ASA9.0.2: 'LogExpensiveQueries' 機能の使い方

要求レベルのロギングとグラフィカルなプランは、パフォーマンス・チューニングを行う場合に非常に役立つ SQL Anywhere の機能です。要求レベルのロギングでは、実行に長い時間がかかっている文の SQL コードを取得し、グラフィカルなプランでは、それだけの時間がかかっている理由を確認できます。これらの機能は非常に便利ですが、少なくともバージョン 9.0.2 の SQL Anywhere に対する Express Bug Fix Build 3124 までは、パフォーマンス分析には依然として多くの作業が必要でした。Express Bug Fix Build 3124 では、LogExpensiveQueries 機能が導入されています。
要求レベルのロギングの出力ファイルでは CONNECT 操作と SELECT 操作の間隔が数時間開くことがあるため、LogExpensiveQueries 機能を使用しなければ、クエリがどこから来ているのかを判断するのは困難な場合があります。
第二に、ファイルが巨大化する可能性があります。すべての CONNECT を取得したり、1 日の勤務時間内に何が起こっているのかを単に確認したりするためにプロセスを何時間も実行した場合は、稼働率の高いサーバでは数ギガバイトのサイズになります。
第三に、SQL Anywhere に付属している sa_get_request_times プロシージャを利用しても、大きなログ・ファイルの分析には長い時間がかかる可能性があります。
また、それはデータ収集にすぎません。要求レベルのロギングを終了すると、低速な文の SQL コードが得られます。ただし、それだけの時間がかかっている理由を調べるには、引き続き各クエリを dbisql で実行してグラフィカル・プランを確認する必要があります。稼働率の高いサーバで低速クエリの「上位 100」リストに直面した場合、これは大変な作業になる可能性があります。
さらに、同じデータ、同じデータベース、場合によっては同じ全体作業負荷を慎重に使用しなければ、dbisql で表示されるプランは運用で使用される実際のプランとはまったく異なる可能性があります。ストアド・プロシージャ内で動作してテンポラリ・テーブルを使用するクエリの場合、グラフィカルなプランを確認するのは困難です。プロシージャ内に GRAPHICAL_PLAN() の呼び出しを埋め込み、後で調べるために結果をファイルに書き込む必要があります。また、誤ってクエリを 2 回連続で実行し、2 回目の実行に対するプランを参照しないようにする必要があります。データベース・キャッシュにはクエリを満たすのに必要なローがすべて含まれているため、このプランは運用で使用されるプランとは完全に異なる可能性があります。

> **ヒント：** dbisql を使用してクエリを実行し、統計情報付きのグラフィカルなプランを取得する場合は、[SQL] - [実行] ([F5] または [F9]) の代わりに [SQL] - [プランの取得] ([Shift+F5]) を使用します。[Execute 実行] を使用すると、クエリは 2 回実行されます。1 回目の実行では結果セットが決定され、2 回目の実行ではプランが取得されます。このプランは、クエリの初回実行時に使用したプランとは完全に異なる可能性があります。また、これはキャッシュにローが含まれていることだけが原因ではありません。たとえば、プランが DELETE を対象としている場合は、最初の実行でローが実際に削除されるため、プランは完全に異なる「ゼロのロー」ケースを対象とする可能性があります。[プランの取得] ([Shift+F5]) を使用した場合は、クエリが 1 回しか実行されないため、プランは一致します。

朗報ですが、新しい LogExpensiveQueries 機能は上記の問題をすべて解決します。ユーザは、運用で SQL 文の実行に使用される実際の実行プランを今回初めて取得できます。dbisql で実行するテスト・シ

ナリオを手動でセットアップする必要はなく、テスト・シナリオが現実にマッチしていることを前提とする必要もなくなっています。
また、取得したプランには、クエリ実行時のサーバおよび接続の状態に関するあらゆる種類の情報が含まれます (たとえば、ピーク・キャッシュ・サイズやユーザ ID など)。
LogExpensiveQueries では、単にプロセスを開始してそのまま待機し、プランが取得されたら、そのプランを確認します。これらは実際のプランです。キャッシュがコールドだった場合は、そのことがプランで示されます。誤ってクエリを 2 回実行して結果をゆがめることはありません。運用における統計が良好または不良の場合は、プランにそれが反映されます。複雑なストアド・プロシージャ内の深いところにクエリがあっても、問題ありません。クエリはテンポラリ・テーブルを使用するため、クエリをコピーして dbisql に貼り付けたり、プロシージャを変更して UNLOAD SELECT GRAPHICAL PLAN() を実行したりする必要はありません。
ここでは、Build 3182 で提供された説明を掲載し、その後にヒントと手法をいくつか示します。

9.0.2.3182 "readme" ファイルからの抜粋：

今回サーバは、コストの高いクエリの SQL テキストまたはグラフィカルなプランを要求レベルのログにダンプできます。この機能をオンにし、クエリが高コストとみなされるしきい値を定義するために、新しいサーバ・コマンド・ライン・オプションの -zx <cost> が追加されています。コストの高いクエリは、2 つの方法で処理できます。-zp オプションを併せて指定した場合は、詳細なグラフィカル・プランが要求レベルのログへ送られます。それ以外の場合は、コストの高いクエリの SQL テキストだけがログに記録されます。

- 推定コストが <cost> ミリ秒よりも大きいクエリは完全な統計収集ツールを使ってビルドされ、グラフィカルなプランはビルド時に要求レベルのログにダンプされます。
- 推定コストが <cost> ミリ秒よりも小さいクエリは、詳細度の低い統計収集ツールを使ってビルドされます。
- 実際の実行時間が <cost> ミリ秒よりも長いクエリでは、カーソルを閉じるとグラフィカルなプランが要求レベルのログにダンプされます。最初のポイントで説明したように、プランには最適時に決定されたレベルの統計詳細が含まれます。

-zx だけを設定すると、次のようになります。

- クエリは、詳細度の低い統計収集ツールを使ってビルドされます。
- 実際の実行時間が <cost> ミリ秒よりも長いクエリでは、カーソルを閉じると SQL テキストが要求レベルのログにダンプされます。

これらの行を表示するには、-zo コマンド・ライン・オプションを使用するか、RequestLogFile プロパティを設定して、要求レベルのログの宛先をファイルにする必要があります。ただし、RequestLogging プロパティが 'None' であっても、これらの行は引き続き表示されます。
要求レベルのログでは、プランは単一の PLAN 行として表示され、SQL テキストは単一の INFO 行として表示されます。これらのプランや SQL テキストの前には、ビルド時またはカーソル完了時にダンプされたかどうか、およびその時点の関連コストは何であったかを示すヘッダが付けられます。

ビルド時にダンプされたプランの前には [XB <cost>]、カーソル完了時にダンプされたプランの前には [XC <cost>]、完了時にダンプされた SQL テキストの前には [XS <cost>] が付けられます (グラフィカルなプランでの表示と同様に、コストはすべて秒単位で示される)。
-zx <cost> コマンド・ライン・オプションを使用するか、sa_server_option() を使って LogExpensiveQueries サーバ・オプションを設定すると、LogExpensiveQueries プロパティを設定できます。

LogExpensiveQueries のオン／オフを切り替える最良の方法は、dbisql から切り替えを行うことです。LogExpensiveQueries をオンにする方法を次に示します。

```
-- On
CALL sa_server_option ( 'RequestLogging', 'NONE' );
CALL sa_server_option ( 'RequestLogFile', 'c:/temp/expensive.txt' ); -- append to file
CALL sa_server_option ( 'RememberLastPlan', 'ON' );
CALL sa_server_option ( 'LogExpensiveQueries', '1000' ); -- cost threshold in milliseconds
```

sa_server_option プロシージャは 2 つのパラメータを取ります。一つはサーバ・オプションの名前、もう一つは対応するオプション値であり、どちらも VARCHAR (128) です。
上記の 4 つの呼び出しがどのように機能するのかを次に示します。

- 古い冗長な「要求レベルのロギング」機能を使用しないことを明確にするため、RequestLogging を 'NONE' に設定します。
- RequestLogFile を 'c:/temp/expensive.txt' に設定して、出力テキスト・ファイルを指定します。このファイル指定は、データベース・エンジンが実行されているコンピュータを基準としています。場合によっては、dbisql を実行しているコンピュータとは異なることもあります。この例の場合、データベース・エンジンと dbisql は、どちらも同じマシン上にあります。expensive.txt ファイルが存在しない場合は、自動的に作成されます。ファイルが存在する場合は、新しいデータが最後に追加されます。RequestLogFile オプションは、要求レベルのロギングの出力ファイルを指定する場合と同じオプションですが、LogExpensiveQueries によって異なるデータが出力ファイルに書き込まれます。
- RememberLastPlan を 'ON' に設定して、クエリの SQL だけでなくグラフィカルなプラン全体が必要であることをサーバに伝えます。
- LogExpensiveQueries を '1000' に設定して、プラン取得の「コスト・スレッショルド」をミリ秒単位で指定します。つまり、'1000' は 1 秒、'60000' は 1 分を表します。このコスト・スレッショルドは、クエリごとに 2 回チェックされます (実行前と実行後) 。また、結果に応じて、1 つまたは 2 つのプランがテキスト・ファイルに書き込まれます。稼働率の高い運用サーバの場合は、慎重にコスト・スレッショルドを選択する必要があります。値が小さすぎると大量の出力が生成されるので、大きい値から始めて、ファイルの増加量を確認しながら値を小さくしてください。

LogExpensiveQueries をオフにする方法を次に示します。

```
-- Off
CALL sa_server_option ( 'LogExpensiveQueries', '0' ); -- "off"
```

```
CALL sa_server_option ( 'RememberLastPlan', 'OFF' );
CALL sa_server_option ( 'RequestLogFile', '' ); -- releases file
```

- LogExpensiveQueries を '0' に設定して、プラン取得プロセスを停止します。値 '0' は、「ゼロのコスト・スレッショルド」ではなく「オフ」を意味します。
- RememberLastPlan を 'OFF' に設定して、プラン処理を無効にします。これは必要のない作業かもしれませんが、DBA は慎重を期するに越したことはありません。
- RequestLogFile を '' に設定して、出力ファイルを解放します。これにより、ワードパッドなどのプログラムで出力ファイルを表示できるようになります。

LogExpensiveQueries をオンにし、推定実行時間が指定のしきい値を超えているクエリを開始すると、グラフィカルなプランがすぐに出力ファイルに書き込まれます。このプランには、「実行前」を表す "XB" というマークが付けられます。実行時間が非常に長いクエリがあり、何が起きているのかをそのクエリの終了を待たずに確認したい場合は、このプランが役立ちます。
メモ帳を使用するか、ファイル・コピーを作成してワードパッドを使用すれば、LogExpensiveQueries をオフにすることなく出力ファイルを表示できます。Windows XP のワードパッドで開いた "Copy of expensive.txt" ファイルを図 1 に示します。

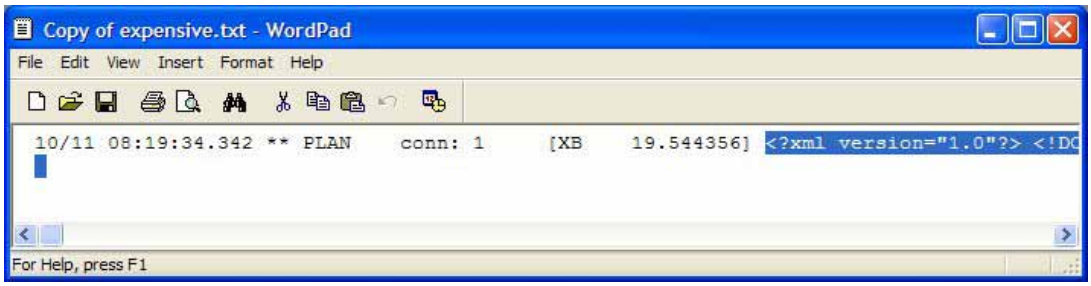


図 1： XB プランが選択されている Copy of expensive.txt ファイル

LogExpensiveQueries ファイルの各行は、次のようになります。

10/11 08:19:34.342	日付と時刻
** PLAN	レコード・タイプ
Conn: 1	接続番号
[XB	プラン・タイプ XB または XC：「実行前」または「完了済み」
19.544356]	XB プランの推定実行時間 (秒単位)、XC の実際の実行時間
<?xml version="1.0"?> ...	グラフィカルなプランの XML テキスト

LogExpensiveQueries によって取得されたグラフィカルなプランを表示するには、XML テキストを抽出して *.XML ファイルに保存してから、そのファイルを dbisql で開きます。最も簡単なテキスト抽出方法は、コピーして貼り付けることです。このプロセスは図 1 と図 2 に示してあります。図 1 では、

XB プランの XML テキストを選択しており、図 2 では、ワードパッドを使用して "XB_plan.xml" ファイルにそのテキストを貼り付けています。

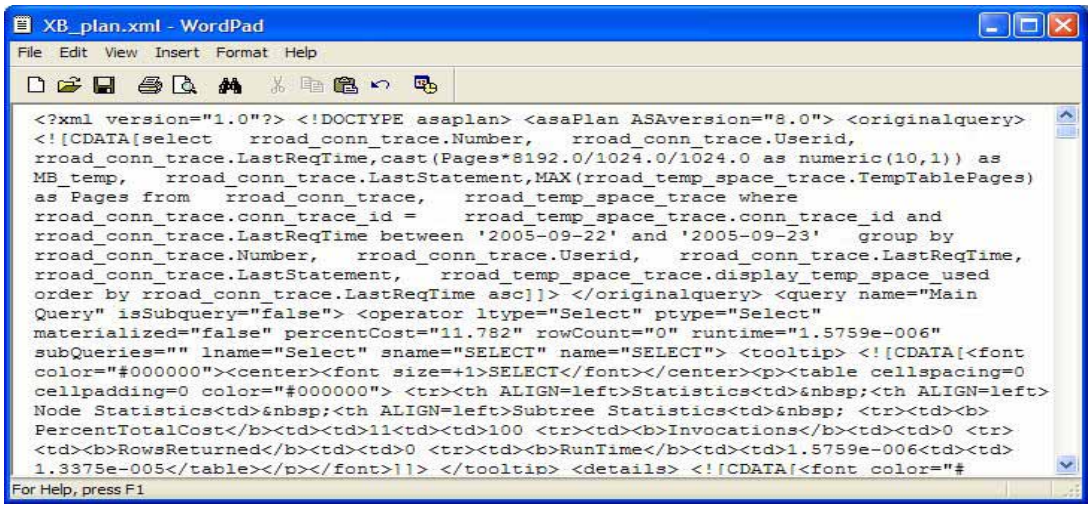


図 2： ワードパッドに貼り付けられた XB_plan.xml

ヒント： ワードパッドの [表示] - [オプション] で [右端での折り返し] を [なし] に設定すると、図 1 のように各プランが単一行に表示されます。[ウィンドウに合わせる] に設定すると、すべてのテキストが図 2 のように表示されます。

ヒント： dbisql では、[File] - [Open] - [Files of type: XML (*.xml)] を使用してプランを開くことができます。また、dbisql -f filename.xml オプションを使用してコマンド・ラインから開くこともできます。[Connect] ダイアログ・ボックスが表示された場合は、[Cancel] をクリックしてください。グラフィカルなプランのファイルを表示するのにデータベース接続は必要ありません。
dbisql で開いた XB プランを図 3 に示します。左側のウィンドウ枠では、トップレベルの "SELECT" ノードが選択されています。右側のウィンドウ枠では、"Subtree Statistics" セクションの "RunTime" 行が強調表示されています。これは、このノードとその下にある他のノードの合計実行時間 (秒単位) です。

ヒント： XB プランの "Actual" 統計は注目せずに、"Estimates" だけに注目してください。この種のプランはクエリの実行前に生成されるため、本当の実績値は存在しません。

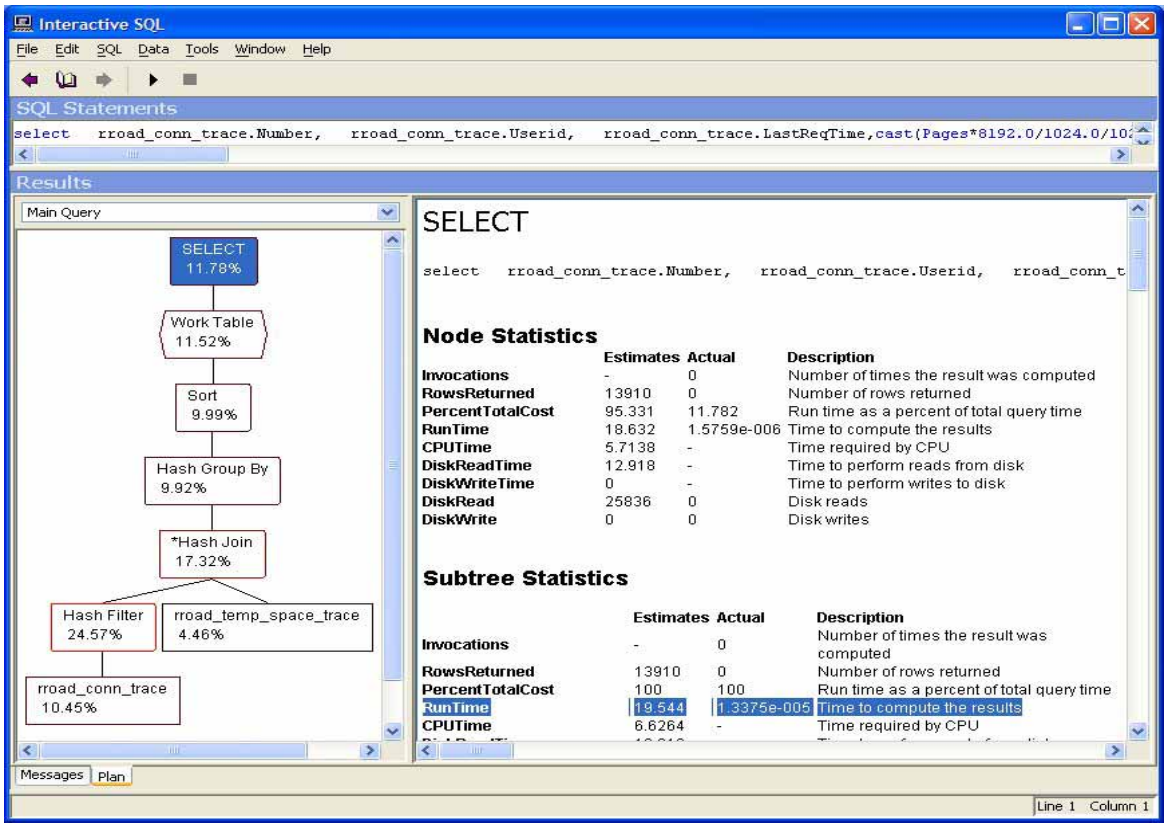


図 3： RunTime が強調表示されている XB プラン

お使いのコンピュータで図 3 のようにダイアグラムが表示されない場合は、dbisql の左側のウィンドウ枠で右クリックして [Customize] を選択し、図 4 のように [Show short names] (オフ) と [Show percent cost] (オン) の設定を変更してください。

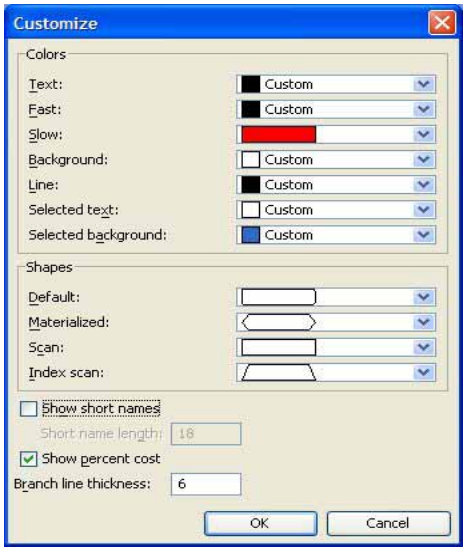


図 4： グラフィカルなプランのダイアグラムのカスタマイズ

図 3 のプランの生成時に実行されたクエリを次に示します。

```
SELECT rroad_conn_trace.Number,
    rroad_conn_trace.Userid,
    rroad_conn_trace.LastReqTime,
    CAST ( Pages * 8192.0 / 1024.0 / 1024.0
        AS NUMERIC ( 10, 1 ) ) AS MB_temp,
    rroad_conn_trace.LastStatement,
    MAX ( rroad_temp_space_trace.TempTablePages ) AS Pages
FROM rroad_conn_trace
    INNER JOIN rroad_temp_space_trace
    ON rroad_conn_trace.conn_trace_id
      = rroad_temp_space_trace.conn_trace_id
WHERE rroad_conn_trace.LastReqTime
  BETWEEN '2005-09-22' AND '2005-09-23'
GROUP BY rroad_conn_trace.Number,
    rroad_conn_trace.Userid,
    rroad_conn_trace.LastReqTime,
    rroad_conn_trace.LastStatement,
    rroad_temp_space_trace.display_temp_space_used
ORDER BY rroad_conn_trace.LastReqTime
```

クエリの実行が終了すると、先に示したように LogExpensiveQueries プロセスがオフになり、出力ファイルをワードパッドで自由に開けるようになりました。図 5 は、XC 「完了済み」プランが expensive.txt ファイルに現れていることを示しています。実際の実行時間である 19.304747 秒は XB 推定値の 19.544356 秒に非常に近い値になっています。

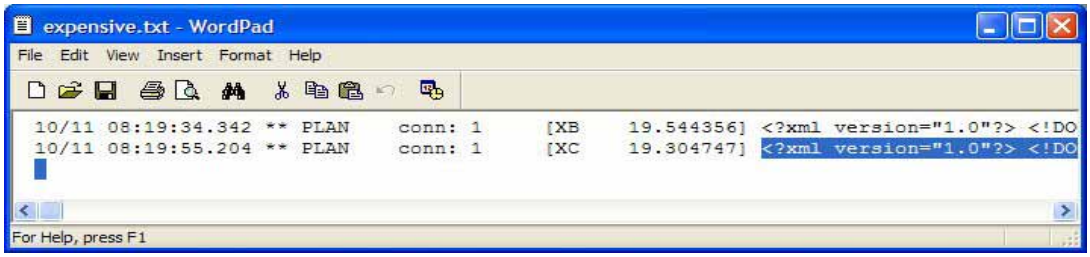
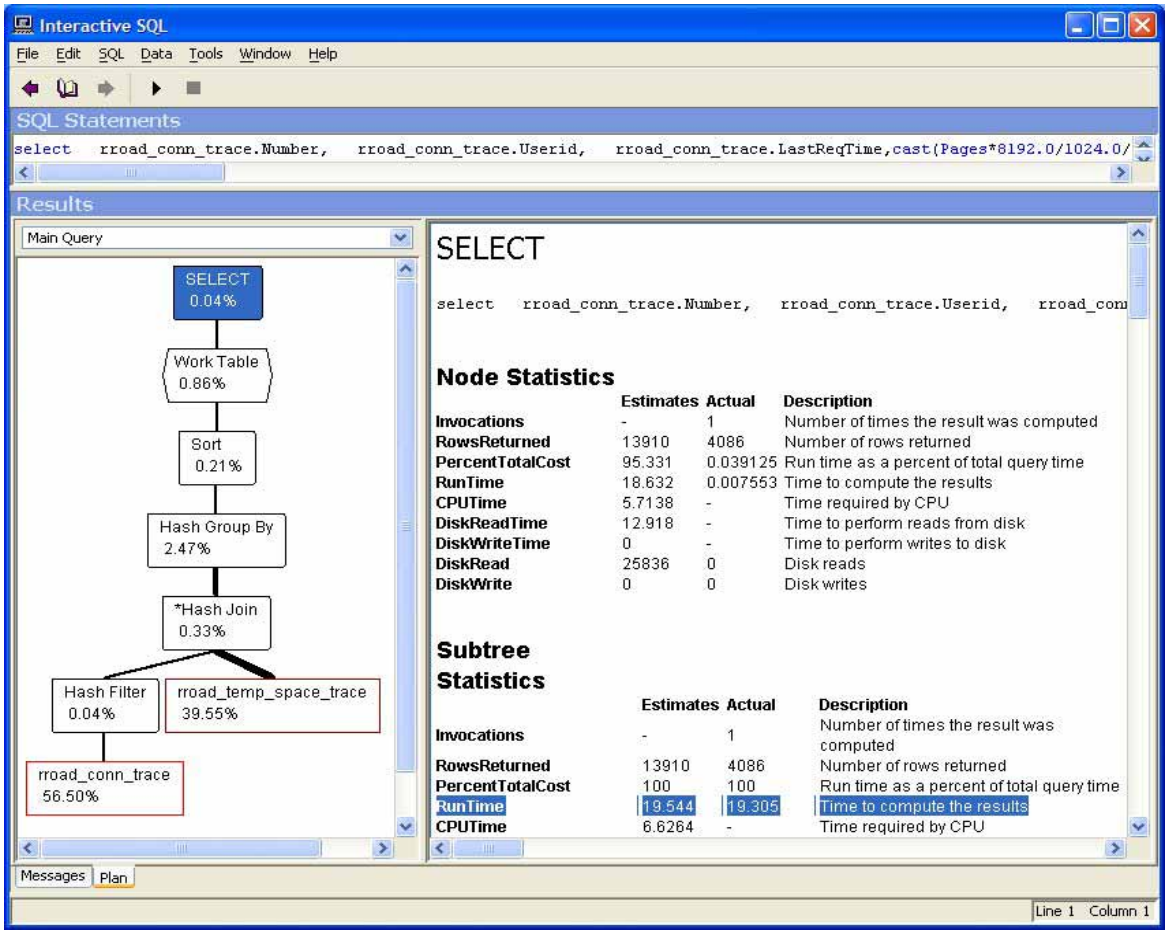


図 5：XC プランが選択されている expensive.txt

図 6 は、RunTime 行全体が強調表示されている dbisql 内の XC プランを示しています。このような XC プランの場合、推定値と実績値はどちらも意味があります。また、左側のウィンドウ枠のダイアグラムには実際のパーセンテージが表示されます。
ダイアグラムの左下にあるノードは、最も大きな問題、つまり rroad_conn_trace テーブルが実行時間の 56.5% を消費していることを示しています。

図 6： RunTime が強調表示されている XC プラン



ヒント： XB プランは、実行時間の長いクエリのプランをクエリの実行終了前に確認する場合は便利ですが、それ以外の場合はあまり役に立ちません。一方、XC プランは、詳細な統計情報が含まれている場合に非常に役立ちます。これは、推定実行時間と実績実行時間の両方がしきい値を超えている場合にのみ起こります。つまり、XC プランで詳細な統計情報が必要な場合は、XB プランを取得する必要があります。推定値は実績値よりも小さいことがあるため、XC プランに統計情報を含める場合は、しきい値を低く設定し、XB プランをすべて無視してください。

ヒント： LogExpensiveQueries しきい値を非常に小さい値 (たとえば、1/100 秒を表す '10' など) に設定すると、不可解なプランが出力ファイルに急に現れることがあるので注意してください。これらのプランは、クエリの取得元 (他のアプリケーション、リモート・サーバ接続、dbisql など) がどこであっても生成される可能性があります。そのため、コピーして .XML ファイルに貼り付ける XC プランを選択するときには注意してください。

図 7 は、テーブル・スキャンを使用して rroad_conn_trace テーブルを処理していることを示しています。SQL Anywhere バージョン 9 では、テーブル・スキャンは必ずしも悪いことではありません。この例では、RowsReturned の Estimates 値は 228,300 であり、偶然にもテーブル内のローの数になっています。それが正しければ、おそらくテーブル・スキャンは良い方法です。

しかし、RowsReturned の Actual 値は 3,845 にとどまっているため、おそらく、もっと良い方法があると思われます。

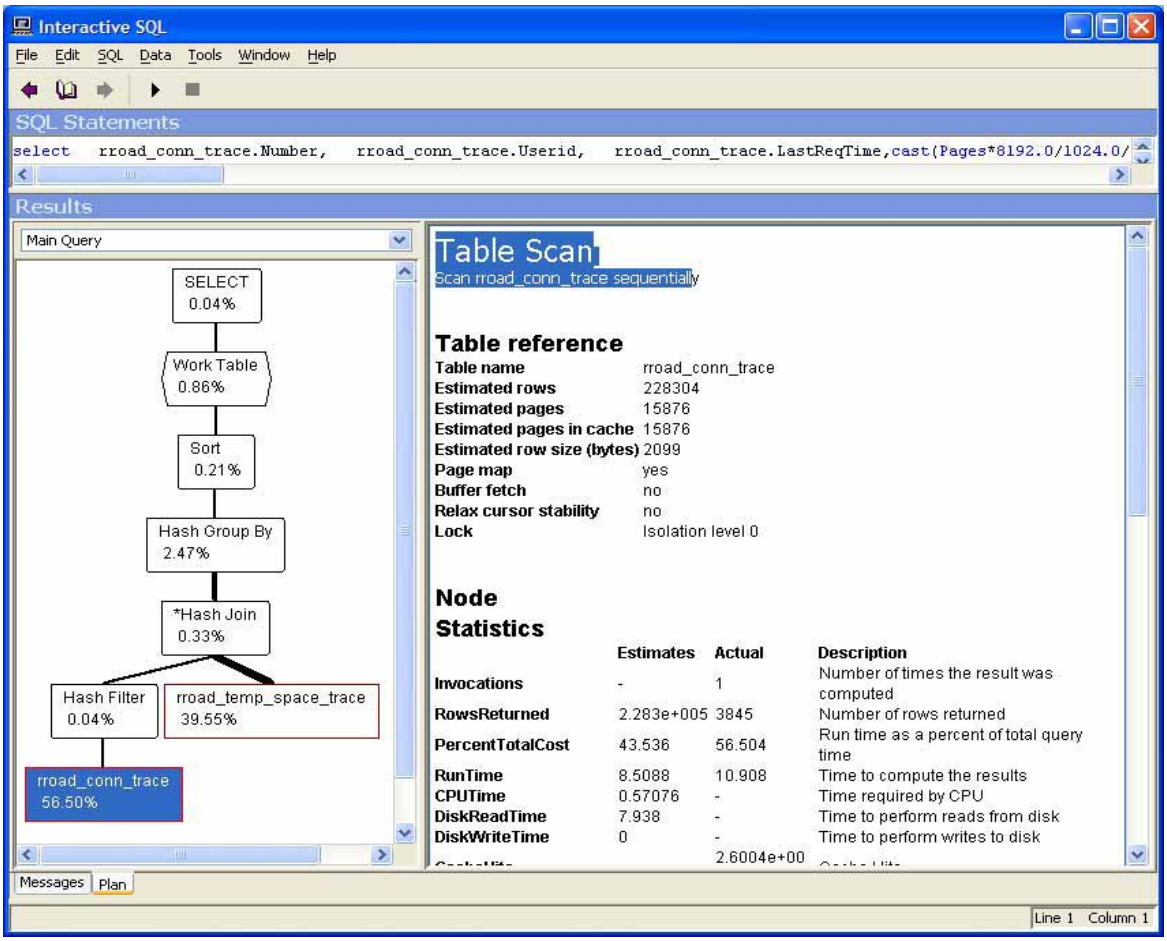


図 7： テーブル・スキャンが強調表示されている XC プラン

図 8 は同じ rroad_conn_trace ノードを示していますが、右側のウィンドウ枠が一番下までスクロールされ、"Residual predicate" が強調表示されています。この場合、"Residual predicate" は「残り」 (高速化のためにクエリ・オプティマイザが使用しなかった WHERE 句の述部) を意味します。

```
Residual predicate
( '2005-09-22' <= rroad_conn_trace.LastReqTime <= '2005-09-23' ) : 1.6842% Statistics
```

この述部の選択性は 1.6842% として表示されており、この数値は SQL Anywhere の最適化ヒストグラム・テーブル SYSCOLSTAT に格納された "Statistics" に基づいています。統計としては、これは非常に良い数値であり、RowsReturned 率の (3,845 / 228,300) * 100 にほぼ等しくなっています。

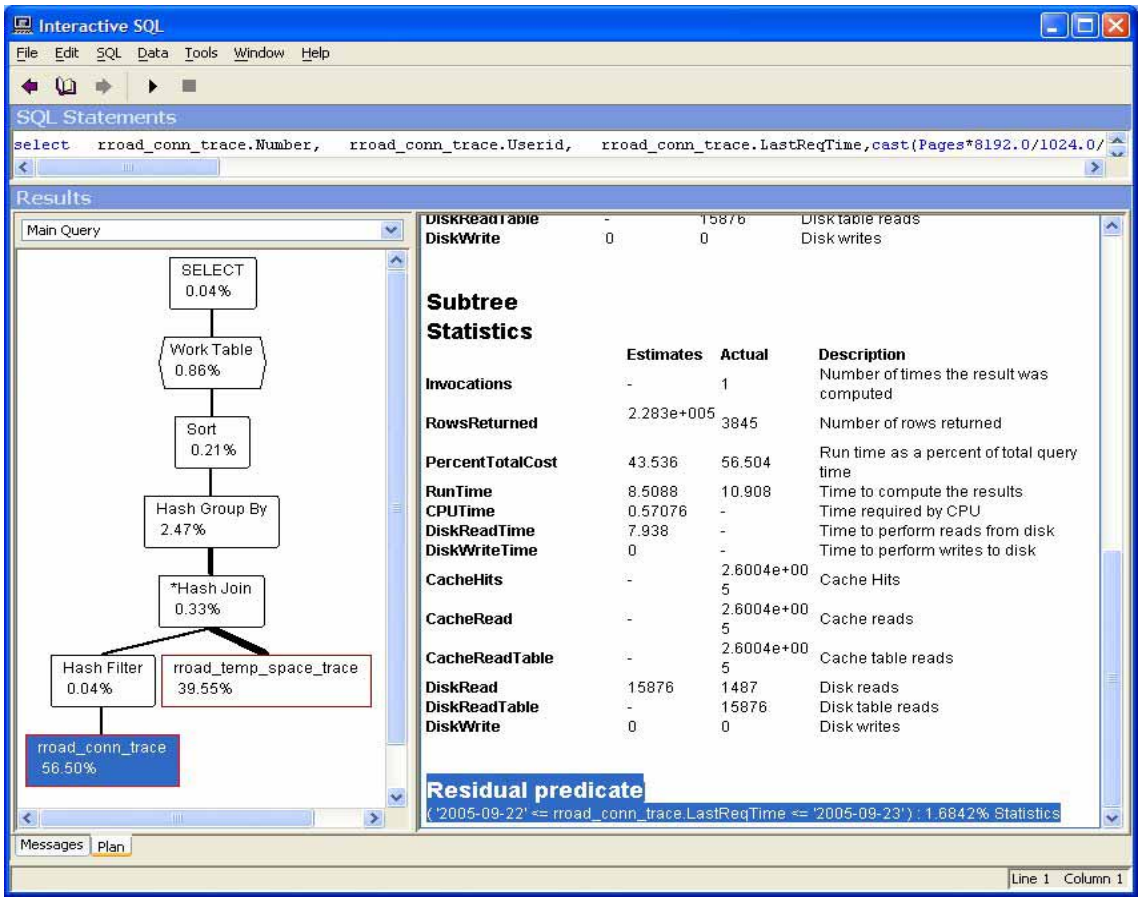


図 8： Residual Predicate が強調表示されている XC プラン

1.6842% の選択性は、rroad_conn_trace.LastReqTime に通常のインデックスを作成する十分な理由にならないかもしれませんが、クラスタード・インデックスはまた別の話です。残りの述部は、元の SELECT で BETWEEN を使用しているレンジ・クエリであり、クラスタード・インデックスの完璧な候補です。

```
CREATE CLUSTERED INDEX xLastReqTime ON rroad_conn_trace ( LastReqTime );
```

rroad_conn_trace のデータは、初めから LastReqTime 順に挿入されているため、REORGANIZE TABLE を実行してローをソートする必要はありません。これらのデータは、すでに効果的にソートされています。図 9 に結果を示します。XC プランは、クラスタード・インデックスを使用して rroad_conn_trace を処理したこと、および、このノードの時間が全体の 4.14% まで低下したことを示しています。
全実行時間 (未表示) は、以前の 19.3 秒から 9.1 秒に減少しています。つまり、速度が 100% 以上向上したことになります。

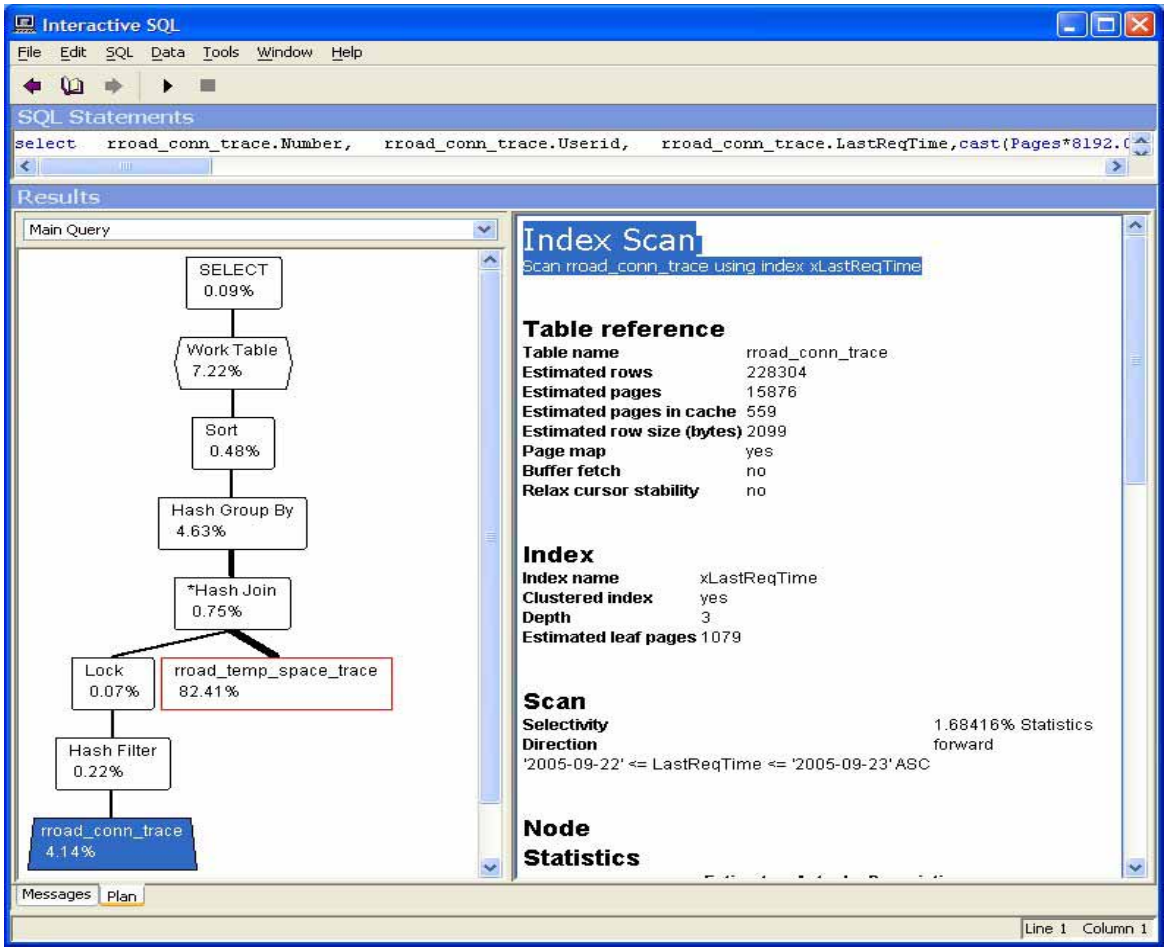


図 9： クラスタード・インデックスが強調表示されている高速の XC プラン

4,000 のローだけを返すクエリとしては、9 秒は依然として長い時間です。図 9 のダイアグラムは、テーブル rroad_temp_space_trace のノードが 82% を超える時間を消費していることを示しており、図 10 は新たなトラブルメーカの詳細を示しています。

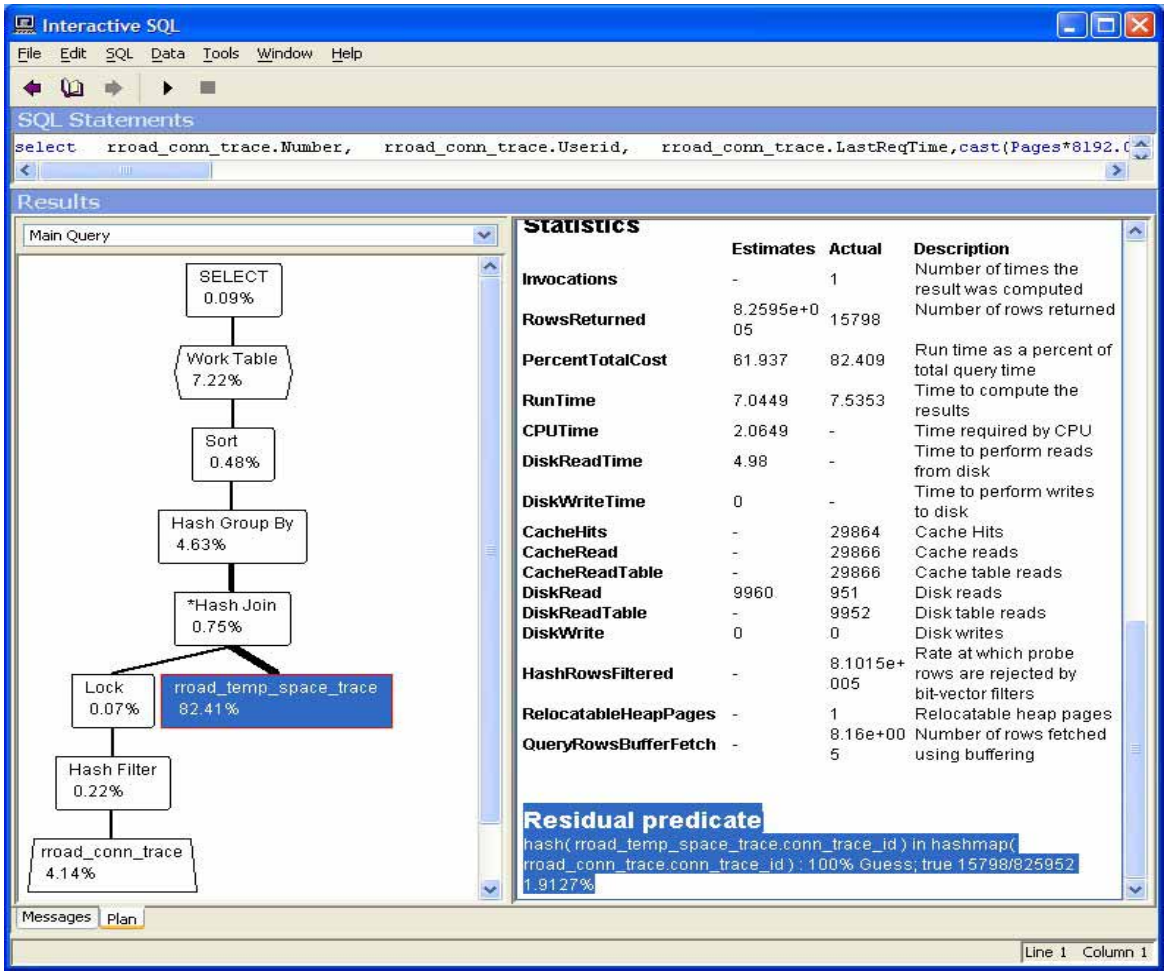


図 10： Residual Predicate が強調表示されている高速の XC プラン

ここでもまたテーブル・スキャンを使用しており、図 10 **も、対象となるテーブル内の単一カラム (今回は** rroad_temp_space_trace **内の** conn_trace_id) **を含んだ** "Residual predicate" **を示しています。**

```
Residual predicate
hash( rroad_temp_space_trace.conn_trace_id ) in hashmap(
rroad_conn_trace.conn_trace_id ) : 100% Guess; true 15798/825952 1.9127%
```

このケースでは、1 **つではなく** 2 **つのインデックスが** rroad_temp_space_trace.conn_trace_id **にすでに存在します。一つは** conn_trace_id **を最初のカラムとして使用したプライマリ・キー・インデックス、もう一つは** rroad_conn_trace **と** rroad_temp_space_trace **の間の外部キー関係です。その両方を表すスキーマを次に示します。**

-- DBA.rroad_temp_space_trace (table_id 452) in sniffer - Oct 13 2005 2:07:52PM - Foxhound © 2005 RisingRoad

```
CREATE TABLE DBA.rroad_temp_space_trace ( -- 825,952 rows, 70.3M total = 38.9M table + 0K ext + 31.4M index
```

```
   conn_trace_id                    /* PK FK     */ BIGINT NOT NULL,
   sample_date_time                 /* PK FK     */ TIMESTAMP NOT NULL,
   TempTablePages                                  BIGINT NOT NULL,
   Temporary_File_free_space                       UNSIGNED BIGINT NOT NULL,
   display_temp_space_used                         VARCHAR ( 200 ) NOT NULL,
   display_free_space_on_temp_file_drive           VARCHAR ( 200 ) NOT NULL,
CONSTRAINT ASA88 PRIMARY KEY ( -- 11M
      conn_trace_id,
      sample_date_time ) );


ALTER TABLE DBA.rroad_temp_space_trace ADD CONSTRAINT rroad_conn_trace NOT NULL
FOREIGN KEY ( -- 8.9M
      conn_trace_id )=
   REFERENCES DBA.rroad_conn_trace (
      conn_trace_id )
   ON UPDATE RESTRICT ON DELETE RESTRICT;


ALTER TABLE DBA.rroad_temp_space_trace ADD CONSTRAINT rroad_trace_summary NOT
NULL FOREIGN KEY ( -- 11.6M
      sample_date_time )
   REFERENCES DBA.rroad_trace_summary (
      sample_date_time )
   ON UPDATE RESTRICT ON DELETE RESTRICT;
```

クエリ・オプティマイザがどちらのインデックスも選択しなかった原因は不明です。おそらく、プライマリ・キー・インデックスに 2 **つのカラムが含まれていて、残りの述部が** 1 **つのカラムだけを指定したため、プライマリ・キー・インデックスが大きすぎたと考えられます。また、**100% **の選択性** "Guess" **により、テーブル・スキャンが大幅に改善されたため、オプティマイザは** conn_trace_id **の外部キー・インデックスを選択しなかったと考えられます。より低い値** (Residual predicate **表示に示されている** 1.9127%) **であっても、外部キー・インデックスを選択するには高すぎた可能性があります。**
ただし、この特定のテーブルの場合、ローは conn_trace_id **および** sample_date_time **順に挿入されるので、両方のインデックスが効果的にクラスタ化されます。**ALTER INDEX **コマンドを使用すると、**1 **つのインデックスについてその事実を記録できます。また、この述部のニーズに正確に適合しているため、単一カラムの外部キー・インデックスが選択されています。**

```
ALTER INDEX FOREIGN KEY rroad_conn_trace ON rroad_temp_space_trace CLUSTERED;
```

ヒント： プライマリ・キーと外部キーでは、ALTER INDEX ... CLUSTERED **および** NONCLUSTERED **コマンドを使用できます。これらの制約を削除して再作成する必要はありません。**

ヒント： ALTER INDEX ... CLUSTERED **は、ほとんど作業を行わないため、非常に速く実行されます。インデックスを作成することもなく、ソートによってローを再編成することもありません。** ALTER INDEX ... CLUSTERED **は、単にインデックスをクラスタードとしてマークしています。**
ヒント：データベース内のデータに関して何か特別なことを知っている場合は、SQL Anywhere **にその内容を伝えれば、パフォーマンスの向上に役立つことがあります。インデックスが一意の場合は、その内容を伝えてください。このケースでは、インデックスが効果的にクラスタ化されるため、**SQL Anywhere **にその内容を伝えています。**

図 11 **に結果を示します。**2 **番目のクラスタード・インデックスにより、実行時間は** 1.15 **秒に減少しています。これは、**1 **つのクラスタード・インデックスだけを使用した場合と比べると、ほぼ** 700% **の高速化であり、元のクエリと比べると** 1,500% **以上の高速化になります。**

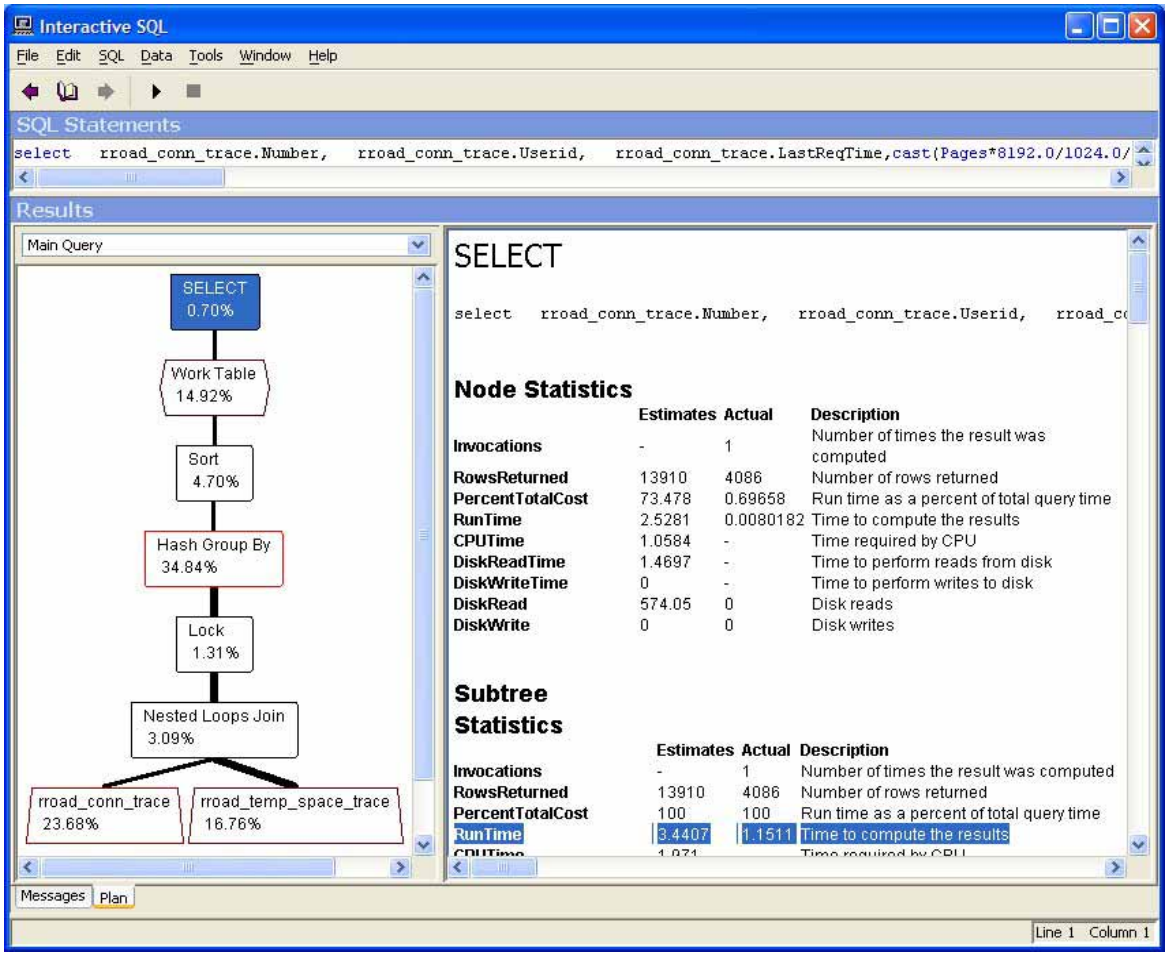


図 11： RunTime が強調表示されている高速の XC プラン

結論

一つの明白な結論として、クラスタード・インデックスは優れているということがいえます。
もう一つの結論は、LogExpensiveQueries 機能はグラフィカルなプランの取得を簡略化することでグラフィカルなプランの処理を簡略化するということです。簡単になるだけでなく、精度も向上するため、運用で実際に起きていることを反映するプランの取得方法の模索ではなく、クエリの最適化に時間を費やすことができます。
このことはすでに認知されています。

Breck Carter は、RisingRoad Professional Services の主席コンサルタントであり、SQL Anywhere データベースと、Oracle、DB2、SQL Server、ASE、および ASA との Mobile Link 同期についてコンサルティングとサポートを提供しています。Breck は、Wordware Publishing の『SQL Anywhere Studio 9 Developer's Guide』(ISBN 1-55622-506-7) の著者でもあります。
Breck の連絡先は breck.carter@risingroad.com です。
