CREATE INDEX

Use the CREATE INDEX statement to create a new index for one or more columns in a table, a functional value on one or more columns, and, optionally, to cluster the physical table in the order of the index.

When more than one columns or functions are listed, the concatenation of the set of columns is treated as a single composite column for indexing. The indexes can be fragmented into separate dbspaces. You can create a unique or duplicate index, and you can set the object mode of either type of index.

Syntax
**CREATE INDEX**

### Usage

A secondary access method (sometimes referred to as an index access method) is a set of server functions that build, access, and manipulate an index structure such as a B-tree, R-tree, or an index structure that a DataBlade module provides. Typically, a secondary access method speeds up the retrieval of data.

Use `CREATE INDEX` to create the following types of indexes:

- Column index
- Functional index
  
  You can create a functional index on the resulting values of a function on one or more columns. For more information, see "Function Specification" on page 1-141.

When you issue the `CREATE INDEX` statement, the table is locked in exclusive mode. If another process is using the table, the database server cannot execute the `CREATE INDEX` statement and returns an error.

For the different secondary access methods that Universal Server provides, see "USING Clause" on page 1-148.

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Restrictions</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbspace</code></td>
<td>The name of the dbspace in which you want to place the index</td>
<td>The dbspace must exist at the time you execute the statement.</td>
<td>Identifier, p. 1-962</td>
</tr>
<tr>
<td><code>percent</code></td>
<td>The percentage of each index page that is filled by index data when the index is created. The default value is 90.</td>
<td>Value must be in the range 1 to 100. Fillfactor does not apply to an R-tree secondary access method.</td>
<td>Literal Number, p. 1-997</td>
</tr>
</tbody>
</table>
**CREATE INDEX**

**UNIQUE and DISTINCT Options**

The following example creates a unique index:

```
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
```

A unique index prevents duplicates in the `customer_num` column. A column with a unique index can have, at most, one null value. The DISTINCT keyword is a synonym for the keyword UNIQUE, so the following statement would accomplish the same task:

```
CREATE DISTINCT INDEX c_num_ix ON customer (customer_num)
```

The index in either example is maintained in ascending order, which is the default order.

If you do not specify the UNIQUE or DISTINCT keywords in a `CREATE INDEX` statement, a duplicate index is created. A duplicate index allows duplicate values in the indexed column.

You can also prevent duplicates in a column or set of columns by creating a unique constraint with the `CREATE TABLE` or `ALTER TABLE` statement. See the `CREATE TABLE` or `ALTER TABLE` statements for more information on creating unique constraints.

**How Unique and Referential Constraints Affect Indexes**

The database server creates internal B-tree indexes for unique and referential constraints. If a unique or referential constraint is added after the table is created, the user-created indexes are used, if appropriate. An appropriate index is one that indexes the same columns that are used in the referential or unique constraint. If an appropriate index is not available, a nonfragmented index is created in the database dbspace.

**CLUSTER Option**

Use the CLUSTER option to reorder the physical table in the order designated by the index. The `CREATE CLUSTER INDEX` statement fails if a CLUSTER index already exists.

```
CREATE CLUSTER INDEX c_clust_ix ON customer (zipcode)
```
This statement creates an index on the `customer` table that orders the table physically by zip code.

If the CLUSTER option is specified in addition to fragments on an index, the data is clustered only within the context of the fragment and not globally across the entire table.

**Warning:** Some secondary access methods (such as R-tree) do not support clustering. Before you specify CLUSTER for your index, be sure that it uses an access method that supports clustering.

**Index Definition**

Use the Index Definition portion of the CREATE INDEX statement to give a name to the index, to specify the table on which the index is created, the value or values to use for the index key, and, optionally, the secondary access method.
CREATE INDEX

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Restrictions</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>table name</td>
<td>The name of the table on which the index is created</td>
<td>The table must exist. The table can be a regular database table or a temporary table. This table cannot be an external table.</td>
<td>Table Name, p. 1-1044</td>
</tr>
<tr>
<td>secondary access method</td>
<td>The name of the secondary access method used with the index you are creating.</td>
<td>The access method can be a B-tree, R-tree, or an access method that has been defined by a DataBlade module. The access method must be a valid access method in the sysams system catalog table. The default secondary access method is B-tree. If the access method is B-tree, you can create only one index for each unique combination of ascending and descending columnar or functional keys with operator classes. This restriction does not apply to other secondary access methods.</td>
<td>Identifier, p. 1-962</td>
</tr>
</tbody>
</table>
**Index Key Specification**

Use the Index Key Specification clause of the CREATE INDEX statement to specify the key value for the index, an operator class, and whether the index will be sorted in ascending or descending order.

### Element | Purpose | Restrictions | Syntax
--- | --- | --- | ---
**column name** | The name of the column or columns that you want to index | You must observe restrictions on the location of the columns, the maximum number of columns, the total width of the columns, existing constraints on the columns, and the number of indexes allowed on the same columns. See “Restrictions on the Column Name Variable in CREATE INDEX” on page 1-140. | Identifier, p. 1-962

**operator class** | The operator class associated with this column or function of the index | If you specify a secondary access method in the USING clause that does not have a default operator class, you must specify an operator class here. If you use an alternative access method, and if the access method has a default operator class, you can omit the operator class here. If you do not specify an operator class and the secondary access method does not have a default operator class, the database server returns an error. | Identifier, p. 1-962
CREATE INDEX

The index key value can be one of the following values:

- One or more columns that contain built-in data types
- One or more columns that contain user-defined data types
- One or more values that a user-defined function returns (referred to as a *functional index*)
- A combination of columns and functions

**Restrictions on the Column Name Variable in CREATE INDEX**

Observe the following restrictions when you specify the *column name* variable:

- All the columns you specify must exist and must belong to the same table—the table being indexed.
- You cannot create an index on a column that belongs to an external table.
- The column you specify cannot be a column whose data type is a collection.
- The maximum number of arguments (columns) you can specify is 16. See “Composite Indexes” on page 1-150.
- You cannot add an ascending index to a column or column list that already has a unique constraint on it. See “ASC and DESC Keywords” on page 1-142.
- The number of indexes you can create on the same column or same sequence of columns is restricted. See “Number of Indexes Allowed” on page 1-151.
Function Specification

This clause specifies the user-defined function whose return value is the key for a functional index.

```
CREATE INDEX
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Restrictions</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>function name</code></td>
<td>The name of the function used as a key to this index</td>
<td>This must be a non-variant function. The return type of the function cannot be BYTE or TEXT. You cannot create an index on built-in algebraic, exponential, log, or hex functions.</td>
<td>Function Name, p. 1-959</td>
</tr>
<tr>
<td><code>column name</code></td>
<td>The name of the column or columns on which the function acts</td>
<td>See “Restrictions on the Column Name Variable in CREATE INDEX” on page 1-140.</td>
<td>Identifier, p. 1-962</td>
</tr>
</tbody>
</table>

You can create an index on an external function or an SPL function. You can also create functional indexes within an SPL routine.

A functional index can be a B-tree index or a user-defined index type provided by a DataBlade module.

Functional indexes are indexed on the value returned by the specified function rather than on the value of a column.

For example, the following statement creates a functional index on table `zones` using the value returned by the function `Area()` as the key:

```sql
CREATE INDEX zone_func_ind ON zones (Area(length,width));
```
CREATE INDEX

Operator Class

An operator class is the set of operators that Universal Server associates with a secondary access method for query optimization and building the index.

Specify an operator class when you create an index if you have one of the following situations:

- There is no default operator class for the secondary access method. For example, some of the DataBlade modules do not provide a default operator class.
- You want to use an operator class that is different from the default operator class that the secondary access method provides.

For more information, see “Default Operator Classes” on page 1-176. The following CREATE INDEX statement creates a B-tree index on the cust_tab table that uses the abs_btree_ops operator class for the cust_num key:

```
CREATE INDEX c_num1_ix ON cust_tab (cust_num abs_btree_ops);
```

ASC and DESC Keywords

Use the ASC option to specify an index that is maintained in ascending order. The ASC option is the default ordering scheme. Use the DESC option to specify an index that is maintained in descending order. When a column or list of columns is defined as unique in a CREATE TABLE or ALTER TABLE statement, the database server implements that UNIQUE CONSTRAINT by creating a unique ascending index. Thus, you cannot use the CREATE INDEX statement to add an ascending index to a column or column list that is already defined as unique.

The ASC and DESC options can be used with B-trees only.
You can create a descending index on such columns, and you can include such columns in composite ascending indexes in different combinations. For example, the following sequence of statements is allowed:

```sql
CREATE TABLE customer (
    customer_num SERIAL(101) UNIQUE,
    fname CHAR(15),
    lname CHAR(15),
    company CHAR(20),
    address1 CHAR(20),
    address2 CHAR(20),
    city CHAR(15),
    state CHAR(2),
    zipcode CHAR(5),
    phone CHAR(18)
);

CREATE INDEX cathtmp ON customer (customer_num DESC)
CREATE INDEX c_temp2 ON customer (customer_num, zipcode)
```

**Bidirectional Traversal of Indexes**

When you create an index on a column but do not specify the ASC or DESC keywords, the database server stores the key values in ascending order by default. If you specify the ASC keyword, the database server stores the key values in ascending order. If you specify the DESC keyword, the database server stores the key values in descending order.

Ascending order means that the key values are stored in order from the smallest key to the largest key. For example, if you create an ascending index on the `lname` column of the `customer` table, last names are stored in the index in the following order: Albertson, Beatty, Currie.

Descending order means that the key values are stored in order from the largest key to the smallest key. For example, if you create a descending index on the `lname` column of the `customer` table, last names are stored in the index in the following order: Currie, Beatty, Albertson.

However, the bidirectional traversal capability of the database server lets you create just one index on a column and use that index for queries that specify sorting of results in either ascending or descending order of the sort column.
CREATE INDEX

Example of Bidirectional Traversal of an Index

An example can help to illustrate the bidirectional traversal of indexes by the database server. Suppose that you want to enter the following two queries:

```sql
SELECT lname, fname FROM customer ORDER BY lname ASC;
SELECT lname, fname FROM customer ORDER BY lname DESC;
```

When you specify the ORDER BY clause in SELECT statements such as these, you can improve the performance of the queries by creating an index on the ORDER BY column. Because of the bidirectional traversal capability of the database server, you only need to create a single index on the `lname` column.

For example, you can create an ascending index on the `lname` column with the following statement:

```sql
CREATE INDEX lname_bothways ON customer (lname ASC)
```

The database server will use the ascending index `lname_bothways` to sort the results of the first query in ascending order and to sort the results of the second query in descending order.

In the first query, you want to sort the results in ascending order. So the database server traverses the pages of the `lname_bothways` index from left to right and retrieves key values from the smallest key to the largest key. The query result is as follows.

<table>
<thead>
<tr>
<th>lname</th>
<th>fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albertson</td>
<td>Frank</td>
</tr>
<tr>
<td>Beatty</td>
<td>Lana</td>
</tr>
<tr>
<td>Currie</td>
<td>Philip</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>Raymond</td>
</tr>
<tr>
<td>Wallack</td>
<td>Jason</td>
</tr>
<tr>
<td>Watson</td>
<td>George</td>
</tr>
</tbody>
</table>

Traversing the index from left to right means that the database server starts at the leftmost leaf node of the index and continues to the rightmost leaf node of the index.
In the second query, you want to sort the results in descending order. So the database server traverses the pages of the `lname_bothways` index from right to left and retrieves key values from the largest key to the smallest key. The query result is as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>First Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watson</td>
<td>George</td>
</tr>
<tr>
<td>Wallack</td>
<td>Jason</td>
</tr>
<tr>
<td>Vector</td>
<td>Raymond</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Currie</td>
<td>Philip</td>
</tr>
<tr>
<td>Beatty</td>
<td>Lana</td>
</tr>
<tr>
<td>Albertson</td>
<td>Frank</td>
</tr>
</tbody>
</table>

Traversing the index from right to left means that the database server starts at the rightmost leaf node of the index and continues to the leftmost leaf node of the index. For an explanation of leaf nodes in indexes, see the *INFORMIX-Universal Server Administrator’s Guide*.

**Choosing an Ascending or Descending Index**

In the preceding example, you created an ascending index on the `lname` column of the `customer` table by specifying the `ASC` keyword in the `CREATE INDEX` statement. Then the database server used this index to sort the results of the first query in ascending order of `lname` values and to sort the results of the second query in descending order of `lname` values. However, you could have achieved exactly the same results if you had created the index as a descending index.

For example, the following statement creates a descending index that the database server can use to process both queries:

```
CREATE INDEX lname_bothways2 ON customer (lname DESC)
```

The resulting `lname_bothways2` index stores the key values of the `lname` column in descending order, from the largest key to the smallest key. When the database server processes the first query, it traverses the index from right to left to perform an ascending sort of the results. When the database server processes the second query, it traverses the index from left to right to perform a descending sort of the results.
CREATE INDEX

So it does not matter whether you create a single-column index as an ascending or descending index. Whichever storage order you choose for an index, the database server can traverse that index in ascending or descending order when it processes queries.

**Use of the ASC and DESC Keywords in Composite Indexes**

If you want to place an index on a single column of a table, you do not need to specify the ASC or DESC keywords because the database server can traverse the index in either ascending or descending order. The database server will create the index in ascending order by default, but the database server can traverse this index in either ascending or descending order when it uses the index in a query.

However, if you create a composite index on a table, the ASC and DESC keywords might be required. For example, if you want to enter a SELECT statement whose ORDER BY clause sorts on multiple columns and sorts each column in a different order, and you want to use an index for this query, you need to create a composite index that corresponds to the ORDER BY columns.

For example, suppose that you want to enter the following query:

```sql
SELECT stock_num, manu_code, description, unit_price
FROM stock
ORDER BY manu_code ASC, unit_price DESC
```

This query sorts first in ascending order by the value of the `manu_code` column and then in descending order by the value of the `unit_price` column.

To use an index for this query, you need to issue a CREATE INDEX statement that corresponds to the requirements of the ORDER BY clause. For example, you can enter either of the following statements to create the index:

```sql
CREATE INDEX stock_idx1 ON stock
    (manu_code ASC, unit_price DESC);

CREATE INDEX stock_idx2 ON stock
    (manu_code DESC, unit_price ASC);
```
Now, when you execute the query, the database server uses the index that you created (either stock_idx1 or stock_idx2) to sort the query results in ascending order by the value of the manu_code column and then in descending order by the value of the unit_price column. If you created the stock_idx1 index, the database server traverses the index from left to right when it executes the query. If you created the stock_idx2 index, the database server traverses the index from right to left when it executes the query.

Regardless of which index you created, the query result is as follows.

<table>
<thead>
<tr>
<th>stock_num</th>
<th>manu_code</th>
<th>description</th>
<th>unit_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ANZ</td>
<td>volleyball</td>
<td>$840.00</td>
</tr>
<tr>
<td>205</td>
<td>ANZ</td>
<td>3 golf balls</td>
<td>$312.00</td>
</tr>
<tr>
<td>110</td>
<td>ANZ</td>
<td>helmet</td>
<td>$244.00</td>
</tr>
<tr>
<td>304</td>
<td>ANZ</td>
<td>watch</td>
<td>$170.00</td>
</tr>
<tr>
<td>301</td>
<td>ANZ</td>
<td>running shoes</td>
<td>$95.00</td>
</tr>
<tr>
<td>310</td>
<td>ANZ</td>
<td>kick board</td>
<td>$84.00</td>
</tr>
<tr>
<td>201</td>
<td>ANZ</td>
<td>golf shoes</td>
<td>$75.00</td>
</tr>
<tr>
<td>313</td>
<td>ANZ</td>
<td>swim cap</td>
<td>$60.00</td>
</tr>
<tr>
<td>6</td>
<td>ANZ</td>
<td>tennis ball</td>
<td>$48.00</td>
</tr>
<tr>
<td>9</td>
<td>ANZ</td>
<td>volleyball net</td>
<td>$20.00</td>
</tr>
<tr>
<td>5</td>
<td>ANZ</td>
<td>tennis racquet</td>
<td>$19.80</td>
</tr>
<tr>
<td>309</td>
<td>HRO</td>
<td>ear drops</td>
<td>$40.00</td>
</tr>
<tr>
<td>302</td>
<td>HRO</td>
<td>ice pack</td>
<td>$4.50</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>SHM</td>
<td>18-spd, assmbld</td>
<td>$685.90</td>
</tr>
<tr>
<td>1</td>
<td>SMT</td>
<td>baseball gloves</td>
<td>$450.00</td>
</tr>
<tr>
<td>6</td>
<td>SMT</td>
<td>tennis ball</td>
<td>$36.00</td>
</tr>
<tr>
<td>5</td>
<td>SMT</td>
<td>tennis racquet</td>
<td>$25.00</td>
</tr>
</tbody>
</table>
CREATE INDEX

The composite index that was used for this query (stock_idx1 or stock_idx2) cannot be used for queries in which you specify the same sort direction for the two columns in the ORDER BY clause. For example, suppose that you want to enter the following queries:

```sql
SELECT stock_num, manu_code, description, unit_price
FROM stock
ORDER BY manu_code ASC, unit_price ASC;

SELECT stock_num, manu_code, description, unit_price
FROM stock
ORDER BY manu_code DESC, unit_price DESC;
```

If you want to use a composite index to improve the performance of these queries, you need to enter one of the following CREATE INDEX statements. You can use either one of the created indexes (stock_idx3 or stock_idx4) to improve the performance of the preceding queries.

```sql
CREATE INDEX stock_idx3 ON stock
(manu_code ASC, unit_price ASC);

CREATE INDEX stock_idx4 ON stock
(manu_code DESC, unit_price DESC);
```

USING Clause

Use the USING clause to specify the secondary access method to use for the new index. A secondary access method is a set of routines that perform all of the operations needed to make an index available to a server, such as create, drop, insert, delete, update, and scan.

Universal Server provides the following secondary access methods:

- The generic B-tree index is the built-in secondary access method. A B-tree index is good for a query that retrieves a range of data values. The database server implements this secondary access method and registers it as `btree` in the system catalog tables of a database.

- The R-tree secondary access method is a registered secondary access method. An R-tree index is good for searches on multi-dimensional data (such as box, circle, and so forth). The database server registers this secondary access method as `rtree` in the system catalog tables of a database.
Important: To use an R-tree index, you must install a spatial DataBlade module such as the 2D DataBlade module, Geodetic DataBlade, or any other 3rd party DataBlade modules that implement the R-tree index. These DataBlade modules implement the R-tree secondary access method.

DataBlade modules might provide other types of secondary access methods. For more information on these other secondary access methods, refer to the DataBlade user guides.

By default, the CREATE INDEX statement creates a generic B-tree index. If you want to create an index with an secondary access method other than B-tree, you must specify that name of the secondary access method in the USING clause.

The following example assumes that the database implements the R-tree index. It creates an R-tree index on the location column that contains a opaque data type, point.

```
CREATE INDEX loc_ix ON TABLE emp (location) USING rtree;
SELECT name FROM emp
  WHERE location N_equator_equals point('500, 0');
```

The sample query has a filter on the location column.

**Index Parameter**

Some DataBlade modules provide indexes that require specific parameters when you create them.
Example of an Index with Parameters

The following CREATE INDEX statement creates an index that uses the secondary access method fulltext, which takes two parameters: WORD_SUPPORT and PHRASE_SUPPORT. It indexes a table t, which has two columns: i, an integer column, and data, a TEXT column.

```
CREATE INDEX tx ON t(data)
USING fulltext (WORD_SUPPORT='PATTERN',
PHRASE_SUPPORT='MAXIMUM');
```

Composite Indexes

A composite index can have up to 16 key parts. An index key part is either a table column or, if the index is a functional index, the result of a function on one or more table columns. A composite index can have any of the following as an index key:

- One or more columns
- One or more values that a user-defined function returns (referred to as a functional index)
- A combination of columns and user-defined functions

---

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Restrictions</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter name</td>
<td>Name of the secondary access method parameter used with this index</td>
<td>The parameter name must be one of the strings allowed for this secondary access method. For more information, refer to the DataBlade module user guide.</td>
<td>Quoted String, p. 1-1010</td>
</tr>
<tr>
<td>parameter value</td>
<td>Value of the specified parameter</td>
<td>The parameter value must be one of the quoted strings or literal numbers allowed for this secondary access method.</td>
<td>Quoted String, p. 1-1010 or Literal Number, p. 1-997</td>
</tr>
</tbody>
</table>
The following example creates a composite index using the `stock_num` and `manu_code` columns of the `stock` table:

```sql
CREATE UNIQUE INDEX st_man_ix ON stock (stock_num, manu_code)
```

The index prevents any duplicates of a given combination of `stock_num` and `manu_code`. The index is in ascending order by default.

The total width of all key parts in a single CREATE INDEX statement cannot exceed 390 bytes. Place key parts in the composite index in the order from most frequently used to least frequently used.

### Number of Indexes Allowed

Restrictions exist on the number of indexes that you can create on the same column or the same sequence of columns.

### Restrictions on the Number of Indexes on a Single Column

You can create only one ascending index and one descending index on a single column. For example, if you wanted to create all possible indexes on the `stock_num` column of the `stock` table, you could create the following indexes:

- The `stock_num_asc` index on the `stock_num` column in ascending order
- The `stock_num_desc` index on the `stock_num` column in descending order

Because of the bidirectional traversal capability of the database server, you do not need to create both indexes in practice. You only need to create one of the indexes. Both of these indexes would achieve exactly the same results for an ascending or descending sort on the `stock_num` column. For further information on the bidirectional traversal capability of the database server, see “Bidirectional Traversal of Indexes” on page 1-143.
CREATE INDEX

Restrictions on the Number of Indexes on a Sequence of Columns

You can create multiple indexes on a sequence of columns, provided that each index has a unique combination of ascending and descending columns. For example, to create all possible indexes on the **stock_num** and **manu_code** columns of the **stock** table, you could create the following indexes:

- The **ix1** index on both columns in ascending order
- The **ix2** index on both columns in descending order
- The **ix3** index on **stock_num** in ascending order and on **manu_code** in descending order
- The **ix4** index on **stock_num** in descending order and on **manu_code** in ascending order

Because of the bidirectional-traversal capability of the database server, you do not need to create these four indexes in practice. You only need to create two indexes:

- The **ix1** and **ix2** indexes achieve exactly the same results for sorts in which the user specifies the same sort direction (ascending or descending) for both columns. Therefore, you only need to create one index of this pair.
- The **ix3** and **ix4** indexes achieve exactly the same results for sorts in which the user specifies different sort directions for the two columns (ascending on the first column and descending on the second column or vice versa). Therefore, you only need to create one index of this pair.

For further information on the bidirectional-traversal capability of the database server, see “Bidirectional Traversal of Indexes” on page 1-143.
FILLFACTOR Clause

Use the FILLFACTOR clause to provide for expansion of a B-tree index at a later date or to create compacted indexes. You provide a percent value ranging from 1 to 100, inclusive. The default percent value is 90.

When the B-tree index is created, Universal Server initially fills only that percentage of the nodes specified with the FILLFACTOR value. If you provide a low percentage value, such as 50, you allow room for growth in your B-tree index. The nodes of the B-tree index initially fill to a certain percentage and contain space for inserts. The amount of available space depends on the number of keys in each page as well as the percentage value. For example, with a 50-percent FILLFACTOR value, the page would be half full and could accommodate doubling in growth. A low percentage value can result in faster inserts and can be used for indexes that you expect to grow.

If you provide a high percentage value, such as 99, your indexes are compacted, and any new index inserts result in splitting nodes. The maximum density is achieved with 100 percent. With a 100-percent FILLFACTOR value, the index has no room available for growth; any additions to the index result in splitting the nodes. A 99-percent FILLFACTOR value allows room for at least one insertion per node. A high percentage value can result in faster selects and can be used for indexes that you do not expect to grow or for mostly read-only indexes.

The FILLFACTOR can also be set as a parameter in the ONCONFIG file. The FILLFACTOR clause on the CREATE INDEX statement overrides the setting in the ONCONFIG file.

For more information about the ONCONFIG file and the parameters you can use with ONCONFIG, see the INFORMIX-Universal Server Administrator’s Guide.
Indexes on Fragmented and Nonfragmented Tables

When you fragment a table and, at a later time, create an index for that table, the index uses the same fragmentation strategy as the table unless you specify otherwise with the FRAGMENT BY EXPRESSION clause or the IN dbspace clause. Any changes to the table fragmentation result in a corresponding change to the index fragmentation.

In Universal Server, all indexes are detached. When indexes are created with a fragmentation strategy or a dbspace is specified in the IN dbspace clause, the indexes are stored in separate dbspaces from the table. If there is no fragmentation scheme and no dbspace is specified in the IN dbspace clause, the index is created in the same dbspace as the table.

For information on the IN dbspace clause, see “IN dbspace Clause”. For information on the FRAGMENT BY EXPRESSION clause, see page 1-155.

IN dbspace Clause

Use the IN dbspace clause to specify the dbspace where you want your index to reside. With this clause, you create a detached index, even though the index is not fragmented. The dbspace that you specify must already exist. If you do not specify the IN dbspace clause, the index is created in the dbspace where the table was created. In addition, if you do not specify the IN dbspace clause, but the underlying table is fragmented, the index is created as a detached index, subject to all the restrictions on fragmented indexes. See page 1-155 for more information about fragmented indexes.

The IN dbspace clause allows you to isolate an index. For example, if the customer table is created in the custdata dbspace, but you want to create an index in a separate dbspace called custind, use the following statements:

```sql
CREATE TABLE customer
  .
  .
  IN custdata EXTENT SIZE 16

CREATE INDEX idx_cust ON customer (customer_num)
  IN custind
```
### FRAGMENT BY EXPRESSION Clause

**Syntax**

```
CREATE INDEX <index_name> ON <table_name> 
  FRAGMENT BY EXPRESSION 
    <frag-expression> IN dbspace <dbspace>, 
    <frag-expression> IN dbspace <dbspace> 
    , \ 
    , \ 
    \, \ 
    \, \ 
REMAINDER IN \ remainder dbspace \ 
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Restrictions</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dbspace&gt;</td>
<td>The dbspace that will contain an index fragment that &lt;frag-expression&gt; defines</td>
<td>You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces. The dbspaces must exist at the time you execute the statement.</td>
<td>Identifier, p. 1-962</td>
</tr>
<tr>
<td>&lt;frag-expression&gt;</td>
<td>An expression that defines a fragment where an index key is to be stored using a range, hash, or arbitrary rule</td>
<td>If you specify a value for remainder dbspace, you must specify at least one fragment expression. If you do not specify a value for remainder dbspace, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values from a single row. The columns contained in a fragment expression must be the same as the indexed columns, or a subset of the indexed columns. No subqueries, stored procedures, current date/time functions, or aggregates are allowed in a fragment expression.</td>
<td>Expression, p. 1-876, and Condition, p. 1-831</td>
</tr>
</tbody>
</table>
CREATE INDEX

You use the FRAGMENT BY EXPRESSION clause to define the expression-based distribution scheme.

In an expression-based distribution scheme, each fragment expression in a rule specifies a dbspace. Each fragment expression within the rule isolates data and aids the database server in searching for index keys. You can specify one of the following rules:

- **Range rule**
  A range rule specifies fragment expressions that use a range to specify which index keys are placed in a fragment, as the following example shows:

  ```
  . . .
  FRAGMENT BY EXPRESSION
  c1 < 100 IN dbsp1,
  c1 >= 100 and c1 < 200 IN dbsp2,
  c1 >= 200 IN dbsp3;
  ```

- **Hash rule**
  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as the following example shows:

  ```
  . . .
  FRAGMENT BY EXPRESSION
  MOD(id_num, 3) = 0 IN dbsp1,
  MOD(id_num, 3) = 1 IN dbsp2,
  MOD(id_num, 3) = 2 IN dbsp3;
  ```

You can specify one or more fragment expressions, but remainder dbspace is optional. If you specify only one fragment expression, remainder dbspace is required. The dbspace specified in remainder dbspace must exist at the time you execute the statement.
Arbitrary rule

An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of OR clauses to group data, as the following example shows:

```sql
FRAGMENT BY EXPRESSION
  zip_num = 95228 OR zip_num = 95443 IN dbsp2,
  zip_num = 91120 OR zip_num = 92310 IN dbsp4,
  REMAINDER IN dbsp5;
```

**Warning:** When you specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the **DBCENTURY** environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the **DBCENTURY** environment variable can affect the distribution scheme and can produce unpredictable results. See the “Informix Guide to SQL: Reference” for more information on the **DBCENTURY** environment variable.

### Creating Index Fragments

When you fragment a table, all indexes for the table become fragmented the same as the table, unless you specify a different fragmentation strategy.

#### Fragmentation of Unique Indexes

You can fragment unique indexes only with a table that uses an expression-based distribution scheme. The columns referenced in the fragment expression must be part of the indexed columns. If your CREATE INDEX statement fails to meet either of these restrictions, the CREATE INDEX fails, and work is rolled back.

#### Fragmentation of System Indexes

System indexes (such as those used in referential constraints and unique constraints) utilize user indexes if they exist. If no user indexes can be utilized, system indexes remain nonfragmented and are moved to the dbspace where the database was created. To fragment a system index, create the fragmented index on the constraint columns, and then add the constraint using the ALTER TABLE statement.
Fragmentation of Indexes on Temporary Tables

You can create explicit temporary tables with the TEMP TABLE clause of the CREATE TABLE statement or with the INTO TEMP clause of the SELECT statement. If you specified more than one dbspace in the DBSPACETEMP environment variable, but you did not specify an explicit fragmentation strategy, the database server fragments the temporary table round-robin across the dbspaces that DBSPACETEMP specifies.

If you then try to create a unique index on the temporary table, but you do not specify a fragmentation strategy for the index, the index is not fragmented in the same way as the table. You can fragment a unique index only if the underlying table uses an expression-based distribution scheme, but the temporary table is fragmented according to a round-robin distribution scheme.

Instead of fragmenting the unique index on the temporary table, the database server creates the index in the first dbspace that the DBSPACETEMP environment variable specifies. To avoid this result, use the FRAGMENT BY EXPRESSION clause to specify a fragmentation strategy for the index.

For more information on the DBSPACETEMP environment variable, see the Informix Guide to SQL: Reference.

Object Modes for Unique Indexes

<table>
<thead>
<tr>
<th>Object Modes for Unique Indexes</th>
<th>DISABLED</th>
<th>ENABLED</th>
<th>FILTERING</th>
<th>WITHOUT ERROR</th>
<th>WITH ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENABLED</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>FILTERING</td>
<td>ENABLED</td>
<td>FILTERING</td>
<td>WITHOUT ERROR</td>
<td>WITH ERROR</td>
<td>-----------</td>
</tr>
<tr>
<td>WITHOUT ERROR</td>
<td>FILTERING</td>
<td>WITHOUT ERROR</td>
<td>ENABLED</td>
<td>FILTERING</td>
<td>-----------</td>
</tr>
<tr>
<td>WITH ERROR</td>
<td>FILTERING</td>
<td>WITH ERROR</td>
<td>ENABLED</td>
<td>FILTERING</td>
<td>-----------</td>
</tr>
</tbody>
</table>
You can set unique indexes in the following modes: disabled, enabled, and filtering. The following list explains these modes.

<table>
<thead>
<tr>
<th>Object Mode</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>A unique index created in disabled mode is not updated after insert, delete, and update operations that modify the base table. Because the contents of the disabled index are not up to date, the optimizer does not use the index during the execution of queries.</td>
</tr>
<tr>
<td>enabled</td>
<td>A unique index created in enabled mode is updated after insert, delete, and update operations that modify the base table. Because the contents of the enabled index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a unique enabled index, the statement fails.</td>
</tr>
<tr>
<td>filtering</td>
<td>A unique index created in filtering mode is updated after insert, delete, and update operations that modify the base table. Because the contents of the filtering mode index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a unique index in filtering mode, the statement continues processing, but the bad row is written to the violations table associated with the base table. Diagnostic information about the unique-index violation is written to the diagnostics table associated with the base table.</td>
</tr>
</tbody>
</table>

If you specify filtering mode, you can also specify one of the following error options.

<table>
<thead>
<tr>
<th>Error Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>WITHOUT ERROR</td>
<td>When a unique-index violation occurs during an insert or update operation, no integrity-violation error is returned to the user. You can specify this option only with the filtering-object mode.</td>
</tr>
<tr>
<td>WITH ERROR</td>
<td>When a unique-index violation occurs during an insert or update operation, an integrity-violation error is returned to the user. You can specify this option only with the filtering-object mode.</td>
</tr>
</tbody>
</table>
Specifying Object Modes for Unique Indexes

You must observe the following rules when you specify object modes for unique indexes in CREATE INDEX statements:

- You can set a unique index to the enabled, disabled, or filtering modes.
- If you do not specify the object mode of a unique index explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering-mode unique index, the default error option is WITHOUT ERROR.
- When you add a new unique index to an existing base table and specify the disabled object mode for the index, your CREATE INDEX statement succeeds even if duplicate values in the indexed column would cause a unique-index violation.
- When you add a new unique index to an existing base table and specify the enabled or filtering-object mode for the index, your CREATE INDEX statement succeeds provided that no duplicate values exist in the indexed column that would cause a unique-index violation. However, if any duplicate values exist in the indexed column, your CREATE INDEX statement fails and returns an error.
- When you add a new unique index to an existing base table in the enabled or filtering mode, and duplicate values exist in the indexed column, erroneous rows in the base table are not filtered to the violations table. Thus, you cannot use a violations table to detect the erroneous rows in the base table.
**Adding a Unique Index When Duplicate Values Exist in the Column**

If you attempt to add a unique index in the enabled mode but receive an error message because duplicate values are in the indexed column, take the following steps to add the index successfully:

1. Add the index in the disabled mode. Issue the CREATE INDEX statement again, but this time specify the DISABLED keyword.

2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.

3. Issue a SET statement to switch the object mode of the index to the enabled mode. When you issue this statement, existing rows in the target table that violate the unique-index requirement are duplicated in the violations table. However, you receive an integrity-violation error message, and the index remains disabled.

4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.

5. Take corrective action on the rows in the target table that violate the unique-index requirement.

6. After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled index to the enabled mode. This time the index is enabled, and no integrity violation error message is returned because all rows in the target table now satisfy the new unique-index requirement.
Object Modes for Duplicate Indexes

If you create a duplicate index, you can set the object mode of the index to the disabled or enabled mode. The following table explains these modes.

<table>
<thead>
<tr>
<th>Object Mode</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>A duplicate index is created in disabled mode. The disabled index is not updated after insert, delete, and update operations that modify the base table. Because the contents of the disabled index are not up to date, the optimizer does not use the index during the execution of queries.</td>
</tr>
<tr>
<td>enabled</td>
<td>A duplicate index is created in enabled mode. The enabled index is updated after insert, delete, and update operations that modify the base table. Because the contents of the enabled index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a duplicate enabled index, the statement does not fail.</td>
</tr>
</tbody>
</table>

Specifying Object Modes for Duplicate Indexes

You must observe the following rules when you specify object modes for duplicate indexes in CREATE INDEX statements:

- You can set a duplicate index to the enabled or disabled mode, but you cannot set a duplicate index to the filtering mode.
- If you do not specify the object mode of a duplicate index explicitly, the default mode is enabled.
How the Database Server Treats Disabled Indexes

Whether a disabled index is a unique or duplicate index, the database server effectively ignores the index during data-manipulation operations.

When an index is disabled, the database server stops updating it and stops using it during queries, but the catalog information about the disabled index is retained. So you cannot create a new index on a column or set of columns if a disabled index on that column or set of columns already exists.

Similarly, you cannot create an active (not disabled) unique, foreign-key, or primary-key constraint on a column or set of columns if the indexes needed by the active constraint exist and are disabled.

References

See the ALTER INDEX, CREATE OPCLASS, DROP INDEX, and CREATE TABLE statements in this manual.

For a more detailed description of the different types of indexes, refer to Chapter 3 of the INFORMIX-Universal Server Performance Guide. For information about when to use the different types of indexes and other performance issues with indexes, refer to Chapter 4 of the INFORMIX-Universal Server Performance Guide.

For information about operator classes, refer to the CREATE OPCLASS statement and the Extending INFORMIX-Universal Server: Data Types manual.

For information about the GLS aspects of the CREATE INDEX statement, refer to the Guide to GLS Functionality.

For information about the indexes provided by DataBlade modules, refer to your DataBlade module user’s guide.