DB2 Everyplace powering SAP mobile solutions

Abstract

This article is intended for developers who design and implement applications for the SAP Mobile Infrastructure using the IBM DB2 Everyplace database as persistent storage. SAP Solutions for Mobile Business such as SAP Mobile Sales R/3 (MSR) or SAP Mobile Direct Store Delivery (DSD) are using the DB2 Everyplace database already today. For those SAP solutions DB2 Everyplace offers rich query and data modification capabilities which provide you the right relational infrastructure needed to process your data on the mobile device efficiently. A short primer on the DB2 Everyplace database engine architecture is providing the foundation needed to understand the performance tuning methodologies. It is described how a good index and query design improves the performance and how it will speedup particularly complex queries. In addition, the JDBC and .NET programming interfaces of DB2 Everyplace are introduced. Especially for the JDBC driver hints are given to keep the memory consumption as small as possible.

Table of content:

This whitepaper provides a short description of the functional scope of the IBM DB2 Everyplace mobile database as starting point. The main part will be organized from a developer's point of view starting with the architecture of the JDBC and the .NET interface. Then the underlying architecture of the database engine will be briefly outlined in order to gain some common ground on which the performance tuning guidelines can be explained. Those guidelines will help you understand how you can modify certain types of queries in order to achieve better performance. With this knowledge you will be able to tune the data access layer of your mobile applications.

A second whitepaper is scheduled for end of Q1/early Q2 2005. The purpose for the second one is to show best practices for the persistence API provided by SAP Mobile Infrastructure.

1) Introduction

Applications running on mobile devices such as PDAs and smart phones need to process data. The applications developed for such devices cover different business areas - for example mobile asset management, service technician solutions, sales force automation and supply distribution, just to name a few. But they share some common needs for data processing:

- search and query capabilities
- efficient data modification capabilities ensuring and maintaining data integrity
- reporting capabilities using complex join queries
- data encryption
- persistent storage with zero admin work for mobile end user

Since data used in this processing is synchronized with relational backend databases the relational data model and data access techniques might be reused. It might be possible for example to re-use parts of the data access related logic of an application using DB2 UDB in a solution using DB2 Everyplace database on a mobile device. This benefit can be gained if you use the relational IBM DB2 Everyplace mobile database which is specifically designed for the mobile and embedded application environment with minimal storage requirements. DB2 Everyplace mobile database provides high performance, a reliable, persistent, transactional and thread-safe storage for your data on mobile devices, without the need of a database administrator due to its zero admin design. Rich query and data modification capabilities offer you the relational infrastructure you need to process your data on the device efficiently. Industry-standard programming interfaces allow rapid application development. Encryption on a per table basis can be used to secure your business success by protecting your confidential information in case a mobile device gets lost or stolen. Customization of your applications for different geographies is easy through the support for all major languages.

2) Functional scope

IBM DB2 Everyplace mobile database is designed for mobile devices such as PDAs, smart phones and for embedded solutions. It is part of the 3-tier DB2 Everyplace data synchronization solution. The focus of this article is the mobile database and the synchronization part of DB2 Everyplace will not be explained here. The database engine is available for the following platforms: Symbian, PocketPC, Win32, embedded Linux, Linux, QNX Neutrino and Palm. The database engine allows data storage on external media such as SD card or MicroDrive. The programming interfaces supported are the Call Level Interface (CLI) and ODBC for C/C++, the JDBC interface for Java and the .NET interface for development support within the .NET framework. Taking the limited computing power into consideration and aiming at the lowest resource consumption possible only functionality most relevant for mobile databases has been included (which is a subset of the SQL 92 standard). In addition, the database engine supports Single Byte Character Sets (SBCS), Double Byte Character Sets (DBCS), and UNICODE where available. The database is globalized for major languages: English, Spanish, French, German, Italian, Czech, Polish, Brazilian Portugese, Slovak, Hungarian, Japanese, Korean, Traditional Chinese and Simplified Chinese, Hebrew and Arabic.

DB2 Everyplace database is a relational database system capable of relational operations such as joins, GROUP BY and ORDER BY. In addition it supports single and multi-column primary keys, aggregation and scalar functions, and scrollable cursors. Advanced indexing capabilities can be used to increase the performance of queries significantly. With V8.2, support for multiple concurrent connections to the same database from within the same process was added supporting the isolation levels UR, CS, RS and RR[1] during transaction processing. A short – but not complete – summary of the supported features is below:

- CREATE/ALTER/DROP TABLE
- CREATE/DROP INDEX (bidirectional, single and multi-column)
- Check constraints, identity columns, DEFAULT VALUE
- Transactional, concurrent processing with isolation levels UR, CS, RS, RR
- INSERT, INSERT with subselect, DELETE and UPDATE
- SELECT with support for joins, DISTINCT, GROUP BY and ORDER BY clause, IN and LIKE predicate
- scrollable cursor support
- Aggregate functions like MAX, MIN, AVG, SUM, COUNT
- Scalar functions like RTRIM[2], LENGTH, DAY, TIME, TIMESTAMP
- EXPLAIN statement
- Encryption support on a per-table basis

After this short coverage of the functional scope supported by DB2 Everyplace mobile database, the remainder of this article will be a developer's approach to the database: First the programming interfaces will be introduced in order to be able to use the database. Second performance tuning guidelines will be given based on some basic understanding on the database architecture which will be introduced as well.

3) Architecture

3.1) The JDBC Interface

3.1.1) Architecture of the JDBC interface

J2ME is the Java platform enabling mobile devices for Java. Multiple vendors are providing JVMs for mobile devices and offer therefore a runtime environment for Java applications in the mobile solution market (IBM offers with the J9 JVM a J2ME platform available for almost all device platforms in the mobile market). DB2 Everyplace database offers support for Java applications on mobile devices by providing a JDBC programming interface. The JDBC interface of DB2 Everyplace implements most of the functionality of JDBC 2.1 with some minor exceptions implied by the mobile computing environment with constraint availability of resources. *Figure 1* shows the architecture of the core classes and interfaces of the JDBC API contained in the java.sql.* package.

The basic steps to use the DB2 Everyplace database from a Java application are loading the JDBC *Driver*, opening a *Connection* to the database and then using a *Statement* or a *PreparedStatement* for performing the required SQL operations. The *Connection* interface defines methods for handling connections. The *DatabaseMetaData* interface provides methods to retrieve global information such as database version, supported functionality (like batch updates) and upper limits (like the maximum row size or the maximum length of a table

[1] UR = uncommited read, „dirty read; CS = cursor stability; RS = read stability; RR = repeatable read
[2] Note that LTRIM or TRIM is not supported. The reason why RTRIM is supported is to cut off trailing blanks of CHAR columns which are added *after* the string value inserted and are therefore on the right side.

name). The *Statement* interface (and its extending interfaces *PreparedStatement* and *CallableStatement*) provides the methods for sending SQL to the database and creating the result sets. If SELECTs are issued then the results are retrieved using instances of the *ResultSet*. DB2 Everyplace supports scrollable *ResultSet*s with relative and absolute positioning. Information about the number of columns and the corresponding data types of a *ResultSet* can be obtained using the *ResultSetMetaData*. If an error situation or a situation requiring a warning is encountered, a *SQLException* or *SQLWarning* will be thrown.

3.1.2) Hints for using the JDBC interface of DB2 Everyplace

The DB2 Everyplace JDBC driver is a Type 2 driver mapping Java to the CLI interface. The driver therefore consists of two parts: A set of pure Java classes plus a JNI bridge to the CLI interface. Many CLI and JDBC concepts are similar, thus a JDBC Type 2 driver could be considered more like a wrapper module. For certain JDBC classes like *Statement* or *ResultSet* only minor parts are implemented in the Java classes which are therefore small and simple. This means most of the memory is allocated by the native code invoked by the JDBC methods using JNI and this memory allocation happens outside the Java heap memory space. However, the garbage collector (GC) prioritizes objects according to their size in the Java heap memory space and memory outside this area is invisible to the GC. Therefore the GC cannot determine correctly the size of instances of the *Statement* or the *ResultSet* class and considers them low priority for cleanup. This is crucial to know because due to this you should write “clean” code in order to avoid relying on the GC. “Clean” code means to call for all objects like

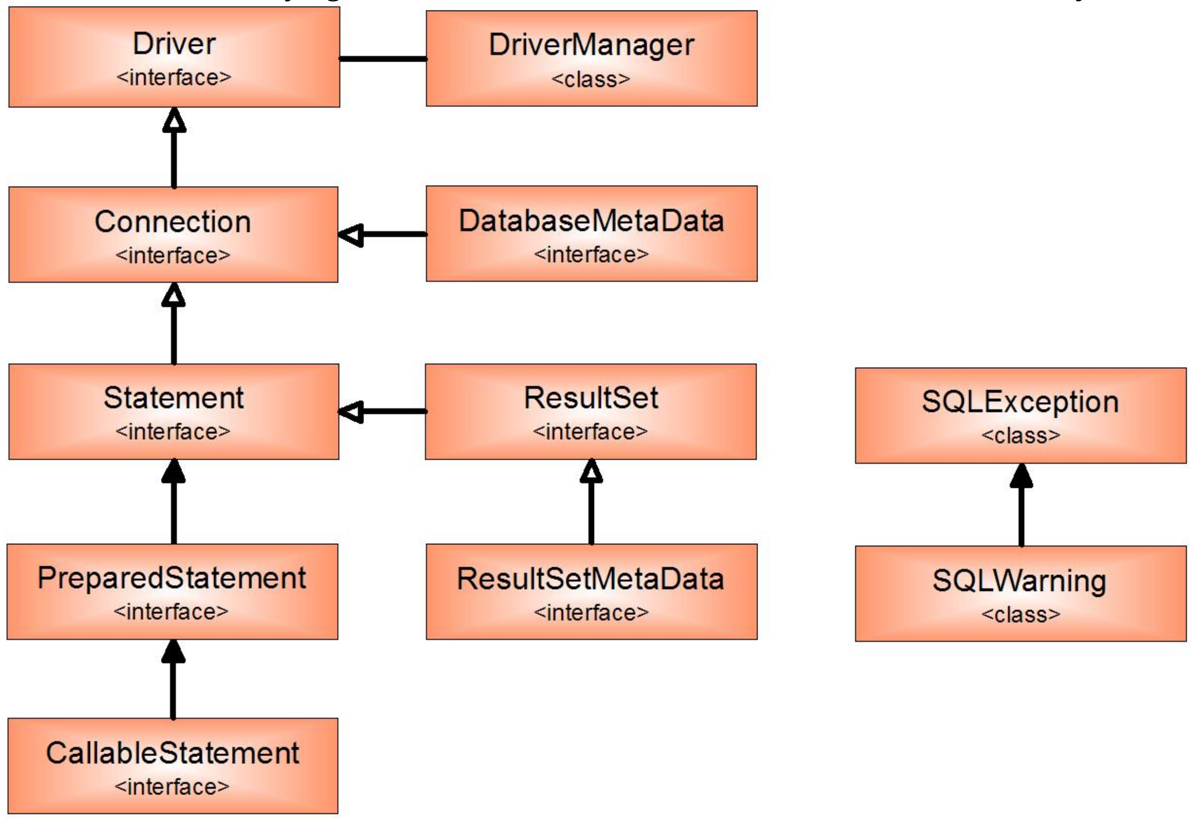


Figure 1: Relation of the JDBC 2.1 core classes and interface of the java.sql. package*

ResultSets, *Statements* or *PreparedStatements* the corresponding *close()* method to ensure that the used memory outside the Java heap memory will be freed before the GC starts collecting unreferenced objects. As a good practice, you should write finally-blocks for all try-catch-blocks where an *SQLException* could be thrown to make sure the *close()* method is also called in the error case (see item 3 below for a code snippet)

If you are developing Java applications using the DB2 Everyplace database through the JDBC interface, there are some things you should consider from the Java perspective:

1. Do not instantiate the DB2 Everyplace JDBC driver like this:
```
try {
        Class.forName(“com.ibm.db2e.jdbc.DB2eDriver“).newInstance();
}
catch (Exception e) {
        // error handling
}
```
   This way you create an instance of the DB2eDriver which is not needed. This just generates unnecessary work for GC. Instead, remove the call for **newInstance()** to avoid useless allocation and freeing work of memory as shown below:
```
try {
        Class.forName(“com.ibm.db2e.jdbc.DB2eDriver“);
}
catch (Exception e) {
        // error handling
}
```
2. In SELECT statements you should specify only the columns in the column list which you really need. The reason for this is that on mobile devices the price for object instantiation is hitting the overall application performance even more than on desktop computers. DB2 Everyplace is instantiating the object representing the column values in a single row of the *ResultSet* only when the column in the row is accessed. Imagine you iterate through the *ResultSet* of a SELECT * FROM <*myTable*> where *myTable* has for example 60 columns and 2000 rows but for your application you actually need only 4 columns of them. Then you are paying the performance penalty for 56 * 2000 = 112.000 object instantiations which are not needed. This will cost you significant runtime on PDAs – especially if the involved data types require BigDecimal (DECIMAL type in database) or String (CHAR or VARCHAR type in database) object instantiation in Java and the removal of these by the GC. For the above example, you should explicitly specify the desired columns: SELECT col_x1, col_x2, col_x3, col_x4 FROM <*myTable*>.
3. You need to ensure that you are invoking for all instances of *Statement*, *PreparedStatement* and *ResultSet* and *Connection* the corresponding *close()* method, even in an error case. This way you ensure that all associated memory will be properly released. An example for a code structure achieving this could look like:
```
try{
        //code here which could throw a SQLException
}
catch(SQLException sqlEx){
        //do error handling here
}
finally
{
        try{
                if( rs != null )
                { rs.close(); }
                if( stmt != null )
                { stmt.close(); }
        }
        catch(SQLException){
                //do error handling here
        }
}
```

At the end of the try-block you should try to close instances of *ResultSets* or *Statements* to release the memory. With the finally-block as shown above you also ensure that the instances of *ResultSet* or *Statement* variables are closed and the memory is released in case of error.

4. By following CLI the JDBC API has slightly different counting semantics compared to what you are used from Java. If you intend to obtain the first column of a *ResultSet*, you start counting at 1 (whereas in Java the first object in an array would be addressed using 0).

   This example get the **first** column of the *ResultSet* using **rs.getString(1)**:

   ```
   rs = stmt.executeQuery("select tname, numcols from DB2eSYSTABLES");
   while( rs.next() )
   {
           tName     = rs.getString(1);
           //do some stuff here
   }
   ```

   This example get the **first** entry from an array:

   ```
   name = nameList[0];
   ```

5. Result sets are not closed automatically if a *commit* or a *rollback* is called. They are kept open *across* transaction boundaries. Therefore you need to call the *close()* method in order to close them.

3.2) Architecture of the .NET interface

DB2 Everyplace mobile database has a .NET interface supporting the Microsoft .NET Standard Framework for Win32 devices as well as the .NET Compact Framework for PDAs running PocketPC. With the .NET managed providers you can write applications in languages such as C#. Within the .NET framework the cornerstone for working with (relational) data and databases is a library known as ADO.NET. At first glance, it looks very much the same like the predecessor technology ADO. However, ADO is built around the concept of record sets whereas ADO.NET has as key element, the *DataSet*, which can be used to model an entire database with tables and relationships among tables. In addition, ADO.NET also abstracts other database concepts like connections and statements into classes. The most important property of the *DataSet* is that it represents a **disconnected** data architecture. Once the data is retrieved and stored in the *DataSet*, the connection between the database and the *DataSet* is closed. Further, the design objective of ADO.NET is a better XML integration and a better update control compared to the predecessor technology.

The architecture of the DB2 Everyplace .NET managed providers as shown in *Figure 2* is the same for the .NET Standard and the Compact Framework. The .NET manages providers of the DB2 Everyplace support through the *DB2eDataAdapter* the *DataSet* (disconnected mode) and the *DB2eDataReader* (connected mode). The connection between the disconnected *DataSet* and the DB2 Everyplace database is established through the *DB2eDataAdapter*. Changes done to the data in the *DataSet* can be written back to the database by re-establishing a connection through the *DB2eDataAdapter*. This means, that you need a connection in disconnected mode only for initially populating the *DataSet* and if you intend to write any changes done to the data back to the database. In between you can work with the data without keeping a connection to the database open (hence the name “disconnected” mode). In case you need to scroll through a query result in a connected fashion conceptually similar to what is provided through the JDBC *ResultSet* you need to use the *DB2eDataReader*. This class offers support just for *forward-only* scrolling. If you use the DB2 Everyplace database through JDBC, you have the advantage of absolute and relative cursor positioning in both directions. Opposite to the *DataSet*, during the entire time you use the *DB2eDataReader* a connection to the database must stay open. A connection to the database is provided by the

Connection class. Using the *Command* class you can tell the database what you intend to do. Assigning the SQL you intend to have executed to a member of this class is basically the way to prepare SQL statements for execution. The *DataSet* and the *DataReader* are very well integrated with the GUI components of the .NET framework such as the *DataGrid* or *ListBox*. This means you can bind query results to such GUI components fairly easily and display the data from the database with a few lines of code.

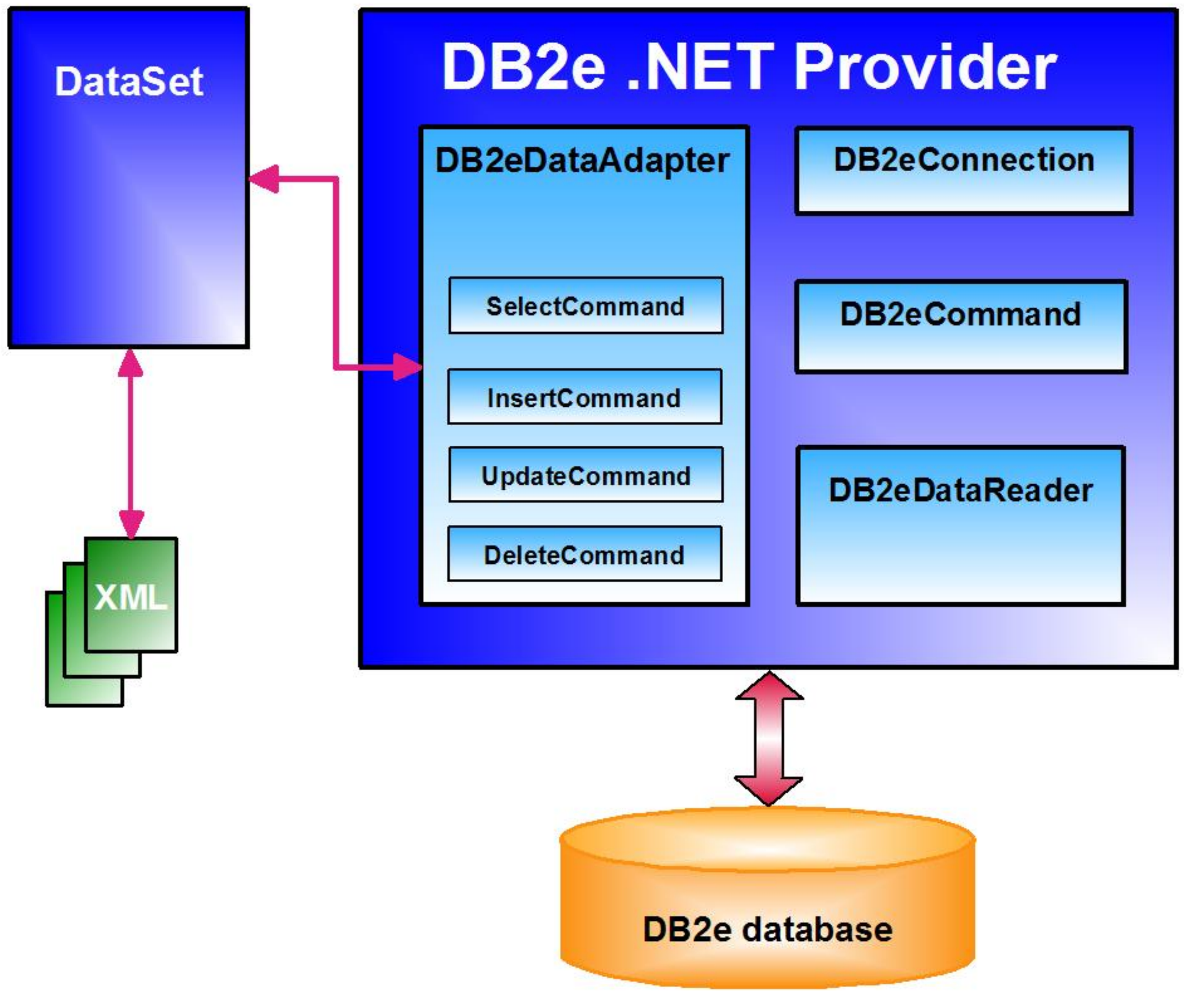


Figure 2: Architecture of the DB2 Everyplace .NET managed providers

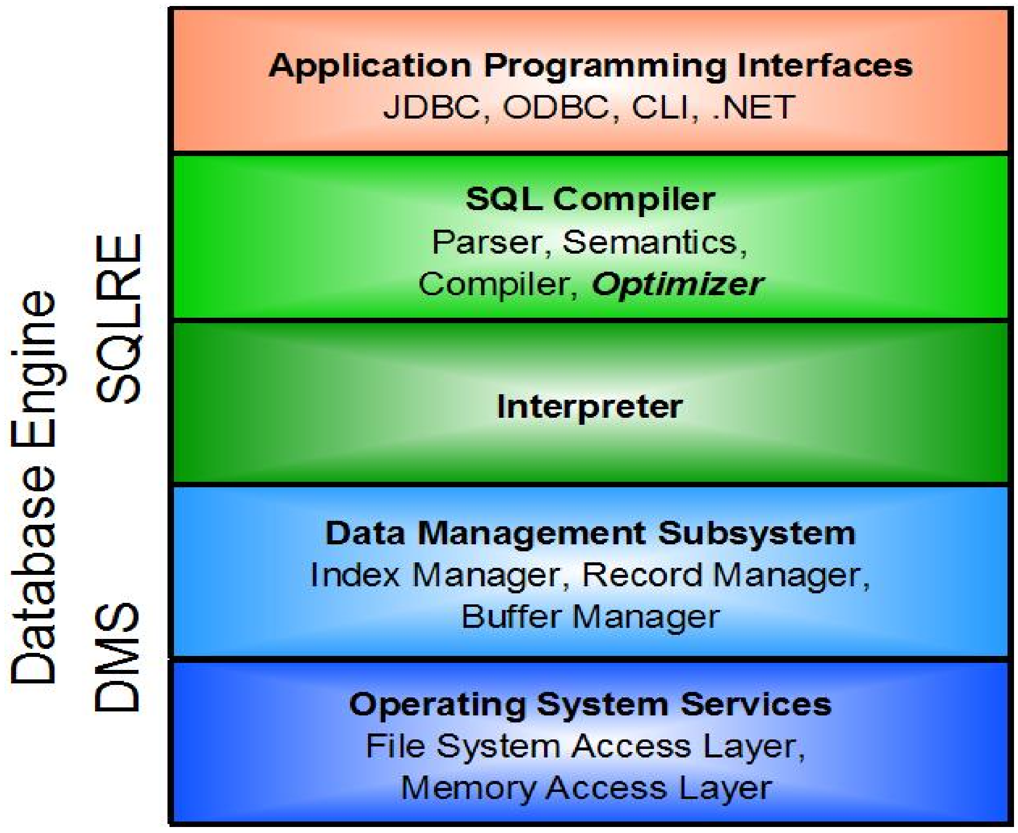
3.3) A short primer on database architecture

In this subsection I will briefly introduce the architecture of the mobile database engine from a high-level view. The main components and the core function of each component will be explained. In *Figure 3* you can see the general outline of the database engine architecture.

For application programming DB2 Everyplace offers various interfaces such as CLI, ODBC, JDBC and .NET. These fulfill industry standards. The JDBC and .NET interface were briefly explained in previous sections. I left out the CLI interface because it is not in the scope of this article. Beneath the application programming interfaces is the database kernel which consists of two major components:

- **SQL Runtime Environment (SQLRE)**
    - SQL Compiler
        - Parser

        - Semantics
        - Compiler
        - *Optimizer*
    - Interpreter
- **Data Management Services (DMS):**
    - Database Management Subsystem
        - Index Manager
        - Record Manager
        - Buffer Manager
    - Operating System Services:
        - File System Access Layer
        - Memory Access Layer


Application Programming Interfaces
JDBC, ODBC, CLI, .NET
SQL Compiler
Parser, Semantics, Compiler, Optimizer
Interpreter
Data Management Subsystem
Index Manager, Record Manager, Buffer Manager
Operating System Services
File System Access Layer, Memory Access Layer
Database Engine
SQLRE
DMS
Figure 3: DB2 Everyplace database architecture

The first component invoked from the SQL compiler is the *parser* which is responsible for the syntactic validation of SQL statements. Syntax errors (SQLState 42601) can be detected at this level. In case the SQL statement is syntactically correct then the *semantics* component performs type checking and verifies that all database objects referenced by the statement exist. The next task performed by the *compiler* is to build an *access plan* – the executable form of a SQL statement. The access plan describes what the steps are the database needs to perform in order to execute the SQL statement. The access plan contains for instance information regarding the order of tables accessed, indexes used for evaluating conditions or if the result set is ordered in memory or using a temporary table on the file system .
The *optimizer*, as the name implies, improves the access plan generated by the compiler in order to achieve a better query execution time. During this step for example redundant conditions will be removed and *index selection* will be performed. Depending on your queries as part of a good database design you should consider index creation. If there are no indexes available obviously the optimizer has nothing to consider for index selection. However, if

there are indexes available, the flip side of the coin is that the way you write your query influences index selection. You will see in the following subsections that a crucial part in writing SQL statements which perform well is to structure them in a way that from the available indexes the most beneficial regarding performance pass the requirements for being selected.
Once the access plan is generated, the SQL *interpreter* executes the access plan by calling services of the Data Management System. Note that for prepared statements the generated access plan can be reused with a different set of parameters in which case the steps for building and optimizing it are omitted. The basic tasks of a SQL compiler are conceptually comparable to a Java compiler. The Java compiler also validates the syntax of a Java program first, and then generates a parse tree and, performs type checking. Then the Java compiler generates the Java byte code (the "access plan" for your Java program) and also removes unnecessary loops, variables, conditions, etc. during this step. The Java interpreter (= JVM) can then execute the compiled class files.
The **Data Management Subsystem** has four major components: The *index manager* is responsible for accessing, modifying and maintaining the index tree structures. This component is always involved if indexes are supporting the SQL statement execution. Tables are stored on disk in pieces of the same size called *data pages*. The *record manager* handles row extraction and compression from or to the data pages. Since file system operations are slow, the database manages a certain amount of memory called *buffer pool* to keep frequently accessed data pages in memory for faster access on subsequent operations. The *buffer manager* manages the buffer pool and controls when data pages are read from the file system or flushed to it if space needs to be freed in the buffer pool.
The **Operating System Services** contain the components for accessing the file system, the device memory or other functions like encryption.

If you are interested in more details on the architecture of DB2 Everyplace mobile database, the article listed as item 6) in the bibliography is providing them for DB2 Everyplace V8.1. Note that some minor details are outdated for V8.2 though.

3.4) Optimizer considerations

Today, the most common implementations of SQL optimizers are either *cost-based* optimizer or *rule-based* optimizers. Neglecting many details, I just sketch the main differences between them from a top-level view:

Access plans received by a SQL optimizer which performs cost-based optimizing is transformed into another equivalent SQL statement regarding the result but with a better performance. For the internal optimization of the SQL a cost-based optimizer uses for example statistical information on tables and other database objects. Depending on the implementation even non-database statistics like performance characteristics of hard drives might be considered. The first huge benefit of this design is that the developer needs to worry less on the SQL statement used in the application because the SQL optimizer will be able to transform it in many cases to a faster performing one. The second benefit is that cost-based optimizers often have a feature supported called *join-reordering* (I will briefly explain this very powerful feature in the performance tuning section). The first drawback of this design is that a database administrator is needed to keep the statistical information used by the SQL optimizer up to date. Failure to do so can result in poor performance because during the access plan optimization the SQL optimizer can not make proper decisions anymore due to using statistical information which is outdated. The second drawback is that an access plan generated by EXPLAIN during *development time* might be different from the access plan generated and used at *execution time* due to different statistics at development and execution time. This means that a developer can not control anymore[3] that a SQL statement is performing well at execution time since the statistics at execution time on which the optimizer decisions are based are unknown (some backend database vendors today support or currently add features for their cost-based SQL optimizers to give the developer some of this control back). Cost-based optimizing is a widely used technique in many commercial databases available for backend.

Rule-based optimizers as the name implies just use rules **without** considering statistical information. The first benefit of this design is that the developer has full control of how the SQL statement will be executed at runtime because the access plan reviewed with EXPLAIN at *development time* and the access plan used at *execution time* will *be the same*. The second benefit of this design is that no database administrator is needed because there is zero admin work to do in order to generate and maintain statistics. Rule-based optimizers also have drawbacks: First of all, the developer has a much higher responsibility in anticipating the data load[4], in writing table and index designs and for the SQL statements which lead to good performance because the rule-based optimizer can not cover up for mistakes as much as a cost-based optimizer can. The second drawback is that join-reordering is usually not supported. The binary footprint of rule-based optimizers is usually smaller than the footprint of cost-based optimizers due to the fact that the complexity of rule-based optimizers is usually lower. Since storage space on mobile devices is small this makes rule-based optimizers also a feasible choice for the mobile computing environment.

Since DB2 Everyplace is a database for the mobile environment it has as a rule-based optimizer SQL optimizer with the benefits and drawbacks as outlined for rule-based optimizers above.

4) Performance Tuning

Since many mobile devices like smart phones or PDAs have very limited computing power, each part of an application needs performance tuning if the application should run fast on such devices. This includes of course the code dealing with the database. Assuming a good entity-relationship model for the data stored in the database additional steps like feasible index creation and SQL statement tuning for optimizing the index usage needs to be done. This section provides you with the information how DB2 Everyplace database takes advantage of indexes and helps you identifying where indexes are useful to increase performance using certain tools.

4.1) EXPLAIN

As outlined in the architecture section the SQL compiler generates an access plan. This plan describes how a SQL statement is executed and includes information on used indexes or temporary table creation for sort operations. In order to improve query performance you might need to create indexes or restructure your query. However, you need access to this information in order to so. The EXPLAIN provides this information by generating the access plan for a query without executing it and saving it into a DB2 Everyplace database table called **DB2ePLANTABLE**. Once you reviewed it you might change the query or create an index and re-execute the EXPLAIN for the SQL statement to see how the modifications you

[3] In many cases due to the cost-based optimization the developer has the freedom to care less about it.
[4] The data load might influence significantly certain queries.

did affect the access plan. Iterating through these steps might be necessary until you find a well performing solution.

You can issue an EXPLAIN for SQL statements through the **DB2eCLP** command line processor. The DB2eCLP is shipped with the DB2 Everyplace. You can also execute the EXPLAIN statement through one of the programming interfaces. Note that the EXPLAIN statement is only supported on Win32 and Linux running on the x86 architecture.

Before we start looking into details of performance tuning you should make yourself familiar with the **EXPLAIN** statement. If you would like to see if indexes are used you could review the access plan by querying the DB2ePLANTABLE. In case your SQL statement has an ORDER BY-clause you can also see if the ordering is supported by an index and done in memory or if the ordering is lacking index support and is instead performed in a temporary table on the file system which is much slower. Analyzing the access plan gives you information how you should change your query or index design to improve query performance. Since you are dealing with a database using a rule-based optimizer the access plan at execution time will be the same as you are seeing at development time[5] leading to the following rough rules of thumb:

- Good access plan at development time leads to good performance and the same access plan at runtime
- Poor access plan at development time leads to poor performance and the same access plan at runtime

Examples how to use EXPLAIN are given in the remainder of this article.

4.2) Performance tuning using indexing techniques

Consider the case of an exact-match query (example: SELECT * FROM t WHERE t.col1=x): If no index is available, the default behavior by the database will be to scan through the entire table examining each row if it satisfies the required conditions. If a table has **n** rows, this approach will run in linear time in the number of rows. This type of scanning through a table is known as **table scan**. If you issue the same query and the database can use an index (in the example an index on column col1), the database will perform a search in the index tree (known as **index scan**) which is running approximately in log(n) runtime. This requires, however, non-degenerated index trees. A degenerated index tree could be generated if you define an index on a column which has the same value for all rows. It is called degenerated because it looks like a list since all nodes are hanging in a linked list fashion from one node of the index tree. The index scan will be most beneficial on reasonable large to huge tables where the column with the index has a high percentage of distinct values compared to the overall number of rows. An index scan selects only the pages from the table which contain at least one qualifying row. This means the amount of data read from the file system could be much lower leading to much better performance because less data pages from the table need to be loaded into memory to retrieve the rows in the result set. Therefore, for optimizing SELECT query performance you need to identify on which columns you need to create indexes to support the WHERE-conditions of your queries properly. The following items are relevant to know for proper index creation if DB2 Everyplace database is used:

- Indexes are implemented bi-directional.

[5] This assumes that the data load at development time is comparable to the data load at runtime.

- If multi-column indexes are defined index prefix scanning is supported (see 4.2.2 for details).
- In case of multi table selects (typically joins) the order of the tables in the FROM-clause matters because Join-reordering is not supported.

4.2.1) Bi-directional indexes

Let us start with the bi-directional index implementation. If you issue a SELECT query with an ORDER BY clause, for each column in the ORDER BY clause you can specify ascending or descending order. Important to note is that the direction specified in the ORDER BY clause does not matter: If an index on a column for the ORDER BY clause is available, it will support ascending and descending orders due to the bidirectional implementation which means that you need only one index to support both order directions. Consider now the examples below:

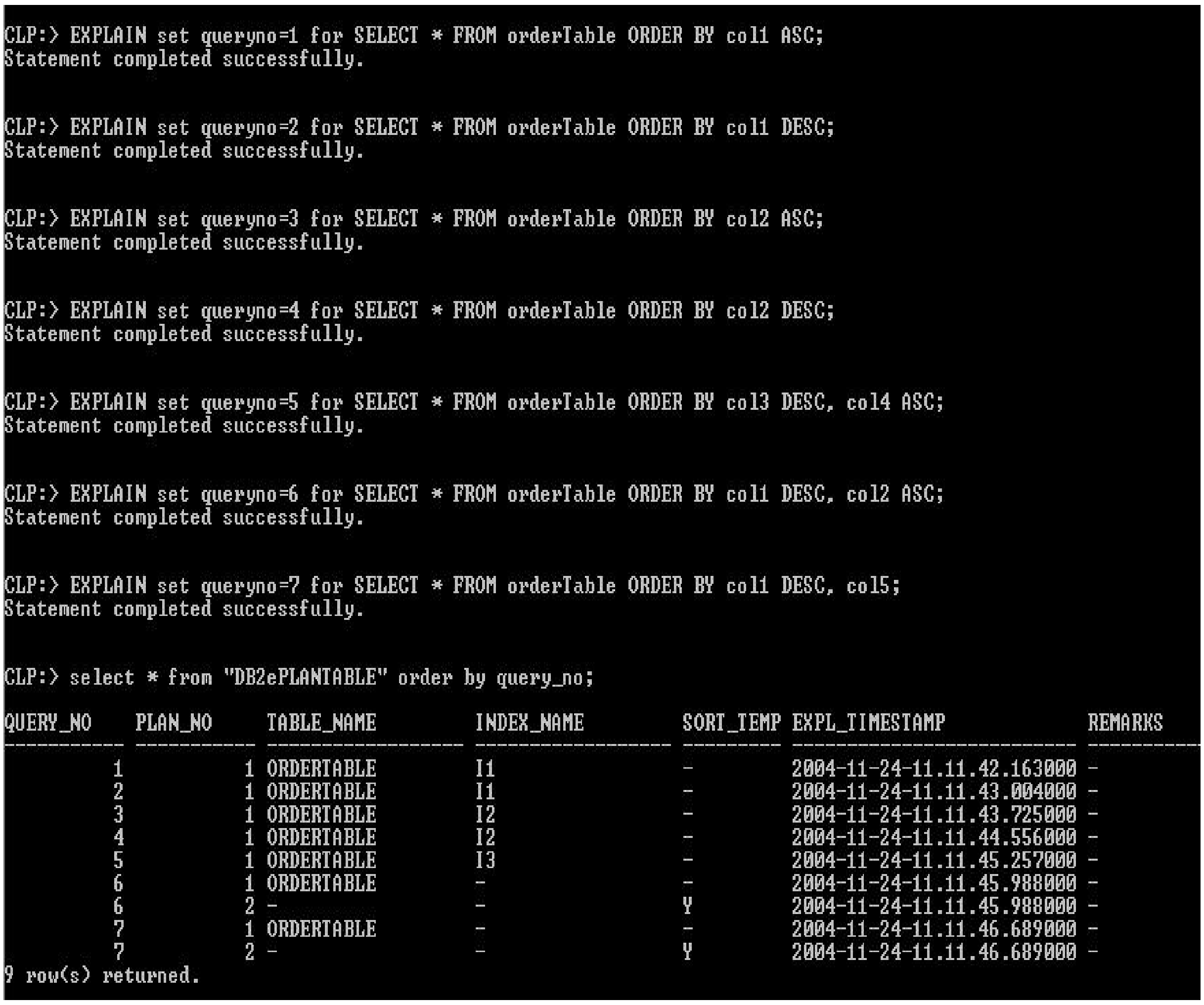
- CREATE TABLE orderTable (pkey int not null primary key, col1 int, col2 int, col3 int, col4 int, col5 int)
- CREATE INDEX i1 ON orderTable(col1 ASC)
- CREATE INDEX i2 ON orderTable(col2 DESC)
- CREATE INDEX i3 ON orderTable(col3 ASC, col4 DESC)

1. SELECT * FROM orderTable ORDER BY col1 ASC
2. SELECT * FROM orderTable ORDER BY col1 DESC
3. SELECT * FROM orderTable ORDER BY col2 ASC
4. SELECT * FROM orderTable ORDER BY col2 DESC
5. SELECT * FROM orderTable ORDER BY col3 DESC, col4 ASC
6. SELECT * FROM orderTable ORDER BY col1 DESC, col2 ASC
7. SELECT * FROM orderTable ORDER BY col1 DESC, col5

First of all *Figure 4* shows how to use the EXPLAIN command. The access plan materialized by the EXPLAIN command is stored in a table called **DB2ePLANTABLE**[6] which will be created if it does not exist. You need to use double quotes if you refer to it in an SQL statement (see the select in the picture). As you can see for the queries 1)-5) from above it neither matters if the order is ascending or descending nor if it is a single or a multi-column index - the indexes i1, i2 or i3 are used for query evaluation and support both scanning directions. However, in case you use a single column contained in another index like in example 6) no index at all will be used even if the used column is part of an index. In addition the sorting happens in a temporary table on the file system as indicated with the "Y" in the **SORT_TEMP** column of the **DB2ePLANTABLE**. A solution of this problem could be the usage of multi-column indexes which have the additional advantage that you need less indexes defined on your table and therefore lead to better insert, update and delete performance. Note that if for a column referenced in an ORDER BY clause no index is available (like for col5 in query 7) above), then the sorting will be done on the file system using a temporary table (again marked with a "Y" in the **SORT_TEMP** column of the DB2ePLANTABLE). Sorting in temporary files on slow I/O hardware will require significantly more time than evaluating the ORDER BY supported by an index (in this case no temporary table is needed). If you use memory cards with PDAs or smart phones make sure temporary table creation for sorting triggered by ORDER BY clauses is avoided if

[6] Note that the DB2ePLANTABLE is not a system table.

possible because many memory cards have slow I/O performance.



```
CLP:> EXPLAIN set queryno=1 for SELECT * FROM orderTable ORDER BY col1 ASC;
Statement completed successfully.


CLP:> EXPLAIN set queryno=2 for SELECT * FROM orderTable ORDER BY col1 DESC;
Statement completed successfully.


CLP:> EXPLAIN set queryno=3 for SELECT * FROM orderTable ORDER BY col2 ASC;
Statement completed successfully.


CLP:> EXPLAIN set queryno=4 for SELECT * FROM orderTable ORDER BY col2 DESC;
Statement completed successfully.


CLP:> EXPLAIN set queryno=5 for SELECT * FROM orderTable ORDER BY col3 DESC, col4 ASC;
Statement completed successfully.


CLP:> EXPLAIN set queryno=6 for SELECT * FROM orderTable ORDER BY col1 DESC, col2 ASC;
Statement completed successfully.


CLP:> EXPLAIN set queryno=7 for SELECT * FROM orderTable ORDER BY col1 DESC, col5;
Statement completed successfully.


CLP:> select * from "DB2ePLANTABLE" order by query_no;

QUERY_NO    PLAN_NO     TABLE_NAME         INDEX_NAME         SORT_TEMP EXPL_TIMESTAMP             REMARKS
----------- ----------- ------------------ ------------------ --------- -------------------------- ----------
          1           1 ORDERTABLE         I1                 -         2004-11-24-11.11.42.163000 -
          2           1 ORDERTABLE         I1                 -         2004-11-24-11.11.43.004000 -
          3           1 ORDERTABLE         I2                 -         2004-11-24-11.11.43.725000 -
          4           1 ORDERTABLE         I2                 -         2004-11-24-11.11.44.556000 -
          5           1 ORDERTABLE         I3                 -         2004-11-24-11.11.45.257000 -
          6           1 ORDERTABLE         -                  -         2004-11-24-11.11.45.988000 -
          6           2 -                  -                  Y         2004-11-24-11.11.45.988000 -
          7           1 ORDERTABLE         -                  -         2004-11-24-11.11.46.689000 -
          7           2 -                  -                  Y         2004-11-24-11.11.46.689000 -
9 row(s) returned.
```

Figure 4: Using the EXPLAIN command

4.2.2) Prefix scanning

For explaining the index prefix scanning let us look at an example again:

- CREATE TABLE myTable (pkey int not null primary key, col1 int, col2 int, col3 int, col4 int, name varchar(50))
- CREATE INDEX myIndex1 ON myTable(col1, col2, col3, col4)

1. SELECT name FROM myTable WHERE col1 = value1
2. SELECT name FROM myTable WHERE ((col1 = value1) AND (col2 = value2))
3. SELECT name FROM myTable WHERE ((col1 = value1) AND (col2 = value2) AND (col3 = value3))
4. SELECT name FROM myTable WHERE ((col1 = value1) AND (col2 = value2) AND (col3 = value3) AND (col4 = value4))
5. SELECT name FROM myTable WHERE ((col1 = value1) OR (col2 = value2))
6. SELECT name FROM myTable WHERE col2 = value2
7. SELECT name, col1 FROM myTable WHERE ((col1 = value1) OR (col2 = value2)) ORDER BY col1

8. SELECT name, col2 FROM myTable WHERE ((col1 = value1) OR (col2 = value2)) ORDER BY col2

- example for using EXPLAIN with DB2eCLP: EXPLAIN SET queryno=1 FOR SELECT name FROM myTable WHERE col1 = value1;
- example to retrieve the access plan from DB2ePLANTABLE with DB2eCLP: SELECT * FROM "DB2ePLANTABLE" WHERE query_no=1;

If you use the EXPLAIN statement (see example above), you will see that the queries 1) to 4) are supported by the index *myIndex1*. This means, with a single multi-column index you can support multiple queries reducing the performance impact on insert, update and delete operations because you only need one instead of four indexes to support all of them. Query 6) outlines why it is called prefix scanning: Only if a complete subset of the first m columns (this is the prefix) starting with the first column in the index definition of an index with n columns are appearing in the WHERE-clause, the index qualifies for query evaluation (in query 6) (col1 is not referenced in the WHERE-clause, therefore no complete prefix can be used and the index does not apply). Query 5) shows another rule that an index must fulfill to be applicable: If there are multiple conditions in the WHERE-clause, only conditions connected by AND operators are applicable for index selection. This means, if a column only appears in conditions coupled with the OR operator, a defined index will not be used. However, if as in example 7) a column is appearing in the WHERE-clause grouped by an OR operator and is appearing in an ORDER BY clause containing the same column, then the index will be chosen again[7], assuming that the prefix rule is not violated in the ORDER BY clause. Query 8) in contrast to query 7) does not use the index because in the ORDER BY clause the prefix rule is violated.

4.2.3) Supporting join conditions with indexes

Computing the result set of a multi table SELECT query means to compute the entire or a subset of cross product operation on all combinatorial possible row combinations. Let us assume you are doing a four table SELECT query on four small tables A,B,C,D each containing 1000 rows with no WHERE clause. The cross product of all possible row combinations would be 1000*1000*1000*1000=1.000.000.000.000 rows in the result set. It is easy to imagine that the computation of this, for example on a PDA, is taking too long for all practical purposes. Besides, in almost all cases you are interested only in those rows in the result set that fulfill additional join or projection operations or both. If such conditions are part of the SELECT, only a subset of the actual cross product might be computed depending on certain factors. Again, indexes are beneficial for computing join conditions. However, an available index must comply with certain optimizer internal rules for being chosen. Let us examine this in more detail in the remainder of this subsection and start again with an example to understand the problem in more detail:

- CREATE TABLE t1 (pkey int not null primary key, col1 int, col2 int, col3 int)
- CREATE TABLE t2 (pkey int not null primary key, col1 int, col2 int, col3 int)
- CREATE TABLE t3 (pkey int not null primary key, col1 int, col2 int, col3 int)
- CREATE INDEX i1 ON t2 (col2)
- CREATE INDEX i2 ON t3 (col3)

[7] The index in this case will be used for the sorting required by the ORDER BY-clause. It is not supporting the OR operator.

1. SELECT t1.col1, t2.col2, t3.col3 FROM t1, t2, t3 WHERE t1.col2=t2.col2 AND t2.col3=t3.col3
2. SELECT t1.col1, t2.col2, t3.col3 FROM t3, t2, t1 WHERE t1.col2=t2.col2 AND t2.col3=t3.col3
3. SELECT t1.col1, t2.col2, t3.col3 FROM t2, t3, t1 WHERE t1.col2=t2.col2 AND t2.col3=t3.col3

As you can see in the queries 1) to 3) the only difference between them is the order of the tables in the FROM-clause. If you use EXPLAIN on each of them, you will see that query 1) uses both indexes i1 and i2, query 2) uses none and query 3) uses only i2. This means there is a difference between the fact that an index *exists* and an index is actually *used* during query execution. For queries containing join-conditions the order of the tables in the from-clause influences the decision if an existing index can be used for computing the result set or not. As you recall from a previous section, join-reordering is not supported by DB2 Everyplace mobile database. Cost-based optimizers are able to perform in many cases a reordering of the tables in the from-clause to increase the number of join-conditions actually using existing indexes.[8] This happens during the optimization of the access plan performed by the SQL optimizer. Join-reordering is a powerful feature for the following reason: If you have n tables, then there are n! permutations possible in the from-clause. *Figure 5* gives you an idea how fast the function n! is growing:

Number of tables	Number of permutations for from-clause: n!
2	2 = 1*2
3	6 = 1*2*3
4	24 = 1*2*3*4
5	120 = 1*2*3*4*5
6	720 = 1*2*3*4*5*6
7	5.040 = 1*2*3*4*5*6*7
…	
11	39.916.800
…	

Figure 5: Illustrating the growth of the n! function

Finding among them a permutation of the tables which is performing well is therefore a not easy to accomplish task. For DB2 Everyplace mobile database here are some guidelines which allow you to still write good performing queries without trying out all the permutations:

- Tables with a small number of rows should be as far as possible to the left in the list of tables in the from-clause.
- If you have n tables and m (m >= n-1) join conditions, at most n-1 of the m join conditions will be supportable by an index. Since in each join condition no more than one index will be used, it is sufficient to have an index created on one of the two columns in the join. This means the creation of an index is useful on at most n-1 columns in n-1 join conditions. For selecting these columns here are some guidelines:
    - Only create indexes on columns with as many different values as possible.

[8] Note that cost-based optimizers normally do a lot more than just trying to increase index usage for complex join queries. For example they analyze the number of rows in intermediate join products and try to keep this number as small as possible.

    - If a table has many rows, the higher the benefits of an index will be in evaluating joins. Given table A with 20 rows, table B with 10.000 rows and the condition A.col1 = B.col2. If col1 from table A has 20 different values and col2 of table B has only 10 different values which appear approximately the same number of times in the column, then create the index on table B. From table B this way approximately 9000 rows are easily identified as not qualifying by using the index significantly improving performance (compared to only 19 in table A if the index would be on col1).
- If the SELECT statement using join conditions has an order by-clause try to support the order by-clause with an index to avoid temporary table creation on the file system for the order operation.

Index support for queries using join conditions is very important. Computing the result set of SELECTs retrieving data from multiple tables using multiple join conditions are among the most resource intensive tasks for a database. In various customer situations where the claim was that queries using join conditions were always running too slow the problem could be easily solved by introducing the proper indexes and reordering the tables in the from-clause of the query so that the created indexes are also used. During development time ensure that SELECTs with a lot of join conditions are using a reasonable amount of indexes.

You got now guidelines when to create indexes for improving query performance. To sum it up these are cases where you should consider proper indexing:

- SELECTs with an ORDER BY clause to avoid temporary table creation involving slow IO
- Multi-table SELECTs with JOIN or projection conditions or both
- Projections
- Conditions connected with AND-operator
- Columns with a high percentage of distinct values compared to the overall number of rows in the table are normally good candidates for an index.

However, index creation has the following drawback:

- Insert, Update and Delete performance will degrade since index tree maintenance has a runtime.

During normal application execution where usually only a few insert, update or delete operations happen due to actions on any application screen (think about the amount of data you expect to change if you input it using a stick or a keyboard on PDA or a smart phone), the time spent for the database work is in many cases only a small fraction compared to a screen refresh or screen switch. This means the overhead for maintaining index trees during insert, update and deletes is small enough compared to the huge gains you can get if indexes can be used to support your queries.

4.3) Design considerations

In this subsection, I briefly introduce a few design considerations you should keep in mind when you are designing database applications using DB2 Everyplace mobile database.

Example 1:

If you design a mobile application, try to design for scalability and try to avoid hard limits imposed by implementation. Imagine you design an application for service technician personal which, from time to time, needs technical information on the mobile device such as images of spare parts or PDF documents with instructions, etc. Now consider the following two table definitions:

- CREATE TABLE docs1 (pkey INT NOT NULL PRIMARY KEY, documents BLOB(32000))
- CREATE TABLE docs2 (pkey INT NOT NULL, piecenumber INT NOT NULL, documents BLOB(32000), PRIMARY KEY(pkey, piecenumber))

If you use the definition of table *docs1*, your application will not be able to handle any documents larger than 32000 bytes because the design assume there will never be a document needed on a mobile device larger than 32000 bytes. Now consider table definition *docs2*: The design in this case assumes that there might be documents larger than 32000 bytes which will be divided into multiple pieces fitting into the column *documents*. The object is stored in multiple rows using unique primary keys using the same value for the *pkey* column and increasing values in the *piecenumber* column. This means with a select like this you can retrieve a document stored in multiple rows:

- SELECT * FROM docs2 ORDER BY pkey, piecenumber

With the order by - clause you will have the pieces in proper order in the *result set* so that combining them in the right order to get the entire document back is fairly easy. With similar considerations you can implement Delete and Update operations for documents larger than the Blob column easily. The only limitation, if you use the design behind the table definition *docs2*, is the available space on the file system.

Example 2:

Let us assume your application has materials which can be divided into eight different groups which you call *type1* to *type8* with value ranges for each group. Now consider the following two table definitions and the following queries:

- CREATE TABLE mat_group(pkey INT NOT NULL PRIMARY KEY, type1 INT, type2 INT, type3 INT, type4 INT, type5 INT, type6 INT, type7 INT, type8 INT)
- CREATE TABLE mat_group2(pkey INT NOT NULL PRIMARY KEY, type CHAR(5), value INT)
- SELECT * FROM mat_group WHERE type1 = 3 OR type3 = 5 OR type7 = 11 ORDER BY type1, type3, type7
- SELECT * FROM mat_group2 WHERE (type = ‘type1’ AND value = 3) OR (type = ‘type3’ AND value = 5) OR (type = ‘type7’ AND value = 11) ORDER BY type

Whenever you need to order the result by the types *type1* to *type8* the second table *mat_group2* only needs one index on the *type* column to achieve this goal whereas for table *mat_group* you will not be able to provide the same functionality with only one single column index for all possibilities of the typeX columns in the order by-clause. You should keep an eye on table designs which allow most queries to run with index usage requiring the minimal number of indexes to maintain to keep the penalty on Insert, Update and Delete operations as small as possible.

Example 3:

Some of the database features require a little more work using them during implementation but offer better performance. Below is an example:

1.
    a. CREATE TABLE test (counter INT NOT NULL, fname CHAR(30) NOT NULL, lname CHAR(30) NOT NULL)
    b. SELECT MAX(counter) FROM TEST
    c. INSERT INTO test VALUES(max+1, 'Joe', 'Smith')
2.
    a. CREATE TABLE test2 (counter INT NOT NULL GENERATED ALWAYS AS IDENTITY, fname CHAR(30) NOT NULL, lname CHAR(30) NOT NULL)
    b. INSERT INTO test2(fname, lname) VALUES('Joe', 'Smith')

If you use option 1 for your implementation and you intend to ensure increasing values in the *counter* column, you will always have the need to issue a query first retrieving the current maximum in the column. Then you need to add 1 in your application to the retrieved maximum before being able to insert a new row with a new maximum value for the *counter* column. This is not the best way to go from a performance point of view. Option 2 shows you how to use a database option – in this case the IDENTIY column feature – to achieve the same goal by avoiding the unnecessary query for the maximum. Note though that the Insert statement is now just a tiny little bit more complex because you are required to specify the column names in parentheses after the table name in which you intend to insert explicitly. Admitting that this is a fairly simple example it shows nonetheless that using the options offered by the database can significantly improve the application performance.

5) Summary

In this whitepaper IBM DB2 Everyplace mobile database was introduced. The architecture of two important supported programming interfaces was described: JDBC and .NET. In the final part of the whitepaper important tuning hints for writing efficient SQL queries were given. Using them, you should now be able to write well performing mobile applications using DB2 Everyplace mobile database.

6) About the author

Martin Oberhofer worked for DB2 Everyplace performance team in the Silicon Valley Laboratories of IBM. In 2003 he joined IBM Germany and is currently working as DB2 Everyplace consultant at SAP. He is interested in mobile and database technology, Java and Linux. You can contact him at martino@de.ibm.com.

7) Bibliography

1. DB2 Everyplace homepage:
   http://www.ibm.com/software/data/db2/everyplace/new81.html
2. DB2 Everyplace library:
   http://www.ibm.com/software/data/db2/everyplace/library.html
3. DB2 Everyplace Performance Tuning Guide:
   http://www.ibm.com/software/data/db2/everyplace/library.html
4. DB2 Everyplace Application Developer Guide:
   http://www.ibm.com/software/data/db2/everyplace/library.html
5. DB2 Everyplace Installation Guide:
   http://www.ibm.com/software/data/db2/everyplace/library.html
6. DB2 Everyplace Architecture details:
   http://mordor.prakinf.tu-ilmenau.de/papers/dbspektrum/dbs-05-09.pdf
7. DB2 Everyplace Success Stories:
   http://www.ibm.com/software/success/cssdb.nsf/softwareL2VW?OpenView&Count=30&RestrictToCategory=db2software_DB2Everyplace
8. FAQ for DB2 Everyplace:
   http://www.ibm.com/software/data/db2/everyplace/support.html
9. DB2 Everyplace Mailing list:
   http://groups.yahoo.com/group/db2everyplace/
10. DB2 Everyplace SDK:
    http://www14.software.ibm.com/webapp/download/product.jsp?s=p&id=JPEN-4HNW2H
11. DB2 Everyplace .NET application development:
    http://www-106.ibm.com/developerworks/edu/dm-dw-dm-0409oberhofer-i.html?S_TACT=104AHW11&S_CMP=LIB%20
12. WebSphere Studio Device Developer homepage to obtain trial version of J9 JVM:
    http://www.ibm.com/software/wireless/wsdd/
13. WebSphere Studio Device Developer library:
    http://www.ibm.com/software/wireless/wsdd/library.html
14. Resources on .NET programming:
    http://www.ondotnet.com/dotnet/