Tuning in Action

Wei Cao†
Department of Computer Science,
Renmin University of China
caowei@ruc.edu.cn

Dennis Shasha*
Department of Computer Science,
Courant Institute of Mathematical Sciences, New York University
shasha@cs.nyu.edu

MOTIVATION

Imagine that your database has all the right indexes. Its buffer manager has been tuned to give a high hit ratio, the buffer fits in RAM, and the data is well distributed on disk. You're done, right? Well, no, because the application code might be poorly written. It might include *delinquent design patterns.* The demoed tuning tool AppSleuth will find those delinquent design patterns but it is the demo visitor's job to fix them.

The demo scenario will consist of several "Test Your Skill" challenges with a tee-shirt as a prize. The scenarios will come from a transactional application, a Data Warehousing application, and an E-travel agency. For concreteness, we illustrate the functionality of AppSleuth on the E-travel agency here.

Keywords

Database tuning, application-level optimization, performance tool

1. BACKGROUND

Much excellent work has been done to study the tuning problem on the internals of DBMSs such as physical database design[1][3], database memory management[4], and buffer management. Commercial DBMS's provide useful tools to tune individual SQL statements as well. For example, Oracle's SQL tuning advisor[2] can tune SQL statements and recommend optimizations such as building indices, restructuring SQL, or obtaining statistics. Other tools like static code analysis tools [5][6][7] and profiling tools [8][9] are less database performance oriented. To us database performance tuners, 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). But it is often the case that an application runs slowly even though every SQL statement is well tuned. For example, loops, which hurt performance severely, may not be present in SQL but rather in Java or some other language that are not accessible to the tuner. The issue is that the problem does not lie at the individual statement level but rather in the overall structure of the code – what we call *delinquent design patterns.*

† 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/ICDT'13, March 18–22, 2013, Genoa, Italy.

2. EXAMPLES OF DELINQUENT DESIGN PATTERNS

Typical delinquent design patterns within an application are:

1) Overuse of Java loops to access the database

```
i = start;
while (i <= end)
{
        rs = stmt.executeQuery("select col1 from tab1 where col2 = " + i);
        while (rd.next()) { /* process the resule set */}
}
```

Figure 1. Overuse of Java looping SQL statements.

Figure 1 shows a JAVA code snippet which fetches one record at a time, while the code snippet in Figure 2 implements the equivalent functionality with a single SQL statement at much high speed.

```
{
        rs = stmt.executeQuery("select col1 from tab1 where col2 between
" + start + " and " + end);
        while (rd.next()) { /* process the resule set */}
}
```

Figure 2. Single SQL statement that realizes the same functionality more efficiently.

2) Excessive loops in stored subprograms

For example, applications might use cursor loops in stored procedures accessing one row at a time rather than use an equivalent SQL statement for all rows. Figure 3 and Figure 4 are equivalent but the latter runs approximately 20 times fater.

```
CURSOR c1 IS SELECT comm_col FROM tab1 FOR UPDATE;
…
LOOP
            FETCH c1 INTO temp_comm;
            EXIT WHEN c1%NOTFOUND;
            SELECT MIN(col2) INTO temp_col2 FROM tab2
WHERE comm_col = temp_comm;
            UPDATE tab1 SET col1 = temp_col2 WHERE
CURRENT OF c1;
```

Figure 3. Inefficient record-at-a-time method.

```
UPDATE tab1 SET col1 = (SELECT MIN(col2)
                        FROM tab2
                        WHERE tab2.comm_col =
                        tab1.comm_col);
```

Figure 4. An equivalent SQL statement processing many rows at one time.

3) Denormalization in schema design

The following Figure 5 is a piece of data from a denormalized table:

emp_id	department	dept_state
…	…	…
emp256	marketing	NY
emp257	marketing	NY
emp258	technology	CA
emp259	accounting	NY
emp260	technology	CA
emp261	technology	CA
---	---	---

Figure 5. Data piece in a denormalized table.

Denormalization exacerbates delinquent design patterns, because it tends to increase the number of loop iterations.

3. APPSLEUTH'S FUNCTIONALITY

To improve database performance, AppSleuth combines static and runtime analysis at the application level to find and validate the delinquent design patterns. AppSleuth's main functionality is to read in users' application source codes and trace files for the application execution, parse them, analyze the delinquent design patterns in the code (e.g. loop structures), collecting useful statistics (e.g. the time consumed by each stored procedure and the number of executions through a loop) and outputs the result from analysis and detection in a visualized global call graph.

The architecture of AppSleuth is in Figure 6:

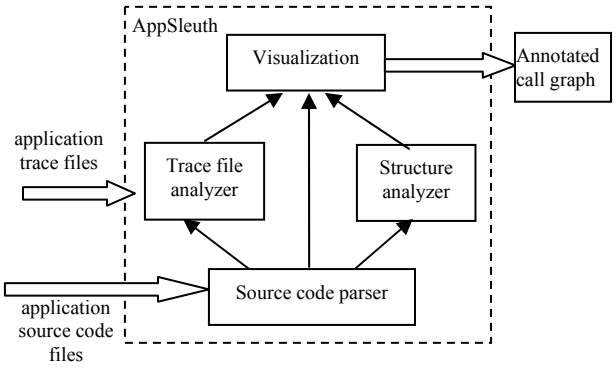

Figure 6. AppSleuth's architecture.

4. DEMO SCENARIO

This demo illustrates a human-system partnership. The visitor arrives and works on three scenarios. In each one, the visitor sees code, is shown the time a workload takes, and the trace graph (optionally – some visitors may think they don't need AppSleuth). Then the visitor is given choices about what to do. Each choice generates new performance statistics and a new trace graph. A visitor who gets the best choice in the first two tries on all three scenarios wins a tee shirt that says "I'm a super-tuner" with an appropriate graphic.

5. EXAMPLE DEMO

An E-travel application has 1000 hotels in the database. Each hotel has different room types such as "double room", "suite", etc. There are 1500 room types for all the hotels in the system. Each room type has a description in English. The application translates all the descriptions in English into 11 languages the application supports. Customers can reserve a certain number of rooms of one or more certain room types in one or more hotels for a certain period of time. So a certain room type in a certain hotel on a given date forms a sku. A lot of the application code (in other parts) depends on the descriptions for the relevant skus. So the following tables are involved in this part of the application:

trans_dict(desc_id, lang, phrase),

sku_translated(sku_id, lang, translated).

Table *trans_dict* acts as the dictionary for all the descriptions in all the 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*). During the processing of each sku, the translations of its description in all the languages are appended to the table *sku_translated*. This is by far the largest table in the application. Underlined column group indicates the primary key of the table. There is an index on *trans_dict(desc_id)*. To do the translation for skus, two other tables are:

hotel_desc(hotel_id, room_type_id, description_EN); and

sku_def(sku_id, hotel_id, room_type_id) with indexes on *hotel_desc(hotel_id, room_type_id)* and *sku_def(hotel_id, room_type_id, sku_id)*. Table *hotel_desc* 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*. Table *sku_def* records the mapping from all the generated skus to hotel–roomtype pairs. The primary key is *sku_id*.

5.1 Original application

The original application is shown in Figure 7 in pseudo code for one typical execution of processing 10 hotels.

```
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 7. Pseudo-code for the original design.

The application core consists of the following stored procedures:

- *manager*
- *preparehotel*
- *skuttran*

- *insertsku.*

Stored procedure *manager* (Figure 8) 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 hotel_id
        Call preparehotel(hotel_id)
    End for;
```

Figure 8. Pseudo code for *manager*.

Stored procedure *preparehotel* (Figure 9) 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 9. Pseudo code for *preparehotel*.

Stored procedure *skuttran* (Figure 10) 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 10. Pseudo code for *skuttran*.

The last stored procedure *insertsku* (Figure 11) does the insertion into *sku_translated*. The pseudo code is

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

Figure 11. Pseudo code for *insertsku*.

AppSleuth detected four stored subprograms involved in the delinquent design patterns (Figure 12). Yellow ellipses represent stored procedures. Blue edges are call relationships. Brown edges are calls that are actually executed.

5.2 What the Demo Visitor Will Do

You are given the following independent tuning choices concerning the application code optimization. Each choice is implemented, executed, traced and eventually fed to AppSleuth for detection and visualization.

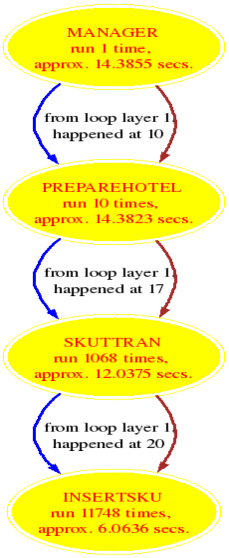

Figure 12. AppSeluth's analysis of the original application design.

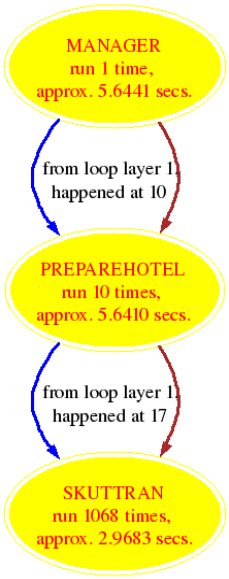

Figure 13. AppSleuth's output after Tuning 1.

5.2.1 Tuning 1- loop elimination in skuttran

In *skuttran*, eliminate the loop of insertion into *sku_translated* table one language at a time by one insert-select for each sku with multiple languages. Implementing such code optimization, with the typical traced execution out of 5 runs, AppSleuth outputs its new detection as Figure 13.

5.2.2 Tuning 2 – replace looped insert multiple table join after redesigning the schema

Another option is to use an insert-select to insert all translations for one hotel into *sku_translated*. The insert-select involves a three-table join between *hotel_desc*, *sku_def*, and *trans_dict* as in Figure 14. For better join performance, change the *hotel_desc* schema to hold *desc_id* instead of *description_EN*. AppSleuth's output for this tuning choice is in Figure 15:

hotel_desc becomes *hotel_desc(hotel_id, room_type_id, desc_id).*

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

Figure 14. Tuning 2's optimization highlighted.

5.2.3 *Tuning 3 - reducing the number of joins by denormalization*

Instead of using a three-way join insert-select, use denormalization to reduce this to a two-way join, e.g. changing the *sku_def*'s schema to this:

sku_def(sku_id, hotel_id, room_type_id, desc_id). Primary key and indexes don't change. AppSleuth's analysis output is in Figure 16.

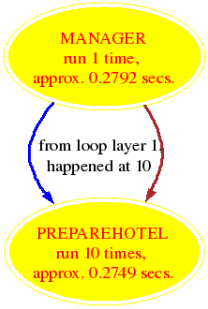

Figure 15. AppSleuth's output after Tuning 2.

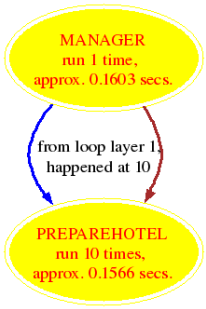

Figure 16. AppSleuth's output after Tuning 3.

5.2.4 *Tuning 4, 5 … – other possible tuning methods.*

Users will be offered other choices including materialized or unmaterialized views (Figure 17). The demo should be fun because most conference participants work on the internals of database management systems but don't face real world tuning problems with them.

Figure 17. AppSleuth's output after Tuning 4 with an unmaterialized view.

If the user ever chooses unmaterialized views to tune, then tests will be conducted for the query performance over the view against over base tables or materialized views to judge the overall performance improvement as the following table Table 1 shows.

Table 1. Comparison of query performance for tuning using an unmaterialized view.

Query performance comparison	Elapsed time for cold buffer	Elapsed time for warm buffer
Tuning 4	0.12 seconds (avg.)	0.08 seconds (avg)
Other tuning methods	0.08 seconds (avg.)	0.02 seconds (avg)

6. DEMO RESULTS

The visitors to our demonstration will get a hands-on feel for using a tuning tool that goes beyond index or statement tuning to application tuning. They will have a chance to test their skill and to evaluate the results of their decisions. Designers will understand the importance of implementation tools. Query processing researchers will see that the boundaries of query processing extend to the multi-statement level.

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

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

[8] 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

[9] 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