Top 5 Performance Tips for Tuning a Data Warehouse on GRID

Executive Overview........................................................................................................... 3
Introduction ......................................................................................................................... 3
Balancing Grid Components ............................................................................................... 3
Many vs. Few Nodes............................................................................................................. 4
Sizing IO Components.......................................................................................................... 5
  Switch ............................................................................................................................ 6
  HBA ............................................................................................................................... 7
  Balance between switch capacity, number of disk arrays and HBAs......................... 7
  Interconnect..................................................................................................................... 7
  Tuning the Interconnect Protocols................................................................................... 8
ORION................................................................................................................................. 9
Partitioning......................................................................................................................... 12
  Partitioning Techniques .................................................................................................. 12
  Partitioning for manageability and availability............................................................ 13
  Partitioning for performance ......................................................................................... 14
Parallel Execution............................................................................................................... 16
  What is Parallelism? ........................................................................................................ 16
  Shared nothing versus shared everything.................................................................... 17
  Parallel execution working on RAC............................................................................... 18
  Parallel_adaptive_multi_user ......................................................................................... 20
Instance Groups............................................................................................................... 21
Conclusion......................................................................................................................... 22
EXECUTIVE OVERVIEW

A Linux GRID is currently one of the most popular data warehouse implementations. Based on the authors’ extensive experiences in the field of TPC-H and real-world customer benchmarks, this paper describes the top 5 performance tips for tuning a data warehouse on a GRID of Linux clusters:

1. How to choose GRID components to ensure a well balanced system
2. Use a few nodes with many CPUs or use many nodes with a few CPUs?
3. Verify expected IO throughput before installing Oracle
4. How to choose an optimal partitioning strategy
5. How to run parallel operations optimally

INTRODUCTION

An emerging trend in the computing industry is the deployment of data warehouse systems on a “Grid” of industry standard commodity server running the Linux operating system. These low-cost systems enable companies to scale their data warehouse to multiple terabytes while still meeting their IT budgets requirements. However, in order for a Linux Grid configuration to be successful it must achieve the level of performance and reliability required for a data warehousing system.

This paper will help system engineers create reliable and performing Linux RAC systems for data warehouse applications. The paper is organized into five sections. The first section supplies guidelines on setting up hardware. The next section addresses the question of whether to employ many nodes with a small number of CPUs or few nodes with a large number of CPUs. We then describe how to measure the IO throughput of a system using the ORION (Oracle I/O Numbers Calibration Tool) tool. Section four describes how to select an optimal partitioning strategy and finally section five discusses parallel operations.

BALANCING GRID COMPONENTS

The four main components to consider when sizing a grid system are: the total number of CPUs, the number of CPUs per node, the IO subsystem, and the interconnect. The total number of CPUs and the maximum disk throughput usually depend on the application and the number of users. This paper does not
give any advice on how many CPUs are needed or what the total throughput of a system should be. However, the following sections outline the rules to ensure that a grid is well balanced.

**MANY VS. FEW NODES**

The first question that needs to be answered in the planning phase of any grid system is whether to employ many nodes with a small number of CPUs or few nodes with a large number of CPUs. The following sections will outline the merits of both solutions. Like any system design decision, the decision whether to go with many small or few large nodes depends on the system requirements. However, there are three system independent considerations that one should bare in mind: manageability, performance and scalability.

Manageability is becoming very important because of ever increasing system complexity and labor costs associated with system maintenance. Although Oracle addresses system manageability in many different areas, the larger the number of nodes in a grid system, the higher the cost to maintain the system is. This is because of increased upgrade costs and more difficult diagnostic procedures. If an OS or database upgrades becomes necessary these upgrades need to be performed, at least partially, on all nodes. The more nodes in a system, the longer it takes to upgrade the entire system. Gathering diagnostic information, especially performance data is more complex on systems with a large number of nodes, since statistics (e.g., IO statistics or statspack data) need to be gathered, sometimes even simultaneously, on multiple nodes. However there are advantages to system with many small nodes. When performing rolling upgrades, where one system is upgraded at a time, the overall system’s computing capacity is reduced far less on a system with many small nodes compared to a system with few large nodes. For example, on a 16-node system with 2 CPUs each, taking one system off-line reduces the system’s computing capacity by 6% (2/32). In contrast, taking one system off-line in a 4-node system with 8 CPUs each reduces the system’s computing capacity by 25% (8/32). As a side comment, with Oracle’s shared everything architecture, taking nodes off-line does not reduce the IO bandwidth available to the other nodes, nor does it disable access to any portion of the database.

Both systems with few and many number of nodes scale very well due to Oracle’s scalable architecture. A system can be either scaled up or scaled out. Scale-up is achieved by adding devices – typically processors, memory, disks and network interface cards – to an existing system. This is also referred to as vertical scaling or vertical growth. Scale-out refers to adding nodes to an existing system. Traditionally, data warehouse applications have been deployed on scale up (high-end SMP) systems. In recent years, the industry has observed another trend: scale-out configurations. This trend is fueled by lower prices and improved performance and reliability for commodity hardware.
The smallest increment a system can be scaled out is one node. Since Oracle recommends employing only nodes with the same number of CPUs, a system with large nodes scales out in large increments compared to a system with small nodes.

As demonstrated per recently published 4-, 8- and 12-node TPC-H benchmarks, Oracle based grid systems with few and many number of nodes perform very well for data warehouse workloads. There are, however, some performance differences between these systems. Some operations, such as sort or hash-joins, need to ship data from one parallel server to another when run in parallel. For intra-node parallelism this communication is done using shared memory. However, if such operations run across nodes (inter-node parallelism), communication is performed using the Interconnect. Latency and bandwidth of communication done in memory is much better than across an Interconnect. In most cases, operations run using inter-node parallelism when their degree of parallelism is larger than twice the number of CPUs. For instance, in a cluster of 4 nodes with 4 CPUs each, if an operation runs with a DOP of less than or equal to 8, the operation runs on one system, without any inter-node parallelism. If the operation’s DOP is 16 it will run on two systems, with a DOP of 8 on one node and a DOP of 8 on another node. Consequently, the larger the number of CPUs per node, the larger the DOP can be for an operation to run locally on one node. The DOP can be adjusted automatically or manually, depending on the system requirements. More about parallelism will be discussed later in this document.

**SIZING IO COMPONENTS**

There are many components involved in completing a disk IO (from the disk into memory). Each component on the IO path, starting with the disk controller, connections, possibly a switch and the Host Bus Adapter (HBA), needs to be orchestrated to guarantee the desired throughput.

Figure 1 shows a conceptual diagram of a 4-node RAC system. Four servers, equipped with 2 host bus adapters (HBA) are connected to 8 disk arrays using 2 8-port switches. Note that the system needs to be configured such that each node in the RAC configuration can access all disk arrays.

The critical components on the IO path are the HBAs, the fibre channel connections, the fibre channel switch, the controller and the disks. In this configuration, the two HBAs per node can sustain a throughput of about 400MB/s (200MB/s each).

In order to saturate full table scans on all 4 nodes the throughput requirement for each of the fibre channel switch (switch) is about 4 x 200 MB/s = 800 MB/s.
It is important to keep in mind that there is only one HBA from each node connected to each switch. On the other side each switch serves 4 disk arrays. Each fibre channel connection to the disk array is capable of delivering 200MB/s, therefore the maximum throughput from the disk array into the switch is also 800MB/s per switch. When sizing the disk array one must assure that the disks and controllers in each disk array can sustain a total of 200MB/s. This is very important as today’s disks keep increasing in capacity without keeping up with speed.

One cannot build a Terabyte warehouse on 8 x 132GB drives, because it will not be sufficient to deliver the throughput needed. If all components are sized as shown in this example the entire system is theoretically capable of delivering an IO throughput of 1.6 GB/s. Each disk array with one controller each can deliver 200MB/s. Since we have 8 disk arrays, the entire disk subsystem can deliver $8 \times 200 \text{ MB/s} = 1.6 \text{ GB/s}$. Each switch is sized to deliver 800MB/s. Half of the IO throughput gets routed through Switch 1, the other half gets routed through Switch 2, and the total switch throughput is 1.6 GB/s. Each system has 2 HBAs, each capable of delivering 200 MB/s. An IO intensive operation running across all 4 nodes can read in 1.6 GB/s ($4 \times 2 \times 200 \text{ MB/s}$). Using this system a 1 Terabyte table can be scanned in 640 seconds.

**Switch**

The size of the switch is determined by the number of nodes, the HBA cards in each node and the number of disk arrays. Some Fibre channel switches are organized in Quads. These switches are optimized for packets sent between ports connected to the same quad. Since IO activity never occurs between HBAs, each

![Figure 2: Conceptual System Diagram](image-url)
quad should have an equal number of HBA and disk array connections to maximize total switch throughput.

**HBA**

Each node will have one or more HBAs. An HBA is a fiber channel host bus adapter. Each HBA is capable of approximately 200 Mb/sec. Each node should have as many HBA as necessary to satisfy the calculated IO demand. Additional software may be required to take advantage of multiple HBAs. This software will control load balancing over the HBAs and failover capabilities.

**Balance between switch capacity, number of disk arrays and HBAs**

In order to ensure that all components on the IO path are of equal capacity, you need to look at the total capacity of all HBAs, the switch and the disk arrays. At the same time you need to keep system limitations in mind. The number of slots for HBAs, and Interconnect Cards (NICs) are limited, as well as the number of ports on a switch and the number of controllers in a disk array. Note that some systems provide one or two on board NICs that can be used for the public network and heartbeat. To increase HBA capacity, one can also use dual port HBAs.

Let’s assume one has a 16 port fibre-channel switch. The total number of disk controllers and HBAs that can be attached to this configuration cannot exceed 16. Furthermore, the total number of HBAs should always be the same as the number of disk controllers providing that there are sufficient slots for HBAs. The ideal configuration is to have 8 HBAs and 8 disk controllers, yielding a total throughput of 8x200MB=1.6GB/s. It is important that the switch’s capacity is also at least 1.6GB/s. If we reduce the number of HBAs to 4 we can attach more disk arrays, but the total throughput will still be determined by the 4 HBAs, which is about 800MB/s.

**Interconnect**

The Interconnect is the network that connects individual machines to form a grid. When talking about the Interconnect we refer to both the actual wire and the protocol that runs over this wire. On Linux, Oracle supports GigE and Infiniband as wires connecting machines. The protocols that can be run on Ethernet are TCP/IP and UDP. On Infiniband one can run TPC/IP, UDP and uDAPL. The performance of GigE is about 100MB/s and Infiniband is about 1GB/s. The protocols, too, have different performance characteristics. TPC/IP is the easiest and most commonly used protocol in today’s IT infrastructure but it is also the slowest protocol with the largest CPU overhead. UDP has been available for quite some time and has less CPU overhead. The latest protocol available with Oracle, uDAPL, has the least CPU overhead.

In general the Interconnect is being used for three different purposes in Oracle systems. As part of the Oracle’s Cluster Ready Services (CRS) each node periodically sends a heartbeat to the other nodes, notifying them that it is still alive.
It is necessary that these heartbeats reach all other nodes because otherwise the cluster might fall apart (split brain). This is why Oracle recommends designating a dedicated network for the Interconnect, so that the heartbeat messages are reliably transmitted.

The Interconnect is also being used for cache fusion. Cache Fusion is used when accessing the buffer cache in a RAC environment. With Cache Fusion a block that is requested by node A and currently cached on Node B is sent directly using the Interconnect from Node B to Node A.

The largest demand for interconnect traffic in data warehouse applications comes from inter process communication (IPC). When performing join, aggregation or load operations involving multiple nodes it might be necessary to re-distribute data and send control messages from one node to another. Processes, which are also called Parallel Servers, communicate with each other using the Interconnect. The amount of interconnect traffic depends on the operation and the number of nodes participating in the operation. Join operations tend to utilize the interconnect traffic more than simple aggregations because of possible communication between Parallel Servers. The amount of interconnect traffic can vary significantly, depending on the distribution method. Which distribution method is used can be found in the PQ Distrib column of the query plan. Cases where one side of the join is broadcasted or both sides are hash-distributed result in the largest interconnect traffic. Partial Partition Wise Joins in which only one side of the join is redistributed result in less interconnect traffic, while Full Partition Wise Joins in which no side needs to be redistributed result in the least interconnect traffic.

The amount of interconnect traffic also depends on how many nodes participate in a join operation. The more nodes participate a join operation the more data needs to be distributed to remote nodes. For instance, in a 4-node RAC cluster with 4 CPUs each, one needs to set the DOP to 32 on both the external and internal tables to maximize load performance with external tables. This will result in 8 Parallel Servers performing read operations from the external table on each node, as well as 8 Parallel Servers performing table creation statements on each node. On the other hand, if there are 4 users on average on the systems issuing queries, it is very likely that each user’s query runs locally on one node, reducing the number of remote data distribution to zero.

**Tuning the Interconnect Protocols**

In order to minimize the overhead of every message sent across the Interconnect, the message size should be set to a large value. Depending on the Interconnect protocol, message sizes up to 64k are allowed. However, the larger the message size, the more memory is being used to buffer messages. Also, very large messages sizes tend to overemphasize the overhead that originates from sending control messages with the large message size. Tests for data warehouse benchmarks have shown that message sizes between 16k and 32k perform best.
All components of the hardware configuration should support Jumbo frames (NIC’s, switches and storage). Larger frame sizes reduce server overhead and increase throughput. Standard Ethernet frames sizes have been 1500 bytes since it was created in 1980. Jumbo frames extends Ethernet frame sizes to 9000 bytes. It is also possible to use Jumbo Frames with UDP over IP. However, jumbo frames can only be used if every component can support them.

In case of UDP it is possible to use multiple networks for IPC traffic. Simply add multiple IP addresses to the `cluster interconnect` parameter separated by “;” and Oracle will load balance between the different IP addresses. Furthermore it is recommended to tune the send and receive buffers to 256k.

In case of UDP over Infiniband it is recommended to add the following OS parameters to tune UDP traffic.

### Measuring IO Throughput On Oracle GRID

Before installing Oracle on a system one should run a simple IO test to make sure that the throughput from the IO system meets the expectations. A tool that is widely available on Linux systems is `dd`. For more information about how to use `dd` please refer to the article “20TB on a Linux Cluster Today: How to build a multiterabyte data warehouse, using Linux and RAC”. Recently, Oracle made a tool available to predict the performance of an Oracle database without having to install Oracle or create a database. This tool is called ORION (Oracle I/O Numbers Calibration Tool). It is available from the Oracle website at [www.oracle.com](http://www.oracle.com). The following sections assume that you followed Oracle’s SAME methodology for storage configuration.

**ORION**

Orion is a tool for predicting the performance of an Oracle database without having to install Oracle or create a database. Unlike other I/O calibration tools, Orion is expressly designed for simulating Oracle database I/O workloads using the same I/O software stack as Oracle. It can also simulate the effect of striping performed by ASM. The following types of I/O workloads are supported: Small Random I/O, Large Sequential I/O, Large Random I/O and Mixed Workloads. For data warehouse type workloads one should check the system for Large Sequential or Large Random I/Os, depending on the kind of I/O subsystem and the type of application. Large Sequential I/O should be tested for data warehousing applications, data loads, backups, and restores. These applications generate sequential read and write streams composed of multiple outstanding 1 MB I/Os. Such applications are processing large amounts of data, like a whole table or a whole database and they typically care about the overall data throughput in MegaBytes Per Second (MBPS). Orion can simulate a given number of sequential read or write streams of a given I/O size with a given number of outstanding I/Os.

---

1 SAME stands for **Stripe And Mirror Everything.**
Orion can optionally simulate ASM striping when testing sequential streams. Large Random I/O should be tested for applications where a sequential stream typically accesses the disks concurrently with other database traffic. With striping, a sequential stream is spread across many disks. Consequently, at the disk level, multiple sequential streams are seen as random 1 MB I/Os, which we also call Multi-User Sequential I/O.

For each type of workload, Orion can run tests at different levels of I/O load to measure performance metrics like MBPS, IOPS and I/O latency. Load is expressed in terms of the number of outstanding asynchronous I/Os. Internally, for each such load level, the Orion software keeps issuing I/O requests as fast as they complete to maintain the I/O load at that level. For random workloads (large and small), the load level is the number of outstanding I/Os. For large sequential workloads, the load level is a combination of the number of sequential streams and the number of outstanding I/Os per stream. Testing a given workload at a range of load levels helps the user understand how performance is affected by load. Please refer to the ORION manual for more detailed information.

The current version of ORION has been developed to generate and measure workloads on one node. In a grid system, however, the workload is usually issued from multiple nodes. In order to simulate the I/O pattern generated by a data warehouse running on a grid system, multiple copies of ORION need to be started simultaneously on all nodes of the system. Set up each copy of ORION to access the same set of disks in the same way. First, make yourself familiar with the basic setup of ORION following the setup steps from the ORION manual (some of the following steps are also included in the ORION manual).

For testing grid-based data warehouse systems the following ORION parameters apply:

- **size_large**: Size of the I/Os (in KB) for the Large Random or Sequential I/O workload. (Default is 1024).
  
  **Recommendation**: This should be set to the I/O size Oracle will be using, such as: \( \text{block\_size} \times \text{mulliblock\_read\_count} \)

- **type**: Type of the Large I/O workload. (Default is rand):
  
  **rand**: Large Random I/O workload.
  
  **seq**: Large Sequential I/O workload.

  **Recommendation**: rand when using ASM and seq when using hardware striping.

- **num_streamIO**: Number of outstanding I/Os per sequential stream. Only valid for -type seq. (Default is 1).
  
  **Recommendation**: ?? missing text???

- **cache_size**: Size of the storage array's read or write cache (in MB). For Large Sequential I/O workloads, Orion will warm the cache by doing random large I/Os before each data point. It uses the cache size to determine the duration for this cache warming operation. If not specified, warming will occur for a default amount of time. If set to 0, no cache warming will be done. (Default is not specified, which
means warming for a default amount of time).

- **duration**: Duration to test each data point in seconds. (Default is 60).
  **Recommendation**: Depends on the number of nodes. 60 to 120 per node is recommended.

- **matrix**: Type of mixed workloads to test over a range of loads. (Default is detailed).
  - basic: No mixed workload. The Small Random and Large Random/Sequential workloads will be tested separately.
  - detailed: Small Random and Large Random/Sequential workloads will be tested in combination.
  - point: A single data point with $S$ outstanding Small Random I/Os and $L$ outstanding Large Random I/Os or sequential streams. $S$ is set by the num_small parameter. $L$ is set by the num_large parameter.
  - col: Large Random/Sequential workloads only.
  - row: Small Random workloads only.
  - max: Same as detailed, but only tests the workload at the maximum load, specified by the num_small and num_large parameters.
  **Recommendation**: max

- **num_large**: Maximum number of outstanding I/Os for the Large Random I/O workload or number of concurrent large I/Os per stream. Can only be specified when matrix is row, point, or max.
  **Recommendation**: 16

- **verbose**: Prints progress and status information to standard output.

For example the command line for ORION might look something like:

```
orion -run advanced -testname GridTest -num_disks 120
   -size_large 1024 -type rand -matrix max -
   num_large 16
```

In order to better characterize the I/O subsystem’s performance we recommend running the above command on each node individually with a fixed duration parameter first. This will reveal any node specific bottleneck or misconfiguration of the I/O path from each node to the I/O subsystem. The result of the tests will be recorded in the output files. On each system you will find the file GridTest_summary.txt. They are a good starting point for verifying your input parameters and analyzing the output. The files GridTest_* .csv contain comma-separated values for several I/O performance measures. Please refer to the ORION manual for an explanation of the files contents. If there are any node specific problems, fix those first before continuing with any other test.

If the same throughput can be achieved on each node individually, the test should be repeated on all nodes simultaneously. This will reveal any bottleneck that might exist in the switches and IO subsystem. After all tests are done you will need to manually merge the files into a spreadsheet. One should not be surprised if the maximum throughput achieved during the single node test is higher than the maximum throughput achieved on each of the nodes in the multi-node test.
However, the closer the per-node throughput comes to the single node throughput, the better balanced the system is.

The throughput measured with dd and ORION will serve as a target IO rate that can be achieved with Oracle. The IO rate achievable with Oracle should be about 85% of the IO rate achievable with dd and ORION, because Oracle is doing other operations in executing the query besides pure IOs. Figure 8 shows the result of throughput test with dd and Oracle. In this test we vary the number of dds from 1 to 9. As the graph shows the disk throughput increases in steps of about 80MB/s for the first 6 dds before it flattens out at a maximum of about 480 MB/s. In the test with Oracle we increased the degree of parallelism from 1 to 9 for a table that is striped across as many disk as the raw devices of the dd test. As one can see performance follows very closely that of the dd test. However, the total throughput flattens out at about 420 MB/s, at about 85% of the maximal dd throughput.

PARTITIONING

Partitioning is a powerful feature that allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity. Each piece of database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually. This gives the administrator considerable flexibility in managing partitioned objects. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands.

Tables are partitioning using a 'partitioning key', a set of columns, which determines which partition a given row will reside in. Oracle Database 10g Release 2 provides six techniques for partitioning tables.

Partitioning Techniques

• Range Partitioning

Each partition is specified by a range of values (for a table with a date column as the partitioning key, the 'January-2005' partition contains rows with the partitioning-key values from '01-JAN-2005' - '31-JAN-2005')

• List Partitioning

Each partition is specified by a list of values of the partitioning key (for a table with a region column as the partitioning key, the 'North America' partition may contain values 'Canada', 'USA', and 'Mexico')

• Hash Partitioning
A hash algorithm is applied to the partitioning key to determine the partition for a given row

- **Composite Range-Hash Partitioning**

A combination of Range and Hash partitioning techniques, whereby a table is first range-partitioned, and then each individual range-partition is further sub-partitioned using the hash partitioning technique.

- **Composite Range-List Partitioning**

A combination of Range and List partitioning techniques, whereby a table is first range partitioned, and then each individual range-partition is further sub-partitioned using the list partitioning technique.

**Partitioning for manageability and availability**

The decision to partition a table by range or list is normally driven by business needs. This type of partitioning improves the manageability and availability of large volumes of data. Consider the case where two year’s worth of sales data is stored in a table. At the end of each month a new batch of data needs to be loaded into the table and the oldest month of data needs to be removed. If the Sales table is range partitioned, the new month worth of data can be loaded into a temporary table away from any online user activity thus have no impact on query response times. Once the data is loaded and indexed in the temporary it can be made visible to the end user by simply issuing an `alter table exchange partition <partition_name> with table <table_name>` command. This is a sub-second operation and should have little or no impact to end user queries. In order to remove the oldest month of data simply issue an `alter table <table_name> drop partition <partition_name>` command.

If business users predominately accesses the Sales data on a monthly basis, e.g. total sales per month then range partitioning this table by month will ensure that the data is accessed in the most efficient manner, as only 1 partition needs to be scanned to answer the business users query instead of the entire table. This will reduce the amount of data scan from 24 months to just 1 month.

However, if the business users predominately access the data by region e.g. total sales in the northeast, a more efficient access method would be to list partition the data by region, again ensuring that only 1 partition needs to be scanned to answer the business users query instead of the entire table. The ability to avoid scanning irrelevant partitions is known as **partition pruning**.
Partitioning for performance

Hash partitioning is used more for performance reasons. Oracle uses a linear hashing algorithm. In order to ensure that the data gets evenly distributed among the hash partitions it is highly recommended that the number of hash partitions is a power of 2 (for example, 2, 4, 8). A good rule of thumb to follow when deciding the number of hash partitions a table should have is 2 X # of CPUs rounded to up to the nearest power of 2. If your system has 12 CPUs then 32 would be a good number of hash partitions. On a clustered system the same rules apply. If you have 3 nodes each with 4 CPUs then 32 would still be a good number of hash partitions. What happens if you add a node to the cluster? Unlike the shared nothing architecture it is not necessary to repartition the data if a new node is added to the Grid. The number of hash partitions is independent of the number of nodes in the Grid.

One of the main performance benefits of hash partitioning is partition-wise joins. Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. This significantly reduces response time and improves the use of both CPU and memory resources. In a grid data warehouse, this significantly reduces response time by limiting the data traffic over the interconnect (IPC), which is the key to achieving good scalability for massive join operations. Partition-wise joins can be full or partial, depending on the partitioning scheme of the tables to be joined.

A full partition-wise join divides a large join into smaller joins between a pair of partitions from the two joined tables. For the optimizer to choose the full partition-wise join method, both tables must be equi-partitioned on their join keys. That is, they have to be partitioned on the same column with the same partitioning

```sql
select sum(sales_amount) from sales
where sales_date between to_date('01-MAR-2005', 'DD-MON-YYYY')
and to_date('31-MAR-2005', 'DD-MON-YYYY');
```

Figure 3 Partition pruning: In a range partitioned table only the relevant partition is accessed.
method. Parallel execution of a full partition-wise join is similar to its serial execution. Instead of joining one partition pair at a time, multiple partition pairs are joined in parallel by multiple parallel query servers. The number of partitions joined in parallel is determined by the DOP.

![Figure 4: Full Partition-Wise Join](image)

Figure 4 illustrates the parallel execution of a full partition-wise join between two tables, lineitem and orders, on 3 nodes. Both tables have the same degree of parallelism and the same number of partitions. They are range partitioned on a date field and sub partitioned by hash on the order_id field. As illustrated in the picture, each partition pair is read from the database and joined directly. There is no data redistribution necessary, thus minimizing IPC communication, especially across nodes. Defining more partitions than the degree of parallelism may improve load balancing and limit possible skew in the execution. If there are more partitions than parallel query servers, each time one query server completes the join of one pair of partitions, it requests another pair to join. This process repeats until all pairs have been processed. This method enables the load to be balanced dynamically when the number of partition pairs is greater than the degree of parallelism (for example, 128 partitions with a degree of parallelism of 32).

Unlike full partition-wise joins, partial partition-wise joins can be applied if only one table is partitioned on the join key. Hence, partial partition-wise joins are more common than full partition-wise joins. To execute a partial partition-wise join, Oracle Database 10g dynamically repartitions the other table based on the partitioning of the partitioned table. Once the other table is repartitioned, the execution is similar to a full partition-wise join. The redistribution operation involves exchanging rows between parallel execution servers. This operation leads to interconnect traffic in grid environments, since data needs to be repartitioned across node boundaries.
Figure 5: Partial Partition-Wise Join

Figure 5 illustrates a partial partition-wise join. It uses the same example as in Figure 4, except that the orders table is not partitioned. Before the join operation is executed, the rows from the orders table are dynamically redistributed on the join key as illustrated by the arrow.

PARALLEL EXECUTION

What is Parallelism?

Parallelism is the idea of breaking down a single task into multiple smaller, distinct parts. Instead of one process doing all the work, the task can be parallelized, having multiple processes working concurrently on the smaller units. For example when using parallelism, a terabyte of data can be processed in minutes, not hours or days. Parallelism achieves this performance by leveraging all of the hardware resources available: multiple CPUs, multiple IO channels, multiple storage arrays and disk drives, and large volumes of memory. This leads to tremendous performance improvements and optimal system utilization.

With Oracle’s dynamic parallel execution framework, all data is shared between the nodes in the cluster. The decision for parallelizing and dividing the work into smaller units is not restricted to any predetermined static data distribution done at database setup (creation) time. In other words the number of nodes in the Grid does not determine the number of parallel execution servers allocated to a query. Every query has its own characteristics of accessing, joining, and processing different portions of data. Consequently, each SQL statement undergoes an optimization and parallelization process when it is parsed. When the data changes, if a more optimal execution or parallelization plan becomes available, or you simply add a new node to the system, Oracle can automatically adapt to the new situation. This provides the highest degree of flexibility for parallelizing any kind of operation.
As soon as the statement is optimized and parallelized at parse time, all subsequent parallel subtasks are known. The original process becomes the query coordinator; parallel execution servers (PX servers) are allocated from the common pool of parallel execution servers on one or more nodes and start working in parallel on the operation. Each parallel execution server works independently on its personal subset of data, and the inter-process communication is minimal.

**Shared nothing versus shared everything**

Traditionally, two approaches have been used for the implementation of parallel execution of database systems. The main differentiation is whether or not the physical data layout is used as a base – and static pre-requisite – for dividing, thus parallelizing, the work.

In pure shared nothing database architectures, database files have to be partitioned on the nodes of a multi-computer system to enable parallel processing. Each node ‘owns’ a subset of the data and all access to this data is performed exclusively by the owning node, using a single process or thread with no provision for intra-partition parallelism.

Conceptually, it is useful to think of a pure shared nothing system as being very similar to a distributed database. A transaction executing on a given node has to send messages to other nodes that own the data being accessed and coordinate the work done on the other nodes, to perform the required read/write activity.

With Oracle’s dynamic parallel execution framework, all data is shared, and the decision for parallelizing and dividing the work into smaller units is not restricted to any predetermined static data distribution done at database setup (creation) time.

However, unlike the shared nothing architecture, each SQL statement executed in parallel is optimized without the need to take any physical database layout restrictions into account. This enables the most optimal data subsetting for each parallel execution, thus providing equal and, in most cases, even better scalability and performance than pure shared nothing architectures. Subsequent steps of a parallel operation are combined and processed by one Parallel Execution server whenever beneficial, reducing the necessity of function and/or data shipping even more.
We will now step through a simple example, demonstrating the differences and the superiority of Oracle’s dynamic parallel execution: In some respects, the architectures of static parallelism versus dynamic parallelism echo a basic problem from queuing theory. For example, consider what happens when you go to a post office. The post office has multiple windows, each staffed by a postal worker.

One approach is to have customers like you line up in front of every window, and wait until the clerk for that window can help them. Another approach is to have a single line for all windows, and the customer at the front of the line goes to the first available window.

In many respects, static parallelism is analogous to the first approach, while dynamic parallelism is analogous to the second. As everyone who has waited in lines knows, having a single line generally minimizes the average wait-time. The analogy is that the postal windows are available parallel execution servers, and the customers are the individual portions of database operations that need to be completed.

In a dynamic model, the query coordinator doles out database operations to the first available parallel execution server. Before a query starts, there is no way to tell which parallel execution server will complete which operation. In a static model, the work is distributed to all of the parallel execution servers at the beginning of the query.

In fact, the static parallelism approach takes this one step further. The static parallelism approach not only requires one customer line for each window, but also specifies that a given customer can only line up in front of a specific window (just as a given set of data can only be accessed by a single node). That is, the post office has one window for all customers with last names starting with the letters A through the letter D, and another window for all customers with last names starting with the letters E through H, and so on. If a customer named Mr. Adams is standing in a long line in the A-D window and notices that the E-H line is empty, Mr. Adams cannot switch lines because the A-D window is the only window that is allowed to assist him. What if even the clerk of the A-D window becomes sick or needs some additional information available at Window J-L? How does the office being run with the static parallelism approach handle peaks like seasonal events?

Which post office would you visit?

**Parallel execution working on RAC**

In Real Application Clusters, Oracle determines at runtime whether it will run parallel execution server processes on only one instance, or whether it will run processes on multiple instances. In general, Oracle tries to use only one instance when sufficient resources are available. This reduces the cross-instance message traffic and synchronization. If you had a two-node cluster with each node having 4 CPUs, a typical Oracle configuration would be to run one Oracle instance on each node, both of which access the same database. The number of parallel execution
servers on each node would be 8 (2 X # of CPUs) by default, giving a total of 16 for the cluster. The equivalent setup for a shared nothing system would be to have 8 virtual nodes – and data partitions – per node, a total of 16 partitions. In the shared nothing environment every object has to be partitioned that way if you want it to access it in parallel. Furthermore all of these objects then have to be accessed with a parallel degree of 16, involving all nodes of the cluster, all the time.

Oracle on the other hand makes intelligent decisions in Real Application Clusters environments with regard to intra-node and inter-node parallelism. In intra-node parallelism for example, if a user A connected to node 1 of the cluster and issued a SQL statement that requires a parallel degree of six and if six parallel execution servers are idle on the local node (node 1: the node which the user is connected to), the SQL statement is processed using local resources (figure 7). This eliminates query coordination overhead across multiple nodes. The shared nothing system has to execute all SQL statements with 64 parallel processes to be able to access all the data, involving all nodes, and forcing communication between all nodes.

Continuing with this example: a second user B connects to Node 2 and issues a SQL statement that requires a parallel degree of four. Again there are enough idle parallel execution servers on the local node (Node 2) so this SQL statement will be processed using local resources (figure 8).

![Figure 7: User A connects to node 1 and issues a query with DOP 6](image)
Finally a third user C connects to Node 2 and issues a SQL statement that requires a parallel degree of six. There are only two parallel execution servers idle on the local node (node 2): those two parallel execution servers and four of the parallel execution servers of the other node (node 1) are used to complete the SQL statement; Oracle uses inter-node parallelism (figure 9). The shared nothing system still has to use all 64 processes on all nodes.

**Figure 8 User B connects to node 2 and issues a query with DOP 4**

**Figure 9 User C connects to node 2 and issues a query with DOP 6**
In other words if the entire parallel execution server group fits on one node, the load balancing algorithm places all the processes on a single node to avoid communications overhead.

**Parallel_adaptive_multi_user**

To optimize execution time for all users and to utilize all system resources, Oracle Database 10g’s dynamic parallelism allows for adjusting the degree of parallelism before each operation starts (Adaptive Multiuser Algorithm). The algorithm automatically reduces the requested degree of parallelism based on the system load at query startup time. The adaptive multiuser algorithm has several inputs. The algorithm first considers the number of active Oracle Server processes as calculated by Oracle. The algorithm then considers the default settings for parallelism as set in the initialization parameter file, as well as parallelism options used in CREATE TABLE and ALTER TABLE statements and SQL hints.

For instance, using a 16-node 4-CPU-per-node grid, when the first user enters the system, it will be granted a DOP of 128. The next user entering the system will be given a DOP of 64, the next 32, and so on. If the system settles into a steady state of, say, 16 concurrent users, all users will be given a DOP of 8, thus dividing the system evenly among all the parallel users. The algorithm is controlled by PARALLEL_ADAPTIVE_MULTI_USER initialization parameter, which is set to TRUE by default.

**Instance Groups**

Oracle Database 10g also offers a more precise method for controlling which instances allocate parallel execution server processes within a grid environment. This feature is called instance groups. It allows you to logically group different instances together and perform parallel operations upon all of the associated instances at once. Instance groups can also be used to effectively partition resources for a specific purpose, such as ETL, batch processing or parallel query.

Each active instance can be assigned to at least one or more instance groups. When a particular instance group is activated, parallel operations will only spawn parallel processes on instances in that group.

Instance group membership can be established on an instance-by-instance basis by setting the instance_groups initialization parameter to a name representing one or more instance groups. For example, on a 4-node system that has both ETL processes and end-user queries running on it, half the nodes can be assigned to one department and the other half to the other one.

The instance group feature can be controlled using initialization parameters.

Example
Use the following parameter syntax in the initialization parameter file to assign node one and two to the ETL group:

```sql
sid[1].INSTANCE_GROUPS=ETL
sid[2].INSTANCE_GROUPS=ETL
```

Then assign nodes three and four to end-user queries, using the following syntax in the parameter file:

```sql
sid[3].INSTANCE_GROUPS=END_USER
sid[4].INSTANCE_GROUPS=END_USER
```

Activate the nodes owned by the ETL process to spawn a parallel execution server process by entering the following:

```sql
ALTER SESSION SET PARALLEL_INSTANCE_GROUP = 'ETL';
```

CONCLUSION

Low-cost Linux Grid systems enable companies to scale their data warehouse to multiple terabytes while still staying within their IT budgets. However in order for a Linux, Grid configuration to be successful it must achieve the level of performance and reliability required for a data warehousing system. When designing and implementing a Linux you should keep 5 key points in mind.
1. Choose GRID components so that the system is well balanced
2. Use a few nodes with many CPUs or use many nodes with a few CPUs?
3. Verify expected IO throughput before installing Oracle
4. Choose an optimal partitioning strategy
5. Run parallel operations optimally