SQL Anywhere のパフォーマンスとチューニング

Jason Hinsperger

Sr. Product Manager

SAP

アジェンダ

パフォーマンス分析
- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類
- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

パフォーマンス分析

主な SQL Anywhere パフォーマンス要素

- アプリケーションのパターン
- サーバーキャッシュサイズ
- 現在のキャッシュコンテンツ (コールド、ウォーム、ホット)
- サーバーマルチプログラミングレベル (-gn/gnh コマンドラインスイッチ)
- サーバースレッドの実行に必要な利用可能なCPU数 (ライセンス依存)
- データベースワーキングセットサイズ
- サーバーのディスクサブシステムのスピードと設定
- ワークロードの特徴: トランザクションの到着時間間隔、接続数、ワークロードミックス
- データベースページサイズ
- 他の二次的要因の可能性

パフォーマンス問題の検出

問題へのシステマティックなアプローチ:「アプリケーションが遅いのですが。どうしたら良いでしょうか？」

リソースが最大限に達しているものがあるためパフォーマンスが悪い

- マシンレベルでリソースを制限している:
  I/O 帯域幅
  CPU サイクル
- マシンは、より多くの I/O や CPU を並列で使用可能かもしれない
  しかし、サーバーがそれらを並行して使用できない状態かもしれない → コンカレンシーバウンド

ボトルネックの種類:

- クライアントアプリケーションの問題
- IOバウンド
- CPUバウンド
- コンカレンシーバウンド

SQL Anywhere パフォーマンス分析ツール

SQL Anywhere プロファイリングツール

いくつかの異なるツールの機能を組み合わせ (それぞれ個別でも利用可能)

- リクエストロギング
- プロシージャープロファイリング
- グラフィカルプランキャプチャリング
- インデックスコンサルタント
- 統計情報モニタリング

→問題がサーバーにあるのかクライアントにあるのかの判断に役立つ

イベントトレーシングの方がよりターゲットできる

興味のある特定のイベントをトレースできる (グラフィカルプランなど)

サーバー内で use sp_read_etd() を使用可能

アジェンダ

パフォーマンス分析

- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類

- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

「毎週日曜 09:00 に全てが遅くなる！」

問題の期間に何のステートメントが実行されているのか判別する
- min/avg/max 時間を「良い」期間と「問題」の期間で比較する
- sa_conn_info and sa_conn_activity, sa_performance_diagnostics を使用する
  - ReqTimeBlockIO, ReqCountBlockLock, ReqCountBlockContention のステートメントをチェックする
- 「良い」と「悪い」で最も異なるプロシージャー / ステートメントを探す
- ロッキングの問題をチェックする
- サーバーまたはネットワークの他のアクティビティを検討する (バックアップ、ルータの再設定、 …)
- 実行プランを比較する

オプティマイザはなぜシーケンシャルスキャンを選択するのか?
- 述部の選択性、キャッシュコンテンツをチェックする
- sysphysidx (seq_transitions, rand_transitions) のインデックスクラスタリング統計情報をチェックする

アジェンダ

パフォーマンス分析

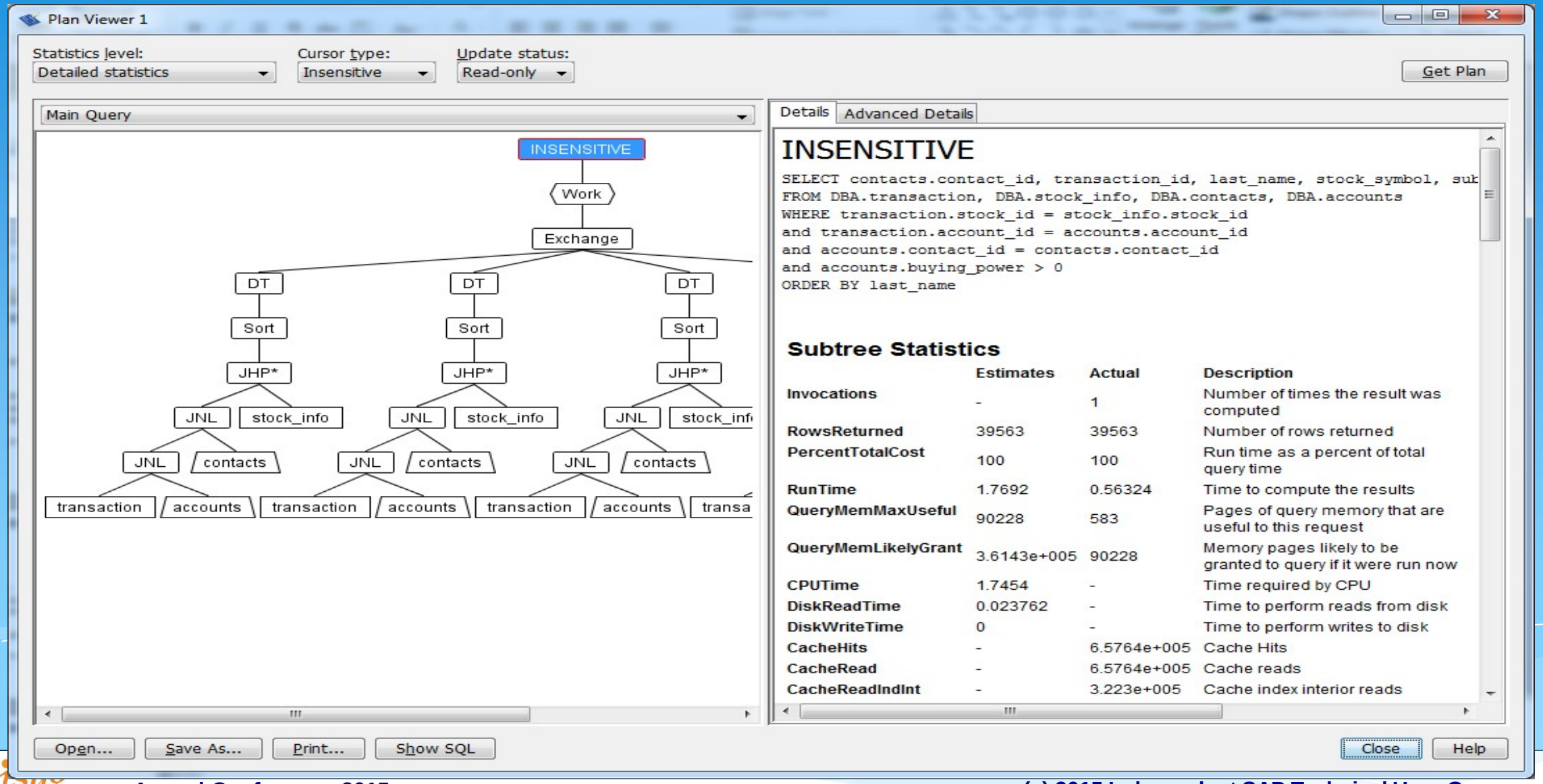
- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類

- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

グラフィカルなプランビューワ



グラフィカルなクエリプラン

クエリへの詳細実行プランをグラフィカルなフォーマットで表示する。
物理オペレーターのツリーとして実行プランを表示

- 例えば ハッシュ結合、テーブルスキャン、ソート
- エッジの厚さは、チャイルドオペレーターによって生成された行数を表します。
- ボックスの色は「速い」から「遅い」までのオペレーターの比較コストを表しています。
- ツリーのどのオペレーターをクリックしても詳細情報が得られます。

グラフィカルなクエリプラン

チェックすべきこと:

- 述部の選択性
  - 選択性は理に適っているか?
  - レポートされている選択性のソースは何か?
- Join の選択性: PK-FK 制約, etc.
  - できるかぎり、Foreign Keys を常に使用する
- 最適化のゴールをチェックする
- レスポンスの最適化の実行時の order-by のインデックススキャン (‘first-row’)
  - 小さなソートでなければ、存在するのであれば、インデックスを使用するのがベスト
  - さもなくば、追加することを検討する

グラフィカルなクエリプラン

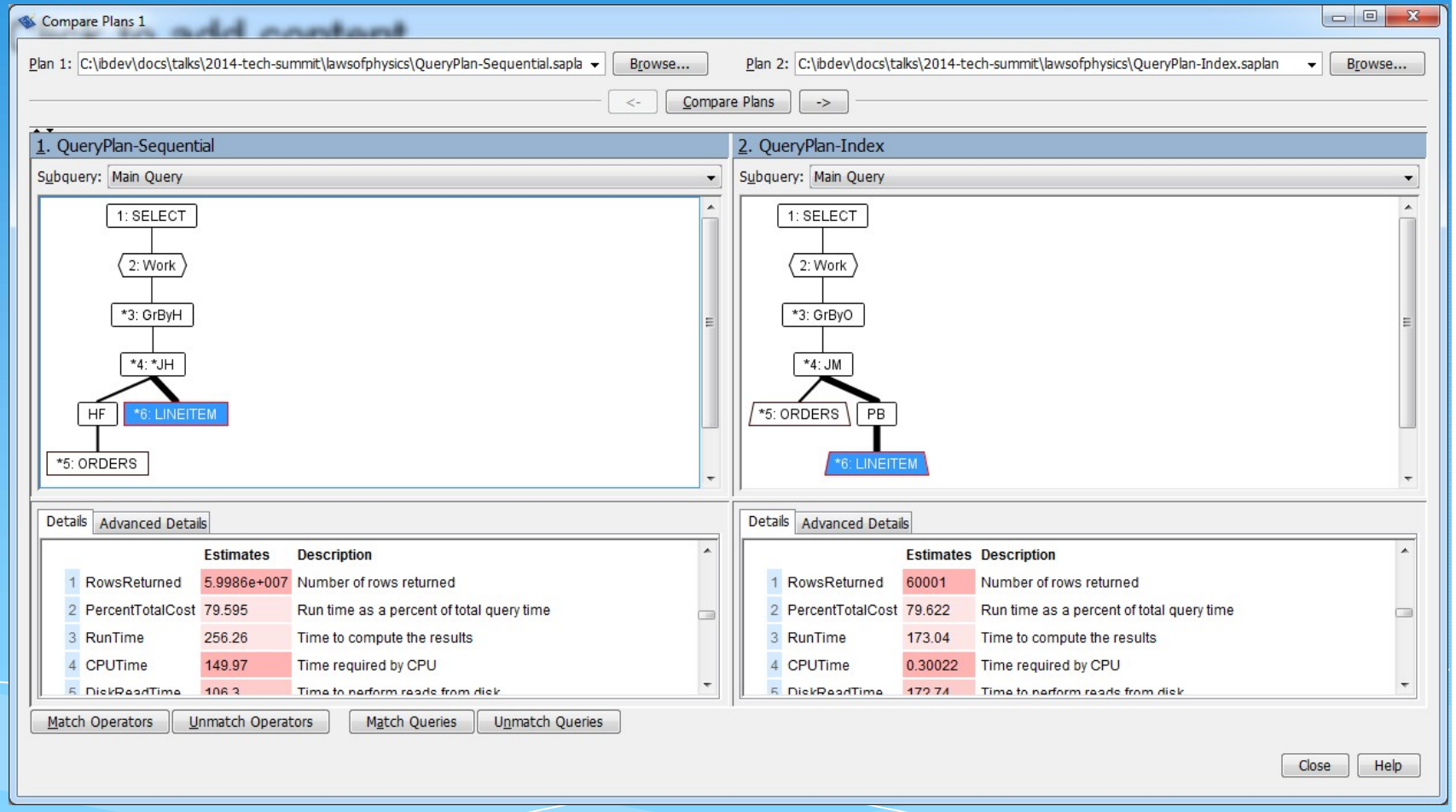
さらにチェックすること:

- 現在キャッシュにあるデータをチェックする
  - キャッシュがコールドの場合は通常シーケンシャルスキャンの方がインデックススキャンより良い
  - インデックススキャンはホットキャッシュではほぼ常にうまく機能する
  - (インデックス探索はシーケンシャルスキャンまたはインデックススキャンよりも良いことが多い)
    - 多数の行を返す場合、シーケンシャルスキャンの方が良い可能性が高い (テーブルページで40:1)
- コストが高いサブクエリをチェック
- 予測 vs. 実際の値 (「統計情報」でプランが設計されている場合)
  - 行数
  - 実行コスト

グラフィカルなプランの比較

保存したプランを2つ選択する

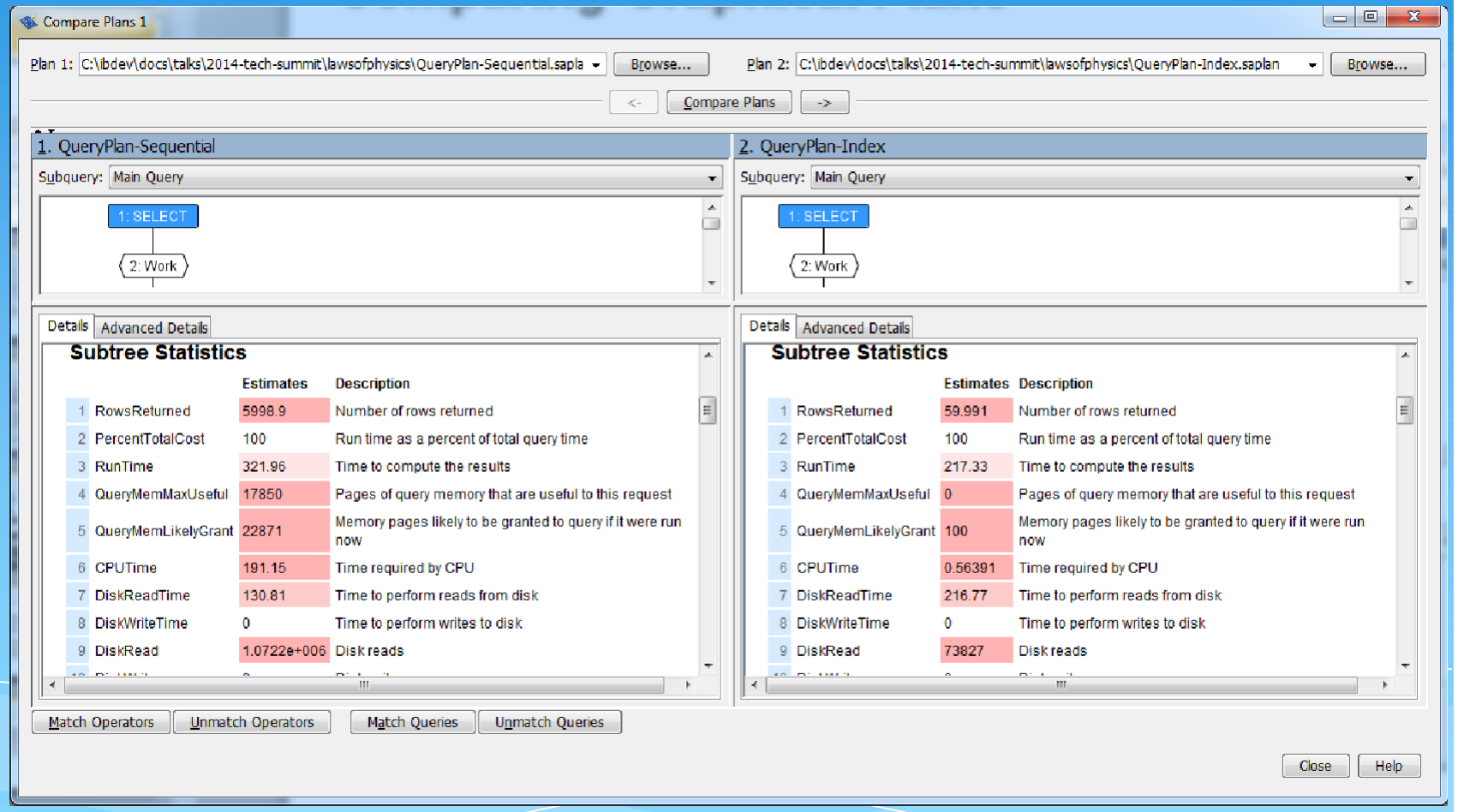
マニュアルでいくつかのノードをマッチさせる
違いをハイライトする
キャッシュしたページを検討する
行数 / 選択性 をチェックする



グラフィカルなプランの比較（続き）

違いをハイライトする

行数は通常重要
　　選択性の違い
option / config の違いをチェックする



アジェンダ

パフォーマンス分析

- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類

- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

パフォーマンスタイミングユーティリティ

What?
- パフォーマンスをテストするのに使えるユーティリティがある
- サーバーとdb 設定から「物理の法則」の兆候を提供

When?
- 正確なタイミングを得るためには、統計情報のグラフィカルなプランではなくこれらのツールを使用する

How?
- %SQLANYSAMP16% にある
- このユーティリティと同じフォルダ内の Readme.txt 内に説明を掲載

dbping システムユーティリティ

dbping ユーティリティは接続のレイテンシを評価するのに使用することができる
–s (と –st) で、より詳細統計情報を計測することができる

統計情報	詳細
DBLib connect and disconnect	DBLib 接続と切断を実行する時間
Round trip simple request	クライアントからサーバーにリクエストを送信するのにかかる時間、プラス、サーバーからクライアントにレスポンスを返すのにかかる時間。往復時間は、平均レイテンシの2倍
Send throughput	100KB/iterationのスループット
Receive throughput	100KB/iterationのスループット

クエリパフォーマンスのテスト

Fetchtst (or odbcfet)

- ¥Samples¥SQLAnywhere¥PerformanceFetch
- 任意のクエリのフェッチ速度を測定
- (デフォルトでは test.sql) ファイルに任意のクエリを置く。“go”で分離
  - SELECT 文を使用することが多いが、update と insert も同様に機能する

便利なスイッチ

- -ga – いくつかの便利な統計情報を測定
- -j nnn – ファイルを数回リピート (または、それぞれのステートメントに –js)
- -p – プランを印刷 (1つのクエリに対するプラン変更を表示)
- -i – 思考時間

テスト時に設定をマッチさせることを試みる

- 分離独立性、カーソルタイプ、プリフェッチ、接続

fetchtst 結果例

SQL Step	count	seconds	min.s/i
plan	10	0.002706	0.000262
PREPARE	10	0.000420	0.000039
DESCRIBE	10	0.000566	0.000054
OPEN	10	0.000293	0.000027
FETCH first row	10	0.000562	0.000052
FETCH remaining rows	10	6.451974	0.581380
CLOSE	10	0.000564	0.000055
DROP	10	0.000041	0.000004
EXECUTE (described non-query)	0	0.000000	
EXECUTE IMMEDIATE (non-query)	0	0.000000	
Total	10	6.457126	0.581962
Total elapsed for whole run	1	6.515817	6.515817
Engine CPU usage (tot)	10	5.288434	0.436803
Engine CPU usage (usr)	10	5.288434	0.436802
Engine CPU usage (sys)	10	0.000000	0.000000
select pk	10	6.457126	0.581962

挿入のパフォーマンスをテスト

Instest
- ¥Samples¥SQLAnywhere¥PerformanceInsert
- 1つのテーブルの挿入パフォーマンスを測定
- ファイルからクエリを読み込み
- PUT を使用して行を挿入

設定をマッチさせることを試みる
- 頻繁にコミット
- 幅を挿入 (リクエスト毎に挿入する行)
- 事前にデータベースファイルを大きくし、確実にcontiguous近接することを確認する

パフォーマンスタイミングユーティリティ

Trantest

- ¥Samples¥SQLAnywhere¥PerformanceTransaction
- あるサーバー設定、データベース設計、トランザクションのセットにおいて、ハンドリング可能な負荷を測定する
- そのサーバーに対してトランザクションを実行しているクライアントマシンの数をシュミレーションする
- 何のトランザクションを実行するか定義する
- 複数のクライアントマシン上で実行可能: master/slave

パフォーマンス問題の種類

パフォーマンス問題の幅広いクラス

時間がかかりすぎるコミュニケーションパターン / チープなリクエスト
- リクエストを形成し、結果を解釈する時間
- ネットワークのレイテンシ
- ビルド / オープン時間

Expensive リクエスト（複雑な、重いクエリ）
- 本質的に expensive なクエリ
- クエリの処理が expensive になる設定
- クエリプランの品質

遅い同時接続リクエスト
- ロックや他のブロッキングが原因の相互除外
- 共有リソースの同時使用はシリアルより遅くなる可能性が高い

ワークロードの大まかなカテゴライズ

評価 (総時間 / # リクエスト)

- 平均リクエスト時間が < 100ms の場合、問題はチープなリクエストの可能性が高い

–zt サーバーオプションと dbo.sa_performance_diagnostics() を使用

- ReqTimeActive または ReqTimeBlockIO が高い場合 – expensive なリクエスト
- ReqTimeBlockLock または ReqTimeBlockContention が高い場合 – 同時リクエストが遅い

同じパラメーターで実質1つのクエリのパフォーマンスが変わる場合、クエリプランの違いの可能性がある

- キャッシュコンテンツ、同時接続のアクティビティ、…などの違いを確実に除外する
- 計測時間内に11回以上SQL実行を行う場合、プランキャッシングに関係している可能性がある

アジェンダ

パフォーマンス分析

- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類

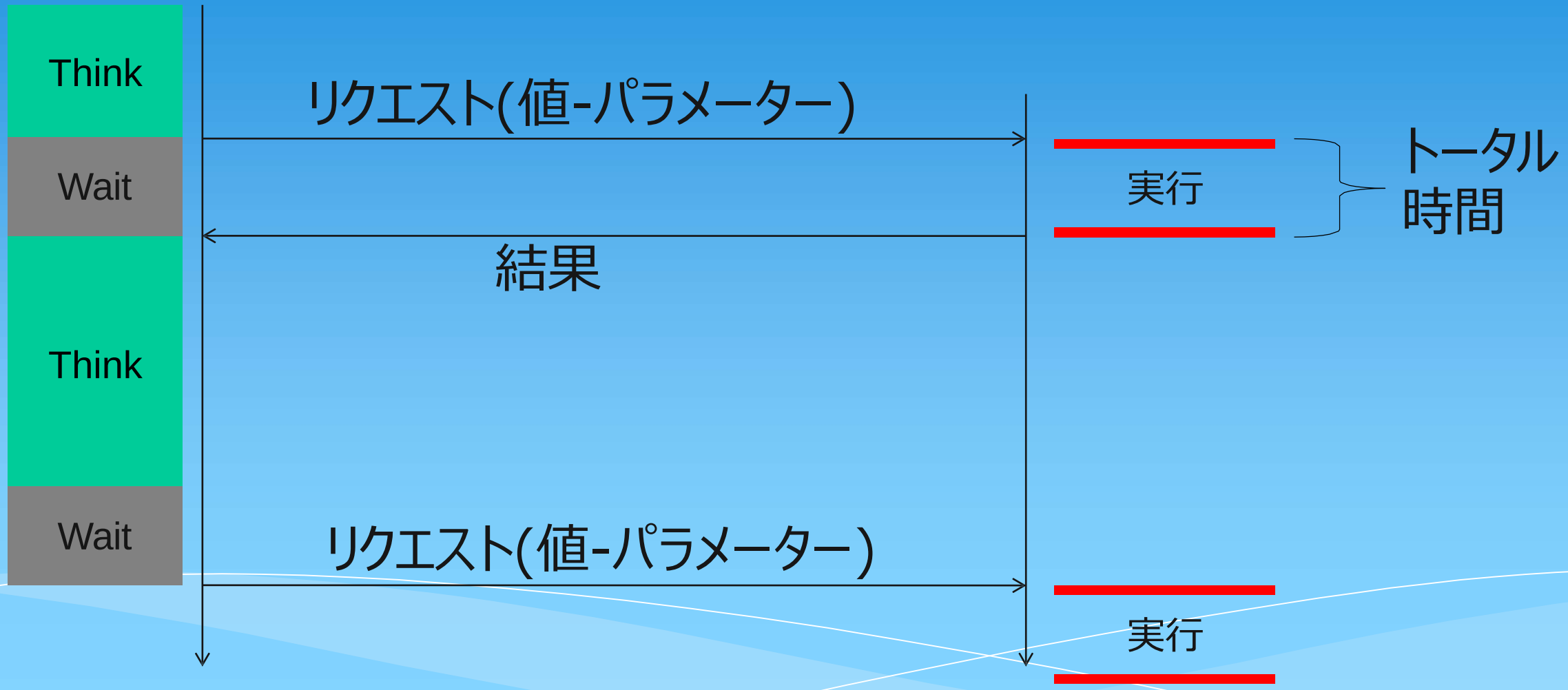
- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

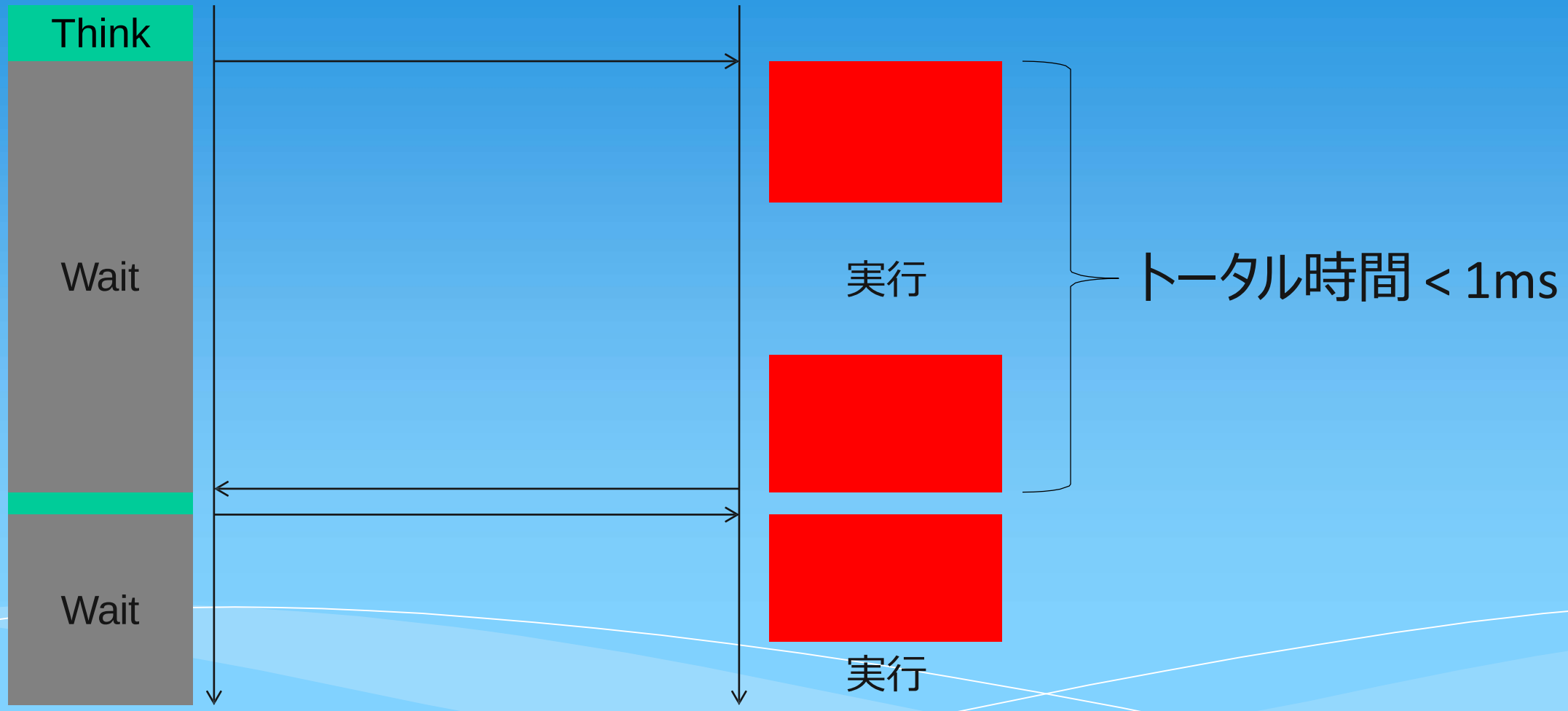
チープなリクエストを多く使用するとオーバーヘッドを繰り返す

- サーバーオーバーヘッド
  - パーシング SQL
  - 構築プラン
- インタープロセス通信
  - レイテンシ
- クライアントオーバーヘッド
  - 文の構築
  - 結果の処理を開始

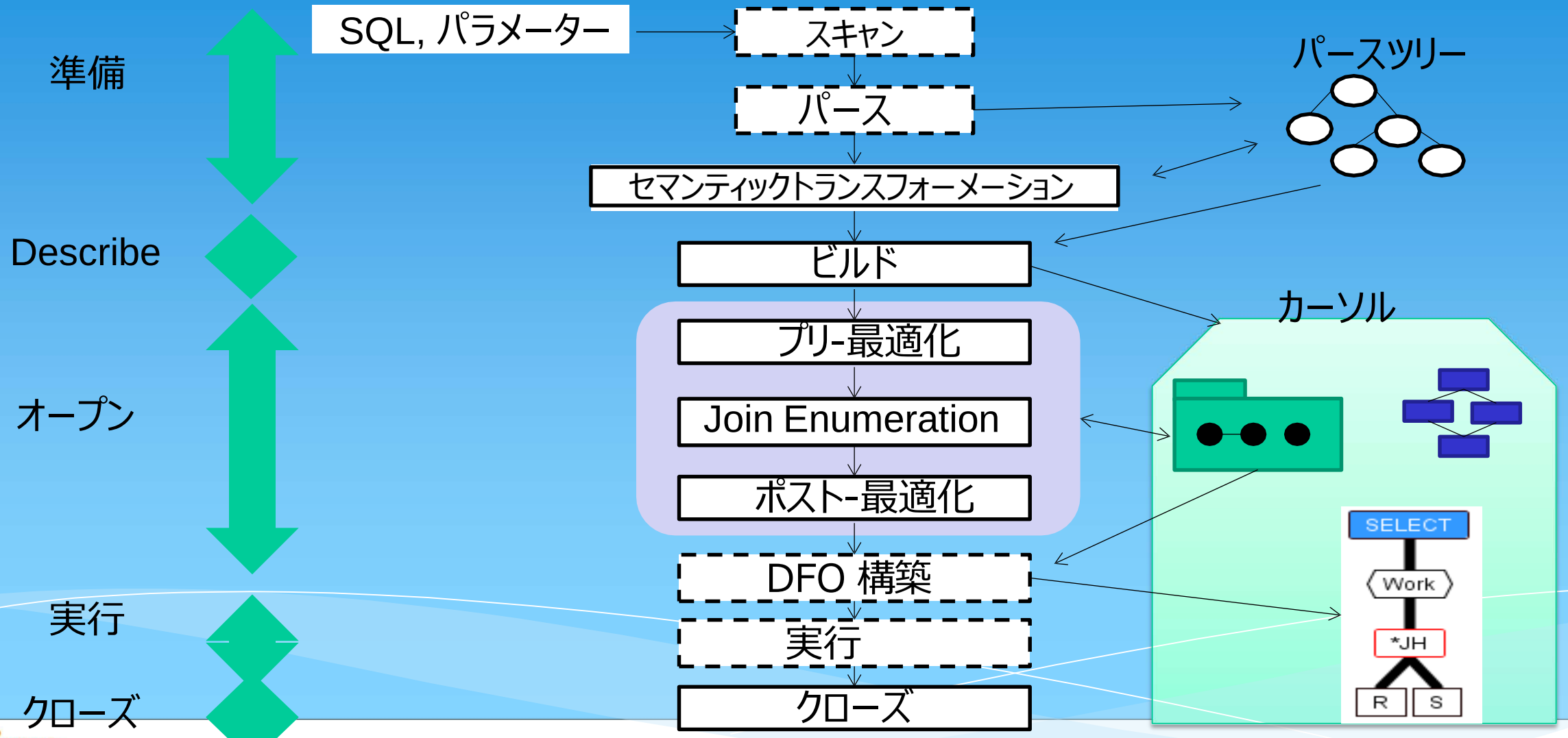
トータルリクエスト時間



ミリ秒ワークロード

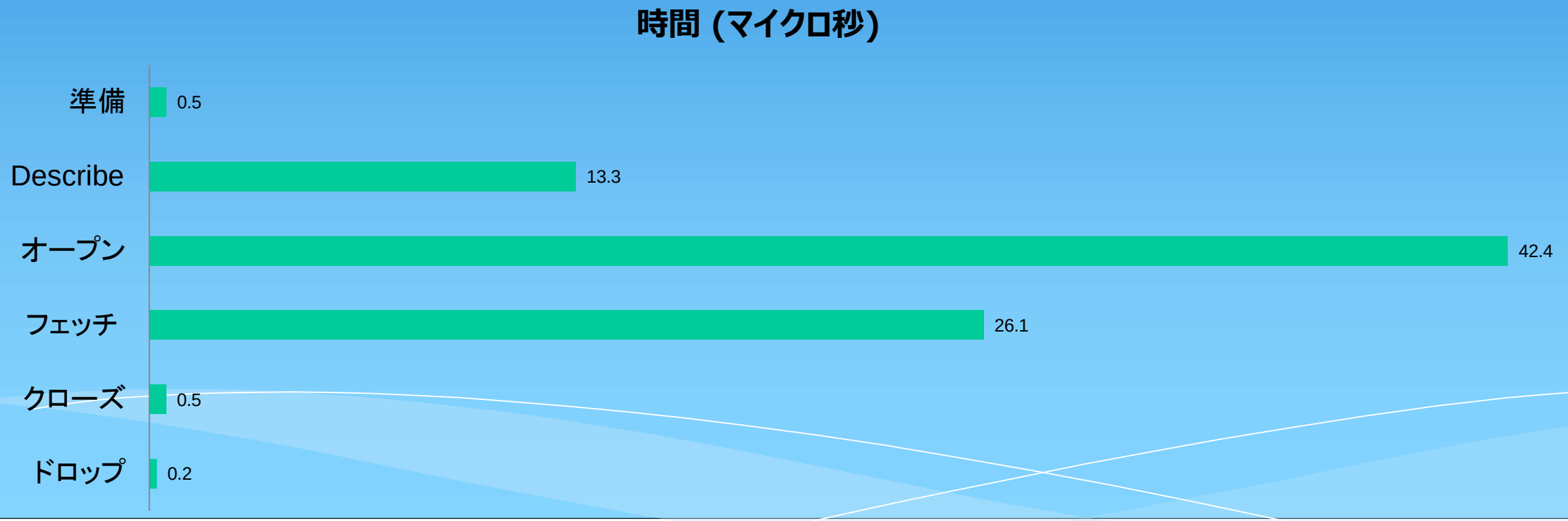


サーバーオペレーションの詳細

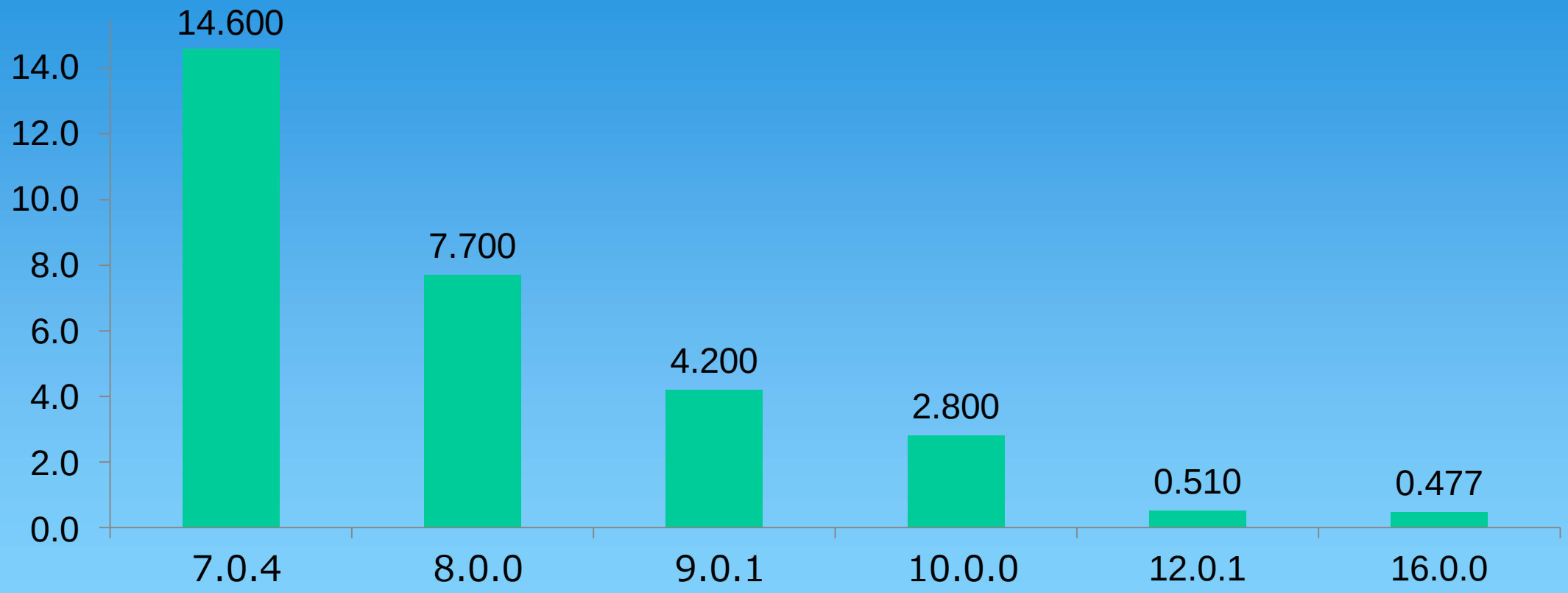


チープなリクエストにおけるオーバーヘッドの影響

```
R1: SELECT * FROM T WHERE T.pk = 100
```



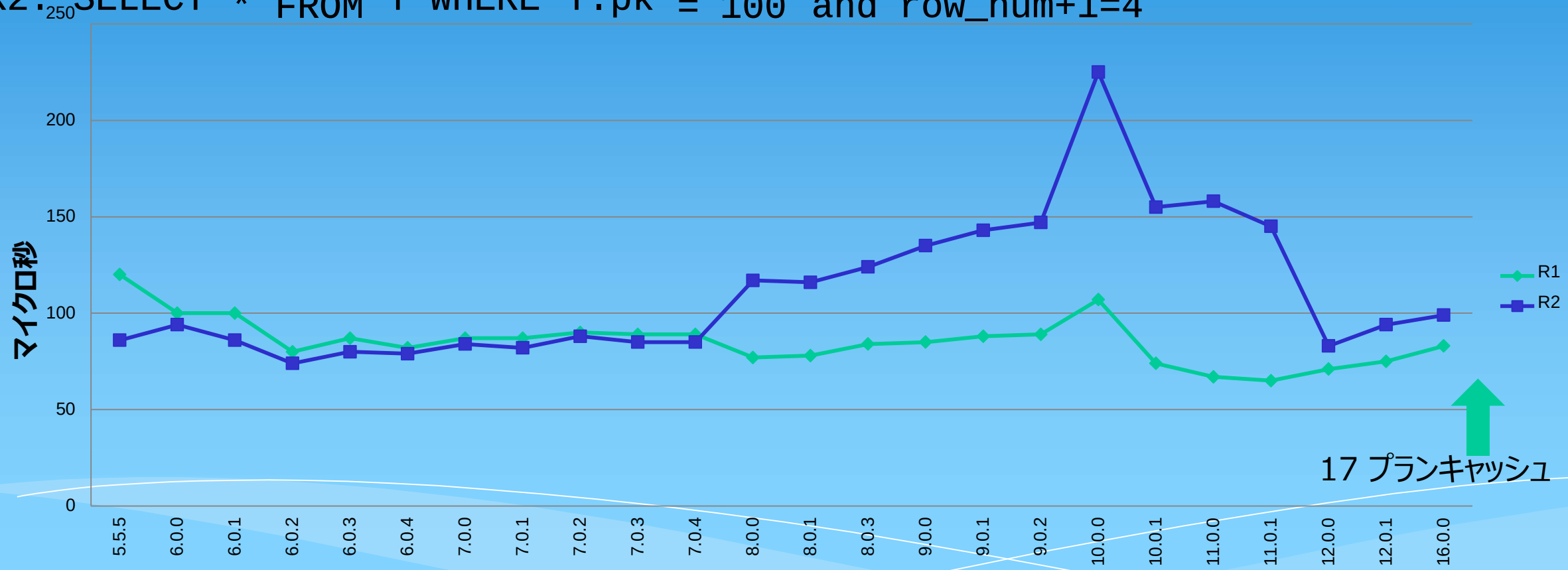
バージョンごとのエクスペンシブな文



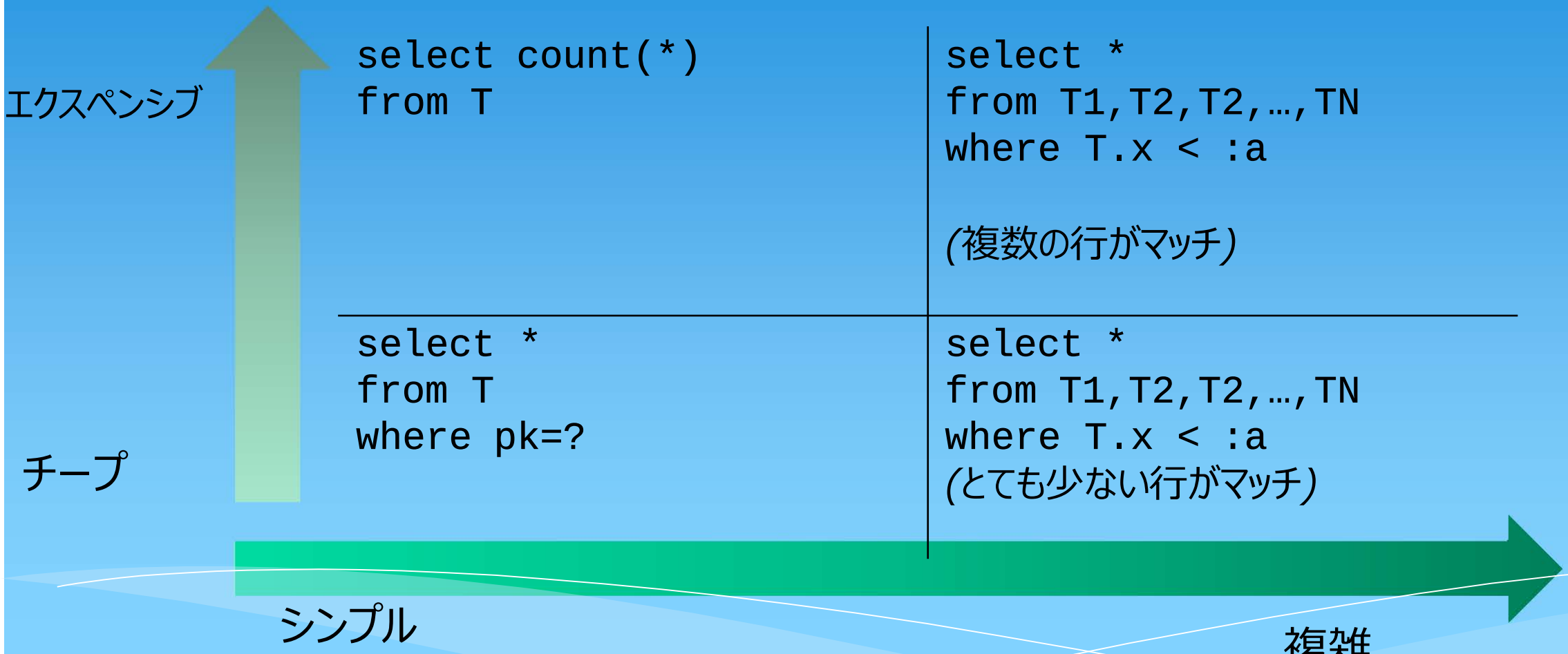
平均 TPC/H SF10 クエリ時間 (分)

バージョンごと シンプルな文のパフォーマンス

```
R1: SELECT * FROM T WHERE T.pk = 100
R2: SELECT * FROM T WHERE T.pk = 100 and row_num+1=4
```



リクエストのカテゴライズ



アジェンダ

パフォーマンス分析
- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類
- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

I/O-バウンドアプリケーション

検出方法 ?

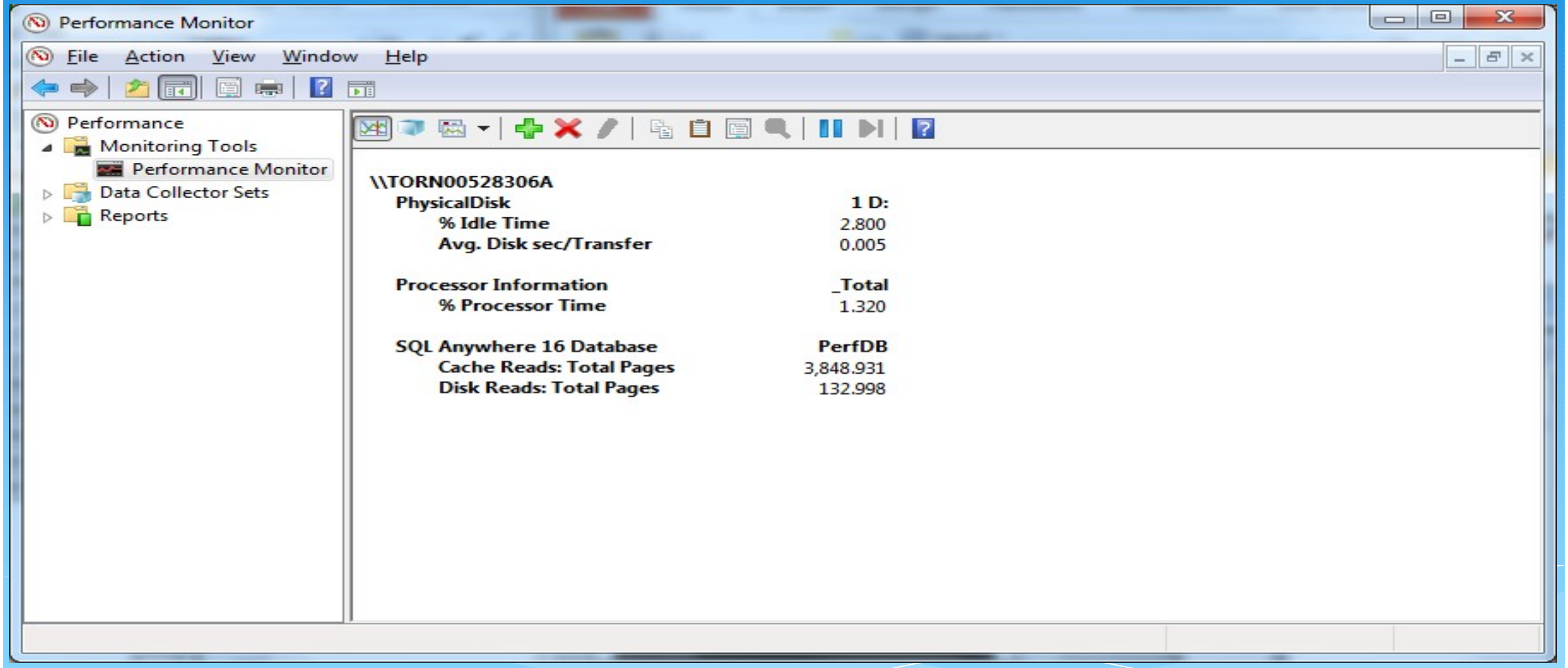
- サーバーは遅いがCPU-バウンドはそうではない
- Windows Task Managerd で、データベースサーバープロセスになる読み込みと書き込みをチェックする
- perfmon で、物理ディスクオブジェクトの %Idle タイムカウンターをチェックする – 1% よりも下の場合、サーバーは I/O-バウンド
- Average sec/Transfer と ReqTimeBlockIO を検討する

判別チェック – うるさく動いている、LEDが頻繁に光っているハードドライブ

I/O-バウンドのアプリケーションは、追加のディスクハードウェアが必要な可能性がある

- SSD の使用が可能。RAID とつなげることもできる。

Windows perfmon



キャッシュサイズが小さすぎる

サーバーがバッファープール内の頻繁に使用されているデータベースページをキープできない場合、スラッシングが発生する。
perfmon または tracing でSQL Anywhere カウンタを使用して検出可能：

- CacheReadTable vs. DiskReadTable
- CacheReadIndex vs. DiskReadIndex

これらのカウンタは、絶対値であるため、ある固定の時間の拡大をチェックする

- CacheReadTable がDiskReadTable よりも10倍速く、CacheReadIndexが100倍以上速く大きくなることをチェックする必要がある
- そうでない場合、キャッシュサイズが小さすぎる可能性があることを示している。

インデックスがない

適切にチューニングされたインデックスは、I/Oの要求を大幅に削減することができる。
- 全行を読み込むのではなく、1 つのクエリを満たすのに必要な行のみを読み込む

ベストな調査方法は、インデックスコンサルタントを使用すること
- SC Profiling Mode または DBISQL から利用可能
- インデックスコンサルタントで、マニュアルで CREATE VIRTUAL INDEX 文を使用しても実行可能
- アプリケーションクエリを検査

インデックスのクラスタリングもまた重要
- sys.sysphysidx のクラスタリング統計情報を systab の clustered_index_id と比較

クエリプロセッシングメモリ

通常の方法では、サーバーはクエリの処理に十分なメモリを得られずexpensive な low-memory 戦略をとらなければならない

- あまりにも小さいキャッシュの特別なケース
- 発生する可能性が高いもの – -gn 値 (100 より上)がとても高い、またキャッシュサイズが小さい
  - 使用する場合には、このように高い値が本当に必要なのか確認する

これが問題の場合には、プロファイリングモードで、expensive と判別されたクエリは、ハッシュオペレーターを使用しない

- あるいは、それらを使用する場合、オペレーターの詳細は、低メモリ戦略に対して再版または複数のパスを示す。
- QueryMemMaxUseful と QueryMemLikelyGrant を比較する

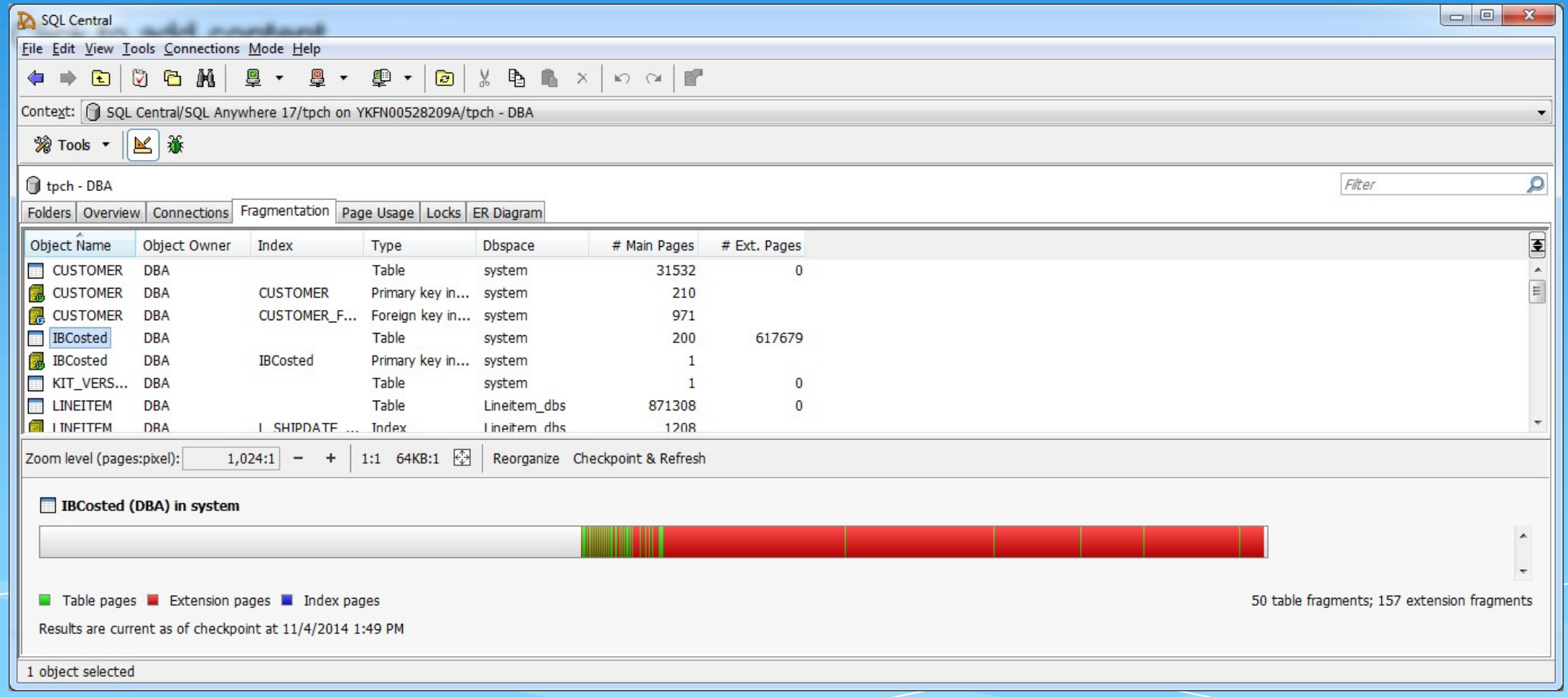
max_statement_count/max_cursor_count オプションをチェックし、リーク（不必要な・クローズし忘れ）をチェック

フラグメンテーション

OS ファイルフラグメンテーションをチェックする
- スタートアップのサーバーウィンドウ、DBFileFragments プロパティ
- OSから提供されたツールを使用して直す
  - 例 contig.exe は、www.sysinternals.com から無償で入手可能

テーブルとインデックスのフラグメンテーションをチェックする
- sa_table_fragmentation()、sa_index_density()、または SQL Central Fragmentation タブ
- 修正
  - REORGANIZE TABLE
  - データベースのアンロード/リロード
- 回避
  - PCTFREE

SAP Central におけるフラグメンテーションビュー



次善のファイル配置

異なるデータベースファイルの配置（システム dbspace、セカンダリ dbspace、テンポラリ dbspace、トランザクションログ）が次善である可能性がある

- 異なるディスクシステムにトランザクションログを配置するのが良いことが多い
- 好きなファイルどれでも、SSDに配置するのは良い

大量データを更新または削除するアプリケーションにおいて：

- 異なる物理ディスク上のセカンダリdbspace にユーザテーブルを押しこむのに役立てることができる → チェックポイントログにより広い帯域幅を確保
- トランザクションログは、物理ディスク自身上に必ずあるようにする
- あらゆるログのディスクに RAID-5 は避ける!

内部的に、table/index フラグメンテーションをチェックする
外部的に、データベースファイルフラグメンテーションをチェックする

アジェンダ

パフォーマンス分析
- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類
- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

CPU-バウンドアプリケーション

演算とメモリアクセスが主
→ グッドニュース – 検出が簡単!
→ バッドニュース – 可能性のある原因が複数…
検出方法?

- サーバプロセスに割り当てられた全てのCPUが98% を超えて使用
- Task Manager が割り当てられたCPUの全てをサーバープロセスで使用
- 大量のCPUに対して競合するアプリケーションがないことを確認する
- CPU の 1/N % しか使用していないシステムを監視 (並行処理なし)

次善のクエリプラン

オプティマイザが最適なプランではなく、本質的にpoorなアクセスプランを1つ以上選択している
最も共通の原因:

- オプティマイザ統計情報が古い
- OPTIMIZATION_GOAL オプションの設定が正しくない
- インデックスがない (十分に大きなバッファープール)

分析するには:

- 統計情報とともにDBISQLからマニュアルで、あるいはトレーシングセッションからプランをキャプチャする
- 遅いクエリをチェックする
- 実際の値と評価した行数やコストを比較した結果が大幅に異なるクエリオペレーションをチェックする

次善のクエリ

オプティマイザは最善の処理ができるにも関わらず、オプティマイザの仕事をしづらくするクエリがよくある
基本的な原因: 本当に必要なデータより多くのデータを計算するようサーバーがリクエストされている。
いくつか共通するもの:

- 使用しない結果セット内の余分な列を要求
- ユーザ定義関数、または、オプティマイザの手を結びつけるクエリ（サブクエリ）の頻繁な呼び出し (例えば述部において)
- 更新に使用されていないクエリの READ ONLY アクセスの特定に失敗する
- 入れ子の相関サブクエリー-- WINDOW 機能の使用を検討

最適化のゴール

最初の行 vs 全行

- アプリケーションはクエリの最初の行数行を表示
- 常にクエリの全行をフェッチするレポーティングのタイプのアプリケーション
- ある特定のアプリケーションインタフェースのパターン:
  - TOP <N> STARTAT <S>
  - API レイヤは、アプリケーションがそれらを使用しなくともクエリの全行をフェッチ

解決方法

- Optimization_goal オプション
- SQL Query での OPTION 句
- FROM 句の TABLE Hints

アジェンダ

パフォーマンス分析
- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類
- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

コンカレンシーバウンドアプリケーション

アプリケーションが、サーバーのCPU バウンドまたは I/O バウンドのどちらでもないようであれば、コンカレンシーバウンドである可能性がある

- CPU または I/O-バウンドの特別なケース – アプリケーションが余分なリソースのメリットを享受できない
- ActiveReq パフォーマンスカウンター: 高い場合には (10 から 20 またはそれ以上)、コンカレンシーバウンドであることを示している
- 低い場合でCPU と I/O も低い場合、問題は、アプリケーションがサーバーがビジーの状態をキープさせる作業を与えていない

同時接続性の問題は、サーバー内部の問題かもしれないしサーバー側アプリケーションコードの問題かもしれない。

- DBCONSOLE または sa_connection_info() を使用して決定する→ 接続が他でブロックされている場合にはアプリケーションベースの同時接続性問題、さもなくば、サーバーベースの内部の問題かもしれない

アプリケーションの同時接続性

ユーザー接続は行ロックを確保し、他の接続が作業を行えようにしている
最もよくある原因:

- ホットロー（集中アクセスされる行） – アプリケーションプロファイリングで検出 (ブロッキング分析)
- 長期読み込みロック – 長い間オープンのカーソルを探すことで検出。分離レベル >0の長いトランザクション
- 長期書き込みロック → 更新トランザクションをできる限り短く保つ

全トランザクションの分離レベルをチェックする

- 0 (デフォルト) – ロックなし; ラッチは、ディスクページからretrieveされた場合には行全体が一貫していることを確実にする
- 1,2 – クエリの結果で行をロックする。しかし level 1 では、カーソルがその行にある間だけロックする
- 3 – クエリ実行中は、全行の読み込みと全挿入ポイントをロック
- スナップショットアイソレーション – ライターは、修正された行のコピーを管理することによって、リーダーをブロックしないことを実現する。

多くのトランザクションが一つの行を更新するような設計を避ける

「1つの行テーブル」で発生
- 例えば、キー生成; シークエンスを使用することを検討する

外部キーが挿入された場合、プライマリ行をロック
- version 12 またはそれ以降では、プライマリキーのみがロックされ、他の列はオープン

内部サーバー同時接続性

サーバーは内部ロックをキープしてサーバー構造を保護
これらのリソースへのコンテンションが発生する可能性がある:

- Transaction ログ → 解決方法: 新しいディスクに移動、pregrow
- Checkpoint ログ → 解決方法: セカンダリ dbspace
- Lock テーブル – 何百もの接続をサービスしている場合にのみ可能性がある

アジェンダ

パフォーマンス分析

- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類

- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

余計な動作を避ける – クライアント問題

サーバーへのリクエスト数を減らすことによってクライアントサーバー通信をより効率的にする

- AUTOCOMMIT をOFF
- クライアント側のjoin を避ける
- アプリケーションからのワイドフェッチまたはワイド挿入を活用する
- PREFETCHing を大きな結果セットで使用する
- アプリケーションの情報をローカルにキャッシュ
- ステートメント文のセットをバッチと組み合わせる、または、ストアドプロシージャー内にステートメント文を埋め込み、1つのCALL 文だけがアプリケーションから送信する必要があるようにする。
- 初期化中に PREPARE/DESCRIBE を1度実行 (または最初の使用で)
  - クライアント文のキャッチによって、同一のSQL 文の複数の DROP/PREPARE シーケンスを避ける
- 可能な時にはいつでも列をバインドする
  - 例. SQLGetData() のかわりに SQLBindCol() を使用する

余計な動作を避ける – AutocommitをOFF

AutoCommit – 大きなペナルティー

- いくつかのインターフェースでは、デフォルトで AutoCommit が ON
  - OLEDB, ODBC, ADO.NET, JDBC, …
- 全Commit オペレーションがトランザクションログに対して少なくとも1つの物理IOを発生させる。サーバーは IOが完了するまではCommitに返答しない。
  - トランザクションログをOFFにするのは良くない。なぜならば、全てのコミットがチェックポイントを発生させている--ダーティページが全て書かれる。
- あらゆる DDL が暗黙的な COMMITs と時々 CHECKPOINTsを起こすことに注意

解決法:

- AutoCommit をOFFにし、アプリケーション内で適切に Commits と Rollbacks を使用する
- トランザクションログを常に使用する
- DDL を避ける – temp テーブルを使用する、永続的なオブジェクトを1度作成する

余計な動作を避ける – クライアント側の Join

クライアント側の join

- アプリケーションは、1つのテーブルからフェッチし、各行に対して、異なるクエリで、1つ以上の他のテーブルからフェッチする
- シンプルなバリアント: 同じ値に繰り返し発行されている同じクエリ

解決法

- サーバー側で変更されない引数部分を探す
- サーバー上で複合文として実行できないかを検討する--複雑なプロシージャーの可能性

余計な動作を避ける - Prefetch

Prefetch は、FETCH リクエストに先行して行のセットをクライアントに転送することで通信を削減する
Prefetch はデフォルトでは ON

- DisableMultiRowFetch 接続パラメータの使用を無効にするまたは Prefetch オプションを OFF に設定する
- Prefetch は、繊細な値のセマンティクスで宣言されたカーソル上では OFF

適用型 prefetching

- アプリケーションの動作によって増加または減少
- Prefetchされる最大行数は 1000
- アプリケーションが1つの 経過elapsed 秒second 内でFETCH できる行数によって影響を受ける

余計な動作を避ける – Wide Fetches/Inserts

比較的大きな結果セットには、wide fetches を使用する
- それぞれの API コールは、いくつかの行を取得; 明示的にアプリケーションで設定
  - Prefetching は、発生するかもしれないししないかもしれない
- Wide fetched 行数は設定可能

With wide (複数行) の挿入 (Ver17では更新、削除も対応):
- ESQL、ODBC、JDBC でサポート
- 適切であれば LOAD TABLE テーブルを検討する
- 通常の インターバルでCOMMITし、ロックのコンテンションを削減。ロールバックログのサイズを制限

Updates: use set based UPDATE … FROM … WHERE operations

ネットワークのレイテンシとパフォーマンス

- レイテンシ: 送信された後、異なるマシンでパケットを受け取るまでにかかる時間
- スループット: ある一定事案に転送できるビット数 (バイト)
- LAN: typically 1ms (おそらくそれよりも少ない) レイテンシ。少なくとも 1MB/秒 スループット
- WAN: 5-500 ms レイテンシー、4-200KB/秒 スループット
  - これらは、概算見積り
- DBPing: サーバーへの往復時間を決定するために使用

→ サーバーへのリクエストを減らすことでネットワークレイテンシのインパクトを削減

ネットワークレイテンシの削減

データベースサーバーのパケットサイズを増やす
- Version 11 のデフォルトは、1460 から 7300 へ増加; より大きなサイズでも、大きな結果セットの場合にメリットがある
- 大きな FETCHes と複数行の fetches または BLOB オペレーションのパフォーマンスを改善 (retrieval と insertion の両方)

CommBufferSize 接続パラメータを使用する
- より大きなパケットサイズからメリットが得られる接続のためにのみパケットサイズを変更

ReceiveBufferSize と SendBufferSize TCP/IP パラメーターを変更することを検討する
- TCP/IP プロトコルスタックで使用されるメモリ量を事前に割り当てし、over the wire でパケットを送受信
- これらの値のデフォルトはマシンに依存 (OS、ドライバー、ネットワークカード製造者)

ネットワークスループットの改善

通信の圧縮によって、モデムまたは WAN のクライアントとサーバー間のスループットが改善される可能性がある。

- パケットは暗号前に圧縮される
- 圧縮されたデータは、オリジナルのサイズの10% 以下。しかしデータとアプリケーションに完全に依存する
- より大きな圧縮のためにパケットサイズを増加。より少ないパケット
- 圧縮には追加で ~46KB/接続が必要
- アプリケーションのパフォーマンスを分析し、結果を確認する必要がある。
    - 圧縮は、追加の CPUパワーを必要とする。LANでは、圧縮コストは通常帯域幅の節約を上回る

パフォーマンスチップスのまとめ

パフォーマンスの TIPS – アプリケーションのパターン

- autocommit モードを切る
- クライアント側の join を避ける
- 適切にオプティマイザのゴールを設定する
- 適切なデータ型を使用する
- 複数行オペレーションを可能な限り使用する
- クライアントとサーバー間のリクエストを減らす
- 同時カーソル型を特定する

パフォーマンスの TIPS

- トランザクションログを常に使用する
- 同時接続問題をチェックする
- オプティマイザーのゴールを選択する
- 小さなテーブルの統計情報を収集する
- 制限を宣言する
- 参照アクションのカスケードを最小化する
- クエリパフォーマンスを監視する
- テーブル構造を正常化する
- 異なるファイルを異なるデバイスに配置する
- データベースを再構築する
- フラグメンテーションを削減する
- ファイルフラグメンテーションを削減する
- テーブルフラグメンテーションを削減する
- フラグメンテーションとskeyを削減する
- プライマリキー幅を削減する
- テーブル幅を削減する
- クライアントとサーバー間のリクエストを削減する
- expensiveなユーザー定義機能を削減する
- expensiveなトリガをリプレースする。
- テーブル内のカラムの順番をレビューする

- クエリ結果の戦略的ソートする
- 現在のカーソル型を特定する
- 明示的な選択性見積りを控えめに提供する
- autocommit モードをoffにする
- 適切なページサイズを使用する
- 適切なデータ型を使用する
- AUTOINCREMENT を使用して、プライマリキーを作成する
- バルクオペレーション方法を使用する
- delayed コミットを使用する
- in-memory モードを使用する
- インデックスとキーを効果的に使用する
- キャッシュを使用してパフォーマンスを改善する
- キャッシュサイズを監視する
- キャッシュワーミングを使用する
- 注意して圧縮を使用する
- バリデーションには WITH EXPRESS CHECK を使用する
- クエリプロセスに作業テーブルを使用する（全行最適化目標）

アジェンダ

パフォーマンス分析
- 分析例
- グラフィカルなプランビューワー
- パフォーマンスタイミングユーティリティー

パフォーマンス問題の種類
- 通信パターン
- I/O バウンドアプリケーション
- CPU-バウンドアプリケーション
- 同時接続性バウンドアプリケーション

パフォーマンスチップス

まとめとリソース

まとめ

SQLA は、新しいハードウェア/使用シナリオに対しての技術革新を継続して行っていきます。

メモリやコア数増加のトレンドや、新しいストレージテクノロジーが多いに関係してきます。より大規模のデータの扱いが重要となってきています。

しかし: シンプルなオペレーションにおけるスピードもまた特定のアプリケーションパターンでは重要です。

パフォーマンス分析とチューニングは、興味深く特異な領域です。

実際何が起こっているのかトレースするのは楽しいことでもあります。

不満を持っているお客様で問題が発生する前に実行できればベストです。

リソース

ホワイトペーパー:

SQL Anywhere における容量計画

http://scn.sap.com/blogs/sqlanywhere_japan/2015/03/11/capacity-planning-for-sql-anywhere

SQL Anywhere によるアプリケーション・パフォーマンス問題の診断

http://scn.sap.com/blogs/sqlanywhere_japan/2015/03/11/diagnosing-application-performance-issues-with-sql-anywhere

SQL Anywhere マニュアル

http://dcx.sap.com

SQL Anywhere Q&A フォーラム（英語）

http://sqlanywhere-forum.sap.com

SAP 日本語コミュニティ

http://scn.sap.com/community/japanese

Thank you

Jason Hinsperger

Sr Product Manager

SAP Labs Canada, 445 Wes Graham Way, Waterloo, ON N2L 6R, Canada

Jason.hinsperger@sap.com

519-883-6492

© 2014 SAP AG or an SAP affiliate company. All rights reserved.