CHAPTER 15

Online Utilities

ONLINE REORGANIZATION

DB2 UDB v8 has the capability of reorganizing both tables and indexes online; that is, while users have full read or write access to these objects. Along with an enhanced `REORG TABLE` command, there is a new `REORG INDEXES` command.

Table Reorganization

The performance of your database is significantly impacted by the physical distribution of the table data. After many transactions, logically sequential data might no longer reside on physically sequential data pages, making additional read operations to access data necessary. Table reorganization corrects this problem while reclaiming space.

There are two main approaches that you can use to determine whether a table needs to be reorganized:

- *Invoke the runstats utility*, which collects a variety of statistics pertaining to data distribution, space utilization, prefetch efficiency, indexes, and so on. Because the optimizer uses these statistics when determining access paths to the data, simply invoking the `RUNSTATS` command to refresh statistical information could improve performance. If you regularly invoke the `RUNSTATS` command and analyze the resulting statistics (by querying the database catalog), you can determine whether table or index reorganization is necessary.
- *Invoke the reorgchk utility*, which returns some of the catalog statistics and recommendations about whether tables or indexes need to be reorganized.

You can query the database catalog to determine whether a table needs to be reorganized:

- Query the OVERFLOW column in SYSCAT.TABLES to monitor the number of rows that no longer fit on their original data page.
- Query the FPAGES and NPAGES columns in SYSCAT.TABLES to estimate the number of empty pages in a table. Subtract the NPAGES value (the number of pages that contain rows) from the FPAGES value (the number of pages that are in use) to obtain an estimate of the number of empty pages in the table.
- Query the CLUSTERRATIO column in SYSCAT.INDEXES to determine how well the data in the table is clustered.
- Query the AVERAGE_SEQUENCE_FETCH_PAGES column, the AVERAGE_RANDOM_FETCH_PAGES column, and the AVERAGE_SEQUENCE_FETCH_GAP column in SYSCAT.INDEXES to determine the effectiveness of the prefetchers when the table is accessed in index order.
- Query the NUM_EMPTY_LEAFS column in SYSCAT.INDEXES to determine the number of leaf pages on which all RIDs are marked as deleted (but not removed). Alternatively, query the NUMRIDS_DELETED column to determine the total number of logically deleted RIDs on leaf pages that have RIDs not all marked as deleted.

Once you have determined that your table needs reorganizing, the actual decision to do so might depend on your assessment of the relative importance of degrading query performance and the costs associated with reorganizing the table. The latter have been significantly reduced with the introduction, in v8, of *online table reorganization*, which promotes database availability by allowing the reorganization process to take place while applications have both read and write access to the table.

Online table reorganization can be paused and then resumed later. In v8, you can reorganize tables in a single partition, a set of partitions, or all partitions in a database partition group.

Online table reorganization can only be performed on a table that has type-2 indexes defined on it.

Type-2 Indexes

DB2 UDB v8 introduces *type-2 indexes*, which enhance concurrency through the elimination of most next-key locking behavior: Keys are marked deleted instead of being physically removed from the index page. A type-2 index can be created on columns that are more than 255 bytes long. Type-2 indexes are also required for MDC.

Any new index that you create is a type-2 index unless your table already has older indexes defined on it. In this case, your new index is also a type-1 index, because you cannot have both type-1 and type-2 indexes defined on the same table. You can use the REORG INDEXES command to convert type-1 indexes to type-2 indexes, using the following syntax:

```
db2 reorg indexes all for table <table-name> convert
```

Although you can use the INSPECT command to find out what type of index a table has, it's faster to just run the conversion (if that's what you ultimately want to do). If the table already has type-2 indexes, the command has no effect.

Index Reorganization

After many transactions against a table, the performance of indexes defined on that table declines. Leaf pages become fragmented and poorly clustered, and sequential prefetching is compromised, causing more leaf pages to be read when table pages are fetched. The resulting increases in I/O waits are expensive. The solution to this problem is index reorganization. You can accomplish this by dropping and then re-creating the indexes that require reorganization, or by invoking the REORG TABLE command to reorganize a table and its indexes offline. These approaches work, but they are impractical and costly in a production environment.

Version 8 introduces the REORG INDEXES command, which enables you to reorganize indexes online. Online index reorganization means that users can read from and write to a table while its indexes are being rebuilt. The REORG INDEXES command reorganizes all indexes defined on a table by rebuilding them into unfragmented, physically contiguous pages.

Table updates that affect the indexes and that occur during index reorganization are logged. These changes are then processed and applied while the indexes are being rebuilt.

Renaming an Existing Index

A radical form of index reorganization is the replacement of one index with another, perhaps better, one with the same name. Prior to v8, the process for doing this was somewhat convoluted. You had to first drop the old index and then create a new one. Because creating an index can take a significant amount of time, particularly with very large tables, this approach could be costly in terms of the performance hit to users while the process is underway.

With v8, you can rename an existing index. This enables you to create a new index, remove the old index, and then change the name of the new index so that it can be used in place of the old index.

You can rename an existing index using the following syntax:

```
db2 rename index <schema-name>.<index-name> to <new-name>
```

The schema name of the source object is used to qualify the new name for the object.

Online Database Inspection Tool

The new INSPECT command (or the db2Inspect API) allows you to inspect the architectural integrity of table spaces and tables while your database remains online. This is a significant improvement over the db2dart utility, which could only be used offline. The tool checks whether the structures of table objects and table spaces are valid. If done offline, such integrity checking of large databases could take a considerable amount of time, a period in which the

database is unavailable for use. The inspect utility is integrated into the DB2 UDB engine and, as such, can take advantage of buffer pools and prefetchers for better performance.

Inspection of a whole database includes the processing of all table objects, including index, long field, or LOB objects located in other table spaces. Inspection of a single table space includes only the processing of objects that reside in that table space.

You can specify the type of objects to be processed; the default behavior is to process all objects. Specifying the `NONE` keyword with an object excludes that object from processing.

`SYSADM`, `DBADM`, `SYSCTRL`, or `SYSMAINT` authority is required for inspect check processing. `CONTROL` privilege on a table is required for processing of that table. A connection to the database must exist before you can invoke the inspect utility.

Online inspect processing accesses database objects using the uncommitted read (UR) isolation level, and runs in the current unit of work. `COMMIT` processing is done during inspect processing, and it is a good idea to end the previous unit of work by issuing a `COMMIT` or a `ROLLBACK` statement before invoking the inspect utility.

This shows part of the syntax for the new `INSPECT` command:

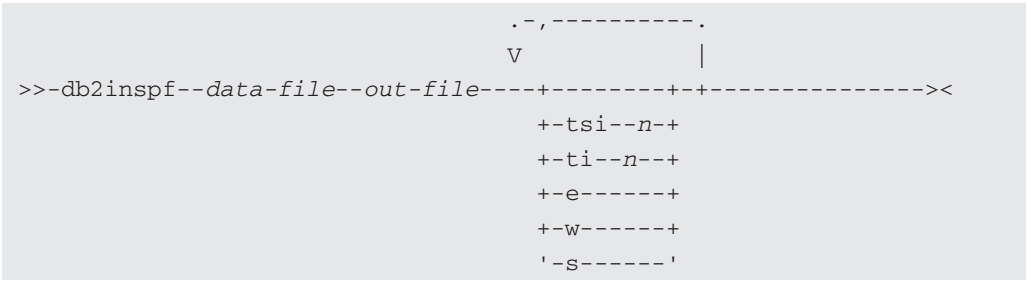
```
>>-INSPECT--CHECK------------------------------------------------>

>----+-+-DATABASE--+-------------------------------------+------------+-+-->
     | |           '-BEGIN TBSPACEID--n--+-------------+-'            | |
     | |                                 '-OBJECTID--n-'              | |
     | '-TABLESPACE--+-NAME--tablespace-name-+--+--------------------+-' |
     |               '-TBSPACEID--n----------'  '-BEGIN OBJECTID--n-'   |
     '-TABLE--+-NAME--table-name--+---------------------+-+-------------'
              |                   '-SCHEMA--schema-name-' |
              '-TBSPACEID--n--OBJECTID--n-----------------'

                          .-LIMIT ERROR TO DEFAULT--.
>--+---------------------+--+-------------------------+--------->
   '-FOR ERROR STATE ALL-'  '-LIMIT ERROR TO--+-n---+-'
                                              '-ALL-'

>--+------------------+--RESULTS--+------+--filename------------>
   '-| Level Clause |-'           '-KEEP-'

>--+----------------------------------+------------------------><
   '-| On Database Partition Clause |-'
```

The RESULTS *file-name* clause specifies a result file that is created in the diagnostic data directory path. If the inspect utility finds no errors during processing, the result file is erased at the end of the inspect operation, unless the optional KEEP keyword is specified.

You can choose from among a number of options to customize error reporting by the inspect utility. Normally, if a table object has an internal status that indicates an error state, the inspect utility reports this status but does not scan through the object. Specifying the FOR ERROR STATE ALL option, however, instructs the utility to scan through an object even if its internal status indicates an error state. The LIMIT ERROR TO *n* option specifies the number of pages in error that an object must have before inspect processing on the rest of that object is to stop. The LIMIT ERROR TO DEFAULT option specifies that inspect processing on an object is to stop when an extent size number of pages in error has been reached for that object. The LIMIT ERROR TO ALL option specifies that there is to be no limit on the number of pages in error reported on an object.

The Level Clause allows you to specify the level of processing (normal, low, or none) that is to be done on various table objects, including extent map, data, block map, index, long, or LOB objects. The On Database Partition Clause allows you to specify one or more database partitions on which inspect processing is to be done. Details about these options are provided in the DB2 product documentation.

The inspect utility writes unformatted inspection data to a specified results file. In a partitioned database environment, each database partition generates its own results file with an extension corresponding to its node number. The results file is written to the database manager diagnostic data directory path. Use the db2inspf utility to format this data so that it can be read. This is the command syntax for db2inspf:



You must specify an unformatted inspection results file and a file for the formatted output. The -tsi option specifies that only the table space with ID equal to *n* is to be processed. The -ti option specifies that only the table with ID equal to *n* is to be processed; in this case, the table space ID must also be provided. The -e, -w, and -s options specify that only error, warning, or summary data, respectively, is to be formatted.

The following example, based on the SAMPLE database, shows how these utilities can be used. After connecting to the SAMPLE database, an application issues an INSPECT command requesting inspection of the entire database. The results file, *inspect.res*, is not to be erased following

the inspect operation. The application then queries the system catalog to obtain the internal identifiers for the DEPARTMENT table and its associated table space; this information is used to filter the formatted output from the db2inspf utility. To do this, the application issues a db2inspf command, specifying the qualified name of the unformatted inspection results file. (In this example, the file is located in the db2dump subdirectory of the sqllib directory, which represents the diagnostic data directory path on UNIX-based systems.) The name of the file to which the formatted output will be written and the table space and table identifiers are also specified.

```
db2 connect to sample

db2 inspect check database results keep inspect.res

db2 "select name, tid as table_space_id, fid as table_ID
 from sysibm.systables
 where name='DEPARTMENT'"

NAME                      TABLE_SPACE_ID TABLE_ID
------------------------- -------------- --------
DEPARTMENT                             2        4
  1 record(s) selected.

db2inspf sqllib/db2dump/inspect.res inspect.for -tsi 2 -ti 4

DATABASE: SAMPLE
VERSION : SQL08010
2002-09-18-14.09.28.459109
INSPECT CHECK DATABASE

    Table phase start (ID Signed: 4, Unsigned: 4; Tablespace ID: 2) :

      Data phase start. Object: 4  Tablespace: 2
      The index type is 1 for this table.
       DAT Object Summary: Total Pages 1 - Used Pages 1 - Free Space 56 %
      Data phase end.
    Table phase end.

db2 connect reset
```

Incremental Maintenance of Materialized Query Tables

An MQT is a table with a definition based on the result of a query (a fullselect). Such a table is populated with precomputed results taken from one or more tables on which the MQT definition is based. A *summary table* is a type of MQT with a fullselect that contains a GROUP BY clause

that summarizes data from the tables that are referenced in the fullselect. A summary table is a descendent of a parent table.

The following statement creates a summary table named AVG_SAL, which has two columns, JOB and AVG_SALARY. The AVG_SALARY column is computed from the SALARY column of the parent table, STAFF:

```
db2 create table avg_sal as
   (select job, avg(salary) as avg_salary
      from staff group by job)
   data initially deferred refresh deferred
```

The DATA INITIALLY DEFERRED clause indicates that data will not be inserted into the table as part of the CREATE TABLE statement; instead, the MQT will be populated with data through the invocation of a REFRESH TABLE statement. The REFRESH DEFERRED clause specifies that the MQT data can be refreshed at any time by invoking a REFRESH TABLE statement. Specifying the REFRESH IMMEDIATE clause, on the other hand, would have requested that changes to the parent table resulting from delete, insert, or update operations be automatically cascaded to the MQT. In this example, specifying the REFRESH IMMEDIATE clause would have created an automatic summary table (AST). With an *AST*, changes made to the underlying (parent) tables are cascaded to the summary table immediately and without the need for a REFRESH TABLE statement.

There are certain restrictions on what the fullselect can include, depending on whether the table is defined as a refresh deferred or a refresh immediate MQT. For details on those restrictions, see the description of the CREATE TABLE statement in the *DB2 Universal Database Version 8 SQL Reference, Volume 2*.

Prior to v8, refreshing an MQT meant completing a *full refresh* of the MQT; in effect, a recomputation of the MQT definition. If a parent table entered check pending state (e.g., following a load insert operation), all of its descendents (including descendent refresh immediate summary tables) would also enter check pending state. Once a table entered check pending state, it was offline and unavailable for select, delete, insert, or update operations.

With v8, the MQT can remain available during a load insert operation on the underlying table. When the load operation is complete, the MQT can be refreshed *incrementally*, using only the appended data, which significantly reduces the time required to update the MQT.

Following is the syntax of the REFRESH TABLE statement, showing the new INCREMENTAL option:

```
                  .-,----------.
                  V            |
>>-REFRESH TABLE----table-name-+--+-----------------+----------><
                                 +-INCREMENTAL-----+
                                 '-NOT INCREMENTAL-'
```

If a refresh deferred MQT is to be incrementally maintained, it must have a staging table associated with it. A *staging table* is used to save data before that data is replicated to a target database; it is used as an intermediate source for updating data to a target table. You can use a `CREATE TABLE` statement to define the staging table that is to be associated with a specific MQT.

When delete, insert, or update operations modify the underlying tables of an MQT, changes resulting from those operations are automatically propagated to the staging table.

The `SET INTEGRITY` statement can be used to do the following:

- Turn off or turn on the immediate refreshing of data in refresh immediate MQTs or in propagate immediate staging tables. This can be done either with or without first performing deferred integrity checking.
- Move tables from normal (no data movement) state to normal (full access) state.
- Prune the contents of one or more staging tables.

You can also use the `SET INTEGRITY` statement to incrementally refresh MQTs or incrementally propagate staging tables with the load-appended portions of underlying tables.

Version 8 introduces three new table states to facilitate higher data availability following load insert operations:

- *Check Pending No Access state*. This new table state replaces the pre-v8 check pending state. Select, delete, insert, and update operations are not allowed on a table in this state.
- *Check Pending Read state*. This table state does not allow write access, but does allow read access to pre-existing data.
- *Normal (No Data Movement) state*. This table state allows both read and write access to a parent table that has been checked for constraints violations, but prevents data movement operations that might invalidate RIDs associated with descendent refresh immediate summary tables.

To understand how these new table states can work, consider a simple scenario in which load insert operations are performed on a single parent table (`T`) that has a single descendent refresh immediate summary table (`S`). Assume that table `T` has a referential integrity relationship with table `C`, and that table `T` also has a check constraint defined on it. Initially, tables `T`, `S`, and `C` are in normal state.

After a load insert operation (specifying `CHECK PENDING CASCADE DEFERRED` and `ALLOW READ ACCESS`) into `T` from *source*:

- `T` enters check pending read state. Read access to `T` (for data other than that contained in *source*) is allowed.
- `S` and `C` are still in normal state.

After a set integrity operation (specifying `IMMEDIATE CHECKED`) for `T`, in which table `T` is incrementally checked for constraints violations (only newly inserted rows from *source* are checked):

- T enters normal (no data movement) state. Queries and updates against T are allowed. Data movement operations that could affect descendent refresh immediate ASTs are not allowed.
- S enters check pending no access state.
- C is still in normal state, because rows appended to a parent table should not affect the validity of data in child tables.

After a refresh table operation against S (during which table S is incrementally checked):

- S enters normal state.
- T enters normal state, because the last (and in this case the only) descendent refresh immediate AST is now in normal state.
- C is still in normal state.