Otto-von-Guericke-Universität Magdeburg

Seminar

Self-Tuning Databases

Goal-Oriented Buffer Management Revisited

André Riedel

21 January 2004

Contents

Introduction

- each workload class may have its own performance goal
- different response times, e.g.
  - 1 second for transaction
  - 1 minute for decision support queries
  - “best effort“ for data mining queries
- today manually tuning with various low level “knobs“ in the DBMS
- ideally DBMS with per-class performance goals as input should adjust its own low-level knobs to achieve goal

Goal-Oriented Basics

- memory allocation is most important knob because it determines the amount of disk and bandwidth consumed
- when all other knobs remain fixed, the goal is:
  - for each class with an average response time goal, find such a memory allocation that its observed response time is as close as possible to its goal
  - while at the same time maximizing the amount of memory available for the no-goal class

- real world DBMS and workloads, accurately predicting the disk buffer allocation for a goal is extremely difficult
- the general approach: feedback coupled with “best guess“ estimation
- observe actual response times and compare with response time goal then adjust knobs
- process of observing, estimating and adjusting is repeated continuously at regular intervals
- length of intervals is predefined number of transaction completions
- should have good balance between responsiveness and statistical stability

Criteria for Success

- class meets its response time goal is not the only criteria for judging algorithm

  Accuracy - how close is average response time to goal

  Responsiveness - number of knob adjustments

  Stability - variance in the response times

  Overhead - reduce of system efficiency

  Robustness - wide range of workloads

  Practically - don't make unrealistic assumptions
- will normally be in conflict (e.g. stability versus responsiveness)
- algorithm must find careful balance between this criteria

Previous Approaches

- goal-oriented buffer allocation algorithms can be described abstractly in terms of three components:

  response time estimator estimates response time as function of buffer hit rate

  hit rate estimator estimates buffer hit rate as a function of memory allocation

  buffer allocation mechanism is used to divide up memory between the competing workload classes

- response time estimator ⇒ hit rate estimator ⇒ buffer allocation mechanism
- these steps are repeated continuously for each class to come closer to the response time goals

Dynamic Tuning

- uses simple linear estimate to predict buffer request response times

$$R^{est} = (1.0 - HIT^{est}(M)) \times D$$

- $HIT^{est}(M)$ is the estimated hit rate for the class that will result from a memory allocation M
- D is the average time required for moving a page from disk to memory

- to estimate hit rate as function of memory the Belady hit rate function is used

$$1 - a/M^b$$

- M is memory allocation
- constants a and b are specific to a particular combination of workload and buffer page replacement policy
- to compute a and b observe the hit rate of the two most recent memory allocations
- solve the two simultaneous equations and get specific a and b
- use the inverse of the Belady equation to estimate the memory
- entire buffer pool is partitioned into separate pools for each class, managed by completely autonomous buffer managers

Dynamic Tuning Issues

- is not a good “fit“ for any particular function
- real hit rate curves have a wide range of shapes and are difficult to capture accurately with a single analytical model

Fragment Fencing

- makes the simplifying assumption that response time and buffer miss rate are directly proportional

$$HIT^{target} = 1.0 - (M^{obsv} * (R^{goal}/R^{obsv}))$$

- R^{obsv} is the observed response time, R^{goal} is the response time goal
- M^{obsv} is the observed miss rate that occurs with the observed response time
- Fragment Fencing estimates hit rate function for each fragment of the database that is referenced by the class
- a fragment is defined as all of the pages within a relatively uniform reference unit, e.g. a single relation or a single level of a tree-structured index
- a uniform reference probability is assumed across the pages of a fragment

- the goal of Fragment Fencing is to determine, for each fragment, the minimum number of pages that must be memory resident
- these minimums are called *target residencies*
- when increasing the hit rate fragments of class are sorted
- increase the target residencies in order
- to enforce the determined target residency the DBMS's buffer replacement policy is changed

Fragment Fencing Issues

- references within fragments are not uniform
  - can easily tested by comparing the estimated hit rate to the actual hit rate
  - not clear what the fragment's memory allocation should be
  - average per-page reference frequency and sorting is not meaningful
- “passive“ memory allocation
  - underlying replacement policy is unaware of which frames are fenced
  - policy wastes time for inspecting good frame candidates only to be overruled by Fragment Fencing

Class Fencing

- also assumes that miss rate and response time are proportional
- uses a more general hit rate prediction technique - *hit rate concavity*
- allows for data sharing between classes
- compromise between the rigid partitions of Dynamic Tuning and the passive fences of Fragment Fencing

The Hit Rate Concavity Assumption

Concavity Theorem:
Regardless of the database reference pattern, hit rate as a function of buffer memory allocation is a concave function under an optimal replacement policy.

- the slope of the hit rate curve never increases as more memory is added
- an optimal buffer replacement policy always chooses the least valuable page to replace
- in practice optimal replacement policies are not realizable
- but industrial-strength DBMS replacement strategies are “optimal enough“

- a DBMS should make fewer page replacement mistakes than an operating system, because:
  - knowledge of future page reference behavior
  - presence of indexes
- concavity implies that there are no “knees“ in an optimal hit rate function
- empirical study showed no knee in commercial DBMS

⇒ hit rate concavity holds for the most commonly occurring workloads running on a typical commercial DBMS

Estimating Hit Rates Using the Concavity Assumption

- only the last two hit rate observations are needed
- straight line approximation always predicts a conservative lower bound for its memory allocation
- can aggressively allocate memory in large increments

Class Fencing's Memory Allocation Mechanism

- a single fence is built to protect all of the pages referenced by a class
- each class has local buffer manager
- a global buffer manager is used for no-goal classes
- no overhead because the global buffer manager contains no fenced frames
- single buffer frame table and associated disk-page-to-buffer-frame mapping table is shared by all buffer managers

- each class C has a limit, poolSize[C] the maximum number of buffer frames that can be managed by class C's local buffer manager
- global buffer manager has poolSize[GLOBAL]
- DBMS buffer pool memory = local and global pool sizes
- local buffer manager “steals“ frames from global buffer manager
- when poolSize limit exceeds replacement policy is called
- no-goal frames are handled by the global buffer manager

Experiments and Results

- with different workloads and goals
- Class Fencing is stable and accurate
- is not restricted to allocate memory in small chunks ⇒ is very responsive
- Responsiveness is key feature, uses very few knob turns
- eliminates primary overhead of Fragment Fencing
- is fairly robust because it applies to a wide range of workloads

Conclusion

- Dynamic Tuning and Fragment Fencing are solutions for goal-oriented DBMS buffer management
- Class Fencing overcomes limitations of these prior solutions
- uses other new DBMS techniques for new assumptions

References:

K. P. Brown, M. J. Carey and M. Livny, *Goal-Oriented Buffer Management Revisited*, Proceedings of the ACM SIGMOD, Jun. 1996, pages 353–364.