Question 1 : How to do performance tunning?

Answer:

Performance tuning in Oracle involves optimizing the performance of your database system to ensure it runs efficiently and meets the desired performance goals. Here are some steps you can follow to perform performance tuning in Oracle:

1. Identify the performance issue: Determine the specific problem areas, such as slow queries, high CPU usage, or excessive I/O.
2. Collect performance data: Use Oracle's performance monitoring tools, such as Oracle Enterprise Manager or Automatic Workload Repository (AWR), to collect data on system performance, including CPU usage, memory usage, disk I/O, and query execution times.
3. Analyze the data: Review the collected performance data to identify any bottlenecks or areas of concern. Look for queries with high resource consumption, long execution times, or excessive I/O operations.
4. Optimize SQL queries: Identify and optimize poorly performing SQL queries. This can involve rewriting queries, adding or modifying indexes, or using hints to guide the optimizer.
5. Optimize database configuration: Review and adjust database configuration parameters, such as memory allocation, buffer cache size, and parallelism settings, to optimize performance.
6. Monitor and tune memory usage: Ensure that the database is properly utilizing available memory resources. Adjust memory settings, such as the shared pool and PGA (Program Global Area), to optimize memory usage.
7. Monitor and tune disk I/O: Analyze disk I/O patterns and identify any I/O bottlenecks. Consider techniques such as partitioning, table/index reorganization, or using faster storage devices to improve disk I/O performance.
8. Monitor and tune concurrency: Identify and address any contention issues related to locks, latches, or other concurrency-related problems. Adjust locking mechanisms or implement appropriate isolation levels to improve concurrency.
9. Monitor and tune network performance: Analyze network traffic and identify any network-related performance issues. Optimize network configuration, such as adjusting TCP/IP settings or using faster network connections, if necessary.
10. Regularly review and maintain performance: Continuously monitor and review the performance of your Oracle database system. Regularly analyze performance data, identify new performance issues, and implement appropriate tuning measures.

Remember, performance tuning is an iterative process, and it may require multiple iterations to achieve optimal performance. It's also recommended to consult Oracle documentation, seek expert advice, or consider engaging Oracle support for complex performance tuning scenarios.

Question: What is materialized view, kindly explain it with e.g.?

In Oracle, a materialized view is a database object that stores the results of a query in a physical table. Unlike standard views, which are virtual and don't store data themselves, materialized views contain precomputed data and act as a snapshot of the underlying data at a specific point in time. Materialized views can significantly improve query performance and reduce the workload on the database, especially when dealing with complex and time-consuming queries.

When a materialized view is created, it stores the result set of a specific query, and the data in the view is refreshed periodically based on a defined schedule or manually triggered. This refresh process updates the materialized view with the latest data from the underlying tables, ensuring that the data in the view remains up-to-date.

Example of creating a materialized view in Oracle:

Suppose we have two tables, "orders" and "order\_items," that store information about customer orders and the items in those orders, respectively. We want to create a materialized view that displays the total revenue for each order.

1. Create the base tables and insert some sample data:

sqlCopy code

CREATE TABLE orders ( order\_id NUMBER PRIMARY KEY, order\_date DATE, customer\_id NUMBER ); CREATE TABLE order\_items ( order\_item\_id NUMBER PRIMARY KEY, order\_id NUMBER, item\_name VARCHAR2(50), quantity NUMBER, price NUMBER ); INSERT INTO orders VALUES (1, '2023-08-01', 1001); INSERT INTO orders VALUES (2, '2023-08-02', 1002); INSERT INTO order\_items VALUES (101, 1, 'Product A', 2, 50); INSERT INTO order\_items VALUES (102, 1, 'Product B', 3, 30); INSERT INTO order\_items VALUES (103, 2, 'Product C', 1, 40);

1. Create the materialized view:

sqlCopy code

CREATE MATERIALIZED VIEW order\_revenue\_mv REFRESH FAST ON COMMIT AS SELECT o.order\_id, o.order\_date, SUM(oi.quantity \* oi.price) AS total\_revenue FROM orders o JOIN order\_items oi ON o.order\_id = oi.order\_id GROUP BY o.order\_id, o.order\_date;

In this example, we created a materialized view named "order\_revenue\_mv." The "REFRESH FAST ON COMMIT" clause indicates that the materialized view will be updated automatically whenever a committed transaction modifies the data in the underlying tables. The SELECT statement calculates the total revenue for each order by joining the "orders" and "order\_items" tables and grouping the results by order\_id and order\_date.

Now, whenever new orders or order items are inserted, updated, or deleted in the base tables, the materialized view "order\_revenue\_mv" will automatically reflect the changes, providing an efficient way to retrieve the total revenue for each order without recalculating it from scratch each time.

Question 3: what is different method to refresh the materialised views?

Answer:

In Oracle, there are different methods to refresh materialized views, which determine how and when the data in the materialized view is updated with the latest data from the underlying tables. The choice of refresh method depends on factors such as the type of materialized view, the data volume, and the desired level of data freshness. Here are the common methods for refreshing materialized views in Oracle:

1. Complete Refresh:

* The complete refresh method recreates the entire materialized view from scratch.
* All existing data in the materialized view is deleted, and the query defining the materialized view is re-executed to fetch the latest data from the base tables.
* This method ensures the materialized view is always up-to-date, but it may be time-consuming and resource-intensive, especially for large datasets.

Syntax for complete refresh:

sqlCopy code

DBMS\_MVIEW.REFRESH('materialized\_view\_name', 'C');

1. Fast Refresh:

* The fast refresh method updates only the changed data in the materialized view, based on the changes that occurred in the base tables since the last refresh.
* This method is significantly faster than the complete refresh, especially when dealing with large datasets and complex queries.
* To enable fast refresh, specific conditions must be met: the materialized view must have a materialized view log on the base table, and the query used to create the materialized view must meet certain criteria.

Syntax for fast refresh:

sqlCopy code

DBMS\_MVIEW.REFRESH('materialized\_view\_name', 'F');

1. Force Refresh:

* The force refresh method tries to perform a fast refresh if possible. If the fast refresh is not possible (e.g., due to missing materialized view logs), it falls back to a complete refresh.
* This method allows you to attempt a fast refresh first and then use a complete refresh as a backup option.

Syntax for force refresh:

sqlCopy code

DBMS\_MVIEW.REFRESH('materialized\_view\_name', 'C', '', TRUE);

1. On Demand Refresh:

* On demand refresh method allows you to refresh the materialized view manually whenever needed.
* You can trigger the refresh using SQL commands or using Oracle Enterprise Manager (OEM).

Syntax for on-demand refresh:

sqlCopy code

DBMS\_MVIEW.REFRESH('materialized\_view\_name');

1. Refresh On Commit:

* The refresh on commit method automatically refreshes the materialized view whenever a transaction commits changes to the base tables.
* This method provides real-time data freshness for the materialized view but may introduce some overhead on write operations due to the additional refresh process.

To enable refresh on commit, it should be specified when creating the materialized view:

sqlCopy code

CREATE MATERIALIZED VIEW materialized\_view\_name REFRESH FAST ON COMMIT AS SELECT ...

The appropriate refresh method for a materialized view depends on the specific use case, data volume, and desired data freshness. Fast refresh is generally preferred when possible, as it minimizes the overhead of refreshing and provides improved performance compared to a complete refresh. However, fast refresh requires additional setup (materialized view logs) and may have certain restrictions on the underlying query structure.

Question : What is ref cursor in Oracles and its usage and types?

Answer:

In Oracle, a REF CURSOR (or Cursor Variable) is a data type that allows you to reference a result set or query result within PL/SQL code. It is a powerful feature that provides a way to pass query results between different program units (such as procedures and functions) and allows dynamic SQL operations.

Usage of REF CURSOR: REF CURSOR is typically used in scenarios where you want to:

1. Return a result set from a stored procedure or function: Instead of using a fixed return type, a REF CURSOR can be used to return variable result sets, making the stored procedure or function more flexible.
2. Pass query results to other program units: You can pass a REF CURSOR as a parameter between procedures or functions, allowing you to work with query results in different parts of your application.
3. Execute dynamic SQL: REF CURSOR can be used to execute dynamic SQL statements, where the exact query to be executed is not known at compile time.

Types of REF CURSOR:

1. Strongly Typed REF CURSOR: A strongly typed REF CURSOR is associated with a specific predefined or user-defined data type. The data type defines the structure of the result set that the REF CURSOR can hold. You can use a strongly typed REF CURSOR when you know the structure of the result set at the time of writing the code.

Example of a strongly typed REF CURSOR:

sqlCopy code

TYPE EmployeeCurTyp IS REF CURSOR RETURN employees%ROWTYPE;

1. Weakly Typed REF CURSOR: A weakly typed REF CURSOR is not associated with any specific data type. It can hold query results with varying structures, providing more flexibility in terms of the result sets it can handle. Weakly typed REF CURSOR is often used when the structure of the result set is not known at compile time or when dealing with dynamic SQL.

Example of a weakly typed REF CURSOR:

sqlCopy code

TYPE AnyCurTyp IS REF CURSOR;

Usage of REF CURSOR in a stored procedure:

Here's an example of how to use a REF CURSOR in a stored procedure to return a result set:

sqlCopy code

CREATE OR REPLACE PROCEDURE GetEmployeesByDepartment( department\_id NUMBER, emp\_cur OUT SYS\_REFCURSOR ) AS BEGIN OPEN emp\_cur FOR SELECT \* FROM employees WHERE department\_id = department\_id; END;

In this example, the procedure **GetEmployeesByDepartment** accepts a department ID as input and returns a REF CURSOR (**emp\_cur**) as output. The cursor variable holds the result of the SELECT query, which retrieves all employees from the specified department.

You can then use this procedure in PL/SQL or from other programming languages (e.g., Java, Python) to fetch the result set and process the data accordingly.

Overall, REF CURSOR is a valuable feature in Oracle that provides flexibility and power when dealing with query results and dynamic SQL operations within PL/SQL code.

Question : what is collections in Oracle and its usage?

In Oracle, collections are composite data types that allow you to store multiple elements of the same or different data types in a single variable. Collections are useful when you need to work with sets of data, such as arrays or lists, within PL/SQL code. There are three main types of collections in Oracle:

1. Nested Tables: Nested tables are one-dimensional, unbounded, and dynamic arrays. They can hold a varying number of elements, and their size can be changed during runtime. Nested tables can be sparse, meaning not all elements need to be initialized, and they are stored in a database table.

Example of a nested table type and usage:

sqlCopy code

-- Define a nested table type TYPE EmpListType IS TABLE OF employees%ROWTYPE; -- Declare a nested table variable emp\_list EmpListType; -- Populate the nested table with data SELECT \* BULK COLLECT INTO emp\_list FROM employees WHERE department\_id = 10; -- Loop through the elements in the nested table FOR i IN emp\_list.FIRST..emp\_list.LAST LOOP DBMS\_OUTPUT.PUT\_LINE(emp\_list(i).first\_name || ' ' || emp\_list(i).last\_name); END LOOP;

1. VARRAYs (Variable-Size Arrays): VARRAYs are one-dimensional arrays with a fixed size specified during the type definition. They behave like regular arrays, and their size cannot be changed after creation. VARRAYs are stored in the database and are useful when you know the exact number of elements needed.

Example of a VARRAY type and usage:

sqlCopy code

-- Define a VARRAY type TYPE EmpIDListType IS VARRAY(10) OF NUMBER; -- Declare a VARRAY variable emp\_ids EmpIDListType; -- Assign values to the VARRAY emp\_ids := EmpIDListType(101, 102, 103, 104); -- Access elements in the VARRAY DBMS\_OUTPUT.PUT\_LINE(emp\_ids(1)); -- Output: 101 DBMS\_OUTPUT.PUT\_LINE(emp\_ids(3)); -- Output: 103

1. Associative Arrays (also known as INDEX BY Tables): Associative arrays are one-dimensional, unbounded, and index-based collections. They are similar to hash tables and are defined by a data type for the index and another data type for the values. Associative arrays do not have a fixed size, and their index can be of various data types like NUMBER or VARCHAR2.

Example of an associative array and usage:

sqlCopy code

-- Define an associative array type TYPE EmpSalaryType IS TABLE OF NUMBER INDEX BY VARCHAR2(30); -- Declare an associative array variable emp\_salaries EmpSalaryType; -- Populate the associative array with data emp\_salaries('John') := 50000; emp\_salaries('Alice') := 60000; emp\_salaries('Bob') := 55000; -- Access elements in the associative array DBMS\_OUTPUT.PUT\_LINE(emp\_salaries('John')); -- Output: 50000 DBMS\_OUTPUT.PUT\_LINE(emp\_salaries('Alice')); -- Output: 60000

Collections are widely used in PL/SQL for various purposes, including holding query results, passing data between stored procedures and functions, and performing operations on sets of data efficiently. They provide a flexible and powerful mechanism to work with arrays and lists within the Oracle database environment.

Question:   
What is different types of Partitioning in Oracle?

Answer: In Oracle, partitioning is a database feature that allows you to divide large tables or indexes into smaller, more manageable segments called partitions. Each partition acts as a separate sub-table or sub-index, and data is distributed across these partitions based on a specified partitioning key. Partitioning provides significant benefits in terms of performance, maintenance, and manageability for large datasets. There are several types of partitioning available in Oracle:

1. **Range Partitioning**: In range partitioning, data is divided based on a range of values in the partitioning key column. Each partition represents a specific range of values. For example, you can partition a sales table based on the sales date, where each partition holds data for a specific time range (e.g., monthly or yearly).
2. **List Partitioning**: List partitioning involves dividing data into partitions based on specific values in the partitioning key column. Each partition corresponds to a list of values. For instance, you can partition a customer table based on the customer's country, where each partition contains data for customers from a specific country.
3. **Hash Partitioning**: In hash partitioning, data is distributed among partitions based on a hash function applied to the partitioning key. This method evenly distributes the data across partitions, and the data distribution is not influenced by the values in the partitioning key column. It is useful for load balancing and high parallelism.
4. **Interval Partitioning**: Interval partitioning is an enhancement to range partitioning, introduced in Oracle 11g. It allows automatic creation of new partitions as data is inserted based on predefined intervals. For example, you can set up interval partitioning on a time-based table, and new partitions will be created automatically for each time interval (e.g., daily or monthly).
5. **Composite Partitioning**: Composite partitioning combines two or more partitioning methods to create more complex partitioning schemes. For instance, you can use range-list partitioning, where data is first divided into ranges, and then each range is further subdivided into lists.
6. **Reference Partitioning**: Reference partitioning is used when you have a parent-child relationship between two tables. The child table's partitioning is determined by the partitioning key in the parent table.
7. **Virtual Column-Based Partitioning**: This type of partitioning uses a virtual (computed) column as the partitioning key. The virtual column's value is calculated based on other columns' values, and data is partitioned accordingly.

Each type of partitioning has its advantages and is suitable for different scenarios, depending on the nature of data, the queries involved, and the specific performance and maintenance requirements. Partitioning can significantly enhance the performance of queries, data loading, and maintenance operations on large tables and indexes in Oracle databases.

Question : What is clustering in Oracle?

In Oracle, clustering refers to a method of organizing data within a table based on a specific column or set of columns. It is different from partitioning, which divides the data into segments (partitions) based on a partitioning key. Clustering arranges the data physically on disk so that rows with similar values in the clustering column(s) are stored together in the same data blocks or close to each other.

The purpose of clustering is to improve the performance of queries that often access rows with similar values in the clustering column(s). When a table is clustered, data retrieval for certain types of queries can be faster because related rows are physically located near each other, reducing disk I/O and improving read efficiency.

Clustering in Oracle is achieved using a feature called the Cluster Index. A cluster index is a type of index that is created on the clustering column(s) of the table. This index determines the physical order of the rows in the table, and it also contains pointers to the actual rows in the clustered table.

It's important to note that clustering is not always the best solution for improving performance. It is most effective when the following conditions are met:

1. The table is frequently queried using the clustering column(s) in the WHERE clause.
2. The clustering column(s) have a limited number of distinct values, meaning that there are relatively few clusters.
3. The clustering column(s) are not updated frequently, as this could cause the rows to be scattered and reduce the benefits of clustering.

Creating a clustered table and cluster index in Oracle involves the following steps:

1. Create the cluster:

sqlCopy code

CREATE CLUSTER cluster\_name (clustering\_column datatype, ...);

1. Create the table that will be clustered:

sqlCopy code

CREATE TABLE table\_name ( column1 datatype, column2 datatype, ..., CLUSTER cluster\_name (clustering\_column) );

1. Create a cluster index on the clustering column:

sqlCopy code

CREATE INDEX index\_name ON CLUSTER cluster\_name;

Oracle will automatically maintain the physical order of rows within the cluster as data is inserted, updated, or deleted. Queries that use the clustering column(s) in the WHERE clause will benefit from the physical arrangement of the data, resulting in improved query performance. However, it is essential to carefully consider the use cases and performance characteristics before deciding to implement clustering in Oracle, as it may not be suitable for all scenarios.