22 Managing Hash Clusters

Hash clusters can improve the performance of data retrieval.

- About Hash Clusters
  Storing a table in a hash cluster is an optional way to improve the performance of data retrieval. A hash cluster provides an alternative to a non-clustered table with an index or an index cluster.
- When to Use Hash Clusters
  You can decide when to use hash clusters by contrasting situations where hashing is most useful against situations where there is no advantage. If you find your decision is to use indexing rather than hashing, then you should consider whether to store a table individually or as part of a cluster.
- Creating Different Types of Hash Clusters
  You can use the `CREATE CLUSTER` statement with the `HASHKEYS` clause to create different types of hash clusters.
- Altering Hash Clusters
  You can alter a hash cluster with the `ALTER CLUSTER` statement.
- Dropping Hash Clusters
  You can drop a hash cluster using the `DROP CLUSTER` statement.
- Hash Clusters Data Dictionary Views
  You can query a set of data dictionary views for information about hash clusters.

22.1 About Hash Clusters

Storing a table in a hash cluster is an optional way to improve the performance of data retrieval. A hash cluster provides an alternative to a non-clustered table with an index or an index cluster.

With an indexed table or index cluster, Oracle Database locates the rows in a table using key values that the database stores in a separate index. To use hashing, you create a hash cluster and load tables into it. The database physically stores the rows of a table in a hash cluster and retrieves them according to the results of a **hash function**.

Oracle Database uses a hash function to generate a distribution of numeric values, called **hash values**, that are based on specific cluster key values. The key of a hash cluster, like the key of an index cluster, can be a single column or composite key (multiple column key). To find or store a row in a hash cluster, the database applies the hash function to the cluster key value of the row. The resulting hash value corresponds to a data block in the cluster, which the database then reads or writes on behalf of the issued statement.

To find or store a row in an indexed table or cluster, a minimum of two (there are usually more) I/Os must be performed:

- One or more I/Os to find or store the key value in the index
- Another I/O to read or write the row in the table or cluster

In contrast, the database uses a hash function to locate a row in a hash cluster; no I/O is required. As a result, a minimum of one I/O operation is necessary to read or write a row in a hash cluster.

> **See Also:**
>
> Managing Space for Schema Objects is recommended reading before attempting tasks described in this chapter.

22.2 When to Use Hash Clusters

You can decide when to use hash clusters by contrasting situations where hashing is most useful against situations where there is no advantage. If you find your decision is to use indexing rather than hashing, then you should consider whether to store a table individually or as part of a cluster.

> **Note:**
>
> Even if you decide to use hashing, a table can still have separate indexes on any columns, including the cluster key.

- Situations Where Hashing Is Useful
  Hashing is useful when most queries are equality queries on the cluster key and the tables in the hash cluster are primarily static in size.
- Situations Where Hashing Is Not Advantageous
  Hashing is not advantageous in certain situations.

22.2.1 Situations Where Hashing Is Useful

Hashing is useful when most queries are equality queries on the cluster key and the tables in the hash cluster are primarily static in size.

Hashing is useful when you have the following conditions:

- Most queries are equality queries on the cluster key:

  ```
  SELECT ... WHERE cluster_key = ...;
  ```

  In such cases, the cluster key in the equality condition is hashed, and the corresponding hash key is usually found with a single read. In comparison, for an indexed table the key value must first be found in the index (usually several reads), and then the row is read from the table (another read).
- The tables in the hash cluster are primarily static in size so that you can determine the number of rows and amount of space required for the tables in the cluster. If tables in a hash cluster require more space than the initial allocation for the cluster, performance degradation can be substantial because overflow blocks are required.

22.2.2 Situations Where Hashing Is Not Advantageous

Hashing is not advantageous in certain situations.

Hashing is not advantageous in the following situations:

- Most queries on the table retrieve rows over a range of cluster key values. For example, in full table scans or queries such as the following, a hash function cannot be used to determine the location of specific hash keys. Instead, the equivalent of a full table scan must be done to fetch the rows for the query.

  ```
  SELECT . . . WHERE cluster_key < . . . ;
  ```

  With an index, key values are ordered in the index, so cluster key values that satisfy the `WHERE` clause of a query can be found with relatively few I/Os.
- The table is not static, but instead is continually growing. If a table grows without limit, the space required over the life of the table (its cluster) cannot be predetermined.
- Applications frequently perform full-table scans on the table and the table is sparsely populated. A full-table scan in this situation takes longer under hashing.
- You cannot afford to preallocate the space that the hash cluster will eventually need.

22.3 Creating Different Types of Hash Clusters

You can use the `CREATE CLUSTER` statement with the `HASHKEYS` clause to create different types of hash clusters.

- Creating Hash Clusters
  You create a hash cluster using a `CREATE CLUSTER` statement, but you specify a `HASHKEYS` clause.
- Creating a Sorted Hash Cluster
  A **sorted hash cluster** stores the rows corresponding to each value of the hash function in such a way that the database can efficiently return them in sorted order. For applications that always consume data in sorted order, sorted hash clusters can retrieve data faster by minimizing logical I/Os.
- Creating Single-Table Hash Clusters
  You can create a **single-table hash cluster**, which provides fast access to rows in a table. However, this table must be the only table in the hash cluster.
- Controlling Space Use Within a Hash Cluster
  When creating a hash cluster, it is important to choose the cluster key correctly and set the `HASH IS`, `SIZE`, and `HASHKEYS` parameters so that performance and space use are optimal. The following guidelines describe how to set these parameters.
- Estimating Size Required by Hash Clusters
  As with index clusters, it is important to estimate the storage required for the data in a hash cluster.

22.3.1 Creating Hash Clusters

You create a hash cluster using a `CREATE CLUSTER` statement, but you specify a `HASHKEYS` clause.

The following statement creates a cluster named `trial_cluster`, clustered by the `trialno` column (the cluster key):

```
CREATE CLUSTER trial_cluster ( trialno NUMBER(5,0) )
    TABLESPACE users
    STORAGE ( INITIAL 250K
              NEXT 50K
              MINEXTENTS 1
              MAXEXTENTS 3
              PCTINCREASE 0 )
    HASH IS trialno
    HASHKEYS 150;
```

The following statement creates the `trial` table in the `trial_cluster` hash cluster:

```
CREATE TABLE trial (
    trialno NUMBER(5,0) PRIMARY KEY,
    ... )
    CLUSTER trial_cluster (trialno);
```

As with index clusters, the key of a hash cluster can be a single column or a composite key (multiple column key). In the preceding example, the key is the `trialno` column.

The `HASHKEYS` value, in this case `150`, specifies and limits the number of unique hash values that the hash function can generate. The database rounds the number specified to the nearest prime number.

If no `HASH IS` clause is specified, then the database uses an internal hash function. If the cluster key is already a unique identifier that is uniformly distributed over its range, then you can bypass the internal hash function and specify the cluster key as the hash value, as in the preceding example. You can also use the `HASH IS` clause to specify a user-defined hash function.

You cannot create a cluster index on a hash cluster, and you need not create an index on a hash cluster key.

See Also:

Managing Clusters for additional information about creating tables in a cluster, guidelines for setting parameters of the `CREATE CLUSTER` statement common to index and hash clusters, and the privileges required to create any cluster

22.3.2 Creating a Sorted Hash Cluster

A **sorted hash cluster** stores the rows corresponding to each value of the hash function in such a way that the database can efficiently return them in sorted order. For applications that always consume data in sorted order, sorted hash clusters can retrieve data faster by minimizing logical I/Os.

Assume that a telecommunications company stores detailed call records for a fixed number of originating telephone numbers through a telecommunications switch. From each originating telephone number there can be an unlimited number of calls.

The application stores calls records as calls are made. Each call has a detailed call record identified by a timestamp. For example, the application stores a call record with timestamp 0, then a call record with timestamp 1, and so on.

When generating bills for each originating phone number, the application processes them in first-in, first-out (FIFO) order. The following table shows sample details for three originating phone numbers:

telephone_number	call_timestamp
6505551212	0, 1, 2, 3, 4, ...
6505551213	0, 1, 2, 3, 4, ...
6505551214	0, 1, 2, 3, 4, ...

In the following SQL statements, the `telephone_number` column is the hash key. The hash cluster is sorted on the `call_timestamp` and `call_duration` columns. The example uses the same names for the clustering and sorting columns in the table definition as in the cluster definition, but this is not required. The number of hash keys is based on 10-digit telephone numbers.

```
CREATE CLUSTER call_detail_cluster (
   telephone_number NUMBER,
   call_timestamp   NUMBER SORT,
   call_duration    NUMBER SORT )
  HASHKEYS 10000
  HASH IS telephone_number
  SIZE 256;

CREATE TABLE call_detail (
   telephone_number     NUMBER,
   call_timestamp       NUMBER   SORT,
   call_duration        NUMBER   SORT,
   other_info           VARCHAR2(30) )
  CLUSTER call_detail_cluster (
   telephone_number, call_timestamp, call_duration );
```

Example 22-1   Data Inserted in Sequential Order

Suppose that you seed the `call_detail` table with the rows in FIFO order as shown in this example.

```
INSERT INTO call_detail VALUES (6505551212, 0, 9, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 1, 17, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 2, 5, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 3, 90, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 0, 35, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 1, 6, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 2, 4, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 3, 4, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 0, 15, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 1, 20, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 2, 1, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 3, 25, 'misc info');
COMMIT;
```

Example 22-2 Querying call_detail

In this example, you `SET AUTOTRACE ON`, and then query the `call_detail` table for the call details for the phone number `6505551212`.

```
SQL> SET AUTOTRACE ON;
SQL> SELECT * FROM call_detail WHERE telephone_number = 6505551212;

TELEPHONE_NUMBER CALL_TIMESTAMP CALL_DURATION OTHER_INFO
---------------- -------------- ------------- ------------------------------
      6505551212              0             9 misc info
      6505551212              1            17 misc info
      6505551212              2             5 misc info
      6505551212              3            90 misc info

Execution Plan
----------------------------------------------------------
Plan hash value: 2118876266

----------------------------------------------------------------------
| Id  | Operation         | Name        | Rows  | Bytes | Cost (%CPU)|
----------------------------------------------------------------------
|   0 | SELECT STATEMENT  |             |     1 |    56 |      0   (0)|
|*  1 |  TABLE ACCESS HASH| CALL_DETAIL |     1 |    56 |             |
----------------------------------------------------------------------
```

The query retrieves the rows ordered by timestamp even though no sort appears in the query plan.

Suppose you then delete the existing rows and insert the same rows out of sequence:

```
DELETE FROM call_detail;
INSERT INTO call_detail VALUES (6505551213, 3, 4, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 0, 15, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 0, 9, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 1, 20, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 2, 1, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 1, 6, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 2, 4, 'misc info');
INSERT INTO call_detail VALUES (6505551214, 3, 25, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 1, 17, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 2, 5, 'misc info');
INSERT INTO call_detail VALUES (6505551212, 3, 90, 'misc info');
INSERT INTO call_detail VALUES (6505551213, 0, 35, 'misc info');
COMMIT;
```

If you rerun the same query of `call_detail`, the database again retrieves the rows in sorted order even though no `ORDER BY` clause is specified. No `SORT ORDER BY` operation appears in the query plan because the database performs an internal sort.

Now assume that you create a nonclustered table `call_detail_nonclustered` and then load it with the same sample values in Example 22-1. To retrieve the data in sorted order, you must use an `ORDER BY` clause as follows:

```
SQL> SELECT * FROM call_detail_nonclustered WHERE telephone_number = 6505551212
  2  ORDER BY call_timestamp, call_duration;

TELEPHONE_NUMBER CALL_TIMESTAMP CALL_DURATION OTHER_INFO
---------------- -------------- ------------- ------------------------------
```

```
      6505551212             0             9 misc info
      6505551212             1            17 misc info
      6505551212             2             5 misc info
      6505551212             3            90 misc info

Execution Plan
----------------------------------------------------------
Plan hash value: 2555750302

----------------------------------------------------------------------------------
|Id| Operation          | Name                     |Rows|Bytes|Cost (%CPU)|Time   |
----------------------------------------------------------------------------------
| 0| SELECT STATEMENT   |                          | 4  | 224 | 4 (25)| 00:00:01 |
| 1|  SORT ORDER BY     |                          | 4  | 224 | 4 (25)| 00:00:01 |
|*2|   TABLE ACCESS FULL| CALL_DETAIL_NONCLUSTERED | 4  | 224 | 3  (0)| 00:00:01 |
----------------------------------------------------------------------------------
```

The preceding plan shows that in the nonclustered case the sort is more expensive than in the clustered case. The rows, bytes, cost, and time are all greater in the case of the table that is not stored in a sorted hash cluster.

22.3.3 Creating Single-Table Hash Clusters

You can create a **single-table hash cluster**, which provides fast access to rows in a table. However, this table must be the only table in the hash cluster.

Essentially, there must be a one-to-one mapping between hash keys and data rows. The following statement creates a single-table hash cluster named `peanut` with the cluster key `variety`:

```
CREATE CLUSTER peanut (variety NUMBER)
   SIZE 512 SINGLE TABLE HASHKEYS 500;
```

The database rounds the `HASHKEYS` value up to the nearest prime number, so this cluster has a maximum of 503 hash key values, each of size 512 bytes. The `SINGLE TABLE` clause is valid only for hash clusters. `HASHKEYS` must also be specified.

See Also:

Oracle Database SQL Language Reference for the syntax of the `CREATE CLUSTER` statement

22.3.4 Controlling Space Use Within a Hash Cluster

When creating a hash cluster, it is important to choose the cluster key correctly and set the `HASH IS`, `SIZE`, and `HASHKEYS` parameters so that performance and space use are optimal. The following guidelines describe how to set these parameters.

- Choosing the Key
  Choosing the correct cluster key is dependent on the most common types of queries issued against the clustered tables.
- Setting HASH IS
  Specify the `HASH IS` parameter only if the cluster key is a single column of the `NUMBER` data type, and contains uniformly distributed integers.

- Setting SIZE
  `SIZE` should be set to the average amount of space required to hold all rows for any given hash key.
- Setting HASHKEYS
  Specify the `HASHKEYS` clause to create a hash cluster and specify the number of hash values for the hash cluster.
- Controlling Space in Hash Clusters
  Examples illustrate how to correctly choose the cluster key and set the `HASH IS`, `SIZE`, and `HASHKEYS` parameters. For all examples, assume that the data block size is 2K and that on average, 1950 bytes of each block is available data space (block size minus overhead).

22.3.4.1 Choosing the Key

Choosing the correct cluster key is dependent on the most common types of queries issued against the clustered tables.

For example, consider the `emp` table in a hash cluster. If queries often select rows by employee number, the `empno` column should be the cluster key. If queries often select rows by department number, the `deptno` column should be the cluster key. For hash clusters that contain a single table, the cluster key is typically the entire primary key of the contained table.

The key of a hash cluster, like that of an index cluster, can be a single column or a composite key (multiple column key). A hash cluster with a composite key must use the internal hash function of the database.

22.3.4.2 Setting HASH IS

Specify the `HASH IS` parameter only if the cluster key is a single column of the `NUMBER` data type, and contains uniformly distributed integers.

If these conditions apply, you can distribute rows in the cluster so that each unique cluster key value hashes, with no collisions (two cluster key values having the same hash value), to a unique hash value. If these conditions do not apply, omit this clause so that you use the internal hash function.

22.3.4.3 Setting SIZE

`SIZE` should be set to the average amount of space required to hold all rows for any given hash key.

Therefore, to properly determine `SIZE`, you must be aware of the characteristics of your data:

- If the hash cluster is to contain only a single table and the hash key values of the rows in that table are unique (one row for each value), `SIZE` can be set to the average row size in the cluster.
- If the hash cluster is to contain multiple tables, `SIZE` can be set to the average amount of space required to hold all rows associated with a representative hash value.

Further, once you have determined a (preliminary) value for `SIZE`, consider the following. If the `SIZE` value is small (more than four hash keys can be assigned for each data block) you can use this value for `SIZE` in the `CREATE CLUSTER` statement. However, if the value of `SIZE` is large (four or fewer hash keys can be assigned for each data block), then you should also consider the expected frequency of collisions and whether performance of data retrieval or efficiency of space usage is more important to you.

- If the hash cluster does not use the internal hash function (if you specified `HASH IS`) and you expect few or no collisions, you can use your preliminary value of `SIZE`. No collisions occur and space is used as efficiently as possible.
- If you expect frequent collisions on inserts, the likelihood of overflow blocks being allocated to store rows is high. To reduce the possibility of overflow blocks and maximize performance when collisions are frequent, you should adjust `SIZE` as shown in the following chart.

Available Space for each Block / Calculated SIZE	Setting for SIZE
1	`SIZE`
2	`SIZE` + 15%
3	`SIZE` + 12%
4	`SIZE` + 8%
>4	`SIZE`

Overestimating the value of `SIZE` increases the amount of unused space in the cluster. If space efficiency is more important than the performance of data retrieval, disregard the adjustments shown in the preceding table and use the original value for `SIZE`.

22.3.4.4 Setting HASHKEYS

Specify the `HASHKEYS` clause to create a hash cluster and specify the number of hash values for the hash cluster.

For maximum distribution of rows in a hash cluster, the database rounds the `HASHKEYS` value up to the nearest prime number.

22.3.4.5 Controlling Space in Hash Clusters

Examples illustrate how to correctly choose the cluster key and set the `HASH IS`, `SIZE`, and `HASHKEYS` parameters. For all examples, assume that the data block size is 2K and that on average, 1950 bytes of each block is available data space (block size minus overhead).

- Controlling Space in Hash Clusters: Example 1
  An example illustrates controlling space in hash clusters.
- Controlling Space in Hash Clusters: Example 2
  An example illustrates controlling space in hash clusters.

22.3.4.5.1 Controlling Space in Hash Clusters: Example 1

An example illustrates controlling space in hash clusters.

You decide to load the `emp` table into a hash cluster. Most queries retrieve employee records by their employee number. You estimate that the maximum number of rows in the `emp` table at any given time is 10000 and that the average row size is 55 bytes.

In this case, `empno` should be the cluster key. Because this column contains integers that are unique, the internal hash function can be bypassed. `SIZE` can be set to the average row size, 55 bytes. Note that 34 hash keys are assigned for each data block. `HASHKEYS` can be set to the number of rows in the table, 10000. The database rounds this value up to the next highest prime number: 10007.

```
CREATE CLUSTER emp_cluster (empno
NUMBER)
. . .
SIZE 55
HASH IS empno HASHKEYS 10000;
```

22.3.4.5.2 Controlling Space in Hash Clusters: Example 2

An example illustrates controlling space in hash clusters.

In this example, conditions are similar to the example in "Controlling Space in Hash Clusters: Example 1 ". In this case, however, rows are usually retrieved by department number. At most, there are 1000 departments with an average of 10 employees for each department. Department numbers increment by 10 (0, 10, 20, 30, . . .).

In this case, `deptno` should be the cluster key. Since this column contains integers that are uniformly distributed, the internal hash function can be bypassed. A preliminary value of `SIZE` (the average amount of space required to hold all rows for each department) is 55 bytes * 10, or 550 bytes. Using this value for `SIZE`, only three hash keys can be assigned for each data block. If you expect some collisions and want maximum performance of data retrieval, slightly alter your estimated `SIZE` to prevent collisions from requiring overflow blocks. By adjusting `SIZE` by 12%, to 620 bytes (see "Setting SIZE"), there is more space for rows from expected collisions.

`HASHKEYS` can be set to the number of unique department numbers, 1000. The database rounds this value up to the next highest prime number: 1009.

```
CREATE CLUSTER emp_cluster (deptno NUMBER)
. . .
SIZE 620
HASH IS deptno HASHKEYS 1000;
```

22.3.5 Estimating Size Required by Hash Clusters

As with index clusters, it is important to estimate the storage required for the data in a hash cluster.

Oracle Database guarantees that the initial allocation of space is sufficient to store the hash table according to the settings `SIZE` and `HASHKEYS`. If settings for the storage parameters `INITIAL`, `NEXT`, and `MINEXTENTS` do not account for the hash table size, incremental (additional) extents are allocated until at least `SIZE*HASHKEYS` is reached. For example, assume that the data block size is 2K, the available data space for each block is approximately 1900 bytes (data block size minus overhead), and that the `STORAGE` and `HASH` parameters are specified in the `CREATE CLUSTER` statement as follows:

```
STORAGE (INITIAL 100K
    NEXT 150K
    MINEXTENTS 1
    PCTINCREASE 0)
SIZE 1500
HASHKEYS 100
```

In this example, only one hash key can be assigned for each data block. Therefore, the initial space required for the hash cluster is at least 100*2K or 200K. The settings for the storage parameters do not account for this requirement. Therefore, an initial extent of 100K and a second extent of 150K are allocated to the hash cluster.

Alternatively, assume the `HASH` parameters are specified as follows:

```
SIZE 500 HASHKEYS 100
```

In this case, three hash keys are assigned to each data block. Therefore, the initial space required for the hash cluster is at least 34*2K or 68K. The initial settings for the storage parameters are sufficient for this requirement (an initial extent of 100K is allocated to the hash cluster).

22.4 Altering Hash Clusters

You can alter a hash cluster with the `ALTER CLUSTER` statement.

For example, the following `ALTER CLUSTER` statement alters the `emp_dept` cluster:

```
ALTER CLUSTER emp_dept . . . ;
```

The implications for altering a hash cluster are identical to those for altering an index cluster, described in "Altering Clusters". However, the `SIZE`, `HASHKEYS`, and `HASH IS` parameters cannot be specified in an `ALTER CLUSTER` statement. To change these parameters, you must re-create the cluster, then copy the data from the original cluster.

22.5 Dropping Hash Clusters

You can drop a hash cluster using the `DROP CLUSTER` statement.

For example, the following `DROP CLUSTER` statement drops the `emp_dept` cluster:

```
DROP CLUSTER emp_dept;
```

A table in a hash cluster is dropped using the `DROP TABLE` statement. The implications of dropping hash clusters and tables in hash clusters are the same as those for dropping index clusters.

See Also:

"Dropping Clusters"

22.6 Hash Clusters Data Dictionary Views

You can query a set of data dictionary views for information about hash clusters.

The following views display information about hash clusters:

View	Description

`DBA_CLUSTERS`
`ALL_CLUSTERS`
`USER_CLUSTERS`
	`DBA` view describes all clusters (including hash clusters) in the database. `ALL` view describes all clusters accessible to the user. `USER` view is restricted to clusters owned by the user. Some columns in these views contain statistics that are generated by the `DBMS_STATS` package or `ANALYZE` statement.

`DBA_CLU_COLUMNS` `USER_CLU_COLUMNS`	These views map table columns to cluster columns.

View	Description
DBA_CLUSTER_HASH_EXPRESSIONS ALL_CLUSTER_HASH_EXPRESSIONS USER_CLUSTER_HASH_EXPRESSIONS	These views list hash functions for hash clusters.