PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#3

2017年度WG3活動報告書
性能トラブル調査編

目次

1. ライセンス

2. はじめに

2.1. PostgreSQLエンタープライズコンソーシアムとWG3について

PostgreSQLエンタープライズコンソーシアム(略称 PGECons)は、PostgreSQL本体および各種ツールの情報収集と提供、整備などの活動を通じて、ミッションクリティカル性の高いエンタープライズ領域へのPostgreSQLの普及を推進することを目的として設立された団体です。

PGECons 技術部会ではPostgreSQLの普及に資する課題を活動テーマとし、3つのワーキンググループで具体的な活動を行っています。

- WG1（新技術検証ワーキンググループ）
- WG2（移行ワーキンググループ）
- WG3（課題検討ワーキンググループ）

これら3つのワーキンググループのうち、WG1、WG3については 2015 年度まではそれぞれ、「性能ワーキンググループ」、「設計運用ワーキンググループ」という名称で活動してきました。2016年度は、従来の活動領域を広げる意図のもとでそれらを再定義し、上記のような名称に改めました。

これに伴い、WG3ではPostgreSQLの設計運用を中心としたさまざまな課題の解決のための調査検証を行い、PostgreSQLが広く活用される事を推進していくこととしました。

2.2. 本資料の概要と目的

本資料はWG3の2017年度の活動として PostgreSQL における性能トラブルについて、調査・検討した結果をまとめたものです。

性能トラブルに対して、『予防』『検知』『対処』の3つの観点で調査を実施しています。

まず『予防』の観点として性能トラブルを発生させないため設計時に考慮すべきポイントや性能トラブルを発生させないために必要となるノウハウを紹介しています。また『検知』の観点から、正常稼働中から取得しておくべき、性能状態を把握するために必要な監視情報や性能トラブルが発生した時の各監視情報での調査観点も紹介しています。最後に『対処』の観点では過去の性能トラブル事例をもとに、取得情報を使用した調査方法を紹介しています。

以上の内容を本資料にまとめる事で今後のPostgreSQL運用において、性能トラブル対策の一助となる情報を提供することを目的としています。

2.3. 本資料の構成

- はじめに
- 性能トラブルを予防するために考慮すべきデータベース設計のポイント
    - 本章の記載内容、方針について
    - 論理設計
    - 物理設計
    - 運用の設計
    - その他
- 性能状態を把握するための監視
    - 調査検証時に利用した環境
    - 性能情報の種別
    - 一定時間内に実行された処理の結果
    - 任意の時間帯の処理実行状況
    - リソースの利用状況
    - データベース性能に影響を与える要素
- ケーススタディ
    - ロングトランザクションによる性能トラブル
    - ディスク性能の考慮漏れによる性能トラブル
    - 適切でない実行計画が選択されてしまうことによる性能トラブル
- 著者

2.4. 想定読者

本資料の読者は以下のような知識を有していることを想定しています。

- DBMSを操作してデータベースの構築、保守、運用を行うDBAの知識
- PostgreSQLを利用する上での基礎的な知識

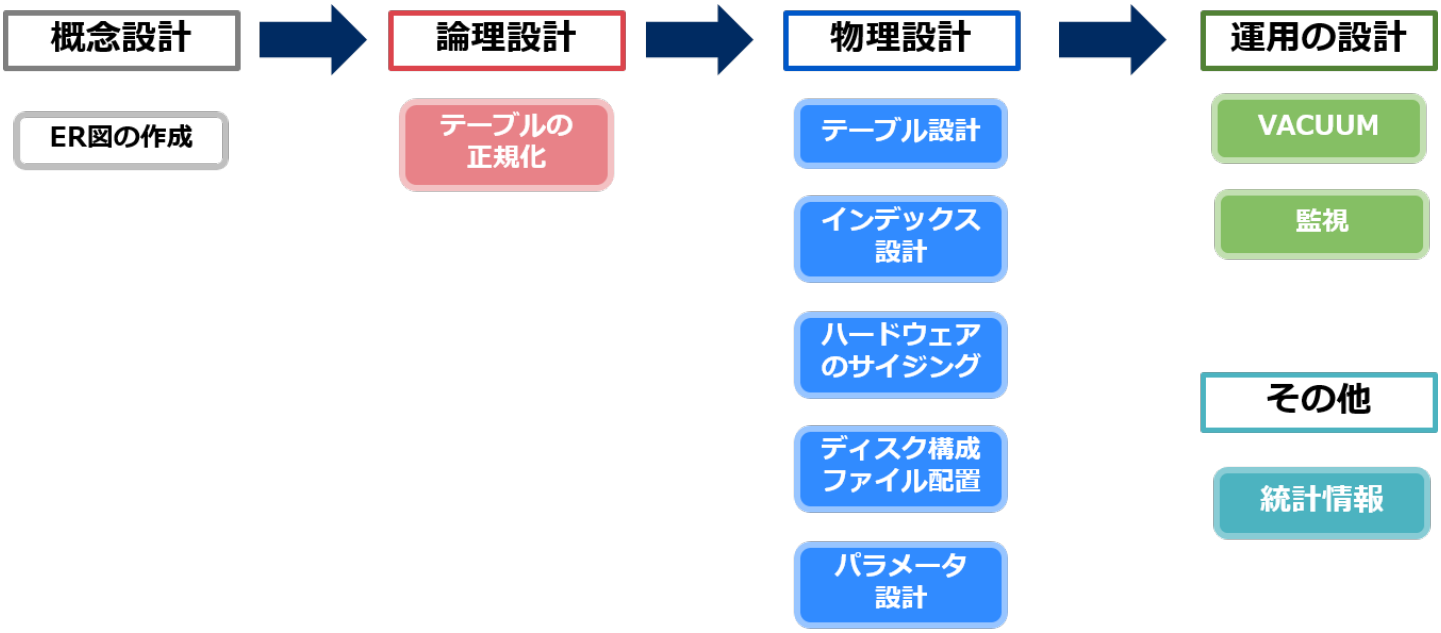
3. 性能トラブルを予防するために考慮すべきデータベース設計のポイント

3.1. 本章の記載内容、方針について

PostgreSQLでデータベースを新たに設計していくにあたり 性能トラブルとして分類される事例や報告をもとにして、トラブルの予防に役立つ対策となるようなポイントをまとめました。

本章ではデータベース設計の流れとして以下のデータベース設計の記事 [1] を参考とし、概念設計、論理設計、物理設計の3つの段階について 性能との関連が深くなる論理設計、物理設計の2つを取り上げます。

その他、システムが実稼働した後のために考慮しておくべき運用管理の部分についても運用の設計と定義し、以下の図に沿った流れで検討項目をピックアップしポイントを記載していきます。



各工程や検討項目については参考となる資料が公開されているものも多いため ここでの詳細な解説は割愛し、概要と参考情報のリンクを付与する形で紹介していきます。

参考情報

注釈	タイトル	URL	概要
[1]	初めてのデータベース設計　第1回　データベース設計とは	http://gihyo.jp/dev/feature/01/database/0001	概念設計、論理設計、物理設計の3つの段階について紹介

3.2. 論理設計

論理設計では概念設計で作成したER図などの概念モデルをリレーショナルモデルなどの論理モデルに変換します。その際に性能に関わる検討項目として正規化をどこまで実施するかという点が挙げられます。

3.2.1. テーブルの正規化

概念設計で作成した概念モデルをリレーショナルデータベースで取り扱うためには正規形と呼ばれる形式に変換を行います。[2] これをテーブルの正規化と呼びます。

考慮すべきポイント

正規化を行えばデータの追加・更新・削除などに伴うデータの不整合や喪失が起きるのを防ぎ、メンテナンスの効率を高めることができます。正規化の段階には第1～第5正規形およびボイスコッド正規形があります。

深く進める程データが細分化されそれぞれの管理や更新が容易になりますが、検索時にはテーブル結合が必要となるため、検索の範囲や条件が複雑なSQLのパフォーマンスが低下してしまいます。テーブルの数を抑え検索のパフォーマンスを向上させるためにあえて正規化の程度を落とすことも、性能面で考慮すべきポイントなります。[3]

考慮漏れによる起こりうる性能トラブルの例

機械的に正規化を進めた結果、都道府県、性別など非常に細かいレベルまで正規化を実施し、結合時のコストが問題になるパターンがあり得ます。

都道府県や性別のように新たな項目が追加されたり変更が起こりえないと考えられるものは正規化を崩し検索のパフォーマンスを優先させることが有効です。

参考情報

注釈	タイトル	URL	概要
[2]	OSS-DB入門　第4回　データベースの正規化	https://oss-db.jp/measures/dojo_info_04.shtml	データベースの正規化について紹介
[3]	データモデリング・テクニック	https://www.slideshare.net/HidekatsuIzuno/ss-67532977	パフォーマンス改善のために正規化を崩すこと、非正規化する場合の一例などを紹介

3.3. 物理設計

物理設計ではテーブル定義の設計からHW構成とサイジングの物理的な設計、パラメータの設計までを実施します。

各フェーズそれぞれで検討すべき項目が多く突き詰めていくと非常に奥が深いため、ここでは見落としがあると性能問題に発生しやすいポイントを取り上げます。

3.3.1. テーブル設計

正規形に変換したテーブルを、実際にPostgreSQLにおけるテーブル定義にあてはめていく工程です。

考慮すべきポイント

テーブルを設計/定義するにあたって性能面に影響する、考慮すべきポイントとして以下の3つの要素が挙げられます。

1. データ型の選択と検討。各列に適切なデータ型を利用する(数値データ型など)
   数値であれば精度の高さから正確な計算のためにnumeric型が採用されることがありますが、性能面でいえばint型やreal型と比較し明らかな差が出ます [4] 要件によって整数しか登場しない場合、ある程度の丸め誤差が許容される場合などint型やreal型を選択すべきです

2. HOTを有効に活用できるようFILFACTORの値を検討する [5]
   デフォルトでは100に設定されていますが、UPDATEが多いケースでは少し変えるだけで効果が出ます [6]

3. パーティショニングによるテーブル分割を活用する
   データサイズが大きくなる場合、また一定の範囲を検索する際にシーケンシャルスキャンが避けられない場合、パーティションの利用は参照性能の向上に有効です [7] 特にPostgreSQL10では宣言的パーティションが実装され課題であった挿入時の性能ネック、トリガによる管理の煩雑さが解決され利用する価値が大きく向上しました。[8]

考慮漏れによる起こりうる性能トラブルの例

UPDATEが多いシステムにおいてFILFACTORをデフォルトである100のまま運用していると、データの規模が大きくなるにつれてある時UPDATEが大幅に遅延するという問題が発生することがあります。

参考情報

注釈	タイトル	URL	概要
[4]	初めてのデータベース設計　第4回　データ型	http://gihyo.jp/dev/feature/01/database/0004?page=1	データ型の選択における概要と、数値型の特性について
[5]	HOTの上手な使い方	https://lets.postgresql.jp/documents/tutorial/hot3	HOTとFILFACTORについて
[6]	FILLFACTORによる性能改善	http://kyabatalian.hatenablog.com/entry/2017/12/16/160754	FILFACTORを100と80に設定した場合の検証結果
[7]	パーティショニングの性能検証結果を踏まえて、性能に影響を与える要素について考察	https://www.pgecons.org/wp-content/uploads/PGECons/2013/WG1/wg1-2013-report.pdf#page=30	パーティショニングによる検索性能の具体的な向上を示す検証結果
[8]	PostgreSQL 10の宣言的パーティション	https://www.ntt-tx.co.jp/column/postgresql_blog/20171005/	PostgreSQL10で実装された宣言的パーティションの改善点について

3.3.2. インデックス設計

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ていきます。

考慮すべきポイント

闇雲なインデックスの付与は更新処理の性能低下にもつながるため、定義すべき列の選択について以下の観点で精査していきます。

- 検索SQLのうち以下の対象となる列に作成する
    1. WHERE句の条件
    2. テーブル結合の条件
    3. ORDER BYやGROUP BYに指定される列
- カーディナリティの高い列に作成する

その他、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 [9] 、例として部分インデックスの利用で大幅に性能を向上させられるケースもあります [10]

また、定義したインデックスが実際にはあまり使われていないと更新性能の足かせとなってしまいます。実運用に入ってからのチューニングとはなりますが、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 [9]

考慮漏れによる起こりう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、seq scanが実行されCPU使用率が高騰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。

参考情報

注釈	タイトル	URL	概要
[9] (1, 2)	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	B-treeインデックスの有効活用のポイントや、利用状況の確認の仕方など
[10]	PostgreSQL SQLチューニング入門 実践編	https://www.slideshare.net/satoshiyamada71697/postgresql-sql	部分インデックスを用いて性能改善させる実践例

3.3.3. ハードウェアのサイジング

データの格納に必要なストレージサイズの見積もり、処理性能を満たすために必要なCPUとメモリの算定を実施します。

考慮すべきポイント

ストレージサイズの見積もりは決まった計算式で算出することができますが [11] 、CPUとメモリのサイジングは総処理数をもとに、実機を用いた検証でレスポンスタイムを確認して見積もる必要があります。

考慮漏れによる起こりうる性能トラブルの例

性能問題が発生した際に、パラメータやSQLの実行計画に特段問題が見当たらず、HWのスペックが絶対的に不足しているというケースがあります。

ロングトランザクション等はなく1つ1つのSQLは100ms以内に終わるものでも、CPUの性能に対し同時処理数が多い場合はパラメータ等では対処できず、物理的なスペック増強による対処が必要となります。

参考情報

注釈	タイトル	URL	概要
[11]	PostgreSQL でのデータベース構築の際に必要となる物理設計のポイント	https://lets.postgresql.jp/documents/tutorial/gihyo_rensai/5	ストーレジサイズの見積もりのための計算式について

3.3.4. ディスクの構成・ファイルの配置検討

PostgreSQLのデータベースを構成する各種ファイルの配置と、それに伴うディスクの構成を検討します

考慮すべきポイント

PostgreSQLのファイル構成はデータベースクラスタという単位で通常PGDATAという環境変数で一つのディレクトリ内に配置されます [12]

それぞれの領域に様々な処理に伴うI/Oが発生しますが [13] 、性能面を考慮する場合に特に推奨されるのがWALの保存領域をわけることであり、WALの書き込みI/O待ちを発生させないことが性能に直結します。

考慮漏れによる起こりうる性能トラブルの例

PostgreSQLのインストール時にPGDATA配下のディレクトリを同一ディスクの同一の論理領域にまとめて構成した場合、WALの書き込み待ちにより更新性能が出ないケースがあります。その場合WAL領域だけでも別のディスクに配置することで大きく性能が改善される可能性があります。

参考情報

注釈	タイトル	URL	概要
[12]	データベースファイルのレイアウト	https://www.postgresql.jp/document/9.5/html/storage-file-layout.html	PostgreSQLのPGDATA以下のディレクトリ構成について。利用するVerに応じたドキュメントを参照してください(特に10以降で名称の変更などあり)
[13]	実践！PostgreSQL運用	https://www.postgresql.jp/sites/default/files/2017-01/T5_jissen_PostgreSQL_unyou.pdf	ディスク設計の考え方と、各領域に発生するI/Oについても整理/解説

3.3.5. パラメータ設計

PostgreSQLの設定パラメータであるpostgresql.confの設定内容を検討、決定していきます。

考慮すべきポイント

PostgreSQLのパラメータについてはこう設定しておけばよい、という普遍的な設定値はありませんが、いくつかのパラメータについてデフォルトのままで運用していてトラブルが発生してしまう事例があります。

新しいVerほどデフォルトパラメータも考慮されつつありますが、最低限見直すべきパラメータ [14] は存在するため、参考情報のリンクをもとに値を精査していく必要があります。

考慮漏れによる起こりうる性能トラブルの例

- PostgreSQLを内部DBとして利用するSWパッケージ製品について、SWと同時にPostgreSQLもインストールされるがpostgresql.confのパラメータは設定されず、利用者が構築時に設定を怠るとデフォルトで運用され性能が出ないケースがある
- checkpoint_segmentsの値をデフォルトで運用しており、チェックポイントの頻発により性能が劣化してしまう。チェックポイント間隔の調整が必要に [15]
- ODBCドライバの利用において、Level of rollback on errorsがデフォルトの設定でSQL実行の度にSAVEPOINTとRELEASEが実行されるよになっており性能劣化を引き起こしてしまうケースがある

参考情報

注釈	タイトル	URL	概要
[14]	Tuning Your PostgreSQL Server/ja	https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server/ja	デフォルトから見直すべき基本的なパラメータについて解説
[15]	まずやっとく PostgreSQLのチューニング	http://dbstudychugoku.github.io/pdf/20140713_postgre_tuning.pdf	チェックポイント間隔の調整について解説

3.4. 運用の設計

データベースの一般的な設計としては以上となりますが、データベースを安定稼働させるためには運用管理の部分も検討が必要です。

ここでは運用の設計として主にVACCUMの取り扱いとDB監視(性能情報、特にスロークエリとロングトランザクション)について検討していきます。

3.4.1. VACUUMの取り扱い

PostgreSQL特有の性質であり運用上必ず必要となるVACUUMの実施タイミングについて検討していきます。

考慮すべきポイント

PostgreSQL特有の問題として定期的なVACUUMの実施によるガベージコレクションが必須となります。[16]

autovaccumをONにしていれば(8.3以降はデフォルトでON)テーブルの状態を監視して、然るべきタイミングに自動でVACUUMを実施してくれますが、データベースの利用状況によっては手動でのVACUUM実施を検討する必要があります。

例として頻繁に更新が発生するテーブルではXID周回防止処理 [17] がネックとなった事例があり、FREEZE処理のために夜間など、システムが利用されてないタイミングでVACUUMを手動実行させるべき場合があります。

また、APやシステムの設計によってはロングトランザクションやセッションの残存により ロックが取得できず、autovacuumが滞り性能が劣化していった事例などもあります。

そのようなケースではAPの設計とも併せて、VACUUMのタイミングを考慮する必要があります。

考慮漏れによる起こりうる性能トラブルの例

- 頻繁に更新のあるテーブルに対してautovacuum発生時のXID周回防止処理が性能ネックに。だんだんと目に見える処理遅延につながるケースがある。VACUUM FREEZEを夜間で手動実行するなどの対処が必要となる

- APの作りに不備がありロングトランザクション、セッション残存でautovacuumが阻害されてしまうケースがある。この場合、時間とともに性能劣化がどんどん進んでしまうことに

参考情報

注釈	タイトル	URL	概要
[16]	PostgreSQLの運用・監視にまつわるエトセトラ	https://www.slideshare.net/hadoopxnttdata/postgresql-operations-monitoring-etc	VACUUMの効用/副作用、実施方法や考えることなどを資料内で解説
[17]	PostgreSQL: XID周回問題に潜む別の問題	https://www.slideshare.net/hadoopxnttdata/postgresql-xid-wraparound-another-issue	PostgreSQLのXID周回問題にについて解説

3.4.2. DB性能情報の監視

DB監視ではDBやOSの様々な情報(ログやCPU/メモリの利用率、統計情報など)を監視し、障害や異常を事前に察知することが目的となります。

ここでは性能トラブルを早めに察知するために、性能情報を監視することを検討していきます。

考慮すべきポイント

導入前の評価では問題がなくても、実運用に入ってから思わぬ原因でスロークエリ、ロングトランザクションが発生することがあります。事例の中で見えてくるポイントとしては急激な性能劣化が発生するのではなく、だんだんと処理時間が伸びていっているパターンが多いです。

そのため、初期のスロークエリをいかに素早く検知し、問題を特定できるかが非常に重要です。

基本的なスロークエリの検知方法としてはlog_min_duration_statementの設定でログに出力させることができますが、クエリの処理時間だけでなくCPUの高騰、メモリの枯渇等がシグナルとなる場合もありますので これらを総合的に監視できるような運用、施策を検討することが性能トラブルへの一番の予防となります。

性能トラブルを予防するために考慮すべきデータベース設計のポイント で挙げた設計ポイントを考慮していっても実運用で発生する問題は予測しきれないものもあるため、この監視を充実させることこそ性能トラブルを発生させない一番の対策となります。

そのため本報告書では監視について別途 性能状態を把握するための監視 に切り出し記載しています。詳細は 性能状態を把握するための監視 の記述を参照してください。

考慮漏れによる起こりうる性能トラブルの例

監視を実施しないことが直接の性能トラブルの原因となることはありませんが、監視をすることで早期対処ができる可能性があります。[18]

参考情報

注釈	タイトル	URL	概要
[18]	DBAサバイバルガイド pg_stats_reporter で性能トラブルを洗い出せ	https://qiita.com/masudakz/items/b3226ba0c1753ad1538d	pg_statsinfoとpg_stats_reporterを用いて性能監視を行い、性能トラブルの予兆に気づき事前に対処した事例の紹介

3.5. その他

データベースの設計、運用の設計とは観点が異なりますが、性能トラブルを予防するにあたり留意すべきその他の検討要素を記載します。

3.5.1. 統計情報の取り扱い

性能トラブルのパターンとして統計情報が問題となるケースがあります。その場合は実行計画や統計情報を制御する外部ツールの導入も検討すべきポイントとなります。

考慮すべきポイント

性能トラブルとして急に処理が遅くなった、という場合に実行計画が適切でないものに変動されてしまった、その原因として統計情報が問題となっているケースがあります。

以下のような実行計画の変動により性能が悪化した事例について、いずれも統計情報が最新でなくプランナによる推定コストが低く見えてしまったことが原因となっており、最適な実行計画を選択してもらうには統計情報を適宜最新化することが必要です。

- HashJoinが選択されていたクエリが、NestedLoopが選択されてしまう
- HashJoinを用いるクエリについて、Hash作成対象テーブルが変更され、より件数の多いテーブルが選択されてしまう
- 全件検索が必要なテーブルにインデックススキャンが選択されてしまう

定期的なANALYZEの実行が有効な手段となりますが、データの傾向として最適なプランがわかっている場合は pg_hint_plan(PostgreSQLでヒント句を実装する外部ツール)の導入による実行計画の固定、またはpg_dbms_stats(PostgreSQL の統計情報管理ツール)による統計情報の固定化も有効です。[19]

考慮漏れによる起こりうる性能トラブルの例

実運用に入りしばらくしてからバッチ処理が大きく遅延するケースとして、ANALYZEの実行が運用に組み入れられておらず、上記ポイントで挙げたような実行計画の変動が発生してしまっている場合があります。

ANALYZEの実行により処理が改善される可能性もありますが、処理の特性により対象テーブルの件数が大きく変わる場合などは、pg_hint_planを導入し都度、Hash対象テーブルを明示的に指定するなどの対処も効果的です。

参考情報

注釈	タイトル	URL	概要
[19]	2015 年度 WG3 活動報告書　データベースツール編	https://www.pgecons.org/wp-content/uploads/PGECons/2015/WG3/PGECons_2015_WG3_DBTools.pdf#page=48	実行計画と統計情報の制御に関連するツールの紹介

4. 性能状態を把握するための監視

DBサーバの性能状態を正確に把握するためには、性能に関連する情報(以降、性能情報)を定期的に取得し、蓄積しておく必要があります。
性能トラブルが発生した場合には、蓄積した性能情報を元にボトルネック(問題を発生させる要素)を特定し、
ボトルネックが発生した原因について詳細に分析します。そして、分析結果を元に性能改善の施策を検討・実施する必要があります。

PostgreSQLが稼働するDBサーバの性能情報を蓄積し、ボトルネックの特定や分析を支援する機能を提供するソフトウェアは数多く存在し、「Monitoring - PostgreSQL wiki」[1] に整理されています。
また、2015年度のWG3の活動にて、性能監視ツールに求められる要件を整理し、pg_statsinfo(およびpg_stats_reporter)やpg_monzの調査を実施しております。
調査結果の詳細は、「2015年度WG3活動報告書(データベースツール編)」[2] や「付録_性能監視ツール」[3] [4] をご参照下さい。

[1] Monitoring/ja - PostgreSQL wiki - 2 外部ツールプロジェクト

[2] 2015 年度 WG3 活動報告書 - データベースツール編

[3] 付録_性能監視ツール1

[4] 付録_性能監視ツール2

性能トラブル発生時のボトルネックの特定および分析は、一般的には以下のような手法を用います。

表 4.1 性能情報の分析手法

項番	概要	説明	備考
1	過去情報との比較	トラブル発生時の情報と、過去の正常時の値を比較し、差分を確認することでボトルネックを絞り込む手法	性能情報を定期的に取得し、正常な状態(ベースライン)の性能情報を保持する必要がある
2	ドリルダウン手法	広い視点で情報を確認し、性能トラブルと関係がないと考えられる情報を捨て、関係が疑われる領域の情報を深堀していく手法	サーバの処理が全体的に遅延した場合に用いることが多い
3	処理時間に着目した分析	特定の処理に着目し、遅延した原因の詳細を深堀していく手法	特定の処理のみが遅延した場合に用いることが多い

ここでは、システム管理者などが性能トラブル発生時のボトルネック特定および分析する際に一助となる情報を提供することを目的し、
性能情報を分析するための観点(取得すべき情報)を整理しております。

4.1. 調査検証時に利用した環境

本章を記載するにあたり、利用した環境の情報は以下の通りです。

表 4.2 利用した環境情報

項目	説明	備考
OSバージョン	CentOS Linux release 7.4.1708 (Core)	-
sysstatバージョン	10.1.5-12.el7	sarコマンド、iostatコマンドを利用するためにインストール
procpsバージョン	3.3.10-16.el7	topコマンドを利用するためにインストール
PostgreSQLバージョン	10.2	-
構成	シングル構成(1台)	-

4.2. 性能情報の種別

OSのツール(sarコマンドやtopコマンド)やPosgresSQLは、コマンドラインやテーブルやビューなどを通して収集した性能情報を提供しております。
性能情報には以下種別の情報が存在します。
性能情報を正確に分析するためには、どの種別の性能情報なのかを事前に把握しておく必要があります。

表 4.3 性能情報の種別

項目	種別	説明	備考
1	累計値	ある時刻から収集された値の合計値	例えば、トランザクションのコミット数やロールバック数など。 単位時間毎の変化量を把握したい場合は、定期的に取得し、差分を確認する必要がある。
2	瞬間値	ある時点での状況や状態を示す値	例えば、トランザクション開始からの経過時間、現在実行中のSQL文など。
3	瞬間値(平均値)	測定開始した時間から、測定終了までの値	例えば、5秒間のCPU使用率の平均値など

4.3. 一定時間内に実行された処理の結果

性能トラブルを分析するには、システム管理者は"正常な状態の時に、一定時間内にどの程度処理ができているか"、
"どの程度処理に時間が掛かっているか"を把握しておく必要があります。
一定時間内に実行された処理の結果を把握しておくことで、性能トラブルが発生した時に以下の観点で調査を行うことができます。
・トランザクション数が増加することで負荷が増加し、性能トラブルに至っていないか
・処理遅延により実行されるトランザクション数が減少し、性能トラブルに至っていないか
このように正常時との比較を行うことで、性能トラブルを引き起こしている原因の手がかりを調査していくことになります。

表 4.4 一定時間内に実行された処理の結果

項番	性能情報	種別	概要
1	トランザクション数	累計値	トランザクションコミット数/ロールバック数
2	SQL処理	累計値	SQLの統計情報
3	遅延SQL	-	遅延SQL

4.3.1. トランザクション数

4.3.1.1. トランザクションコミット数/ロールバック数

pg_stat_databaseビュー [1] では、トランザクションコミット数とロールバック数の累計値が収集されています。
以下のようなSQLを用いることで、データベース毎に情報を取得することができます。

```
(サンプルSQL)
=#
SELECT
    datname,
    xact_commit,
    xact_rollback
FROM
    pg_stat_database;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]-+----------
datname       | postgres
xact_commit   | 22665
xact_rollback | 0
-[ RECORD 2 ]-+----------
datname       | testdb
xact_commit   | 73986
xact_rollback | 23
-[ RECORD 3 ]-+----------
datname       | template1
xact_commit   | 0
xact_rollback | 0
-[ RECORD 4 ]-+----------
datname       | template0
xact_commit   | 0
xact_rollback | 0
```

出力されるカラム情報は以下になります。

```
datname : データベース名
xact_commit : トランザクションコミット数
xact_rollback : トランザクションロールバック数
```

・確認観点

トランザクション数が低下している場合

以下のような事象が考えられます。

1. 実行されているトランザクション数が少なくなっている。(夜間帯などでトランザクション実行数が少ない)
2. 何らかのボトルネックが原因でPostgreSQLでのトランザクション処理数が低下している。(性能トラブル)

・参考情報

[1] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_stat_databaseビュー

4.3.2. SQL処理

4.3.2.1. SQLの統計情報

pg_stat_statementsモジュール [2] では、SQLの実行回数や総実行時間などに関する統計情報を取得しています。
以下のようなSQLを用いることで、SQLの統計情報を取得することができます。
実行時間が掛かっているSQL(= 遅延SQL)や性能影響の大きいSQLを検知することができます。

```
(サンプルSQL)
=#
SELECT
    query,
    calls,
    total_time,
    rows,
    100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS
hit_percent
FROM
    pg_stat_statements
ORDER BY
    total_time DESC LIMIT 5;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]----------------------------------------------------------------------------
-----------------------------
query       | EXPLAIN ANALYZE SELECT * FROM pgbench_branches b JOIN pgbench_accounts a
ON b.bid = a.bid ORDER BY a.aid
calls       | 4
total_time  | 14268.609267
rows        | 0
hit_percent | 91.0248161764705882
-[ RECORD 2 ]----------------------------------------------------------------------------
-----------------------------
query       | SELECT * FROM pgbench_branches b JOIN pgbench_accounts a ON b.bid = a.bid
ORDER BY a.aid
calls       | 4
total_time  | 1390.60727
rows        | 400000
hit_percent | 100.0000000000000000
```

出力されるカラム情報は以下になります。

```
query : SQL文の文字列(track_activity_query_size バイトまで)
calls : 実行回数
total_time : SQLの総実行時間
rows : SQL文によって取得された、あるいは影響を受けた行の総数
hit_percent : キャッシュヒット率
```

・確認観点

pg_stat_statment [2] の利用で、データベースへの性能影響が大きい以下のSQLを抽出することが可能です
1. SQLの総実行時間が長いSQL
2. 一回あたりの実行時間が長いSQL(total_time÷calls)

・参考情報

[2] (1, 2) PostgreSQL 10.0文書 - 付録F 追加で提供されるモジュール F.30. pg_stat_statements

4.3.3. 遅延SQL

4.3.3.1. 遅延SQL

auto_explainモジュール [3] は、指定した時間よりも遅延したSQLの実行計画を自動的にログに出力します。
遅延SQLの特定や遅延時に選択された実行計画を確認する際に効果的な情報です。
※ log_min_duration_statementパラメータ [4] に任意の時間を指定することで、指定した時間より遅延したSQL文をログに出力することも可能です。

```
監視対象ログファイル: PostgreSQLログ
監視対象文字列:LOG、duration
```

```
LOG:  duration: 252.673 ms  plan:
        Query Text: SELECT * FROM pgbench_branches b JOIN pgbench_accounts a ON b.bid =
a.bid ORDER BY a.aid;
        Nested Loop  (cost=0.29..6121.39 rows=100060 width=465)
          Join Filter: (b.bid = a.bid)
          ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts a
(cost=0.29..4619.47 rows=100060 width=97)
          ->  Materialize  (cost=0.00..1.01 rows=1 width=364)
                ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=364)
```

```
cost : コスト(シーケンシャルI/Oで1ページを読み込むコストを1.0とした際の相対値)
rows : 推定された行数
```

・確認観点

指定した時間よりも処理が遅延しているSQLを検知

該当のSQLが遅延している事を示します。
遅延SQLを特定した後、詳細な調査が必要な場合には、
EXPLAIN ANALYZE [遅延SQL] を実行することで、SQL実行時の各計画ノード(スキャンや結合などの内部処理)にかかった時間を取得することが可能です。
上記結果を元に問題箇所の特定を行います。

・参考情報

[3] PostgreSQL 10.0文書 - 付録F 追加で提供されるモジュール F.4. auto_explain

[4] PostgreSQL 10.0文書 - 第19章 サーバの設定 19.8. エラー報告とログ取得 log_min_duration_statementパラメータ

4.4. 任意の時間帯の処理実行状況

性能トラブル時にボトルネックを探すため、"このシステムでどういった処理が実行されているか"を把握する必要があります。
任意の時間帯の処理実行状況を把握しておくことで、性能トラブルが発生した時に以下の観点で調査を行うことができます。
・性能トラブル発生時の処理は、正常動作時と比べ違いがあるか
・性能トラブル発生時に実行中のプロセスで、ロック待ちが発生していないか
・正常動作時に確認されていない処理が行われていないか
・autovacuumなどのメンテナンス処理は正常に実行されているか
このような判断を行った上で、性能トラブルを引き起こしている原因やボトルネックになっている事象を調査していきます。

表 4.5 任意の時間帯の処理実行状況

項番	性能情報	種別	概要
1	実行中のトランザクション	瞬間値	トランザクション確認
2	実行中のautovacuum処理	瞬間値	VACUUM進捗状況

4.4.1. 実行中のトランザクション

4.4.1.1. トランザクション確認

pg_stat_activityビュー [1] では、プロセスの状態とSQLの開始時間が収集されています。
以下のようなSQLを用いることで、実行中のSQL情報を取得することができます。

```
(サンプルSQL)
=#
SELECT
    pid,
    wait_event_type,
    wait_event,
    state,
    (current_timestamp - xact_start)::interval(3) AS duration,
    query
FROM
    pg_stat_activity
WHERE
    pid <> pg_backend_pid()
ORDER BY
    duration DESC;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]---+-----------------------------------------------------
pid             | 1447
wait_event_type | Activity
wait_event      | LogicalLauncherMain
state           |
duration        |
query           |
-[ RECORD 2 ]---+-----------------------------------------------------
pid             | 1445
wait_event_type | Activity
wait_event      | AutoVacuumMain
state           |
duration        |
query           |
-[ RECORD 3 ]---+-----------------------------------------------------
pid             | 1443
wait_event_type | Activity
wait_event      | BgWriterHibernate
state           |
duration        |
query           |
-[ RECORD 4 ]---+-----------------------------------------------------
pid             | 1442
wait_event_type | Activity
wait_event      | CheckpointerMain
state           |
duration        |
query           |
-[ RECORD 5 ]---+-----------------------------------------------------
pid             | 1444
wait_event_type | Activity
wait_event      | WalWriterMain
state           |
duration        |
query           |
-[ RECORD 6 ]---+-----------------------------------------------------
pid             | 443
wait_event_type | Client
wait_event      | ClientRead
state           | idle in transaction
duration        | 00:00:52.961
query           | LOCK TABLE pgbench_accounts IN ACCESS EXCLUSIVE MODE;
```

出力されるカラム情報は以下になります。

```
pid : バックエンドのpid(postgresプロセスのpid)
wait_event_type : バックエンドが待機しているイベントの型
wait_event : 待機イベント名
duration : トランザクション開始からの経過時間
state : バックエンドの状態
query : 直近のSQL
```

・確認観点

長時間実行されているトランザクションが存在する場合

1. トランザクション内のSQLが完了しない(ロック待ちやハードウェアの割り当て待ちなど)
2. アプリケーションのコミット処理の漏れなどが原因でロングトランザクション化している。

ロングトランザクションは、autovacuumによる不要領域の回収などを妨げる等の問題を発生させるため、
ロングトランザクションを検知した場合は発生させないように対処が必要です。

・参考情報

[1] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_stat_activityビュー

4.4.2. 実行中のautovacuum処理

4.4.2.1. VACUUM進捗状況

pg_stat_progress_vacuumビュー [2] では、現在実行中のVACUUMに関する情報が収集されています。
以下のようなSQLを用いることで、実行中のVACUUM処理の状態を取得することができます。
ただし、VACUUM FULLの情報はサポートされておりません。

```
(サンプルSQL)
SELECT
    v.pid AS pid,
    v.datname AS datname,
    c.relname AS relname,
    v.phase AS phase,
    v.heap_blks_total AS heap_blks_total,
    v.heap_blks_scanned AS heap_blks_scanned,
    v.heap_blks_vacuumed AS heap_blks_vacuumed,
    v.index_vacuum_count AS index_vacuum_count,
    v.max_dead_tuples AS max_dead_tuples,
    v.num_dead_tuples AS num_dead_tuples
FROM
    pg_stat_progress_vacuum AS v
JOIN
    pg_class AS c
    ON v.relid = c.relfilenode;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]------+--------------
pid                | 24477
datname            | testdb
relname            | test
phase              | scanning heap
heap_blks_total    | 2655
heap_blks_scanned  | 141
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuples    | 772605
num_dead_tuples    | 32073
```

出力されるカラム情報は以下になります。

```
pid:バックエンドのプロセスID
datname:バックエンドが接続されているデータベース名
relname:バキューム処理が行われているテーブル名
phase:現在のバキュームの処理フェーズ
heap_blks_total:テーブルのヒープブロックの総数
heap_blks_scanned:スキャンされたヒープブロックの数
heap_blks_vacuumed:バキューム処理されたヒープブロックの数
index_vacuum_count:完了したインデックスバキュームサイクルの数
max_dead_tuples:インデックスバキュームサイクルの実行に必要となる前に格納することが出来る、maintenance_work_memに基づいた無効なタプルの数
num_dead_tuples:最後のインデックスバキュームサイクルから収集された無効タプルの数
```

・確認観点

VACUUM(autovacuum)が長時間終了しない場合

1. テーブルのサイズが大きく処理に時間がかかっている可能性があるため、VACUUM(autovacuum)のパラメータを検討する必要がある。
2. VACUUMが停止している場合は、VACUUM処理を阻害する要因が存在しないかを確認する。
例えば、カーソルが閉じられていないトランザクションはVACUUMの処理を阻害することがあります。
上記のトランザクションがロングトランザクション化しないように対処が必要です。

・参考情報

[2] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.4. 進捗状況のレポート pg_stat_progress_vacuumビュー

4.5. リソースの利用状況

性能トラブルを分析する際に、"正常時のHWリソース(CPUやメモリ、ディスク、ネットワークなど)の利用状況"を把握する必要があります。
正常時のHWリソースの利用状況を把握することで、性能トラブル時に以下の観点で調査を行うことができます。
・性能トラブル発生時のリソース使用状況は、正常な状態と比べ違いがあるか
・リソース利用に違いがある場合は、多くのリソースを使用するプロセスは存在するか
特定のプロセスが多くのリソースを使用している場合は該当プロセスが実行した処理を確認する必要があります。
(プロセスが実行した処理については「 任意の時間帯の処理実行状況 」を参照して下さい。)

表 4.6 リソースの利用状況

項番	性能情報	種別	概要
1	CPU利用率	瞬間値(平均値)	CPU利用率
2	メモリ状況	瞬間値(平均値)	メモリ状況
3	スワップ発生状況	瞬間値(平均値)	スワップ発生状況
4	ネットワーク使用状況	瞬間値(平均値)	ネットワーク使用状況
5	プロセスのリソース使用状況	瞬間値	プロセスのリソース使用状況

4.5.1. CPU利用率

sarコマンド [2] でシステムの統計情報を取得します。
-uオプションでCPU情報を表示させ、こちらを監視します。

```
# sar -u 5
```

一定間隔で結果が出力されます。

```
Linux 3.10.0-693.17.1.el7.x86_64 (localhost.localdomain)     2018年02月23日   _x86_64_
(1 CPU)

13時23分18秒     CPU     %user     %nice   %system   %iowait    %steal     %idle
13時23分23秒     all      0.64      0.00      0.64     98.73      0.00      0.00
13時23分28秒     all      0.41      0.00      0.41     99.18      0.00      0.00
13時23分33秒     all      0.21      0.00      0.21     99.59      0.00      0.00
13時23分38秒     all      0.21      0.00      0.42     99.38      0.00      0.00
```

出力される情報は以下になります。

```
%user ： ユーザプロセスによるCPU使用率
%nice ： 実行優先度を変更した(niceコマンド)ユーザプロセスによるCPU使用率
%system ： システムプロセスによるCPU使用率
%iowait ： ディスクI/O要求がある間のCPUが、アイドル状態である時間の%表示
%steal ： 仮想プロセッサを実行している間、強制的に待たされている時間の%表示
%idle ： CPUがアイドル状態で、ディスクI/O要求がない状態の時間の%表示
```

・確認観点

%user+%niceが高い

OS上で稼働するユーザプロセス(PostgreSQL含む)の処理でCPU負荷が高いことを示しています。
topコマンドなどでCPU負荷の高いプロセスを特定し、実行している処理に問題がないかを確認します。

%systemが高い

カーネル処理でCPU負荷が高いことを示しています。
ユーザプロセスから実行したシステムコール(カーネルが処理)で負荷が高くになっている場合もあります。
また、DBサーバで大容量メモリを使用している場合は、
THP(Transparent Huge Page)に関連したdefrag(compaction alloc)処理による性能トラブル [1] がよく知られておりますので、注意して下さい。

%iowaitが高い

ディスクに対するI/O処理が競合し、待ちが発生していることを示しています。
ユーザプロセスが実行する処理の中で不要なディスクアクセスが発生し、ボトルネックになっていないかを確認します。

・参考情報

[1] THP(Transparent Huge Page)に関連したdefrag(compaction alloc)処理による性能トラブル

[2] (1, 2, 3, 4, 5) sar manual page

4.5.2. メモリ状況

sarコマンド [2] でシステムの統計情報を取得します。
-rオプションでメモリ情報を表示させ、こちらを監視します。

```
# sar -r 5
```

一定間隔で結果が出力されます。

```
Linux 3.10.0-693.17.1.el7.x86_64 (localhost.localdomain)     2018年02月23日   _x86_64_
(1 CPU)

13時25分24秒 kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit
kbactive   kbinact   kbdirty
13時25分29秒    392572    624632     61.41         0    480112    618340     29.93
251696    255408       956
13時25分34秒    392324    624880     61.43         0    480504    618340     29.93
251932    255568       964
13時25分39秒    391952    625252     61.47         0    480896    618340     29.93
252156    255736       980
13時25分44秒    391580    625624     61.50         0    481228    618340     29.93
252364    255808       988
```

出力される情報は以下になります。

```
kbmemfree ： メモリ空き容量(KB)
kbmemused ： メモリ使用量(KB)
%memused ： メモリ使用率
kbbuffers ： カーネルによるバッファとしての使用量(KB)
kbcached ： カーネルによるキャッシュとしての使用量(KB)
kbcommit ： 現在の負荷量を処理するにあたり必要なメモリ量(KB)
%commit ： 現在の負荷量を処理するにあたり必要な使用率
kbactive ： アクティブなメモリの量(KB)
kbinact ： アクティブでないメモリの量(KB)
kbdirty ： ディスクに書きもどされるのを待っているメモリの量(KB)
```

・確認観点

kbmemused - (kbbuffers + kbcached) が徐々に増えている
メモリ使用量からページキャッシュ等を引いた実質のメモリ使用量(目安)を示しています。
この値が徐々に増加しているのであれば、使用するメモリ量が増加しています。
(単調に増加するパターンや増減を繰り返しながら増加するパターンがあります。)

・参考情報

sar manual page

4.5.3. スワップ発生状況

sarコマンド [2] でシステムの統計情報を取得します。
-Sオプションでスワップ情報を表示させ、こちらを監視します。

```
# sar -S 5
```

一定間隔で結果が出力されます。

```
Linux 3.10.0-693.17.1.el7.x86_64 (localhost.localdomain)     2018年02月23日   _x86_64_
(1 CPU)

13時26分27秒 kbswpfree kbswpused  %swpused  kbswpcad   %swpcad
13時26分32秒    971700     76872      7.33      5740      7.47
13時26分37秒    971700     76872      7.33      5740      7.47
13時26分42秒    971700     76872      7.33      5740      7.47
13時26分47秒    971700     76872      7.33      5740      7.47
```

出力される情報は以下になります。

```
kbswpfree ： スワップ領域の空き容量
kbswpused ： スワップ領域の使用量
%swpused ： スワップ領域の使用率
kbswpcad ： キャッシュされたスワップメモリ容量
%swpcad ： 使用されたスワップスペースに対する、キャッシュされたスワップメモリ使用率
```

・確認観点

kbswpusedと%swpusedが増加
物理メモリが不足し、スワップ領域が使用されている事を示します。
特定のユーザプロセスが大量にメモリを消費している事がなければ、物理メモリが
不足していると判断できますので、物理メモリの増設を検討する必要があります。

4.5.4. ディスクI/O

sarコマンド [2] でシステムの統計情報を取得します。
-dオプションでディスクI/O 情報を表示させ、こちらを監視します。

```
# sar -d 5
```

一定間隔で結果が出力されます。

```
Linux 3.10.0-693.17.1.el7.x86_64 (localhost.localdomain)     2018年02月23日  _x86_64_
(1 CPU)

13時27分18秒       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await
svctm     %util
13時27分23秒  dev252-0     61.76      0.00    516.97      8.37      0.99     16.10
16.07     99.24
13時27分23秒  dev253-0     61.76      0.00    516.97      8.37      0.99     16.10
16.07     99.24
13時27分23秒  dev253-1      0.00      0.00      0.00      0.00      0.00      0.00
0.00      0.00

13時27分23秒       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await
svctm     %util
13時27分28秒  dev252-0    116.35      0.00    977.36      8.40      1.04      8.88
8.46     98.39
13時27分28秒  dev253-0    115.51      0.00    977.36      8.46      1.04      8.95
8.52     98.43
13時27分28秒  dev253-1      0.00      0.00      0.00      0.00      0.00      0.00
0.00      0.00

13時27分28秒       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await
svctm     %util
13時27分33秒  dev252-0    107.25      0.00    901.04      8.40      0.98      9.12
9.18     98.45
13時27分33秒  dev253-0    107.25      0.00    901.04      8.40      0.98      9.12
9.18     98.45
13時27分33秒  dev253-1      0.00      0.00      0.00      0.00      0.00      0.00
0.00      0.00

13時27分33秒       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await
svctm     %util
13時27分38秒  dev252-0    102.27      0.00    895.87      8.76      0.99      9.72
9.63     98.49
13時27分38秒  dev253-0    102.27      0.00    895.87      8.76      0.99      9.73
9.63     98.53
13時27分38秒  dev253-1      0.00      0.00      0.00      0.00      0.00      0.00
0.00      0.00
```

出力される情報は以下になります。

```
DEV : デバイス名
tps : 1秒あたりのリクエスト数 (転送回数)
rd_sec/s : 1秒あたりの読み込みセクタ数
wr_sec/s : 1秒あたりの書き込みセクタ数
avgrq-sz : デバイスへのI/Oリクエストの平均ブロックサイズ
avgqu-sz : デバイスへのI/Oリクエストの平均待ちプロセス数 (キューの長さ)
await : デバイスへのI/Oリクエストの平均待ち時間 (msec)
svctm : デバイスへのI/Oリクエストの平均処理時間 (msec)
%util : デバイスへのI/Oリクエスト中のCPU使用率
```

・確認観点

awaitとsvctmの差が大きくなっている
ディスク使用率が高く、待ち行列が発生している事を示しています。

svctmが大きく増減している
要求に対しての平均レスポンスタイムが安定していない事を示しています。
高負荷システムでは性能トラブルの原因にもなりますので、データベース用途に向かない
ディスクと考えられます。

%utilが継続的に100%になっている
ディスクの使用頻度が高い事を示しています。
性能トラブルが発生している場合はディスク性能がボトルネックになっていると考えられます。

4.5.5. ネットワーク使用状況

sarコマンド [2] でシステムの統計情報を取得します。
-nオプションでネットワーク情報を表示させ、こちらを監視します。

```
# sar -n DEV 5
```

一定間隔で結果が出力されます。

```
Linux 3.10.0-693.17.1.el7.x86_64 (localhost.localdomain)     2018年02月23日  _x86_64_
(1 CPU)

13時28分30秒     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s
rxmcst/s
13時28分35秒      eth0      0.84      0.21      0.04      0.03      0.00      0.00
0.00
13時28分35秒        lo     12.76     12.76      2.64      2.64      0.00      0.00
0.00

13時28分35秒     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s
rxmcst/s
13時28分40秒      eth0      0.62      0.21      0.03      0.05      0.00      0.00
0.00
13時28分40秒        lo     12.58     12.58      2.70      2.70      0.00      0.00
0.00

13時28分40秒     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s
rxmcst/s
13時28分45秒      eth0      0.83      0.21      0.04      0.09      0.00      0.00
0.00
13時28分45秒        lo     12.19     12.19      2.55      2.55      0.00      0.00
0.00

13時28分45秒     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s
rxmcst/s
13時28分50秒      eth0      0.64      0.21      0.03      0.09      0.00      0.00
0.00
13時28分50秒        lo     13.46     13.46      2.86      2.86      0.00      0.00
0.00
```

出力される情報は以下になります。

```
IFACE : インターフェイス名
rxpck/s : 1秒あたりの受信パケット数
txpck/s : 1秒あたりの送信パケット数
rxkB/s : 1秒あたりの受信バイト数(KB)
txkB/s : 1秒あたりの送信バイト数(KB)
rxcmp/s : 1秒あたりの受信圧縮パケット数
txcmp/s : 1秒あたりの送信圧縮パケット数
rxmcst/s : 1秒あたりのマルチキャスト受信パケット数
```

・確認観点

rxkB/sとtxkB/sの合計値が大きい

rxkB/sとtxkB/sは1秒間に「受信したデータ量」と「送信したデータ量」を示します。
これらの値の合計値が帯域を越えていないかを確認します。

4.5.6. プロセスのリソース使用状況

topコマンド [3] で稼働するプロセスのCPUおよびメモリの統計情報を取得します。

```
# top -b -d 5
```

取得した時点の結果が出力されます。

```
top - 06:23:46 up 7 days,  7:38,  2 users,  load average: 0.00, 0.01, 0.05
Tasks:  94 total,   2 running,  92 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.2 sy,  0.0 ni, 99.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1016236 total,   157660 free,   128764 used,   729812 buff/cache
KiB Swap:  1048572 total,  1048188 free,      384 used.   679384 avail Mem

PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
24095 root      20   0  157588   2008   1448 R  0.4  0.2   0:00.46 top
1 root      20   0  128172   6748   3980 S  0.0  0.7   0:43.56 systemd
2 root      20   0       0      0      0 S  0.0  0.0   0:00.06 kthreadd
```

出力される情報は以下になります。

```
[各プロセスの情報のみ抜粋]
PID : プロセスID
USER : ユーザー名
PR : 優先度
NI : nice値
VIRT : 使用している仮想メモリ量
RES : 使用している物理メモリ量
SHR : 利用している共有メモリ量
%CPU : CPU 時間の占有率(総 CPU 時間のパーセンテージで表される)
%MEM : 利用可能な物理メモリの占有率
TIME+ : 開始してから利用した CPU 時間の総計
COMMAND : コマンドラインまたはプログラム名
```

iotopコマンド [4] で稼働するプロセスのディスクIOの統計情報を取得します。

```
# iotop -b -d 5
```

取得した時点の結果が出力されます。

```
Total DISK READ :       0.00 B/s | Total DISK WRITE :    1688.73 K/s
Actual DISK READ:       0.00 B/s | Actual DISK WRITE:     734.68 K/s
TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN      IO    COMMAND
26896 be/4 postgres    0.00 B/s  741.11 K/s  0.00 % 13.46 % postgres: wal writer process
26901 be/4 postgres    0.00 B/s  947.62 K/s  0.00 %  0.16 % postgres: postgres postgres
[local] COPY
```

出力される情報は以下になります。

```
[各プロセスの情報のみ抜粋]
TID : プロセスID
PRIO : IO優先度
USER : ユーザー
DISK READ : 1秒毎の読み込み量
DISK WRITE : 1秒毎の書き込み量
SWAPIN : スワップインに費やした時間の割合
IO : IO待ちに費やした時間の割合
COMMAND : マンドラインまたはプログラム名
```

・確認観点

リソース利用の割合が高いプロセスを特定します。PotgreSQLのプロセスが多くのリソースを消費している場合は、
「 任意の時間帯の処理実行状況 」を参照し、該当プロセスが実行中の処理の調査を実施して下さい。

・参考情報

[3] Linux User's Manual TOP

[4] iostat manual page

4.6. データベース性能に影響を与える要素

性能トラブルを分析する際には、"特定の要素がデータベース性能にどのような影響を与えるか"を調査する必要があります。
データベースの性能に影響を与えると考えられる要素について、事前に洗い出し情報を蓄積しておきます。
以下に性能に影響を与える要素の一例を記載します。

表 4.7 データベース性能に影響を与える要素

項番	性能情報	種別	概要
1	ロングトランザクション	瞬間値	ロングトランザクション
3	オブジェクトサイズ	瞬間値	データベースサイズ テーブルサイズ インデックスサイズ
4	オブジェクトスキャン	累計値	テーブルスキャン時の読み込み行数 インデックススキャンの割合
5	オブジェクトの状態	累計値	テーブル毎の不要領域確認 テーブル断片化 インデックス断片化
6	キャッシュヒット率	累計値	データベース毎のキャッシュヒット率 テーブル毎のキャッシュヒット率 インデックス毎のキャッシュヒット率
7	共有メモリ状況	瞬間値	共有メモリ状況
8	接続数	瞬間値	データベース毎の同時接続数
9	ロック状態	瞬間値	ロック待ち状態 ロック待ちSQL
10	デッドロック	累計値	デッドロック回数 デッドロックSQL
11	ディスクソート	累計値	SQLによるディスクソートの処理回数(work_memの不足回数) ログメッセージの確認(ディスクソート)

4.6.1. ロングトランザクション

実行中のトランザクション 内で長時間実行中となっているトランザクションが存在しないかを確認します。

4.6.2. オブジェクトサイズ

4.6.2.1. データベースサイズ

pg_databaseカタログ [1] には使用可能なデータベースの情報が格納されています。
以下のようなSQLを用いることで、データベース毎のサイズを取得することができます。

```
(サンプルSQL)
=#
SELECT
    datname,
    pg_size_pretty(pg_database_size(datname)) AS database_size
FROM
    pg_database;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]-+----------
datname       | postgres
database_size | 7861 kB
-[ RECORD 2 ]-+----------
datname       | testdb
database_size | 4280 MB
-[ RECORD 3 ]-+----------
datname       | template1
database_size | 7729 kB
-[ RECORD 4 ]-+----------
datname       | template0
database_size | 7729 kB
```

出力されるカラム情報は以下になります。

```
datname : データベース名
database_size : データベースサイズ
```

・確認観点

データベースサイズが顕著に増加

以下のような事象が考えられます。
1.レコードの増加に伴い、データベースサイズが増加している。
2.autovacuumが実行されているが、不要領域の回収を阻害する要因があるため、データベースが肥大化している。

・参考情報

[1] PostgreSQL 10.0文書 - 第51章 システムカタログ 51.15. pg_database

4.6.2.2. テーブルサイズ

pg_classカタログ [2] にはテーブルあるいはその他のオブジェクトの情報が格納されています。
以下のようなSQLを用いることで、テーブル毎のサイズを取得することができます。

```
(サンプルSQL)
=#
SELECT
    relname,
    reltuples,
    pg_size_pretty(pg_relation_size(oid)) AS tablesize
FROM
    pg_class
WHERE
    relnamespace = (SELECT oid FROM pg_namespace WHERE nspname='public')
    AND relkind in ('r', 't');
※ スキーマ名(nspname)は要件に応じて設定して下さい
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]---------------
relname   | pgbench_tellers
reltuples | 10
tablesize | 8192 bytes
-[ RECORD 2 ]---------------
relname   | pgbench_branches
reltuples | 1
tablesize | 8192 bytes
-[ RECORD 3 ]---------------
relname   | pgbench_accounts
reltuples | 100000
tablesize | 13 MB
-[ RECORD 4 ]---------------
relname   | pgbench_history
reltuples | 41356
tablesize | 2136 kB
-[ RECORD 5 ]---------------
relname   | foo
reltuples | 5e+07
tablesize | 2111 MB
```

出力されるカラム情報は以下になります。

```
relname : テーブル名
reltuple : テーブルの推定行数
tablesize : テーブルサイズ
```

・確認観点

テーブルサイズが顕著に増加

以下のような事象が考えられます。
1.レコード数の増加に伴い、テーブルサイズが増加している。
2.autovacuumが実行されているが、不要領域の回収を阻害する要因があるため、テーブルが肥大化している。

・参考情報

[2] PostgreSQL 10.0文書 - 第51章 システムカタログ 51.11. pg_class

4.6.2.3. インデックスサイズ

pg_classカタログ [3] にはテーブルあるいはその他のオブジェクトの情報が格納されています。
以下のようなSQLを用いることで、インデックス毎のサイズを取得することができます。

```
(サンプルSQL)
=#
SELECT
    relname as indexname,
    pg_size_pretty(pg_relation_size(oid)) AS indexsize
FROM
    pg_class
WHERE
    relnamespace = (SELECT oid FROM pg_namespace WHERE nspname='public')
    AND relkind = 'i';
※ スキーマ名(nspname)は要件に応じて設定して下さい
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]--------------------
indexname | val_index
indexsize | 1071 MB
-[ RECORD 2 ]--------------------
indexname | pgbench_branches_pkey
indexsize | 16 kB
-[ RECORD 3 ]--------------------
indexname | pgbench_tellers_pkey
indexsize | 16 kB
-[ RECORD 4 ]--------------------
indexname | pgbench_accounts_pkey
indexsize | 2208 kB
-[ RECORD 5 ]--------------------
indexname | foo_pkey
indexsize | 1071 MB
```

出力されるカラム情報は以下になります。

```
indexname : インデックス名
indexsize : インデックスサイズ
```

・確認観点

インデックスサイズが顕著に増加

インデックスサイズが肥大化している事を示しています。
レコード量の増加以外にインデックスの断片化が発生している可能性がありますので、
pgstatindex [4] を使用して、インデックスの断片化を確認して下さい。
必要であれば、REINDEXやインデックスの再作成を検討して下さい。

・参考情報

[3] PostgreSQL 10.0文書 - 第51章 システムカタログ 51.11. pg_class

[4] PostgreSQL 10.0文書 - 付録F 追加で提供されるモジュール F.31. pgstattuple(pgstatindex)

4.6.3. オブジェクトスキャン

4.6.3.1. テーブルスキャン時の読み込み行数

pg_stat_user_tablesビュー [5] では、テーブルスキャンが発生した回数と読み込まれた行数の累計値が収集されています。
以下のようなSQLを用いることで、テーブルスキャン時の読み込み行数を取得することができます。

```
(サンプルSQL)
=#
SELECT
    relname,
    seq_scan,
    seq_tup_read,
    seq_tup_read/seq_scan AS tup_per_read
FROM
    pg_stat_user_tables
WHERE
    seq_scan > 0
ORDER BY
    tup_per_read DESC;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]+-----------------
relname      | foo
seq_scan     | 11
seq_tup_read | 200000000
tup_per_read | 18181818
-[ RECORD 2 ]+-----------------
relname      | pgbench_accounts
seq_scan     | 3
seq_tup_read | 300000
tup_per_read | 100000
-[ RECORD 3 ]+-----------------
relname      | pgbench_tellers
seq_scan     | 50975
seq_tup_read | 509750
tup_per_read | 10
-[ RECORD 4 ]+-----------------
relname      | pgbench_branches
seq_scan     | 50979
seq_tup_read | 50979
tup_per_read | 1
```

出力されるカラム情報は以下になります。

```
relname : テーブル名
seq_scan: シーケンシャルスキャンの実行回数
seq_tup_read : シーケンシャルスキャンによって取り出された行数
seq_tup_read/seq_scan AS tup_per_read : シーケンシャルスキャン1回で取り出されてた平均行数
```

・確認観点

シーケンシャルスキャンの実行回数が顕著に増加した場合

以下のような事象が考えられます。

・インデックススキャンからシーケンシャルスキャンに実行計画が変わっている。

シーケンシャルスキャン1回で取り出されてた平均行数が数万行以上

以下のような事象が考えられます。

・全てのレコードを参照するSQLが実行されている。

・インデックスの定義漏れでシーケンシャルスキャンが実行されている。

・参考情報

[5] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_stat_all_tablesビュー(pg_stat_user_tablesビュー)

4.6.3.2. インデックススキャンの割合

pg_stat_user_tablesビュー [6] では、インデックススキャンが発生した回数と読み込まれた行数の累計値が収集されています。
以下のようなSQLを用いることで、インデックススキャンの割合を取得することができます。

```
(サンプルSQL)
=#
SELECT
    relname,
    seq_scan,
    idx_scan,
    CASE
        WHEN (seq_scan + idx_scan) = 0 THEN 0.00
        ELSE round(100 * idx_scan/(seq_scan + idx_scan),2)
    END AS idx_scan_ratio
FROM
    pg_stat_user_tables
ORDER BY
    idx_scan_ratio;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]--+-----------------
relname        | pgbench_branches
seq_scan       | 50979
idx_scan       | 0
idx_scan_ratio | 0.00
-[ RECORD 2 ]--+-----------------
relname        | pgbench_tellers
seq_scan       | 50975
idx_scan       | 0
idx_scan_ratio | 0.00
-[ RECORD 3 ]--+-----------------
relname        | foo
seq_scan       | 11
idx_scan       | 7
idx_scan_ratio | 38.00
-[ RECORD 4 ]--+-----------------
relname        | pgbench_accounts
seq_scan       | 3
idx_scan       | 101950
idx_scan_ratio | 99.00
-[ RECORD 5 ]--+-----------------
relname        | pgbench_history
seq_scan       | 0
idx_scan       |
idx_scan_ratio |
```

出力されるカラム情報は以下になります。

```
relname : テーブル名
seq_scan: シーケンシャルスキャンの実行回数
idx_scan : インデックススキャンの実行回数
round(100 * idx_scan/(seq_scan + idx_scan),2) : インデックススキャンの割合
```

・確認観点

シーケンシャルスキャンの実行回数が顕著に増加した場合

以下のような事象が考えられます。

・インデックススキャンからシーケンシャルスキャンに実行計画が変わっている。

シーケンシャルスキャン1回で取り出されてた平均行数が数万行以上

以下のような事象が考えられます。

・インデックスの定義漏れでシーケンシャルスキャンが実行されている。

・参考情報

[6] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_stat_all_tablesビュー(pg_stat_user_tablesビュー)

4.6.4. オブジェクトの状態

4.6.4.1. テーブル毎の不要領域確認

pg_stat_user_tablesビュー [7] では、推定値となりますが有効行と不要行の現在値が収集されています。
以下のようなSQLを用いることで、全体の行数の内と不要行の割合から不要領域を取得します。

```
(サンプルSQL)
=#
SELECT
    relname,
    n_live_tup,
    n_dead_tup,
    CASE n_dead_tup
        WHEN 0 THEN 0
        ELSE round(n_dead_tup*100/(n_live_tup+n_dead_tup),2)
    END AS ratio
FROM
    pg_stat_user_tables;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]----------------
relname    | foo
n_live_tup | 50000000
n_dead_tup | 0
ratio      | 0
-[ RECORD 2 ]----------------
relname    | pgbench_history
n_live_tup | 41819
n_dead_tup | 0
ratio      | 0
-[ RECORD 3 ]----------------
relname    | pgbench_accounts
n_live_tup | 100000
n_dead_tup | 1813
ratio      | 1.00
-[ RECORD 4 ]----------------
relname    | pgbench_branches
n_live_tup | 1
n_dead_tup | 0
ratio      | 0
-[ RECORD 5 ]----------------
relname    | pgbench_tellers
n_live_tup | 10
n_dead_tup | 22
ratio      | 68.00
```

出力されるカラム情報は以下になります。

```
relname : テーブル名
n_live_tup : 有効行の推定値
n_dead_tup : 不要行の推定値
round(n_dead_tup*100/(n_live_tup+n_dead_tup),2) AS ratio : 不要領域の割合
```

・確認観点

不要領域の割合が、長時間の間、autovacuum_vacuum_scale_factorパラメータ以上の場合

不要領域の割合が、長時間の間、autovacuum_vacuum_scale_factorパラメータ [8] 以上の場合、以下のような事象が考えられます。
1.autovacuumが該当テーブルで実行されていない。
2.autovacuumが実行されているが、不要領域の回収を阻害する要因がある。

・参考情報

[7] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_stat_all_tablesビュー(pg_stat_user_tablesビュー)

[8] PostgreSQL 10.0文書 - 第19章 サーバの設定 19.10. 自動Vacuum作業 autovacuum_vacuum_scale_factorパラメータ

4.6.4.2. テーブル断片化

pgstattupleモジュール [10] では、現在のタプルレベルの統計情報を入手するための各種関数が提供されています。
pgstattuple関数だと常に全件走査を実行し、有効タプルと無効タプルの正確な数(およびそのサイズ)と
空き領域を返しますが、性能への影響を考え、pgstattuple_approx関数を使用しています。
こちらは全件走査を避け、無効タプルの正確な統計情報および有効タプルと空き領域の数とサイズは近似値となります。
以下のようなSQLを用いることで、テーブルの断片化情報を取得することができます。

```
(サンプルSQL)
SELECT
    objectname,
    pg_size_pretty(pg_relation_size(objectname)) AS total_size,
    app.approx_tuple_count,
    app.approx_tuple_percent,
    app.dead_tuple_count,
    app.dead_tuple_percent,
    app.approx_free_space,
    app.approx_free_percent
FROM (
    SELECT
        *, quote_ident(schemaname) || '.' || quote_ident(relname) AS objectname
    FROM
        pg_stat_user_tables
) AS t, pgstattuple_approx(objectname::regclass) AS app
ORDER BY app.approx_free_percent DESC;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]--------+------------------------
objectname           | public.pgbench_branches
total_size           | 8192 bytes
approx_tuple_count   | 1
approx_tuple_percent | 10.9375
dead_tuple_count     | 0
dead_tuple_percent   | 0
approx_free_space    | 7296
approx_free_percent  | 89.0625
-[ RECORD 2 ]--------+------------------------
objectname           | public.pgbench_tellers
total_size           | 8192 bytes
approx_tuple_count   | 10
approx_tuple_percent | 4.39453125
dead_tuple_count     | 22
dead_tuple_percent   | 9.66796875
approx_free_space    | 6176
approx_free_percent  | 75.390625
-[ RECORD 3 ]--------+------------------------
objectname           | public.test
total_size           | 21 MB
approx_tuple_count   | 200000
approx_tuple_percent | 41.1436264124294
dead_tuple_count     | 0
dead_tuple_percent   | 0
approx_free_space    | 12801120
approx_free_percent  | 58.8563735875706
-[ RECORD 4 ]--------+------------------------
objectname           | public.pgbench_accounts
total_size           | 13 MB
approx_tuple_count   | 100000
approx_tuple_percent | 88.4461545658683
dead_tuple_count     | 0
dead_tuple_percent   | 0
approx_free_space    | 269044
approx_free_percent  | 1.96660390157186
-[ RECORD 5 ]--------+------------------------
objectname           | public.pgbench_history
total_size           | 2136 kB
approx_tuple_count   | 41817
approx_tuple_percent | 91.7683462078652
dead_tuple_count     | 0
dead_tuple_percent   | 0
approx_free_space    | 5304
approx_free_percent  | 0.242494733146067
```

出力されるカラム情報は以下になります。

```
objectname : テーブル名
total_size : テーブルサイズ
approx_tuple_count : 有効タプル数(推定)
approx_tuple_percent : 有効タプルの割合
dead_tuple_count : 無効タプル数(実測)
dead_tuple_percent : 無効タプルの割合
approx_free_space : 空き領域の合計バイト数(推定)
approx_free_percent : 空き領域の割合
```

・確認観点

オブジェクトサイズが大きくなっており、空き領域の割合が高い場合

テーブルが肥大化していることを示しています。
肥大化したテーブルサイズを縮小させるには、VACUUM FULLを実行するか、

テーブルを再作成する必要があります。(業務影響大のため、要検討)

オブジェクトサイズが大きくなっており、無効タプルの割合が高い場合

以下の事象が考えられます。
1. VACUUMが該当テーブルで実行されていない。
2. VACUUMが実行されているが、不要領域の回収を阻害する要因がある。

VACUUMが実行されていない場合は、autovacuumが実行される為の閾値 [9] が
高く設定されている可能性がありますので、見直しを検討してください。
不要領域が回収されていない場合は、ロングトランザクションが
VACUUM処理を阻害している可能性があります。
稼動統計情報からロングトランザクションが発生していないかを確認してください。

・参考情報

[9] PostgreSQL 10.0文書 - 第24章 定常的なデータベース保守作業 24.1. 定常的なバキューム作業 24.1.6. 自動バキュームデーモン

バキューム基礎閾値 + バキューム規模係数 * タプル数
(autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * pg_class.reltuples)

[10] PostgreSQL 10.0文書 - 付録F 追加で提供されるモジュール F.31. pgstattuple

4.6.4.3. インデックス断片化

pgstattupleモジュール [12] では、現在のタプルレベルの統計情報を入手するための各種関数が提供されています。
pgstatindex関数だとB-treeインデックスに関する情報を示すレコードを返却しますが、GINインデックスに関する情報は
pgstatginindex関数を使用してください。
以下のようなSQLを用いることで、インデックスの断片化情報を取得することができます。

```
(サンプルSQL)
SELECT
    objectname,
    pg_size_pretty(pg_relation_size(objectname)) AS total_size,
    app.tree_level,
    app.index_size,
    app.root_block_no,
    app.internal_pages,
    app.leaf_pages,
    app.empty_pages,
    app.deleted_pages,
    app.avg_leaf_density,
    app.leaf_fragmentation
FROM (
    SELECT
        *, quote_ident(schemaname) || '.' || quote_ident(indexrelname) AS objectname
    FROM
        pg_stat_user_indexes
) AS t, pgstatindex(objectname::regclass) AS app
ORDER BY app.leaf_fragmentation DESC;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]------+-----------------------------
objectname         | public.pgbench_branches_pkey
total_size         | 16 kB
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 0.29
leaf_fragmentation | 0
-[ RECORD 2 ]------+-----------------------------
objectname         | public.pgbench_tellers_pkey
total_size         | 16 kB
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 2.5
leaf_fragmentation | 0
-[ RECORD 3 ]------+-----------------------------
objectname         | public.pgbench_accounts_pkey
total_size         | 2208 kB
tree_level         | 1
index_size         | 2260992
root_block_no      | 3
internal_pages     | 1
leaf_pages         | 274
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 91.46
leaf_fragmentation | 0
-[ RECORD 4 ]------+-----------------------------
objectname         | public.test_pkey
total_size         | 11 MB
tree_level         | 2
index_size         | 11247616
root_block_no      | 412
internal_pages     | 4
leaf_pages         | 547
empty_pages        | 0
deleted_pages      | 821
avg_leaf_density   | 90
leaf_fragmentation | 0
```

出力されるカラム情報は以下になります。

```
objectname : インデックス名
total_size : インデックスサイズ
tree_level : ルートページのツリーレベル
index_size : バイト単位のインデックスサイズ
root_block_no : ルートページの場所 (存在しない場合はゼロ)
internal_pages :「内部」(上位レベル) ページ数
leaf_pages : リーフページ数
empty_pages : 空ページ数
deleted_pages : 削除ページ数
avg_leaf_density : リーフページの平均密度
leaf_fragmentation : リーフページの断片化
```

・確認観点

リーフページの断片化の割合が高い場合

インデックスが断片化していることを示しています。
REINDEXの実行、またはインデックスの再作成を検討する必要があります。
また、CREATE INDEXコマンドのCONCURRENTLYオプション [11] を指定することで、
PostgreSQLは、対象テーブルに対する同時挿入、更新、削除を防止するような
ロックを獲得せずにインデックスを作成することが可能です。

・参考情報

[11] PostgreSQL 10.0文書 - SQLコマンド CREATE INDEX

[12] PostgreSQL 10.0文書 - 付録F 追加で提供されるモジュール F.31. pgstattuple(pgstatindex)

4.6.5. キャッシュヒット率

4.6.5.1. データベース毎のキャッシュヒット率

pg_stat_databaseビュー [13] では、データベース毎にバッファキャッシュから読み込んだバッファヒット数と
ディスクから読み込んだディスクブロック数の累計値が収集されています。
以下のようなSQLを用いることで、全体のブロック数とバッファキャッシュから読み込んだバッファヒット数の割合から
キャッシュヒット率を取得することができます。

```
(サンプルSQL)
=#
SELECT
    datname,
    blks_hit,
    blks_read,
    round(blks_hit*100/(blks_hit+blks_read), 2) AS cache_hit_ratio
FROM
    pg_stat_database
WHERE
    blks_read > 0;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]---+----------
datname         | postgres
blks_hit        | 785946
blks_read       | 976
cache_hit_ratio | 99.00
-[ RECORD 2 ]---+----------
datname         | testdb
blks_hit        | 210105873
blks_read       | 3629745
cache_hit_ratio | 98.00
```

出力されるカラム情報は以下になります。

```
datname : データベース名
blks_hit : キャッシュから読み取られたブロック数
(PosgreSQLのshared_buffersでのキャッシュヒットのみを含み、OSのファイルシステムキャッシュはblks_hitに含まない)
blks_read : 読み取られたディスクブロック数
round(blks_hit*100/(blks_hit+blks_read), 2) AS cache_hit_ratio : キャッシュヒット率(%)
```

・確認観点

データベースのキャッシュヒット率が低下している場合

以下のような事象が考えられます。
1. アクセスするデータ量に対して、shared_buffersの値が不足している。
2. テーブルへのシーケンシャルスキャンもしくはインデックススキャンで多くのレコードが読み取られた。
3. あまり参照されていないデータにアクセスするSQLが実行された。

キャッシュヒット率が低い場合は、CPU利用率を確認し、iowaitの値が高くなっているか否かを確認します。
(iowaitの値が高くなっていない場合は、OSのファイルシステムキャッシュにて、キャッシュヒットしている場合があります。)
また、キャッシュヒット率が低い場合は、「テーブル毎のキャッシュヒット率」や「SQLのキャッシュヒット率」を確認し、
キャッシュヒット率低下の要因となっているテーブルやSQLを確認します。

・参考情報

[13] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ

4.6.5.2. テーブル毎のキャッシュヒット率

pg_statio_user_tablesビュー [14] では、テーブル毎にバッファキャッシュから読み込んだバッファヒット数と
ディスクから読み込んだディスクブロック数の累計値が収集されていますので、こちらを監視対象とします。
以下のようなSQLを用いることで、全体のブロック数とバッファキャッシュから読み込んだバッファヒット数の割合から
キャッシュヒット率を取得することができます。

```
(サンプルSQL)
=#
SELECT
    relname,
    heap_blks_hit,
    heap_blks_read,
    round(heap_blks_hit*100/(heap_blks_hit+heap_blks_read), 2) AS cache_hit_ratio
FROM
    pg_statio_user_tables
WHERE
    heap_blks_read > 0
ORDER BY
    cache_hit_ratio;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]---+-----------------
relname         | foo
heap_blks_hit   | 51259357
heap_blks_read  | 2938831
cache_hit_ratio | 94.00
-[ RECORD 2 ]---+-----------------
relname         | pgbench_accounts
heap_blks_hit   | 180899
heap_blks_read  | 8468
cache_hit_ratio | 95.00
-[ RECORD 3 ]---+-----------------
relname         | pgbench_history
heap_blks_hit   | 53243
heap_blks_read  | 330
cache_hit_ratio | 99.00
-[ RECORD 4 ]---+-----------------
relname         | pgbench_tellers
heap_blks_hit   | 102129
heap_blks_read  | 10
cache_hit_ratio | 99.00
-[ RECORD 5 ]---+-----------------
relname         | pgbench_branches
heap_blks_hit   | 102143
heap_blks_read  | 12
cache_hit_ratio | 99.00
```

出力されるカラム情報は以下になります。

```
relname : テーブル名
heap_blks_hit : キャッシュから読み取られたブロック数
heap_blks_read: 読み取られたディスクブロック数
round(heap_blks_hit*100/(heap_blks_hit+heap_blks_read), 2) AS cache_hit_ratio キャッシュヒット率(%)
```

・確認観点

テーブルのキャッシュヒット率が低下している場合

以下のような事象が考えられます。

1. アクセスするデータ量に対して、shared_buffersの値が不足している。
2. テーブルへのシーケンシャルスキャンもしくはインデックススキャンで多くのレコードが読み取られた。
3. あまり参照されていないデータにアクセスするSQLが実行された。

・参考情報

[14] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_statio_all_tablesビュー(pg_statio_user_tablesビュー)

4.6.5.3. インデックス毎のキャッシュヒット率

pg_statio_user_indexesビュー [15] では、インデックスにおけるバッファヒット数とインデックスから読み取られたディスクブロック数の累計値が収集されています。
以下のようなSQLを用いることで、全体のブロック数とインデックスにおけるバッファヒット数の割合からキャッシュヒット率を取得することができます。

```
(サンプルSQL)
=#
SELECT
    indexrelname,
    relname,
    idx_blks_hit,
    idx_blks_read,
    round(idx_blks_hit*100/(idx_blks_hit+idx_blks_read), 2) AS cache_hit_ratio
FROM
    pg_statio_user_indexes
WHERE
    idx_blks_read > 0
ORDER BY
    cache_hit_ratio;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]---+----------------------
indexrelname    | val_index
relname         | foo
idx_blks_hit    | 123
idx_blks_read   | 278892
cache_hit_ratio | 0.00
-[ RECORD 2 ]---+----------------------
indexrelname    | pgbench_branches_pkey
relname         | pgbench_branches
idx_blks_hit    | 42
idx_blks_read   | 6
cache_hit_ratio | 87.00
-[ RECORD 3 ]---+----------------------
indexrelname    | pgbench_tellers_pkey
relname         | pgbench_tellers
idx_blks_hit    | 34
idx_blks_read   | 4
cache_hit_ratio | 89.00
-[ RECORD 4 ]---+----------------------
indexrelname    | foo_pkey
relname         | foo
idx_blks_hit    | 157394582
idx_blks_read   | 400482
cache_hit_ratio | 99.00
-[ RECORD 5 ]---+----------------------
indexrelname    | pgbench_accounts_pkey
relname         | pgbench_accounts
idx_blks_hit    | 208539
idx_blks_read   | 1105
cache_hit_ratio | 99.00
```

出力されるカラム情報は以下になります。

```
indexrelname : インデックス名
relname : インデックスが定義されたテーブル名
idx_blks_hit : キャッシュから読み取られたブロック数
idx_blks_read: 読み取られたディスクブロック数
round(idx_blks_hit*100/(idx_blks_hit+idx_blks_read), 2) AS cache_hit_ratio : キャッシュヒット率
```

・確認観点

インデックスのキャッシュヒット率が低下している場合

以下のような事象が考えられます。

1. アクセスするデータ量に対して、shared_buffersの値が不足してる(レコード数の増加などが考えられます)。
2. インデックスを利用して多くのデータが読み取られた。
3. あまり参照されてないインデックス(データ)にアクセスするSQLが実行された。

・参考情報

[15] pg_statio_all_indexesビュー(pg_statio_user_indexesビュー)

4.6.6. 共有メモリ状況

4.6.6.1. 共有メモリ状況

pg_buffercacheビュー [16] では、現在の共有バッファの情報が収集されています。
以下のようなSQLを用いることで、共有バッファの状態を取得することができます。
ただし、pg_buffercacheビュー [16] にアクセスがあると、ビューが表示する全てのバッファ状態をコピーするために
十分な期間、内部バッファマネージャはロックを取得しますので、頻繁にアクセスするとデータベース性能に多少影響が発生することにご注意願います。

```
(サンプルSQL)
SELECT
    c.relname,
    count(*) AS buffers
FROM
    pg_buffercache AS b
INNER JOIN
    pg_class AS c
    ON b.relfilenode = pg_relation_filenode(c.oid)
    AND b.reldatabase IN (0, (SELECT oid FROM pg_database WHERE datname =
current_database()))
GROUP BY c.relname
ORDER BY 2 DESC;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]---------------------------------
relname | test
buffers | 2659
-[ RECORD 2 ]---------------------------------
relname | pgbench_accounts
buffers | 1670
-[ RECORD 3 ]---------------------------------
relname | test_pkey
buffers | 1373
-[ RECORD 4 ]---------------------------------
relname | pgbench_accounts_pkey
buffers | 276
    :
```

出力されるカラム情報は以下になります。

```
relname : オブジェクト名（テーブル、インデックスなどの名前）
buffers : 共有キャッシュ内のバッファ数
```

・確認観点

特定のオブジェクトのバッファ数が共有バッファ上に少ない場合

以下のような事象が考えられます。

1. 共有メモリ領域が小さいために、特定のオブジェクトのバッファを載せきれていない
2. アクセス頻度が少ないため、共有バッファ上からキャッシュアウトされやすくなっている

・参考情報

[16] *(1, 2)* PostgreSQL 10.0文書 - 付録F 追加で提供されるモジュール F.25. pg_buffercache

4.6.7. 接続数

4.6.7.1. データベース毎の同時接続数

pg_stat_activityビュー [17] では、サーバプロセス毎の情報が収集されています。
以下のようなSQLを用いることで、表示されているプロセスの数を現在の接続数として取得することができます。

```
(サンプルSQL)
=#
SELECT
    datname,
    count(*) AS connections
FROM
    pg_stat_activity
WHERE
    datname IS NOT NULL
GROUP BY
    datname;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]----------
datname     | postgres
connections | 1
-[ RECORD 2 ]----------
datname     | testdb
connections | 1
```

出力されるカラム情報は以下になります。

```
datname : データベース名
connections : 現在の接続数
```

・確認観点

現在の接続数が徐々に増加
接続数が増加している事を示しています。
稼動統計情報を確認し、トランザクションが閉じられないまま長時間が経過している状態
(ロングトランザクション)になっていないかを確認します。

・参考情報

[17] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ pg_stat_activityビュー

4.6.8. ロック状態

4.6.8.1. ロック待ち状態

pg_locksビュー [18] では、データベース内でアクティブなプロセスによって保持されたロックに関する情報の現在値が収集されています。
以下のようなSQLを用いることで、ロック待ちの状態を取得することができます。

```
(サンプルSQL)
=#
SELECT
    date_trunc('second', current_timestamp(0)) AS time,
    blocked_locks.pid AS blocked_pid,
    blocked_activity.usename AS blocked_user,
    blocking_locks.pid AS blocking_pid,
    blocking_activity.usename AS blocking_user,
    blocked_activity.query AS blocked_statement,
    blocking_activity.query AS current_statement_in_blocking_process,
    (current_timestamp - blocked_activity.xact_start)::interval(3) AS duration
FROM
    pg_catalog.pg_locks blocked_locks
JOIN
    pg_catalog.pg_stat_activity blocked_activity
    ON blocked_activity.pid = blocked_locks.pid
JOIN
    pg_catalog.pg_locks AS blocking_locks
    ON blocking_locks.locktype = blocked_locks.locktype
    AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
    AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
    AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
    AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
    AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
    AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
    AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
    AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
    AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
    AND blocking_locks.pid != blocked_locks.pid
JOIN
    pg_catalog.pg_stat_activity blocking_activity
    ON blocking_activity.pid = blocking_locks.pid
WHERE
    NOT blocked_locks.GRANTED;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]-------------------------+-------------------------------------------------
-----
time                                | 2018-03-01 18:05:46+09
blocked_pid                         | 32603
blocked_user                        | postgres
blocking_pid                        | 32522
blocking_user                       | postgres
blocked_statement                   | SELECT * FROM pgbench_accounts;
current_statement_in_blocking_process | LOCK TABLE pgbench_accounts IN ACCESS EXCLUSIVE
MODE;
duration                            | 00:03:05.212
```

出力されるカラム情報は以下になります。

```
time : 取得時間
blocked_pid : ロック待ちプロセスID
blocked_user : ロック待ちユーザ
blocking_pid : ロックしているプロセスID
blocking_user : ロックしているユーザ
blocked_statement : ロック待ちSQL文
current_statement_in_blocking_process : ロックしているSQL文
duration : トランザクションの実行時間
```

・確認観点

大量のプロセスでロック待ちが発生

以下の様な事象が考えられます。
1. 特定のプロセスで実行された処理が取得するロックにより他の処理でロックが取得できていない。
2. 該当のSQLで遅延が発生し、本来完了後に実行されるはずの他の処理でロックが取得できなかった。

これらの事象を解消するには、ロックしているSQLを確認して下さい。
VACUUM FULLやREINDEXなど強いロックを必要とする処理であれば、実行時間の変更や代替案を検討して下さい。
該当のSQLが遅延しているのであれば、原因を調査し、遅延を解消する必要があります。

・参考情報

[18] PostgreSQL 10.0文書 - 第51章 システムカタログ 51.73. pg_locks

4.6.8.2. ロック待ちSQL

log_lock_waitsを有効にすることでプロセスでデッドロックのタイムアウト時間よりもロック待ちが発生した場合、自動的にログに出力します。
特定の処理がロック待ちの原因になっていないかを確認するために監視対象とします。

監視対象ログファイル： PostgreSQLログ
監視対象文字列:LOG、still waiting

```
LOG:  process 11856 still waiting for AccessShareLock on relation 24880 of database
16384 after 1000.324 ms at character 15
DETAIL:  Process holding the lock: 11854. Wait queue: 11856.
STATEMENT:  SELECT * FROM pgbench_accounts;
```

・確認観点

ロック待ちを検知

該当のSQLでロック待ちが発生している事を示しています。
ロック待ちが発生している該当のSQLを確認し、ロック待ちを引き起こしている原因となる事象を確認して下さい。

4.6.9. デッドロック

4.6.9.1. デッドロック回数

pg_stat_databaseビュー [19] では、データベース内で検知されたデッドロック数の累積値が収集されています。
以下のようなSQLを用いることで、デッドロックの発生回数を取得することができます。

```
(サンプルSQL)
=#
SELECT
    datname,
    deadlocks
FROM
    pg_stat_database;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]--------
datname   | postgres
deadlocks | 0
-[ RECORD 2 ]--------
datname   | testdb
deadlocks | 1
-[ RECORD 3 ]--------
datname   | template1
deadlocks | 0
-[ RECORD 4 ]--------
datname   | template0
deadlocks | 0
```

出力されるカラム情報は以下になります。

```
datname : データベース名
deadlocks : デッドロック発生回数
```

・確認観点

デッドロック回数が増加

デッドロックが頻発している事を示しています。
ログメッセージを確認し、デッドロックが発生した該当のSQLを特定した後、
該当SQLを実行するアプリケーションにてデッドロックを引き起こす制御になっていないかを確認して下さい。

・参考情報

[19] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ

4.6.9.2. デッドロックSQL

log_lock_waitsを有効にすることでプロセスでデッドロック [20] が発生した場合、自動的にログに出力します。
特定の処理でデッドロックが発生していないかを確認するために監視対象とします。

```
監視対象ログファイル： PostgreSQLログ
監視対象文字列：ERROR、deadlock
```

```
ERROR:  deadlock detected
DETAIL:  Process 11957 waits for AccessExclusiveLock on relation 24880 of database
16384; blocked by process 11948.
        Process 11948 waits for AccessExclusiveLock on relation 24874 of database 16384;
blocked by process 11957.
        Process 11957: LOCK TABLE pgbench_accounts IN ACCESS EXCLUSIVE MODE;
        Process 11948: LOCK TABLE pgbench_history IN ACCESS EXCLUSIVE MODE;
HINT:  See server log for query details.
STATEMENT:  LOCK TABLE pgbench_accounts IN ACCESS EXCLUSIVE MODE;
LOG:  process 11948 acquired AccessExclusiveLock on relation 24874 of database 16384
after 5009.589 ms
STATEMENT:  LOCK TABLE pgbench_history IN ACCESS EXCLUSIVE MODE;
```

・確認観点

デッドロックを検知

該当のSQLでデッドロックが発生した事を示しています。
deadlock_timeoutパラメータ [21] に指定した時間毎の検出処理でデッドロックが検出された場合、片方のトランザクション処理は
強制的に終了され、もう片方のトランザクション処理は完了されます。
デッドロックが発生した該当のSQLを確認し、アプリケーション側でデッドロックを引き起こす制御になっていないかを確認して下さい。

・参考情報

[20] PostgreSQL 10.0文書 第13章 同時実行制御 13.3. 明示的ロック 13.3.4. デッドロック

[21] PostgreSQL 10.0文書 - 第19章 サーバの設定 19.12. ロック管理 deadlock_timeoutパラメータ

4.6.10. ディスクソート

4.6.10.1. SQLによるディスクソートの処理回数(work_memの不足回数)

pg_stat_databaseビュー [22] では、問い合わせによって書き出された一時ファイルの個数とデータ量の累計値が収集されています。
work_memのサイズ [23] が不足している場合、一時ファイルに書き出して処理を行うので、この値を監視対象としています。
以下のようなSQLを用いることで、ディスクソートの発生情報を取得することができます。

```
(サンプルSQL)
=#
SELECT
    datname,
    temp_files,
    pg_size_pretty(temp_bytes) AS temp_bytes ,
    pg_size_pretty(round(temp_bytes/temp_files,2)) AS temp_file_size
FROM
    pg_stat_database
WHERE
    temp_files > 0;
```

SQLの実行結果は以下の通りです。

```
-[ RECORD 1 ]--+-------
datname        | testdb
temp_files     | 46
temp_bytes     | 10 GB
temp_file_size | 233 MB
```

出力されるカラム情報は以下になります。

```
datname : データベース名
temp_files : SQLによって書き出された一時ファイルの個数
pg_size_pretty(temp_bytes) AS temp_bytes : 書き出された一時ファイルの合計サイズ
pg_size_pretty(round(temp_bytes/temp_files,2)) AS temp_file_size : 一時ファイルの平均サイズ
```

・確認観点

一時ファイルが作成されている場合

ソート処理やハッシュ処理でメモリ(work_mem) [23] が不足し、作業用の一時ファイルがディスク上に書き出されています。
OSのメモリ使用量に空き領域がある場合は、work_memの設定値を大きくすることを検討します。

・参考情報

[22] PostgreSQL 10.0文書 - 第28章 データベース活動状況の監視 28.2. 統計情報コレクタ

[23] (1, 2) PostgreSQL 10.0文書 - 第19章 サーバの設定 19.4. 資源の消費 work_memパラメータ

4.6.10.2. ログメッセージの確認(ディスクソート)

log_temp_filesで指定したサイズ以上の一時ファイルが作成された場合、ログに出力します。
一時ファイルによるディスクアクセスが発生していないかを確認するために監視対象とします。

監視対象ログファイル： PostgreSQLログ
監視対象文字列：LOG、temporary file

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp1125.19", size 14131200
STATEMENT:  SELECT COUNT(*) FROM (SELECT i FROM generate_series(1,1000000) AS i ORDER BY
i DESC limit 500000 OFFSET 0) AS x;
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp1125.18", size 14000000
STATEMENT:  SELECT COUNT(*) FROM (SELECT i FROM generate_series(1,1000000) AS i ORDER BY
i DESC limit 500000 OFFSET 0) AS x;
```

・確認観点

一時ファイルを検知

該当のSQLで一時ファイルを作成している事を示しています。
該当のSQLにEXPLAIN (ANALYZE, BUFFERS)を付与し、実際に実行した情報を確認して下さい。
ハッシュ処理やソート処理で一時ファイルが作成されているようであれば、work_memのサイズ [24] が足りていませんので、
work_memサイズ [24] の増加を検討して下さい。

・参考情報

[24] (1, 2) PostgreSQL 10.0文書 - 第19章 サーバの設定 19.4. 資源の消費 work_memパラメータ

5. ケーススタディ

5.1. ロングトランザクションによる性能トラブル

5.1.1. 概要

PostgreSQLは追記型のアーキテクチャを採用していますので、更新処理が行われる度に行レコード(タプル)が増えていきます。
不要となったタプルはVACUUM処理によって回収され、その領域は再利用されます。
このようにVACUUM処理が定期的に実行されることで、データベースは肥大化せずに安定動作を実現しています。

このVACUUM処理を阻害する要因となるのが、「ロングトランザクション」です。
ロングトランザクションとは、BEGINを発行してからCOMMITするまで、長時間かかっている処理を指します。
VACUUM処理を実行する際に不要領域の判定を行いますが、この判定は各トランザクションの可視性から判断されます。
つまり、ロングトランザクションが存在することで、不要となったタプルが不要領域と判断されずに、いつまでたっても
VACUUM処理で回収されなくなります。
このような状態で運用を続けますと、データベースが肥大化していき、性能トラブルを引き起こすトラブルに発展していきますので、
予めロングトランザクションが発生していないかを監視しておくことが必要となります。

下記に「ロングトランザクションによる性能トラブル」についての事例を記載します。

5.1.2. 性能トラブル内容

データベースを運用していたところ、しばらくは問題なく動作していましたが
いつ頃からか、処理遅延が頻発するようになりました。
発生した処理遅延は以下の通りです。

1. トランザクション処理数の低下
日中帯(9:00～17:00)にオンラインで実行されているトランザクションの処理数が減少しています。

正常時の遅延発生時のTPS値での比較

(正常期間)
```
tps = 773.078073
```

(処理遅延発生期間)
```
tps = 400.656880
```

2. 夜間帯(深夜0:00時頃)にバッチで実行されているSQLの遅延
夜間帯に実行されるSQLが遅延SQLとして、PostgreSQLのログファイルに出力されるようになりました。
なお、本システムではSQLの遅延の閾値を10秒としているため、「log_min_duration_statement = 10s」を指定し、
遅延が発生したSQLをPostgreSQLのログファイルに出力しています。

■ログ確認(遅延SQL)

```
[20XX-XX-XX XX:XX:XX.XXX JST][<ユーザ名>][<データベース名>][[<ホスト名]][<PID>][<セッションID>-<各セッションのログ行の番号>][0][00000]
psql LOG:  duration: 17374.466 ms  statement: SELECT bid ,count(*) FROM pgbench_accounts GROUP BY bid;
```

5.1.3. 調査・分析結果

■テーブルサイズ確認

処理時間の遅延が発生した場合、特定の処理で発生しているのか、または全体的に遅延しているかを確認します。
今回の事例では、全体的に処理時間が遅延しているとのことなので、まずはデータベースのテーブルサイズを確認します。
(テーブルサイズを確認するSQLは、こちら をご参照下さい。)
通常期間と比べ、テーブルサイズが増加していることが確認できます。

```
(正常期間)
     relname      | reltuples | tablesize
------------------+-----------+-----------
 pgbench_history  |    256897 | 15 MB
 pgbench_tellers  |      1000 | 128 kB
 pgbench_branches |       100 | 96 kB
 pgbench_accounts |     1e+07 | 1299 MB
(4 rows)
```

```
(処理遅延発生期間)
     relname      | reltuples | tablesize
------------------+-----------+-----------
 pgbench_history  |    975506 | 51 MB
 pgbench_tellers  |      1000 | 42 MB     ---レコード数は変化していないが、テーブルサイズが顕著に増加
 pgbench_branches |       100 | 34 MB     ---レコード数は変化していないが、テーブルサイズが顕著に増加
 pgbench_accounts |     1e+07 | 1407 MB
(4 rows)
```

■不要領域の確認

データベースおよびテーブルの肥大化が確認できましたら、それがレコード数増加によるものか、またはVACUUM処理(autovacuum含む)による
不要領域の回収で問題が発生しているかを確認します。
(テーブル毎の不要領域を確認するSQLは、こちら をご参照下さい。)
n_dead_tupが増加していることからVACUUM処理による不要領域の回収で問題が発生していることが確認できます。(※)
※テーブルサイズの20%不要領域が溜まったら、autovacuumが実行されることから、autovacuum処理が阻害されていると判断できます。

```
(処理遅延発生期間)
     relname      | n_live_tup | n_dead_tup | ratio
------------------+------------+------------+-------
 pgbench_branches |        100 |     984767 | 99.00
 pgbench_history  |     998339 |          0 |     0
 pgbench_accounts |   10000006 |     999295 |  9.00
 pgbench_tellers  |       1000 |     984779 | 99.00
(4 rows)
```

■稼動統計情報の確認

VACUUM処理による不要領域の回収で問題が発生していることから、ロングトランザクションが発生していると推測し、稼動統計情報を確認します。
稼動統計情報の中から、ロングトランザクションに該当する処理(stateが「idle in transaction」のまま、長時間トランザクションが閉じられていない処理)が
発生していないかを確認します。
(ロングトランザクションを確認SQLは、こちら をご参照下さい。)
ロングトランザクションが発生していた場合、queryから該当の処理がロングトランザクション化した原因を調査し、対処します。(※)
※アプリケーション側のコミット漏れ等

```
(処理遅延発生期間)
     :
-[ RECORD 7 ]---+-------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------
pid             | 6763
wait_event_type | Client
wait_event      | ClientRead
state           | idle in transaction
duration        | 00:30:10.235
query           | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
(111111111, 1, 1, 0, CURRENT_TIMESTAMP);
     :
```

5.1.4. 性能トラブルの改善案

調査の結果、特定の処理がロングトランザクション化していることが性能トラブルの原因となっていることが確認できます。
一般的にロングトランザクション化する要因としてアプリケーション側のコミット漏れがあります。
このようにアプリケーション側に問題がある場合、改修までに時間が掛かることがあります。

運用時にこの性能トラブルを回避するには、以下のような方法があります。
1. PostgreSQLの設定ファイルで「idle_in_transaction_session_timeout (PostgreSQL 9.6～)」を任意の時間に設定し、設定ファイルの変更を反映(pg_ctl reload)
2. 対象のプロセスをpg_terminate_backendコマンドで終了

運用時でもこれらの方法で性能トラブルを回避することはできますが、あくまで暫定対策とし、アプリケーションの改修など根本的な改善策の実施をご検討ください。

5.1.5. 備考

PostgreSQLでは、削除されたレコード(=不要領域)を回収する際に他のトランザクションからの可視性を確認します。
よって、他のトランザクションの状態やトランザクション分離レベルによっては、VACUUM処理を阻害することになります。
今回の事象のようにトランザクションがVACUUM処理を阻害するケースを以下に示します。

リードコミッティド
・実行中のクエリ
・書き込みを行いアイドル状態になっているトランザクション
(書き込みを行っていないトランザクションであればVACUUM処理を阻害しません)

シリアライザブル
・実行中のクエリ
・読み込み、または書き込みを行いアイドル状態になっているトランザクション

5.2. ディスク性能の考慮漏れによる性能トラブル

5.2.1. 概要

PostgreSQLはコストベースオプティマイザを採用しており、SQL文と統計情報を元に選択可能な実行計画を列挙し、
各プランのコストを計算します。その中で最もコストの低い実行計画を選択します。
(コストと実行計画の詳細は、「PostgreSQLクエリ実行の基礎知識 ～Explainを読み解こう～」にて
解説されていますので、参照して下さい。)

PostgreSQLのデフォルト設定では、seq_page_costパラメータ(シーケンシャルスキャンアクセスのコスト)は1、
random_page_costパラメータ(ランダムアクセスのコスト)は4に設定されています。
このデフォルト設定は、ランダムアクセスはシーケンシャルアクセスより、4倍コストが高い設定になっており、
ハードディスクにデータベースが格納された場合を想定し、設定されている値です。

しかし、SSDなどの高速なストレージを利用した場合には、ランダムアクセスは、シーケンシャルアクセスと比較して4倍も遅くはありません。
そのため、デフォルト設定では、ランダムアクセスの方がシーケンシャルアクセスより、
高速に結果を得られるにも関わらず、PostgreSQLがシーケンシャルアクセスを選択してしまい、
性能遅延(性能トラブル)が発生するケースがあります。

5.2.2. 性能トラブル内容

SSDをストレージに採用しているDBサーバにおいて、
SELECT文が遅延し、PostgreSQLログファイルに遅延SQLが出力されています。

■ ログの確認状況

今回例示しているPostgreSQLではSQLの遅延の閾値を3秒としているため、
「log_min_duration_statement = 3s」を指定し、遅延が発生したSQLをPostgreSQLのログファイルに出力しています。

```
LOG:  duration: 4229.749 ms  statement: SELECT * FROM sample_a JOIN sample_b ON
sample_a.id2 = sample_b.bid2 WHERE sample_a.id1 = '0500000';
```

遅延が発生するSQLは、ログファイルから抜粋した以下です。

```
SELECT * FROM sample_a JOIN sample_b ON sample_a.id2 = sample_b.bid2 WHERE sample_a.id1
= '0500000';
```

■ テーブル定義

遅延が発生するSQLでは、sample_aテーブルとsample_bテーブルのテーブル結合を実施していますが、
大規模なテーブルため、必要なインデックスをそれぞれ定義しています。テーブルの定義は以下の通りです。

```
=# \d sample_a
              Table "public.sample_a"
     Column |       Type        | Collation | Nullable | Default
    --------+-------------------+-----------+----------+---------
     id1    | character varying |           |          |
     id2    | integer           |           |          |
     date   | date              |           |          |
    Indexes:
    "sample_a_idx" btree (id2, id1)
```

```
=# \d sample_b
                  Table "public.sample_b"
     Column |       Type        | Collation | Nullable | Default
     -------+-------------------+-----------+----------+---------
     bid2   | integer           |           | not null |
     name   | character varying |           |          |
     Indexes:
            "sample_b_pkey" PRIMARY KEY, btree (bid2)
```

5.2.3. 調査・分析結果

■ 実行計画の確認

遅延が発生したSQLの実行計画をEXPLAIN ANALYZEコマンドを用いて取得したところ、
以下のように各テーブルのスキャンにシーケンシャルスキャン(Seq Scan)が選択され、
結合にはハッシュ結合(Hash Join)が選択されていました。
また、PostgreSQLが予測した行数と、実際に取得した行数(rowsにて確認)に差がないため、統計情報の更新漏れ(ANALYZEが実行されいない)状況ではないことがわかりました。

```
=# EXPLAIN ANALYZE SELECT * FROM sample_a JOIN sample_b ON sample_a.id2 = sample_b.bid2
WHERE sample_a.id1 = '0500000';
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------
---------------------------------------
 Hash Join  (cost=187056.15..570987.47 rows=460327 width=53) (actual
time=1005.652..5342.292 rows=500000 loops=1)
   Hash Cond: (sample_b.bid2 = sample_a.id2)
   ->  Seq Scan on sample_b  (cost=0.00..183332.58 rows=9999858 width=37) (actual
time=0.009..1335.758 rows=10000000 loops=1)
   ->  Hash  (cost=179054.03..179054.03 rows=460330 width=16) (actual
time=1003.073..1003.073 rows=500000 loops=1)
         Buckets: 131072  Batches: 8  Memory Usage: 3953kB
         ->  Seq Scan on sample_a  (cost=0.00..179054.03 rows=460330 width=16) (actual
time=0.012..905.575 rows=500000 loops=1)
               Filter: ((id1)::text = '0500000'::text)
               Rows Removed by Filter: 9500000
 Planning time: 0.260 ms
 Execution time: 5379.779 ms
(10 rows)
```

■ 遅延しているSQLで参照するレコード

遅延しているSQLで取得するレコードが全レコードの内、どの程度の割合で存在するのかを確認しました。
まずは各テーブルに格納されたレコード数を確認しました。いずれのテーブルでもおおよそ1,000万件のレコードが格納されていることがわかります。

```
(PostgreSQLの統計情報を用いた確認)
=# SELECT relname,n_live_tup from pg_stat_user_tables WHERE relname IN
('sample_a','sample_b');
 relname  | n_live_tup
----------+------------
 sample_a |    9999922
 sample_b |    9999858
(2 rows)
```

```
(実際のレコードを取得して確認)
=# SELECT count(*) FROM sample_a ;
  count
----------
 10000000
(1 row)
=# SELECT count(*) FROM sample_b ;
  count
----------
 10000000
(1 row)
```

問題となるSQLでWHERE句の条件で指定されている、「sample_a.id1 = '0500000'」に合致するレコードが
どの程度の割合で存在するのかをか確認しました。
以下SQLの結果より、「0500000」の値を持つレコードは、「0.0460333」(おおよそ4.6%)の割合で存在することがわかりました。

```
(PostgreSQLの統計情報を用いた確認)
=# SELECT * FROM pg_stats WHERE tablename = 'sample_a' AND attname = 'id1';
-[ RECORD 1 ]----------+-----------------------------------------
schemaname             | public
tablename              | sample_a
attname                | id1
inherited              | f
null_frac              | 0
avg_width              | 8
n_distinct             | 5
most_common_vals       | {[省略],0500000}
most_common_freqs      | {[省略],0.0460333}
histogram_bounds       |
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

```
(実際のレコードを取得して確認)
=# SELECT count(*) as record_num ,count(*)/10000000::FLOAT * 100 AS percent FROM
sample_a WHERE id1 = '0500000';
 record_num | percent
------------+---------
     500000 |       5  -- id1 = '0500000';を満たす行数は500,000件存在し、全体の5%
(1 row)
```

■ コストパラメータの変更

sample_aテーブル全体のレコード内、5%のレコードを取得するSQLであるため、

シーケンシャルスキャンよりインデックスを利用して実行した方が高速に処理される可能性があると考え、
試しにrandom_page_costパラメータ(ランダムアクセスのコスト)を4から2に変更し、実行しました。
random_page_costを2変更した結果、SQL実行時間が、5379.779 ms⇒ 886.744 msに改善しました。
SQLの実行計画も、シーケンシャルスキャンから、インデックススキャンに変わり、マージ結合が実施されるようになりました。

```
=# SET random_page_cost TO 2;
=# EXPLAIN ANALYZE SELECT * FROM sample_a JOIN sample_b ON sample_a.id2 = sample_b.bid2
WHERE sample_a.id1 = '0500000';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
-------------------------------------------------------
 Merge Join  (cost=5.88..524232.63 rows=460327 width=53) (actual time=0.027..849.794
rows=500000 loops=1)
   Merge Cond: (sample_a.id2 = sample_b.bid2)
   ->  Index Scan using sample_a_idx on sample_a  (cost=0.43..205313.90 rows=460330
width=16) (actual time=0.012..522.793 rows=500000 loops=1)
         Index Cond: ((id1)::text = '0500000'::text)
   ->  Index Scan using sample_b_pkey on sample_b  (cost=0.43..288175.30 rows=9999858
width=37) (actual time=0.009..118.594 rows=500000 loops=1)
 Planning time: 0.835 ms
 Execution time: 886.744 ms
(7 rows)
```

5.2.4. 性能トラブルの改善案

SSDなどの従来のハードディスクと比較すると、ランダムスキャンが高速に実行可能なストレージ装置を利用する場合においては、
random_page_costパラメータをseq_page_costパラメータに近い値に設定すると、SQLの実行でランダムスキャン(インデックススキャン)が
選択され、性能が改善したとの報告がされています。

[翻訳] たった一つの設定変更が如何にしてクエリのパフォーマンスを50倍も改善したか
(How a single PostgreSQL config change improved slow query performance by 50x)
http://pgsqldeepdive.blogspot.jp/2017/11/50-how-single-postgresql-config-change.html

本事例で紹介したSQLでも、random_page_costパラメータを1に近づけることで、SQLの性能を改善することが確認できました。

5.2.5. 備考

SSDの環境において、インデックススキャンとシーケンシャルスキャンの性能差を確認するため、簡単な試験を実施しました。
pgbenchで生成した、10,000,000(1,000万件)のデータが格納されるpgbench_accountsテーブルを利用しています。

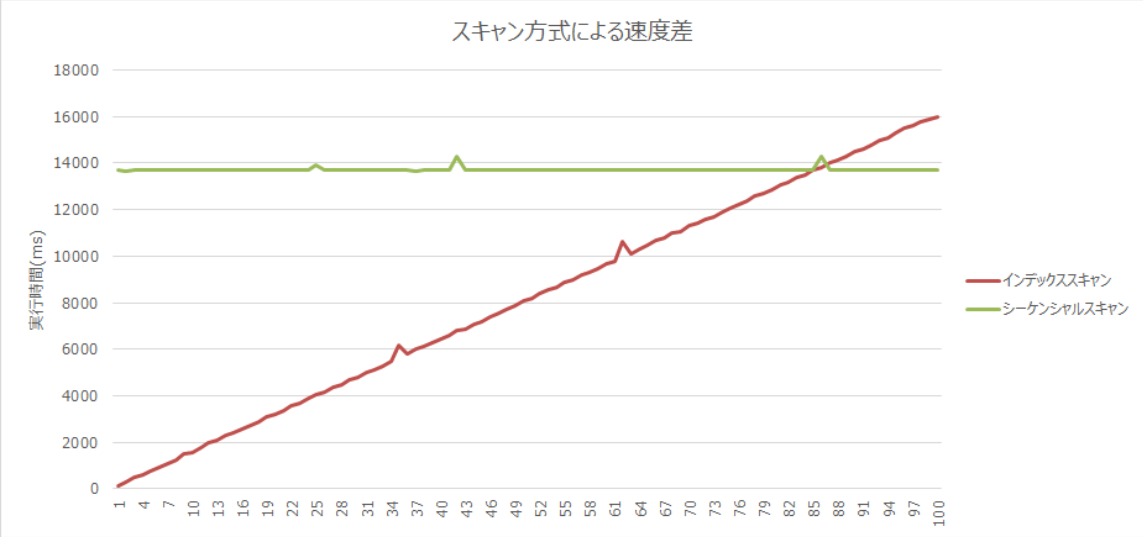
SQLで取得するpgbench_accountsテーブルのレコードの割合を1～100%(1%ずつ増加)変化させて、下記トランザクションの実行時間を測定しました。また、下記トランザクションを実行する前に下記コマンドでキャッシュの破棄を実行しています。

```
(シーケンシャルスキャンで実行するSQL)
=# set enable_indexscan to 'off';
=# set enable_bitmapscan to 'off';
=# SELECT * FROM pgbench_accounts WHERE aid >= 1 and aid <= 100000 * [レコードを取得する割合(1～
100に変化させる)]
```

```
(インデックススキャンで実行するSQL)
=# set enable_seqscan to 'off';
=# set enable_bitmapscan to 'off';
=# SELECT * FROM pgbench_accounts WHERE aid >= 1 and aid <= 100000 * [レコードを取得する割合(1～
100に変化させる)]
```

```
(キャッシュを破棄するために実行したコマンド)
$ pg_ctl restart
$ echo 1 > /proc/sys/vm/drop_caches
```

結果は以下のようになり、インデックススキャン(ランダムスキャン)の実行時間は、
テーブル内のすべてのレコードを取得する際に、1.16倍となりました。
インデックススキャン : 16000.297ms
シーケンシャルスキャン : 13688.574
インデックススキャンはシーケンシャルスキャンと比較すると、
インデックス分読み込むサイズが大きいにも関わらず、1.16倍の差のみのため、
random_page_costを1に近い値に設定する設定に大きな問題はないと考えます。



5.3. 適切でない実行計画が選択されてしまうことによる性能トラブル

5.3.1. 概要

ディスク性能の考慮漏れによる性能トラブル で記載していますがPostgreSQLは統計情報をもとに実行計画のコストを推定し、最もコストの低い実行計画に基づいてクエリを実行します。

そのため統計情報とデータベース内のデータの実態に剥離があった場合は適切でない実行計画が選択されてしまい、性能劣化(性能トラブル)が発生してしまうケースがあります。

下記に「適切でない実行計画が選択されてしまった場合に発生する性能トラブル」の事例を記載します。※事例内のオブジェクト名や数値については架空のものとして記載しております。

5.3.2. 性能トラブル内容

夜間に大規模な業務データに対し集計処理をバッチで実施するシステムを構築、運用していましたが、運用開始後データが蓄積/更新されていくにつれて夜間のバッチ処理が遅延し、当初の想定時間内に終わらなくなってきました。

このままではいつか、翌営業日までに終わらなくなるかもしれません。

5.3.3. 調査・分析結果

データベース性能に影響を与える要素 の記載内容をもとに、バッチ処理内のSQLの実行状況(ロックウェイト、一時ファイル、スロークエリ、explain)からボトルネックを特定していきます。

log_min_duration_statementを設定し、以下のスロークエリを特定しました。

```
LOG: duration: 14000123.456 ms statement: select public.func_sample();
```

次に当該クエリの実行計画を確認しようとしましたが、このクエリは関数が用いられており、通常のEXPLAINでは関数内で実行されるSQLの実行計画を取得することができません。

そこで、一定時間内に実行された処理の結果 でも紹介しているauto_explainをインストール、設定します。ANALYZEの結果も取得することと、関数内のSQL文の実行計画を取得するため以下の設定を施します。

```
SET auto_explain.log_analyze = true;           --EXPLAIN ANALYZEの結果を取得する。
SET auto_explain.log_nested_statements = true; --関数内のSQL文の実行計画も取得する。
```

そうして実行計画を確認したところ、以下のような結果が判明しました。

適切でないジョイン方式が選択されていた

構築時の評価、当初の想定では当該のクエリは効率的なハッシュジョインとなるべきでしたが、マージジョインが選択されてしまっていました。

```
Merge Join  (cost=592815.42..8795552.29 rows=189878 width=669) (actual
time=59000.961..13739668.541 rows=292192 loops=1)
```

インデックススキャンの時間がクエリ全体の実行時間をほぼ占めていた

test_idxインデックスのスキャン時間(13704s)がほぼクエリ全体の実行時間(14000s)となっていました

```
Index Scan using test_idx on sample_test t  (cost=0.00..7978601.06 rows=4237321
width=669) (actual time=13.852..13704413.467 rows=4314638 loops=1)
```

当該のクエリはマージジョインが選択されていましたが、マージジョインがソート済みの結果に基づいて処理を行うために、ソート済みが保証されるtest_idxインデックスを利用したと考えられます。

5.3.4. 実施対応策

当該のクエリはハッシュジョインが効率的と想定しているため、当該のクエリを実行する前に、enable_mergejoinパラメータをoffに設定することでハッシュジョインを選択させるようにしました。

```
SET enable_mergejoin TO OFF;
```

クエリの実行後はDEFAULTに戻しておきます

```
SET enable_mergejoin TO DEFAULT;
```

5.3.5. その後の展開

ハッシュジョインが選択されるようになり実行時間は短縮されましたが、しばらく経った後にまた実行時間の遅延が見受けられるようになりました。

auto_explainにて再度実行計画のANALYZE結果と、テーブルの状況を確認したところ、ハッシュジョインにおけるハッシュ作成対象テーブルについて、よりデータ件数が多いテーブルが選択されていたため非効率な走査になっていました。

```
Hash Join  (cost=1513012.17..8320833.14 rows=490606 width=669) (actual
time=801456.867..1626321.178 rows=298112 loops=1)
   Hash Cond: (((testA_small.tenant_id)::text = (t.tenant_id)::text) AND
((testA_small.member_id)::text = (t.member_id)::text) AND (testA_small.use_date =
t.use_date) AND ((testA_small.use_type)::text = (t.use_type)::text) AND
((testA_small.data_type)::text = (t.data_type)::text) AND ((testA_small.note)::text =
(t.note)::text))
   ->  Seq Scan on testA_small  (cost=0.00..6054.33 rows=295933 width=14) (actual
time=3.931..1941.207 rows=298112 loops=1)
   ->  Hash  (cost=745488.05..745488.05 rows=4356205 width=669) (actual
time=790818.816..795818.816 rows=4325556 loops=1)
         Buckets: 1024  Batches: 16384  Memory Usage: 396kB
         ->  Seq Scan on testB_big t  (cost=0.00..725488.05 rows=4356205 width=669)
(actual time=0.022..238413.991 rows=4325556 loops=1)
 Total runtime: 1626401.249 ms

・推奨されるハッシュ作成対象であり件数の少ないテーブル: testA_small
・今回のハッシュ作成対象であり件数の多いテーブル: testB_big
```

ハッシュ対象テーブルの選択についてはPostgerSQLのプランナが対象テーブルのデータの特徴などに基づいて決定するため、統計情報が持つデータ件数と実際のデータ件数との間にズレがあったからと考えられます。

VACUUM ANALYZEを実施し統計情報を最新化すると適切なテーブルが選択されるようになったので、手順としてANALYZEの定期実行を組み込むようにし、最適な実行計画が策定されるように改善させました。

5.3.6. 備考

本番運用に入ると実データの蓄積傾向により事前検証では予想もしなかったネックが発生します。本件は統計情報と実態の剥離が発生したことにより実行計画が正しく策定されなかったことがトラブルの原因となりましたが、設計/構築時の評価と運用開始後のギャップに備えるためにも以下のような対策を事前に検討しておくことを推奨いたします。

- auto_explainを組み込んで解析を容易にしておく
- 統計情報の最新化のため定期的なVACUUM ANALYZE実施を手順に組み込む
- 実行計画制御のため、pg_hint_planやpg_dbms_statsを後から組み込める余地を残しておく

6. 著者

（企業・団体名順）

版	所属企業・団体名	部署名	氏名
第1.0版 (2017年度WG3)	株式会社富士通ソーシアルサイエンスラボラトリ	ソリューション開発センター ソリューションビジネス部	小山田 政紀
	日本電気株式会社	AIプラットフォーム事業部	湯村 昇平