US 20050177557A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2005/0177557 A1**

Ziauddin et al. (43) **Pub. Date: Aug. 11, 2005**

(54) **AUTOMATIC PREVENTION OF RUN-AWAY QUERY EXECUTION**

(75) Inventors: **Mohamed Ziauddin**, Pleasanton, CA (US); **Benoit Dageville**, Foster City, CA (US); **Khaled Yagoub**, San Mateo, CA (US); **Mohamed Zait**, San Jose, CA (US)

Correspondence Address:
BINGHAM, MCCUTCHEN LLP
THREE EMBARCADERO CENTER
18 FLOOR
SAN FRANCISCO, CA 94111-4067 (US)

(73) Assignee: **ORACLE INTERNATIONAL CORPORATION**, REDWOOD SHORES, CA

(21) Appl. No.: **10/936,779**

(22) Filed: **Sep. 7, 2004**

Related U.S. Application Data

(60) Provisional application No. 60/500,490, filed on Sep. 6, 2003.

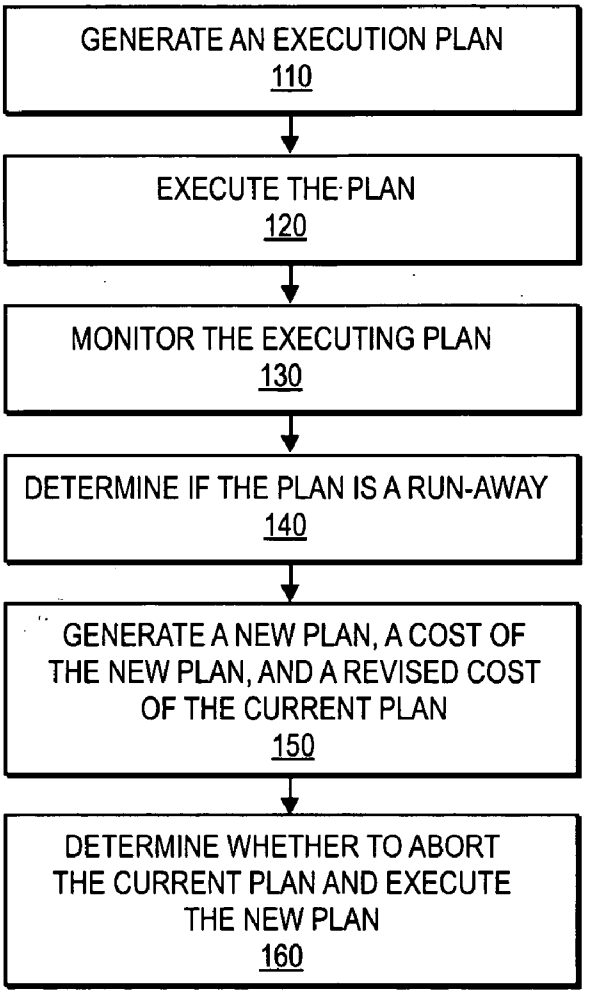
Publication Classification

(51) **Int. Cl.**[7] **G06F 7/00**

(52) **U.S. Cl.** **707/3**

(57) **ABSTRACT**

A run-away query execution is automatically identified by a background process that periodically looks at each of the currently executing queries and compares the current execution time with the execution time estimated by the optimizer. Each query execution having a negative execution time difference can be automatically identified as a run-away query execution. The query execution plans that result in run-away executions can then be automatically tuned to produce more efficient execution plans.



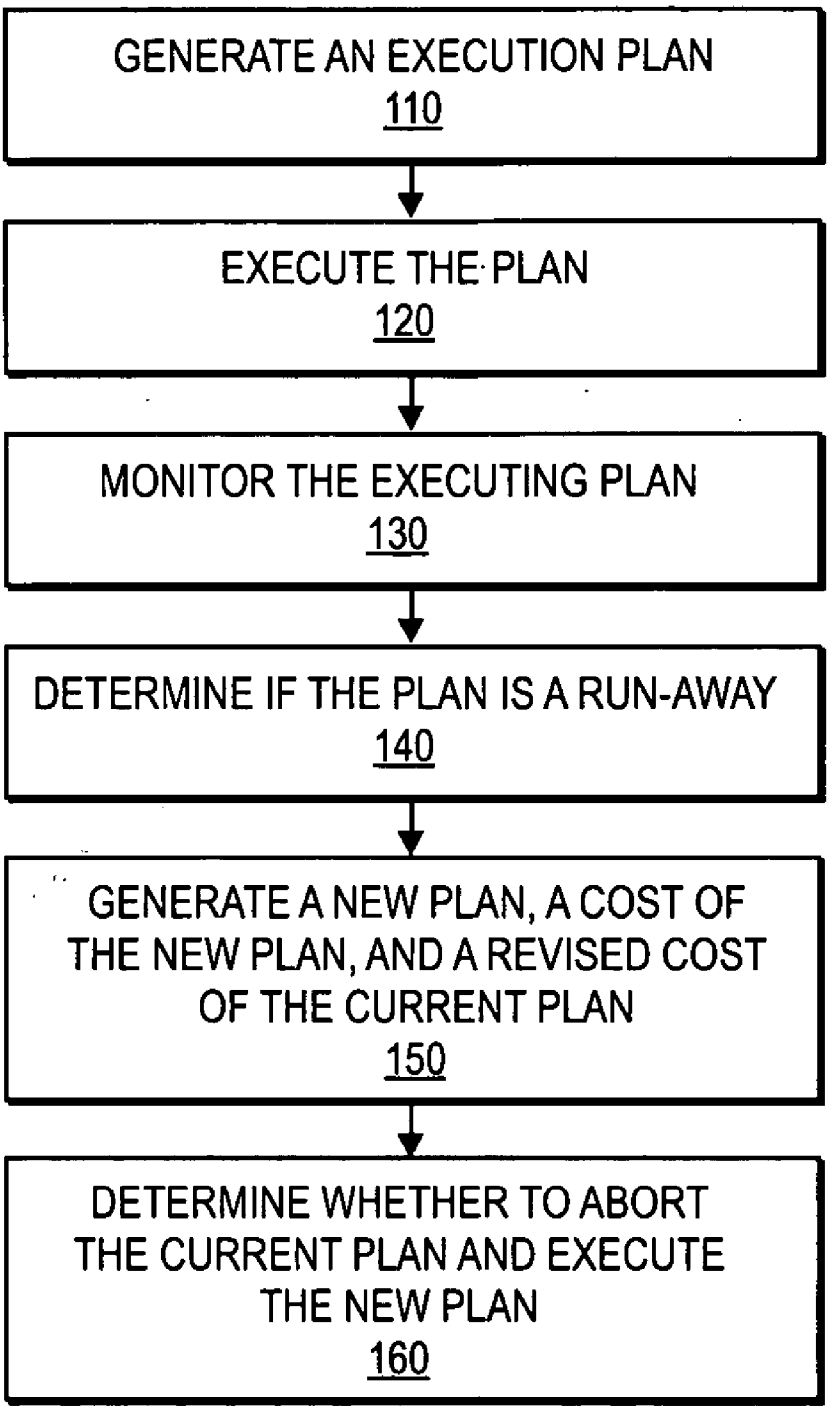

GENERATE AN EXECUTION PLAN
110
EXECUTE THE PLAN
120
MONITOR THE EXECUTING PLAN
130
DETERMINE IF THE PLAN IS A RUN-AWAY
140
GENERATE A NEW PLAN, A COST OF THE NEW PLAN, AND A REVISED COST OF THE CURRENT PLAN
150
DETERMINE WHETHER TO ABORT THE CURRENT PLAN AND EXECUTE THE NEW PLAN
160

FIG. 1

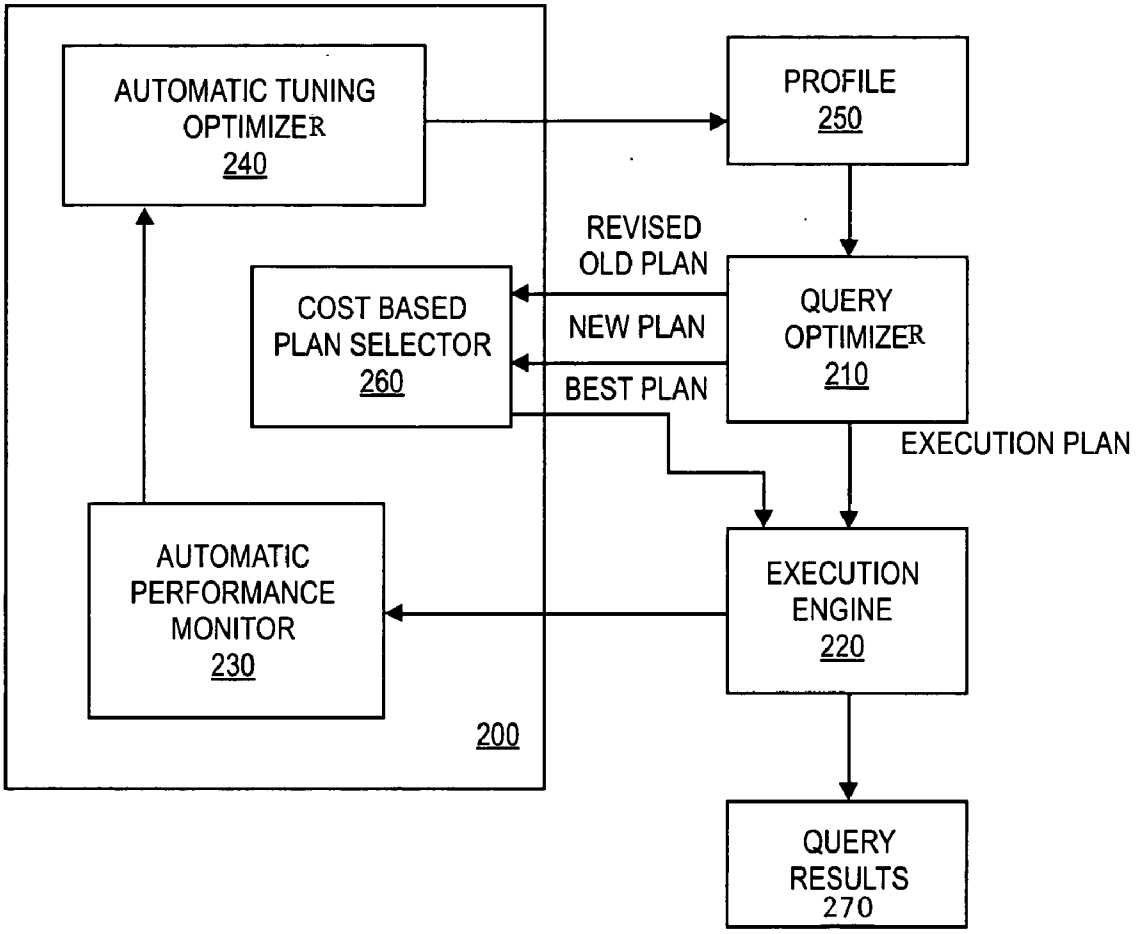

AUTOMATIC TUNING OPTIMIZER 240
PROFILE 250
REVISED OLD PLAN
COST BASED PLAN SELECTOR 260
NEW PLAN
QUERY OPTIMIZER 210
BEST PLAN
EXECUTION PLAN
AUTOMATIC PERFORMANCE MONITOR 230
EXECUTION ENGINE 220
200
QUERY RESULTS 270

FIG. 2

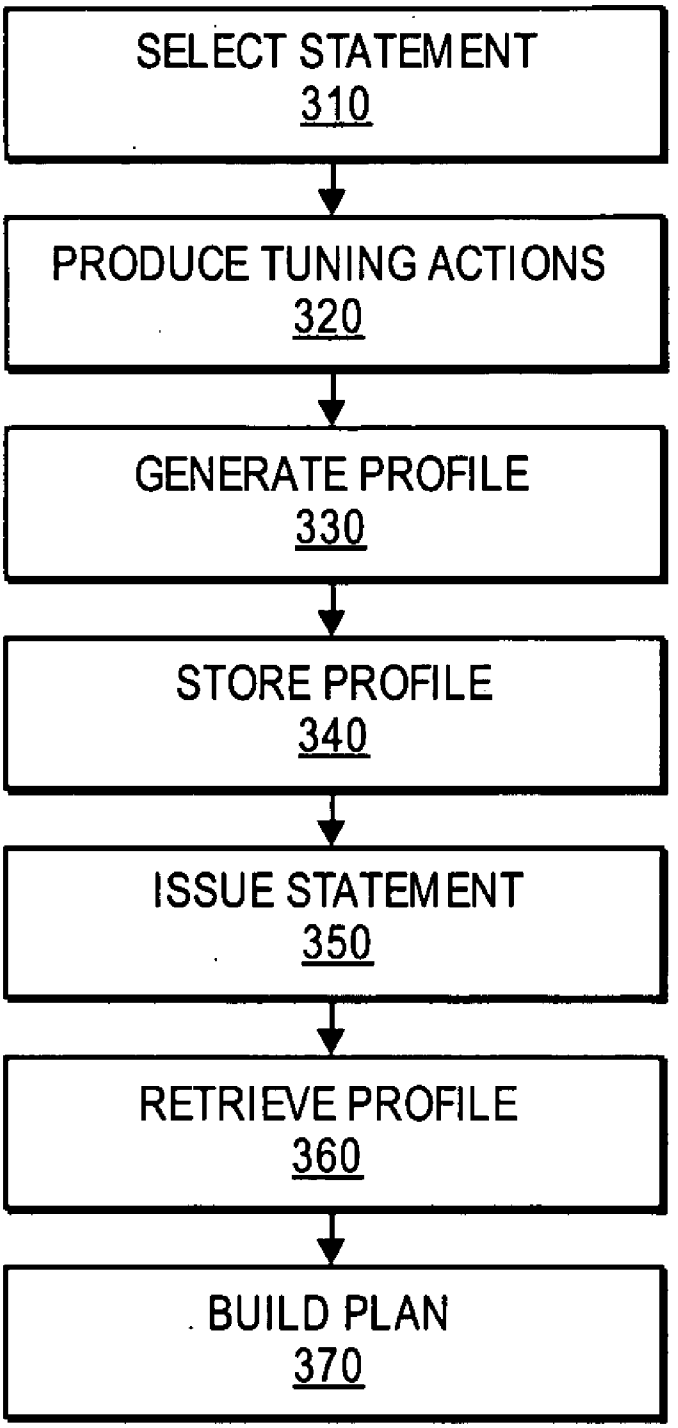

SELECT STATEMENT
310
PRODUCE TUNING ACTIONS
320
GENERATE PROFILE
330
STORE PROFILE
340
ISSUE STATEMENT
350
RETRIEVE PROFILE
360
BUILD PLAN
370

FIG. 3

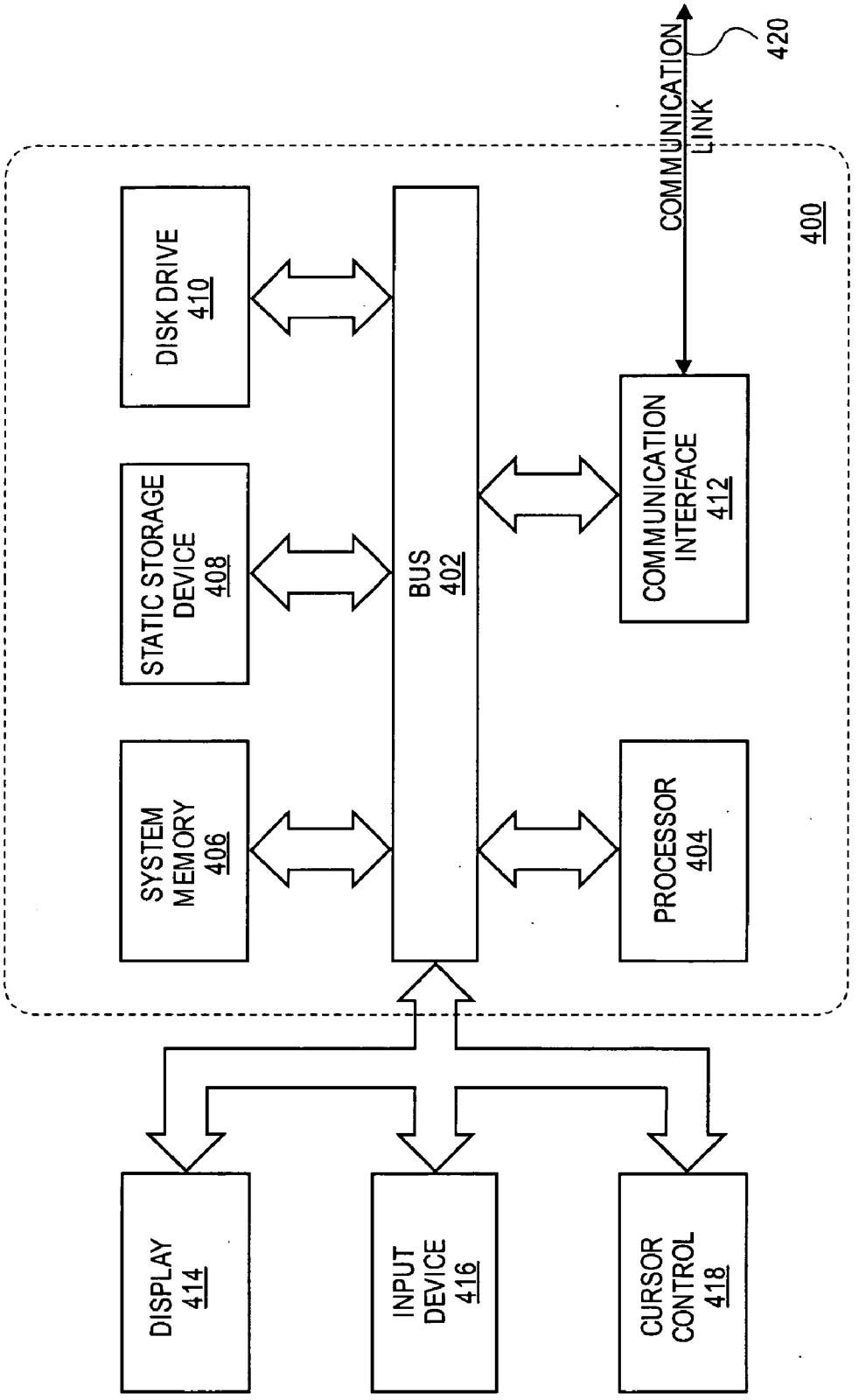

COMMUNICATION LINK
420
400
DISK DRIVE 410
STATIC STORAGE DEVICE 408
SYSTEM MEMORY 406
BUS 402
COMMUNICATION INTERFACE 412
PROCESSOR 404
DISPLAY 414
INPUT DEVICE 416
CURSOR CONTROL 418

FIG. 4

AUTOMATIC PREVENTION OF RUN-AWAY QUERY EXECUTION

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. Provisional Application No. 60/500,490, filed Sep. 6, 2003, which is incorporated herein by reference in its entirety. This application is related to co-pending applications "SQL TUNING SETS," Attorney Docket No. OI7036272001; "AUTO-TUNING SQL STATEMENTS," Attorney Docket No. OI7037042001; "SQL PROFILE," Attorney Docket No. OI7037052001; "GLOBAL HINTS," Attorney Docket No. OI7037062001; "SQL TUNING BASE," Attorney Docket No. OI7037072001; "AUTOMATIC LEARNING OPTIMIZER," Attorney Docket No. OI7037082001; "METHOD FOR INDEX TUNING OF A SQL STATEMENT, AND INDEX MERGING FOR A MULTI-STATEMENT SQL WORKLOAD, USING A COST-BASED RELATIONAL QUERY OPTIMIZER," Attorney Docket No. OI7037102001; "SQL STRUCTURE ANALYZER," Attorney Docket No. OI7037112001; "HIGH-LOAD SQL DRIVEN STATISTICS COLLECTION," Attorney Docket No. OI7037122001; "AUTOMATIC SQL TUNING ADVISOR," Attorney Docket No. OI7037132001, all of which are filed Sep. 7, 2004 and are incorporated herein by reference in their entirety.

FIELD OF THE INVENTION

[0002] This invention is related to the field of electronic database management.

BACKGROUND

[0003] The generation of optimal execution plans is critical to the performance of applications. For example, a single SQL statement with very poor performance can bring an application down to its knees. Sometimes a poorly performing SQL statement is due to user error, such as a blind query issued with without filtering conditions that would have reduced the amount of data processed. Other times the SQL statement is well formed, but the associated execution plan that is generated by the optimizer is suboptimal.

[0004] The suboptimal plan results in a run-away execution of the query. In other words, the plan, when executed, causes a SQL statement to run for a long time with enormous use of system resources. The problem of fixing the execution plan is usually addressed through a manual SQL tuning process. This process involves a tuning expert analyzing the SQL statement as well as its associated execution plan, then determining that the problem lies in the execution plan and not in the way the SQL statement is used (for example, an accidental use of a Cartesian join by not joining one of the tables to any of the other tables in the query). The manual SQL analysis process is a time-consuming task.

[0005] After this analysis, the expert performs a manual SQL tuning process to influence the optimizer to generate a good plan. This involves the tuning expert adding one or more tuning actions to the statement. These actions may be to identify and collect missing statistics and refresh stale statistics, change the value of some configuration parameter which directly affects the plan generation methodology of the optimizer, add one or more hints to the SQL statement which will give the directives to the optimizer in coming up with the right plan, create a new access path (such as an index) or modify an existing one to help avoid large scans of data. The manual SQL tuning process is also a time-consuming and complex task.

[0006] Many vendors have addressed the problem of run-away query execution by using a query governor control mechanism. The query governor can be either reactive or proactive. In a reactive mode, an execution-time threshold is set to abort any query whose cumulative execution time exceeds to threshold. In a proactive mode, an optimized-estimated-time threshold is set which is applied to the time optimizer has estimated for the query to run. Any query having an estimated run-time that exceeds the threshold is never run. With either of these methods, there is no attempt made to look at the root cause of the problem.

[0007] Some vendors have used the idea of setting execution-time thresholds at various places in the execution plan to detect a case of run-away query execution. When a threshold is crossed during query execution, the run is aborted and the query sent back to the optimizer for re-optimization. But this method suffers from two drawbacks: setting of the thresholds and monitoring them at runtime incurs overhead, which can be significant and undesirable especially for light-weight queries, and the method of aborting a run and re-optimizing a query can be quite disruptive, especially if the run is aborted right before it was about to complete.

SUMMARY

[0008] A run-away query execution is automatically identified by a background process that periodically looks at each of the currently executing queries and compares the current execution time with the execution time estimated by the optimizer. Each query execution having a negative execution time difference can be automatically identified as a run-away query execution. The query execution plans that result in run-away executions can then be automatically tuned to produce more efficient execution plans.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] **FIG. 1** shows an example of a method for performing automatic prevention of run-away query execution.

[0010] **FIG. 2** shows example of a system for automatic prevention of run-away queries.

[0011] **FIG. 3** represents an illustration of the prevention process.

[0012] **FIG. 4** is a block diagram of a computer system suitable for implementing an embodiment of automatic run-away query prevention.

DETAILED DESCRIPTION

[0013] Overview

[0014] The embodiments of the invention are described using the term "SQL", however, the invention is not limited to just this exact database query language, and indeed may be used in conjunction with other database query languages and constructs.

[0015] The automatic performance monitoring of query executions identifies run-away query executions, then performs a re-optimization for the corresponding execution plans in a background process. The automatic prevention of run-away query executions may abort a current execution of a query run if the automatic process has produced an improved plan in the background, and further, has determined a benefit to aborting the current execution and performing an execution of the new plan.

[0016] This process can be implemented by an automatic SQL tuning optimizer and a SQL tuning base. In one embodiment, the run-away query execution is identified by a background process that periodically looks at each of the currently executing queries and compares the time spent in executing it so far (current-time) vs. the time the optimizer has estimated the execution to take (estimate-time). The top N queries with the largest negative difference (estimate-time−current-time) may be selected as run-away executions. An alternate method of identifying run-away query executions can be based on the current-time, that is, the process can select the top N queries with the longest current execution time as run-away query executions.

[0017] The automatic tuning optimizer (ATO), in a background process, then optimizes the execution plan for each query having a run-away execution by performing various analyses of the corresponding SQL statement, such as automatic identification and correction of inaccurate statistics, cardinality estimates, and cost estimates related to the statement, for example. If the execution plan built by the ATO is different from the one that is currently executing, the ATO can estimate how much more time the current plan execution is going to take to complete (remaining-time), as well as estimate how much time the new plan will take to execute (new-time). If the new-time is less than the remaining-time then the current plan run may be aborted and replaced with the new plan.

[0018] Since the ATO uses validated estimates of the cost, selectivity and cardinality, it can compute the total execution time of the new plan much more accurately. Similarly, it can regenerate the original run-away plan that is currently executing with validated estimates to compute its remaining execution time. Because the identification of run-away query executions, and the automatic generation of improved plans for the corresponding queries are performed by the ATO in the background, this automatic process is transparent to the database user.

[0019] Automatic Identification and Tuning of Run-Away Execution Plans

[0020] The automatic prevention of run-away query executions is performed by a process as shown in **FIG. 1**. A query execution plan is generated for an SQL statement by a query optimizer, **110**. The execution plan is executed by an execution engine, **120**. The executing plan is monitored, **130**, to detect that the plan is a run-away, or sub-optimal, execution plan. For example, in addition to generating the execution plan, the query optimizer can also estimate the amount of time that the execution engine will spend executing the plan. If the actual execution time exceeds the estimated time, then the plan is potentially a run-away plan. Alternatively, the amount of execution time can be compared to a threshold time, such as two hours for example. If the execution plan is still running after two hours, then the plan may be a run-away plan. The potential run-away plan is further analyzed to determine if the plan actually is a run-away plan, **140**. For example, a profile for the SQL statement can be generated to correct or adjust errors in statistics and estimates associated with the plan, and to determine appropriate parameter settings for the statement.

[0021] Then, a new execution plan, along with a time estimate for executing the new plan, can be generated using the profile. Also, a revised estimate of the execution time of the run-away execution plan is generated using the profile, **150**. If the new plan can be executed faster than the currently executing run-away plan, then the current plan is identified as a run-away plan. A second comparison of execution times is performed to determine whether to abort the current execution of the run-away plan and executing the new plan, or to allow the run-away plan to run to completion, **160**. If the remaining execution time of the run-away plan is less than the execution time of the new plan, then the current plan is allowed to finish. If the execution time of the new plan is less than the remaining execution time of the currently executing run-away plan, then the run-away plan is aborted and the new plan is executed.

[0022] Automatic Prevention Architecture

[0023] An example of a system **200** for automatic prevention of run-away queries is shown in **FIG. 2**. A query optimizer, **210**, receives a SQL statement, and generates an execution plan for the statement, which is executed by execution engine **220**. An automatic performance monitor **230** identifies a potential run-away execution plan by observing the elapsed execution time of the plan, for example. The corresponding SQL statement is then input into an automatic tuning optimizer **240**, which generates a profile **250** for the SQL statement. The profile can contain information related to missing or stale statistics. The profile can also include one or more tuning actions that can be used by an optimizer to generate an execution plan for the statement. The profile and the statement are received by the query optimizer **210**, which generates a new execution plan, along with an estimated amount of time for executing the plan, based on the profile. The query optimizer also revises the estimated amount of time for executing the current plan using the profile. The time estimates are analyzed by a cost based plan selector, **260**, which can determine that the current plan is a run-away plan if the corresponding execution time estimate is longer than that of the new plan. The plan selector **260** can also cause the execution engine **220** to abort the run-away plan and execute the new plan if the remaining amount of time to execute the run-away plan is more than the amount of time to execute the new plan. Otherwise, the execution engine continues to execute the current plan. In either case, query results **270** are returned by the system.

[0024] SQL Profiling

[0025] A profiling process is performed by the automatic tuning optimizer to produce a set of tuning actions in generating an execution plan for a SQL statement. The profiling process verifies that statistics are not missing or stale, validates the estimates made by the query optimizer for intermediate results, and determines the correct optimizer settings. Tuning actions are created based on the results of the profiling process, to provide missing statistics for an object, validate intermediate results estimate, and

select the best setting for optimizer parameters. Then, the Automatic Tuning Optimizer builds a SQL Profile for these tuning actions.

[0026] The statistics analysis verifies that statistics are not missing or stale. The query optimizer logs the types of statistics that are actually used during the plan generation process, in preparation for the verification process. For example, when a SQL statement contains an equality predicate, it logs the column number of distinct values, whereas for a range predicate it logs the minimum and maximum column values information. Once the logging of used statistics is complete, the query optimizer checks if each of these statistics is available on the associated query object (i.e. table, index or materialized view). If the statistic is available then it verifies whether the statistic is up-to-date. To verify the accuracy of a statistic, it samples data from the corresponding query object and compares it to the statistic. If a statistic is found to be missing, the query optimizer will generate auxiliary information to supply the missing statistic. If a statistic is available but stale, it will generate auxiliary information to compensate for staleness.

[0027] One feature of a cost-based query optimizer is its ability to derive the size of intermediate results. For example, the optimizer estimates the number of rows from applying table filters when deciding which join algorithm to pick. One factor that causes the optimizer to generate a sub-optimal plan is wrong estimate of intermediate result sizes. Wrong estimates can be caused by a combination of the following factors: The predicate (filter or join) is too complex to use standard statistical methods to derive the number of rows (e.g., the columns are compared thru a complex expression like (a*b)/c=10), The data distribution of the column used in the predicate is skewed, and there is no histogram, leading the optimizer to assume a uniform data distribution, or The data in column values is correlated but the optimizer is not aware of it, causing the optimizer to assume data independence. During SQL Profiling, the Automatic Tuning Optimizer validates the estimates made by the query optimizer, and compensates for missing information or wrong estimates. The validation process may involve running part of the query on a sample of the input data.

[0028] The Automatic Tuning Optimizer uses the past execution history of a SQL statement to determine the correct optimizer settings. For example, if the execution history shows that a SQL statement is only partially executed in the majority of times then the appropriate setting will be to optimize it for first n rows, where n is derived from the execution history. This constitutes a customized parameter setting for the SQL statement. (Note that past execution statistics are available in the Automatic Workload Repository (AWR) presented later).

[0029] The tuning information produced from the statistics, estimates, and settings analyses is stored in a SQL Profile. Once a SQL Profile is created, it is used in conjunction with the existing statistics by the compiler to produce a well-tuned plan for the corresponding SQL statement. **FIG. 3** shows the process flow of the creation and use of a SQL Profile. The process can have two separate phases: an Automatic SQL Tuning phase, and a regular optimization phase. During the Automatic SQL Tuning phase, a SQL statement with a run-away execution **310** is selected as an input to the SQL Tuning Advisor, which invokes the Automatic Tuning Optimizer to generate tuning actions, **320**. The Automatic Tuning Optimizer generates a SQL Profile, along with other recommendations, **330**. After a SQL Profile is built, it is stored in the data dictionary, once it is accepted by the user, **340**. Later, during the regular optimization phase, a user issues the same SQL statement, **350**. The query optimizer finds the matching SQL profiles from the data dictionary, **360**, and uses the SQL profile information to build a well-tuned execution plan, **370**. The use of SQL Profiles is completely transparent to the user. The creation and use of a SQL Profile doesn't require changes to the application source code. Therefore, SQL profiling provides a way to tune SQL statements issued from packaged applications where the users have no access to or control over the application source code.

[0030] The automatic prevention of run-away queries can identify a plan that is a potential run-away plan. The process analyzes the SQL statement for the plan to determine if the potential run-away plan is caused by a bad plan. For example, the process can create a profile for the statement, use the profile to generate a new plan, and compare the new plan to the old plan to determine if the old plan is a run-away plan. The process can also use the profile to determine whether the run-away plan is close to finishing, and therefore should run to completion, or if the run-away plan should be aborted and the new plan should be executed in its place. Thus, the automatic prevention of run-away query executions eliminates the overhead incurred by conventional methods, such as monitoring of thresholds and aborting a run just before it finishes.

[0031] **FIG. 4** is a block diagram of a computer system **400** suitable for implementing an embodiment of automatic prevention of run-away query execution. Computer system **400** includes a bus **402** or other communication mechanism for communicating information, which interconnects subsystems and devices, such as processor **404**, system memory **406** (e.g., RAM), static storage device **408** (e.g., ROM), disk drive **410** (e.g., magnetic or optical), communication interface **412** (e.g., modem or ethernet card), display **414** (e.g., CRT or LCD), input device **416** (e.g., keyboard), and cursor control **418** (e.g., mouse or trackball).

[0032] According to one embodiment of the invention, computer system **400** performs specific operations by processor **404** executing one or more sequences of one or more instructions contained in system memory **406**. Such instructions may be read into system memory **406** from another computer readable medium, such as static storage device **408** or disk drive **410**. In alternative embodiments, hardwired circuitry may be used in place of or in combination with software instructions to implement the invention.

[0033] The term "computer readable medium" as used herein refers to any medium that participates in providing instructions to processor **404** for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as disk drive **410**. Volatile media includes dynamic memory, such as system memory **406**. Transmission media includes coaxial cables, copper wire, and fiber optics, including wires that comprise bus **402**. Transmission media can also take the form of acoustic or light waves, such as those generated during radio wave and infrared data communications.

[0034] Common forms of computer readable media includes, for example, floppy disk, flexible disk, hard disk, magnetic tape, any other magnetic medium, CD-ROM, any other optical medium, punch cards, paper tape, any other physical medium with patterns of holes, RAM, PROM, EPROM, FLASH-EPROM, any other memory chip or cartridge, carrier wave, or any other medium from which a computer can read.

[0035] In an embodiment of the invention, execution of the sequences of instructions to practice the invention is performed by a single computer system **400**. According to other embodiments of the invention, two or more computer systems **400** coupled by communication link **420** (e.g., LAN, PTSN, or wireless network) may perform the sequence of instructions to practice the invention in coordination with one another. Computer system **400** may transmit and receive messages, data, and instructions, including program, i.e., application code, through communication link **420** and communication interface **412**. Received program code may be executed by processor **404** as it is received, and/or stored in disk drive **410**, or other non-volatile storage for later execution.

[0036] In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than restrictive sense.

We claim:

1. A method comprising:

automatically identifying an executing query as having a run-away execution plan; and

automatically replacing the run-away execution plan with a tuned execution plan.

2. The method of claim 1, wherein automatically replacing comprises:

automatically generating tuning actions for the query; and

placing the tuning actions in a profile.

3. The method of claim 2, further comprising:

using the profile to revise an execution time of the run-away execution plan.

4. The method of claim 3, further comprising:

receiving the query at an optimizer;

retrieving the profile for the query from the tuning base to the optimizer; and

generating, at the optimizer the tuned execution plan for the query with the profile.

5. The method of claim 1, further comprising:

comparing an execution time of the tuned execution plan with a remaining execution time of the run-away execution plan;

determining that the execution time of the tuned execution plan is less than the remaining execution time of the run-away execution plan; and

executing the tuned execution plan.

6. The method of claim 1, wherein the query is a SQL statement.

7. An apparatus comprising:

means for automatically identifying a query with a run-away execution plan; and

means for automatically replacing the run-away query plan with a tuned execution plan.

8. The apparatus of claim 7, wherein said means for automatically replacing comprises:

means for automatically generating tuning actions for the query; and

means for placing the tuning actions in a profile.

9. The apparatus of claim 8, further comprising:

means for persistently storing the profile in a tuning base.

10. The apparatus of claim 9, further comprising:

means for receiving the query at a compiler;

means for retrieving the profile for the query from the tuning base; and

means for generating the tuned execution plan for the query with the profile.

11. The apparatus of claim 7, wherein said means for automatically identifying comprises:

means for comparing an execution time of the tuned execution plan with an estimated remaining execution time of the run-away query plan; and

means for determining that the execution time of the tuned execution plan is less than the estimated remaining execution time of the run-away query plan.

12. The apparatus of claim 7, wherein the query is a SQL statement.

13. A computer readable medium storing a computer program of instructions which, when executed by a processing system, cause the system to perform a method comprising:

automatically identifying a query with a run-away execution plan; and

automatically replacing the run-away execution plan with a tuned execution plan.

14. The medium of claim 13, wherein automatically replacing comprises:

automatically generating tuning actions for the query; and

placing the tuning actions in a profile.

15. The medium of claim 14, further comprising:

persistently storing the profile in a tuning base.

16. The medium of claim 15, further comprising:

receiving the query at a compiler;

retrieving the profile for the query from the tuning base; and

generating the tuned execution plan for the query with the profile.

17. The medium of claim 13, wherein the query is a SQL statement.

* * * * *