# SQL Memory Management in Oracle9i

Benoît Dageville
Oracle Corporation
500 Oracle Parway
Redwood Shores, CA 94065
U.S.A
Benoit.Dageville@oracle.com

Mohamed Zait
Oracle Corporation
500 Oracle Parway
Redwood Shores, CA 94065
U.S.A
Mohamed.Zait@oracle.com

## Abstract

Complex database queries require the use of memory-intensive operators like sort and hash-join. Those operators need memory, also referred to as SQL memory, to process their input data. For example, a sort operator uses a work area to perform the in-memory sort of a set of rows. The amount of memory allocated by these operators greatly affects their performance. However, there is only a finite amount of memory available in the system, shared by all concurrent operators. The challenge for database systems is to design a fair and efficient strategy to manage this memory.

Commercial database systems rely on database administrators (DBA) to supply an optimal setting for configuration parameters that are internally used to decide how much memory to allocate to a given database operator. However, database systems continue to be deployed in new areas, e.g, e-commerce, and the database applications are increasingly complex, e.g, to provide more functionality, and support more users. One important consequence is that the application workload is very hard, if not impossible, to predict. So, expecting a DBA to find an optimal value for memory configuration parameters is not realistic. The values can only be optimal for a limited period of time while the workload is within the assumed range.

Ideally, the optimal value should adapt in response to variations in the application workload. Several research projects addressed this problem in the past, but very few commercial systems proposed a comprehensive solution to managing memory used by SQL operators in a database application with a variable workload.

This paper presents a new model used in Oracle9i to manage memory for database operators. This approach is automatic, adaptive and robust. We will present the architecture of the memory manager, the internal algorithms, and a performance study showing its superiority.

**Proceedings of the 28th VLDB Conference,**
**Hong Kong, China, 2002**

## 1. Introduction

Queries in On-Line Analytical Processing (OLAP) applications and Decision-Support Systems (DSS) tend to be very complex: join many tables, and process large amounts of data. They make heavy use of SQL operators such as sort and hash join. The sort is used not only to produce the input rows in sorted order but also as the basis in other operators, e.g, grouping, duplicate elimination, rollup, analytic functions, and index creation. In the rest of the paper, the term “SQL operators” is used to exclusively refer to memory-intensive operators, e.g. nestedloops join is excluded.

Those operators need memory space to process their input data. For example, a sort operator uses a work area to perform the in-memory sort of a set of rows. Similarly, a hash-join operator uses a work area to build a hash table on its left input (called build input). Generally, larger work areas can significantly improve the performance of a particular operator. Ideally, the size of a work area is big enough such that it can accommodate the input data and auxiliary memory structures allocated by the operator. This is referred to as the *cache* size of a work area. When the size of the work area is smaller than cache, the response time increases since an extra pass is performed over all or part of the input data. This is referred to as the *one-pass* size of the work area. When the work area size is less than the one-pass threshold, multiple passes over the input data are needed, causing dramatic increase of the operator response time. This is referred to as the *multi-pass* size of the work area. For example, a sort operation which needs to sort 10GB of data needs a little more than 10GB of memory to run in cache mode and at least 40MB to run in one-pass mode. It will run in multi-pass mode with less than 40MB.

***Figure 1: Effect of Memory on Sort Performance***

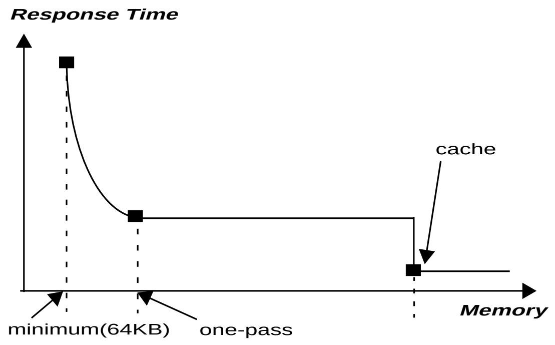


***Figure 2: Effect of Memory on HashJoin Performance***

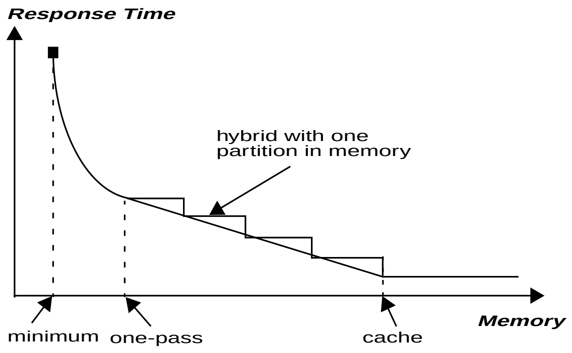


Figure 1 (sort) and Figure 2 (hash join) show the response time of the sort and hash-join operators as a function of the memory allocated by the operators. We are interested in the *one-pass* and *cache* points on both curves. The *one-pass* point is the start of the area where the operator runs in one-pass mode, and the *cache* point corresponds to case when the work area size is equal to the cache size. The sort curve is flat between these two points because a sort operator doesn't benefit from additional memory if it cannot use the cache size. The hash-join curve decreases in a step-like shape between the one-pass and cache points. Each step corresponds to an extra build partition that can be kept in memory. Contrary to the sort, the hash join can benefit from additional memory between the one-pass and cache points.

In On-Line Transaction Processing (OLTP) systems, the size of input data to SQL operators is generally small, thus, they run in cache mode most of the time. This is not the case in OLAP or DSS, where the input data is very large. Thus, it is important to *properly* size their work area in order to obtain good response time for the queries, maximize the usage of the hardware resources, and be fair in allocating them to competing operators.

In most commercial systems the burden has been put on the DBA to provide an optimal setting for configuration parameters that are internally used to decide how much memory to allocate to a given database operator. This is a challenging task for the DBA because it's difficult to estimate memory utilization for an operator work area, for a query, and the database system. The operator work area size depends on the size of the operator input data. The memory utilization of a query depends on the operators scheduling and the number of parallel processes assigned to the query, while the memory utilization in a database system depends on the current workload. Most probably, the memory will either end up being *under-utilized* (if the settings are based on pessimistic assumptions about the workload) or *over-allocated* (if the DBA makes mistakes or under-estimates the workload). Generally, the DBA tries to avoid over-allocation by assuming the worst workload in order to avoid paging (with dramatic degradation in performance) or query failure. The challenge for database systems is to design a fair and efficient strategy to manage this memory: allocate enough memory to each operation to minimize response time, but not too much memory so that other operators can receive their share of memory as well.

In Oracle9i, we introduced a new memory manager that *dynamically adapts* the memory allocation based on the operation's *need* and the system *workload*. This improves both *manageability* and *performance*. The manageability is improved by relieving the DBA from his "role" of finding optimal values for memory configuration parameters. The performance is improved by allocating the memory to operators to maximize throughput and make the operators dynamically adapt their memory consumption to respond to changes in the workload.

Section 2 presents an overview of related works in commercial systems. In Section 3, we give an overview of the Oracle database system memory model, and in Section 4 we present the new memory manager for database operators, including the architecture and algorithms. In Section 5, we discuss the memory advisor component. Section 6 presents the results of a performance study that validates and shows the superiority of our approach. Section 7 concludes the paper.

## 2. Related Work

In this section we analyze the approaches to SQL memory management and classify commercial database systems based on the most important features of a memory manager.

## 2.1. Approaches to Memory Management

A very simple and common approach is to assign a fixed amount of memory to each operator. This amount can be either a constant internal value or derived from configuration parameters set by a DBA. This approach is obviously flawed because there is no ideal static configuration. The DBA will have to know:

- The performance characteristics of each operator (e.g, sort and hash join performance are different with regard to memory usage) and its requirements which depend on the input data size.
- The scheduling and degree of parallelism of operators inside each query to estimate how much memory the query needs.
- The application workload to estimate the memory needed at different points in time.

An improvement on this approach is to give each operator an amount of memory based on a size estimate of its input data. For example, a sort with a 1GB input will be assigned 10MB of memory, while a sort with a 10GB input will be assigned 100MB. This approach can also be improved to take into account operators scheduling and the degree of parallelism, but is still flawed because:

- the size estimates are made by the optimizer and can be off by orders of magnitude.
- the amount of memory assigned to operators is not constrained by the workload.

A third approach would take into account the current workload by checking the total amount of memory used by existing operators and assign an amount that keeps the total memory used below a certain threshold. This approach is not fair because it penalizes new operators to compensate for excesses made by operators already in the system.

In the fourth approach, an operator adapts its memory usage to respond to the memory demand in the system, so that all operators are treated equally.

## 2.2. Commercial Database Systems

Each one of the commercial database systems considered in this analysis implements a unique policy to manage memory used by the operators. However, based on the discussion above we identified three criteria that can be used to describe and classify those systems.

- *initial work area size*. Is the work area size of an operator at the beginning of its execution *dynamically* computed by the database system based on dynamic factors such as the application workload and the input characteristics of the operation, or is it *statically* derived from configuration parameters set by the DBA?
- *work area size during execution*. Once the operator has been started, is it able to *adapt* its memory consumption (either to use less or more memory)?
- *response to memory pressure*. How does the system respond when demands for memory (either from new operators or existing operators) cannot be satisfied, e.g. the total memory used by all operators reaches a limit (hard or soft)? Does it ask the operators running in the system to reduce their consumption, queue the new operators, or make the new query fail?

Table 1 summarizes the characteristics of the memory management policy used by each system. The dynamic nature of the initial work area size is different in SQLServer7 [SQLServer7] and Oracle9i [Oracle9i]. In SQLServer7 the optimizer produces minimum and maximum estimates for each operator. When the operator is started, the memory manager grants the operation its maximum memory if there is enough available memory and its minimum memory otherwise. In Oracle9i, the operation can get up to a maximum memory size computed by the system based on the current workload. See Section 4.2 to learn how this maximum is computed.

The initial work area size is static in the other systems. For example, in DB2 [DB2V7.1] the DBA sets the *sortheap* parameter to specify the amount of memory to allocate to a sort operator.

***Table 1: Classification of Commercial Systems***

| Database System | Initial work area size | size during execution | response to memory pressure |
|---|---|---|---|
| Oracle8i | static | static | none |
| Oracle9i | dynamic | adaptive | adaptive |
| DB2/UDB7.1 | static | static | minimum |
| Informix9.3 | static | static | limit ops |
| SQLServer7 | dynamic[a] | static | queueing |
| Teradata | static | static | ? |

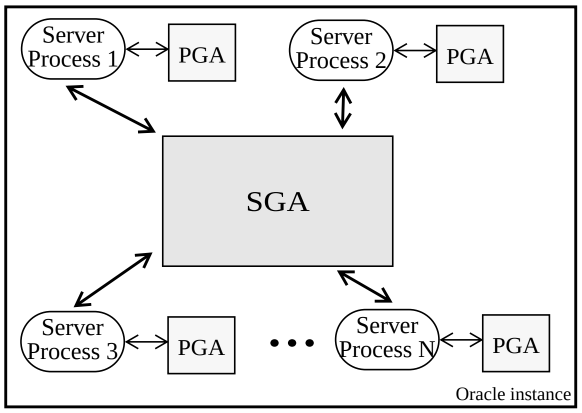
a. using a minimum and maximum thresholds

Oracle9i is the only system where operators can adapt during their execution. This is very important if we want to adapt to an increase in the workload and at the same time make the memory management policy fair to all operators, regardless of the time of entry in the system. Other systems try to compensate for this by allocating a minimum memory for the new operators [DB2V7.1], queue the new queries [SQLServer7], or don't accept the new queries [Informix9.3].

## 3. Oracle Memory Architecture

Oracle uses a hybrid memory model based on both shared and private memory. The System Global Area (SGA) is the memory region where the Oracle database server, also referred to as an Oracle instance, allocates all the shared data and control structures. Hence, any server process acting on behalf of an Oracle instance can allocate and access structures located in SGA. The main SGA data structures are the buffer cache and the library cache. The former holds a copy of the most recently accessed database blocks while the later is used to cache dictionary metadata and SQL execution plans. Unlike the SGA, the Process Global Area (PGA) is a private memory region allocated by Oracle when a database server process is created. There is one PGA memory region per Oracle server process which contains control and data information for that server process. Generally, a big portion of the PGA memory holds cursor run-time memory. Figure 3 illustrates the Oracle memory model.

***Figure 3: Oracle Memory Model***



Two dynamic configuration parameters, *db_cache_size* and *shared_pool_size* control the size of each SGA component and hence the overall size of the SGA memory. By altering the value of these parameters, the DBA can dynamically reconfigure the SGA. Oracle provides "advice statistics" for both the library cache and the buffer cache. These statistics, updated in real-time based on the database workload, are aimed at assisting the DBA in determining the best configuration for the SGA memory. For example, the buffer cache advice statistics predicts the value of the cache hit ratio for various sizes of the buffer cache, selected around its current size.

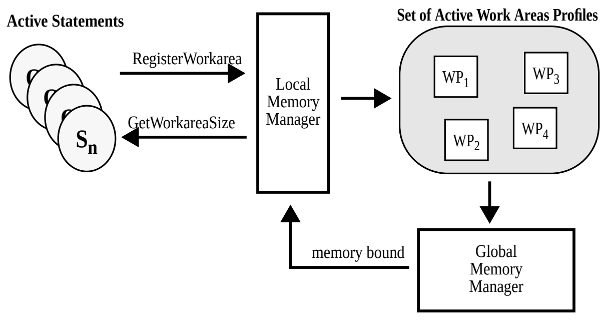
This paper focuses on the PGA memory management since this memory holds the run-time memory of executing SQL statements. A large part of that memory is dedicated to SQL work areas, used by sort, hash-join, bitmap-index merge, and bitmap-index create operators. The total PGA memory allocated by each server process attached to an Oracle instance is referred to as the **aggregated PGA memory** of that instance. The DBA can control the aggregated PGA memory consumed by the instance by setting the configuration parameter *pga_aggregate_target*. For example, the DBA can set this parameter to the value "10G" to limit the overall consumption of PGA memory to 10 giga-bytes.

## 4. Automatic PGA Memory Management

### 4.1. Overview

PGA memory management in Oracle9i is mainly based on the feedback loop mechanism depicted in Figure 4. The left side of the figure represents active statements, i.e. statements which are executing. When a SQL operator starts, it registers its work area profile using the "local memory manager" services. A work area profile is the only interface between a SQL operator and the memory manager. It is a piece of metadata which describes all the characteristics of a work area: its type (e.g. sort, hash-join, group-by), its current memory requirement to run with minimum, one-pass and cache memory, the number of instances of that work area (effectively the degree of parallelism of the operator), and finally the amount of PGA memory currently used by this work area.

***Figure 4: Memory Management Feedback Loop***



The set of active work area profiles are maintained by the local memory manager in shared memory (SGA). This set is always mutating. First, new work area profiles are added when memory intensive SQL operators start processing their input rows. These profiles are removed when corresponding operators complete their execution. Second, the content of each work area profile is frequently updated by its corresponding SQL operator to reflect its current memory need and consumption. Hence, at any point of time, the set of all active work area profiles closely captures the PGA memory need and consumption of the Oracle instance.

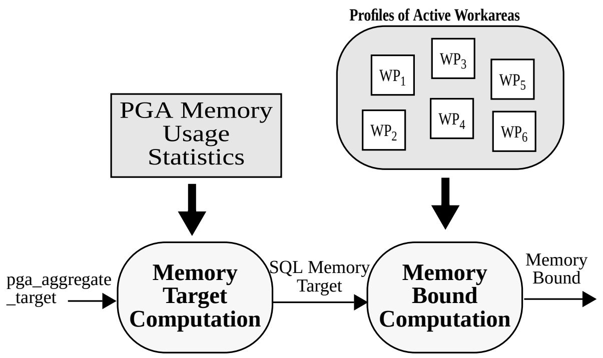
The global memory manager is a background daemon which indirectly determines the size of each active work area by publishing a “memory bound” at a regular interval, generally every three seconds. The memory bound is automatically derived from the number and the characteristics of all active work area profiles. It is used to constrain the size of each work area. Hence, the memory bound is high when the overall memory requirement of all active work areas is low and vice-versa.

The feedback loop is closed by the local memory manager. It uses the current value of the memory bound and the current profile of a work area to determine the correct amount of PGA memory, called *expected size*, which can be allotted to this work area. The expected size is checked periodically by SQL operators which are then responsible to adapt their work area size to the specified value.

### 4.2. Global Memory Bound Computation

As mentioned in the above section, the Oracle global memory manager daemon refreshes the global memory bound periodically. Figure 5 shows the two steps involved in that process. The first step computes the SQL memory target which is derived from the dynamic parameter “*pga_aggregate_target*” set by the DBA. In simple terms, the target SQL memory is the amount of PGA memory which can only be allotted to active SQL work areas. The second step translates this global limit into a local limit, the memory bound. The memory bound is derived from the SQL memory target and the active set of work area profiles. The processes that compute those values are discussed in the sections below.

***Figure 5: Bound Computation Control Flow***



#### 4.2.1. SQL Memory Target

The concept of SQL memory target is critical to the proper functioning of the SQL memory manager. Primarily, the role of the SQL memory target is to account for other usages of PGA memory. Indeed, besides SQL, various components in the database server make use of PGA memory, leaving only a portion of that memory to SQL work areas. This portion represents most of the PGA memory for decision support systems, e.g, up to 90% of the overall PGA memory, while it could be relatively small in pure OLTP systems, e.g. only 10%.

The target SQL memory has another important function. It also regulates the overall allocation of PGA memory and automatically compensates for any shortcoming with the dynamic memory management mechanism. If the Oracle database server starts to over-allocate PGA memory, the value computed above is further lowered to act as a counterpoise. Conversely, if not all the PGA memory is used by the database server, the target is further increased to maximize PGA memory utilization and improve performance. This explains why the “Memory Target Computation” module shown in Figure 5 uses some allocation statistics for the PGA memory to determine if the value of the SQL memory target needs to be further increased or decreased.

Over or under allocation of PGA memory can be explained by two main factors:

- **operators can be slow to adapt:** when the global memory manager publishes a new value for the bound, either higher or lower, there is always some delay before the SQL operators in the system can fully adapt to the new bound. Indeed, although all memory intensive operators, like hash-joins or sorts, have been made highly adaptive to memory changes, memory cannot be released and acquired instantaneously. Moreover, some of these operators have phases during which releasing memory is difficult if not impossible. As a result, the system can lag in following the global memory manager directives expressed via the memory bound.
- **memory fragmentation:** it is sometimes hard for an operator to precisely know how much memory it is using since memory can get fragmented. For instance, when an operator allocates 10MB of PGA memory, in reality it may be using 11MB. Oracle uses various techniques to prevent memory fragmentation but, practically, a small fraction of the allocated memory directly results from this fragmentation. Hence, an operator might use slightly more memory than the limit imposed by the memory bound. The global memory manager will compensate for these type of accounting mistakes by automatically reducing the SQL memory target.

To summarize, the SQL memory target acts like a watch dog for the auto memory management feature. It varies in response to changes in the value of the parameter “*pga_aggregate_target*” made by the DBA, it shrinks or grows based on the amount of PGA memory used by other consumers of PGA memory and finally reacts to any over or under allocation of PGA memory.

### 4.2.2. Global Memory Bound

Once the value of the target SQL memory is determined, the bound can be computed. In simple terms, finding a proper value for the memory bound is finding the maximum value for which the sum of the expected work area size of each operator will be less than or equal to the target SQL memory. We use the term "expected" since there is no formal guarantee that every active SQL operator will comply immediately with the new published bound. Formally, computing the bound involves finding the maximum value B which solves the following equation:

***Equation 1: Property of the Memory Bound***

$$\sum_{i=1}^{N} \text{ExpectedWorkareaSize}(WP_i, B) \leq \text{SQL Memory Target}$$

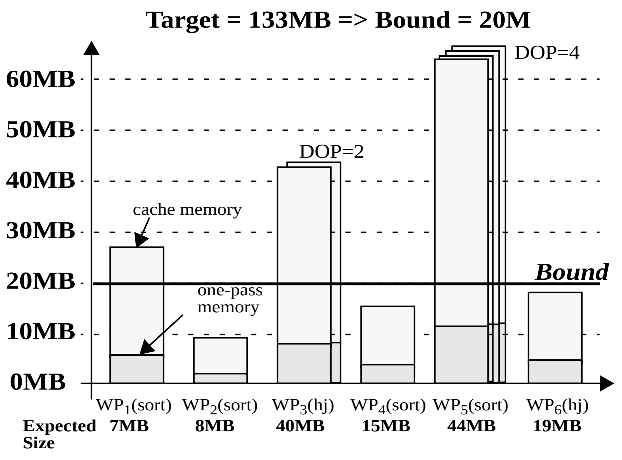
In the above equation, N is the number of work area profiles and $WP_i$ is the memory profile associated to work area number i. Given a work area profile and the bound value, the *ExpectedWorkareaSize* function returns the expected size of the work area and monotonically increases with the memory bound. It is beyond the scope of this paper to detail the algorithm used by Oracle to compute the memory bound. But it is important to mention that its complexity is almost linear to the number of work area profiles. This is critical since the global memory manager refreshes the bound at a high frequency.

Work area profiles are data structures shared between the global memory manager and SQL operators. A profile holds the main characteristics of a SQL work area at a given instant of time, mainly the minimum, the one-pass and the cache memory thresholds of the work area. Based on the work area profile and the bound value, the *ExpectedWorkareaSize* function determines the size of the work area using the following simple rules:

- **rule 1:** the expected size can never be less than the minimum memory requirement.
- **rule 2:** the expected size can never be more than the cache requirement.
- **rule 3:** if the bound is between the minimum and the cache requirement of the work area, the expected size will be equal to the memory bound, except if the operator is a sort. The reason is that, as shown in Section 1, the sort does not benefit from more memory than the one-pass memory size unless it can run cache. Hence, if the operator is a sort and the bound is between one-pass and cache, the sort will get only one-pass memory requirement.
- **rule 4:** if the work area is parallel, the expected memory derived by the above three rules will be multiplied by the degree of parallelism (DOP). This is because a parallel operation gets IO, CPU and memory resources in proportion to its DOP.
- **rule 5:** finally, the expected memory can never exceed 5% of the overall memory target if the operator is running serial. This amount is increased to 30% for operators running parallel. This limit guarantees that a reasonable number of operators are active in the system before we get to a point where most of the PGA memory is allocated. At that point, a sharp increase in the workload is less likely since a large number of operators are running. This smooths the variation in memory consumption and leaves more time for the memory manager to react to any variation in the memory demand.

Figure 6 shows how the global memory manager computes the expected work area size given six work area profiles. For example, the first work area profile, $WP_1$ is used by a sort which needs 7MB to run one-pass and 27MB to run cache. $WP_3$ is used by a parallel hash-join running with degree 2. It requires 67MB to run cache and 11MB to run one-pass. Assuming that the SQL memory target is 133MB, the global memory manager sets the bound to 20MB. This value of the bound would limits the memory consumption of $WP_1$ to its one-pass memory (i.e. 7MB) since $WP_1$ corresponds to a sort and 20MB is not enough to run cache. With a bound set to 20MB, $WP_3$ would get up to 40MB, two times the bound since this work area runs parallel with degree 2.

***Figure 6: Bound Computation for a 133MB target***



### 4.3. Drift Management

There is a potential issue with the background computation of the memory bound. Indeed, the feedback loop is executed often but this might not be enough in some cases. For instance, consider the scenario where many complex queries are submitted simultaneously when the system is idle. When these queries are started, the bound is at its maximum value since there was no load on the system at the time the bound was last recomputed by the global memory manager. Hence, using this "stale" memory bound could cause some severe memory over-allocation since a lot of memory could be allocated before the memory bound is finally refreshed to reflect this sharp variation of the workload.

To tackle this issue, Oracle has also the ability to automatically refresh the bound in a foreground mode, i.e. on behalf of a running query. This only happens in the very rare event where a local memory manager detects that the bound is stale. Oracle maintains the staleness factor of the memory bound using a global statistic named the memory "drift". The drift represents the amount of expected memory acquired (positive drift) or released (negative drift) by queries since the memory bound was last recomputed. Every time the bound is recomputed, the memory manager resets the drift to 0. When the PGA memory consumption on the system is close to the maximum limit set by the DBA, we only accept 10% variation around the SQL memory target. If the current expected memory, which is the sum of the SQL memory target and the drift, crosses this 10% tolerance threshold, the bound is automatically recomputed in foreground mode.

***Figure 7: Foreground computation of memory bound***

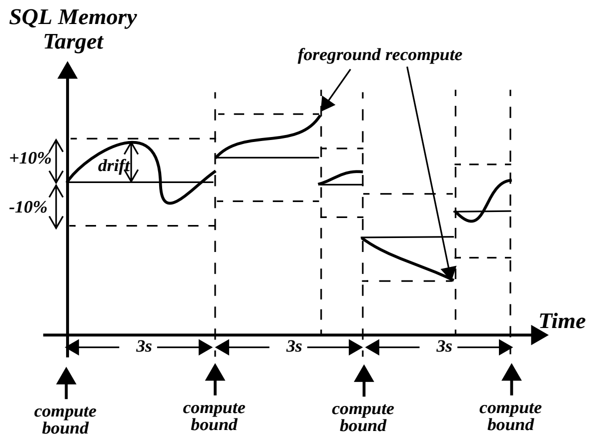


Figure 7 shows how foreground and background computation of the memory bound are combined. This example starts when the bound has just been refreshed by the global memory manager. At that point, the SQL memory target is computed and the drift is reset to 0. From that point on, any change in the set of work area profiles automatically triggers an update of the drift. For instance, when a work area profile is added (respectively removed) on behalf of a SQL operator start (respectively stop) action, the drift is increased (respectively decreased) by the expected size of that work area. Similarly, the drift is updated every time a SQL operator changes the memory requirement of its associated work area. Generally, the state of the system is more or less steady and the total expected memory stays within 10% of the SQL memory target within two "compute" points. This is exactly what happens in the above example, at least the first time. But the second and the third times, the total expected memory exceeds this 10% limit, triggering a foreground refresh of the memory bound.

In summary, by combining a background and a foreground method to refresh the bound, the SQL memory manager always maintains a memory bound whose value closely reflects the current workload. The background refresh allows the memory manager to keep up with reasonable variation in the workload. It is the preferred refresh method since it is performed asynchronously to running queries and has therefore no performance impact on running queries. In the very rare event where this is not enough, a foreground refresh is automatically triggered and ensures that a work area is not sized using a stale bound.

## 4.4. Memory Adaptive SQL Operators

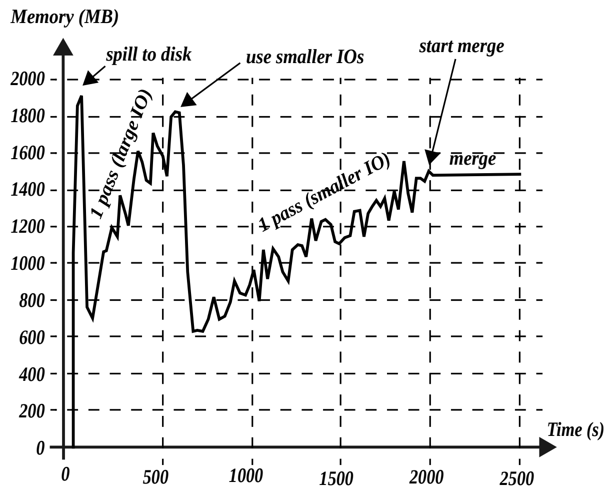
In Oracle9i, all SQL operators have been modified to dynamically adjust their memory usage during the course of their execution. This capability is critical for the proper functioning of the automatic PGA memory management, because it prevents:

- **over-allocation of PGA memory.** SQL operators must rapidly react to any change of the memory bound value. When the memory manager lowers the memory bound in response to an increase in memory pressure, it expects the system to quickly adjust to the new bound. The SQL memory target provides a mechanism to compensate for sharp variations in the workload, however, rapid adjustment by SQL operators is also critical to avoid any severe memory over-allocation.
- **sub-optimal memory utilization due to bad memory requirements estimate.** It's also critical to ensure optimal performance of SQL operators. When a SQL operator starts its execution, it can only make an *estimate* of its memory requirement since the input data has not yet been processed. This initial estimate is derived by the query optimizer, and is used by the local manager to derive the work area size it can grant. For instance, if a hash-join operator estimates that its cache memory requirement is 1MB, it will be

granted exactly one megabyte of PGA memory assuming that the value of the memory bound is high enough, e.g, 10MB. But the initial estimate is sometimes inaccurate and the actual data size may be much larger than anticipated, e.g, 2MB. Without adjusting dynamically its memory requirement while processing the build input, this hash-join will have to spill to disk after 1MB. Instead, by dynamically increasing its memory requirement while "discovering" the input data, the hash-join could be granted more memory and continue to run cache.

Let us illustrate this feature using a real-life example. Figure 8 shows the PGA memory consumption during the execution of a parallel index creation. The index was created on a 6 billion rows table, the *lineitem* table from the TPC-H benchmark [TPC00]. The create index statement is executed stand alone on a SUN E10000 with 64 processors (450Mhz). The create index is mainly a big sort operation. For the purpose of this experiment, the *pga_aggregate_target* parameter is set such that the overall memory used by the create index is limited to 2GB.

***Figure 8: Example of Adaptive Sort***



This example shows clearly how the sort dynamically adjusts, as input rows are processed, its memory requirement and therefore its overall memory consumption. The sort algorithm used in Oracle9i is highly adaptive. Initially, it assumes that the input data is tiny, no more than 64KB. Then it starts processing its input and keeps track of the actual input size as it discovers it. When the actual input size becomes larger than the current estimated input size, the sort operator reconfigures its memory by increasing the estimated input size by a constant factor (e.g. 20%). Based on this new estimate, the sorts recomputes its memory requirement and updates its work area profile.

The create index example illustrates perfectly this mechanism. During the first 10 seconds of its execution, the sort attempts to run cache by repetitively adjusting its cache memory requirement based on the size of the input already processed. This action automatically triggers an increase of its work area size. When the overall memory consumed hits the 2GB limit, the cache memory requirement exceeds the value of the memory bound and the sort has to spill to disk. At that point, it resizes its work area and runs using one-pass memory. From that point on, the sort estimate its one-pass memory requirement as $M_{one\text{-}pass} = 2 \times (R_{done} + R_{left}) \times S$, where $R_{done}$ is the actual number of sorted runs already produced on disk, and is initialized to 1 when the sort spills its first run to disk. $R_{left}$ is the remaining number of runs to produce and S is the IO size which will be used to merge these runs. $R_{left}$ is computed to achieve minimum memory requirement such that:

$$R_{left} = I_{left} / M_{one\text{-}pass} \qquad \textbf{(a)}$$

$$M_{one\text{-}pass} = 2 \times (R_{done} + R_{left}) \times S \qquad \textbf{(b)}$$

In equation (a), $I_{left}$ is the size of the input which remains to be consumed. Let $I_{cur}$ denotes the size of the input already processed and $I_{estim}$ the estimated input size. Hence, we have: $I_{left} = I_{estim} - I_{cur}$. From the above equations (a) and (b), we can derive a second-degree equation based on $R_{left}$:

***Equation 2: Estimation of $R_{left}$***

$$R_{left}^2 + R_{done} R_{left} - \frac{I_{left}}{2S} = 0 \Rightarrow R_{left} = \frac{-R_{done} + \sqrt{R_{done}^2 + \frac{2I_{left}}{S}}}{2}$$

The above equation is used to compute the ideal number $R_{left}$ such that the sort will use the minimum amount of memory to run one-pass. The one-pass memory is then derived from this number, i.e. $M_{one\text{-}pass} = 2 \times (R_{done} + R_{left}) \times S$. Once the sort spills to disk, there is no point to use more memory than the one-pass requirement hence, from that point on, the sort sets its cache requirement to the one-pass requirement.

$M_{one\text{-}pass}$ is directly proportional to the factor S which represents the IO size used during the merge phase that produces the final sorted result. The sort algorithm automatically adjusts S based on the amount of memory available at run-time. On one hand, performing large IOs (e.g. 256KB) will improve IO performance but on the other hand it could cause the one-pass memory requirement to be larger than the memory bound. When this happens, the sort automatically lowers the value of S down to a minimum limit. If this is not enough, the sort operation will start running multi-pass.

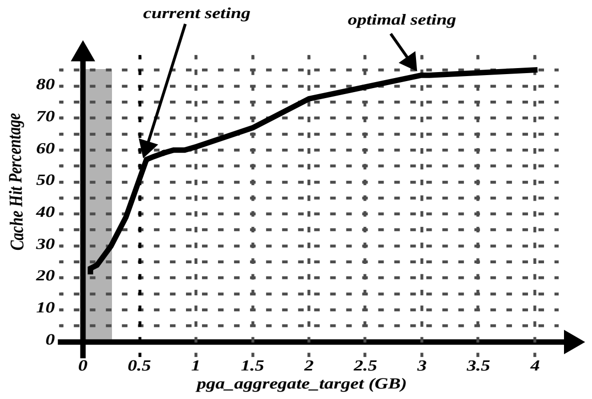
Figure 8 illustrates very well the above explanation. After spilling its first run, the sort releases a lot of memory since the one-pass memory requirement is small compared to the cache requirement. As the sort discovers that the input data is larger than anticipated, $I_{estim}$ is increased and the one-pass requirement is adjusted using the above formula. This explains why the memory consumed by the sort slowly increases until it reaches again the 2GB limit, at 600 seconds in the run. At that point, the sort operator knows that merging all runs which have been already produced would require more than 2GB of memory, unless the IO size (factor S in the above formula) is decreased. This adjustment is performed and the execution is resumed. Since smaller IOs will be used, the one-pass memory requirement is lowered again and explains the second drop in memory consumption. After that, once again, the memory consumption slowly increases as the estimated input size is augmented, up to the point where the input data is fully consumed. Merging the runs can then start (merge phase is missing from the graph).

## 5. PGA Advice Statistics

Oracle9i has many manageability features to assist the DBA in tuning the performance of an Oracle instance. Properly sizing the PGA memory via the configuration parameter *pga_aggregate_target* is a key aspect of that tuning process. In Oracle9i Release 2, Oracle helps the DBA in that task by providing two advice performance views, *v$pga_target_advice* and *v$pga_target_advice_histogram*. By examining these two views, the DBA no longer needs to use an empirical approach to the tuning process. Instead, he/she can use the content of these views to determine how key PGA statistics will be impacted by changing the value of *pga_aggregate_target*. In both views, values of *pga_aggregate_target* used for the prediction are derived from fractions and multiples of the current value of that parameter, to assess possible higher and lower values.

Oracle generates PGA advice performance views by recording the workload history and then simulating this history for different values of *pga_aggregate_target*. The simulation process is performed continuously in the background by the global memory manager. Figure 9 shows as an example the content of the *v$pga_target_advice* obtained after running several concurrent query streams against a small TPC-H database (30GB scale factor).

***Figure 9: Displaying v$pga_target_advice***



The curve shows how the PGA "cache hit percentage" metric improves as the value of the configuration parameter *pga_aggregate_target* increases. This "cache hit percentage" metric reflects the average percentage of SQL work areas which are able to run cache, weighted by their cache requirement. Hence, this metric is impacted both by the number and the size of the set of work areas which are able to run cache. The maximum value is 100% and is reached when every work area runs cache.

The shaded zone in the graph is the "over allocation" zone which indicates that *pga_aggregate_target* is insufficient to meet the minimum PGA memory needs. If *pga_aggregate_target* is set within the over-allocation zone, the memory manager will over-allocate memory and actual PGA memory consumed will be more than the limit set by the DBA. It is therefore meaningless to set a value of *pga_aggregate_target* in that zone. In this particular example *pga_aggregate_target* should at least be set to 375MB.

Beyond the over-allocation zone, the value of the PGA cache hit percentage increases rapidly. This is due to an increase in the number of work areas which run cache or one-pass and a decrease in the number of multi-pass executions. At some point, somewhere around 500MB in this example, there is an inflection in the curve which corresponds to the point where most (probably all) work areas can run at worst one-pass. After this inflection, the cache hit percentage keeps increasing but at a lower pace up to the point where it starts to taper off and shows only slight improvement when *pga_aggregate_target* is increased. In the above example, this happens when *pga_aggregate_target* reaches 3GB. At that point, the cache hit percentage is 83% and only improves marginally (2%) with one extra giga-byte of PGA memory.

In this particular example, 3GB is probably the optimal value for the initialization parameter *pga_aggregate_target*. Based on this advice statistic and the maximum amount of memory which can be dedicated to the PGA memory, the DBA can decide to change the setting of *pga_aggregate_target*.

## 6. Performance Evaluation

In this section, we validate our approach to SQL memory management introduced in Oracle9i. The performance evaluation is not comprehensive because the intention is not to prove that the some algorithms are better than others. The results should mostly be viewed as an illustration of what to expect from our approach.

### 6.1. Manual versus Automatic

The first performance evaluation compares manual memory management (e.g. Oracle8i) against the Oracle9i automatic memory management. As explained above, when the SQL memory is manually managed, the DBA specifies the maximum work area size for each type of operator. Since the workload is always changing, either the specified values are too high and the memory will be over-allocated or too low and the memory will be under-utilized. In this evaluation, we would like to illustrate those two cases and compare them to the performance when the memory is automatically managed.

The performance evaluation was conducted on a SUN E4000 with 10 processors (167Mhz) and 2GB of main memory using a 30GB TPC-H database [TPC00]. For this experiment, we used a variant of the TPC-H query 9 which is executed using a degree of parallelism of 10. This query was executed several times by each database user and its average response time was recorded. This test was performed using a varying number of users, from 1 to 20. Three different settings were used: manual memory management with a hash and sort area size set to 5MB (Manual-5), manual memory management using 15MB (Manual-15) and automatic memory management with *pga_aggregate_target* parameter set to 1.5GB, i.e, 3/4 of the 2GB memory in the system.

Figure 10 shows the PGA memory consumption for these three cases. As expected, for both manual settings, the memory consumption increases linearly with the number of users, about 100MB per user in the Manual-5MB case and 250MB per user in the Manual-15 case. Observe that the memory consumed per user is not directly related to the 5MB or 15MB setting. This is because the query is a complex parallel query where one work area is used per parallel server process and multiple SQL operators are active simultaneously.

For Manual-5, the PGA memory is not fully utilized and remains below what the system can handle up to the point where 16 users are running. So, most of the time, users could have used more memory. The Manual-15 run represents the other extreme: memory starts to be over-allocated with 6 or more users causing the system to thrash very quickly. At 20 users, almost 5GB of memory will be used while the system is configured with only 2GB of physical memory. Automatic memory management is obviously doing a much better job: it maximizes memory utilization while keeping it always on target (i.e. 1.5GB), irrespective of the number of users. It means that when few users are active each of them will be able to use much more memory than in the manual mode, and when many users are active they will not thrash the system like in Manual-15.

***Figure 10: Automatic versus Manual (Memory Used)***

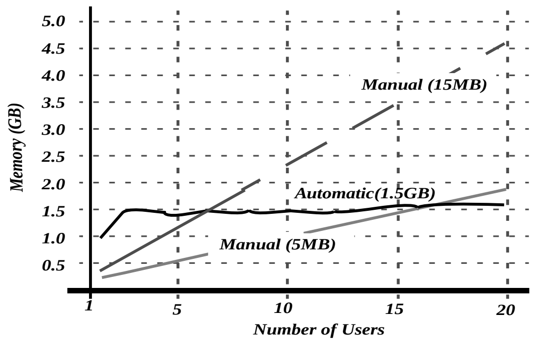
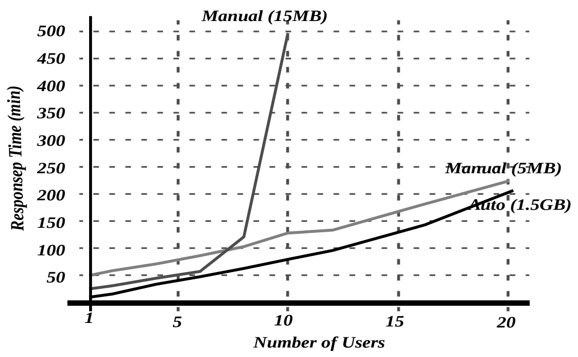


Figure 11 shows the average response time of the query for the above three measurements. The results conform to the predictions: with one user, the average response time is the highest for Manual-5 (51 minutes) since this user cannot use all the available memory. The response time gets better for Manual-15 (26 minutes) since this user uses more memory than Manual-5. The best result (11 minutes) is obtained with automatic memory management since that single user can use up to 1GB of memory. When more users are executing, the response time for Manual-15 shoots up dramatically when the number of active users reaches 6 because the system runs out of physical memory and has to page. In all cases, the automatic memory manager has superior performance since it manages memory better.

***Figure 11: Automatic versus Manual (Response Time)***



### 6.2. Variable Workload

In this experiment we analyze how the memory manager responds to variations in the database workload. We vary the workload by changing the number of queries running concurrently and the complexity of those queries. To achieve this we created two TPC-H schemas using different scale factors: 100G and 1TB. We then defined two classes of users: the *light* user class which runs on the 100GB schema and the *heavy* user class which runs on the 1TB schema. Users in both classes continuously submit queries selected randomly from the set of 22 TPC-H queries. To really stress the memory manager, the number of users in both classes is varied over time. In summary, the workload varies along three dimensions: mixed type of users, variable query complexity with a wide range of memory need, and varying number of users over time.

The number of users from each class is shown in Figure 12 with less users for the heavy class. The workload runs for 10 hours. For example, at 4 hours in the run, there are 3 heavy users and 24 light users running simultaneously. Users of both classes are running in parallel and the combined degree of parallelism for the instance is limited to 384 via the adaptive degree of parallelism feature. For the entire run, the value of *pga_aggregate_target* is set to 10GB.

***Figure 12: User Load Profile***

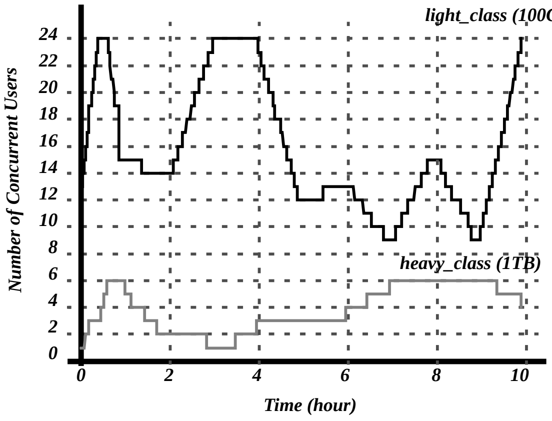


Figure 13 shows how the PGA memory consumption and the SQL memory target varies over time. The PGA allocated varies within a tight range and never exceeds 5.5% of the target at any time. This excess memory allocation is expected to happen as operators takes time to adjust. This is acceptable since the excess is short-lived and considering that the load contains a high ratio of long running operators with high memory need. Figure 13 highlights how the SQL memory target immediately reacts when PGA memory is over-allocated and acts as a counter balance to the PGA memory allocated, i.e, it moves in the opposite direction to regulate the amount of memory that can be allocated to SQL operators. For example, at 5 hours in the run, the PGA memory allocated increases and passes the *pga_aggregate_target* limit. Immediately, the SQL memory target is lowered.

***Figure 13: Memory Consumption***

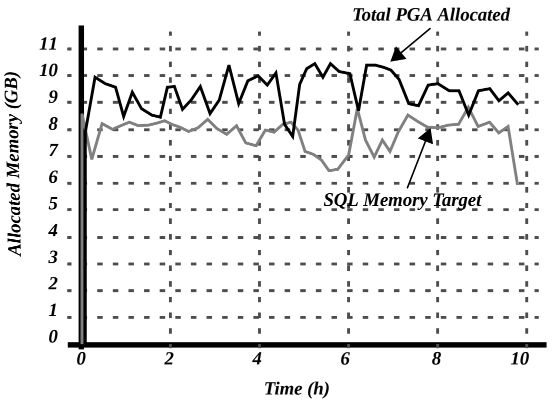
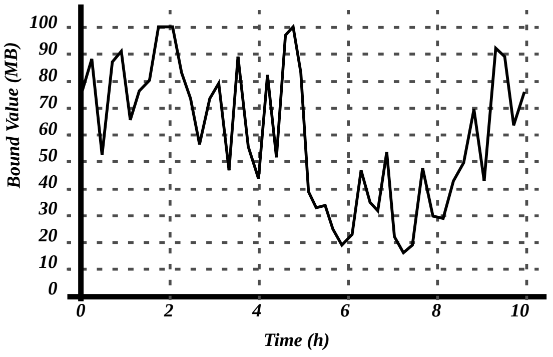


Figure 1 illustrates how the bound is continuously adjusted to respond to the workload variation.

***Figure 1: Dynamic Bound Adjustment***



## 7. Conclusion and Future

In this paper we showed the importance of memory management and its impact on the performance of a database system. First, we showed the relationship between the work area size and the performance of two commonly used operators, sort and hash join. Then we discussed the different approaches to the management of SQL memory, and exposed their benefits and drawbacks. Later, we presented the memory model in Oracle9i. We showed the internal details, architecture and algorithms, of the SQL memory manager and the SQL memory advisor components. Finally, we validated our approach through a set of performance experiments.

## References

[DB2V7.1]

DB2 UDB V7.1 Performance Tuning Guide, December 2000.

[Informix9.3]

Performance Guide for Informix Dynamic Server, Version 9.3, August 2001.

[Oracle9i]

Oracle9i Database Performance Guide and Reference, July 2001.

[Oracle8i]

Oracle8i Designing and Tuning for Performance, December 1999.

[SQLServer7]

Microsoft SQL Server 7.0 Storage Engine Capacity Planning Tips, March 1999.

[TeradataV2R4.1]

Teradata RDBMS, Utilities - Volume 1, V2R4.1, June 2001.

[TPC00]

Transaction Processing Council, TPC-H Rev. 1.2.1 specification, 2001