



CHAPTER 2

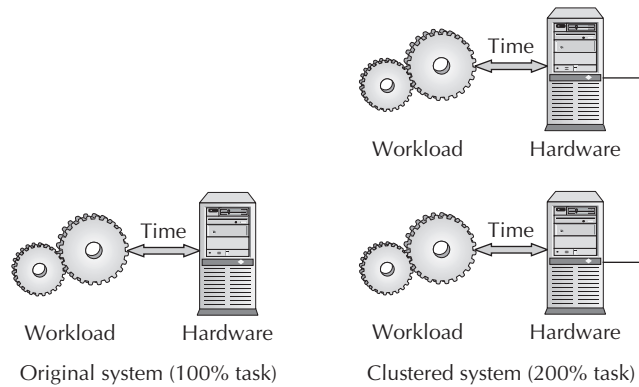
Clustering Basics and History



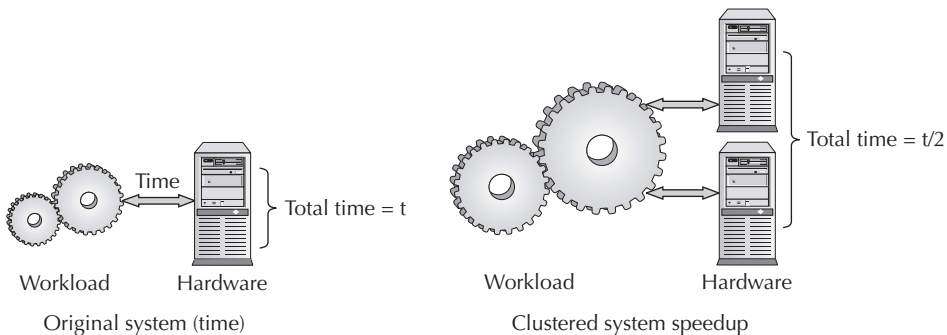
As defined in Chapter 1, a cluster is a group of interconnected nodes that acts like a single server. In other words, clustering can be viewed logically as a method for enabling multiple standalone servers to work together as a coordinated unit called a *cluster*. The servers participating in the cluster must be *homogenous*—that is, they must use the same platform architecture, operating system, and almost identical hardware architecture and software patch levels—and independent machines that respond to the same requests from a pool of client requests.

Traditionally, clustering has been used to *scale up* systems, to *speed up* systems, and to *survive failures*.

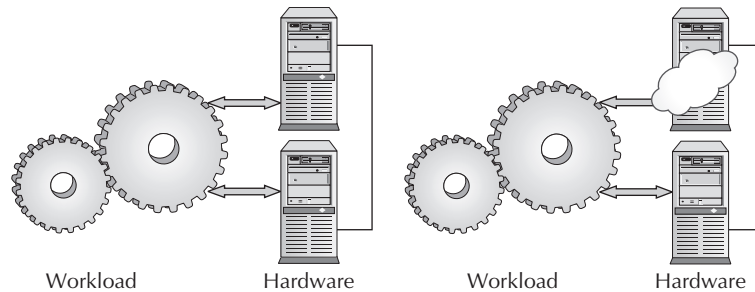
Scaling is achieved by adding extra nodes to the cluster group, enabling the cluster to handle progressively larger workloads. Clustering provides horizontal on-demand scalability without incurring any downtime for reconfiguration.



Speeding up is accomplished by splitting a large workload into multiple smaller workloads and running them in parallel in the available CPUs. Parallel processing provides massive improvement in performance for jobs that can be run in parallel. The simple “divide and capture” approach is applied to the large workloads, and parallel processing uses the power of all the resources to get work done faster.



Because a cluster is a group of independent hardware nodes, failure in one node does not halt the application from functioning on the other nodes. The application and the services are seamlessly transferred to the surviving nodes and the application continues to function normally as it was functioning before the node failure. In some cases, the application or the user process may not even be aware of such failures, as the failover to the other node is transparent to the application.



When a uniprocessor system reaches its processing limit, it imposes a big threat to scalability. Symmetric multiprocessing (SMP)—the use of multiple processors (CPUs) sharing memory (RAM) within a single computer to provide increased processing capability—solves this problem. SMP machines achieve high performance by *parallelism*, in which the processing job is split up and run on the available CPUs. Higher scalability is achieved by adding more CPUs and memory.

Figure 2-1 compares the basic architectural similarities between symmetric multiprocessing and clustering. However, the two architectures maintain cache coherency at totally different levels and latencies.

Grid Computing with Clusters

Clustering is part of Oracle's grid computing methodology, by which several low-cost commodity hardware components are networked together to achieve increased computing capacity. On-demand scalability is provided by supplying additional nodes and distributing the workload to available machines.

Performance improvement of an application can be done via three methods:

- By working harder
- By working smarter
- By getting help

Working harder means adding more CPUs and more memory so that the processing power increases to handle any amount of workload. This is the usual approach and often helps as additional CPUs address the workload problems. However, this approach is not quite economical, as the average cost of the computing power does not always increase in a linear manner. Adding computing power for a single SMP box increases the cost and complexity at a logarithmic scale.

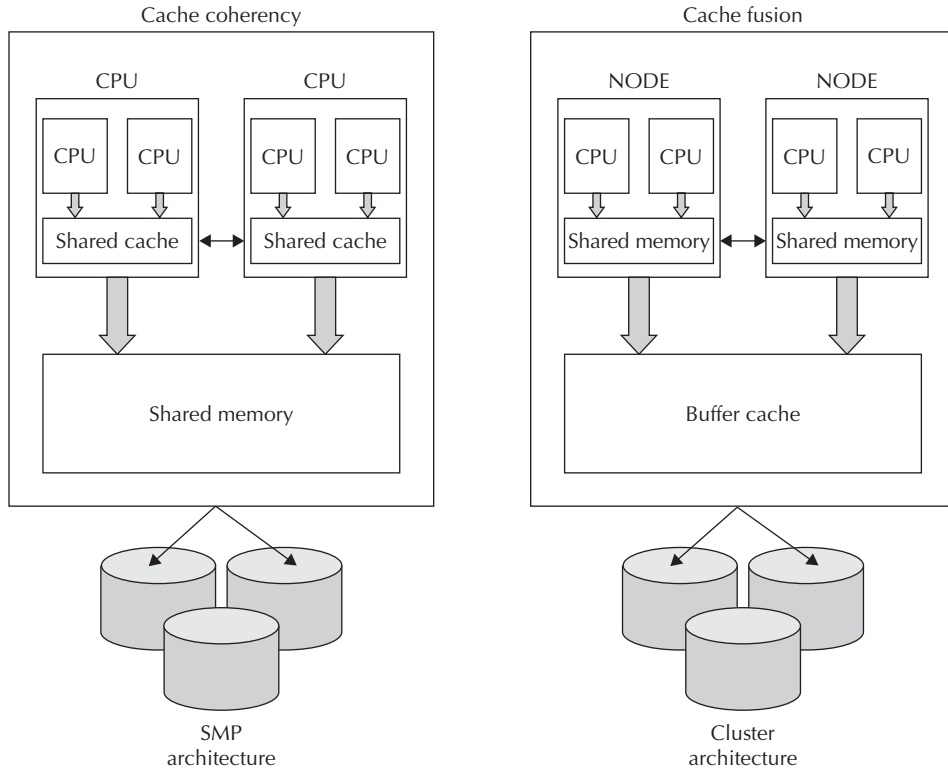


FIGURE 2-1. SMP vs. clusters

Working smarter is accomplished by employing intelligent and efficient algorithms. These reduce the total amount of work to be done to achieve the desired results. Working smarter often requires rewriting the application or changing the way it works (or sometimes changing the application design itself), which is quite impossible for a running application and requires unacceptable downtime. This option sometimes becomes almost impossible for a third-party vendor and packaged applications, as getting everyone onboard can become a tedious and time-consuming task.

Getting help can be as simple as using other machines' computing power to do the work. In other words, getting help is simply clustering the hardware, using the spare processing capacity of the idle nodes, and combining the processing transaction results at the end. More importantly, this approach does not require any changes to the application because it is transparent to the application. Another advantage to using this approach is that it allows on-demand scalability—you can choose to get help whenever required, and you do not need to invest in massive hardware.

Shared Storage in Clustering

One of the key components in clustering is *shared storage*. Storage is accessible to all nodes, and all nodes can parallelly read and write to the shared storage depending on the configuration of the cluster. Some configurations allow the storage to be accessible by all the nodes at all times; some allow sharing only during failovers.

More importantly, the application will see the cluster as a single system image and not as multiple connected machines. The cluster manager does not expose the system to the application, and the cluster is transparent to the application. Sharing the data among the available nodes is a key concept in scaling and is achieved using several types of architectures.

Types of Clustering Architectures

Clustering architecture can be broadly categorized in three types based on how storage is shared among the nodes:

- Shared nothing architecture
- Shared disk architecture
- Shared everything architecture

Table 2-1 compares the most common functionalities across the types of clustering architectures and their pros and cons along with implementation details with examples. Shared disks and shared everything architecture slightly differ in the number of nodes and storage sharing.

Shared Nothing Architecture

Shared nothing architecture is built using a group of independent servers, with each server taking a predefined workload (see Figure 2-2). If, for example, a number of servers are in the cluster, the total workload is divided by the number of servers, and each server caters to a specific workload. The biggest disadvantage of shared nothing architecture is that it requires careful application partitioning, and no dynamic addition of nodes is possible. Adding a node would require complete redeployment, so it is not a scalable solution. Oracle does not support the shared nothing architecture.

In shared nothing clusters, data is typically divided across separate nodes. The nodes have little need to coordinate their activity, since each is responsible for different subsets of the overall database. But in strict shared nothing clusters, if one node is down, its fraction of the overall data is unavailable.

The clustered servers neither share disks nor mirror data—each has its own resources. Servers transfer ownership of their respective disks from one server to another in the event of a failure. A shared nothing cluster uses software to accomplish these transfers. This architecture avoids the distributed lock manager (DLM) bottleneck issue associated with shared disks while offering comparable availability and scalability. Examples of shared nothing clustering solutions include Tandem NonStop, Informix OnLine Extended Parallel Server (XPS), and Microsoft Cluster Server.

The biggest advantage of shared nothing clusters is that they provide linear scalability for data warehouse applications, as they are ideally suited for that. However, they are unsuitable for online transaction processing (OLTP) workloads, and they are not totally redundant, so failure of one node will make the application running on that node unavailable. Still, most major databases, such as IBM DB2 Enterprise Edition, Informix XPS, and NCR Teradata, do implement shared nothing clusters.

Function	Shared Nothing	Shared Disks	Shared Everything
Disk ownership/sharing	Disks are owned by individual nodes and are not shared among any of the nodes at any time.	Disks are usually owned by active nodes and the ownership is transferred to the surviving node during failure of the active node—i.e., disks are shared only during failures.	Disks are always shared and all instances have equal rights to the disk. Any instance can read/write data to any of the disks as no disk is owned by any nodes. (Basically JBOD shared between all the nodes.)
Number of nodes	Typically very high number.	Normally two nodes, with only one node active at any time.	Two or more depending on the configuration. Number of nodes sometimes limited by the cluster manager or distributed lock manager (DLM) capacity when vendor supplied clusterware is used.
Data partitioning	Strictly partitioned as one node cannot access the data from the other nodes. Local nodes can access data local to that node.	Data partitioning not required as only one instance will access the complete set of data.	No data partitioning required as the data can be accessed from any nodes of the cluster.
Client coordinator	External server or any group member.	No coordinator required as the other node is used only during failover.	Not required. Any node can be accessed for any set of data.
Performance overhead	No overhead.	No overhead.	No overhead after three nodes. Less overhead up to three nodes.
Lock manager	Not required.	Not required.	DLM required to manage the resources.
Initial/on-demand scalability	Initially highly scalable, but limited to capacity of the individual node for local access. On-demand scalability not possible.	Not very scalable. Scalability limited to the computing capacity of the single node.	Infinitely scalable as nodes can be added on demand.

TABLE 2-1. *Functionalities of Clustering Architectures*

Function	Shared Nothing	Shared Disks	Shared Everything
Write access	Each node can write but only to its own disks. One instance cannot write to the disks owned by another instance.	Only one node at a time. One node can write to all the disks.	All nodes can write to all disks as the lock manager controls the writes.
Load balancing	Not possible.	Not possible.	Near perfect load balancing.
Application partitioning	Strictly required.	Not required as only one node is active at any time.	Not required.
Dynamic node addition	Not possible.	Possible, but does not make any sense as only one node will be active at any time.	Very possible. A key strength of this architecture.
Failover capacity	No failover as the nodes/disks are not accessible by other nodes.	Can be failed over to other nodes.	Very capable. Often it is transparent.
I/O fencing	Not required.	Not required.	Provided by the DLM and cluster manager.
Failure of one node	Makes a subset of data inaccessible momentarily: $100/N\%$ of the data is inaccessible, with N being the number of nodes in the cluster. Then the ownership is transferred to the surviving nodes.	Data access momentarily disrupted, until the applications are failed over to the other node.	As connections spread across all the nodes, a subset of sessions may have to reconnect to the other node and total data is accessible to all the nodes. No data loss due to node failures.
Addition of nodes	Requires complete reorganization as redeployment of the architecture.	Not possible as addition of nodes does not help anything.	Can be added and removed on the fly. Load balancing is done automatically during reconfiguration.
Example	IBM SP2, Teradata, Tandem NonStop, Informix OnLine XPS, Microsoft Cluster Server	HP M/C ServiceGuard, Veritas Cluster Servers	Oracle Parallel Server/RAC, IBM HACMP

TABLE 2-1. *Functionalities of Clustering Architectures (continued)*

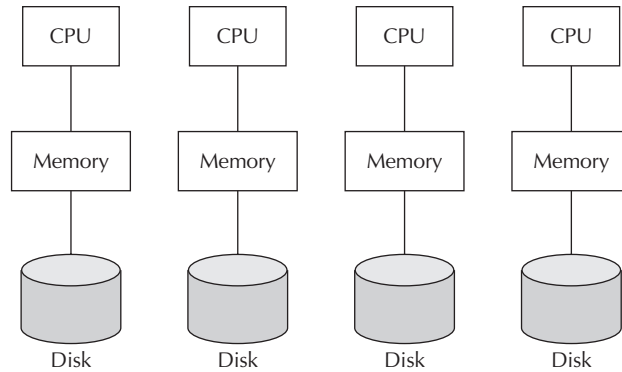


FIGURE 2-2. Shared nothing clusters

Shared Disk Architecture

For high availability, some shared access to the data disks is needed (see Figure 2-3). In shared disk storage clusters, at the low end, one node can take over the storage (and applications) if another node fails. In higher-level solutions, simultaneous access to data by applications running on more than one node at a time is possible, but typically a single node at a time is responsible for coordinating all access to a given data disk and serving that storage to the rest of the nodes. That single node can become a bottleneck for access to the data in busier configurations.

In simple failover clusters, one node runs an application and updates the data; another node stands idle until needed, and then takes over completely. In more sophisticated clusters, multiple nodes may access data, but typically one node at a time serves a file system to the rest of the nodes and performs all coordination for that file system.

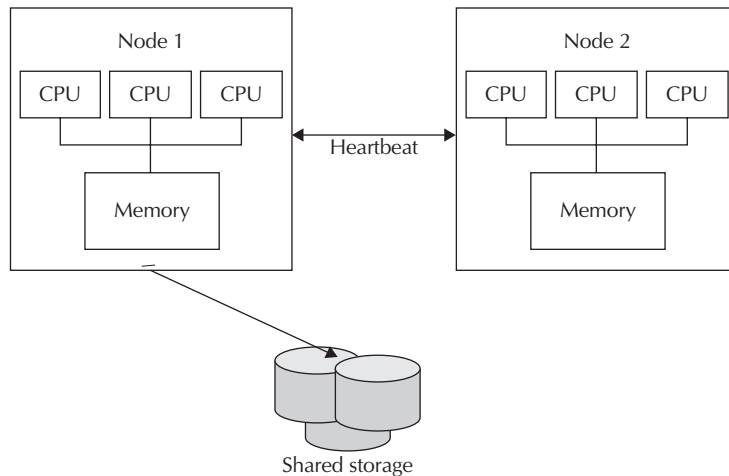


FIGURE 2-3. Shared disk cluster

Shared Everything Architecture

Shared everything clustering utilizes the disks accessible to all computers (nodes) within the cluster. These are often called shared disk clusters because the I/O involved is typically disk storage for normal files and/or databases. These clusters rely on a common channel for disk access as all nodes may concurrently write or read data from the central disks. Because all nodes have equal access to the centralized shared disk subsystem, a synchronization mechanism must be used to preserve coherence of the system. An independent piece of cluster software, the DLM, assumes this role.

In a shared everything cluster, all nodes can access all the data disks simultaneously, without one node having to go through a separate peer node to get access to the data (see Figure 2-4). Clusters with such capability employ a cluster-wide file system (CFS), so all the nodes view the file system(s) identically, and they provide a DLM to allow the nodes to coordinate the sharing and updating of files, records, and databases.

A CFS provides the same view of disk data from every node in the cluster. This means the environment on each node can appear identical to both the users and the application programs, so it doesn't matter on which node the application or user happens to be running at any given time.

Shared everything clusters support higher levels of system availability: If one node fails, other nodes need not be affected. However, higher availability comes at a cost of somewhat reduced performance in these systems because of overhead in using a DLM and the potential bottlenecks that can occur in sharing hardware. Shared everything clusters make up for this shortcoming with relatively good scaling properties.

Oracle RAC and IBM HACMP (High Availability Cluster Multiprocessing) are classic examples of the shared everything architecture. RAC is a special configuration of the Oracle database that leverages hardware clustering technology and extends the clustering to the application level. The database files are stored in the shared disk storage so that all the nodes can simultaneously read and write to them. The shared storage is typically networked storage, such as Fibre Channel SAN or IP-based Ethernet NAS, which is either physically or logically connected to all the nodes.

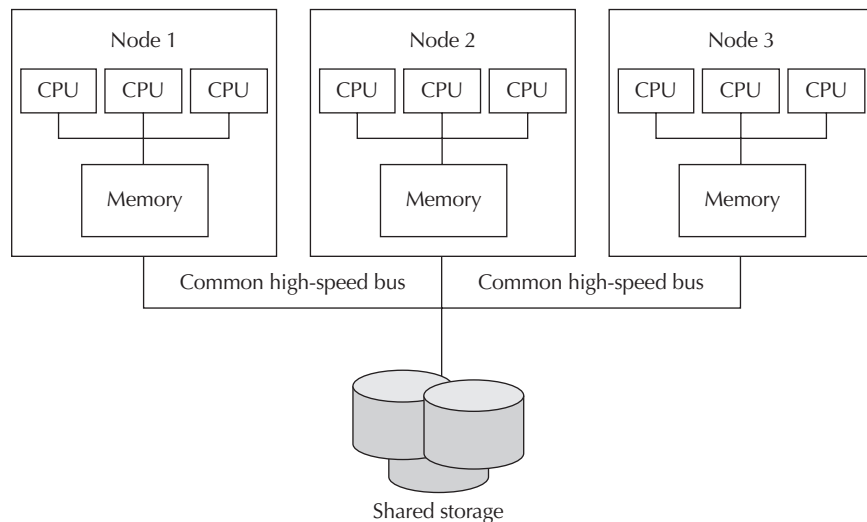


FIGURE 2-4. *Shared everything cluster*

History of Oracle RAC

The ever-hungry vagabonds of the Silicon Valley pioneered the concept of *massive parallel processing* (MPP) and christened the same as *clustering*. Digital, IBM, and Cray were some of the pioneers in the field of clustering. The first success toward creating a clustering product, ARCnet, was developed by DataPoint in 1977.

ARCnet, though a good product in the research labs and a darling in the academic world (for university-based research groups, departments, and computer cluster resources for an entire university), was not a commercial success and clustering didn't really take off until Digital Equipment Corporation (DEC) released its VAX cluster product in the 1980s for the VAX/VMS operating system. The ARCnet and VAX cluster products not only supported parallel computing, but they also shared file systems and peripheral devices. They were intended to provide the advantage of parallel processing while maintaining data atomicity.

Oracle's cluster database was introduced with Oracle 6 for the Digital VAX cluster product and on nCUBE machines, and Oracle was the first commercial database that supported clustering at the database level. Oracle created the lock manager for VAX/VMS clusters, as the original lock manager from Digital was not very scalable for database applications, and a database requires fine-grained locking at the block level. Oracle 6.2 gave birth to Oracle Parallel Server (OPS), which used Oracle's own DLM and worked very well with Digital's VAX clusters. Oracle was the first database to run the parallel server.

In the early '90s, when open systems dominated the computer industry, many UNIX vendors started clustering technology, mostly based on Oracle's DLM implementation. Oracle 7 Parallel Server (OPS) used vendor-supplied clusterware. OPS was available with almost all UNIX flavors and worked well, but it was complex to set up and manage as multiple layers were involved in the process.

When Oracle introduced a generic lock manager in version 8, it was a clear indication of the direction for Oracle's own clusterware and lock manager for future versions. Oracle's lock manager is integrated with Oracle code with an additional layer called OSD (Operating System Dependent). Oracle's lock manager soon integrated with the kernel and became known as the IDLM (Integrated Distributed Lock Manager) in later versions of Oracle.

Oracle Real Application Clusters version 9i used the same IDLM and relied on external clusterware. The following table lists the most common clusterware for various operating systems. Oracle provided its own clusterware for Linux and Windows in Oracle 9i and for all operating systems starting from 10g.

Operating System	Clusterware
Solaris	Sun Cluster, Veritas Cluster Services
HP-UX	HP MC/ServiceGuard, Veritas
HP Tru64	TruCluster
Windows	Microsoft Cluster Services
LinuxX86	Oracle Clusterware
IBM AIX	HACMP (High Availability Cluster Multiprocessing)

Oracle Parallel Server Architecture

An Oracle parallel or cluster database consists of two or more physical servers (nodes) that host their own Oracle instances and share a disk array. Each node's instance of Oracle has its own System Global Area (SGA) and its own redolog files, but the data files and control files are common to all instances. Data files and control files are concurrently read and written by all instances; however, redologs can be read by any instance but written to only by the owning instance. Some parameters, such as `db_block_buffers` and `log_buffer`, can be configured differently in each instance, but other parameters must be consistent across all instances. Each cluster node has its own set of background processes, just as a single instance would. Additionally, OPS-specific processes are also started on each instance to handle cross-instance communication, lock management, and block transfers.

Figure 2-5 shows the architecture of OPS.

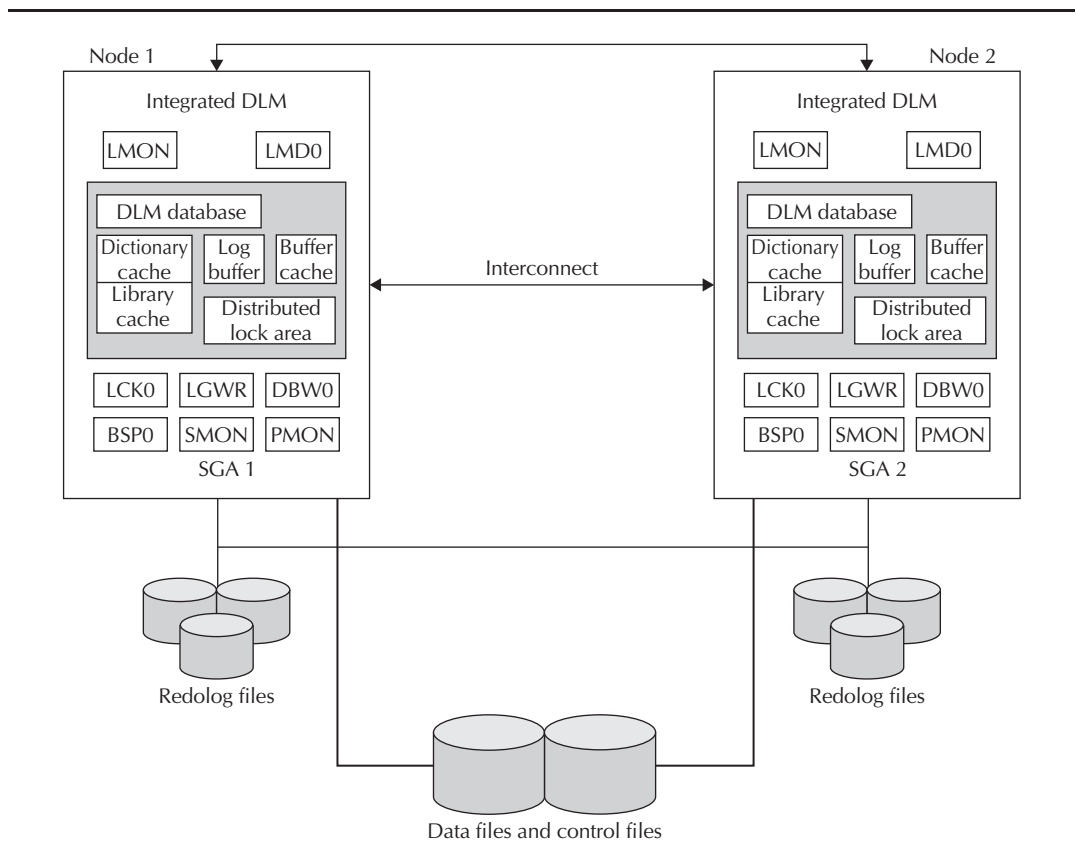


FIGURE 2-5. OPS architecture

Components of an OPS Database

On every instance of an Oracle 8i Parallel Server database, the following components can be found:

- Cluster manager—OS vendor specific (except Windows), including node monitoring facility and failure detection mechanism
- Distributed Lock Manager (DLM), including deadlock detection and resource mastering
- Cluster interconnect
- Shared disk array



NOTE

Vendor-specific cluster manager software is not discussed here and mentioned only for completeness. The vendor product is the basic cluster software that is mandatory before installing and configuring OPS. This is generally termed as the cluster manager (CM), which has its own group membership services, node monitor, and other core layers.

Cluster Group Services (CGS)

One of the key “hidden” or lesser known components of OPS is the CGS, which was formally known as the Group Membership Services (GMS) in Oracle 8. CGS has some OSD components (node monitor interface), and the rest of it is the GMS part that is built into the Oracle kernel. CGS holds a key repository used by the DLM for communication and network-related activities. This layer in the Oracle 8i kernel (and beyond) provides some key facilities without which an OPS database cannot operate:

- Internode messaging
- Group membership consistency
- Cluster synchronization
- Process grouping, registration, and deregistration

Some cluster components were still OS specific, so an OSD layer had to be developed and implemented. Certain low-level cluster communication can be done only by this OSD layer, which communicates with the Oracle kernel using Oracle-specified application programming interfaces (APIs). These OSD components are included with the Node Monitor (NM) that is part of the CGS.

A whole set of cluster communication interfaces and APIs became an internal part of the Oracle code in OPS 8i. GMS provided many services such as member status, node evictions, and so on, which was external in Oracle 8 but was made internal within Oracle 8i. GMS, now CGS in Oracle 8i (8.1.6), basically includes the GMS as well as the NM component of cluster management. Below this layer is the cluster monitor, or the cluster software provided by an OS vendor (such as TruCluster, Veritas Cluster Services, HP MC/ServiceGuard, and so on).

Oracle 8i also introduced network redundancy, in which problems or breakdowns of the network path were communicated up to the cluster monitor level (part of CGS) and allowed for the instance to be shut down or to provide for a failover of the network path using redundant network hardware. Until OPS version 8i, network problems would cause an indefinite hang of the instance.

Distributed Lock Manager (DLM)

The DLM is an integral part of OPS and the RAC stack. As mentioned, earlier Oracle versions relied on OS vendors to provide this component, which coordinated resources globally and kept the system in sync. Data consistency and integrity are maintained by this layer in all versions of Oracle OPS from 7 to 10g. Any reference to DLM from Oracle 8/8i onward pertains to the Integrated DLM (IDLDM), which was introduced with Oracle 8 and integrated into the Oracle OPS kernel. The terms *DLM* and *IDLDM* are used to describe a single entity and are one and the same.

In versions prior to version 8, the DLM API module had to rely on external OS routines to check the status of a lock (since DLM functions were provided by the OS vendor). This communication was done using UNIX sockets and pipes. With IDLDM, the required data is in the SGA of each instance and requires only a serialized lookup using latches and/or enqueues (see the next section) and may require global coordination, the algorithm for which was built into Oracle kernel code.

With OPS 8i, the IDLDM has undergone significant development and stabilization. IDLDM holds an inventory of all the locks and global enqueues that are held by all the instances in the OPS database. Its job is to track every lock granted to a resource. The coordination of requests from various instances for lock acquisition and release is done by the DLM. The memory structures required by DLM to operate are allocated out of the shared pool. The lock resources, messages buffers, and so on are all in the shared pool of each instance. DLM is designed such that it can survive node failures in all but one node of the cluster.

The DLM is always aware of the current holders or requestors of locks and the grantee. In case locks are not available, the DLM queues the lock requests and informs the requestor when a lock resource becomes available. Some of the resources that the DLM manages are data blocks and rollback segments. Oracle resources are associated with the DLM locks by instance, using a complex hashing algorithm. The lock and enqueue functionality in OPS is the same as in a single instance RDBMS server, except that OPS takes a global view.

DLM relies on the core RDBMS kernel for locking and enqueues services. These established and already proven techniques and functionalities need not be rewritten again in the DLM layer. The DLM coordinates locking at the global level, and this is a service that the core layers don't provide. To make things simpler, the locking and enqueues kernel services in Oracle 8i OPS, Oracle 9i RAC, and Oracle 10g RAC are kept separate from the DLM functionality.

Locking Concepts in Oracle Parallel Server

In an OPS database, a user must acquire a lock before he can operate on any resource. This is also applicable in a single instance scenario. In pure DLM terminology, a *resource* is any object accessed by users, and a *lock* is a client operational request of certain type or mode on that resource.

A new concept in OPS called *Parallel Cache Management (PCM)* means that coordination and maintenance of data blocks exists within each data buffer cache (of an instance) so that the data viewed or requested by users is never inconsistent or incoherent. The access to data is controlled using the PCM framework using data blocks with global coordinated locks. In simple terms, PCM ensures that only one instance in a cluster can modify a block at any given time. Other instances have to wait.

Broadly speaking, locks in OPS are either PCM locks or non-PCM locks. PCM locks almost exclusively protect the data blocks, and non-PCM locks control access to data files, control files, data dictionary, and so on. PCM locks are static in OPS, and non-PCM locks are dynamically set using the *init.ora* settings of certain parameters. Locks on PCM resources are referred to as *lock elements* and non-PCM locks are called *enqueues*. DLM locks are acquired on a resource and are typically granted to a process. PCM locks and row level locks operate independently.

PCM Lock and Row Lock Independence PCM locks and row locks operate independently. An instance can disown a PCM lock without affecting row locks held in the set of blocks covered by the PCM lock. A row lock is acquired during a transaction. A database resource such as a data block acquires a PCM lock when it is read for update by an instance. During a transaction, a PCM lock can therefore be disowned and owned many times if the blocks are needed by other instances.

In contrast, transactions do not release row locks until changes to the rows are either committed or rolled back. Oracle uses internal mechanisms for concurrency control to isolate transactions so modifications to data made by one transaction are not visible to other transactions until the transaction modifying the data commits. The row lock concurrency control mechanisms are independent of parallel cache management: concurrency control does not require PCM locks, and PCM lock operations do not depend on individual transactions committing or rolling back.

IDLM lock modes and Oracle lock modes are not identical, though they are similar. In OPS, locks can be local or global depending on the type of request and operations. Just as in a single instance, locks take this form:

<Type, ID1, ID2>

where *Type* consists of two characters and *ID1* and *ID2* are values dependant on the lock type. The ID is a 4-byte, positive integer.

Local locks can be divided into latches and enqueues. These could be required for local instance-based operations. A shared pool latch is a simple example of a local latch, irrespective of whether OPS is present or not. Enqueues can be local or global. They take on a global role in an OPS environment and remain local in a single instance. A TX (transaction) enqueue, control file enqueue (CF), DFS (Distributed File System) enqueue lock, or a DML/table lock are examples of a global enqueues in an OPS database. The same enqueue is local in a single instance database. Similarly, data dictionary and library cache locks are global in an OPS environment.

Local locks provide transaction isolation or row-level locking. Instance locks provide for cache coherency while accessing shared resources. GV\$LOCK and GV\$LOCK_ELEMENT are important views that provide information on global enqueues and instance locks.

In Oracle 8i, the DLM is implemented by two background processes, Lock Manager Daemon (LMD) and Lock Monitor (LMON) (see Figure 2-5). Each instance has its own set of these two processes. The DLM database stores information on resources, locks, and processes. In Oracle 9i, the DLM has been renamed Global Cache Services (GCS) and Global Enqueue Services (GES). While some of the code has changed, the basic functionality of GES and GCS remain the same as the prior versions of DLM.

DLM Lock Compatibility Matrix Every resource is identified by its unique resource name. Each resource can potentially have a list of locks currently granted to users. This list is called the *Grant Q*. Locks that are in the process of converting or waiting to be converted from one mode to another are placed on *Convert Q* of that resource. For each lock, a resource structure exists in the memory that maintains a list of owners and converters. Each owner, waiter, or converter has a lock structure, as shown in the following table.

Lock	NL	CR	CW	PR	PW	EX
NL	Grant	Grant	Grant	Grant	Grant	Grant
CR	Grant	Grant	Grant	Grant	Grant	Queue
CW	Grant	Grant	Grant	Queue	Queue	Queue
PR	Grant	Grant	Queue	Grant	Queue	Queue
PW	Grant	Grant	Queue	Queue	Queue	Queue
EX	Grant	Queue	Queue	Queue	Queue	Queue

Every node has directory information for a set of resources it manages. To locate a resource, the DLM uses a hashing algorithm based on the name of the resource to find out which node holds the directory information for that resource. Once this is done, a lock request is done directly to this “master” node. The directory area is nothing but a DLM memory structure that stores information on which node masters which blocks.

The traditional lock naming convention (such as SS, SX, X) are provided in the following table along with the DLM mode:

Conventional naming	NL	CR	CW	PR	PW	EX
DLM naming	NL	SS	SX	S	SSX	X

Note that in this table, NL means null mode; CR/SS means concurrent read mode; CW/SX means concurrent write mode; PR/S means protected read mode; PW/SSX means protected writemode; and EX/X means exclusive mode.

Lock Acquisition and Conversion Locks granted on resources are in the *Grant Q* (as discussed previously). Locks are placed on a resource when a process acquires a lock on the *Grant Q* of that resource. It is only then that a process owns a lock on that resource in a compatible mode.

A lock can be acquired if there are no converters and the mode the Oracle kernel requires is compatible with the modes already held by others. Otherwise, it waits on the *Convert Q* till the resource becomes available. When a lock is released or converted, the converters run the check algorithm to see if they can be acquired.

Converting a lock from one mode to another occurs when a new request arrives for a resource that already has a lock on it. *Conversion* is the process of changing a lock from the mode currently held to a different mode. Even if the mode is NULL, it is considered as holding a lock. Conversion takes place only if the mode required is a subset of the mode held or the lock mode is compatible with the modes already held by others and according to a conversion matrix within the IDLM.

Processes and Group-Based Locking When lock structures are allocated in the DLM memory area, the operating system Process ID (PID) of the requesting process is the key identifier for the requestor of the lock. Mapping a process to a session is easier inside Oracle, and the information is available in V\$SESSION. However, in certain clients, such as Oracle multithreaded servers (MTS) or

Oracle XA, a single process may own many transactions. Sessions migrate across many processes to make up a single transaction. This would disable identification of the transaction and the origin of the same. Hence, lock identifiers had to be designed to have session-based information where the transaction ID (XID) is provided by the client when lock requests are made to the DLM.

Groups are used when group-based locking is used. This is preferred particularly when MTS is involved, and when MTS is used the shared services are implicitly unavailable to other sessions when locks are held. From Oracle 9i onward, process-based locking no longer exists. Oracle 8i OPS (and later) uses group-based locking irrespective of the kind of transactions. As mentioned, a process within a group identifies itself with the XID before asking Oracle for any transaction locks.

Lock Mastering The DLM maintains information about the locks on all nodes that are interested in a given resource. The DLM nominates one node to manage all relevant lock information for a resource; this node is referred to as the *master node*. Lock mastering is distributed among all nodes.

Using the Interprocess Communications (IPC) layer, the distributed component of the DLM permits it to share the load of mastering (administering) resources. As a result, a user can lock a resource on one node but actually end up communicating with the LMD processes on another node. Fault tolerance requires that no vital information about locked resources is lost irrespective of how many DLM instances fail.

Asynchronous Traps Communication between the DLM processes (LMON, LMD) across instances is done using the IPC layer using the high-speed interconnect. To convey the status of a lock resource, the DLM uses *asynchronous traps* (AST), which are implemented as interrupts in the OS handler routines. Purists may differ on the exact meaning of AST and the way it is implemented (interrupts or other blocking mechanism), but as far as OPS or RAC is concerned, it is an interrupt. AST can be a *blocking AST* or an *acquisition AST*.

When a process requests a lock on a resource, the DLM sends a blocking asynchronous trap (BAST) to all processes that currently own a lock on that same resource. If possible and necessary, the holder(s) of the lock may relinquish the lock and allow the requester to gain access to the resource. An acquisition AST (AAST) is sent by DLM to the requestor to inform him that the resource (and the lock) is now owned by him. An AAST is generally regarded as a “wake-up call” for a process.

How Locks Are Granted in DLM To illustrate how locking works in OPS’s DLM, consider an example two-node cluster with a shared disk array:

1. Process p1 needs to modify a data block on instance 1. Before the block can read into the buffer cache on instance 1, p1 needs to check whether a lock exists on that block.
2. A lock may or may not exist on this data block, and hence the LCK process checks the SGA structures to validate the buffer lock status. If a lock exists, LCK has to request that the DLM downgrade the lock.
3. If a lock does not exist, a lock element (LE) has to be created by LCK in the local instance and the role is local.
4. LCK must request the DLM for the LE in exclusive mode. If the resource is mastered by instance 1, DLM continues processing. Otherwise, the request must be sent to the master DLM in the cluster.
5. Assuming the lock is mastered on instance 1, the DLM on this instance does a local cache lookup in its DLM database and finds that a process on instance 2 already has an exclusive (EX) lock on the same data block.

6. DLM on instance 1 sends out a BAST to DLM on instance 2 requesting a downgrade of the lock. DLM on instance 2 sends another BAST to LCK on the same instance to downgrade the lock from EX to NULL.
7. The process on instance 2 may have updated the block and may not have committed the changes. Dirty Buffer Writer (DBWR) is signaled to write out the block to disk. After the write confirmation, the LCK on instance 2 downgrades the lock to NULL and sends an AAST to DLM on the same instance.
8. DLM on instance 2 updates its local DLM database about the change in lock status and sends a AAST to DLM on instance 1.
9. The master DLM on instance 1 updates the master DLM database about the new status of the lock (EX) that can now be granted to the process on its instance. DLM itself upgrades the lock to EX.
10. DLM on instance 1 now sends another AAST to the local LCK process informing it about the lock grant and that the block can be read from disk.

Cache Fusion Stage 1, CR Server

OPS 8i introduced Cache Fusion Stage 1. Until version 8.1, cache coherency was maintained using the disk (ping mechanism). Cache Fusion introduced a new background process called the *Block Server Process* (BSP). The major use or responsibility of BSP was to ship consistent read (CR) version(s) of a block(s) across instances in a read/write contention scenario. The shipping was done using the high-speed interconnect and not the disk. This was called Cache Fusion Stage 1 because it was not possible to transfer all types of blocks to the requesting instance, especially with the write/write contention scenario.

Cache Fusion Stage 1 laid the foundation for Oracle 9i and 10g for Cache Fusion Stage 2, in which both types of blocks (CR and CUR) can be transferred using the interconnect, although a disk ping is still required in some circumstances.

Oracle 8i also introduced the GV\$ views, or “global views.” With the help of GV\$ views, DBAs could view cluster-wide database and other statistics sitting on any node/instance of the cluster. This was of enormous help to DBAs as they earlier had to club or join data collected on multiple nodes to analyze all the statistics. GV\$ views have the instance_number column to support this functionality.

Block Contention Block contention occurs where processes on different instances need access to the same block. If a block is being read by instance 1 or is in the buffer cache of instance 1 in read mode, and another process on instance 2 requests the same block in read mode, *read/read contention* results. This situation is the simplest of all cases and can easily be overcome since there are no modifications to the block. A copy of the block is shipped across by BSP from instance 1 to instance 2 or read from the disk by instance 2 without having to worry about applying an undo to get a consistent version. In fact, PCM coordination is not required in this situation.

Read/write contention occurs when instance 1 has modified a block in its local cache, and instance 2 requests the same block for a read. In Oracle 8i, using the Cache Fusion Stage 1, instance locks are downgraded, and the BSP process builds a CR copy of the block using the undo data stored in its own cache and ships the CR copy across to the requesting instance. This is done in coordination with DLM processes (LMD and LMON).

If the requesting instance (instance 2) needs to modify the block that instance 1 has already modified, instance 1 has to downgrade the lock, flush the log entries (if not done before), and

then send the data block to disk. This is called a *ping*. Data blocks are pinged only when more than one instance need to modify the same block, causing the holding instance to write the block to disk before the requesting instance can read it into its own cache for modification. Disk ping can be expensive for applications in terms of performance.

A *false ping* occurs each time a block is written to disk, even if the block itself is not being requested by a different instance, but another block managed by the same lock element is being requested by a different instance.

A *soft ping* occurs when a lock element needs to be down converted due to a request of the lock element by another instance, and the blocks covered by the lock are already written to disk.

Write/Write Contention This situation occurs when both instances have to modify the same block. As explained, a disk ping mechanism results where the locks are downgraded on instance 1 and the block is written to disk. Instance 2 acquired the exclusive lock on the buffer and modifies it.

Limitations of Oracle Parallel Server

OPS's scalability is limited on transactional systems that perform a lot of modifications on multiple nodes. Its scalability is also limited to I/O bandwidth and storage performance.

OPS requires careful and clear application partitioning. For example, if two different applications require two logical sets of data, the application should be configured in such a way that one node is dedicated to one application type, and no overlapping of the connections occurs on the other node. Hence the scalability is also restricted to the computing capacity of the node. Because of this "application partitioning" limitation, OPS is of limited suitability for packaged applications.

On-demand scalability is also limited, as we cannot dynamically add the nodes in an OPS cluster. Sometimes adding nodes will require careful analysis of the application partitioning and sometimes application repartitioning. This greatly limits true scalability.

OPS does require careful setup and administration, as a third-party cluster manager is necessary for clustering, requiring additional cost to the total application deployment. In other words, OPS is not cheap in any context.

Lock Configuration

OPS requires a highly skilled database architect for tuning and initial configuration. OPS supports many types of locks, and each needs to be carefully analyzed; the number of locks is also limited by physical memory. DLM requires additional memory for lock management, and careful analysis is necessary before configuring fixed/releasable/hash locks.

OPS lacks diagnosable capacity. Operating systems, the cluster manager, and the database are integrated, and diagnosing the errors in this stack is quite complex, even for experienced support staff. Deadlock detection, lock tracing, and cluster-wide statistics are inadequate in OPS. These features or functionalities are difficult to implement or expensive to run, and customers are usually not cooperative enough to understand the complexities involved in dealing with the technical challenges, difficulties, or requirements.

For example, to diagnose an application locking problem or a deep-rooted OPS issue, a cluster system state dump of all the instances is required. In a four-node OPS cluster, this task *will consume time*. This is not a limitation but a requirement. Dumping the contents of the entire SGA, all its lock structures, and so on, is an expensive task. Oracle 9i RAC has come a long way in its diagnostics capability and Oracle 10g RAC has improved on those capabilities.

Manageability

OPS requires RAW partitions for almost all operating systems. RAW devices are complex to set up and manage for novice DBAs. A RAW device does not support dynamic file extensions, and there are some limitations in the number of RAW partitions an operating system can support. RAW device backup and recovery is different from normal file system backup, and extra care should be taken for RAW device management. Reconfiguration of a DLM takes a lot of time, and cluster availability (as well as database availability) is compromised due to the enormously heavy OSD-level API calls when a node crash is noticed by OPS. The RDBMS kernel has to make calls to many APIs to communicate with the OS cluster software, and the DLM has to wait for OS clusterware to finish its job before doing its own configuration. In Oracle 9i, tightly integrated code in the kernel for cluster communication has reduced this time lag.

The Oracle RAC Solution

Oracle RAC is a natural evolution of OPS. The limitations of the OPS are addressed through improvements in the code, the extension of Cache Fusion (discussed in detail later in the book), and dynamic lock remastering. Oracle 10g RAC also comes with an integrated clusterware and storage management framework, removing the dependency of the vendor clusterware and providing ultimate scalability and availability for database applications.

Availability

Oracle RAC systems can be configured to have no single point of failure, even when running on low-cost, commodity hardware and storage. With Oracle RAC, if database servers fail, applications simply keep running. Failover is frequently transparent to applications and occurs in seconds.

Scalability

Oracle RAC allows multiple servers in a cluster to manage a single database transparently. Oracle RAC allows database systems to scale out rather than scale up. This means that the previous ceilings on scalability have been removed. Collections of servers can work together transparently to manage a single database with linear scalability.

Reliability

Oracle RAC has proven time and again in all “near death” situations that it can hold the honor of the hour with its controlled reliability. The paradigm has shifted from reliability to maximum reliability. Analyzing and identifying the subsystem and system failure rates in accordance with the appropriate standard ensures reliability, which is a key component in the RAC technology.

Affordability

Oracle RAC allows organizations to use collections of low-cost computers to manage large databases rather than needing to purchase a single large, expensive computer. Clusters of small, commodity-class servers can now satisfy any database workload. For example, a customer needing to manage a large Oracle database might choose to purchase a cluster of 8 industry-standard servers with 4 CPUs each, rather than buying a single server with 32 CPUs.

Transparency

Oracle RAC requires no changes to existing database applications. A clustered Oracle RAC database appears to applications just like a traditional single-instance database environment. As a result, customers can easily migrate from single-instance configurations to Oracle RAC without needing to change their applications. Oracle RAC also requires no changes to existing database schemas.

Commoditization

Oracle RAC allows clusters of low-cost, industry-standard servers running Linux to scale to meet workloads that previously demanded the use of a single, larger, more expensive computer. In the scale-up model of computing, sophisticated hardware and operating systems are needed to deliver scalability and availability to business applications. But with Oracle RAC, scalability and availability are achieved through functionality delivered by Oracle above the operating system.

In a Nutshell

On demand scalability and uninterrupted availability can be achieved by the right clustering technology. Oracle database has a good history on clusters with the OPS and its successor RAC. The previous limitations of OPS are addressed in the current versions of RAC, with the introduction of the Cache Fusion framework. Oracle RAC 10g provides enormous scalability, availability, and flexibility at low cost. It makes the consolidation of databases affordable and reliable by leveraging scale across architecture.