AppSleuth: a Tool for Database Tuning at the Application Level

ABSTRACT
Excellent work [1]][2][3][4][5][6] has shown that memory management and transaction concurrency levels can often be tuned automatically by the database management system. Other excellent work [7]][8][9][10][11][12][13][14] has shown how to use the optimizer to do automatic physical design. Our tuning experience across various industries (finance, gaming, data warehouses, and travel) has shown that enormous additional tuning benefits (sometimes amounting to two 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 a popular database management system in order to lead a competent tuner to the hot spots in an application. Specifically, this paper discusses representative application "delinquent design patterns", an application code parser to find them, a log parser to identify the patterns that are critical, and a display to give a global view of the issue. We present an extended sanitized example from a travel application to show the results of the tool at different stages of a tuning engagement. 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][8][9][10][11][12][13][14], system monitoring and maintenance [20][21][22][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 what-if analyses using a cost-based query optimizer. Beyond that effort in automatic physical design, Oracle's SQL Tuning advisor [19] 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.)

Self-tuning memory management in database systems has also gained much attention. [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].

Quest Software's TOAD[24] is a commercial tool that offers help at the application programming level (see http://www.orafaq.com/node/846 for a tutorial explanation). Primarily, it consists of software engineering advice of the form "make your variable names self-describing" and encouragement to reduce code complexity as measured by metrics such as McCabe's cyclomatic complexity (which measures the number of independent paths through program code -- the fewer the better).

Within TOAD, a special module called CodeXpert offers 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.

Whereas we applaud the general use of tools that either suggest indexes or flag complex SQL statements, many design patterns that cause the greatest problems span multiple statements, sometimes even multiple subprograms. That is the target 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 disk capacities increase. The problem with this explanation is that indexes

should have mitigated this effect enough so data access time would grow only with the log of the data size, not linearly. We think the deeper reason is that application programmers mistreat their databases. 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 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 in about one minute on the same hardware and with the same indexes (a factor of 20 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.

```
{
    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 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 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:

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

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

```
UPDATE DatePrice
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:

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

Figure 7. Code after introducing 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 school, 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 SQL statement tracing facility to discover superdelinquents. AppSleuth currently works only on Oracle PL/SQL. We plan to implement versions for other popular commercial DBMS's and other delinquent design patterns (as found in database tuning books and online guides) in the future. The basic architecture – perform a global parse, identify critical paths, and match against the database trace – will change little.

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.

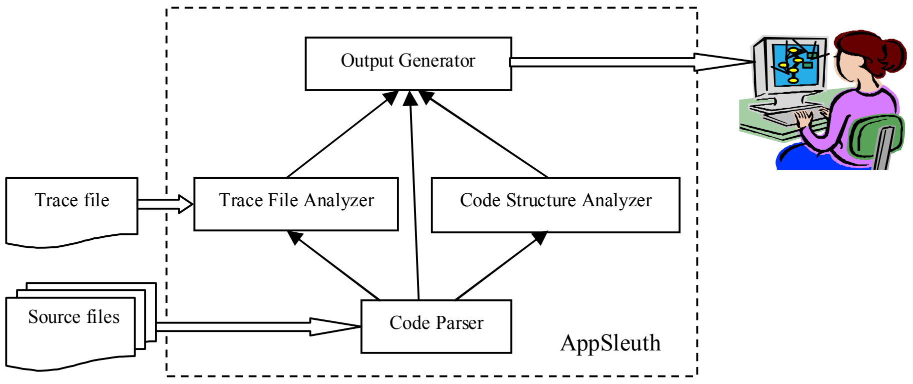

Figure 8. Components of AppSleuth. Source files are code. The trace file contains SQL that hits the database, but does not identify the source of that SQL.

1) Brief introduction to PL/SQL

PL/SQL is a full (Turing-Complete) programming language including procedures, conditionals, loops, exceptions, overloading, and integrated SQL. 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, functions, three kinds of subprograms, and four kinds of loops (basic, for, while, and cursor).

Here are two examples of loop constructs:

```
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.

2) Inputs to AppSleuth.

AppSleuth takes one or more source code files as inputs and locates delinquent design patterns as well as the intra- and inter-file call graph. For example, in our running travel application case study, AppSleuth 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.

A second input is the trace file holding the SQL statements that hit the DBMS, for example, the Oracle SQL trace files. Because some SQL that hits the database comes from programming language (e.g. Java, Matlab) files, problems may arise that have nothing to do with PL/SQL. AppSleuth finds the source of problems 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) Output of AppSleuth.

The output of AppSleuth presents a global picture of database problems by showing a call graph with critical paths highlighted.

3.1 AppSleuth Analysis

The LALR parser scans each 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 lines for calls from the top level of the calling procedure to the called procedure and thick lines if the calling procedure makes the call from within a loop.

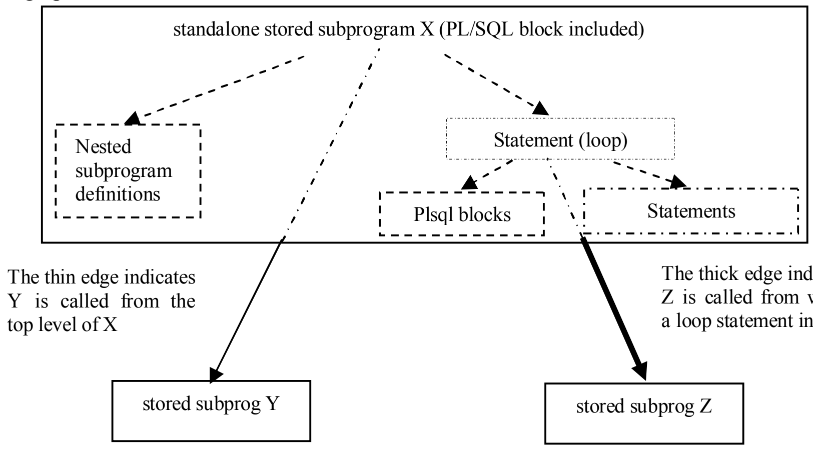
standalone stored subprogram X (PL/SQL block included)
Nested subprogram definitions
Statement (loop)
Plsql blocks
Statements
The thin edge indicates Y is called from the top level of X
The thick edge indicates Z is called from within a loop statement in X
stored subprog Y
stored subprog Z

Figure10. Example of AppSleuth's inter-file call edges.

3.1.1 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.

3.1.2 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 subprogram and packaged subprogram names, AppSleuth examines all subprogram overloading mechanisms as well as forward subprogram declaration mechanisms to disambiguate subroutine calls having the same name.

When some code block X calls Y several times, the graph represents the "most pessimistic" call, i.e. the one from the most deeply nested loop. That pessimistic call is likely to reveal a critical path.

3.2 Trace File Analyzer

SQL Trace files tell which SQL statements hit the database, when they begin and other information for the statements. These SQL statements can come from stored procedures or programming language (e.g. Java) code. AppSleuth links those SQL statements back to the 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 zero, one or more times. AppSleuth parses the trace file and determines which standalone stored subprogram left footprints in the 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. The eventual goal is to assign a time duration to each subprogram in PL/SQL or in a programming language.

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 2000 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 11) 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 11. 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 12) stores all the already translated descriptions for the skus. 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 12. 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:

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 13).

```
skut_manager(i_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 13. Pseudo-code for skut_manager.

2) skut_loop

Procedure skut_loop just does the translation for each sku through the procedure skut_tran.

```
skut_loop(hotel_id, home_lang, target_lang)
    For every sku (hotel, 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 14. Pseudo-code for skut_loop.

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 15) 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 loop in the application.

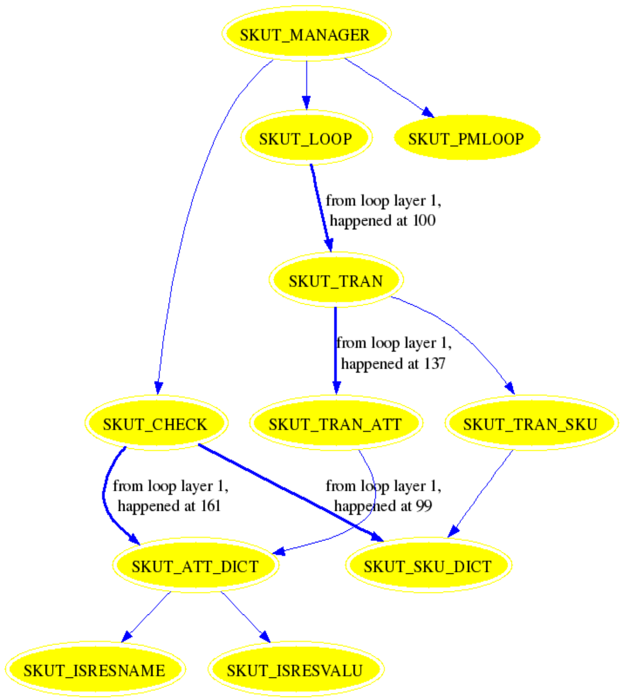
SKUT_MANAGER
SKUT_LOOP
SKUT_PMLOOP
from loop layer 1, happened at 100
SKUT_TRAN
from loop layer 1, happened at 137
SKUT_CHECK
SKUT_TRAN_ATT
SKUT_TRAN_SKU
from loop layer 1, happened at 161
from loop layer 1, happened at 99
SKUT_ATT_DICT
SKUT_SKU_DICT
SKUT_ISRESNAME
SKUT_ISRESVALU

Figure 15. Output of AppSleuth for the original application code

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

1) Two other working tables

- hotel_desc table:

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

Figure 16. 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)

Table hotel_desc (Figure 16) records descriptions in English for hotel-room_type 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.

- sku_def table:

sku_def table records the mapping from all the generated skus to hotel – room_type pairs. The primary key is sku_id.

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

Figure 17. 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).

2) Stored procedures involved

The application core consists of the following stored procedures:

- manager
- preparehotel
- skuttran
- insertsku.

Stored procedure "manager" 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

```
manager(a set of hotel ids)
    For each hotelid
        preparehotel(hotelid)
    End for;
```

Figure 18. Pseudo code for "manager".

Stored procedure preparehotel 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 19. Pseudo code for preparehotel.

Stored procedure skuttran does the translation of a sku's English description into all the languages:

```
skuttran(sku_id, descriptioninEN)
    Find the desc_id for this description in trans_dict
    For each of the phrases with the same desc_id
        insert into sku_translated with sku_id, phrase, and the corresponding language.
    End for;
```

Figure 20. Pseudo code for skuttran.

The last stored procedure insertsku does the insertion into sku_translated. The pseudo code is

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

Figure 21. Pseudo code for insertsku.

4.3.1 AppSleuth's output without a trace file

After analysis of the code, AppSleuth outputs the call graph of Figure 22. We can see the loop structures detected by AppSleuth which form a critical path.

4.3.2 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 23. 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

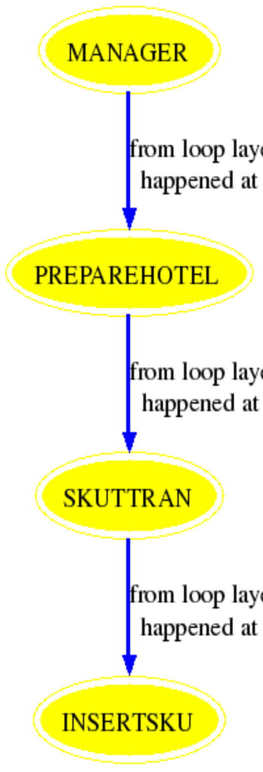
MANAGER
from loop layer 1, happened at 10
PREPAREHOTEL
from loop layer 1, happened at 17
SKUTTRAN
from loop layer 1, happened at 20
INSERTSKU

Figure 22. Output of AppSleuth of the original simplified version.

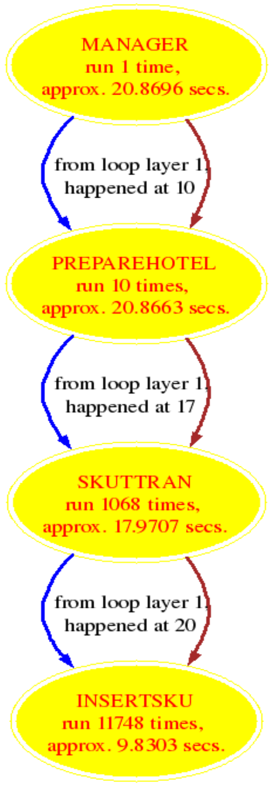
MANAGER run 1 time, approx. 20.8696 secs.
from loop layer 1 happened at 10
PREPAREHOTEL run 10 times, approx. 20.8663 secs.
from loop layer 1 happened at 17
SKUTTRAN run 1068 times, approx. 17.9707 secs.
from loop layer 1 happened at 20
INSERTSKU run 11748 times, approx. 9.8303 secs.

Figure 23. Output of AppSleuth for the original application code as well as the trace.

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).

The graph of Figure 23 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.)

4.3.3 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).

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

Figure 24. 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 25).

```
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 25. A single insert-select replaces nested loops.

This improvement greatly reduces the numbers of calls and the elapsed time as shown by Figure 26:

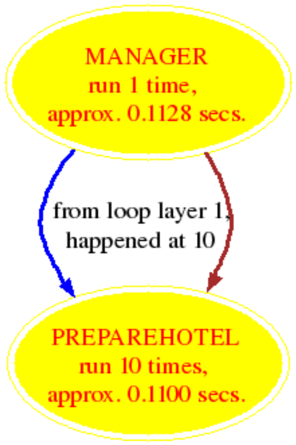

Figure 26. AppSleuth's output after the first improvement.

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.

4.3.4 .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 25, but neither suggested any improvement.) Table sku_def becomes (Figure 27):

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

Figure 27. Optimized table schema for sku_def to store description ids.

The insert-select with the two-way join is much simpler:

```
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 28. An even more optimized insert-select statement.

Denormalization improves the query performance by a factor of nearly 50% as shown in Figure 29.

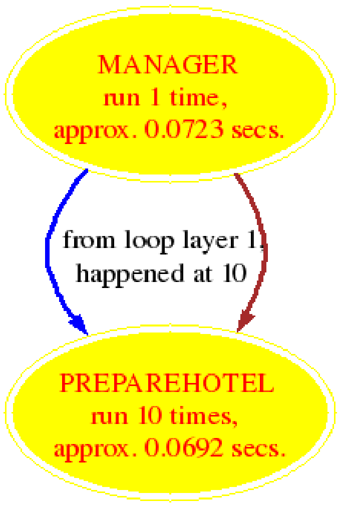

Figure 29. Output of AppSleuth after the second improvement.

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

AppSleuth parses stored procedure code and the trace log of a database application to find delinquent design patterns and uses the timing information from the database trace log 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 detection of problems to explicit suggestions for improvement.

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. 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, 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, 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, 962- 973.

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

[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, 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), 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 Fransisco, CA, 2004, 1110 – 1121.

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

[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 Fransisco, CA, 2004, 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, 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 Fransisco, CA, 2000, 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), 3 – 11.

[13] Aboulnaga, A., Gebaly, K. El., 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, 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, 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, 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 Fransisco, CA, 2001, 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 Fransisco, CA, 2004, 1337 – 1340.

[18] Ross, K. A., Cieslewicz, J., Rao, J., and Zhou J., Architecture sensitive database design: examples from Columbia group. *IEEE Data Engineering Bulletin, 28, 2* (Jun. 2005), 5 – 10.

[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 Fransisco, CA, 2004, 1110 – 1121.

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

[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, 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), 365 – 376.

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

[24] Quest Software. Toad: SQL Tuning Software for Database Development & Administration. http://www.quest.com/toad/, 2008.

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

[26] RYU, I. K., and Thomasian, A., Analysis of database performance with dynamic locking. *J. ACM 37, 3* (Jul. 1990a), 491 – 523. [Dennis: Are the following two for transaction tuning?]

[27] Tay, Y. C., Issues in modeling locking performance. In *Stochastic Analysis of Computer and communication Systems,* H. Takagi, Ed., North-Holland, New York, 631 – 658.