AppSleuth: a Tool for Database Tuning at the Application Level

Wei CAO*

School of Information, Renmin University of China

59 Zhongguancun Avenue, Haidian, Beijing, 100872, P.R.C.

caowei@ruc.edu.cn

Dennis SHASHA†

Department of Computer Science, New York University

251 Mercer Street, New York, NY 10012, U.S.A.

shasha@cs.nyu.edu

ABSTRACT

Excellent work ([1]-[6]) has shown that memory management and transaction concurrency levels can often be tuned automatically by the database management systems. Other excellent work ([7]]-[14]) has shown how to use the optimizer to do automatic physical design or to make the optimizer itself more self-adaptive ([15]-[17]). Our performance tuning experience across various industries (finance, gaming, data warehouses, and travel) has shown that enormous additional tuning benefits (sometimes amounting to orders of magnitude) can come from reengineering application code and table design. The question is: can a tool help in this effort? We believe so. We present a tool called AppSleuth that parses application code and the tracing log for two popular database management systems in order to lead a competent tuner to the hot spots in an application. This paper discusses (i) representative application "delinquent design patterns", (ii) an application code parser to find them, (iii) a log parser to identify the patterns that are critical, and (iv) a display to give a global view of the issue. We present an extended sanitized case study from a real travel application to show the results of the tool at different stages of a tuning engagement, yielding a 300 fold improvement. This is the first tool of its kind that we know of.

Keywords

Database tuning, application-level optimization, performance tool

1. INTRODUCTION

Database administrators can call on a variety of tools to help with physical configuration ([7]-[14]), system monitoring and maintenance ([20]-[23]).

Current automatic physical tuning tools have become sophisticated. Given a representative workload of SQL statements; they find the best physical design for the workload. They do this based on tight interaction (what-if analysis [11] or instrumentation [10]) with the cost-based query optimizer. Beyond that effort in automatic physical design, Oracle's SQL Tuning advisor [18] can collect statistics, correct system parameters, and recommend changes to SQL statements. (In the running example of this paper, the Tuning Advisor found high load SQL statements and identified bad query features like Cartesian products.) Such tools work at the SQL statement level, aiming to find beneficial physical structures to the SQL workloads or to spot the problematic SQL statements. AppSleuth, as a database tuning tool at the application level, can incorporate and work with such tools to offer better performance tuning suggestions to users.

Self-tuning memory management in database systems has also gained much attention. Reference [1] proposes adaptive memory allocation in DB2 based on monitoring the characteristics of the workload during run time. Other commercial products also have implemented self-tuning memory management facilities to improve the performance of the database systems ([3], [4]). DBMS designers have worked hard to make the internals self-tuning and self-managing. Since how the internals work adaptively is beyond the control at the application level, techniques in this category are orthogonal to that of AppSleuth. AppSleuth works at the application code level. In our tuning consulting experience, changing application code can lead to 2 to 100 times performance improvements. Thus, tuning efforts at different levels can combine to obtain the best performance for database applications.

Oracle 11g has a helpful feature called Hierarchical Profiler [18] which can profile PL/SQL program executions with the number of calls and the elapsed time of subprograms and SQL statements., Hierarchical Profiler differentiates self time from descendant time within caller-callee relationships. The package offers analysis of the raw profile data and generation of a group of reports in HTML format. That achieves the same functionality as a subset of what we have in our analysis of SQL trace below.

Database applications' code run in two different contexts: the programming context, which deals with programming logic in languages like Java or Python, and the database context, which entails database accesses, such as SQL statement processing, stored procedure calls etc. Frequent switches between the two contexts will hurt performance seriously. Reference [37] proposes a way to partition database applications into two parts: one part runs on the application server, the other part runs on the database server. The goal is to minimize the roundtrips between the two servers while retaining the semantics of the original application. The proposal does an elegant job of allocating proper burdens onto eligible servers. But such optimized separation at the server level deals with only one of the factors that affect performance. One needs to look at the application level and its many "delinquent" design patterns.

There are some third-party tools to help database application developers produce program code of high quality. Quest Software's (Now Dell's) TOAD [24] is a proprietary tool that offers help at the application programming level for different DBMS's (see http://www.orafaq.com/node/846 for a tutorial explanation for Oracle's PL/SQL). Primarily, it consists of software engineering advice of the form "make your variable names self-describing" and encouragement to reduce code complexity as is measured by metrics such as McCabe's cyclomatic complexity (which measures the

* Work done while the author visited New York University under the support of China Scholarship Council's Graduate Education Program, and later partially supported by Natural Science Foundation of China (No. 61202331, 60170013, 60833005,61070055, 91024032, 91124001), and the National 863 High-tech Program (No. 2012AA010701, 2013AA013204).

† Work supported by U.S. National Science Foundation grants 0922738, 0929338, 1158273

EDBT'13, March 18–22, 2013, Genoa, Italy.

number of independent paths through program code -- the fewer the better).

Within TOAD, a special module called CodeXpert offers SQL performance tuning help. CodeXpert allows the user to invoke a pre-defined set of rules or to create new ones. The rules pertain to single SQL statements. An example would be: find all SQL statements that join more than four tables. A more sophisticated example is to find queries that have insufficient index support. To identify the latter, CodeXpert runs the queries through the Explain Plan facility. CodeXpert then simulates the addition of possibly useful indexes and reruns Explain Plan. Such tools work well for single SQL statements. AppSleuth's methodology extends the tuning capabilities to the multi-statement level.

Other third-party tools, proprietary or open source, which can do sophisticated static code analysis include klocwork[31], fortify[32], coverity[33], Enterprise Architect[34], Findbugs[35], PMD[36], etc. All these tools can statically analyze code written in one or more of the languages like C/C++, java, C#, Delphi, VB etc. NIST annually holds a Static Analysis Tool Exposition (SATE) [27] to advance research in static analysis tools to find bugs related to security problems. But these tools or solutions analyze code structures and dependencies to find security vulnerabilities and programming bugs like resource leaks, unreferenced variables etc. and report the defects in details. They mainly work in the context of a specific programming language, ignoring database interactions.

Researchers from Microsoft proposed a static analysis methodology for database application binaries in a general sense [28]. Their method enhances traditional optimizing compilers with knowledge about data access APIs (e.g. ADO.NET) and database domains. The solution is based on a compiler framework, adopting data flow and control flow analysis customized for database access, forming "a layer of static analysis services for database applications", on top of which vertical tools are built with different functionalities such as detecting SQL injection vulnerabilities, "extracting the SQL workloads from the binaries", or identifying potential data integrity violations. The static analysis framework aims to make the application code more DBMS-friendly, but treats performance as one feature among many auxiliaries of collecting workload etc.

[29] proposed a profiling infrastructure that, during application run time, logs events from different contexts: instrumented application events, ADO.NET tracing and Microsoft SQL Server tracing. After correlating and matching traces from the application context and those from DBMS context, a summary/detail view is given involving various attributes like function names, execution time, number of invocations, SQL text, number of reads/writes, etc. The "global" profiled data view is the basis for database application developing tasks like detecting problematic functions which have caused DB server deadlocks, or suggesting query hints in application code. Profiling is helpful to spot performance problems and to give tuning suggestions. AppSleuth currently takes advantage of DB server side profiling together with static source code analysis to locate the delinquent design patterns in application code. Extensions to profiling in other contexts is part of the future work.

As database performance tuners, we applaud the general use of database tools that either suggest indexes or flag high-load SQL statements while other tools seem less database performance oriented (e.g. static code analysis tools or profiling tools). To us, static methods are inherently limited, because the performance of SQL statements depends on their runtime behavior (e.g. how often they are invoked, the size of the data on which they operate). On the other hand, many code design patterns that cause the greatest performance problems in database context may go beyond the single-SQL-statement level, spanning multiple statements, or sometimes even multiple programs. For example, loops may not be present in SQL but rather in Java or some other language that are not accessible to the tuner. So combining static and runtime analysis at the application level to find and validate the delinquent design patterns is necessary to improve database performance. That is the philosophy of AppSleuth.

2. DELINQUENT DESIGN PATTERNS

Even though hardware has become vastly faster over the last decades, database tuning continues to be necessary. The accepted reason for this is that databases grow in size as new data sources arise. The problem with this explanation is that indexes should have mitigated this effect enough so data access time would grow only with the logarithm of the data size, not linearly. But the reality is often the opposite. We think the deeper reason is that application programmers mistreat their databases in their code. Typical application delinquent design patterns include:

1) Insert records into a table one at a time, crossing protection boundaries and flushing the instruction cache each time. For example, the following code snippet (In Figure. 1) in Oracle PL/SQL runs for 20 minutes with appropriate indexes on the table *sku_word* (several hours without indexes) having about 3,400,000 rows and on the table *hotel_desc* with 220,000 rows. (A sku is a particular instance of a product type. In our running example, it's a particular room type in a hotel on a particular night.)

```
DECLARE
l_sku_id            INTEGER;
l_hotel_id          VARCHAR2(10);
l_room_type_id      INTEGER;
l_desc              hotel_desc.description%TYPE;
CURSOR c1 IS SELECT sku_id, hotel_id, room_type_id
FROM sku_word;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO l_sku_id, l_hotel_id,
l_room_type_id;
        EXIT WHEN c1%NOTFOUND;
        FOR item IN (SELECT description FROM
hotel_desc WHERE hotel_id = l_hotel_id AND
room_type_id = l_room_type_id)
        LOOP
            INSERT INTO drs_sku(id, description)
            VALUES (l_sku_id, item.description);
        END LOOP;
    END LOOP;
    CLOSE c1;
END;
```

Figure 1. Delinquent design patterns for insert.

By contrast, all the work can be done in one insert-select statement (Figure 2) in about one minute on the same hardware and with the same indexes (a factor of 20 times in improvement).

```
INSERT INTO drs_sku(id, description)
SELET sku_id, description
FROM sku_words, hotel_desc
WHERE sku_words.hotel_id = hotel_desc. hotel_id
 AND sku_words.room_type_id = hotel_desc.room_type_id;
```

Figure 2. An equivalent single insert-select statement that is 20 times faster.

2) Fetching one record at a time from within, say, a Java loop as opposed to selecting many records into an array. For example, the following java code queries the descriptions for the first 1000 hotels (Figure 3).

```
{
    ResultSet   rs = null;
    Statement   stmt = conn.createStatement();
    l_hotel_id = 1;
    while (l_hotel_id <= 1000)
    {
        rs = stmt.executeQuery("select description from
hotel_desc where hotel_id =" + l_hotel_id);
        while (rs.next())
        {…}
    }
}
```

Figure 3. Execute an SQL statement many times in a Java loop.

By contrast, the following code (Figure 4) issues one query to the database and fetches the result into a collection data type.

```
{
    ResultSet   rs = null;
    Statement   stmt = conn.createStatement();
    rs = stmt.executeQuery("select description from
hotel_desc where hotel_id between 1 and 1000");
    while (rs.next())
    {…
    }
}
```

Figure 4. Single SQL statement that implements the same functionality.

3). Processing one record at a time within the stored procedures of a database management system, testing for conditions within that record using if statements. For example, suppose that for each hotel and each room_type, the price varies based on the day of the week. We may have code like the following Figure 5 using if statements.

A better way would be to translate the if condition into one or more where clauses to update many records at a time (Figure 6). The speed-up is over 6 times for updating a Reservation table having approximately 40 thousand records.

```
…
IF weekday(this_date) == 0 THEN
    Price = Sunday_price;
…
ELSIF weekday(this_date) == 1 THEN
    Price = Monday_price;
ELSIF …
END IF;
```

Figure 5. Process data using IF statements to insert the price into a table *Reservation* for a particular date "this_date". (We omit constraints on hotel and room_type for the sake of exposition.)

```
UPDATE Reservation
SET price = Sunday_price
WHERE weekday(this_date) = 0;
```

Figure 6. The if condition becomes a where clause that can apply to many rows at a time.

But even this improvement would require 7 update statements, one for each day of the week. So an even better way would be to have a table of prices *S(hotel, room_type, dayofweek, price)* with 7 items for every *(hotel, room_type)* pair and then do a join (Figure 7), which, under the same setting, obtains a factor of three speed-up compared to 7-update-statement approach, bringing the overall time to 1/20th the time compared to the original "if-condition" way. If proper indexes are built on the involved tables, the final improvement goes to 1/500th the original time.

```
UPDATE Reservation
SET price = (SELECT price FROM S
        WHERE
    weekday(Reservation.this_date) = S.dayofweek
        AND  Reservation.hotel = S.hotel
        AND Reservation.room_type = S.room_type)
```

Figure 7. Code after introducing a new table. Here we have included the hotel and room_type constraints, so the full logic of the query is given.

These examples of poor performance in the initial design show the tendency of programmers to do record-at-a-time programming as opposed to set-at-a-time programming. This is compounded by the use of stored procedures, because a subprogram A may loop on records and call a subprogram B for each record. B may do some joins and then again call subprograms for each record produced. Tools like the Oracle Tuning Advisor or CodeXpert don't look for such problems.

These problems may appear to be a symptom of what Dave Maier famously called the "impedance mismatch" between the record-at-a-time C-style language and the bulk database language. But the deeper problem is that application programmers perversely embrace the impedance mismatch by treating the database as a giant record store. Programmers are trained (in schools, alas) to write programs on small amounts of data. As a result, they reach the workplace and test their program on 100 record databases. With such small databases,

inserting records one at a time works blindingly fast. They are then surprised when it takes hours to insert a million records.

4). Denormalizing tables for convenience of query performance at horrendous costs to updates

This is a schema rather than code delinquent design pattern. From our tuning experience, we have noticed that delinquent designs occur together – denormalization, record-at-a-time processing, poor use of indexes and excessive use of subqueries happen in close proximity to one another.

Code copying causes delinquent design patterns to proliferate across an application. The tuner normally doesn't have time to correct every problem. For this reason, it is essential to know which procedures are costing the most time. To do this, a tool must examine the database statements found in the log and determine where they come from. The goal is to find the superdelinquents -- delinquent design patterns that take up lots of time -- and then turn them over to a competent tuner.

AppSleuth both analyzes code and the DBMS's tracing facility to discover superdelinquents. AppSleuth's current implementation targets Oracle PL/SQL and Microsoft TSQL. We plan to implement versions for other popular commercial DBMS's and other delinquent design patterns (as found in database tuning books [25] and online guides) in other popular programming languages in the future. The basic architecture – performs a global parse, identifies critical paths, and matches against the database trace – will change little. We are happy to share the source code of AppSleuth to members of the community who are interested to work on it.

The rest of this paper contains three sections: the architecture of AppSleuth, a case study, and a conclusion.

3. COMPONENTS OF APPSLEUTH

AppSleuth parses and analyzes the application source code, collects useful statistics from the tracing log, detects potential critical hot spots in them and presents visualized output to a tuner. AppSleuth has four main parts: (i) a parser which underlies both (ii) a structure analyzer for the application source code, and (iii) a log analyzer for trace files. All three components feed a (iv) visualization output generator. The different components are shown in Figure 8.

In addition, AppSleuth builds on top of Oracle's DBMS advisor to suggest physical design decisions based on the collected workload of SQL statements. Our recommendation is to consider those changes after eliminating delinquent design patterns.

3.1 Brief Introduction to PL/SQL and Transact-SQL

PL/SQL and Transact-SQL (TSQL for short) are two full (Turing-Complete) programming languages at the database server side which includeprocedures, conditionals, loops, error handlings, and integrated SQL. PL/SQL offers subprogram overloading while TSQL does not. In order to do a global analysis of performance issues, AppSleuth parses the code and identifies delinquent design patterns in semantic actions. Because these patterns include loop and subroutine calls, the parser has to detect blocks, subprograms possibly at different levels, and different kinds of loops (e.g. basic, for, while, and cursor loops for PL/SQL).

Here are specifications of for loop constructs in PL/SQL (Figure 9):

```
for_loop_statement ::= [<<label_name>>]
    FOR index_name IN [REVERSE] lower_bound .. upper_bound
    LOOP
        Statements
    END LOOP [label_name] ';'

cursor_for_loop_statement ::= [<<label_name>>]
    FOR record_name IN (cursor | '('select_statement ')' )
    LOOP
        Statements
    END LOOP [label_name] ';'
```

Figure 9. Syntax for basic and cursor for loops statements in PL/SQL.

The following is the syntax of loops in TSQL[26] (Figure 10):

```
while_statement ::=
    WHILE Boolean_expression
        { sql_statement | statement_block | BREAK | CONTINUE }
```

Figure 10. Syntax for while loop statements inTSQL.

3.2 Inputs to AppSleuth

AppSleuth takes one or more source code files as inputs and locates delinquent design patterns as well as forms the intra- and inter-file call graph. For example, in our running travel application case study, AppSleuth for PL/SQL reads in all the source code files, analyzes the structure in each file and finds the inter-file calling relationships between them. After viewing the inter-file call graph, the tuner can zoom into one specific file to look at its internal structures and intra-file call graph which illustrates nested subprogram calling logic. AppSleuth for TSQL works in a similar but more straightforward way because in TSQL all the subprograms are standalone. It doesn't have PL/SQL's packaged or nested subprograms.

AppSleuth locates the relevant programming language code by comparing the database trace against programming language files that issue SQL statements as well as stored procedures. The parser also records execution times to see which programming language source code files and stored subprograms require special attention.

3.3 Output of AppSleuth

The output of AppSleuth presents a global picture of database problems by showing a call graph with critical paths highlighted. Besides the visualization, AppSleuth also (i) collects the SQL workload of the application, which is very helpful for physical database design, (ii) counts the number of SQL statements in subprograms, and (iii) analyzes SQL statement attributes such as the number of tables referenced in a SQL statement. AppSleuth for PL/SQL can also output index building suggestions by calling the database tuning advisor interface with the workloads collected during the analysis.

3.4 AppSleuth Static Code Analysis

We have built a lexer and a parser using flex [39] and bison [38] respectively for both PL/SQL and TSQL source code, implementing the full grammar of each of those languages. During static analysis, the LALR parser scans each PL/SQL or TSQL source file and analyzes its structure to detect loops and subroutine calls as well as more local performance-related features such as the number of SQL statements in subprograms, the variables and arguments which are declared but never referenced in the source code, and the number of tables in SQL statements. The output generator produces a call graph with thin arrowed-lines for calls from the top level of the caller procedure to the callee procedure and thick arrowed-lines if the caller procedure makes the call from within a loop, which suggests a possibly delinquent pattern that might hurt performance. Figure 11 illustrates this.

3.4.1 Finding Loop Structures

Inside loop statements there is much information worth analyzing. For example, SQL statements within cursor loops are a delinquent design pattern. Replacing them by a single SQL statement might help as we saw in section 2.

3.4.2 Finding Subprogram Calls

AppSleuth must determine which subprogram is being called in the source code based on the name of the subprogram and the calling parameters. Because PL/SQL allows overloading of nested-level subprogram and packaged subprogram names, AppSleuth for PL/SQL examines all subprogram overloading mechanisms as well as forward subprogram declaration mechanisms to disambiguate subroutine calls having the same name based on discerning different argument lists.

When a callee, say Y, is called by a caller, X, at different locations in the code, the graph uses the “most pessimistic” call, i.e. the one from the most deeply nested loop, to represent the caller-callee relationship of X and Y.

3.5 AppSleuth Trace File Analyzer

The goal of trace file analyzer is to get the profiling information of subprograms calls and SQL statements in terms of duration, number of executions, etc., and to combine the profiling information with the static code analysis to get a whole picture of the potential delinquent design patterns in the source code. This will also give insight into delinquent patterns in external languages such as Java and C.

The hierarchical profiler in Oracle provides a temporal trace of the basic events at the call stack of subprograms and records simple information for each individual event such as subprogram entrance and returns, elapsed time between neighboring events, etc. The subprogram duration and number of executions of a particular code segment can be obtained by parsing and analyzing such a trace file.

SQL Server can trace over 200 kinds of events. Start and complete events for stored procedures can trace the performance times of stored procedures. SP:StmtStarting and SP:StmtCompleted can be utilized to trace stored function calls, if any. SQL Server’s trace files are binary files which cannot be directly parsed by AppSleuth. So a built-in function (fn_trace_gettable) loads the trace file into tables which can be exported as a text file and analyzed by AppSleuth to get information like duration of stored subprograms, numbers of executions, etc.

In both systems, the profiling mechanism gives more detailed performance information like CPU time, number of block reads, and so on for each SQL statement execution. Oracle has a separate SQL trace functionality. SQL Server presents SQL trace using the same uniform view as all the other tracing events. But neither system relates SQL statements to the stored subprograms which issue them. AppSleuth does this.

AppSleuth links those SQL statements back to the PL/SQL source code files as follows: When parsing the source code files, AppSleuth collects the static SQL statements of a procedure into a “footprint”. Because some of these SQL statements appear inside a conditional or inside a looping construct, they may appear in the trace several times, perhaps with slight changes to constants. AppSleuth parses the SQL trace file and determines which standalone stored subprogram left footprints in the SQL trace file. If more than one subprogram contains the same SQL statement s, then neighboring SQL statements in the trace may help to disambiguate the source of s. For example, if s1 could come from subprograms P1 or P2 and s2 could come from P2 or P3, then if the trace shows s1 and s2 in close proximity, they probably come from an invocation of subprogram P2.

If there is no such stored procedure, then statements that differ only by a constant or in some other minor way may come from a programming language (e.g. Java, C) loop. The footprint notion identifies which SQL statements belong to which stored procedure or suggests the need to look at some programming language code that might be causing the issue. Our current implementation performs this trace analysis for Oracle. Achieving this on SQL Server is still in the works.

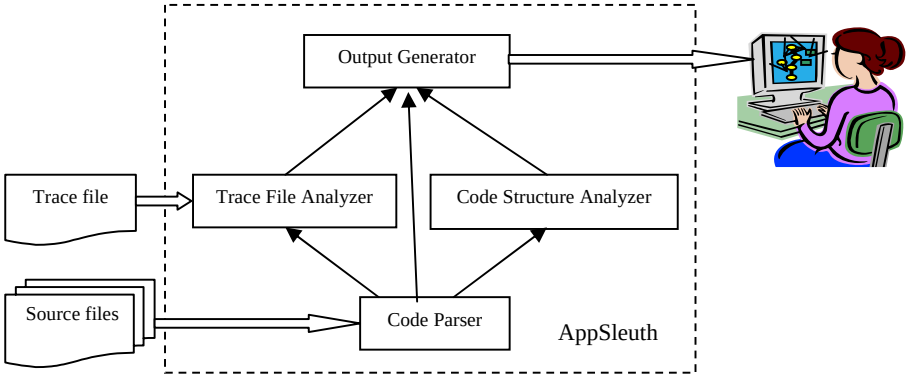

Figure 8. Components of AppSleuth. Source files are code. The trace file contains SQL that hits the database, but does not identify the source of that SQL.

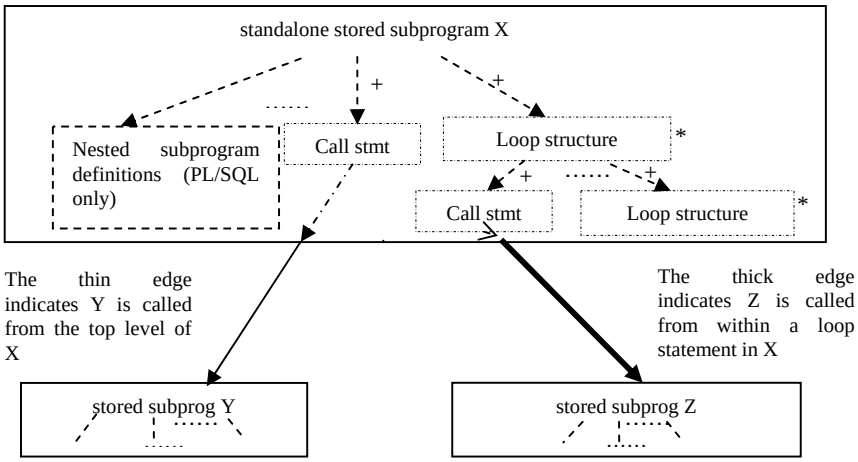

Figure11. AppSleuth's anatomy of subprograms and two kinds of edges between caller and callee in its output. Inside subprogram *X*, dash lined arrows with sign '+' means multiple call statements or loop structures may exist; sign '*' near loop structures means these structures can be nested to arbitrary depth. We omit other kinds of information using ellipsis.

4. TRAVEL IS US: a sanitized case study

This section presents a case study of global tuning at the application level. The application is a web-based travel agency whose database consists of 1000 hotels, each having between one and fifteen room types. A room type could be "double room with sea view", "suite with balcony", etc. There are approximately 1500 different room types for all the hotels. Each hotel for each room type may charge different amounts depending on the day of the week (or the season, though season and vacation periods are processed separately). A customer can make a reservation for a certain number of rooms of one or more certain room types in one or more hotels for a period of time. So a certain room type in a certain hotel on a given date forms a sku.

In the application, every room type in every hotel has a literal description in English (the base language). The descriptions must be translated into 10 other languages.

This excerpted part from the application deals with translating the descriptions for designated languages for each sku.

4.1 Schema Information

Tables involved in this part of the application include (throughout this example, we present only those columns relevant to tuning; all indexes are non-clustered):

4.1.1 trans_dict

The table *trans_dict* (Figure 12) stores the dictionary of translations for all descriptions in all languages. Here the column *phrase* stores the description in the language indicated by the column *lang*; each description, indicated by *desc_id*, is stored in as many rows as there are the languages. So the primary key of *trans_dict* is *(desc_id, lang)*.

```
trans_dict (
    desc_id         SMALLINT,
    phrase          VARCHAR2(255),
    lang            CHAR(2)
```

Figure 12. Columns of table *trans_dict*, with primary key *(desc_id, lang)* and an index on *desc_id*

4.1.2 sku_translated

The table *sku_translated* (Figure 13) stores all the already translated descriptions for the skus. During the processing of each sku, the translations of its description to all languages are appended to the table *sku_translated*. This is by far the largest table in the application. The primary key for this table is *(sku_id, lang)*.

```
sku_translated (
    sku_id          SMALLINT,
    translated      VARCHAR2(255),
    lang            CHAR(2),
    …
)
```

Figure 13. Columns of *sku_translated* with primary key *(sku_id, lang)*

4.2 Pseudo Code of the Application

In the application's initial design, each hotel is processed as follows:

4.2.1 skut_manager

Skut_manager receives as an input argument a hotel id and calls *skut_loop* to do the translation of all room types for all dates (i.e. all skus) for this hotel unless the hotel needs to be checked (Figure 14).

```
skut_manager(hotel_id)
1. Get the status for hotel_id, and from_language, to_language, for its translation
2. If the hotel's status is 'need checking' then
     skut_check(hotel_id, from_language, to_language);
   Else if the hotel's status is 'passed checking' then
     skut_loop(hotel_id, from_language, to_language);
   End if;
```

Figure 14. Pseudo-code for *skut_manager*.

4.2.2 skut_loop

Procedure *skut_loop* (Figure 15) just does the translation for each sku through the procedure *skut_tran*.

```
skut_loop(hotel_id, home_lang, target_lang)
    For every sku (hotel_id, room_type, date) of the given hotel
        call skut_tran to do the actual translation for the current sku of its description in the home language;
    End loop;
```

Figure 15. Pseudo-code for *skut_loop*.

4.2.3 Other stored procedures along the way

In *skut_tran*, the step of performing the translation is implemented by the stored procedure *skut_tran_sku*.

Procedure *skut_tran_sku*, in turn, calls *skut_sku_dict* to look up the dictionaries for the designated translation of the sku. After every translated entry for the sku is returned, the procedure inserts a row into *sku_translated*.

4.3 AppSleuth in Application Tuning

The first graph (Figure 16) presents the analysis of structure (before the analysis of the trace log). The graph shows more than we've discussed, but one can see the flow from *skut_mangager* through *skut_loop* in the description translation path. It turns out that another path translates "attributes of rooms" though we don't analyze this further.

Calls from within loops are represented by bold edges and the "loop layer" is the depth of the nested loop in the application. The line numbers of the calls are also shown.

For purposes of exposition, we restrict our attention to the core of the application.

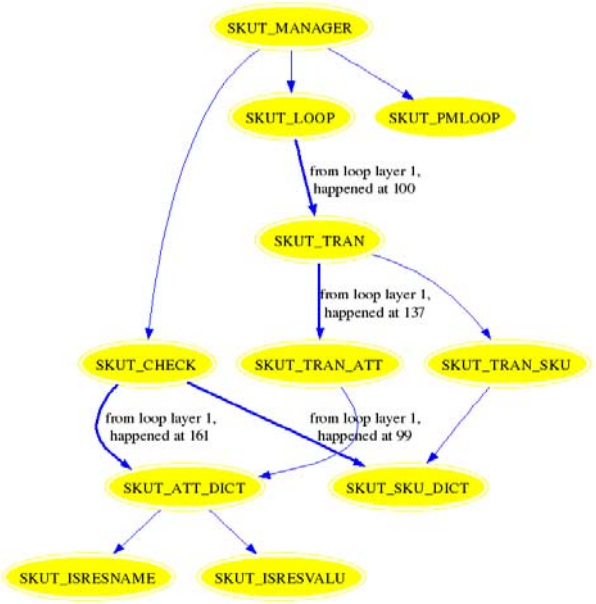

Figure 16. Output of AppSleuth for the original application code

4.3.1 Two Other Working Tables

- *hotel_desc* table:

Table *hotel_desc* (Figure 17) records descriptions in English for hotel-roomtype pairs. Translating such descriptions from English to all other languages entails a lookup in the dictionary table *trans_dict* and the appending of the translated descriptions to the table *sku_translated*. The primary key of *hotel_desc* is *(hotel_id, room_type_id)* pair. There is an index on columns of *(hotel_id, room_type_id)*.

```
hotel_desc (
        hotel_id            SMALLINT,
        room_type_id        SMALLINT,
        descriptioninEN     VARCHAR2(255)
    )
```

Figure 17. Columns of the description table for hotels and room types with primary key *(hotel_id, room_type_id)*. There is an index on *(hotel_id, room_type_id)*

- *sku_def* table:

Table *sku_def* (Figure 18) records the mapping from all the generated skus to hotel – roomtype pairs. The primary key is *sku_id*.

```
sku_def (
        sku_id          SMALLINT,
        hotel_id        SMALLINT,
        room_type_id    SMALLINT
    )
```

Figure 18. Columns of the table *sku_def*, with primary key *sku_id* and an index on *(hotel_id, room_type_id, sku_id)*

There is an index on the columns of *(hotel_id, room_type_id, sku_id).*

4.3.2 The Original Application and Stored Procedures Involved

The original application is shown in Figure 19 for one typical execution of processing 10 hotels. The processing logic in pseudo code is as follows:

```
input: a set of hotel ids
 for each hotel_id,
      find all the skus in this hotel.
      For every such sku, get its description in English
           For all the supporting languages
                Append the description in the current
                language for the sku
```

Figure 19. Pseudo-code for the original application design.

The application core consists of the following stored procedures:

- *manager*
- *preparehotel*
- *skuttran*
- *insertsku*.

Stored procedure *manager* (Figure 20) receives a set of hotel ids to work on. For each hotel id, *manager* calls *preparehotel* to prepare for the translation. The pseudo code is like the following:

```
manager(a set of hotel_ids)
    For each hotel_id
        Call preparehotel(hotel_id)
    End for;
```

Figure 20. Pseudo code for *manager*.

Stored procedure *preparehotel* (Figure 21) finds all the skus belonging to the hotel, and does translation for each sku:

```
preparehotel (i_hotel_id)
    Find all the skus belonging to this i_hotel_id from
sku_def;
    For each sku
        get its description from the hotel_desc table;
        do translation for this description (calling
skuttran(sku_id, descriptioninEN))
    End for;
```

Figure 21. Pseudo code for *preparehotel*.

Stored procedure *skuttran* (Figure 22) does the translation of a sku's English description into all the languages:

```
skuttran(sku_id, descriptioninEN)
    Find the desc_id for this descriptionEN in trans_dict
    For each of the phrases with the same desc_id
        Call insertsku to do the insertion.
    End for;
```

Figure 22. Pseudo code for *skuttran*.

The last stored procedure *insertsku* (Figure 23) does the insertion into *sku_translated*. The pseudo code is

```
insertsku(sku_id, description, language)
    insert into sku_translated(sku_id, description, language);
```

Figure 23. Pseudo code for *insertsku*.

4.3.3 AppSleuth's output without a trace file

After analysis of the code, AppSleuth outputs the call graph of Figure 24. We can see the loop structures detected by AppSleuth which form a critical path.

4.3.4 AppSleuth's output with a trace file

After doing the translation for a set of 10 hotels with the execution traced, AppSleuth outputs the result with trace analysis in Figure 24. The brown edges show the actually executed calls. The call graph does a best effort guess of the number of times each stored procedure has executed. The elapsed time in each node is the total execution time of that stored subprogram. So the time shown in the top procedure manager is the total elapsed time for processing translations for 10 hotels (including all subroutines).

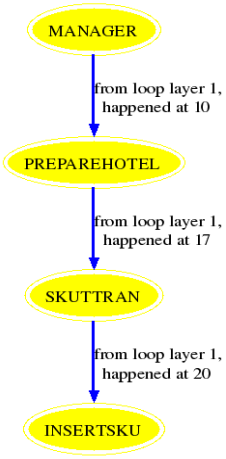

Figure 24. Output of AppSleuth of the original simplified version.

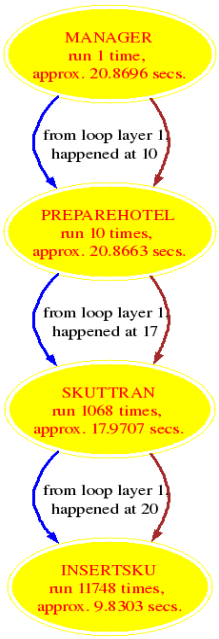

Figure 25. Output of AppSleuth for the original application code as well as the trace.

The graph of Figure 25 shows that the delinquent design pattern starting at *preparehotel* is in fact a superdelinquent, because the total elapsed time is large and the number of subroutine calls grows as one descends the tree from 10 calls to 1068 calls to 11748. (We applied both the Oracle SQL Tuning Advisor and Quest SQL Optimizer, but neither recommended any changes.)

4.3.5 Table design improvement

A tuner looking at this graph would follow the critical path from *preparehotel* to *skuttran* to *insertsku* and start to take a look at the queries and the table design. Analysis of the code shows that translations are done for each sku. The inserted description for each sku depends on the possible language. There are 11 languages involved in the application, so each of the 1068 skus in the 10 hotels is inserted into *sku_translated* table for all the 11 languages (1068 * 11 = 11748) .On the other hand, the call to the translation routine depends only on *hotel_id* and *room_type*. (This makes sense because the description “double bedroom with a sea view” does not change over time.) So the denormalization of *sku_translated* table is one root cause of the slow performance.

On the other hand, lots of (unshown) application code depends on the existence of the *sku_translated* table, so we first consider how to insert into it more efficiently. We do so by taking descriptions from a table that depends only on *hotel_id, room_type_id*. So the first fundamental improvement is to alter the *hotel_desc* table by replacing *descriptioninEN* by *desc_id* (having values from the domain of *trans_dict.desc_id*) (Figure 26).

```
hotel_desc (
        hotel_id        SMALLINT,
        room_type_id    SMALLINT,
        desc_id         SMALLINT
    )
```

Figure 26. Optimized table schema for *hotel_desc*.

To shorten the length of the critical path of repeatedly called subprograms, given the *i_hotel_id* as the input argument, the insertion into *sku_translated* table can be implemented using one insert-select statement in a three table join (Figure 27).

```
INSERT INTO sku_translated (sku_id, translated, lang)
SELECT sku_def.sku_id, trans_dict.phrase, trans_dict.lang
FROM sku_def, hotel_desc, trans_dict
WHERE sku_def.hotel_id = hotel_desc.hotel_id
  AND sku_def.room_type_id = hotel_desc.room_type_id
  AND hotel_desc.hotel_id = i_hotel_id
  AND hotel_desc.desc_id = trans_dict.desc_id
```

Figure 27. A single insert-select replaces nested loops.

This improvement greatly reduces the numbers of calls and the elapsed time as shown by Figure 28:

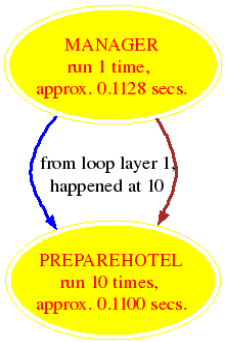

Figure 28. AppSleuth's output after the first improvement.

Specifically, the total elapsed time improves by a factor of nearly 200 (from 21 seconds to 0.11 seconds). The call graph is of course radically simplified too, potentially enhancing maintainability.

4.3.6 *Second Improvement of the Application*

Reexamining the table schema design of the application, we noticed that it would be beneficial to reduce the three-table join to a two-table join by adding the *desc_id* column to the *sku_def* table instead of to the *hotel_desc* table. Although this denormalizes the *sku_def* table, the number of rows remains unchanged and one table is eliminated from the join. (We tried Quest SQL Optimizer and Oracle SQL Tuning Advisor to tune the SQL statement of Figure 27, but neither suggested any improvement.) Table *sku_def* becomes (Figure 29):

```
sku_def (
        sku_id          SMALLINT,
        hotel_id        SMALLINT,
        room_type_id    SMALLINT,
        desc_id         SMALLINT
    )
```

Figure 29. Optimized table schema for *sku_def* to store description ids.

The insert-select with the two-way join is much simpler (Figure 30):

```
INSERT INTO sku_translated(sku_id, translated, lang)
SELECT sku_def.sku_id, trans_dict.phrase, trans_dict.lang
FROM sku_def, trans_dict
WHERE sku_def.hotel_id = i_hotel_id
  AND sku_def.desc_id = trans_dict.desc_id
```

Figure 30. An even more optimized insert-select statement.

Denormalization improves the query performance by a factor of nearly 50% as shown in Figure 31.

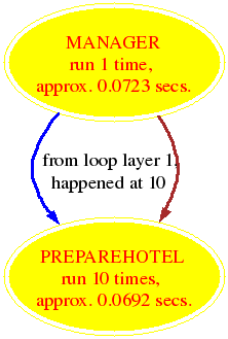

Figure 31. Output of AppSleuth after the second improvement.

Overall, these two improvements reduce the overall elapsed time, by a factor of 300 compared to the original design (from 21 seconds to 0.07s). This occurred without changing indexes, the buffer management, or hardware. No tool that we know of would help point the way leading to either improvement.

5. CONCLUSION AND FUTURE WORK

AppSleuth parses database engine source code and the trace log. Happily, it does not need to parse programming language (e.g. C++, Java, R etc) code. The reason is that delinquent design patterns in the programming context can be detected by seeing their effects on the trace log. For example, a cluster of queries that differ only by a constant indicate an iteration through a loop in some external programming language context. Further, timing information from the database trace log helps to find those delinquents that are on a critical path, the "superdelinquents". AppSleuth displays these in a global flow graph to focus the attention of a tuner who can often (as in our sanitized travel application example) improve performance by an order of magnitude or more. As far as we know, this is the first global application code analyzer for database tuning ever built.

We have focused on the misuse of loops, because that was the most challenging-to-detect tuning problem we knew of that has great practical importance. Detecting other tuning bugs (like sequences of SQL statements that take a long time) falls out naturally.

Future work includes generalizing the tool to discover other delinquents and exploiting the synergy between our tool and statement-at-a-time and physical design tools. The eventual goal is to go beyond the detection of problems to explicit suggestions for improvement. Right now, that is the programmer's job.

When we do database tuning professionally, we find that we can sometimes so much improve applications by correcting delinquent design patterns that we upset our clients. It's remarkably hard to show an application programmer that his or her "extremely complicated" application which takes 9 hours in production can in fact run in under a minute using much less code. Often such a programmer will ignore the suggestion. With a tool like AppSleuth, the tuner can deflect the anger to the software.

6. ACKNOWLEDGEMENTS

We would like to thank Laura Puglisi for helpful discussions as well as both the conference reviewers and our shepherd for their thoughtful comments.

7. REFERENCES

[1] Storm, A. J., Garcia-Arellano, C., Lightstone, S., Diao, Y., and Surendra, M. Adaptive self-tuning memory in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)* (Seoul Korea, September 12 – 15, 2006). VLDB Endowment, pp 1081-1092.

[2] Baryshnikov, B., Clinciu, C., Cunningham, C., Giakoumakis, L., Oks, S., and Stefani, S. Managing query compilation memory consumption to improve DBMS throughput. In *Proceedings of he 3rd Biennial Conference on Innovative Database Systems Research (CIDR'07)* (Asilomar, CA, January 7 – 10, 2007). www.crdrdb.org, 2007, pp 275 – 280.

[3] Dageville, B., and Zait, M. SQL memory management in Oracle 9i. In *Proceedings of the 28nd International Conference on Very Large Data Bases (VLDB'02)* (Hong Kong China, August 20 – 23, 2002). VLDB Endowment, pp 962- 973.

[4] Microsoft Corporation. SQL Server 2005 books online: Dynamic memory management. *SQL Server product documentation.* (September 2007), DOI = http://msdn.microsoft.com/en-us/library/ms178145(SQL.90).aspx.

[5] Larson, P., Graefe, G., Memory management during run generation in external sorting. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)* (Seattle, Washington, June 2 – 4, 1998). ACM Press, New York, NY, 1998, pp 472 – 483.

[6] Weikum, G., Hasse, C., MoenKeberg, A., and Zabback, P. The COMFORT automatic tuning project. Invited Project Review. *Inf. Syst., 19, 5* (Jan. 1994), pp 381 – 432.

[7] Zilio, D., Rao, J., Lightstone, S., Lohman, G., Storm, A. J., Garcia-Arellano, C., and Fadden, S. DB2 Design Advisor: integrated automatic physical database design. . In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)* (Toronto, Canada, August 31 – September 3, 2004). Morgan Kaufmann, San Francisco, CA, 2004, pp 1110 – 1121.

[8] Oracle Corporation. Performance tuning using the SQL Access Advisor. *Oracle White Paper*. (2007), DOI = http://otn.oracle.com.

[9] Agrawal, S., Chaudhuri, S., Koll{\'a}r, L., Mathare, A. P., Narasayya, V. R., and Syamala, M. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)* (Toronto, Canada, August 31 – September 3, 2004). Morgan Kaufmann, San Francisco, CA, 2004, pp 1110 – 1121.

[10] Bruno, N., and Chaudhuri, S. Automatic physical database tuning: a relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)* (Baltimore, Maryland, June 13 – 16, 2005). ACM Press, New York, NY, 2005, pp 227 – 238.

[11] Agrawal, S., Chaudhuri, S., Narasayya, V. R. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the 26nd International Conference on Very Large Data Bases (VLDB'00)* (Cairo, Egypt, September 10 – 14, 2000). Morgan Kaufmann, San Francisco, CA, 2000, pp 496 – 505.

[12] Kornacker, M., Shah, M., and Hellerstein, J. M., Amdb: a design tool for access methods. *IEEE Data Engineering Bulletin, 26, 2* (Jun. 2003), pp 3 – 11.

[13] Aboulnaga, A., Gebaly, K. EI., Robustness in automatic physical design. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT'08)* (Nantes, France, March 25 -29, 2008). ACM Press, New York, NY, 2008, pp 145 – 156.

[14] Papadomanolakis, S., Dash, D., Ailamaki, A., Efficient use of the query optimizer for automated physical design. . In *Proceedings of the 33th International Conference on Very Large Data Bases (VLDB '07)* (University of Vienna, Austria, September 23 – 27, 2007). ACM Press, New York, NY, 2008, pp 1093 – 1104.

[15] Babu, S., Bizarro, P., DeWitt, D., Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)* (Baltimore, Maryland, June 13 – 16, 2005). ACM Press, New York, NY, pp 107 – 118.

[16] Stillger, M., Lohman, G. M., Markl, V., Kandil, M., LEO: DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)* (Roma, Italy, September 11 – 14, 2001) Morgan Kaufmann, San Francisco, CA, 2001, pp 19 – 28.

[17] Raman, V., Markl, V., Simmen, D., Lohman, G., and Pirahesh, H., Progressive optimization in action. . In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)* (Toronto, Canada, August 31 – September 3, 2004). Morgan Kaufmann, San Francisco, CA, 2004, pp 1337 – 1340.

[18] Oracle Database Advanced Application Developer's Guide on Hierarchical Profiler. DOI = http://docs.oracle.com/cd/B28359_01/appdev.111/b28424/adfns_profiler.htm#g3157198.

[19] Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., Ziauddin, M. Automatic SQL tuning in Oracle 10g. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)* (Toronto, Canada, August 31 – September 3, 2004). Morgan Kaufmann, San Francisco, CA, 2004, pp 1110 – 1121.

[20] Oracle Corporation. The self-managing database: automatic performance diagnosis. *Oracle White Paper*, (2007) , DOI = http://otn.oracle.com.

[21] Dias, K., Ramacher, M., Shaft, U., Ventakaramani, V., and Wood, G., Automatic performance diagnosis and tuning in Oracle. In *Proceedings of he 2nd Biennial Conference on Innovative Database Systems Research (CIDR'05)* (Asilomar, CA, January 4 – 7, 2005). www.crdrdb.org, 2005, pp 84 – 94.

[22] Garcia-Arellano, C. M., Lightstone, S., Lohman, G., Markl, V., Storm, A., Autonomic features of the IBM DB2 Universal Database for Linux, UNIX, and Windows. *IEEE Transactions on Systems, Man, and Cybernetics special issue on Engineering Autonomic Systems, 36, 3* (May 2006), pp 365 – 376.

[23] Microsoft Corporation. SQL Server 2005 books online: Automating administrative tasks. *SQL Server product documentation*. (September 2007), DOI = http://msdn.microsoft.com/en-us/library/ms187061(SQL.90).aspx.

[24] Quest Software. Toad: SQL Tuning, Database Development & Administration Software. (2012), DOI = http://www.quest.com/toad/, 2012.

[25] Shasha, D., and Bonnet, P. *Database Tuning: principles, experiments and troubleshooting techniques.* Morgan Kaufmann, San Francisco, CA, 2002.

[26] Microsoft Tansact-SQL reference. DOI = http://msdn.microsoft.com/en-us/library/ms178642.aspx

[27] The SAMATE website. (2012) , DOI = ://samate.nist.gov/SATE.html

[28] Arjun Dasgupta, Vivek Narasayya, Manoj Syamala, A Static Analysis Framework for Database Applications, ICDE '09 Proceedings of the 2009 IEEE International Conference on Data Engineering, pp 1403-1414

[29] Surajit Chaudhuri, Vivek Narasayya, and Manoj Syamala, Bridging the Application and DBMS Profiling Divide for Database Application Developers, VLDB '07 Proceedings of the 33rd international conference on Very large data bases, pp 1252-1262

[30] Surajit Chaudhuri, Vivek Narasayya, Manoj Syamala, Bridging the application and DBMS divide using static analysis and dynamic profiling, SIGMOD '09 Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp 1039-1042

[31] The Klocwork website. (2012) , DOI = http://www.klocwork.com/

[32] The Fortify website. (2012) , DOI = https://www.fortify.com/

[33] The Coverity website. (2012), DOI = http://www.coverity.com/

[34] The Enterprise-architect homepage on Sparx Systems Website. (2012), DOI = http://www.sparxsystems.com.au/enterprise-architect.

[35] The FindBugs homepage on Sourceforge. (2012) , DOI = http://findbugs.sourceforge.net/

[36] The PMD homepage on SourceForge. (2012), DOI = : http://pmd.sourceforge.net/pmd-5.0.0/

[37] Cheung, A., Arden, O, Madden, S., Myers, A., Automatic Partitioning of Database Applications. *In Proceedings of the 38th International Conference on Very Large Data Bases (VLDB'12)*

(Istanbul, Turkey, August 27th – 31st, 2012). Morgan Kaufmann, San Francisco, CA, 2012, pp 1471-1482

[38] The GNU Bison Project (2012), DOI = http://www.gnu.org/software/bison

[39] Paxson, V. Flex. DOI = http://flex.sourceforge.net/