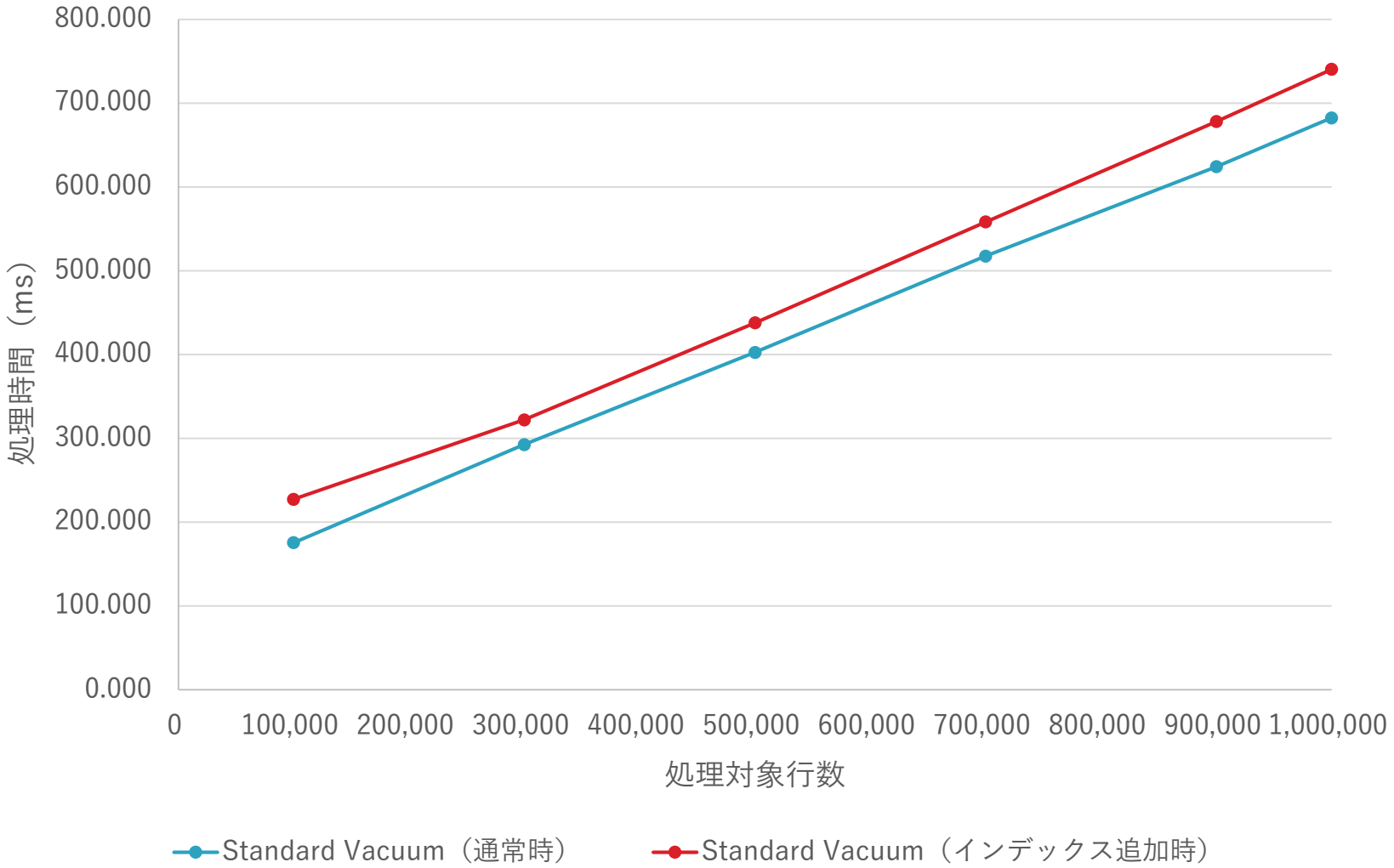
手順: val列にインデックスを新規に設定した状態でUPDATEコマンドを実行し、
意図的に不要領域を作成してVacuumの処理時間を計測する

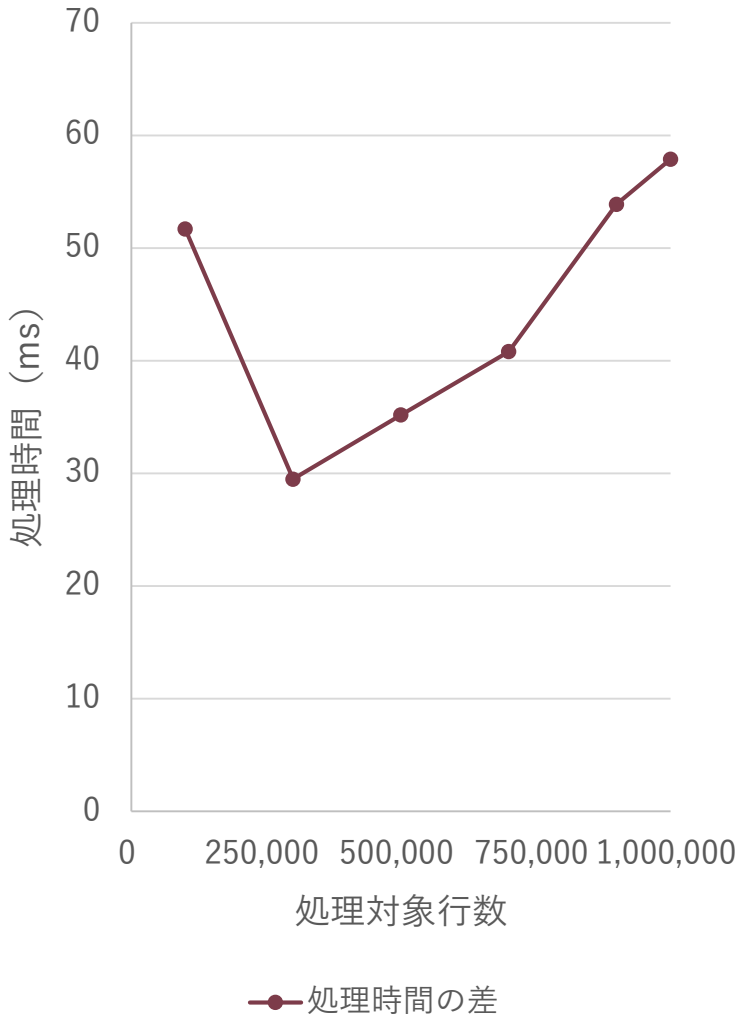


検証②: インデックスの量による処理時間変化 - クエリ

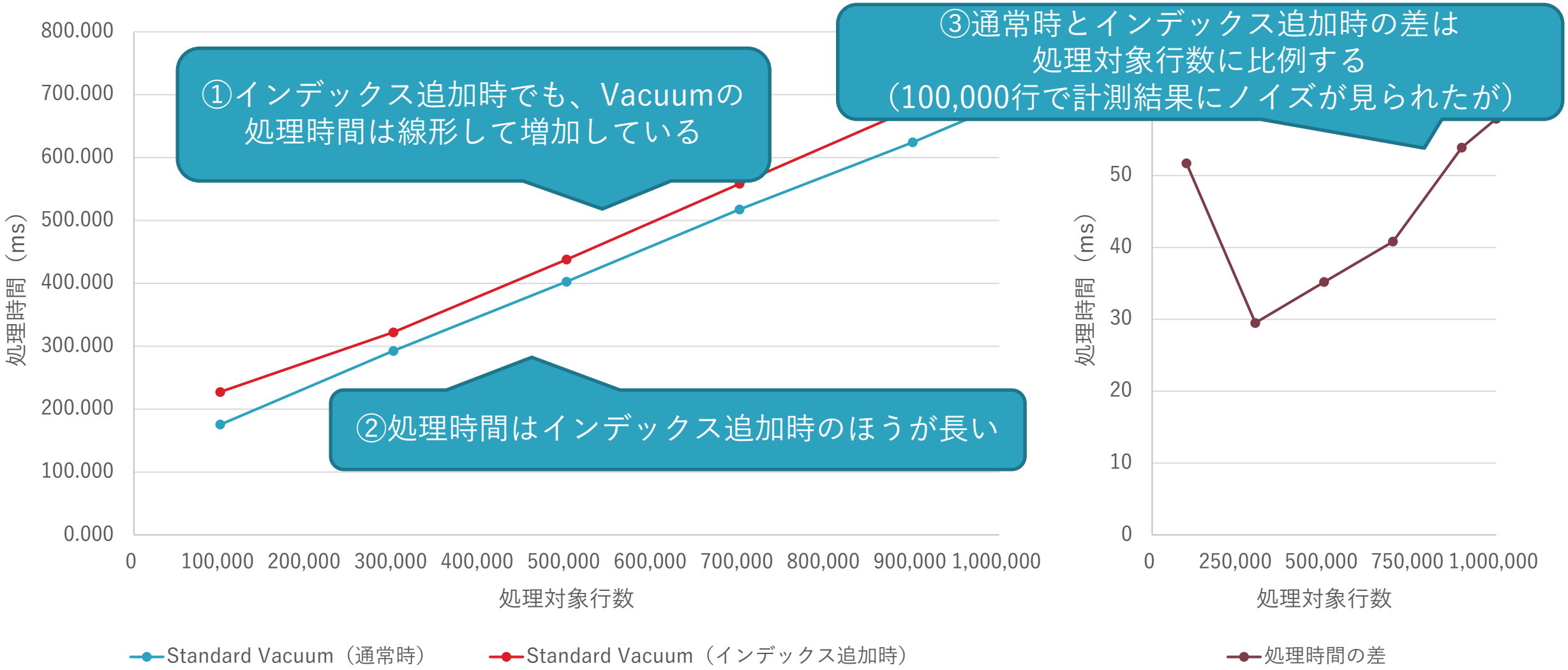
```
--val列にインデックスを設定
CREATE INDEX valkey ON 検証テーブル( val );

--テーブルの再編成を行う
CLUSTER 検証テーブル;

--タプルの更新を行う。以下は10万行のタプルを更新する際のもので、テーブル全体を均一に更新する
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN      1 AND   10000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 100001 AND  110000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 200001 AND  210000;
/*中略*/
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 800001 AND  810000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 900001 AND  910000;

--Standard Vacuumを実行し計測
VACUUM VERBOSE 検証テーブル ;
```

検証②: インデックスの量による処理時間変化 – 結果





検証②: インデックスの量による処理時間変化 – 考察



検証②: インデックスの量による処理時間変化 – 結論

- 検証①のStandard Vacuumの処理時間と比較したところ、インデックスが存在している場合のほうがVacuumの処理時間が長くなっている

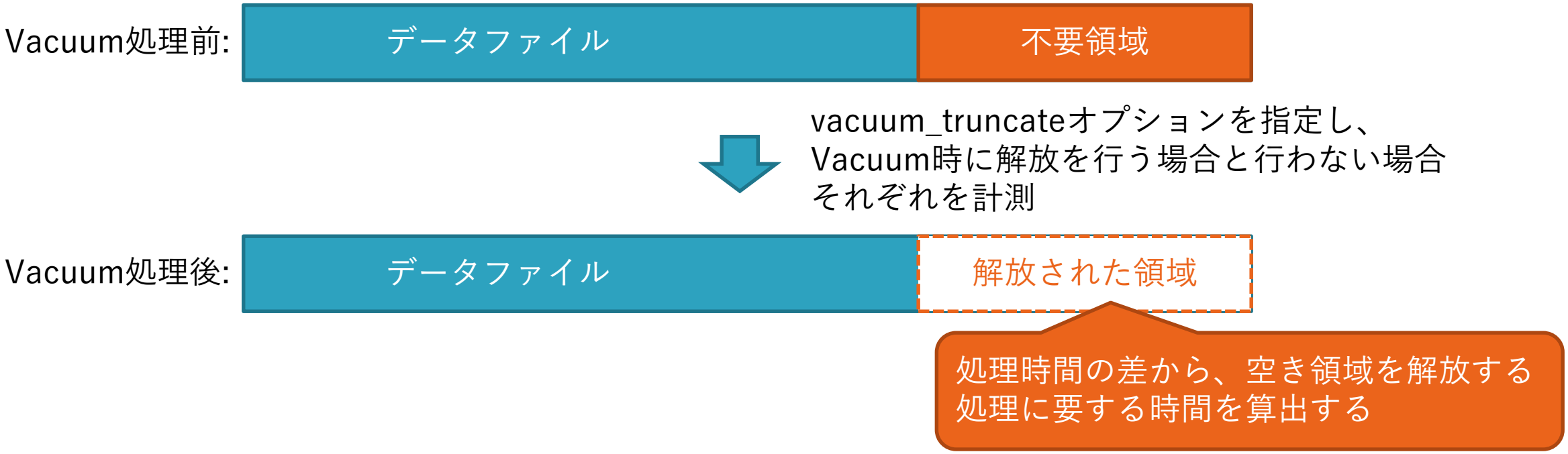
- インデックスを設定する列が多い / 処理行数が多いほど、Vacuumでの処理時間の差は大きくなる

- そのため、不必要にインデックスを設定するとVacuumの処理性能に影響を与える危険性がある

検証③: 末尾の空白の切り落とし処理の負荷 – 概要

目的: Standard vacuumでは、空き領域が末尾に存在するとその部分を切り落とし、ファイルサイズを小さくする操作を行う。その処理負荷の差を確認する

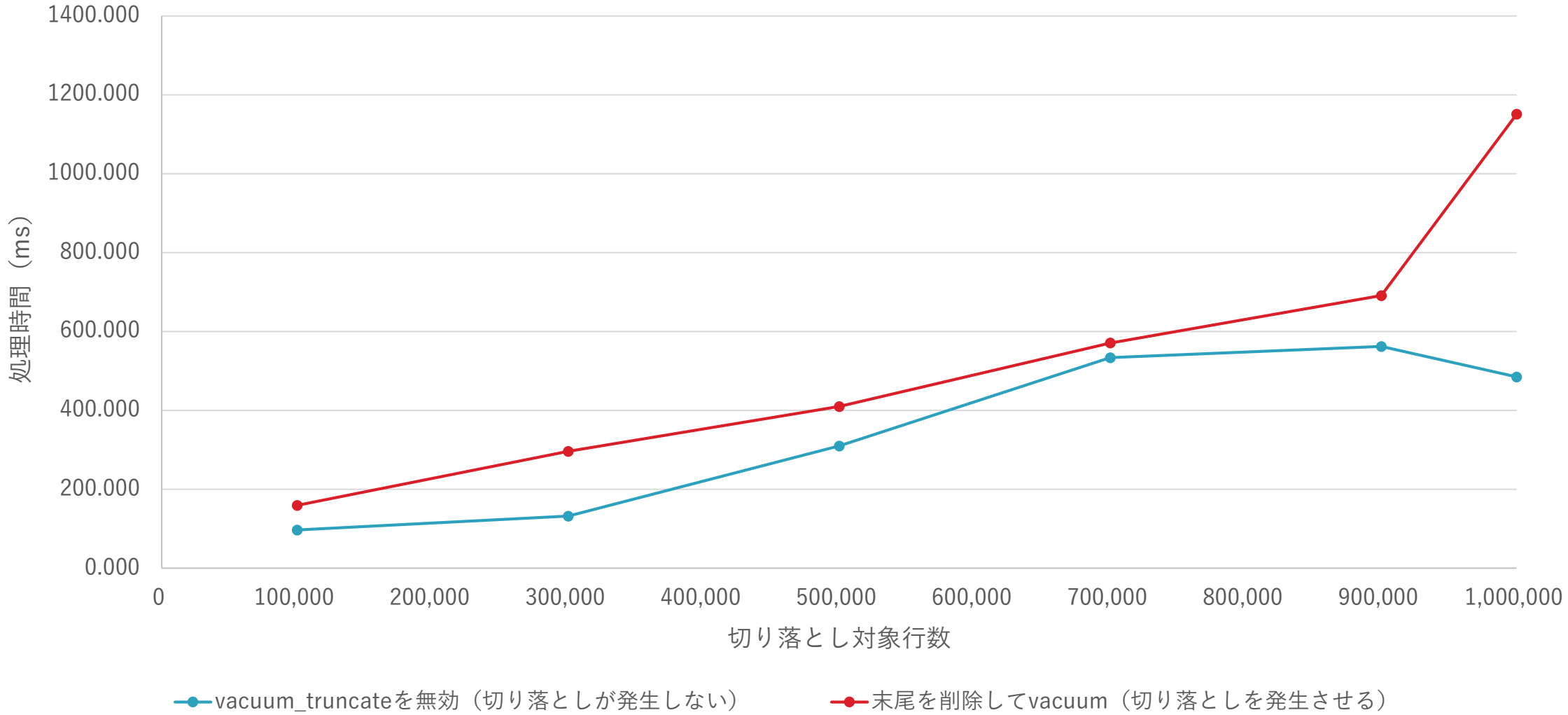
手順: 切り落としの有効/無効を設定するオプション「vacuum_truncate」が存在する。それを利用し、有効な状態と無効な状態それぞれでテーブルの末尾タプルを削除しVacuumを実行。その処理時間の差を計測する



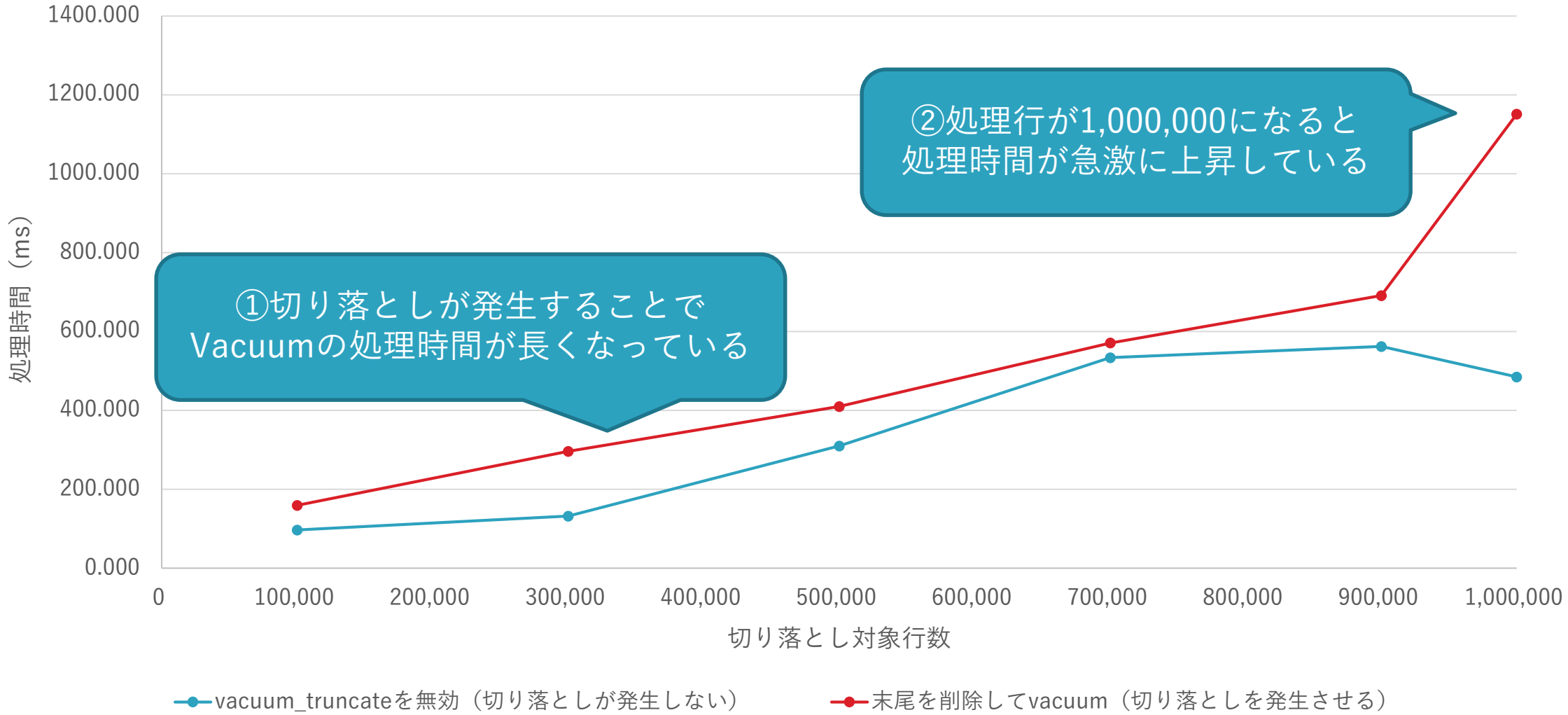
検証③: 末尾の空白の切り落とし処理の負荷 - クエリ

```
--テーブルの内容を全て削除し、CSVファイルからタプルを読み込む
--※CSVファイルの内容は検証①、②でテーブルに格納されている内容と同じ
TRUNCATE TABLE 検証テーブル;
COPY 検証テーブル FROM 'テーブル情報を格納したCSVファイル' WITH CSV HEADER;

--テーブルの再編成を行う
CLUSTER 検証テーブル;

--切り落とし処理を行うかのオプションを変更(行う場合はTRUE、行わない場合はFALSE)
ALTER TABLE 検証テーブル SET (vacuum_truncate = TRUE, toast.vacuum_truncate = TRUE);

--テーブル末尾にあるタプルの削除を行う。以下は10万行のタプルを削除する際のもの
DELETE FROM 検証テーブル WHERE id > 900000;

--Standard Vacuumを実行し計測
VACUUM VERBOSE 検証テーブル ;
```

検証③: 末尾の空白の切り落とし処理の負荷 – 結果



検証③: 末尾の空白の切り落とし処理の負荷 – 考察



検証③: 末尾の空白の切り落とし処理の負荷 – 結論

- 末尾の切り落としが有効な場合と無効な場合を比較した時、全体の処理時間は切り落としが発生する場合のほうが長い。
  この処理時間の差が切り落とし処理の時間であり、テーブルの排他的ロックが発生する時間となる

- また、処理行が1,000,000を超える（削除ディスクサイズが約50MB）と処理時間が急激に上昇しているが、原因はディスクI/Oの処理能力によりiowaitが発生したためと思われる

- このため、末尾の切り落とし発生とその影響はディスク処理性能に依存する。
  切り落とされる領域が小さいうちにVacuumを行うことで、ロック時間を小さくすることができる

検証④: クエリ処理時間について(シーケンシャルスキャン) - 概要

目的: 不要領域、空き領域が存在すると、シーケンシャルスキャンを伴うクエリにどのような影響を与えるか確認する

手順: 意図的に不要領域を作成した状態およびStandard Vacuumを実行した直後で
シーケンシャルスキャンを伴うクエリを実行し、その処理時間の差を計測する

id	txt	val
1	AAA	24565
2	BBB	6576
3	CCC	34287
4	DDD	345
…	…	…

UPDATE
による更新

id	txt	val
1	AAA	24565
2	BBB	6576
3	CCC	34287
4	DDD	345
…	…	…
1	aaa	
2	bbb	
3	ccc	
4	ddd	345
…	…	…

不要領域が存在する状態
でクエリを実行する

Standard Vacuum
を実行

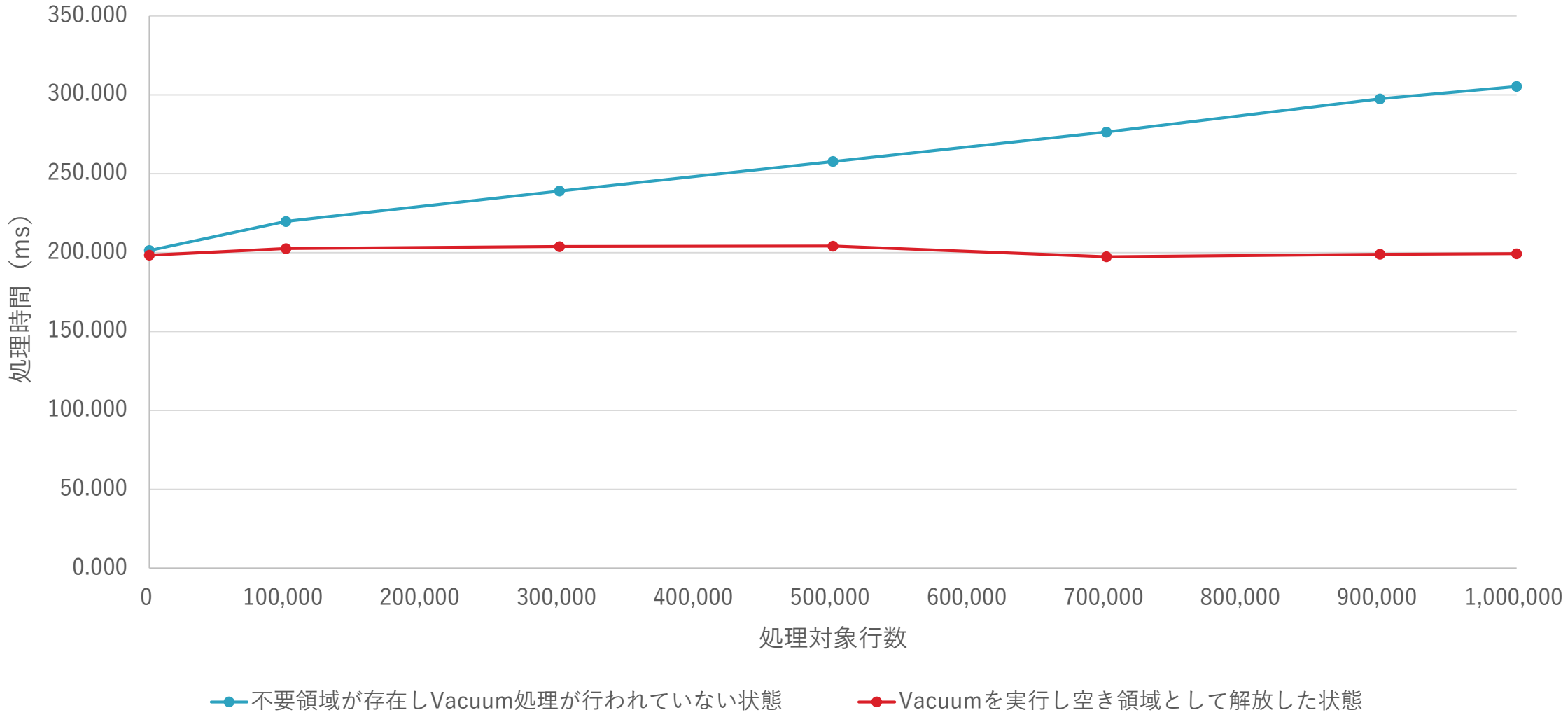
id	txt	val
1	aaa	24565
2	bbb	6576
3	ccc	34287
4	ddd	345
…	…	…

不要領域が空き領域に
変換された状態で
クエリを実行する

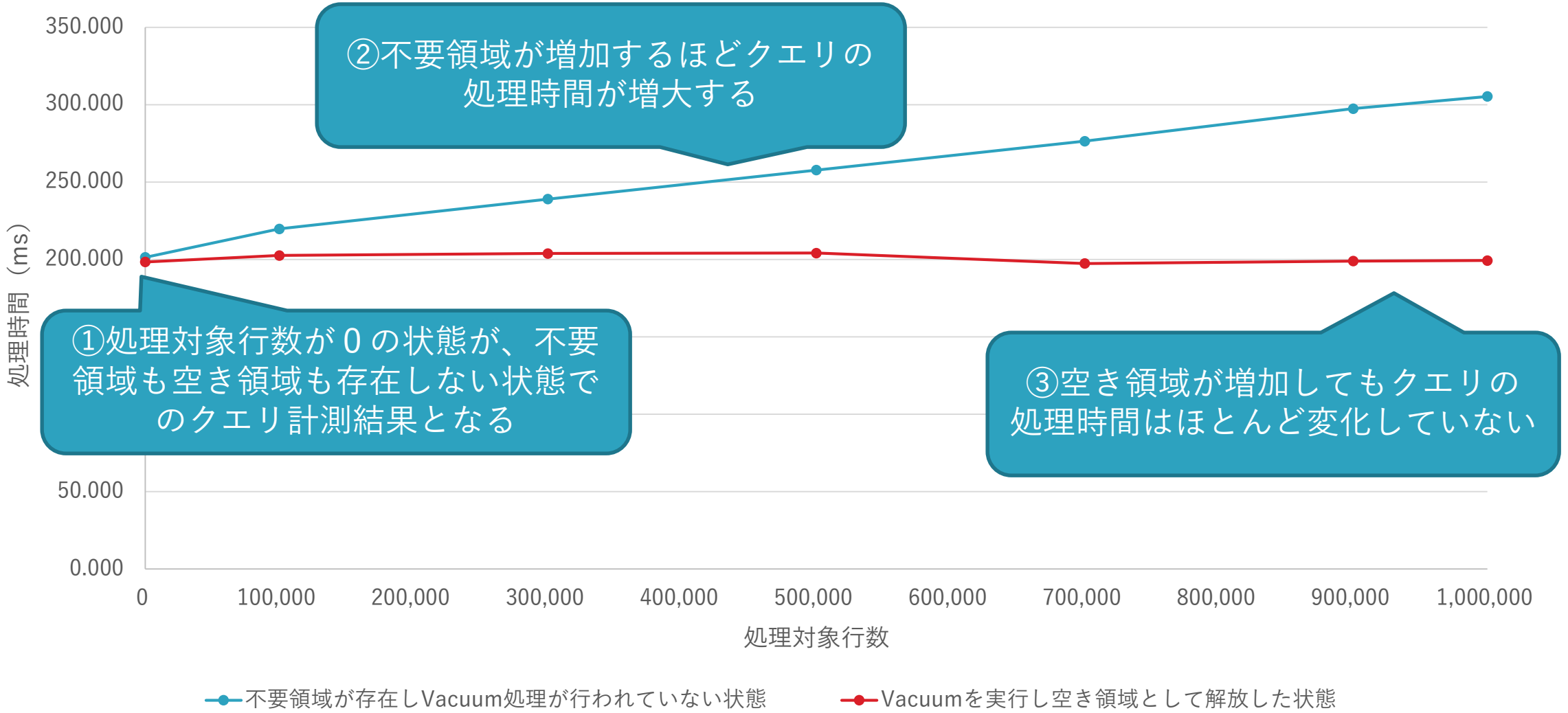
検証④: クエリ処理時間について(シーケンシャルスキャン)- クエリ

```
--テーブルの再編成を行う
CLUSTER 検証テーブル;

--タプルの更新を行う。以下は10万行のタプルを更新する際のもので、テーブル全体を均一に更新する
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN      1 AND   10000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 100001 AND  110000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 200001 AND  210000;
/*中略*/
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 800001 AND  810000;
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 900001 AND  910000;

--空き領域が存在することによる影響を計測する場合のみ、Standard Vacuumを実行
--不要領域が存在することによる影響を計測する場合は実行しない
VACUUM 検証テーブル

--シーケンシャルスキャンが発生するクエリを実行し、処理時間を計測
EXPLAIN (analyze , buffers ) SELECT SUM(val) FROM 検証テーブル ;
```

検証④: クエリ処理時間について(シーケンシャルスキャン) - 結果



検証④: クエリ処理時間について(シーケンシャルスキャン) – 考察



検証④: クエリ処理時間について(シーケンシャルスキャン) - 結論

- Vacuumによる処理がなされていない不要領域が含まれていると、
  シーケンシャルスキャンの読み込み対象に不要領域も含まれてしまい、
  処理時間が長くなる

- Vacuumにより不要領域が空き領域に変わると、
  シーケンシャルスキャンの対象ではなくなり処理時間も改善される

- タプル更新が多く、かつシーケンシャルスキャンが多様される環境であれば、
  Vacuumを頻繁に行うことで処理性能の劣化を防ぐことができる

検証⑤: クエリ処理時間について(インデックススキャン) – 概要

目的: 複数回更新が発生したタプルに対してのインデックススキャンが、どのような影響を受けるか確認する

手順: 同じタプルに対して複数回UPDATEを実行し、その後そのタプルに対してインデックススキャンを行う。UPDATEの回数が変化することで処理時間がどのように変化するかを確認する

id	txt	val
1	AAA	24565
2	BBB	6576
3	CCC	34287
4	DDD	345
…	…	…

UPDATE テーブル SET txt = '変更後文字列' WHERE id BETWEEN 1 AND 90000;

SELECT COUNT(*) FROM テーブル WHERE id BETWEEN 1 AND 90000;

セットで複数回上記クエリを実行し、回数を重ねるごとのSELECT文の処理時間変化を計測する

検証⑤: クエリ処理時間について(インデックススキャン) - クエリ

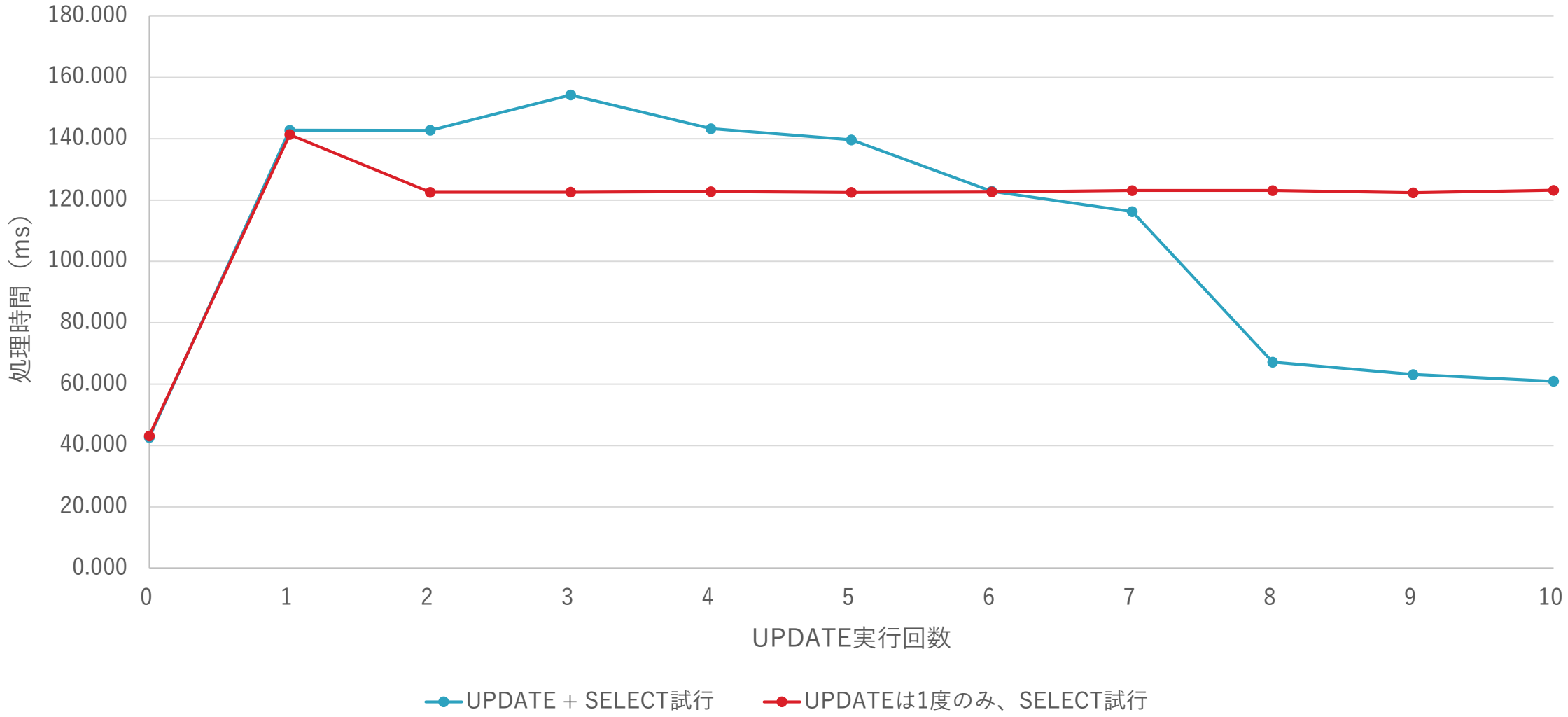
```
--テーブルの再編成を行う
CLUSTER 検証テーブル;

--インデックススキャンが発生するクエリを実行し、更新やVACUUMが行われていない状態(更新0回目)の処理時間を計測
EXPLAIN (analyze , buffers ) SELECT SUM(val) FROM 検証テーブル WHERE id BETWEEN 1 and 100000 ;

--【UPDATE + SELECT試行の場合】
--idが1～100000のタプルを更新する
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 1 AND 100000;
--インデックススキャンが発生するクエリを実行し、処理時間を計測
EXPLAIN (analyze , buffers ) SELECT SUM(val) FROM 検証テーブル WHERE id BETWEEN 1 and 100000 ;

--【UPDATEは1度のみ、SELECT試行の場合】
--idが1～100000のタプルを更新する
UPDATE 検証テーブル SET txt = '更新後文字列' WHERE id BETWEEN 1 AND 100000;
--インデックススキャンが発生するクエリを実行し、処理時間を計測
EXPLAIN (analyze , buffers ) SELECT SUM(val) FROM 検証テーブル WHERE id BETWEEN 1 and 100000 ;
```

UPDATEとSELECTを複数回実施し、それぞれのSELECTの実行結果を計測

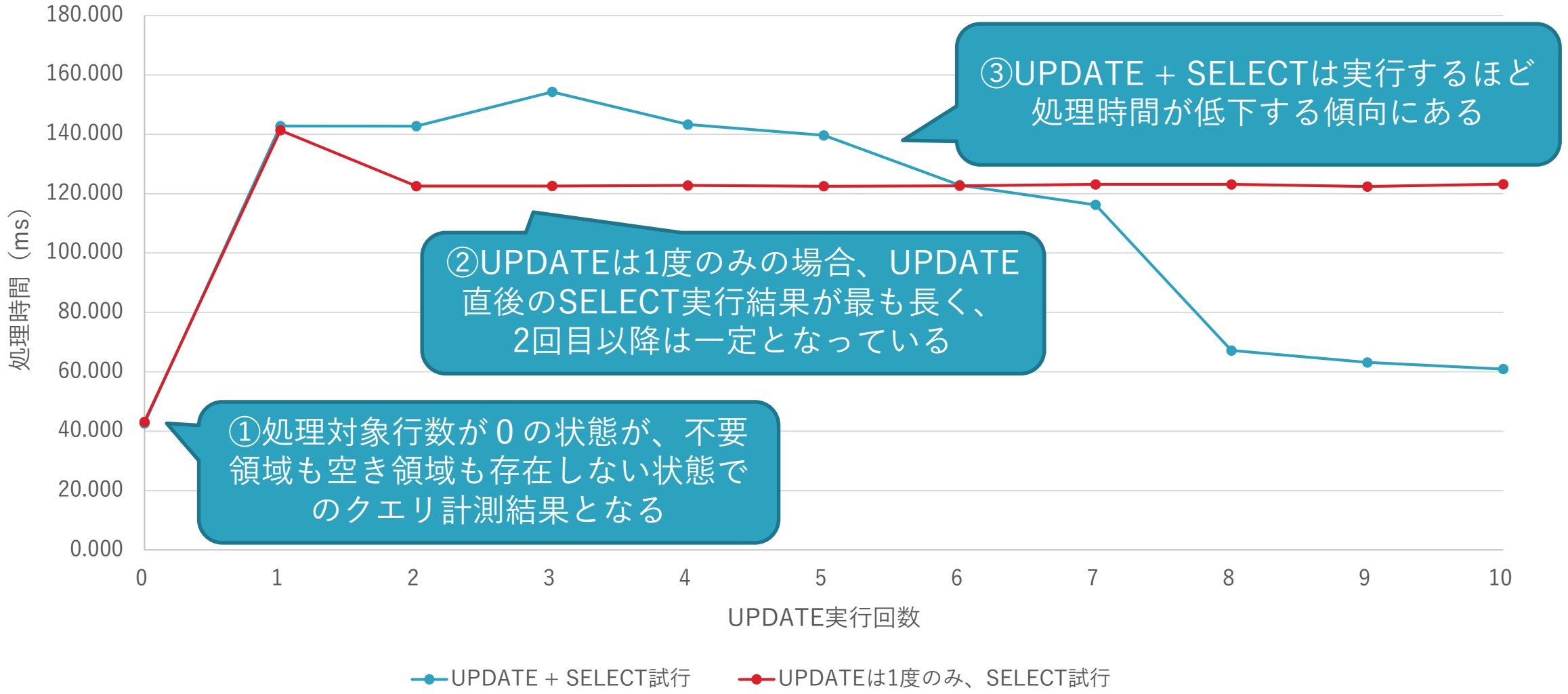
UPDATEの実行は初回のみ

SELECTを複数回実行し計測

検証⑤: クエリ処理時間について（インデックススキャン） – 結果



検証⑤: クエリ処理時間について（インデックススキャン）– 考察



検証⑤: クエリ処理時間について(インデックススキャン) – 結論

- 回数を重ねるほど処理速度が向上する要因について、
  不要になったインデックスが削除されたことによるものと思われる

- UPDATEクエリによる更新が完了し不要となったタプルを指すインデックスは、次のクエリ実行時に削除が行われる。
  複数回UPDATEとSELECTを行うことで不要なインデックスが削除されたものと予想

  ※ 詳細は「P34. ⑤インデックスの削除」を参照

- このため、Vaccumが行われなくてもある程度はインデックスの改善が働く。
  それにより処理速度が際限なく低下することはなく、影響は小さい

Summary

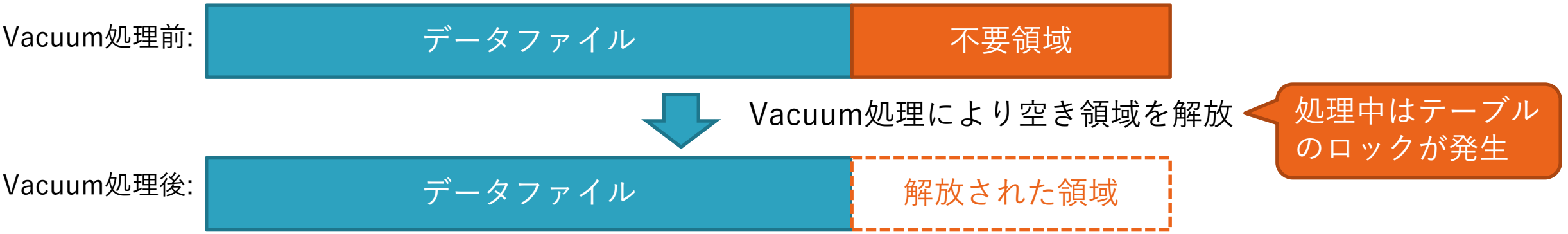
まとめ

事例 – Vacuumによるトラブル

■ とある企業でVacuumによるトラブルが発生

- あるデータベースにてクエリのレスポンス遅延が発生。遅延は47秒間継続し、遅延発生中に合計55件ものクエリでタイムアウトが発生した
- レスポンス遅延が発生した47秒は、AutoVacuumによる**切り落とし処理**に要した時間。
  切り落とし処理中はテーブルがロックされるため、クエリがロック解放待ちとなってしまった
- Vacuum実行時の負荷は考慮していたが、切り落とし処理のロックまでは考慮されていなかった。
  それが今回のトラブル発生の原因なのだが……

切り落とし処理については、検証③を参照



レスポンス遅延発生の“真因”はVacuumへの理解不足であった

事例原因解説

- **障害の原因は、AutoVacuumで実行された切り落とし処理によるもの**
  - レスポンス遅延が発生した47秒は、AutoVacuumによる切り落とし処理に要した時間であった。切り落とし処理の間はテーブルがロックされるため、切り落とし処理中のクエリが実行できず、処理遅延となった
  - データベース構築当時、切り落とし時のロック時間については考慮されていなかった
  - それまでは切り落とされる行数が少なかったこともあり影響が出ていなかったが、今回は偶然にも切り落とされる行数が多くなってしまい、長時間のロックが発生することとなった
  - 対策としては、次のような方法で大量の切り落とし発生を回避することが挙げられる
    - Vacuumの実施間隔を短くし、切り落とし行数が少ないうちに実行させる
    - ピーク時間やバッチ処理中に切り落としが発生しないよう、その時間帯でvacuum_truncateを一時的に無効化する
    - 大量の更新処理や削除処理を行った後にVacuumを実行する

総括

- Vacuumの機能を確認した結果、Vacuumは定期的に実行しなければならないことが分かる

- しかし検証結果から、一度に大量の不要領域を処理しようとするとサービスへ影響を与えかねないことが分かる。
  そのため、処理する領域が少なくなるようにこまめに実行するのが望ましい

- ただし、 AutoVacuumがバッチ処理中や日中のピーク時間に実行されると、少ない行数を処理する場合でもサービスに影響を与える危険性がある

ワークロードに応じてサービスに影響を与えない範囲で
定期的にVacuumを行うよう、チューニングを行う必要がある

PostgreSQL Enterprise Consortium

POSTGRESQL ENTERPRISE CONSORTIUM

Supplementary material

補足資料

Vacuum各種コマンド

- **Vacuumの手動運用を行う上で使用するコマンドを列挙**

VACUUM テーブル名;	Standard Vacuumを実行する
VACUUM FULL テーブル名;	Vacuum Fullを実行し、テーブルを再構築する
VACUUM FULL テーブル名 USING インデックス名;	テーブルの再構築時、インデックスに基づいてテーブルをクラスタ化する
pg-repack -t テーブル名	テーブルの再構築を行うが、排他ロックの時間を最小限にする。ただし空きディスクの容量はテーブルサイズの2倍必要になることに注意
REINDEX インデックス名; REINDEX テーブル名;	指定したインデックスを再構成する。テーブルを指定した場合はそのテーブルに含まれるすべてのインデックスを再構成する

AutoVacuum コンフィグパラメータ

- **AutoVacuumのパラメータを変更することで、処理負荷の平準化が可能**

autovacuum_max_workers	同時に実行することができるautovacuum worker のプロセス数。 数値を多く設定するほど同時に実行されるプロセスが多くなる。 デフォルトは3
autovacuum_vacuum_cost_delay	autovacuumによるVACUUM処理のsleep時間。数値を多く設定 するほどディレイ時間が長くなり、時間当たりの負荷が減少する。 デフォルトは20ms
autovacuum_vacuum_cost_limit	AutoVacuumに使用されるコストの限界値。数値を大きくする ほど高負荷でもプロセスをスリープさせずに処理を行う。 デフォルトは-1（vacuum_cost_limitを参照する）
autovacuum_vacuum_threshold	ANALYZEを起動するのに必要な、挿入、更新、もしくは 削除されたタプルの最小数
autovacuum_vacuum_scale_factor	VACUUMを起動するか否かを決定する時、 autovacuum_vacuum_thresholdに足し算するテーブル容量の割合

テーブル別パラメータ

- **テーブルごとにVacuumの設定が可能**

ALTER TABLE テーブル名 SET (autovacuum_enabled = false); ALTER TABLE テーブル名 SET (toast.autovacuum_enabled = false);	AutoVacuumの実行対象に含めない
ALTER TABLE テーブル名 SET (vacuum_truncate = false); ALTER TABLE テーブル名 SET (toast.vacuum_truncate = false);	Vacuumによる切り落とし処理を無効化する。適用はVacuumとAutoVacuumの両方に適用される

参考文献

- 鈴木啓修.PostgreSQL 全機能バイブル.株式会社技術評論社,2012

- (著作)The PostgreSQL Global Development Group /(翻訳)日本PostgreSQLユーザ会 PostgreSQL 15.4文書. "https://www.postgresql.jp/document/15/html/"

- 日本PostgreSQLユーザ会.Let's POSTGRES. "https://lets.postgresql.jp/"