CHAPTER 6

Tuning Application Performance

- USING SELECT STATEMENTS
- MINIMIZE DATA TRANSMISSION
- ENBEDDED SQL PROGRAMS
- CLI/ODBC PROGRAMS
- CONCURRENCY

When you are trying to design a new database system or analyze an existing database system, one of the most important considerations you should take is the application design. Even though your database is well designed and tuned, inappropriate design of applications may cause performance problem. If your application has a design problem, fixing it often improves the application performance much more than tuning configuration parameters of DB2 UDB.

For example, SQL is a high-level language with much flexibility and different `SELECT` statements can be written to retrieve the same data; however, the performance can vary for the different forms of select statements. This is because one statement may have a higher processing cost than another. In such a case, you should choose the SQL statement which has the lower processing cost, so that the application will have good performance.

In this section, we will discuss application design considerations to obtain better performance. They includes:

- Tips to write better SQL statements
- Minimizing data transmission between applications and the database
- Considerations for embedded SQL programs
- Considerations for ODBC/CLI programs
- Concurrency on database objects

Read this chapter and apply these considerations when you develop or evaluate you database applications.

Writing Better SQL Statements

DB2 UDB provides the SQL compiler which creates the compiled form of SQL statements. When the SQL compiler compiles SQL statements, it rewrites into a form that can be optimized more easily. This is known as *query rewrite*.

The SQL compiler then generates many alternative execution plans for satisfying the user's request. It estimates the execution cost of each alternative plan using the statistics for tables, indexes, columns, and functions, and chooses the plan with the smallest execution cost. This is known as *query optimization*.

It is important to note that the SQL compiler (including the query rewrite and optimization phases) must choose an access plan that will produce the result set for the query you have coded. Therefore, as noted in many of the following guidelines, you should code your query to obtain only the data that you need. This ensures that the SQL compiler can choose the best access plan for your needs.

The guidelines for using a `SELECT` statement are:

- Specify only needed columns.
- Limit the number of rows.
- Specify the `FOR UPDATE` clause if applicable.
- Specify the `OPTIMIZED FOR n ROWS` clause.
- Specify the `FETCH FIRST n ROWS ONLY` clause if applicable.
- Specify the `FOR FETCH ONLY` clause if applicable.
- Avoid numeric data type conversion.

Each of these guidelines are further explored in the next section.

Specify Only Needed Columns in the Select List

Specify only those columns that are needed in the select list. Although it may be simpler to specify all columns with an asterisk (*), needless processing and returning of unwanted columns can result.

Limit the Number of Rows by Using Predicates

Limit the number of rows selected by using predicates to restrict the answer set to only those rows that you require. There are four categories of predicates and the processing cost of predicates are different. The category is determined by how and when that predicate is used in the evaluation process. These categories are listed below, ordered in terms of performance, starting with the most favorable:

- Range delimiting predicates
- Index SARGable predicates
- Data SARGable predicates
- Residual predicates

Note: SARGable refers to something that can be used as a search argument.

Range delimiting predicates are those used to bracket an index scan. They provide start and/or stop key values for the index search. Index SARGable predicates are not used to bracket a search, but can be evaluated from the index because the columns involved in the predicate are part of the index key. For example, assume that an index has been defined on the `NAME`, `DEPT`, and `YEARS` columns of the `STAFF` table, and you are executing the following select statement:

```
SELECT name, job, salary FROM staff
    WHERE name  = 'John'
          dept  = 10
          years > 5
```

The first two predicates (`name='John'`, `dept=10`) would be range delimiting predicates, while `years > 5` would be an index SARGable predicate, as the start key value for the index search cannot be determined by this information only. The start key value may be 6, 10, or even higher. If the predicate for the `years` column is `years => 5`, it would be a range delimiting predicate, as the index search can start from the key value 5.

```
SELECT name, job, salary FROM staff
    WHERE name  = 'John'
          dept  = 10
          years => 5
```

The database manager will make use of the index data in evaluating these predicates, rather than reading the base table. These range delimiting predicates and index SARGable predicates reduce the number of data pages accessed by reducing the set of rows that need to be read from the table. Index SARGable predicates do not affect the number of index pages that are accessed.

Data SARGable Predicates are the predicates that cannot be evaluated by the Index Manager, but can be evaluated by Data Management Services (DMS). Typically, these predicates require the access of individual rows from a base table. If required, Data Management Services will retrieve the columns needed to evaluate the predicate, as well as any others to satisfy the columns in the `SELECT` list that could not be obtained from the index.

For example, assume that a single index is defined on the `projno` column of the `project` table but not on the `deptno` column, and you are executing the following query:

```
SELECT projno, projname, repemp FROM project
    WHERE deptno='D11'
    ORDER BY projno
```

The predicate `deptno='D11'` is considered data SARGable, because there are no indexes on the `deptno` column, and the base table must be accessed to evaluate the predicate.

Residual predicates, typically, are those that require I/O beyond the simple accessing of a base table. Examples of residual predicates include those using quantified sub-queries (sub-queries with `ANY`, `ALL`, `SOME`, or `IN`), or reading `LONG VARCHAR` or large object (LOB) data (they are stored separately from the table).

These predicates are evaluated by Relational Data Services (RDS). Residual predicates are the most expensive of the four categories of predicates.

As residual predicates and data SARGable predicates cost more than range delimiting predicates and index SARGable predicates, you should try to limit the number of rows qualified by range delimiting predicates and index SARGable predicates whenever possible.

Let us briefly look at the following DB2 UDB components: Index Manager, Data Management Service, and Relational Data Service. Fig. 6–1 shows each DB2 UDB component and where each category of predicates is processed.

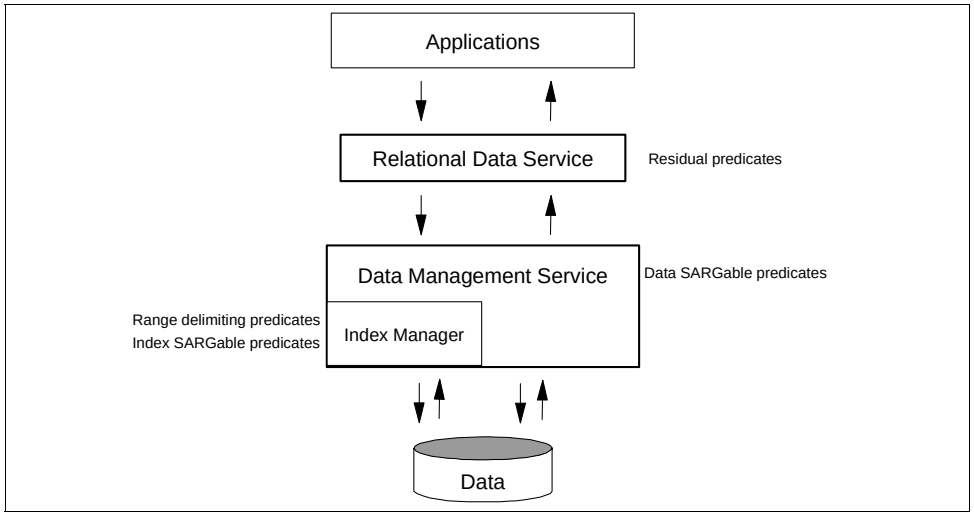

Fig. 6–1 *DB2 UDB Components and Predicates*

Note: Fig. 6–1 provides a simplified explanation. Actually, DB2 UDB has more components than are shown in this diagram.

Relational Data Service (RDS) receives SQL requests from applications and returns the result set. It sends all predicates to Data Management Service (DMS) except residual predicates. Residual predicates are evaluated by Relational Data Service (RDS).

DMS evaluates data SARGable predicates. Also, if the select list has columns which cannot be evaluated by the index search, DMS scans data pages directly.

Index Manager receives range delimiting predicates and index SARGable predicates from DMS, evaluates them, and then returns row identifiers (RIDs) to the data page to DMS.

Specify the FOR UPDATE Clause

If you intend to update fetched data, you should specify `FOR UPDATE` clause in the `SELECT` statement of the cursor definition. By doing this, the database manager can initially choose appropriate locking levels, for instance, U (update) locks instead of S (shared) locks. Thus you can save the cost to perform lock conversions from S locks to U locks when the succeeding `UPDATE` statement is processed.

The other benefit to specifying `FOR UPDATE` clause is that can decrease the possibility of deadlock. As we will discuss later in "Deadlock Behavior" on page 445, deadlock is the situation that more than one application is waiting for another application to release a lock on data, and each of the waiting applications is holding data needed by other applications through locking. Let us suppose two applications are trying to fetch the same row and update it simultanously in the following order:

1. Application1 fetches the row
2. Application2 fetches the row
3. Application1 updates the row
4. Application2 updates the row

On step 4, Application2 should wait for Application1 to complete the update and release the held lock, and then start its updating. However, if you don't specify `FOR UPDATE` clause when declaring a cursor, Application1 acquires and holds a S (shared) lock on the row (step 1). That means the second application can also acquires and holds a S lock without lock-waiting (step 2). Then the first application tries to get a U (update) lock on the row to process an `UPDATE` statement but must be wait for the second application to release its holding S lock (step 3). Meanwhile the second application also tries to get a U lock and gets into the lock-waiting status due to the S lock held by the first application (step 4). This situation is a dead lock and the transaction of the first or second application will be rolled back (see Fig. 6–2).

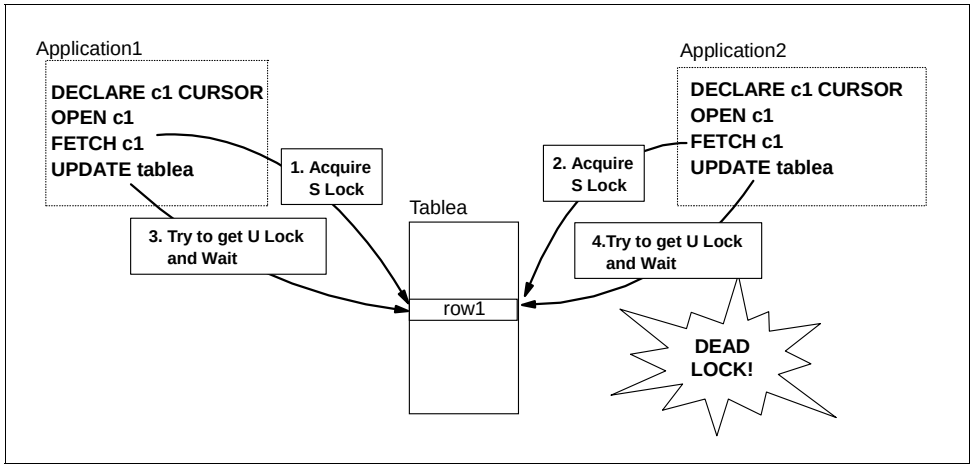

Fig. 6–2 *Deadlock between two applications updating same data*

If you specify `FOR UPDATE` clause in the `DECLARE CURSOR` statement, the U lock will be imposed when Application1 fetches the row and the second application will wait for the first application to release the U lock. Thus, no deadlock will occur between the two applications.

Note: In this example, we assume either of the two applications does not use the isolation level UR (Uncommitted Read). We will discuss isolation levels in “Concurrency” on page 435.

Here is an example to use the `UPDATE OF` clause in a `SELECT` statement.

```
EXEC SQL DECLARE c1 CURSOR FOR select * from employee
   FOR UPDATE OF job;
EXEC SQL OPEN c1;
EXEC SQL FETCH c1 INTO...;
if ( strcmp (change,"YES") == 0)
EXEC SQL UPDATE employee SET job=:newjob
  WHERE CURRENT OF c1;
EXEC SQL CLOSE c1;
```

For CLI programs, you can set SQL_MODE_READ_WRITE to the DB2 CLI connection attribute SQL_ATTR_ACCESS_MODE using SQLSetConnectAttr() function to achieve the same results. Refer to the SQLSetConnectAttr() section of the *Call Level Interface Guide and Reference* for more information.

Specify the OPTIMIZE FOR n ROWS Clause

Specify the OPTIMIZE FOR n ROWS clause in the SELECT statement when the number of rows you want to retrieve is significantly less than the total number of rows that could be returned. Use of the OPTIMIZE FOR clause influences query optimization based on the assumption that n rows will be retrieved. This clause also determines the number of rows that are blocked in the communication buffer.

```
SELECT projno,projname,repemp FROM project
    WHERE deptno='D11' OPTIMIZE FOR 10 ROWS
```

Row blocking is a technique that reduces database manager overhead by retrieving a block of rows in a single operation. These rows are stored in a cache, and each FETCH request in the application gets the next row from the cache. If you specify OPTIMIZE FOR 10 ROWS, a block of rows is returned to the client every ten rows.

Note: The OPTIMIZE FOR n ROWS clause does not limit the number of rows that can be fetched or affect the result in any way other than performance. Using OPTIMIZE FOR n ROWS can improve the performance if no more than n rows are retrieved, but may degrade if more than n rows are retrieved.

Specify the FETCH FIRST n ROWS ONLY Clause

Specify the FETCH FIRST n ROWS ONLY clause if you do not want the application to retrieve more than n rows, regardless of how many rows there might be in the result set when this clause is not specified. This clause cannot be specified with the FOR UPDATE clause.

For example, with the following coding, you will not receive more than 5 rows:

```
SELECT projno,projname,repemp FROM project
    WHERE deptno='D11'
    FETCH FIRST 5 ROWS ONLY
```

The FETCH FIRST n ROWS ONLY clause also determines the number of rows that are blocked in the communication buffer. If both the FETCH FIRST n ROWS ONLY and OPTIMIZE FOR n ROWS clause are specified, the lower of the two values is used to determine the communication buffer size.

Specify the FOR FETCH ONLY Clause

If you have no intention of updating rows retrieved by a SELECT statement, specify the FOR FETCH ONLY clause in the SELECT statement. It can improve performance by allowing your query to take advantage of row blocking. It can also improve data concurrency since exclusive locks will never be held on the rows retrieved by a query with this clause specified (see "Concurrency" on page 435).

Note: Instead of the FOR FETCH ONLY clause, you can also use the FOR READ ONLY clause. 'FOR READ ONLY' is a synonym for 'FOR FETCH ONLY'.

Avoid Data Type Conversions

Data type conversions (particularly numeric data type conversions) should be avoided whenever possible. When two values are compared, it may be more efficient to use items that have the same data type. For example, suppose you are joining TableA and TableB using column A1 of TableA and column B1 of TableB as in the following example.

```
SELECT * FROM TableA,TableB WHERE A1=B1
```

If columns A1 and B1 are the same data type, no data type conversion is required. But if they are not the same data type, a data type conversion occurs to compare values at run time and it might affect the performance. For example, if A1 is a decimal column and B1 is an integer column and each has a value '123', data type conversion is needed, as TableA stores it as x'123C', whereas TableB stores it as x'7B'.

Also, inaccuracies due to limited precision may result when data type conversions occur.

Other Considerations for Data Types

DB2 UDB allows you to use various data types. You can use SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, and DOUBLE for numeric data; CHAR, VARCHAR, LONG VARCHAR, CLOB for character data; GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB for the double byte character data, and so on. As the amount of database storage and the cost to process varies depending on the data type, you should choose the appropriate data type.

The following are guidelines when choosing a data type:

- Use character (CHAR) rather than varying-length character (VARCHAR) for short columns. The varying-length character data type can save database storage when the length of data values varies, but there is a cost to check the length of each data value.
- Use VARCHAR or VARGRAPHIC rather than LONG VARCHAR or LONG VARGRAPHIC. The maximum length for VARCHAR and LONG VARCHAR columns, VARGRAPHIC and LONG VARGRAPHIC are almost same (32,672 bytes for VARCHAR, 32,700 bytes for LONG VARCHAR, 16,336 characters for VARGRAPHIC, and 16,350 characters for LONG VARGRAPHIC) while LONG VARCHAR and LONG VARGRAPHIC columns have several restrictions. For example, data stored in LONG VARCHAR or LONG VARGRAPHIC columns is not buffered in the database buffer pool. See Use VARCHR or LONG VARCHAR? on Page XX (XREF) for futher description.
- Use integer (SMALLINT, INTEGER, BIGINT) rather than floating-point number (REAL or DOUBLE) or decimal (DECIMAL) if you don't need to have the fraction part. Processing cost for integers is much more inexpensive.
- Use date-time (DATE, TIME, TIMESTAMP) rather than character (CHAR). Date-time data types consume less database storage, and you can use some built-in functions for date-time data types such as YEAR and MONTH.
- Use numeric data types rather than character.

For detailed information about the supported data types, refer the *DB2 UDB SQL Reference*.

Minimize Data Transmission

Network costs are often the performance gating factor for applications. A good first step in investigating this is to run the application, or individual queries from it, locally on the server and see how much faster it runs. That, and the use of network monitoring tools, can indicate if network tuning or a faster network is called for. Note that here "network" includes the local case, since even though local connections have much less overhead than a network protocol, the overhead is still significant.

Note: If there is a relatively small set of queries in the application, db2batch is a good tool to investigate the queries. See XREF.

If the network itself is in good shape, you should focus on reducing the number of calls that flow from the application to the database (even for local applications).

There are several ways to reduce network costs. Here we will introduce two ways which involve having multiple actions take place through one call.

Compound SQL

Compound SQL is a technique to build one executable block from several SQL statements. When a compound SQL is being executed, each SQL statements in the block is executed individually, but the number of requests transmitting between client and server can be reduced.

Here is an example executing two UPDATE statements and one INSERT statement using compound SQL:

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
    UPDATE tablea SET cola = cola * :var1;
    UPDATE tableb SET colb = colb + :var2;
    INSERT INTO tablec (colc,cold,cole) VALUES (:i,:j,0);
  END COMPOUND;
```

When you execute a compound SQL, you can choose two types of compound SQL, atomic and not atomic. The type determines how entire block is handled when one or more SQL statements in the block end in error. The type of the example above is atomic.

Atomic: If one of the statements in the block ends in error, the entire block is considered to have ended in and error, and any changes made to the database within the block will be rolled back.

Not Atomic: When all statements have completed, the application receive a response. Even if one or more statements in the block end in error, the database manager will attempt to execute all statements in the block. If the unit of work containing the compand SQL is rolled back, then all changes made to the database within the block will be rolled back.

You can also use compound SQL to improve performance of the `IMPORT` utility, which repeats data inserts. When you specify `MODIFIED BY COMPOUND=x` option (x is the number of inserts compounded into one block), the `IMPORT` utility build a block from multiple inserts. You will gain significant performance improvement from this option. See Import utility on Page?? for more information.

Stored Procedure

A stored procedure resides on a database server, executes, and accesses the database locally to return information to client applications. Using stored procedures allows a client application to pass control to a stored procedure on the database server. This allows the stored procedure to perform intermediate processing on the database server, without transmitting unnecessary data across the network. Only those records that are actually required are transmitted. This can result in reduced network traffic and better overall performance.

A stored procedure also saves the overhead of having a remote application pass multiple SQL statements to a database on a server. With a single statement, a client application can call the stored procedure, which then performs the database access work and returns the results to the client application. The more SQL statements that are grouped together for execution in the stored procedure, the larger the savings resulting from avoiding the overhead associated with network flows for each SQL statement when issued from the client.

To create a stored procedure, you must write the application in two separate procedures. The calling procedure is contained in a client application and executes on the client. The stored procedure executes at the location of the database on the database server.

Note: Since Version 7.1 release, you can write a stored procedure using SQL statements within a `CREATE PROCEDURE` statement.

If your system has a large number of stored procedure requests, you should consider tuning some database manager configuration parameters, which are explored in the following sections.

Nested Stored Procedures

Nested stored procedures are stored procedures that call another stored procedure. Up to 16 levels of nested stored procedure calls are supported. By using this technique, you can implement more complex procedural logic without increasing client side overhead.

Keep Stored Procedure Processes

You can define two types of stored procedure, non-fenced or fenced. Non-fenced procedures run in the database manager operating environment's process, whereas fenced procedures are processed in other processes to be insulated from the internal resources on the database manager. Therefore, not-fenced procedures can provide better performance, but could cause problems to the database manager if they contain bugs.

In an environment in which the number of fenced stored procedure requests is large, you should consider keeping a stored procedure process (called DARI process) idle after a stored procedure call is completed. This consumes additional system resources; however, the performance of a stored procedure can be improved because the database manager does not have to create a new DARI process for each stored procedure request.

To keep DARI processes idle, set the database manager configuration parameter `KEEPDARI` to `YES`. This is the default value. You can set this configuration parameter by using the Control Center, or by executing the following command from the Command Line Processor in the DB2 UDB Server:

```
db2 UPDATE DBM CFG USING keepdari yes
```

If you are developing a stored procedure, you may want to modify and test loading the same stored procedure library a number of times. This default setting, `KEEPDARI=YES` may interfere with reloading the library and therefore you need to stop the database manager to load the modified stored procedure library. It is best to change the value of this keyword to `no` while developing stored procedures, and then change it back to `yes` when you are ready to load the final version of your stored procedure.

Note: For Java stored procedures, even though you set KEEPDARI=YES, you can force DB2 UDB to load new classes instead of stopping the database manager. See “Refresh Java Stored Procedure Classes” on page 418

You can use the database manager configuration parameter MAXDARI to control the maximum number of the DARI processes that can be started at the DB2 UDB server. The default value is dictated by the maximum number of coordinating agents, which is specified by the MAX_COORDAGENTS database manager configuration parameter. Since no more than one DARI process can be active per coordinating agent, you cannot set a bigger value than the maximum number of coordinating agents. Make sure this parameter is set to the number of applications allowed to make stored procedure calls at one time.

You can set the NUM_INITDARIS database manager configuration parameter to specify the number of initial idle DARI processes when the database manager is started. The default value is 0. By setting this parameter and KEEPDARI parameter, you can reduce the initial startup time for fenced stored procedures.

Load Java Virtual Machine for Stored Procedure Processes

When your application call a fenced Java stored procedure, the DARI process for the stored procedure loads the Java Virtual Machine (JVM). Therefore, if you want to reduce the initial startup time for fenced Java stored procedures, you should set the database manager configuration parameter INITDARI_JVM=YES so that each fenced DARI process loads the JVM when starting. This will reduce the initial startup time for fenced Java stored procedures when used in conjunction with the NUM_INITDARI parameter and KEEPDARI parameter.

This parameter could increase the initial load time for non-Java fenced stored procedures as they do not require the JVM.

Refresh Java Stored Procedure Classes

When a fenced DARI process execute a Java stored procedure, the process loads the JVM and the JVM locks the Java routine class for the stored procedure. If you set KEEPDARI=NO, the lock will be released after the stored procedure is completed and the DARI process is terminated; however, if you set KEEPDARI=YES, the DARI process and the JVM is up even after the stored procedure terminates. That means even though you update the Java routine class for the stored procedure, DB2 UDB will continue to use the old version of the class.

To force DB2 UDB to load the new class, you have two options. One is restarting the database manager, which may be not always acceptable. The other option is executing a `SQLJ.REFRESH_CLASSES` statement. By executing this command, you can replace Java stored procedure classes without stopping the database manager even if you set `KEEPDARI=YES`. Execute the following from the command line:

```
db2 CALL SQLJ.REFRESH_CLASSES()
```

Note: You can not update not-fenced Java stored procedure without stopping and restarting the database manager.

Embedded SQL Program

Embedded SQL programs are those statement in which SQL statements are embedded. You can write embedded SQL programs in the C/C++, COBOL, FORTRAN, Java (SQLJ), and REXX programming languages, and enable them to perform any task supported by SQL, such as retrieving or storing data.

Static SQL

There are two types of embedded SQL statements: static and dynamic. Static SQL statements are ones where the SQL statement type and the database objects accessed by the statement, such as column names, are known prior to running the application. The only unknowns are the data values the statement is searching for or modifying. You must pre-compile and bind such applications to the database so that the database manager analyzes all of static SQL statements in a program, determines its access plan to the data, and store the ready-to-execute application package before executing the program. Because all logic to execute SQL statements is determine before executing the program, static SQL programs have the least run-time overhead of all the DB2 UDB programming methods, and execute faster.

To prepare all SQL statements in a static embedded SQL program, all database objects being accessed must exist when binding the package. If you want to bind the package when one or more database objects are missed, specify the option `SQLERROR CONTINUE` in conjunction with `VALIDATE RUN` in the `BIND` or `PREP` command. Though you encounter errors for SQL statements which try to access missing database objects, the package will be bound. For the SQL statements which had errors during the bind, the rebind process will be performed at execution time. If you want to have the least run-time overhead, make sure that all database objects being accessed exists during the bind.

When and How the Access Plan is Determined?

The DB2 optimizer determines the best access plans for static SQL programs when the bind operation is performed. The determination is done based on the statistic information stored in the system catalog. Obsolete statistics information may lead the DB2 optimizer to select a inefficient access plans and may cause performance problem. Therefore, it is very important to collect the up-to-date statistics information using `RUNSTATS` utility before binding packages.

Since static SQL statements are processed based on the access plans determined during the bind, there might be better access plans if you make a lot of changes to the database after the bind. For example, assuming you have a very small table with an index and your application has static SQL statements retrieving data from the index keys, then the application tends to access the table using table scans rather than index scans because the size is too small to benefit from index scans; however, if the table considerably grows up, index scans are preferable. In such a case, you should consider executing `RUNSTATS` to refresh the table and index's statistic information stored in the system catalog, and execute `REBIND` command to take new access plans for the static SQL statements.

There are various forms of `RUNSTATS`, but a good default to use is:

```
RUNSTATS ON TABLE xxx AND DETAILED INDEXES ALL
```

Adding the `WITH DISTRIBUTION` clause can be a very effective way to improve access plans where data is not uniformly distributed.

Access Path to Volatile Tables

If you have volatile tables whose size can vary from empty to quite large at run time, relying on the statistics collected by `RUNSTATS` to generate an access path to a volatile table can be misleading. For example, if the statistics were collected when the volatile table was empty the optimizer tends to favor accessing the volatile table using a table scan rather than an index scan. If you know a table is volatile, you can let the DB2 optimizer to select index scans regardless of the existing statistics of this tables by executing `ALTER TABLE` statement with `VOLATILE` option.

```
ALTER TABLE tablename VOLATILE
```

When a table is known to be volatile to the optimizer, it will favor index scans rather than table scans. This means that access paths to a volatile table will not depend on the existing statistics on this table. These statistics will be ignored by the optimizer because they can be misleading in the sense that they are static and do not reflect the current content of the table.

To deactivate the volatile option and let the DB2 optimizer choose access paths based on the existing statistics, execute the following statement:

```
ALTER TABLE tablename NOT VOLATILE
```

Dynamic SQL

Dynamic SQL statements are ones that your application builds and executes at run time. An interactive application that prompts the end user for key parts of an SQL statement, such as the names of the tables and columns to be searched, is a good example of dynamic SQL. The application builds the SQL statement while it is running, and then submits the statement for processing. Generally, dynamic SQL statements are well-suited for applications that run against a rapidly changing database where transactions need to be specified at run time.

When and How Access Plan is Determined?

Dynamic Embedded SQL requires the precompile, compile and link phases of application development. However, the binding or selection of the most effective data access plan is performed at program execution time, as the SQL statements are *dynamically prepared*.

An embedded dynamic SQL programming module will have its data access method determined during the statement preparation and will utilize the database statistics available at query execution time. Choosing access plan at program execution time has some advantages and a drawback. The advantages are:

- Current database statistics are used for each SQL statement.
- Database objects do not have to exist before run time.
- More flexible than static SQL statements.

A drawback is that dynamic SQL statements can take more time to execute since queries are optimized at run time. To improve your dynamic SQL program's performance, the followings are keys:

- Execute `RUNSTATS` after making significant update to tables or creating indexes
- Minimize preparation time for dynamic SQL statements

Keeping statistics information up-to-date helps the DB2 optimizer to choose the best access plan. You do not need to rebind packages for dynamic SQL programs after excuting `RUNSTATS` since access plan is determined at run time.

We will discuss how to minimize preparation time for dynamic SQL statements in the following section.

Avoid Repeated Prepare

When an SQL statement is prepared it is parsed, optimized, and made ready to be executed. The cost of preparing can be very significant, especially relative to the cost of executing very simple queries (it can take longer to prepare such queries than to execute them). To improve the performance of dynamic SQL, the global package cache was introduced in DB2 UDB Version 5.0. The generated access plan for an SQL statement is stored in the global package cache, and it can be reused by the same or other applications. Thus, if the same exact statement is prepared again, the cost will be minimal. However, if there is any difference in syntax between the old and new statements, the cached plan cannot be used.

For example, suppose the application issues a `PREPARE` statement for the statement `"SELECT * FROM EMPLOYEE WHERE empno = '000100' "`, then issues another `PREPARE` statement for `"SELECT * FROM EMPLOYEE WHERE empno = '000200' "` (the same statement but with a different literal value). The cached plan for the first statement cannot be reused for the second, and the latter's `PREPARE` time will be non-trivial. See the example shown in Fig. 6–3.

```
strcpy (st1,"SELECT * FROM EMPLOYEE WHERE empno='000100'");
strcpy (st2,"SELECT * FROM EMPLOYEE WHERE empno='000200'");
EXEC SQL PREPARE s1 FROM :st1;
EXEC SQL PREPARE s2 FROM :st2;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL DECLARE c2 CURSOR FOR s2;
EXEC SQL OPEN c1;
EXEC SQL OPEN c2;
...
```

Fig. 6–3 *Dynamic SQL statements without a parameter marker*

The solution is to replace the literal '000100' by a question mark (?), issue a PREPARE, declare the cursor for the statement, assign the literal when open the cursor. By changing the program variable(s) appropriately before each OPEN statement, you can reuse the prepared statement. See the example shown in Fig. 6–4.

```
strcpy (st,"SELECT * FROM EMPLOYEE WHERE empno='?'");
EXEC SQL PREPARE s1 FROM :st;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL DECLARE c2 CURSOR FOR s1;
strcpy (parmvar1,"000100");
strcpy (parmvar2,"000100");
EXEC SQL OPEN c1 using :parmvar1;
...
EXEC SQL OPEN c2 using :parmvar2;
...
```

Fig. 6–4 *Dynamic SQL statements with a parameter marker*

Parameter markers can and should be used not just for SELECT, but also for repeated executions of INSERT, UPDATE, or DELETE statements. For example, if your application is using EXECUTE IMMEDIATE to execute multiple statements that differ only in the literal values they contain, those EXECUTE IMMEDIATE statements should be replaced by PREPARE and EXECUTE statement using parameter markers. See the following example to read records from a file and insert it into a table:

```
for ( end of file ) {
   ...
   //Read a record from the input file;
   //Generate INSERT statement to store the record
   //into a table and save the statement into
   //the host variable stmt;
   ...
   EXEC SQL EXECUTE IMMEDIATE :stmt
}
```

In this example, generated `INSERT` statements are prepared and executed for each record being inserted. If the input file has many rows, preparing all `INSERT` statements will be expensive. You should change this example as following:

```
strcpy( stmt,"INSERT INTO tablea VALUES (?,?,?)");
EXEC SQL PREPARE st FROM :stmt;
for ( end of file ) {
   ...
   //Read a record from the input file;
   //Assign read values into the host
   //variables (var1,var2,var3) for the parameter markers;

   EXEC SQL EXECUTE st USING :var1,:var2,:var3;
}
```

This example can complete the insert job faster since only one `INSERT` statement is prepared and reused for all rows being inserted.

Tune Optimization Level

Sometimes another cause of long prepare times is the use of a query optimization class that is higher than necessary. That is, the DB2 UDB Optimizer can spend more time finding the best access plan for a query than is justified by a reduction in execution time.

For example if the database has large number of concurrent activity, lots of simple SQL statement to process, and a requirement to perform them in seconds, set the optimization class to a lower value such as 1 or 2 by `SET CURRENT QUERY OPTIMIZATION` statement. If you do not set any optimization level in the `CURRENT QUERY OPTIMIZATION` special register, the DB2 optimizer will table the value set in the `DFT_QUERYOPT` database configuration parameter.

Call Level Interface and ODBC

The DB2 UDB Call Level Interface (CLI) is a programming interface that your C and C++ applications can use to access DB2 UDB databases. DB2 CLI is based on the Microsoft Open Database Connectivity Standard (ODBC) specification, and the X/Open and ISO Call Level Interface standards. Many ODBC applications can be used with DB2 UDB without any modifications. Likewise, a CLI application is easily ported to other database servers.

DB2 CLI and ODBC is a dynamic SQL application development environment. The SQL statements are issued through direct API calls. The DB2 optimizer prepares the SQL statements when the application runs. Therefore, the same advantages as dynamic embedded SQL programs are also true for DB2 CLI and ODBC programs. As we saw in the previous section, the advantages are:

- Current database statistics are used for each SQL statement.
- Database objects do not have to exist before run time.
- More flexible than static SQL statements.

Moreover, DB2 CLI and ODBC applications have the following advantages:

- Can store and retrieve sets of data.
- Can use scrollable and updatable cursors.
- Easy porting to other database platforms.

A drawback to use DB2 CLI and ODBC is that the dynamic preparation of SQL statements can result in slower query execution.

Improve Performance of CLI/ODBC Applications

Since DB2 optimizer prepares the SQL statements in CLI/ODBC programs at run time like dynamic SQL programs, the follows are considerations to improve performance:

- Execute `RUNSTATS` after making significant update to tables or creating indexes
- Minimize preparation time for SQL statements

As we have already discussed, since the DB2 optimizer trys to find the best access plan for each SQL statement based on the current statistics information saved in the system catalog, refreshing statistics information using `RUNSTATS` will help the DB2 optimizer to determine the best access plan.

To minimize preparation time for SQL statements in CLI/ODBC programs, you should consider to avoid repeated prepare, and use appropriate optimization level as discussed in the dynamic embedded SQL program section (see "Dynamic SQL" on page 421). In the following sections, we will discuss how to avoid repeated prepare and set optimization level in CLI/ODBC programs. We will also introduce two methods to minimize preparation time for CLI/ODBC applications.

Avoid repeated prepare

When you need to execute multiple statements that differ only in the literal values they contain, you can use `SQLExecDirect` repeatedly for each statements; however, this approach is expensive since each statement is prepared one by one. To avoid preparing similar SQL statements repeatedly, you can use an `SQLPrepare` call instead of multiple `SQLExecDirect`. Your program should perform the following steps:

1. Call an `SQLPrepare` to prepare the SQL statement with parameter markers.
2. Issue an `SQLBindParameter` to bind a program variable to each parameter marker.
3. Issue an `SQLExecute` call to process the first SQL statement.
4. Repeat `SQLBindParameter` and `SQLExecute` as many times as required.

The ready-to-execute package prepared by `SQLPrepare` will be reused for each SQL statement.

Tune Optimization Level

As discussed in the dynamic embedded SQL section, if the database is in such environment as Online Transaction Processing (OLTP), which typically has lots of simple SQL statement to process, set the optimization class to a lower value such as 1 or 2. To set the optimization level within the application, use `SQLExecDirect` to issue a `SET CURRENT QUERY OPTIMIZATION` statement. To set the same optimization level for all the CLI/ODBC applications on a client, use the `UPDATE CLI CFG` command from the client as the following example:

```
UPDATE CLI CFG FOR SECTION database1 USING DB2OPTIMIZATION 2
```

This command sets the CLI/ODBC keyword `DB2OPTIMIZATION=2` in the `db2cli.ini` file so that the DB2 optimizer will use the optimization level 2 to optimize SQL statements of all the CLI/ODBC applications accessing the database `database1` from this client.

Use an Optimized Copy of Catalog

Many applications written using ODBC or DB2 CLI make heavy use of the system catalog. Since the tables that make up the DB2 catalog contain many columns that are not required by the ODBC driver, ODBC/CLI applications can cause DB2 UDB to retrieve a lot of extraneous data when reading DB2 catalog data pages. Also the ODBC driver often has to join results from multiple DB2 catalog tables to produce the output required by the ODBC driver's callable interfaces.

While this does not usually present a problem for databases with a small number of database objects (tables, views, synonyms and so on), it can lead to performance problems when using these applications with larger DB2 UDB databases.

This performance degradation can be attributed to 2 main factors: the amount of information that has to be returned to the calling application and the length of time that locks are held on the catalog tables.

The db2ocat tool solves both problems by creating separate system catalog tables called the *ODBC optimized catalog tables* that has only the columns necessary to support ODBC/CLI operations.

The db2ocat tool is a 32-bit Windows program that can be used on Windows workstations running the DB2 Version 6.1 (or later) Client. You can create ODBC optimized catalog tables in DB2 databases on any platform from this tool running on Windows workstations.

Using the db2ocat tool, you can identify a subset of tables and stored procedures that are needed for a particular application and create a ODBC optimize catalog that is used to access data about those tables. The following is the db2ocat tool GUI to select tables which will be accessible through the ODBC optimized catalog:

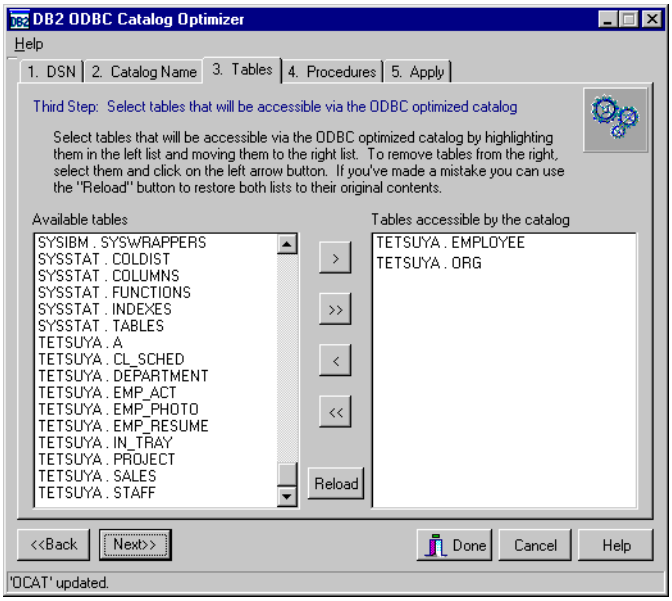

Fig. 6–5 *db2ocat tool*

An ODBC optimized catalog consists of ten tables with specified schema name. If you specify `OCAT` as the schema name during the creation of the ODBC optimized catalog, the following tables will be created:

- OCAT.TABLES
- OCAT.COLUMNS
- OCAT.SPECIALCOLUMNS
- OCAT.TSTATISTICS
- OCAT.PRIMARYKEYS
- OCAT.FOREIGNKEYS
- OCAT.TABLEPRIVILEGES
- OCAT.COLUMNTABLES
- OCAT.PROCEDURES
- OCAT.PROCEDURESCOLUMNS

These tables contain only the rows representing database objects you selected and the columns required for ODBC/CLI operations. Moreover, the tables are pre-formatted for the maximum ODBC performance. By using the ODBC optimized catalog, the ODBC driver does not need to acquire locks on the real system catalog tables or perform join operations for results from multiple tables. Therefore, catalog query times and amount of data returned as a result of these queries are substantially reduced.

You can have multiple ODBC optimized catalog for different clients. The ODBC optimized catalog is pointed to by the `CLISCHEMA` keyword. If the schema name of the ODBC optimized catalog is `OCAT`, then set `CLISCHEMA=OCAT` in `db2cli.ini` file on the client. You can directly edit the db2cli.ini file or execute the following command:

```
UPDATE CLI CFG FOR SECTION database1 USING CLISCHEMA OCAT
```

The contents in the ODBC optimized catalog is not replicated automatically from the system catalog. You must refresh the ODBC optimized catalog using the db2ocat tool when you perform something which changes the system catalog such as executing `RUNSTATS` or adding new columns (Fig. 6–6).

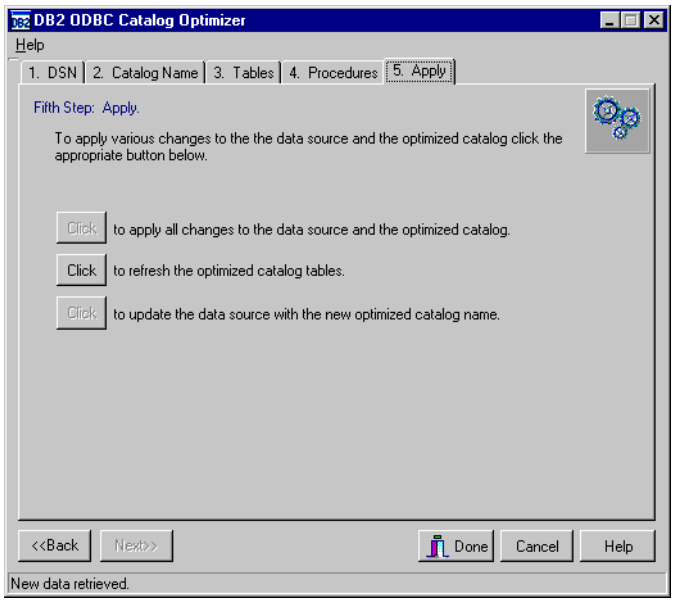

Fig. 6–6 *db2ocat tool (refresh ODBC optimized catalog)*

The db2ocat tool is available at: `ftp://ftp.software.ibm.com/ps/products/db2/tools/` in the file `db2ocat.exe`. The readme file is available in the `db2ocat.zip` file at the same site.

Convert ODBC/CLI into Static SQL

As ODBC/CLI applications are dynamic SQL applications, the most effective data access plan of each query is generated at program execution time. This process is expensive since the system catalog tables must be accessed for the resolution for the SQL statements and the statements are optimized. By using the db2ocat tool (see “Use an Optimized Copy of Catalog” on page 427), the cost to access the system catalog can be reduced; however, ODBC/CLI applications and dynamic SQL applications can be still slower than static SQL applications whose SQL statements are ready-to-execute.

In this section, we introduce the method to convert ODBC/CLI applications into static SQL applications. The information of an executed ODBC/CLI application can be captured, and the executable form of statements are stored in the database as a package. Other ODBC/CLI applications can use it like static SQL applications without the preparation cost of the SQL statements.

Note: You need to use DB2 UDB Version 7.1 or later to convert ODBC/CLI applications to static SQL applications.

ODBC/CLI applications run in the following three different modes:

- **Normal Mode**
  This is the default value and the traditional ODBC/CLI usage.
- **Capture Mode**
  This is the mode used by the database administrator who will run an ODBC/CLI application to capture its connection and statement attributes, SQL statements, and input as well as output SQLDAs. When a connection is terminated, the captured information is saved into an ASCII text file specified by `STATICCAPFILE` keyword in the `db2cli.ini` file. This file should be distributed to other clients as well as the application, and also the package should be created using the `db2cap bind` command, just as you would create a package using the `bind` command for a static SQL application.
- **Match Mode**
  This is the mode used by the end user to run ODBC/CLI applications that were pre-bound and distributed by the database administrator. When a connection is established, the captured information associated with the data source name will be retrieved from the local capture file specified by `STATICCAPFILE` keyword in the `db2cli.ini` file. If a matching SQL statement is found in the capture file, the corresponding static SQL statement will be executed. Otherwise, the SQL statement will still be executed as a dynamic SQL statement.

These modes are specified using the `STATICMODE` keyword of the `db2cli.ini` file as in the following example:

```
[SAMPLE]
STATICCAPFILE=/home/db2inst1/pkg1.txt
STATICPACKAGE=DB2INST1.ODBCPKG
STATICMODE=CAPTURE
DBALIAS=SAMPLE
```

This example specifies the capture mode. Captured information of the ODBC/CLI application accessed `SAMPLE` database is saved into the `/home/db2inst1/pkg1.txt` file. The `STATICPACKAGE` keyword is used to specify the package name to be later bound by the `db2cap bind` command.

You can directly edit the db2cli.ini file as shown above, or use the `UPDATE CLI CFG` command as the following example:

```
UPDATE CLI CFG FOR SECTION sample
     USING STATICCAPFILE /home/db2inst1/pkg1.txt
           STATICMODE CAPTURE
           STATICPACKAGE DB2INST1.ODBCPKG
```

The captured file is a text file, as shown in Fig. 6–7:

```
    [COMMON]
    CREATOR=
    CLIVERSION=07.01.0000
    CONTOKENUR=
    CONTOKENCS=
    CONTOKENRS=
    CONTOKENRR=
    CONTOKENNC=

    [BINDOPTIONS]
    COLLECTION=DB2INST1
    PACKAGE=ODBCPKG
    DEGREE=
    FUNCPATH=
    GENERIC=
    OWNER=DB2INST1
    QUALIFIER=DB2INST1
    QUERYOPT=
    TEXT=

    [STATEMENT1]
    SECTNO=
    ISOLATION=CS
    STMTTEXT=select DEPTNO,DEPTNAME,MGRNO,ADMRDEPT,LOCATION
    from DEPARTMENT
    STMTTYPE=SELECT_CURSOR_WITHHOLD
    CURSOR=SQLCURCAPCS1
    OUTVAR1=CHAR,3,,FALSE,FALSE,DEPTNO
    OUTVAR2=VARCHAR,29,,FALSE,FALSE,DEPTNAME
    OUTVAR3=CHAR,6,,FALSE,TRUE,MGRNO
    OUTVAR4=CHAR,3,,FALSE,FALSE,ADMRDEPT
    OUTVAR5=CHAR,16,,FALSE,TRUE,LOCATION
```

Fig. 6–7 *Captured File*

Note: Although this captured file has only one SQL statement, you can capture more statements in a captured file.

If necessary, you can edit the captured file to change the bind options such as QUALIFIER, OWNER, and FUNCPATH.

Then the db2cap bind command should be executed to create a package. The captured file and the database name must be specified as the following example:

```
db2cap bind /home/db2inst1 -d sample
```

The created package name have the suffix number depending on its isolation level. The suffix for the package is one of the following:

- 0 = Uncommitted Read
- 1 = Cursor Stability
- 2 = Read Stability
- 3 = Repeatable Read

In our example, the only one package DB2INST1.ODBCPKG1 will be created since our example shown in Fig. 6–7 has only one SQL statement using the isolation level Cursor Stability. If the captured file has more than one statement and their isolation levels are different, multiple packages will be created with different suffixes.

You can have more than one captured file to create multiple packages in the same database by executing db2cap bind command for each captured file. Be sure that the PACKAGE keyword of each captured file has different value since it is used to determine the package name.

Lastly, you should distribute both the captured file and the application to all client machines on which you intend to utilize the pre-bound package. On each client, the STATICMODE keyword of the db2cli.ini file should be set as MATCH, and the captured file should be specified using the STATICCAPFILE keyword.

```
[SAMPLE]
STATICCAPFILE=/home/db2inst1/pkg1.txt
STATICMODE=MATCH
DBALIAS=SAMPLE
```

Java Interfaces (JDBC and SQLJ)

DB2 UDB provides support for many different types of Java programs including applets, applications, servlets and advanced DB2 UDB server-side features. Java programs that access and manipulate DB2 UDB databases can use the Java Database Connectivity (JDBC) API, and Embedded SQL for Java (SQLJ) standard. Both of these are vendor-neutral SQL interfaces that provide data access to your application through standardized Java methods. The greatest benefit of using Java regardless of the database interface, is its *write once, run anywhere* capability, allowing the same Java program to be distributed and executed on various operating platforms in a heterogeneous environment. And since the two Java database interfaces supported by DB2 UDB are industry open standards, you have the added benefit to use your Java program against a variety of database vendors.

For JDBC programs, your Java code passes *dynamic* SQL to a JDBC driver that comes with DB2 UDB. Then, DB2 UDB executes the SQL statements through JDBC APIs which use DB2 CLI, and the results are passed back to your Java code. JDBC is similar to DB2 CLI in that you do not have to precompile or bind a JDBC program, since JDBC uses dynamic SQL.

JDBC relies on DB2 CLI, thus performance considerations which we discussed in the previous section are also applicable to JDBC applications. See “Improve Performance of CLI/ODBC Applications” on page 425.

With DB2 UDB SQLJ support, you can build and run SQLJ programs that contain *static* embedded SQL statements. Since your SQLJ program contains static SQL, you need to perform steps similar to precompiling and binding. Before you compile an SQLJ source file, you must translate it with the SQLJ translator to create native Java source code. After translation, you can create the DB2 UDB packages using the DB2 for Java profile customizer (`db2profc`). Mechanisms contained within SQLJ rely on JDBC for many tasks, like establishing connections.

Since SQLJ contains static SQL statements and their access plans are determined before being executed, the same considerations as static embedded SQL programs are applicable to SQLJ applications. See “Static SQL” on page 419.

Concurrency

When many users access the same database, you must establish some rules for the reading, inserting, deleting, and updating of data records. The rules for data access are set by each application connected to a DB2 UDB database and are established using locking mechanisms. Those rules are crucial to guarantee the integrity of the data but they may decrease concurrency on database objects. On a system with many unnecessary locking, your application may take very long time to process queries due to lock-waiting even if the system is rich in hardware resources and well tuned. In this section we will discuss how you can control concurrency appropriately and minimize lock-waits to improve your application's performance.

To minimize lock-waits, what you should consider first is eliminating unnecessary locks by performing the followings:

- Issue `COMMIT` statements at right frequency.
- Specify `FOR FETCH ONLY` clause in `SELECT` statement.
- Perform `INSERT`, `UPDATE`, and `DELETE` at the end of a unit of work if possible.
- Choose the appropriate isolation level.
- Eliminate next key locks by setting `DB2_RR_TO_RS=YES` if acceptable.
- Release read locks using `WITH RELEASE` option of `CLOSE CURSOR` statement if acceptable.
- Avoid lock escalations impacting concurrency by tuning `LOCKLIST` and `MAXLOCKS` database configuration parameter.

Each of these guidelines are further explored in the next sections.

Issue COMMIT Statements

Executing `COMMIT` statements takes overhead due to disk I/O such as flushing logged data into disks; however, since all locks held in a unit-of-work are released at the end of the unit-of-work, putting `COMMIT` statements frequently in your application program improves concurrency. When your application is logically at a point of consistency; that is, when the data you have changed is consistent, put a `COMMIT` statement.

Be aware that you should commit a transaction even though the application only read rows. This is because shared locks are acquired in read-only applications (except uncommitted read isolation level, which will be discussed in the next section) and held until the application issues a `COMMIT` or closes the cursor using the `WITH RELEASE` option (it will be discussed later in this chapter).

Note: If you opened cursors declared using `WITH HOLD` option, locks protecting the current cursor position of them will not be released when a `COMMIT` is performed. See the pages describing `DECLARE CURSOR` in the *DB2 UDB SQL Reference* for `WITH HOLD` option in detail.

Specify FOR FETCH ONLY Clause

A query with `FOR FETCH ONLY` clause never hold exclusive locks on the rows, thus you can improve concurrency using this clause. See “Specify the FOR FETCH ONLY Clause” on page 413

INSERT, UPDATE, and DELETE at End of UOW

When an application issues an `INSERT`, `UPDATE`, or `DELETE` statement, the application aquires exclusive locks on the affected rows and will keep them until the end of the unit of work. Therefore, perform `INSERT`, `UPDATE`, and `DELETE` at the end of a unit of work if possible. This provides the maximum concurrency.

Isolation Levels

DB2 Universal Database provides different levels of protection to isolate the data from each of the database applications while it is being accessed.

These levels of protection are known as *isolation levels***,** or locking strategies. Choosing an appropriate isolation level ensures data integrity and also avoids unnecessary locking. The isolation levels supported by DB2 UDB are listed below, ordered in terms of concurrency, starting with the maximum:

- Uncommitted Read
- Cursor Stability
- Read Stability
- Repeatable Read

Uncommitted Read

The Uncommitted Read (UR) isolation level, also known as dirty read, is the lowest level of isolation supported by DB2 UDB. It can be used to access uncommitted data changes of other applications. For example, an application using the Uncommitted Read isolation level will return all of the matching rows for the query, even if that data is in the process of being modified and may not be committed to the database. You need to be aware that two identical queries may get different results even if they are issued within a unit of work, since other concurrent applications can change or modify those rows that the first query retrieves.

Uncommitted Read transactions will hold very few locks. Thus they are not likely to wait for other transaction to release locks. If you are accessing read-only tables or it is acceptable for the application to retrieve uncommitted data updated by another application, use this isolation level because it is most preferable in terms of performance.

Note: Dynamic SQL applications using this isolation level will acquire locks on the system catalog tables.

Cursor Stability

The Cursor Stability (CS) isolation level locks any row on which the cursor is positioned during a unit of work. The lock on the row is held until the next row is fetched or the unit of work is terminated. If a row has been updated, the lock is held until the unit of work is terminated. A unit of work is terminated when either a `COMMIT` or `ROLLBACK` statement is executed.

An application using Cursor Stability cannot read uncommitted data. In addition, the application locks the row that has been currently fetched, and no other application can modify the contents of the current row. As the application locks only the row on which the cursor is positioned, two identical queries may still get different results even if they are issued within a unit of work.

When you want the maximum concurrency while seeing only committed data from concurrent applications, this isolation level should be chosen.

Read Stability

The Read Stability (RS) isolation level locks those rows that are part of a result set. If you have a table containing 10,000 rows and the query returns 10 rows, then only 10 rows are locked until the end of the unit of work.

An application using Read Stability cannot read uncommitted data. Instead of locking a single row, it locks all rows that are part of the result set. No other application can change or modify these rows. This means that if you issue a query twice within a unit of work, the second run can retrieve the same answer set as the first. However, you may get additional rows, as another concurrent application can insert rows that match to the query.

Note: Remeber that selected rows are locked until the end of the unit of work. Thus don't forget to issue a `COMMIT` (or `ROLLBACK`) statement even if your application is read-only. A `COMMIT` (or `ROLLBACK`) statement will terminate the unit of work and release held locks.

Repeatable Read

The Repeatable Read (RR) isolation level is the highest isolation level available in DB2 UDB. It locks all rows that an application references within a unit of work, no matter how large the result set. In some cases, the optimizer decides during plan generation that it may get a table level lock instead of locking individual rows since an application using Repeatable Read may acquire and hold a considerable number of locks. The values of `LOCKLIST` and `MAXLOCKS` database configuration parameters (see "Configure LOCKLIST and MAXLOCKS parameter" on page 443) affects this decision.

An application using Repeatable Read cannot read uncommitted data of a concurrent application. As the name implies, this isolation level ensures the repeatable read to applications, meaning that a repeated query will get the same record set as long as it is executed in the same unit of work. Since an application using this isolation level holds more locks on rows of a table, or even locks the entire table, the application may decrease concurrency. You should use this isolation level only when changes to your result set with in a unit of work are unacceptable.

Choosing an Isolation Level

When you choose the isolation level for your application, decide which concurrency problems are unacceptable for your application and then choose the isolation level which prevents these problems. Remeber that the more protection you have, the less concurrency is available.

- Use the Uncommitted Read isolation level only if you use queries on read-only tables, or if you are using only `SELECT` statements and getting uncommitted data from concurrent applications is acceptable. This isolation level provides the maximum concurrency.
- Use the Cursor Stability isolation level when you want the maximum concurrency while seeing only committed data from concurrent applications.
- Use the Read Stability isolation level when your application operates in a concurrent environment. This means that qualified rows have to remain stable for the duration of the unit of work.
- Use the Repeatable Read isolation level if changes to your result set are unacceptable. This isolation level provides minimum concurrency.

Setting an Isolation Level

The isolation level is defined for embedded SQL statements during the binding of a package to a database using the `ISOLATION` option of the `PREP` or the `BIND` command. The following `PREP` and `BIND` examples specify the isolation level as the Uncommitted Read (UR).

```
PREP program1.sqc ISOLATION UR
```

```
BIND program1.bnd ISOLATION UR
```

If no isolation level is specified, the default level of Cursor Stability is used.

If you are using the command line processor, you may change the isolation level of the current session using the `CHANGE ISOLATION` command.

```
CHANGE ISOLATION TO rr
```

For DB2 Call Level Interface (DB2 CLI), you can use the `SQLSetConnectAttr` function with the `SQL_ATTR_TXN_ISOLATION` attribute at run time. This will set the transaction isolation level for the current connection referenced by the `ConnectionHandle`. The accepted values are:

- `SQL_TXN_READ_UNCOMMITTED` : Uncommitted Read
- `SQL_TXN_READ_COMMITTED` : Cursor Stability

- SQL_TXN_REPEATABLE_READ : Read Stability
- SQL_TXN_SERIALIZABLE : Repeatable Read

You can also set the isolation level using the TXNISOLATION keyword of the DB2 CLI configuration as follows:

```
UPDATE CLI CFG FOR SECTION sample USING TXNISOLATION 1
```

The following values can be specified for the TXNISOLATION keyword: 1, 2, 4, 8, or 32. Here are their meanings:

- 1 = Uncommitted Read
- 2 = Cursor Stability (default)
- 4 = Read Stability
- 8 = Repeatable Read

You can use the DB2 CLI configuration for JDBC applications as well. If you want to specify the isolation level within the JDBC application program, use the setTransactionIsolation method of java.sql.Connection. The accepted values are:

- TRANSACTION_READ_UNCOMMITTED : Uncommitted Read
- TRANSACTION_READ_COMMITTED : Cursor Stability
- TRANSACTION_REPEATABLE_READ : Read Stability
- TRANSACTION_SERIALIZABLE : Repeatable Read

Eliminate next key locks

Next key locking is a mechanism to support Repeatable Read isolation level. If an application modify a table using such operations as INSERT, DELETE, UPDATE, the database manager will obtain key locks on the next higher key value than the modified key so that other applications using Repeatable Read can get the same result sets as long as it executes queries in the same unit of work.

However, if you don't have any applications using Repeatable Read, it is no point to use next key locking mechanism, and next key locks may cause the lock-contention. In this case, you should set DB2_RR_TO_RS=YES and eliminate next locking as following:

```
db2set DB2_RR_TO_RS=YES
```

You may significantly improve concurrency on your database objects by setting this registry variable.

This setting affects at instance level, and you need to stop and start the database manager after changing the value.

CLOSE CURSOR WITH RELEASE

You should use the Read Stability isolation level when qualified rows have to remain stable for the duration of the unit of work. If changes to your result set are unacceptable, you should use the Repeatable Read isolation level. When using Read Stability or Repeatable Read, more locks are hold than using Cursor Stability or Uncommitted Read.

If you look at your application using Read Stability or Repeatable Read, you may find that all queries in the application do not need the protection which Read Stability or Repeatable Read provides, that means, there may be queries which can release locks before the end of the unit of work. For such queries, use `CLOSE CURSOR` statement that includes `WITH RELEASE` clause when closing the cursor.

```
CLOSE c1 WITH RELEASE
```

If `WITH RELEASE` clause is specified, all read locks (if any) that have been held for the cursor will be released. If it is not specified, held locks will not be released until the unit-of-work ends.

The `WITH RELEASE` clause has no effect for cursors that are operating under the CS or UR isolation levels. When specified for cursors that are operating under the RS or RR isolation levels, the `WITH RELEASE` clause ends some of the guarantees of those isolation levels because all read locks will be released before the end of the unit of work. An RS and an RR cursor may experience the nonrepeatable read phenomenon, which means if you open a cursor, fetch rows, close the cursor with `WITH RELEASE` clause, reopen the cursor, and fetch rows again, then the second query can retrieve the different answer set as the first because other applications can update the rows that match to the query. An RR cursor may experience the phantom read phenomenon as well. After closing the cursor with `WITH RELEASE` clause, the second query can retrieve the additional rows which were not returned by the first query because other applications can insert the rows that match to the query.

If a cursor that is originally RR or RS is reopened after being closed using the `WITH RELEASE` clause, then new read locks will be acquired.

Lock Escalation

If your application acquires and holds locks on almost of all rows in one table, it may be better to have one lock on the entire table. Each database allocates memory area called *lock list*, which contains all locks held by all applications concurrently connected to the database. Each lock requires 72 bytes of memory for an object that has no other locks held on it, or 36 bytes of memory for an object that has existing locks held on it. If a number of row locks can be replaced with a single table lock, the locking storage area can be used by other applications.

When DB2 UDB converts the row locks to a table lock on your behalf, this is called *lock escalation*. DB2 UDB will perform lock escalation to avoid resource problems by too many resources being held for the individual locks.

Two database configuration parameters have a direct effect on lock escalation. They are:

- LOCKLIST — defines the amount of memory allocated for the locks.
- MAXLOCKS — defines the percentage of the total lock list permitted to be allocated to a single application.

There are two different situations for lock escalation:

- One application exceeds the percentage of the lock list as defined by the `MAXLOCKS` configuration parameter. The database manager will attempt to free memory by obtaining a table lock and releasing row locks for this application.
- Many applications connected to the database fill the lock list by acquiring a large number of locks. DB2 UDB will attempt to free memory by obtaining a table lock and releasing row locks.

Also note that the isolation level used by the application has an effect on lock escalation:

- Cursor Stability will acquire row level locks initially. If required, table level locks can be obtained in such a case as updating many rows in a table. Usually, a very small number of locks are acquired by each cursor stability application since they only have to guarantee the integrity of the data in the current row.
- Read Stability locks all rows in the original result set.
- Repeatable Read may or may not obtain row locks on all rows read to determine the result set. If it does not, then a table lock will be obtain instead.

If a lock escalation is performed, from row to table, the escalation process itself does not take much time; however, locking entire tables decreases concurrency, and overall database performance may decrease for subsequent accesses against the affected tables.

Once the lock list is full, performance can degrade since lock escalation will generate more table locks and fewer row locks, thus reducing concurrency on shared objects in the database. Your application will receive an SQLCODE of -912 when the maximum number of lock requests has been reached for the database.

Configure LOCKLIST and MAXLOCKS parameter

To avoid decreasing concurrency due to lock escalations or errors due to a lock list full condition, you should set appropriate values for both `LOCKLIST` and `MAXLOCKS` database configuration parameter. The default values of them may not be big enough (`LOCKLIST`: 10 pages, `MAXLOCKS`: 10%) and cause excessive lock escalations.

To determine the lock list size, estimate the following numbers:

- Average number of locks per application
- Maximum number of active applications

If you have no idea for the average number of locks per application, execute an application and monitor the number of held locks at the application level using the Snapshot Monitor. To get the number of locks held by a particular application, execute Snapshot Monitor as the following example:

```
GET SNAPSHOT FOR LOCKS FOR APPLICATION AGENTID 15
```

In this example, 15 is application handle number, which you can obtain using `LIST APPLICATIONS` command.

See Chapter 4 (XREF) for detailed information about the database system monitor including the Snapshot Monitor and Event Monitor.

For the maximum number of active applications, you can use the value of `MAXAPPLS` database configuration parameter.

Then perform the following steps to determine the size of your lock list:

- Calculate a lower and upper bound for the size of your lock list using the following formula:

  (Average number of locks per application * 36 * maxappls) / 4096

  (Average number of locks per application * 72 * maxappls) / 4096

  In the formula above, 36 is the number of bytes required for each lock against an object that has an existing lock, and 72 is the number of bytes required for the

first lock against an object.

- Estimate the amount of concurrency you will have against your data and based on your expectations, choose an initial value for LOCKLIST parameter that falls between the upper and lower bounds that you have calculated.

You may want to increase LOCKLIST if MAXAPPLS is increased, or if the applications being run perform infrequent commits.

When setting MAXLOCKS, you should consider the size of the lock list (LOCKLIST) and how many locks you will allow an application to hold before a lock escalation occurs. If you will allow any application to hold twice the average number of locks, the value of the MAXLOCKS would be calculated as following:

100 * (average number of locks per application * 2 * 72 bytes per locks) / (locklist * 4096 bytes)

You can increase MAXLOCKS if few applications run concurrently since there will not be a lot of contention for the lock list space in this situation.

Once you have set LOCKLIST and MAXLOCKS database configuration parameters, you can use the Snapshot Monitor and Event Monitor to validate or adjust the value of the values of these parameters. Here are the monitor elements which you should be interested in:

- Maximum number of locks held by a given transaction
- Total lock list memory in use
- Number of occurred lock escalations

You can check the maximum number of locks held by a given transaction using the Event Monitor. You need to create an event monitor to get transaction events to get this information. This information can help you to determine if your application is approaching the maximum number of locks available to it, as defined by the LOCKLIST and MAXLOCKS database configuration parameter. In order to perform this validation, you will have to sample several applications. Note that the monitor information is provided at a transaction level, not an application level.

To check total lock list memory in use, you should use the Snapshot Monitor at database level. If you notice that the monitored value is getting closer to the locklist size, consider to increase the value of the LOCKLIST parameter. Note that the LOCKLIST configuration parameter is allocated in pages of 4K bytes each, while this monitored value is in bytes.

To check the number of occurred lock escalations, you can use the Snapshot Monitor at database level. If you observe many lock escalations, you should increase the value of LOCKLIST and/or MAXLOCKS parameter.

See Chapter 4 (XREF) for detailed information about the Event Monitor and Snapshot Monitor.

Lock Wait Behavior

What happens if one application requests to update a row that is already locked with an exclusive (X) lock? The application requesting the update will simply wait until the exclusive lock is released by the other application.

To ensure that the waiting application can continue without needed to wait indefinitely, the `LOCKTIMEOUT` database configuration parameter can be set to define the length of the time-out period. The value is specified in seconds. By default, the lock time-out is disabled (set to a value of -1). This means the waiting application will not receive a time-out and wait indefinitely.

Statement Level Rollback

If a transaction waits for a lock longer than the time the `LOCKTIMEOUT` parameter specifies, the entire transaction will be rolled back by default. You can make this roll back due to time-out at statement level by setting the db2 registry variable `DB2LOCK_TO_RB=STATEMENT` by the following command:

```
db2set DB2LOCK_TO_RB=STATEMENT
```

This command should be executed by the instance owner, and you need to stop/start the database manager to make this change effective. If you set `DB2LOCK_TO_RB=STATEMENT`, lock time-outs cause only the current statement to be rolled back.

Deadlock Behavior

In DB2 UDB, contention for locks by processes using the database can result in a deadlock situation.

A deadlock may occur in the following manner:

- A locks record 1.
- B locks record 5.
- A attempts to lock record 5, but waits since B already holds a lock on this record.
- B then tries to lock record 1, but waits since A already holds a lock on this record.

In this situation, both A and B will wait indefinitely for each others locks until an external event causes one or both of them to rollback.

DB2 UDB uses a background process, called the deadlock detector, to check for deadlocks. The process is activated periodically as determined by the `DLCHKTIME` parameter in the database configuration file. When activated it checks the lock system for deadlocks.

When the deadlock detector finds a deadlock situation, one of the deadlocked applications will receive an error code and the current unit of work for that application will be rolled back automatically by DB2 UDB. When the rollback is complete, the locks held by this chosen application are released, thereby allowing other applications to continue.

To monitor deadlocks, you can use the Snapshot Monitor at database level as well as application level.

Since eliminating unnecessary locks minimize the possibility of deadlocks, tips we have discussed this section are also applicable to avoid deadlocks, therefore:

- Issue `COMMIT` statements at right frequency
- Specify `FOR FETCH ONLY` clause in `SELECT` statement
- Specify `FOR UPDATE` clause in SELECT statement
- Choose appropriate isolation level
- Eliminate next key locks by setting `DB2_RR_TO_RS=YES` if acceptable
- Release read locks using `WITH RELEASE` option of `CLOSE CURSOR` statement if acceptable
- Avoid lock escalations impacting concurrency by tuning `LOCKLIST` and `MAXLOCKS` parameter.