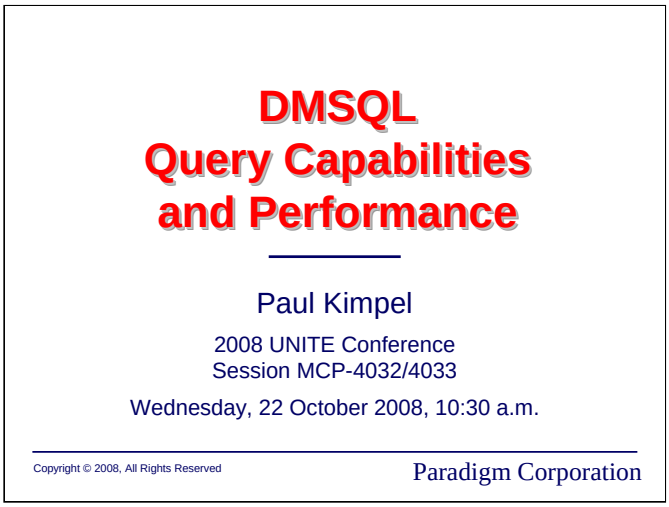


DMSQL Query Capabilities and Performance

2008 UNITE Conference
Orlando, Florida

Session MCP-4032/4033

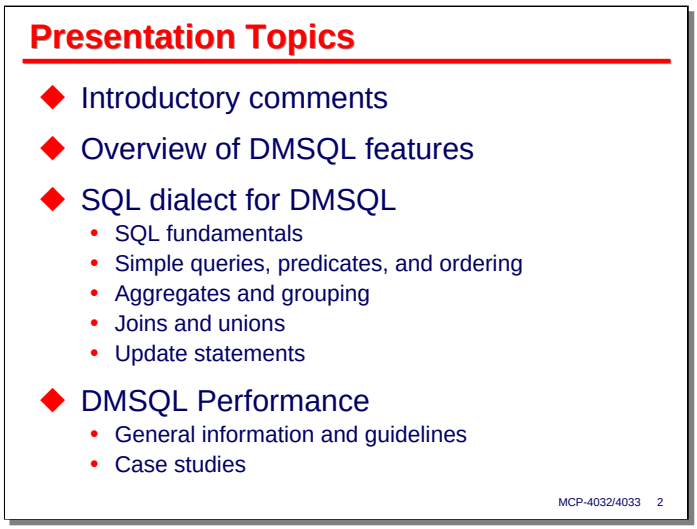
Wednesday, 22 October 2008, 10:30 a.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com

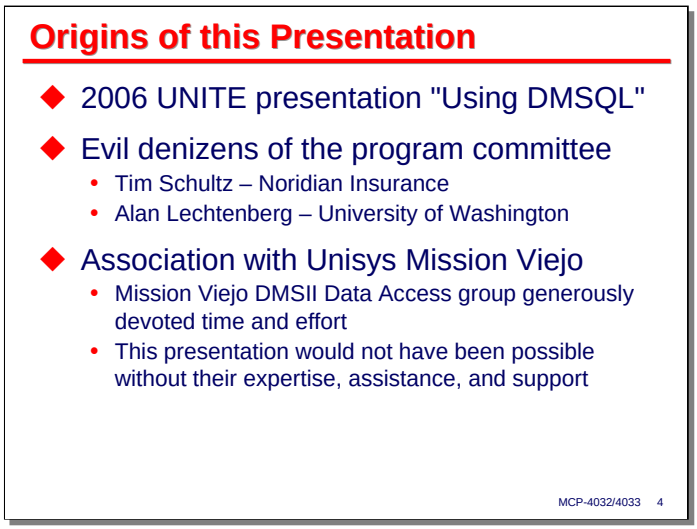


Today I would like to discuss two aspects of DMSQL – its capabilities and its performance for data query. DMSQL is a relatively new feature that implements the SQL data definition (DDL) and data manipulation (DML) languages for the DMSII data management system in the MCP environment.

I will begin with some brief introductory comments and an equally brief overview of DMSQL features. About half of the presentation will be devoted to a discussion of the specific dialect of SQL that DMSQL supports. If you already know SQL, this section hopefully will give you a good idea of what you have to work with in DMSQL. If you don't know SQL, then this section hopefully will give you an idea of what it is like an how it can be used to query data in a DMSII database.

The last section of the presentation will attempt to give you an idea of how well DMSQL performs against other methods of querying DMSII data. In addition to discussing some general information and guidelines, this section discusses a few query case studies.

Let me begin with a few introductory comments on DMSQL



Two years ago at the UNITE conference in Garden Grove, California, I did a presentation titled "Using DMSQL." That was well received and seemed to generate quite a bit of interest in the product.

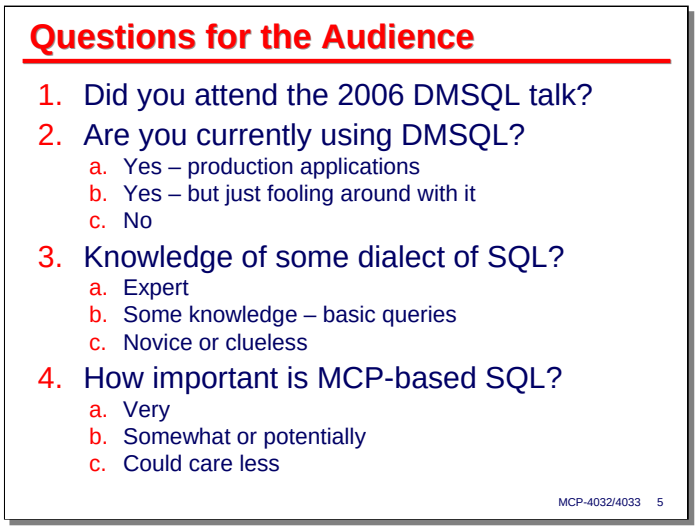
A few of the larger MCP customers seem to be very interested in DMSQL – largely, I think, because of Java, which they want to start using in their client environments. JDBC, the database connector technology typically used with Java, requires a query language interface. DMSQL was developed at least partly to enable JDBC access to DMSII.

Late last year, Tim Schultz of Noridian Insurance expressed an interest in have a presentation on "efficiency and technical issues" involved with accessing a DMSII database using SQL. He and Alan Lechtenberg of the University of Washington are members of the MCP program committee, and Alan approached me with a request that I consider putting together a presentation along these lines.

I tentatively agreed to do so, but expressed a desire to have some assistance from Unisys in two areas – understanding more about the internals of DMSQL and what affects its performance, and access to a MCP system with significant performance capabilities.

This request for assistance wound its way through Unisys, and I was contacted by the DMSII Data Access group at the Mission Viejo facility. Over the past couple of months, they have generously devoted time and effort to supporting my work in developing this presentation, and have shared with me quite a bit about performance considerations for DMSQL. They were also able to arrange access to a Libra 300-20 system for part of that time, on which I was able to run some timing tests.

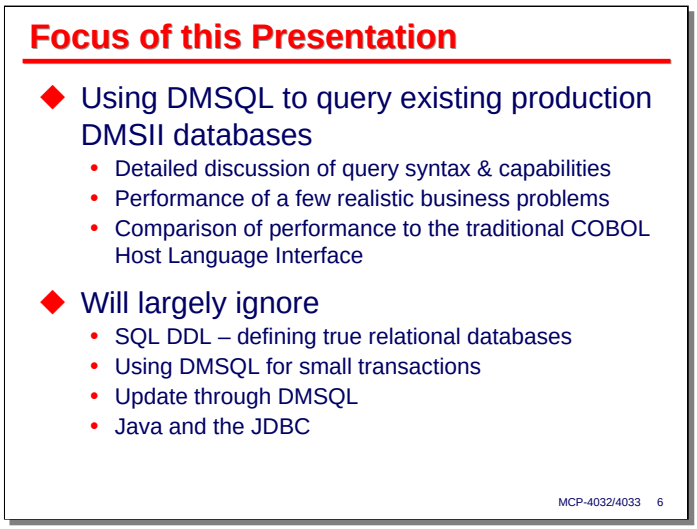
Two people from the Mission Viejo group were particularly generous with their time. They have asked not to be mentioned by name, but without their expertise, assistance, and support, this presentation would not have been possible.



Before going further, there are some questions I would like to ask of the audience. This will give me a better feel for the interest others have in DMSQL and how far along they are in the process of integrating it into their applications.

In addition to a show of hands, I would appreciate it if you would write your answers on your session evaluation cards, either in the comment area on the front, or on the back of the stiffer of the sheets for this form. You can just use the numbers and letters shown on this slide to identify your answers.

[Poll the audience here]

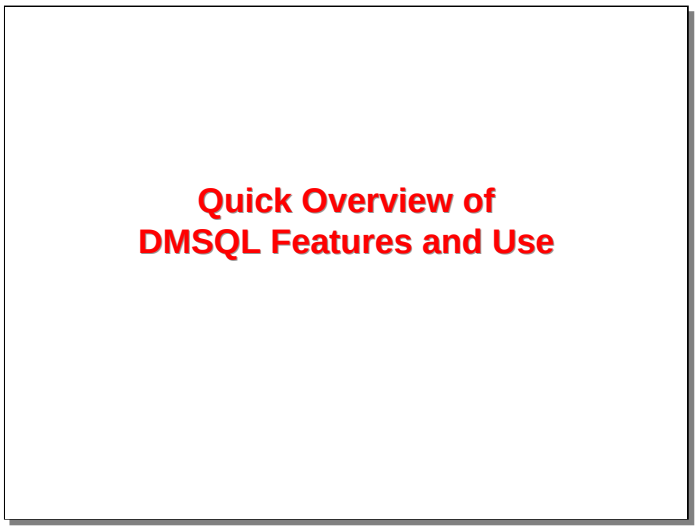


The focus of this presentation is primarily on the query of existing production DMSII databases, and in particular

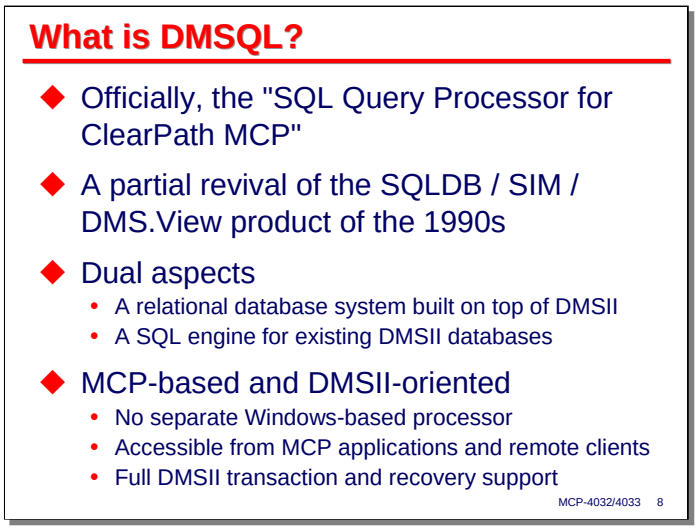
- Discussing the detailed syntax of the SQL dialect that DMSQL supports.
- The capabilities of that dialect of SQL.
- Performance of a few queries that are intended to be representative of realistic business problems.
- Comparison of the performance of those DMSQL queries against the performance of other methods for querying that same data.

DMSQL has capabilities beyond just querying existing DMSII databases, and there are other important issues involving its use. In this presentation I will largely ignore such issues, specifically:

- Using SQL DDL to define true relational databases on top of DMSII.
- Using DMSQL for small transactions. The case studies in this presentation focus on larger data extract and business reporting problems.
- Updating databases through DMSQL.
- Using Java and the JDBC with DMSQL.



Next, let's take a quick look at the major features of DMSQL and how they are used.



Officially, DMSQL is known as the SQL Query Processor for ClearPath MCP. It is a partial revival of the SQLDB/SIM/DMS.View product that was developed and available on MCP systems in the 1990s.

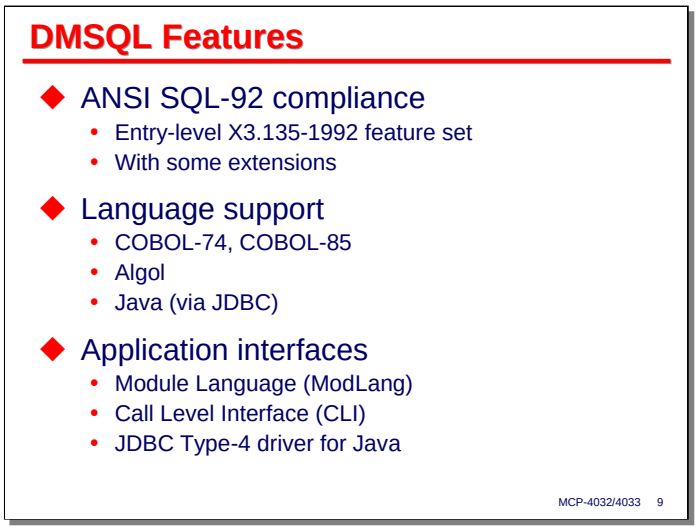
DMSQL has two main aspects.

- It can be used to to define and manipulate a true relational database. This relational database is defined using DMSII constructs and, underneath DMSQL, runs as a standard DMSII database.
- It can function as a SQL query and update engine for existing DMSII databases. Most DMSII data items and structures can be mapped to a relational model for the database. DMSQL operates against that relational model. This relational mapping was discussed in some detail in my 2006 UNITE presentation.

Another important aspect of DMSQL is that it is entirely MCP based and was designed specifically for use with DMSII. There is no separate Windows-based processor involved, as there is with the Data Access ODBC product, or with access to DMSII databases through OLE DB from Microsoft SQL Server.

DMSQL supports interfaces for both MCP-resident applications and remote clients. The primary remote interface at present is the Type-4 JDBC driver.

Since DMSQL runs on top of DMSII and accesses the database through DMSUPPORT and the Accessroutines, all DMSII locking and integrity checking is maintained. DMSQL fully supports DMSII transactions and recovery.

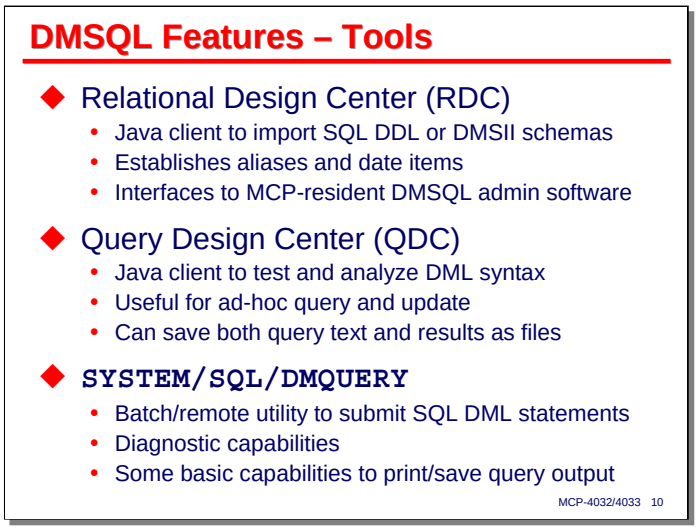


DMSQL supports the ANSI X3.135-1992 Entry Level feature set, plus a few extensions. It is qualified for use with programs written in COBOL-74, COBOL-85, and (through the Type-4 JDBC driver) Java. At least in principle, it could be accessed from any MCP-based language that can call a server library.

DMSQL supports three application programming interfaces:

- The Module Language, also known as ModLang.
- The Call Level Interface, also known as CLI.
- The JDBC Type-4 driver for Java. There was a Type-2 JDBC driver previously, but this is now obsolete, and Unisys recommends that you use only the Type-4 driver, which is a pure Java implementation.

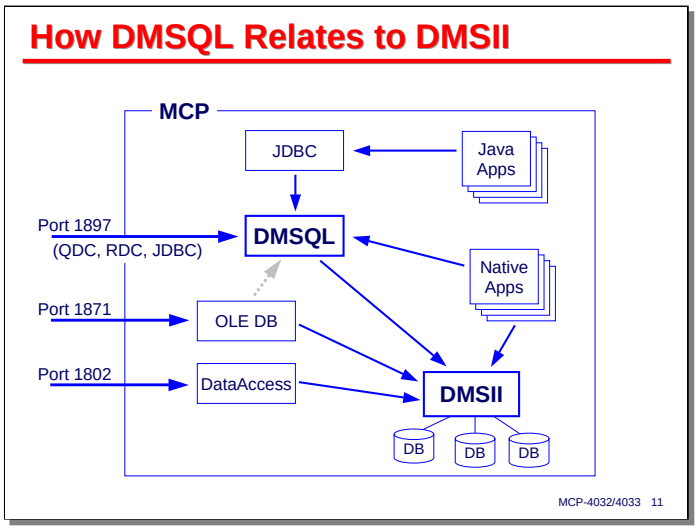
I'll have more to say about ModLang and CLI shortly.



There are three main user interface tools available in DMSQL:

- The Relational Design Center, or **RDC**, is a workstation-based Java application. The RDC can be used to import a file of SQL DDL statements to define a true relational database. It can also be used to import the schema for an existing DMSII database and create a relational mapping for that database. In the latter case, some minor modifications to the DMSII schema can be made – particularly, it can be used to assign alias names for DMSII tables and data items, and it can identify fields in a table that are mapped to virtual SQL date columns.
- The Query Design Center, or **QDC**, is a second workstation-based Java application. It can be used to design and analyze SQL statements, either by entering the SQL text directly into the program, or by constructing a SQL statement using a Query By Example (QBE) mode. The SQL text can be saved to a standard client-side text file and recalled again at a later time. The results of queries can be viewed in a data grid, and the contents of the data grid can be saved in a few standard formats, including comma- and tab-delimited text files. The QDC provides access to some diagnostic information available from DMSQL. It is also useful simply as an ad hoc query and update tool.
- **`SYSTEM/SQL/DMQUERY`** is an MCP-resident utility with much the same capability as the QDC. It can be invoked in both batch and interactive modes. It also has some basic capability to print or save query output in the MCP environment.

The RDC and QDC have been qualified to run on both Windows and Linux systems.

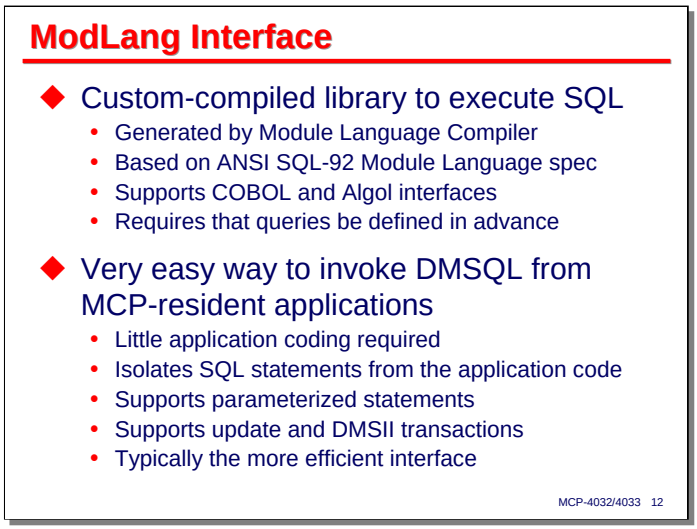


I mentioned earlier that DMSQL runs on top of DMSII. This diagram shows a high-level view of how DMSQL relates to DMSII.

Native MCP applications can access DMSQL directly through server library interfaces exposed by the ModLang and CLI facilities. MCP-resident Java applications access DMSQL through a host-resident JDBC driver.

Remote applications, including QDC, RDC, and Java applications invoking the JDBC driver, access DMSQL through a DSS interface. This DSS interface in turn initiates worker tasks to connect to a DMSII database.

DataAccess ODBC and the OLE DB Provider have similar TCP/IP interfaces to MCP-resident agents that connect to DMSII databases. One potential capability is to interface the OLE DB Provider with DMSQL. This would support submitting SQL queries directly to the MCP host through OLE DB, rather than first going through Microsoft SQL Server and accessing the OLE DB provider through a linked server, as is currently necessary.



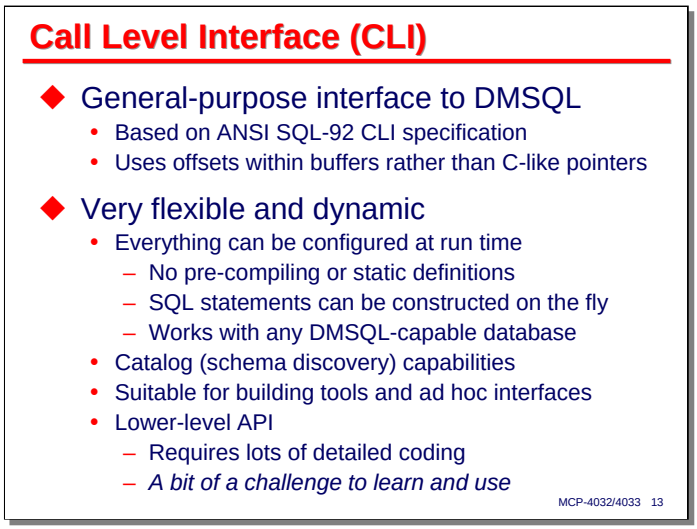
The Module Language interface is one of two APIs that MCP-resident applications can use to access DMSQL and invoke SQL statements against DMSII databases. This capability is based on the SQL-92 standard's Module Language specification.

With ModLang, SQL statements are embedded in a separate Module Language module and compiled into a standard MCP server library. The Module Language syntax permits definition of cursors and procedures. The procedures are wrappers for individual SQL statements. These statements can manipulate cursors or execute certain SQL statements directly. Once the module is compiled into a server library, MCP COBOL and Algol programs can call these procedures as entry points in that library. The application programs pass arguments to the entry point procedures to supply parameters to and return results from the SQL query and update statements.

Because Module Language statements are pre-compiled into a library, use of ModLang requires that you be able to define your query and update statements in advance. Again because the ModLang is pre-compiled, it is typically the more efficient of the two application interfaces.

The ModLang is very easy to use. You need to declare the library and its entry points in your application program, along with any variables in your program that will be passed as parameters. One advantage of the ModLang is that MCP application programmers do not need to know SQL in order to use it. The modules can be prepared by someone who has SQL skills. Application programmers only need to know the names and parameter sequences for the procedures in the module library.

The Module Language has full support for DMSII update, transactions, and database recovery.

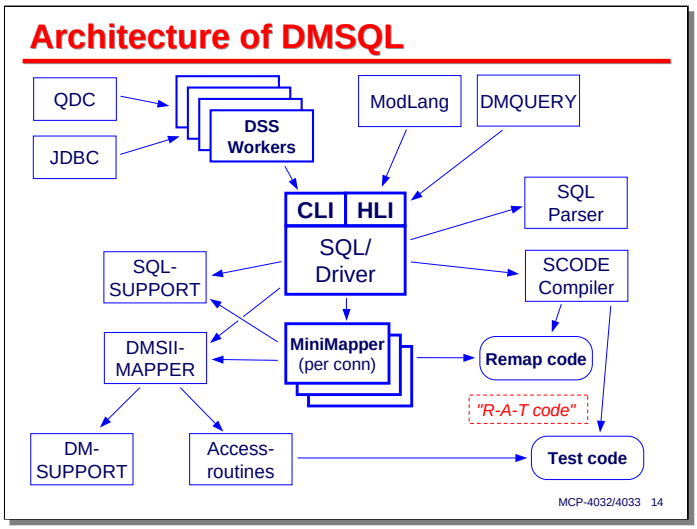


The Call Level Interface is the second application programming interface to DMSQL available for use by MCP-resident programs. It is also based on a portion of the ANSI SQL-92 standard. The parameters for the API calls have been changed somewhat from the standard's specification to allow CLI to be used with COBOL and Algol programs in the MCP environment. In general, these changes involved replacing pointers (addresses) with offsets to fields within a buffer area.

The CLI is very flexible and dynamic. Every aspect of its interface to DMSQL can be configured and activated at run time through API calls. There is no predefinition or pre-compilation of SQL statements. In fact, SQL statements can be generated on the fly at run time, be prepared, and then be executed by the application program. Results of queries can be returned to the application either a field at a time, or a whole row at a time through a mechanism known as column binding.

Because the CLI is so dynamic, it can potentially connect to any DMSQL-enabled database, and is suitable for building general purpose tools. In addition to dynamic query generation, the CLI can perform schema discovery on a database, determining at run time the tables, fields, indexes, and other elements comprising the database definition.

The price for all of this capability and flexibility is that the CLI is a lower-level API than the Module Language. It requires quite a bit of detailed coding to set up and call the API routines. As a result it is something of a challenge to learn and use.



This slide shows a fairly detailed schematic of the architecture of DMSQL.

At the center of the DMSQL implementation is a library called the Driver (**`SYSTEM/SQL/DRIVER`**). This module controls the execution of SQL statements and invokes the other MCP-resident elements.

The driver supports two APIs. The first is the Call Level Interface (CLI) discussed previously. This interface can be invoked directly by MCP applications. It is also the interface used by the DMSQL DSS worker tasks to service requests from the QDC utility and Java JDBC connections.

The second Driver API is the High-Level Interface (HLI). This is an internal interface used only by Unisys software. The server libraries compiled from ModLang modules use this interface, along with the **`SYSTEM/SQL/DMQUERY`** utility program.

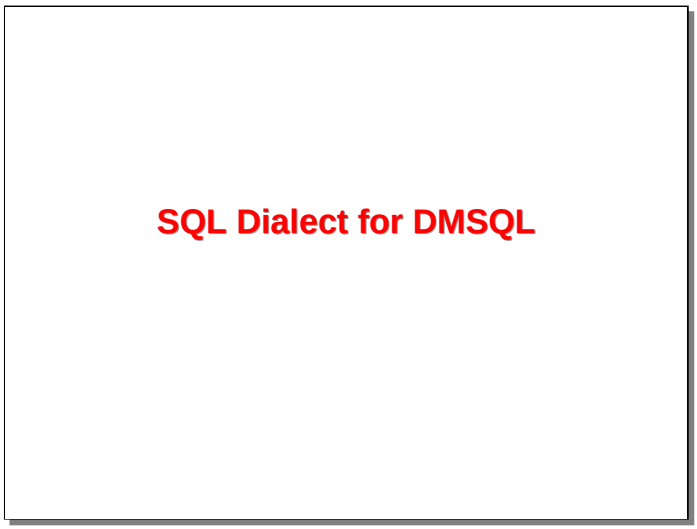
The DMSQL Driver invokes a number of other libraries to support the processing of SQL statements. The Parser is responsible for analyzing SQL text and translating it to an internal form the rest of the system can use. The SCODE compiler translates portions of the query specification into a pseudo-code and then compiles that pseudo-code into native E-mode machine language. These compiled elements fall into two categories:

- "Remap" code, which handles the transfer (and where necessary, conversion) of data values from DMSII record areas to internal DMSQL workareas and the memory areas passed as parameters by application programs.
- "Test" code, which is used to filter records during the processing of a SQL statement (e.g., to implement the predicate conditions for a **`WHERE`** clause). This code is called directly by the Accessroutines to minimize the number of records that need to be passed back to the query engine for further processing.

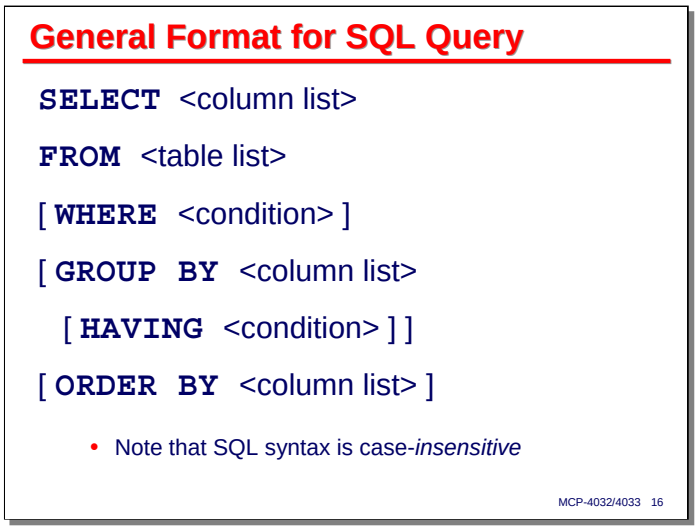
Together, the Remap and Test code segments are known as "R-A-T" code. Compiling these elements into native E-mode instructions eliminates the overhead of having to interpret pseudo-code and enhances the efficiency with which DMSQL can execute SQL statements.

When an application program (or a DSS worker task) opens a connection to a DMSQL-enabled DMSII database, the Driver initiates a small private library for that connection known as the "Mini-Mapper." This entity is primarily an environment to hold the SIB (Set Information Block – the data structure the DMSII uses to support access to a database from an application program) and the state for the connection. The Mini-Mapper in turn connects to a library called DMSIIMAPPER (**`SYSTEM/SQL/DMSIIMAPPER`**) which translates the SQL data retrieval and update requests to the necessary DMSII operations. DMSIIMAPPER calls entry points in DMSUPPORT and the Accessroutines, which perform the actual accesses against the database.

Another important library is **`SYSTEM/SQL/SUPPORT`**, which supplies service routines used during SQL statement execution. For example, string functions and the pattern matching algorithm for the **`LIKE`** predicate are implemented in this library.



With that background, let us now look at the specific dialect of SQL that is supported by DMSQL.



The SQL statement that queries and retrieves data from a database is called the "select" statement. It has six clauses, of which only the first two are required.

The **SELECT** clause defines the values that are to be returned from the query. The result of a query is always a rectangular table (strictly speaking, a "relation") with columns and rows. You can think of the columns as fields and the rows as records. The table is sometimes called a *result set*. Therefore, the **SELECT** clause defines the columns that will make up the result set.

The **FROM** clause names the tables from which the query will search for and retrieve data. In DMSII terms, a table is a data set. A "view" (a predefined query specification) can be used instead of the name of a physical table. As with the result set, SQL considers these tables to be composed of rows with columns. The values in a column for a table must all be of the same data type. Note that the **FROM** clause can name multiple tables. This extremely important and powerful capability, called a "join," will be discussed shortly.

The **WHERE** clause filters records from the tables based on a Boolean condition. This condition applies tests, called "predicates," to columns in rows of the tables. If condition evaluates to true, rows from the tables are processed further; if the condition evaluates to false, those rows are ignored.

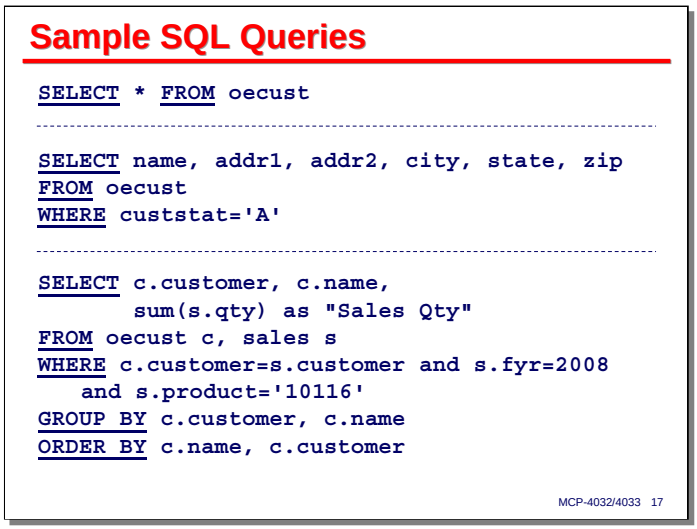
The **GROUP BY** clause is used, along with constructs called *aggregate functions* to summarize data, e.g., to compute subtotals or averages over a group of records.

The **HAVING** clause is an option of the **GROUP BY** clause, and allows further filtering of results after they have been grouped.

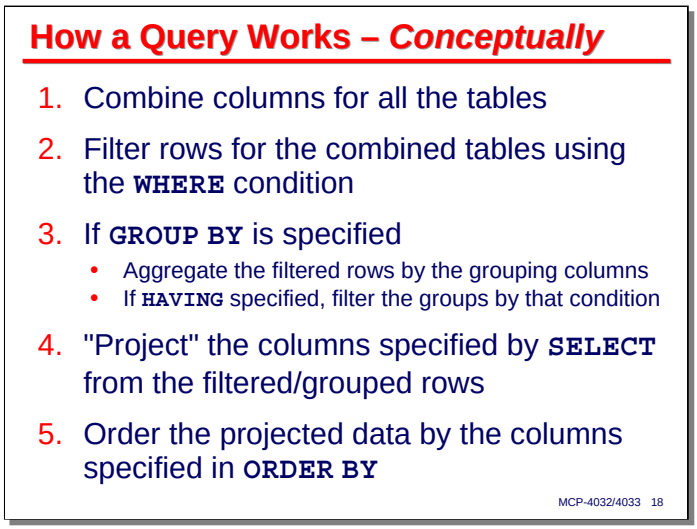
Finally the **ORDER BY** clause can be used to sort the result set by any combination of its columns.

We will discuss each of these clauses in more detail in the following slides and describe how they can be used to retrieve and process data from a database.

One final note – SQL is case insensitive. Keywords, table names, column names, and other syntax elements can be written in any mix of upper- and lower-case letters. The contents of literal strings (which in SQL must be written using single quotation marks), of course are case sensitive.



This slide shows three SQL query statements, progressing from very simple to more complex. I am showing these just to give you an idea of what a SQL SELECT statement looks like and how the various clauses appear. We will discuss how each clause can be constructed and more about what it does as we go along.

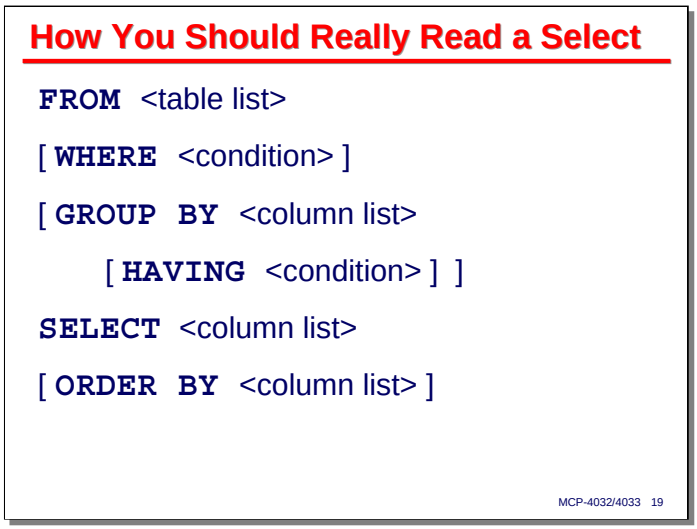


One thing about SQL that can be confusing, especially when you are new to the language, is just how the various clauses interact. SQL is a non-procedural language – you specify the type of result you want, not how to go about achieving that result. Therefore, you need to understand the conceptual model behind evaluation of a SQL query. It turns out that – *conceptually* – a SQL **SELECT** statement is not evaluated in the order that it is written.

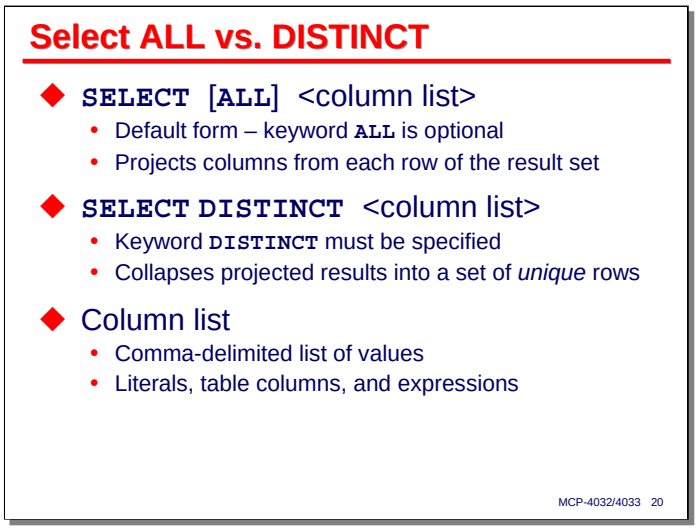
- First, all columns of all rows specified in the **FROM** clause are combined using something called the Cartesian Product. We will discuss this in more detail in the section on joins, but basically, this means combining every row of every table with every row of every other table, appending the columns of each table, to form a virtual working table. The size of this virtual table is the product of the number of rows in each table by sum of the number of columns. If this seems a ridiculous way to go about retrieving data, remember that this is a *conceptual* description, not how it's actually expected to work.
- Second, the condition specified by the **WHERE** clause is applied to each row of this virtual table, exactly *once*. Any of the rows which do not satisfy this condition are discarded from the virtual table.
- Next, if **GROUP BY** was specified in the query, the rows are arranged according to the grouping specification and the grouped results replace those in the virtual table.
- If the **HAVING** clause was specified as part of **GROUP BY**, the grouped results are filtered by the **HAVING** condition. Any groups which do not satisfy this condition are eliminated from the result
- Now the **SELECT** clause is applied to the virtual table to select the columns that will form the result set. This is termed the "projection" of the query, in that the result columns are projected from the virtual table to the result set.
- Finally, if the **ORDER BY** clause was specified, the rows of the result set are ordered by the specified columns.

Once again, this is a *conceptual* description of how a SQL query is evaluated. If SQL query processors actually worked this way, SQL would never have made its way out of the lab and become the dominant database retrieval technology it is today.

The important thing to remember is that SQL queries are driven by their **FROM** clause, and the **SELECT** clause takes effect fairly late in the process. In this sense, the **SELECT** statement is misnamed – it could more properly be referred to as the **FROM-WHERE** statement.



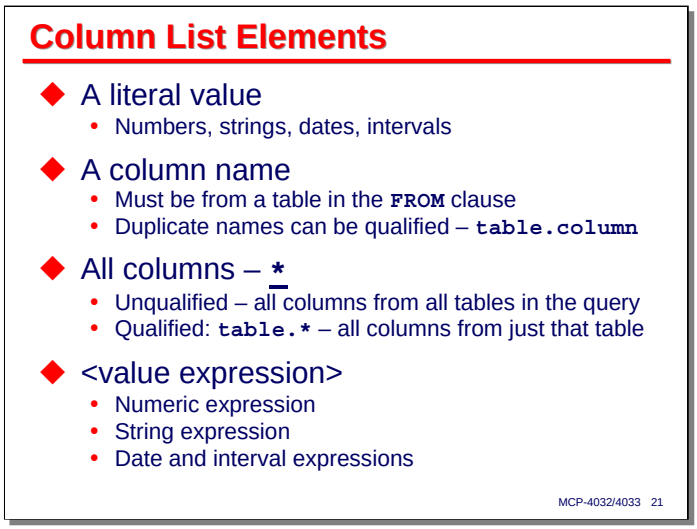
Based on that conceptual view of a SQL query, here is how you should analyze a SQL **SELECT** statement. Start with the **FROM** statement to understand where the data is coming from, then apply the **WHERE** statement to understand which records will be retrieved. The result of the **SELECT** statement will depend primarily on those first two clauses.



The **SELECT** clause actually has two forms:

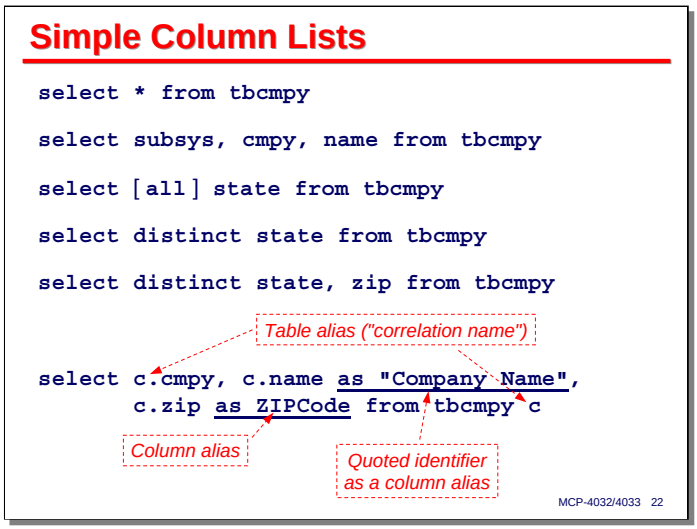
- **SELECT ALL** – this is the default form, and the keyword **ALL** may be omitted. With this form, all rows generated by the query appear in the final result set.
- **SELECT DISTINCT** – this form differs from **SELECT ALL** in that if the result set has duplicate rows (i.e., two or more rows where the corresponding columns have equal values), those duplicate rows are collapsed into a single row. The final result set is guaranteed to contain only unique rows.

Both forms of **SELECT** clause define the list of columns that will appear in the final result set. This list is specified as a comma-delimited list of values.



A column list is composed of elements that can take one of four forms:

- **Literal values** – literals can represent numbers, character strings delimited by single quotation (') marks, dates, and intervals. We will discuss the format of date and interval literals later.
- **Column names** – these are typically the most commonly-used elements in column lists. The columns must be from the tables or views named in the **`FROM`** clause. If two or more tables have columns with the same name, those columns must be qualified by their table or view name in the form **`tableName.columnName`**. As long as there is no conflict in names among the tables referenced in a query, this qualification by table or view name does not need to by done.
- **`*`** – the asterisk is shorthand for "all columns." Used without qualification, it causes the names of all columns from all tables in the **`FROM`** clause to be inserted into the column list. These column names are inserted in the order the tables are named in the **`FROM`** clause, and in the order they are defined in their respective tables. When the asterisk is qualified by a table or view name (**`tableName.*`**), it causes the columns of just that table or view to be inserted into the column list, again in the order that columns are defined for that table or view.
- **Value expressions** – SQL supports a number of types of expressions. You can perform calculations and other manipulations on table columns and literal values and use the result of those as elements in a column list. We will discuss the details of these expressions a little later.



This slide shows some simple **SELECT** statements and their column lists. Note that **SELECT** with or without the **ALL** keyword will allow the result set to contain duplicate rows. **SELECT DISTICT** will collapse any duplicate rows and return a set of unique rows.

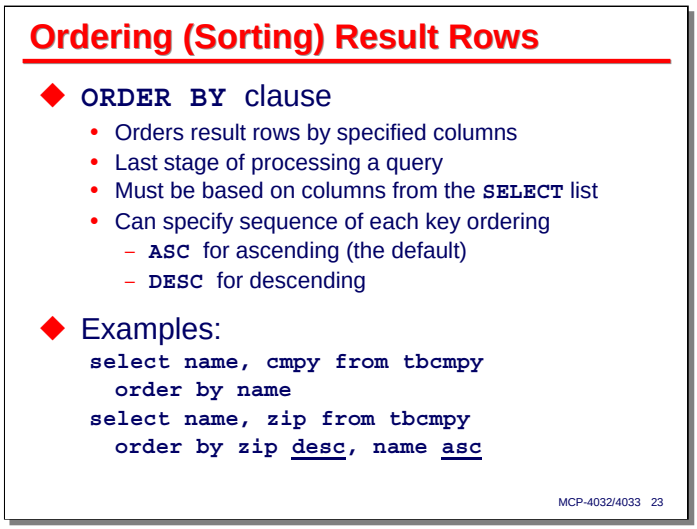
This slide also shows two important features of select lists. First, each column element may be assigned an alias name, prefixed by the keyword **AS**. In some SQL dialects the **AS** keyword is optional, but in DMSQL it is required. The alias can take two forms:

- A simple SQL identifier. This must not conflict with any unqualified column names, SQL keywords, or other aliases used in the query.
- A "quoted identifier." This is a string delimited by *double* quotes ("). This string can contain spaces and punctuation marks, and can conflict with SQL keywords. Unlike most of SQL, quoted identifiers are case sensitive.

Column aliases are useful for naming expressions, supplying a more meaningful name for a result column, and avoiding conflicts caused by duplicate columns among multiple tables.

The second feature is table aliases, sometimes called "correlation names." Table and view names in the **FROM** clause can be followed by a SQL identifier or quoted identifier that will be used as an alias name. Note that some SQL dialects allow the use of the keyword **AS** between the table or view name and the alias, similar to the way that keyword is used for column aliases. DMSQL does not permit the use of **AS** in this context.

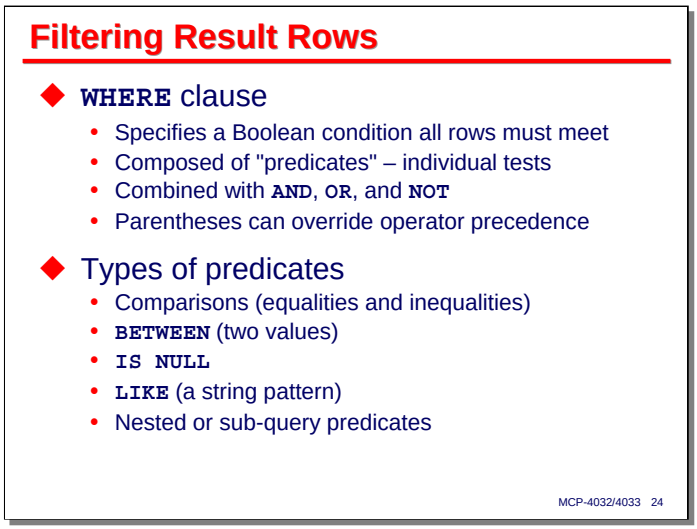
Table aliases may be used in place of the table name as qualifiers to resolve duplicate column names. Note that once a table or view is assigned an alias, that alias must be used throughout the rest of the query syntax to refer to that table.



The **ORDER BY** clause is one of the simplest of the query clauses to understand. It simply orders the rows of the final result set by the list of column names (or column aliases) specified. The first column specified is the most-major key of the sort; the last column specified is the most-minor key. As discussed earlier, this is conceptually the last clause to be applied to the generation of the result set.

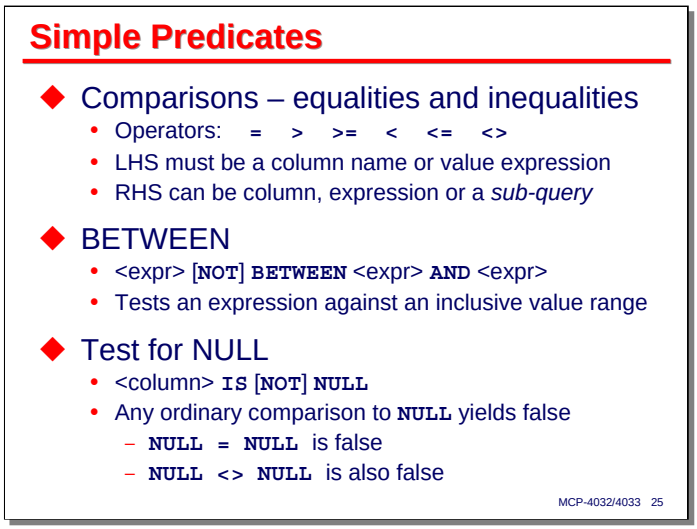
Note that the result set can only be ordered by columns that appear in the **SELECT** clause.

Be default the columns are sorted in ascending order. You can explicitly specify this by following a column name in the **ORDER BY** list with the keyword **ASC**. A column can also be sorted in descending order by following it with the **DESC** keyword.



The **WHERE** clause determines which rows from the tables in the **FROM** clause will be used to construct the result set. It specifies a Boolean condition which all rows must satisfy in order for them to be included. This condition is made up of individual tests, or *predicates*. These predicates may be combined using the Boolean **AND**, **OR**, and **NOT** operators, which have their customary meaning. **NOT** has the highest precedence, followed by **AND**, and then **OR**. Parentheses can be used to override this default precedence, and are recommended for clarity when **AND**s and **OR**s must be mixed in a condition.

There are several types of predicates. Comparisons (equalities and inequalities) are the most common, but SQL provides some additional types which are both powerful and interesting. The various types of predicates will be discussed on the next several slides.



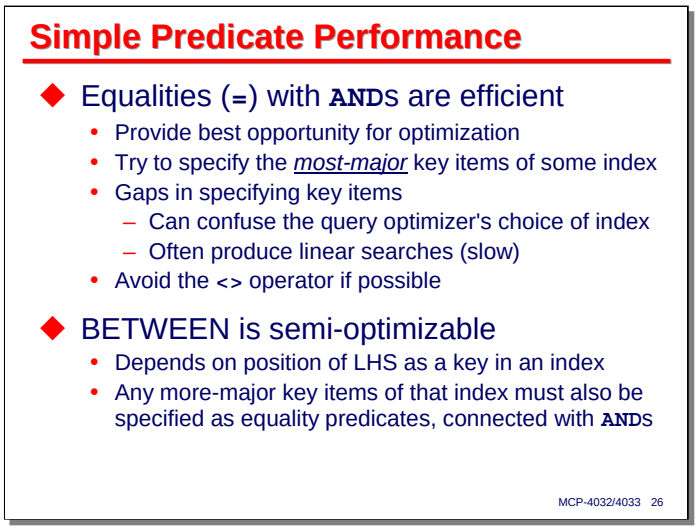
Comparison predicates are the most common type, consisting of the six common relational operators equal (**=**), greater than (**>**), greater than or equal to (**>=**), less than (**<**), less than or equal to (**<=**), and not equal (**<>**). Of these, equalities (**=**) are by far the most common in most applications.

The left-hand side (LHS) of a comparison predicate may be a column name or value expression. The right-hand side (RHS) may be a column name, expression, or a value determined by a nested query, or sub-query. Sub-queries will be discussed in more detail shortly.

The comparison predicates can operate on numbers, strings, dates, and intervals, but both the LHS and RHS must be of compatible types. SQL provides a **CAST()** operator to convert between types, which can be used to alter the type of one side of the comparison to make it compatible with the other.

The **BETWEEN** predicate is a convenient shorthand for a combination of the **>=** and **<=** predicates. It determines if the value of an expression is inclusively between two other values. **NOT BETWEEN** can be used to test for a value that is not within the inclusive range.

SQL implements a special predicate to test for the null value. This predicate is necessary, as comparing anything to a null value yields false. Comparing null to null in any fashion also yields false. The **IS NULL** predicate can only operate on a table column; it cannot operate on a literal or expression. **IS NOT NULL** can be used to test that a column is not null.

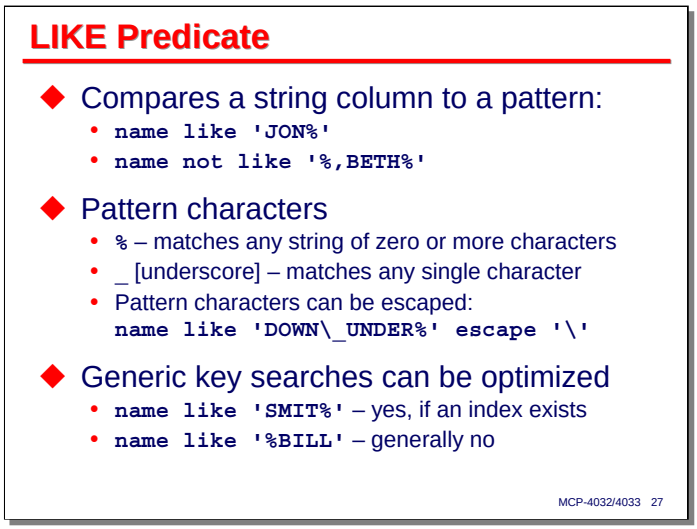


When using comparison predicates, keep in mind that equality tests connected by **AND** operators are usually the most efficient, and generally provide the best opportunity for the query processor to optimize the retrieval of data using an index.

WHERE clauses are generally most efficient when they can specify a set of the *most-major* key items from an index. Doing so will allow DMSQL to request that DMSII use its native index search mechanism to locate records. If there are gaps in the sequence of key items (i.e., the predicates do not specify a contiguous set of items as defined for the key of an index), this can confuse the query optimizer and may result in a linear search of the table (or in more records retrieved through an index than would minimally be necessary). Such cases generally run much slower than if a contiguous set of most-major key items can be specified.

You do not normally need to specify which index for a table is to be used (although there is a way to do that we will discuss later). DMSQL will analyze the entire **WHERE** clause (and the join conditions in the **FROM** clause, if any) and try to determine the best index to use for the retrieval.

BETWEEN predicates may or may not be optimizable, depending on whether there is an appropriate index for the subject of the BETWEEN operator and the presence of all the more-major key items for that index as equality predicates, with everything connected by **AND** operators. Index optimization generally proceeds as if **`A BETWEEN B AND C`** had been written **`(A>=B) AND (A<=C)`**.



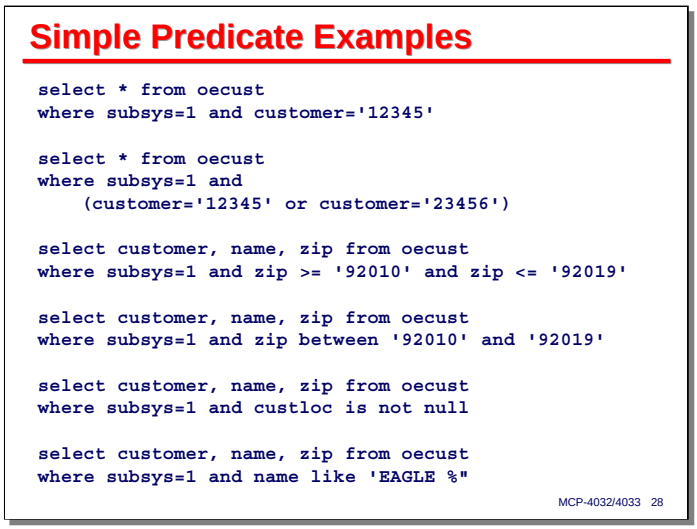
The **LIKE** predicate provides an extremely useful capability, especially for retrievals based on names. It tests the value of a column (not an expression) against a wild-card pattern. If the name matches the pattern, the predicate yields true.

The pattern is a literal string that can contain two types of wildcard characters:

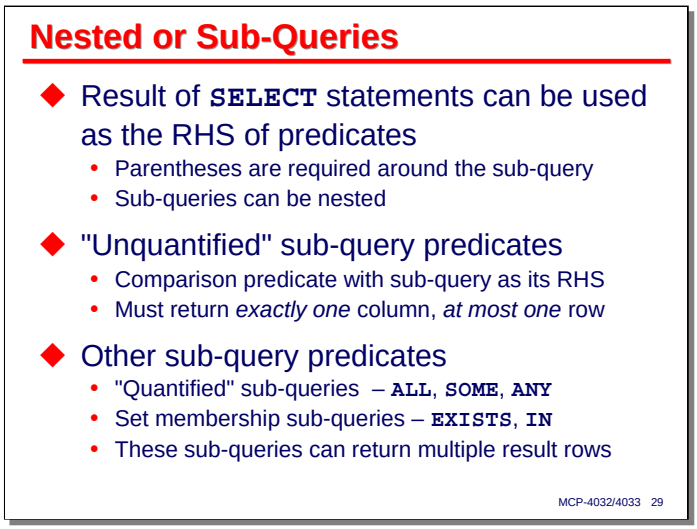
- **%** – matches to any sequence of zero or more characters.
- _ (underscore) – matches to any single character.

If the pattern needs to specify one of the wildcard characters literally, that character may be escaped by prefixing it with another character. That escape character is specified after the pattern string using the keyword **ESCAPE**. As shown in the example on the slide, the pattern contains an underscore as a literal character and not as a wildcard.

DMSQL does a good job of optimizing so-called "generic" key searches if there is an appropriate index for the column being searched. In order for this optimization to take place, the pattern must have some number of literal characters before the first wildcard character. Given an ordered index (such as index sequential or ordered list) on the column being tested, DMSII can quickly locate the sequence of index entries that have the leading literal characters of the pattern string as a prefix. Patterns that begin with a wildcard character generally result in a linear search, as the index cannot be efficiently applied to narrow the set of records that need to be examined.



This slide shows several examples using simple comparison, **`BETWEEN`**, and **`LIKE`** predicates.



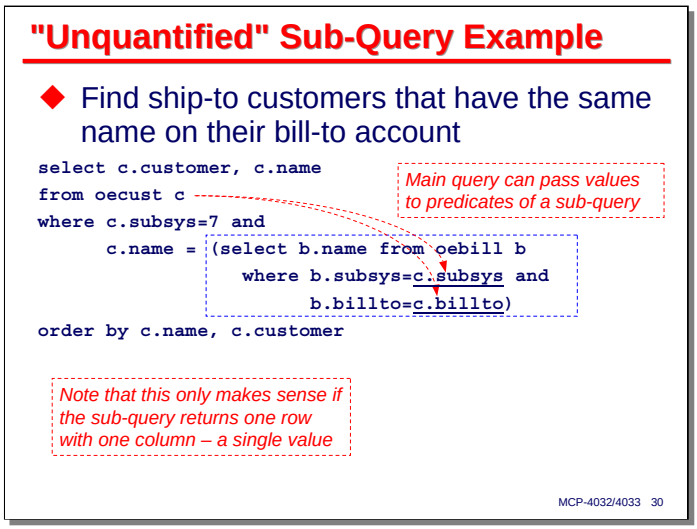
One of the more powerful features of SQL predicates is the ability to use a nested, or sub-query as the right-hand side of a predicate. The sub-query is a complete **SELECT** statement, sans any **GROUP BY**, **HAVING**, or **ORDER BY** clauses. The text of the sub-query must be enclosed in parentheses. The purpose of the sub-query is to supply one or more values that will be used in evaluating whether the predicate yields true or false.

Sub-queries can be nested. That is, a sub-query may have a predicate which itself references a sub-query.

There are three main types of sub-queries:

- **"Unquantified" sub-query predicates** use a single value supplied by the sub-query as the RHS of a comparison predicate. As such, these sub-queries must be defined to return a single-column, single-row result. The query processor will report a syntax error if the query has more than one result column and a run-time error if the query returns more than one row.
- **"Quantified" sub-query predicates** look similar to unquantified ones, but instead of comparing the value of an expressing against a single value returned from the sub-query, these predicates compare the expression against a set of values returned by the sub-query. They are *quantified* in that the comparison must hold true either for every value returned by the sub-query (the **ALL** keyword) or at least one of the values (the **ANY** and **SOME** keywords).
- **Set membership sub-query predicates** (identified by the **EXISTS** and **IN** keywords) test whether the sub-query returns any results at all or whether the value of an expression is a member of a set of other values.

There is no restriction on the number of rows that can be returned by sub-queries for set membership and quantified predicates, although all but the **EXISTS** query are restricted to returning a single column result.

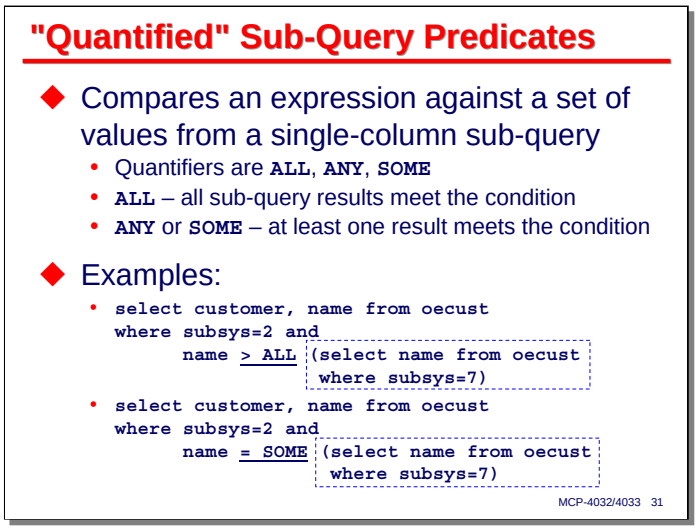


This slide shows a typical unquantified sub-query predicate, and illustrates two important points.

First, the text of the sub-query **`SELECT`** statement must be enclosed in parentheses and used as the RHS of a comparison predicate. This sub-query *must* be designed to return a single column and a single row (i.e., a single value) as its result, since the LHS and RHS of a comparison must be scalar values.

Second, the sub-query often is not independent – its predicates relate back to the main query in some fashion. Another way of thinking about this is that the rows of the main query can pass some of their values as parameters to be used in the predicates of a sub-query. Therefore, it is quite common for a sub-query to reference columns from the tables of the main query, and it is good practice to use table aliases as a prefix to the column names to help make this explicit.

Recall that, conceptually, SQL queries combine the rows and columns of the tables specified in the **`FROM`** clause into a virtual table, then filter the rows of that virtual table by the condition specified by the **`WHERE`** clause. Conceptually, therefore, the sub-query is evaluated once for each row of the virtual table produced by the main query. The conceptual behavior is a candidate for optimization, but it is possible for the number of database reads to grow geometrically if the sub-query is poorly specified or there is not a suitable index available to optimize the sub-query's database accesses.

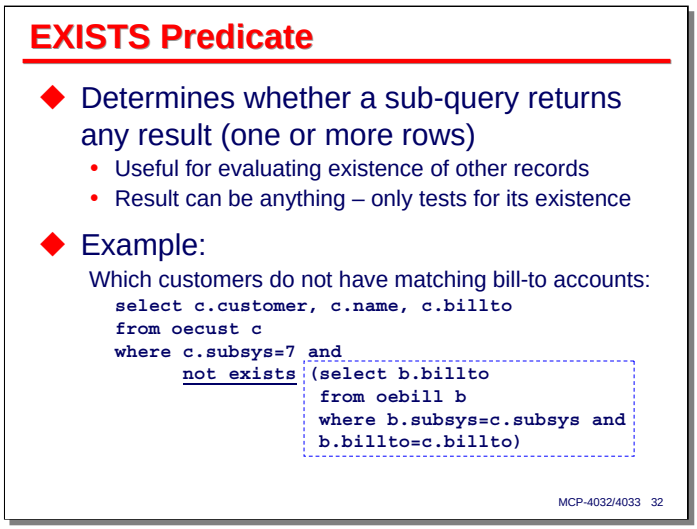


Quantified sub-query predicates look like an unquantified predicate with the addition of a keyword between the comparison operator and the sub-query.

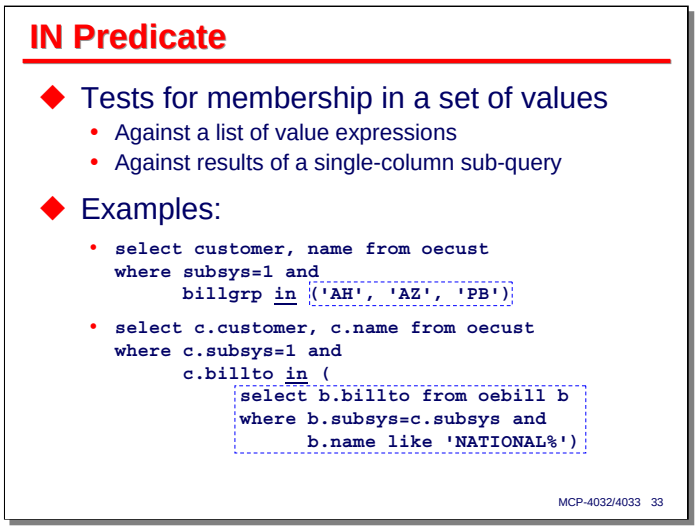
Where the sub-query for an unquantified comparison predicate is restricted to returning a single-column, single-row result, sub-queries for quantified predicates can return multiple rows, but still must return only one column. The multiple results for a set of values, and the LHS value is compared to this set using both the comparison operator (**=**, **<**, **<=**, **>**, **>=**, or **<>**) and the quantifier. There are two quantifiers, represented by three keywords:

- **ALL** – this quantifier causes the predicate to yield true if the comparison of the LHS value to *each and every one* of the values returned by the sub-query is also true. That is, each of the sub-query results must meet the comparison condition.
- **ANY** or **SOME** – this quantifier causes the predicate to yield true of the comparison of the LHS value to *at least one* of the values returned by the sub-query is also true.

When using quantified sub-query predicates, beware of the informal meaning of the word *any*. In informal speech, boasting that "I'm better than any of you guys" implies that you are better than *all* of them. In SQL terms, however, this means that you are only better than *some* (at least one) of them – not much of a boast. Because of this informal meaning of *any*, it is best to avoid the keyword **ANY** and use **SOME** instead.



The **EXISTS** predicate yields true if its sub-query returns a non-empty result, i.e., one or more rows. It does not matter what this result is, just whether it is empty or non-empty. Therefore, the sub-query used with an **EXISTS** predicate can be written to project any number of columns. It is usually more efficient, however, to return just a single, small column, such as a small numeric or character column.



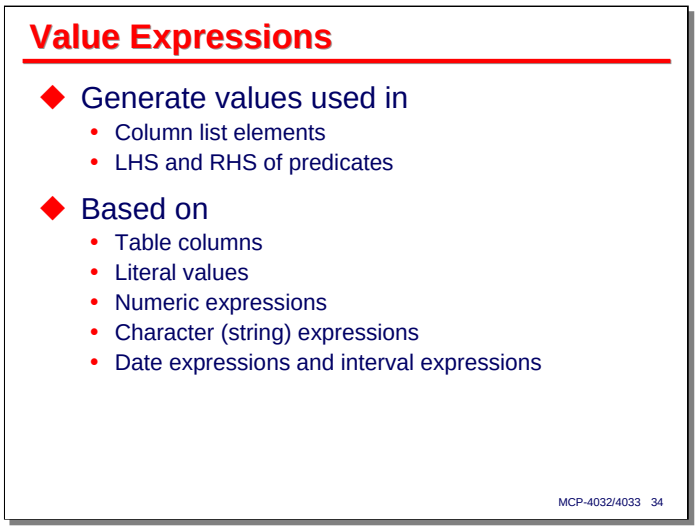
The **IN** predicate tests whether the value of an expression on the left-hand side is a member of a set of values specified on the right-hand side. The set of values to be tested against can be specified in one of two ways:

- Explicitly – the RHS can consist of a comma-delimited list of value expressions enclosed in parentheses.
- As a sub-query – the RHS can be a sub-query that returns a single-column result with zero or more rows.

In either case, the predicate yields true if the LHS value is equal to at least one of the members of the set specified by the RHS.

DMSQL appears to handle the form of this predicate having a list of values by expanding the list to a set of comparisons connected by **OR** operators. Depending on whether the other predicates in the **WHERE** clause can filter the virtual table rows to a small number on which the **OR**s are evaluated, this may or may not be very efficient.

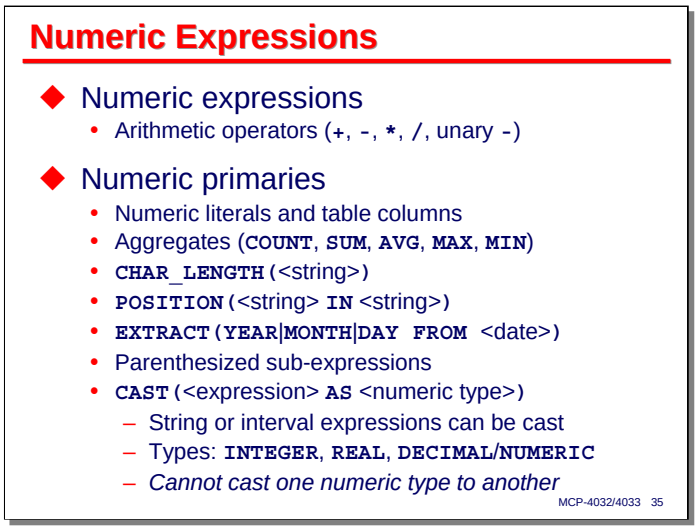
The efficiency of the sub-query form of the predicate appears to depend on the efficiency with which the sub-query can be evaluated and the efficiency with which the LHS can be compared to the results of the sub-query. Often the LHS value can be looked up directly in an index for a table in the RHS sub-query, in which case the evaluation of the predicate can be fairly efficient.



Thus far, we have discussed the use of value expressions in column lists and the left- and right-hand side of predicates, but have not said much about what they are. The next few slides describe what value expressions are and how they can be constructed.

Value expressions can be composed of table columns from the query, literal values, and the results of operators and functions that we will discuss next. In the DMSQL dialect of SQL, the values of these expressions fall into four types:

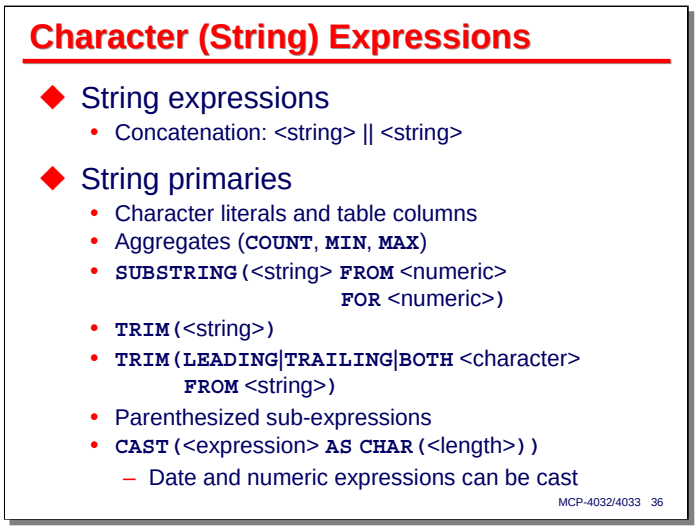
- Numeric expressions. Within this type are integers, fixed-point values (those with a decimal point), and floating point values (single and double precision).
- Character or string expressions.
- Date expressions. SQL considers dates to be a primitive type, distinct from numbers. The RDC can be used to specify that certain data elements within tables are dates and identify how those dates are represented in the underlying DMSII data sets. DMSQL does not currently support the concept of time values.
- Interval expressions. Intervals are an offset from a date, and can be represented in units of years, months, or days. You add or subtract an interval to a date to get another date. You subtract two dates to get an interval.



Numeric expressions consist either of a single numeric primary construct, or a combination of primaries and other expressions with the five arithmetic operators (addition, subtraction, multiplication, division, and unary negation).

Numeric primaries consist of the following types of constructs:

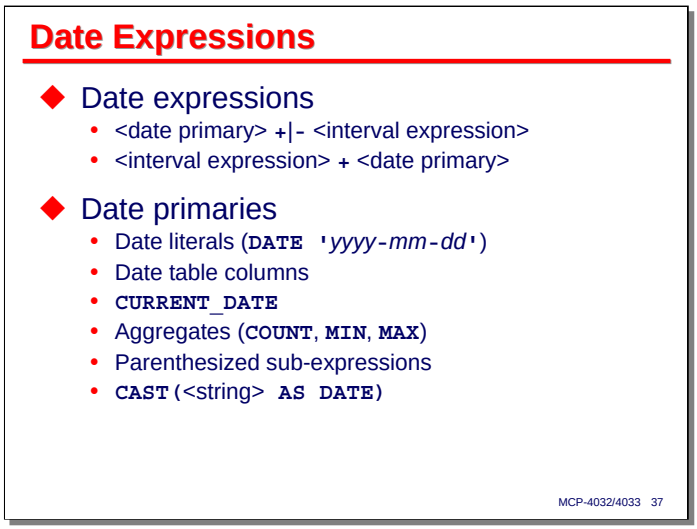
- Numeric literals, which may be signed or unsigned, with or without a decimal point.
- Table columns having a numeric type.
- Aggregates (**COUNT**, **SUM**, **AVG**, **MAX**, **MIN** – also called set functions) having a numeric expression as their argument. Aggregates will be discussed in more detail later in the section on grouping of results.
- The **CHAR_LENGTH** function, which returns the number of characters in a string expression.
- The **POSITION** function, which returns the ordinal position of a substring within another string expression.
- The **EXTRACT** function, which will return any of the year, month, or day components of a date expression.
- A numeric expression enclosed in parentheses.
- A **CAST** function that returns a numeric type. You can cast string expressions and interval expressions to numbers. DMSQL does not permit you to cast one numeric value to another (e.g., floating point to decimal). The numeric types into which an expression can be cast are:
  - **INTEGER** or **INT**
  - **REAL**
  - **DECIMAL**, **DEC**, or **NUMERIC**, optionally followed by precision and scale values in parentheses e.g., **DECIMAL(4)** or **NUMERIC(8,3)**.



String expressions consist of string primaries or string expressions concatenated using the double-bar (||) operator.

String primaries consist of the following types of constructs:

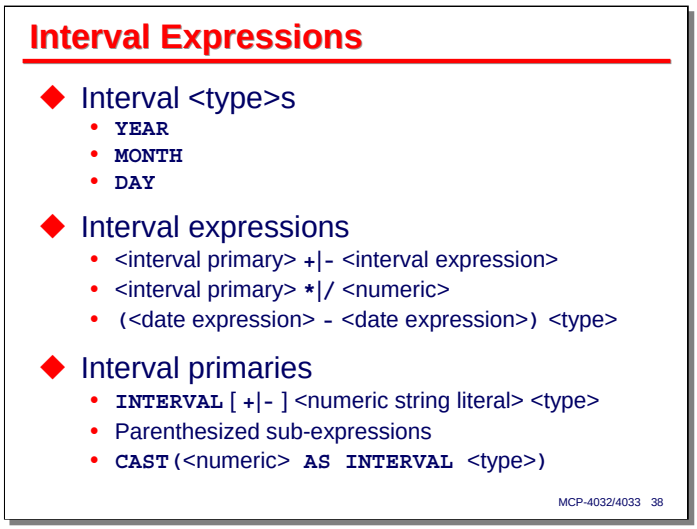
- Character literals. In SQL strings are delimited by the single-quote (') character. An embedded quote may be represented by two adjacent quote characters, e.g., **`'This isn''t a string'`**.
- Character-valued columns from tables. Note that **ALPHA** items from DMSII data sets are typically of fixed length, and thus in SQL will be padded with trailing spaces to their defined length.
- The aggregate functions **COUNT**, **MIN**, and **MAX**. **SUM** and **AVG** cannot be used with string expressions.
- The **SUBSTRING** function, which will extract a smaller string starting at a specified ordinal position for a specified number of characters from a string expression.
- The **TRIM** function, which will strip leading and/or trailing characters from a string expression. The simplest form of **TRIM** strips both leading and trailing spaces. Using additional syntax, you can control whether the trimming is done for leading or trailing characters only, and whether some character other than space is trimmed.
- A character expression enclosed in parentheses.
- A **CAST** function that returns a character type. A character type is specified as the keyword **CHAR** or **CHARACTER** followed by an integer length in parentheses. Only date and numeric expressions can be cast to character types.



SQL provides a rich mechanism for manipulating dates. Date expressions can consist of a date primary that is incremented or decremented by an interval expression or an interval expression that is incremented by a date primary. The result in either case is a new date value. Interval values and expressions are discussed on the next slide.

Date primaries consist of the following constructs:

- A date literal, which is composed of the keyword **DATE** followed by a literal string in the form **'***yyyy-mm-dd***'**, where *yyyy*, *mm*, and *dd* are numeric characters representing the year, month, and day, respectively. Leading zeroes do not need to be specified in the month and day portions of this string. The hyphen between date parts is required.
- Table columns of type date.
- The **CURRENT_DATE** function, which returns the system date.
- The aggregate functions **COUNT**, **MIN**, and **MAX**.
- A date expression enclosed in parentheses.
- A **CAST** function that returns a date type. DMSQL permits only character string expressions (having the same yyyy-mm-dd format as the date literal described above) to be cast as a date.



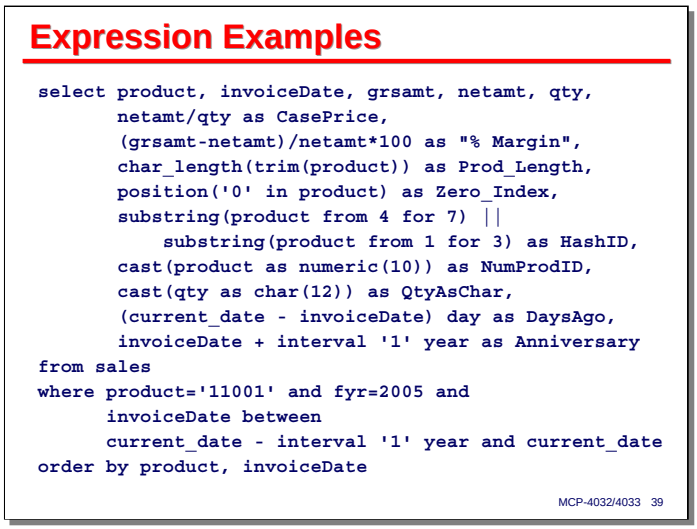
Interval values represent an offset from a date. Intervals can be expressed in units of years, months, or days. Their values can be positive, negative, or zero.

An interval expression consists of one of the following constructs:

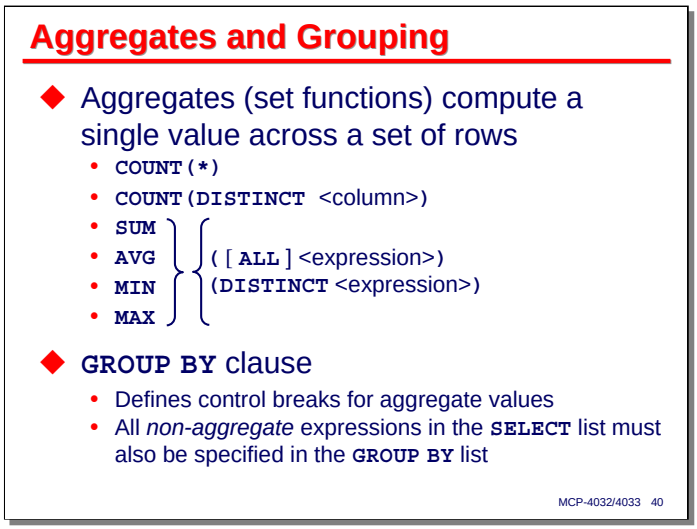
- An interval primary incremented or decremented by an interval expression, yielding a new interval value.
- An interval primary multiplied or divided by a numeric expression, yielding a new interval value.
- The difference between two date expressions enclosed in parentheses and followed by an interval qualifier keyword – **YEAR**, **MONTH**, or **DAY**. This yields a new interval value with the units specified by the interval qualifier (years, months, or days). The sign of the interval will be negative if the second date expression represents a later date than the first one.

Interval primaries consist of the following constructs:

- The **INTERVAL** keyword, optionally followed by a plus or minus sign, followed by a numeric character string in single quotes, followed by an interval qualifier keyword.
- An interval expression enclosed in parentheses.
- The **CAST** function specifying an interval qualifier. Only numeric expressions can be cast as intervals.



This slide shows examples of a variety of value expressions. Note that value expressions can be used both in column elements and in predicates.



We have seen aggregate functions mentioned in the discussion on value expressions. Aggregates compute a single value across a set of other values. These aggregate functions can be used in two ways:

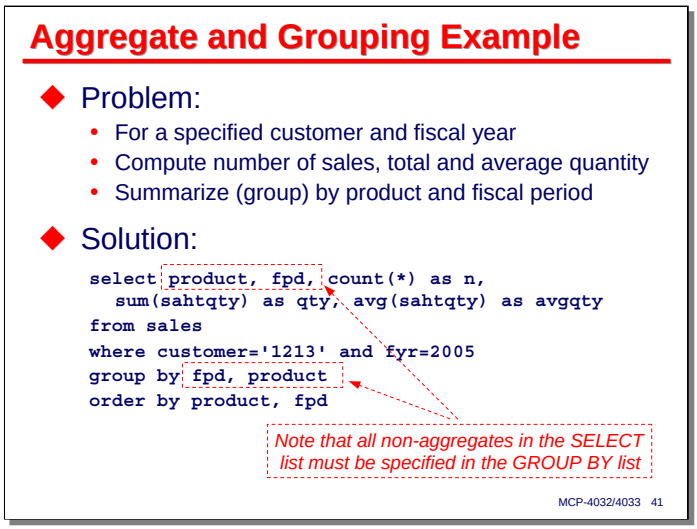
- If the query does not contain a **GROUP BY** clause, the **SELECT** list of the query may consist only of aggregate functions, or expressions containing only aggregate functions. The result set for the query will consist of a single row containing the aggregated values.
- If the query contains a **GROUP BY** clause, the **SELECT** list may contain non-aggregate expressions, but any table columns or value expressions (other than literals) that are not parameters to an aggregate function must be elements of the **GROUP BY** list.

In the latter case, the result set rows are grouped by the unique combinations of the elements in the **GROUP BY** list. The aggregate functions are computed over the rows for each group. In the result set, the aggregated values form one row, which replaces all of the original rows for its group. In terms of traditional report design, the aggregate functions generate subtotals (or averages, counts, minima, or maxima) and the **GROUP BY** list specifies the columns that determine where control breaks occur.

The aggregate functions work as follows:

- **COUNT(*)** simply counts the number of rows in the current group.
- **COUNT(DISTINCT** <column>**)** counts the number of distinct (unique) values within the group for that column.
- **SUM(**<expression>**)** computes the total value of the expression over all the rows in the group. The expression must be numeric-valued.
- **SUM(DISTINCT** <expression>**)** computes the total value of all unique values of the expression over the rows in the group. If the expression has the same value in two or more rows, it it used only once.
- **AVG(**<expression>**)** computes the average value of the expression over all the rows in the group. **AVG(DISTINCT** <expression>**)** computes the average of all the unique values of the expression over the rows of the group.
- **MIN(**<expression>**)** computes the minimum value of the expression over all the rows in the group. **DISTINCT** can be specified for this function, but it does not alter the result.
- **MAX(**<expression>**)** computes the maximum value of the expression over all the rows in the group. As with **MIN**, **DISTINCT** can be specified, but does not alter the result.

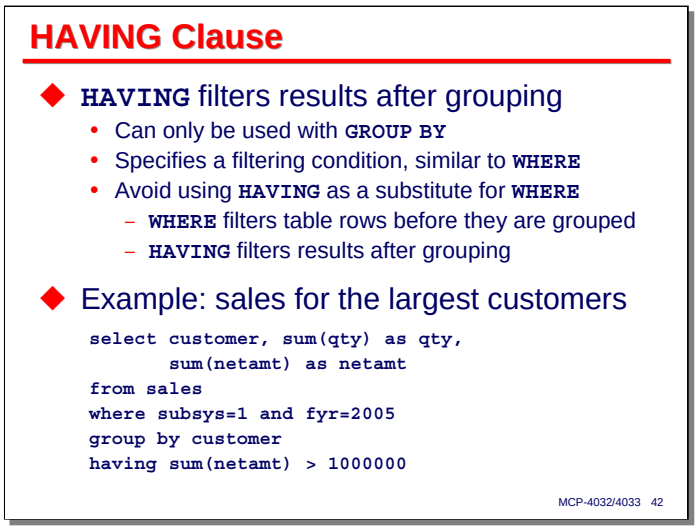
In each of these aggregate functions, the keyword **ALL** can be specified to distinguish the method of computation from **DISTINCT**, but **ALL** is the default, so is not normally written.



This slide shows a sample query illustrating the use of aggregate functions and the **`GROUP BY`** clause. It queries a table of sales data for a particular customer and fiscal year, summarizing sales by the product code and fiscal period (month). For each combination of product and period, the query will report the number of sales (actually, sales records), the total sales quantity, and the average quantity per sale.

Note that since the product and period columns are not aggregate functions or parameters to aggregate functions, they must be specified in the **`GROUP BY`** list. The query processor will generate a syntax error if the non-aggregate elements of the **`SELECT`** list do not match the **`GROUP BY`** list.

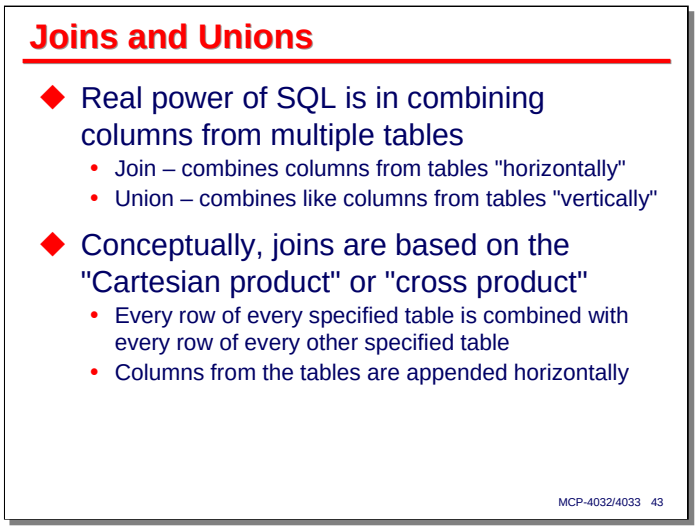
Also note that ordering is independent of grouping, although grouping usually requires a sort. DMSQL will attempt to use the same sort for both grouping and ordering, if possible.



The **HAVING** clause can be used only with the **GROUP BY** clause, because it operates on the grouped results. The condition specified with the **HAVING** clause filters the result rows after grouping, eliminating any that do not meet the condition.

HAVING works somewhat like the **WHERE** clause, but should not be used as a substitute for it. If it is possible to eliminate result rows using the **WHERE** clause, this is much more efficient, as such rows then do not have be grouped and aggregated before being eliminated.

The example on the slide shows a simple sales summarization query that eliminates any grouped results that have a total net amount value of one million or less. The final query result set will contain only rows for customers where the "netamt" column is greater than one million.



If SQL could only query and return results from a single table, it would be useful, but its real power comes from being able to combine data from multiple tables and manipulate the combined result.

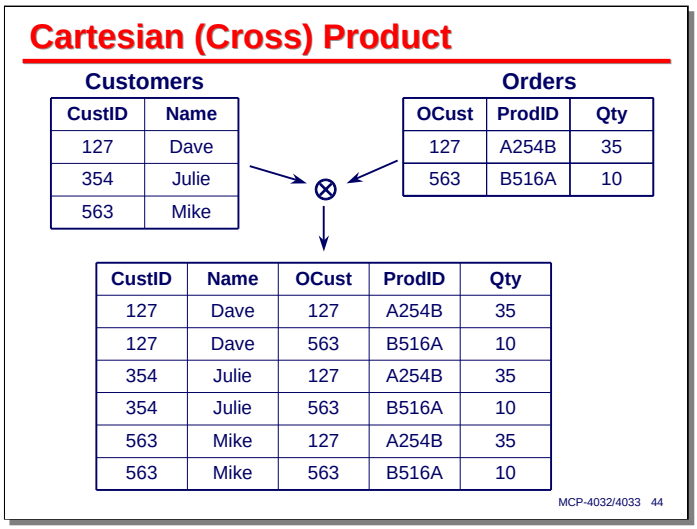
There are two primary ways SQL can combine data from multiple tables:

- The **join** – this combines columns from multiple tables. You can think of this as combining data along the horizontal axis of a table.
- The **union** – this combines like columns from multiple tables or intermediate result sets. You can think of this as combining data along the vertical axis of a table.

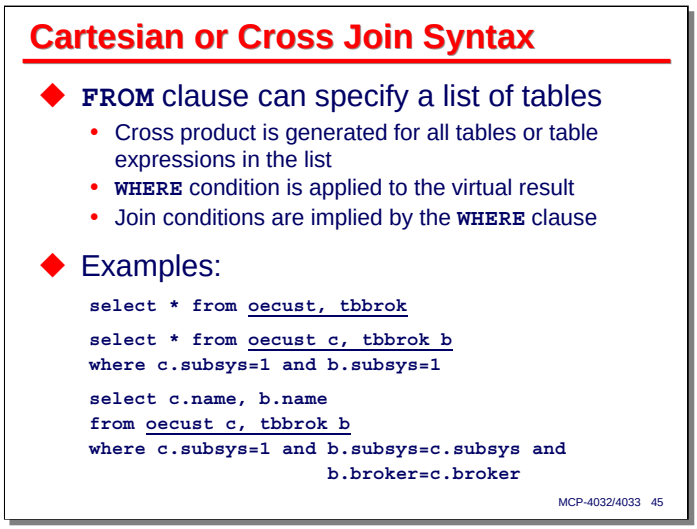
We will discuss joins first and unions a little later in the presentation.

Conceptually, joins are based on an operation from the relational algebra known as the Cartesian, or cross, product. The Cartesian product simply takes every row of every table and combines it with every row of every other table. The columns of these combined tables are appended horizontally. The result is a virtual table whose number columns is the sum of the number of columns in the joined tables, and whose number of rows is the product of the number of rows in the joined tables. Obviously, if the tables have very many rows, or there are very many tables, the size of the virtual table can get rather large.

Once again, it is important to remember that this Cartesian product is a *conceptual* model. SQL query processors try very hard not to have to actually generate this product during the evaluation of a query.



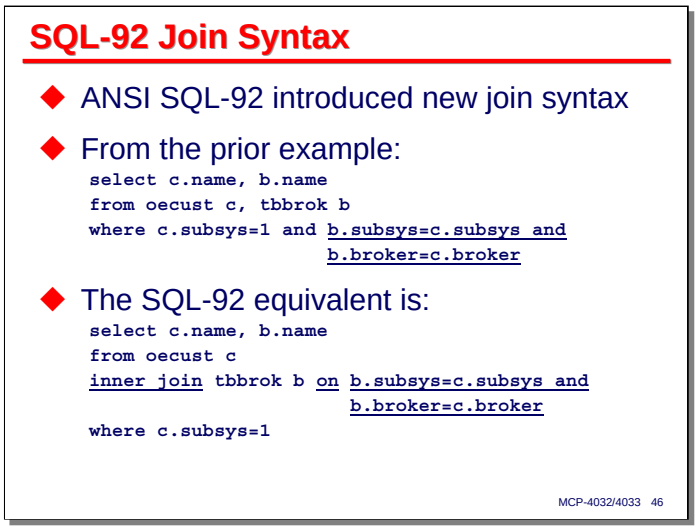
This slide shows a simple Cartesian product of two tables. Joining a Customers table having two columns and three rows with an Orders table having three columns and two rows, we end up with a table having five (2+3) columns and six (3×2) rows. Every row of Customers has been combined with every row of Orders.



The simplest form of join to specify is a comma-delimited list of tables (actually, table expressions) in the **FROM** clause. Each table in this list is combined with the others using the Cartesian product. For this reason, this form of join is called a Cartesian join or cross join.

Conceptually (there's that word again), this join produces a virtual table. The condition specified by the **WHERE** clause (if there is one) is used to filter this virtual table, eliminating rows that do not meet that condition.

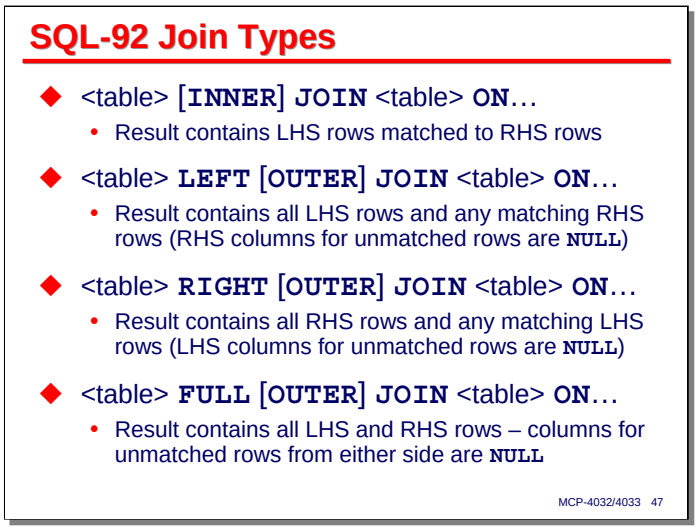
The examples on the slide illustrate a few simple Cartesian joins.



The ANSI SQL-92 standard introduced a new way to specify joins explicitly. It introduced a series of join operators and an **`ON`** phrase that explicitly specifies the condition on which tables are to be joined.

The two examples on the slide show equivalent queries, based on the example from the previous slide. Note how the **`ON`** phrase specifies the condition on which rows from the customer and broker tables are to be joined – only those records where the respective **`subsys`** and **`broker`** columns have matching values.

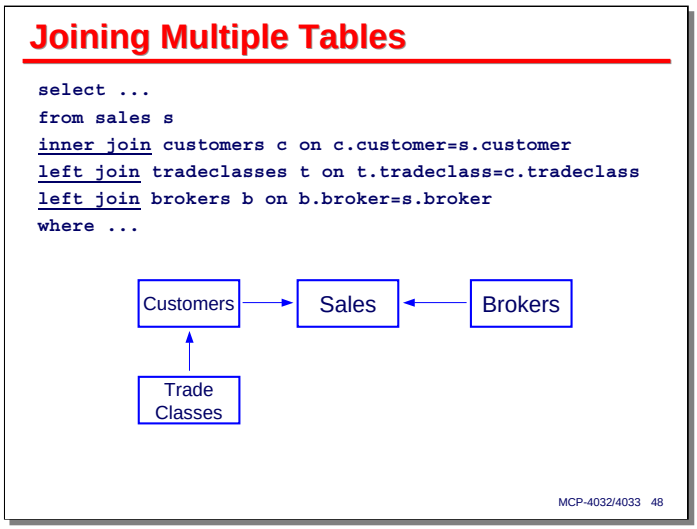
DMSQL supports both forms of join syntax, but the newer SQL-92 form is generally preferred, as it is easier to understand, especially when many tables are to be joined.



SQL -92 defined four types of join operators:

- The **INNER JOIN** is the default type if only the keyword **JOIN** is specified. The result of this operation will contain rows from the LHS and RHS tables only where a row from the LHS table can be joined with one or more rows from the RHS table. If there is no row from the RHS table that can be joined to a row from the LHS table, the LHS table row is eliminated from the result. If a row from the LHS table can be matched with a row from the RHS table, the columns from both rows are appended and appear in the result. If a row from the LHS table can be matched to multiple rows from the RHS table, the columns of the LHS row are appended to each of the matching rows from the RHS table, and those multiple joined rows appear in the result. An inner join is equivalent to a Cartesian join if the join conditions are the same.
- A **LEFT OUTER JOIN** is similar to an inner join in that rows from the LHS table will be joined with rows from the RHS table if they can be matched by the join condition. Where the left outer join differs is what happens if there is no match in the RHS table to a row in the LHS table. In that case, the columns from the LHS table appear in the result unconditionally, with sufficient null values appended to represent the columns of the missing RHS table row.
- A **RIGHT OUTER JOIN** is simply a left outer join with the order of the tables reversed. In fact, DMSQL tries to convert right outer joins into left outer joins. Since you can always write a right outer join as a left outer join, many SQL programmers avoid right outer joins altogether.
- A **FULL OUTER JOIN** is essentially a combination of a left and right outer join. All rows from both tables appear in the result of the join. Where rows from one table can be matched to rows of another table, their columns are joined and appear in the result, where a row from either the LHS or RHS tables cannot be matched with the other side, that row appears in the result with columns of nulls representing the missing row from the other table.

The inner and left outer joins are the most frequently used types. Note that the keywords **INNER** and **OUTER** are optional in all forms.



SQL permits you to specify multiple joins to create a table expression in the **`FROM`** clause. Conceptually, the result of the first join is a virtual table that effectively becomes the LHS of the second join. That second join conceptually generates another virtual table that becomes the LHS of the third join, and so forth.

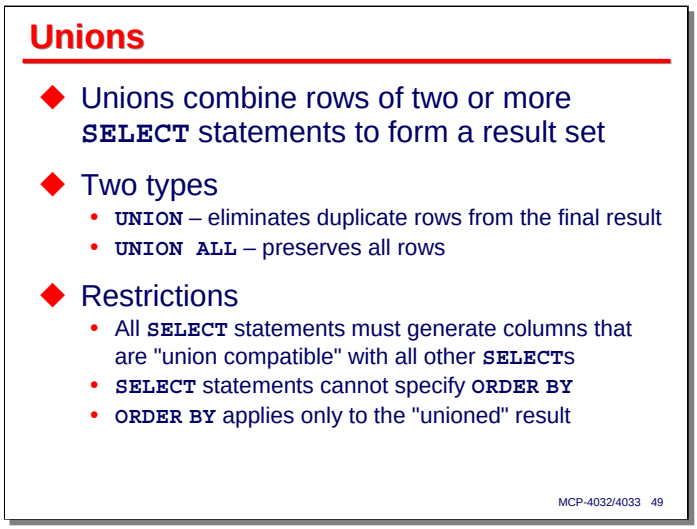
This pipelining of joins means that the **`ON`** condition of a join can reference columns from any of the tables used in earlier joins (i.e., those to the left).

The slide shows a snippet from a SQL query that does multiple joins. We have some sales data that we would like to annotate with additional customer data (such as customer name and class of trade) and broker data. In addition, we would like to look up the class-of-trade code in the customer record and translate that to the description that is stored in the **`tradeclasses`** table.

The query is (conceptually) driven from the **`sales`** table. For each sales row, we attempt to find a matching **`customers`** row based on the value of the **`customer`** column in both tables. Because this is an inner join, if there is no matching customer row, that sales row will be excluded from the result.

Similarly, we attempt to find a matching row in the **`tradeclasses`** table based on the value of the **`tradeclass`** column in the **`customers`** and **`tradeclasses`** rows. Since this is a left [outer] join, however, failure to find a matching **`tradeclasses`** row will not cause the LHS row for the join (the combined sales and customer columns) to be excluded from the result. Instead, nulls will be appended to the LHS columns in place of data from the missing RHS **`tradeclasses`** row.

Finally, we attempt to find a matching row in the **`brokers`** table based on the value of the **`broker`** column in the **`sales`** table. Again, since this is a left join, a mismatch will result in nulls for the **`brokers`** table columns instead of eliminating the LHS data from the result.

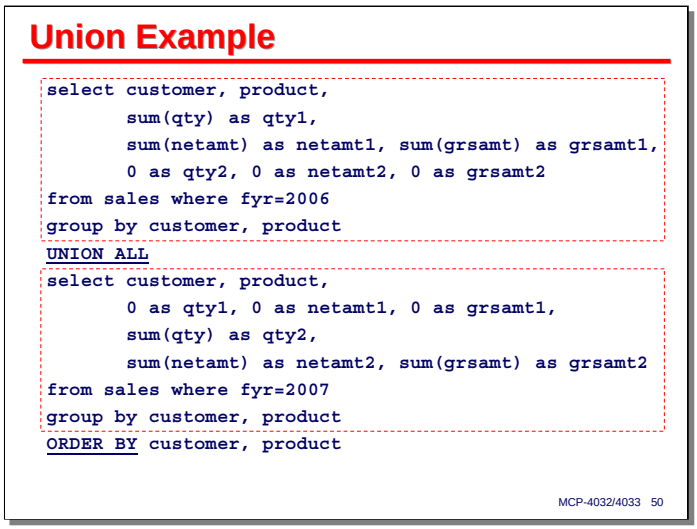


The SQL **UNION** operator combines the result set rows from two or more **SELECT** statements to form a new result set. The **SELECT** queries can be from the same or different sets of tables. The only restrictions are that the result sets have the same number of columns, and that the columns be "union compatible" (i.e., that column for column, the column data types be compatible).

There are two types of **UNION** operator.

- By default, **UNION** by itself eliminates duplicate rows from the final result set (as if you could have specified **UNION DISTINCT**, which you cannot).
- **UNION ALL** specifies that all rows from both result sets will be returned in the final result set.

Note that if you wish to order the results, the **ORDER BY** clause goes after the final **SELECT** statement and applies to all of the **UNION**-ed results. The individual **SELECT** statements forming the LHS and RHS of **UNION** operators cannot specify **ORDER BY** clauses.

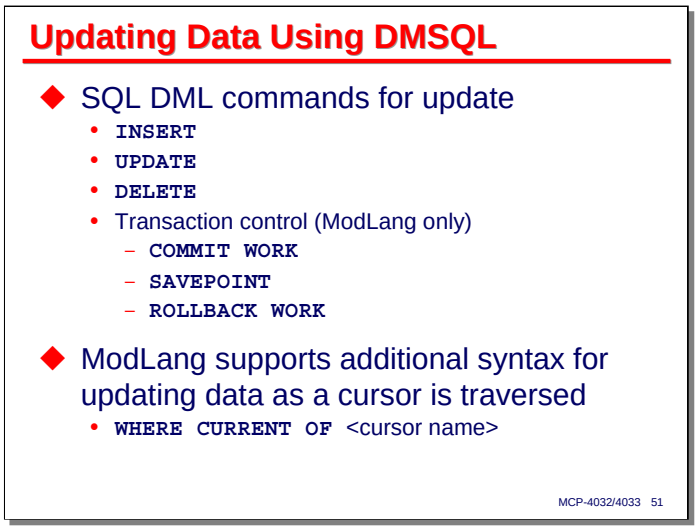


This slide shows an example of a **UNION ALL** for two result sets. In this case, the result sets are produced from the same columns of the same table, but have different **WHERE** clauses.

Also, the aggregate function columns are are aligned in separate columns for the two **SELECT** statements, with zeroes to fill in for the aggregate values that are supplied by the other **SELECT**. This is one way to identify which side of the **UNION** the result came from. In this particular case, it would allow the application program invoking this query to simply sum the **qty1**, **netamt1**, **grsamt1**, **qty2**, **netamt2**, and **grsamt2** columns in the result set without concern for which year the results came from. The totals for the "**-1**" columns would have the results from fiscal 2006 and the totals for the "**-2**" columns would have the results from fiscal 2007.

Note that the **ORDER BY** clause comes at the very end, and affects the result of the **UNION**, not the second **SELECT** statement.

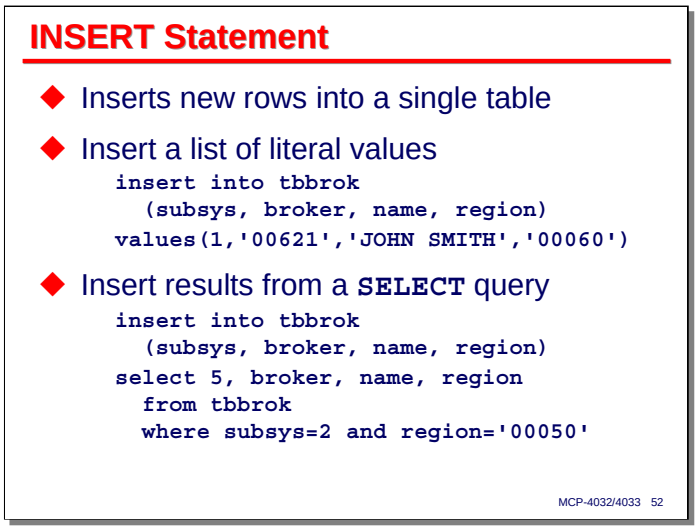
As with joins, you can chain **UNION** operators to combine results from more than two **SELECT** statements. The result of the first **UNION** becomes the LHS of the second **UNION**; the result of the second **UNION** becomes the LHS of the third **UNION**, and so forth.



I indicated early on that the focus of this presentation would be on query instead of update, but I want to at least mention the three SQL Data Manipulation Language (DML) statements that implement updates – **`INSERT`**, **`UPDATE`**, and **`DELETE`**.

In addition to these three statements, the ModLang has statements for transaction control. The CLI also supports transaction control, but commits, rollbacks, etc. are implemented through API calls rather than DML statements.

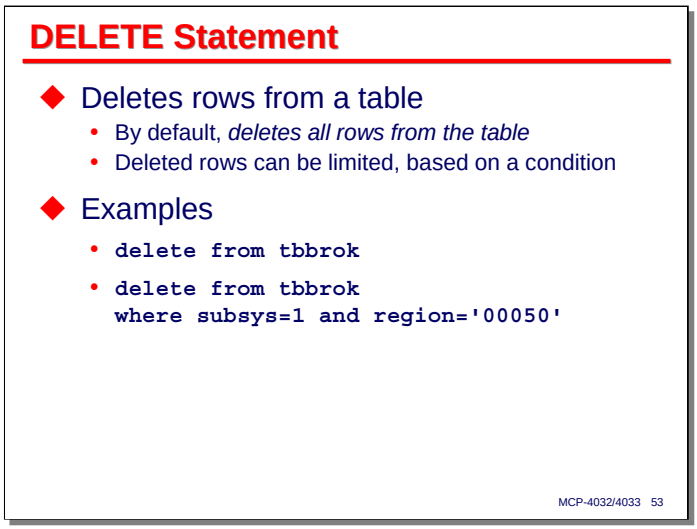
In addition to the forms of statements shown on the next few slides, the ModLang supports the **`WHERE CURRENT OF`** syntax to allow updating the current row of a cursor as an application program is traversing through a cursor's result set.



The **INSERT** statement creates a new row and inserts it into the table. This statement has two forms:

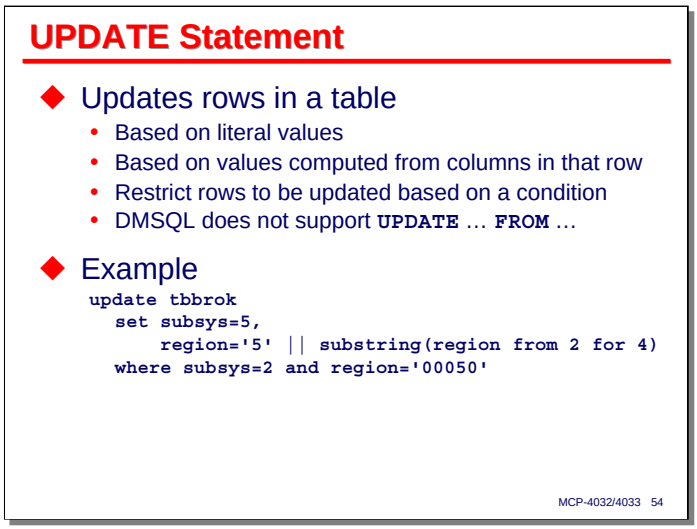
- The simpler form allows you to specify a list of columns for the table and a corresponding list of literal values for those columns. Any columns not specified for the new row will receive their default value, as specified by the table's schema. Note that the correspondence between column names and values in this form of **INSERT** is positional. Also note that this form will insert only one row into the table.
- The second form of **INSERT** uses a **SELECT** statement to supply the column values to be inserted into a new row. This form of **INSERT** can create multiple new rows in the table, one for each row in the **SELECT** statement's result set. The **SELECT** statement can generate the result set from the same or any combination of different tables – the only requirement is that the number of columns and corresponding types of columns be compatible with the columns specified in the parenthesized list following the table name.

INSERT statements are subject to the constraints imposed by DMSII validity checking – required values, DASDL **WHERE** clauses, **NO DUPLICATES** specifications on index sets and subsets, etc.



The **DELETE** statement deletes rows from a table. By default (if there is no **WHERE** clause), it will delete all rows from the table. SQL is one of the few languages where you have to say more in order to do less.

Typically, the rows to be deleted are restricted by a **WHERE** clause. The predicates in this clause can operate on any combination of literal values and value expressions based on columns from the row currently being considered for deletion.



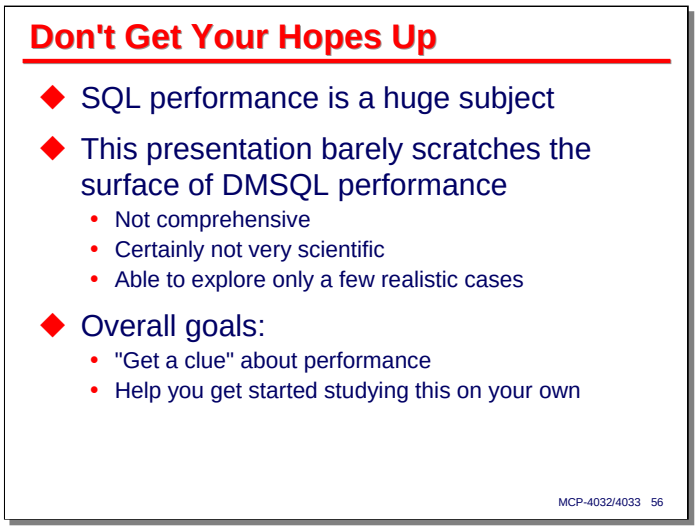
The **UPDATE** statement modifies the values of columns in a table. As with **DELETE**, all rows of the table are updated by default, but typically the rows to be updated are constrained by a **WHERE** clause.

The **SET** keyword introduces a list of column-name=value pairs that assign new values to the indicated columns. The RHS of each pair can be composed from literals and value expressions based on columns from the row currently being updated.

DMSQL does not currently support the **UPDATE** … **FROM** syntax that some other query processors do.

With that, we come to the end of the discussion on the dialect of SQL that DMSQL supports.

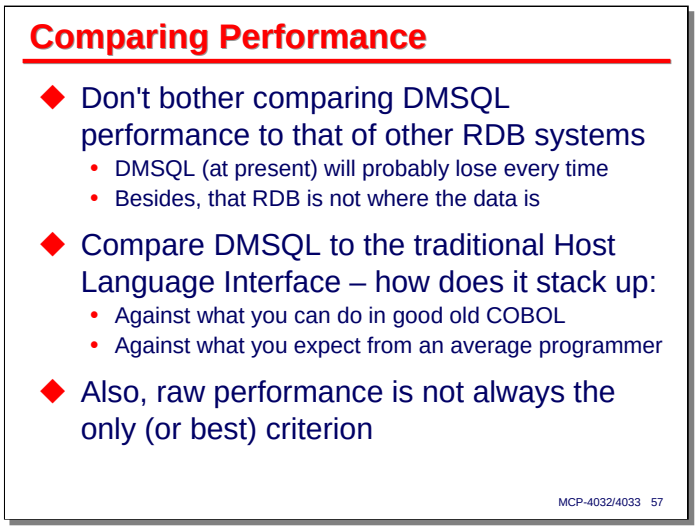
Next, I want to try to discuss and summarize some things I have learned about DMSQL performance.



The first thing I want to try to do is contain your expectations. SQL performance is a huge subject. One of the things I have come to appreciate in preparing this presentation is just how huge a subject it is. A comprehensive treatment of just DMSQL performance could easily be a life's work.

As a result, this presentation barely scratches the surface of DMSQL performance. It is not comprehensive, and is certainly not very scientific. The variety of ways that you can form just SQL query statements, coupled with the variety of ways that database schemas, physical attributes, and system hardware configurations can be arranged, makes it very difficult to draw useful conclusions about performance in general. I have been able to explore only a few realistic case studies, and the information presented here is based primarily on those.

Overall, the goal of this section of the presentation is to help you get a clue about DMSQL performance, and hopefully, help you to get started studying this subject on your own for your particular environment.

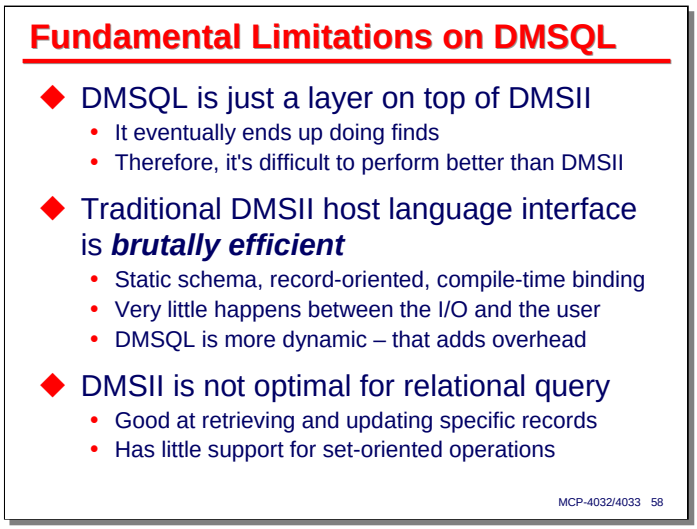


As another caution, I suggest that you not try to compare DMSQL performance to that of other relational database systems. The major database vendors have invested decades of work and billions of dollars in designing and improving their SQL-based products. DMSQL at this point cannot hope to compete with that and will probably lose every time.

Another reason not to try comparing DMSQL performance to other RDBs is that those RDBs is not where your data is. Your data is in DMSII.

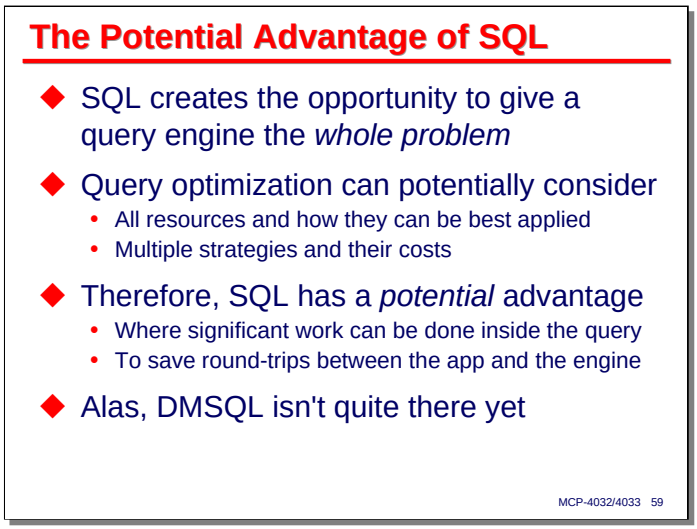
A better point of comparison is how well DMSQL stacks up against the performance of the traditional host language interface – what you can currently do in COBOL, and especially what you can expect in terms of query design and execution from the average COBOL programmer.

Another thing to keep in mind is that raw performance is not always the only criterion on which to judge a software facility. It is often not the best criterion, either.



DMSQL has some fundamental limitations that constrain how well it can perform with respect to the traditional host language interface.

- First, DMSQL is just a layer that sits on top of DMSII. It has a somewhat more intimate interface with DMSII than most application programs, but in the end it basically does what application programs do – DMSII finds. Therefore, it's going to be difficult for DMSQL to be dependent on DMSII but perform better than DMSII.
- Second, the traditional DMSII host language interface is brutally efficient. DMSII has a static schema (you have to recompile the DASDL and regenerate a bunch of code in order to change it), all retrievals are record oriented, and a lot of the schema information is bound to the application at compile time. A DMSII find is a fairly low-level database operation – there is not a lot that happens between DMSII doing an I/O and the resulting record arriving in the application program's record area. DMSQL is much more dynamic and flexible, and that in itself is going to add more overhead. In terms of performance, it's hard to compete with something that doesn't do much to begin with.
- Third, the design and physical structure organization of DMSII tables and indexes is not the best for relational query evaluation. DMSII is really good at retrieving and updating specific records – that is what it does best. It has little support, however, for the set-oriented operations that are needed by a SQL query processor.

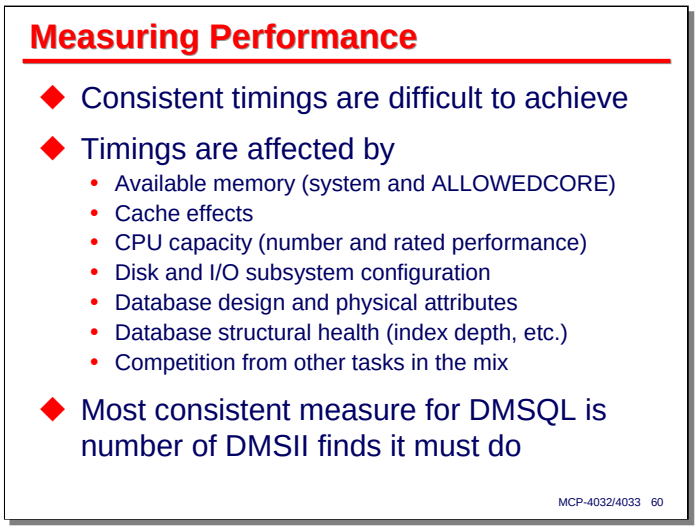


Even with these limitations, however, SQL has a potential advantage over the record-at-a-time approach that traditional DMSII applications use.

This potential advantage comes from the opportunity to give the query engine a description of the whole program – not just requests for individual record retrievals. Given a larger and more complete problem statement, the query optimizer can potentially consider all of the resources that are involved the the retrieval and how those resources can be best applied. The optimizer can also consider multiple strategies and their relative costs, picking the one with the lowest cost.

Therefore, SQL-based data retrieval has a potential advantage over record-at-a-time retrieval where significant work can be done inside the query. If it can do filtering and joining and summarization inside the query engine, that can save round-trips between the application and the engine to pass back low-level results where the application does the filtering, joining, and summarization.

Alas, DMSQL is not yet at the point where it can fulfill this potential. That doesn't mean that it can't, just that at present is doesn't. I hold out hope that it can be improved to the point where that potential is realized.



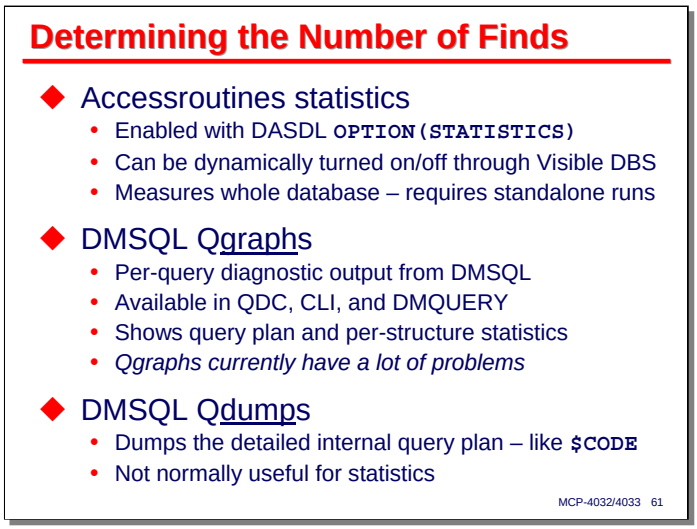
If we are going to talk about performance and compare performance, we need to have a way of measuring performance. The traditional way to measure performance is to determine how much time various activities take and compare those times.

When measuring the performance of transactions and database queries, however, this approach is problematic, because consistent timings are difficult to achieve.

Transaction and database query timings can be affected by a host of factors, some of which are hard to control, and some of which you don't really want to control because they are there to opportunistically improve performance whenever possible. Such factors include:

- Available memory – both system memory and the DMSII **`ALLOWEDCORE`** cache.
- Cache effects. If you run the same test twice is succession, the second run often performs better, because some of the records from the first run are still in the system and database caches. This can be controlled somewhat within the MCP environment, but its difficult in the vmMCP environment, where the underlying Windows operating system is doing caching of its own.
- CPU capacity, both number of processors and their rated performance.
- Configuration of the I/O subsystem and individual disk units. SAN devices add their own caching and opportunistic performance optimizations, which also make consistency of measurement difficult.
- Design of the database and the physical attributes of the database structures. The same data can be represented multiple ways by database designs, and can be physically stored in more than one manner. Thus the performance of a retrieval process will probably differ between the different designs. Conclusions on performance drawn for one particular design may not (and often do not) carry over to other designs.
- Even within one design, the "structural health" of the data can significantly affect the performance of retrieval. The presence of large numbers of deleted records, index B-tree depths and balancing, physical scattering of file areas on disk drives, etc., can all affect performance.
- Finally, performance of individual queries can be affected by competition from other tasks in the mix. This can generally be controlled by running tests standalone, but that is a luxury that many sites do not have – especially big sites with big databases and big workloads.

After thinking about these effects and how they affect DMSQL performance, I finally decided that the most consistent measure of DMSQL performance was the number of DMSII finds a query must do. Given a particular database design, this measure should remain constant, regardless of the size or type of system, the amount of memory available, the I/O configuration, database structural health, or competition from other tasks in the mix. The time to do those finds may vary, but the number of finds should remain constant.



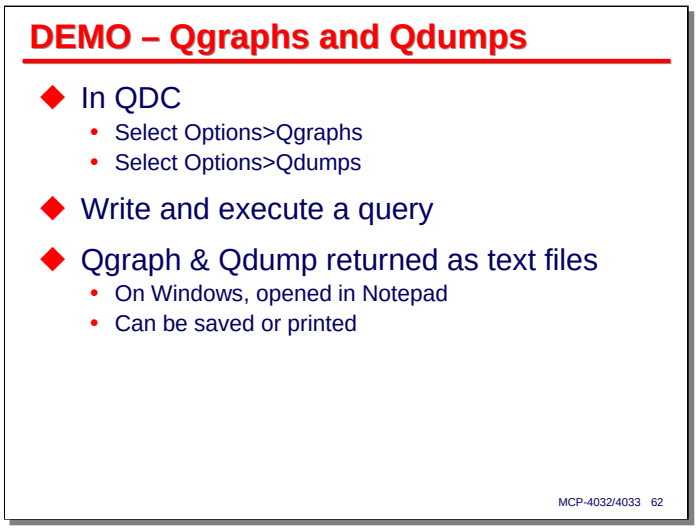
How does one measure the number of finds a DMSQL query does? At present, there are two ways:

- You can use the statistics capability in the DMSII Accessroutines. This is enabled by the DASDL **`OPTION(STATISTICS)`** syntax. It must be initially enabled through a DASDL compile, but once that is done, actual accumulation and reporting of statistics can be turned on and off through Visible DBS commands. Among other things, these statistics report the number of finds done on a per-table and per-index basis. The big disadvantage to Accessroutines statistics is that they cover the entire database. In order to make them meaningful for one user or program, you have to run that user's work or that program standalone.
- DMSQL has a diagnostic report called the "Qgraph." This can be requested on a query-by-query basis. It is available when using the CLI API, QDC (which uses CLI), and the batch/remote DMQUERY utility. A Qgraph report shows:
  - The text of the query.
  - An indication of DMSQL's query strategy or query plan.
  - Overall timings.
  - Number of finds on a per-table and per-index basis.

  The idea of the Qgraph is great. The problem is that the current implementation does not work very well. For simple queries, it's fine (although the timings always seem to be wrong). For more complex queries, however, the query plan is often unintelligible, and the find statistics are unreliable. There also appears to be some interaction between parameterized queries and Qgraphs that either cause a corrupted Qgraph to be generated, or inhibit its generation altogether. These problems have been reported to Unisys, so hopefully it is just a matter of time until they are corrected. Once that happens, Qgraphs will be very useful in diagnosing DMSQL performance.

DMSQL has an additional diagnostic report, the "Qdump," which is available through the same interfaces as the Qgraph. Its name is easily confused with that of the Qgraph, but the two are very different. Qdumps give a detailed view of DMSQL's query plan. They are a little like the **`$CODE`** output from a compiler. Qdumps are not normally useful for statistics, and since they require quite a bit of knowledge of DMSQL internals to interpret, are not normally a useful user-level tool.

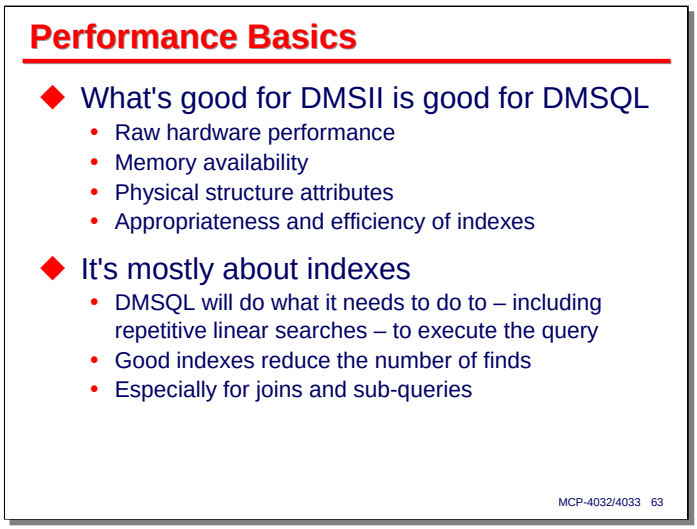
Note that the ModLang is conspicuously missing from the list of interfaces that support Qgraphs and Qdumps. There are currently no diagnostic tools available for use with the ModLang. Generally you develop and test queries using something like QDC or **`DMQUERY`** before encapsulating them in the Module Language, and your performance tuning for ModLang should be done up front as well.



Here is a quick demo using QDC of Qgraph and Qdump diagnostic reports. Note that these are enabled from the Options menu and (at least in Windows) are returned to your workstation as text files opened in Notepad.

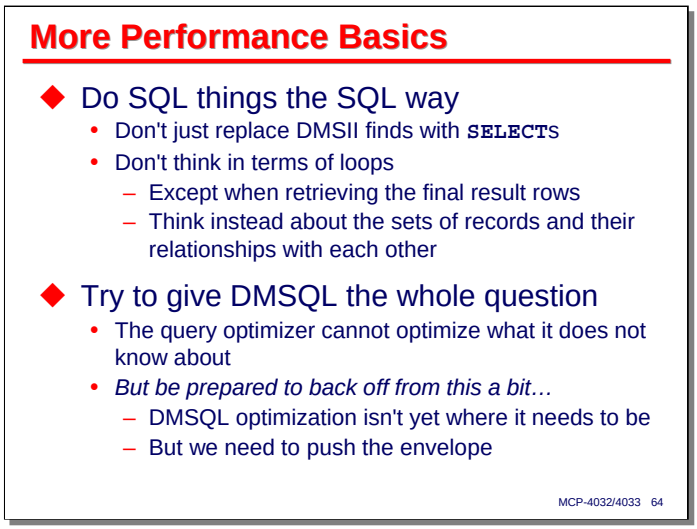
In the CLI, Qgraphs and Qdumps are enabled and retrieved through API calls. In **DMQUERY**, you would use the **DIAGNOSE** and **QD** commands.

[Perform demo for audience]



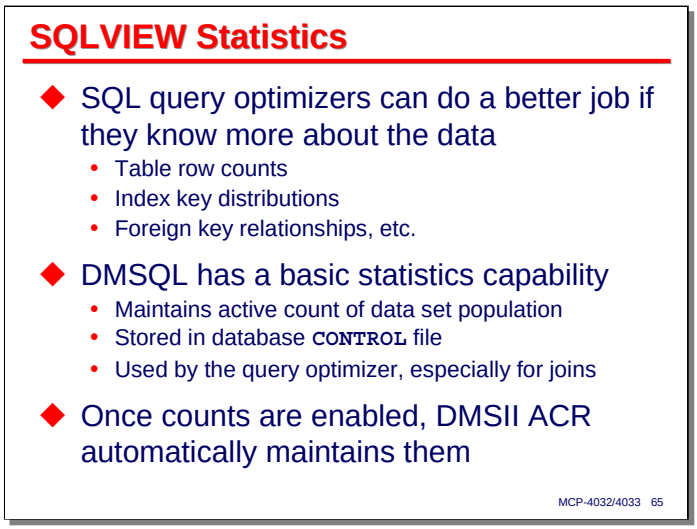
Despite the wide variety of database structures and query constructions that exist, there are some general things one can say about DMSQL performance:

- First, and most important, what is good for DMSII performance is also good for DMSQL performance. By the same token, a poorly-performing DMSII database cannot be made to perform better through DMSQL. The same things that will benefit DMSII data retrieval will benefit DMSQL – raw hardware performance, memory availability, physical structure attributes (such as blocksize and readahead), and the availability and relative health of suitable indexes.
- Second, DMSQL performance is mostly about indexes – having index structures with keys that map well to the predicates and join conditions of a query. One of the nice things about SQL is that you can get an answer to a query regardless of the availability of suitable indexes, or whether there are any indexes at all. The downside is that it may take a very long time to get that answer, because DMSQL will do whatever is necessary, including repetitive linear searches, to execute the query. Good indexes (i.e., those with key structures that map well to the query predicates) reduce the number of record accesses that DMSII must do and the number of records that DMSQL must filter, group, aggregate, etc., in order to produce the final result. The availability of good indexes is especially important in evaluating joins and sub-queries.



Continuing with performance basics…

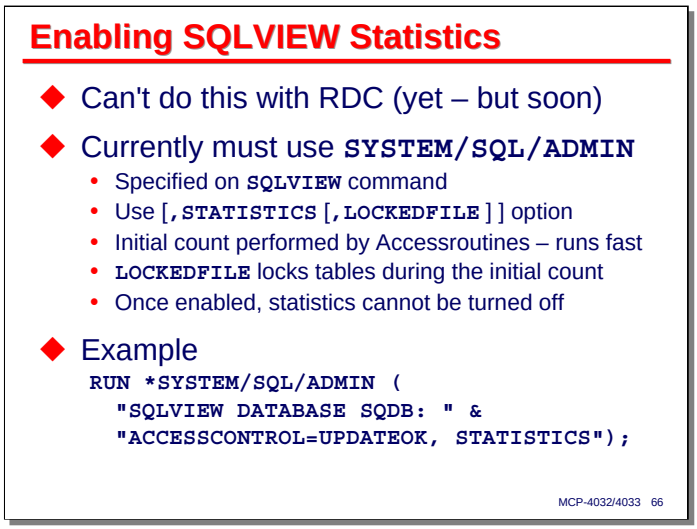
- Next, it is a good idea to at least try to use SQL in the way it was intended.
  - One of the worst things you can do is simply replace DMSII finds with equivalent **`SELECT`** statements – that in most cases is guaranteed to get you very poor performance.
  - With DMSII we are used to thinking in record-level terms, and that in turn usually leads to designing queries in terms of loops. When designing SQL queries, however, you need to think about *sets* of records and their relationships. What you need to specify is not how to get the records you want, but a definition of the set of records based on their attributes (columns) and relationships with other records. The transition from record-at-a-time and loop-based design to set-oriented design is one of the most difficult things to learn when starting to use SQL, and it takes some practice to achieve.
- Finally, try to give DMSQL the whole question – the whole retrieval problem. The query optimizer can't optimize what it doesn't know about, so if you break up the retrieval problem into multiple pieces and feed those one at a time to the query processor, you are potentially robbing the potential for optimization to occur across those pieces. DMSQL is not yet as good as it needs to be optimizing large queries, however, so be prepared to back of from this a bit if necessary. Try being a little aggressive in this direction first, though. DMSQL will not get better unless we press its envelope and report back to Unisys what does not work well.



One problem with query optimizers is that in order to produce an efficient strategy, they often need to know something about the data – particularly the relative populations of tables and the distribution of key values. DMSII programmers often have an intuitive feel (or outright knowledge) of this for their databases, and can use that to advantage when designing how they are going to navigate through a group of tables and indexes. Most major SQL query engines now keep detailed statistics on table populations and index key values, and use those statistics in evaluating the cost of alternate strategies.

DMSQL now has a basic statistics capability, available since MCP 11.1. This feature maintains an active count of the current population in each data set. These counts are stored in the database **CONTROL** file.

The population counts are actually maintained by the DMSII Accessroutines, so they remain accurate regardless whether updates are done through DMSQL or the host language interface.



To enable these population statistics, you must generate the relational mapping using the **SYSTEM/SQL/ADMIN** utility that runs within the MCP environment on the host. You cannot enable statistics through the RDC yet, but as this is being written, an RDC implementation is being readied for release and should be available sometime in the Fall of 2008.

Statistics are enabled as an option on the **SQL/ADMIN** program's **SQLVIEW** command. At the end of the command, you can append

```
, STATISTICS
```

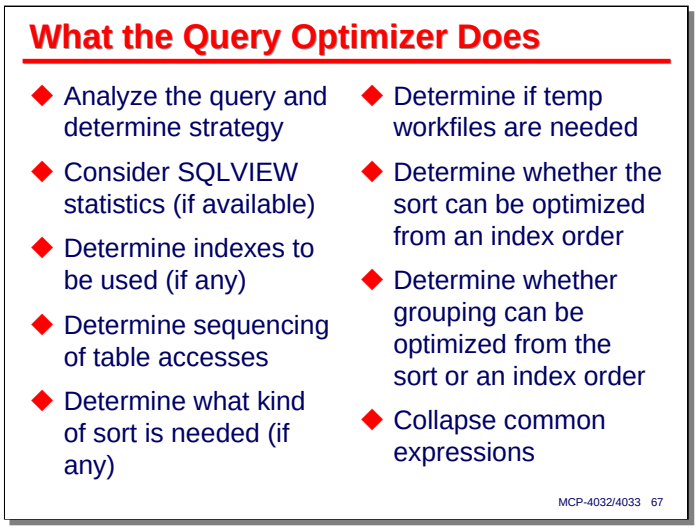
which will cause initial counts to be gathered and stored in the database **CONTROL** file at the end of the relational mapping process.

The initial count is performed by the Accessroutines, and it runs quite fast. Running on an LX170 laptop SDK, this process took about three minutes for a 14GB, freshly-loaded database.

The initial count can be generated while the database is in use and being updated. Since other tasks could be updating data sets at the same time their population is being determined, the initial count could be incorrect. You an specify an additional option to cause the Accessroutines to lock individual tables while their initial count is being determined:

```
, STATISTICS, LOCKEDFILE
```

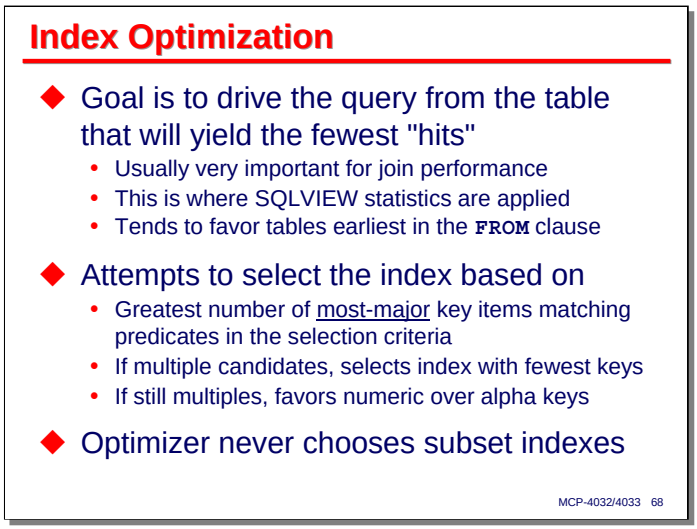
The example on the slide shows an SQLVIEW command that includes the **STATISTICS** option.



The job of the query optimizer is to analyze the non-procedural, set-oriented description of the query as it is expressed in SQL and translate that to a strategy that can be implemented using ordinary database operations. A goal of this translation process is to produce a strategy that executes the query with the least cost – usually that means in the least amount of elapsed time.

In analyzing the query and determining the strategy it will use, the DMSQL query optimizer takes the following things into account. This list is not intended to be a complete statement of all that the query optimizer does, just a representative sample.

- If SQLVIEW statistics are available in the database, these will be used to weight the costs of various strategies.
- Determine which index will be used to access each of the tables used in the query.
- Determine the sequence in which tables will be accessed. This is especially important when joins are involved.
- Determine what kind of sort, if any is required. Sorts may be required due to an **`ORDER BY`** clause, a **`GROUP BY`** clause, the **`DISTINCT`** option, or some combination of all three. Note that even if there is an **`ORDER BY`** clause, a sort may not be required if the ordering can be taken from an index.
- Determine if temporary workfiles will be needed. Workfiles are generally needed when a sort must be done, and for **`UNION`** queries, among other cases.
- Determine if an **`ORDER BY`** clause can be satisfied by the ordering of an index.
- Determine if the **`GROUP BY`** clause can be supported using the same sort required for an **`ORDER BY`**, or by the ordering of an index.
- Collapse common expressions. It is not unusual to have the same expressions repeated in the **`SELECT`** column list, **`GROUP BY`** clause, and/or **`HAVING`** clause.

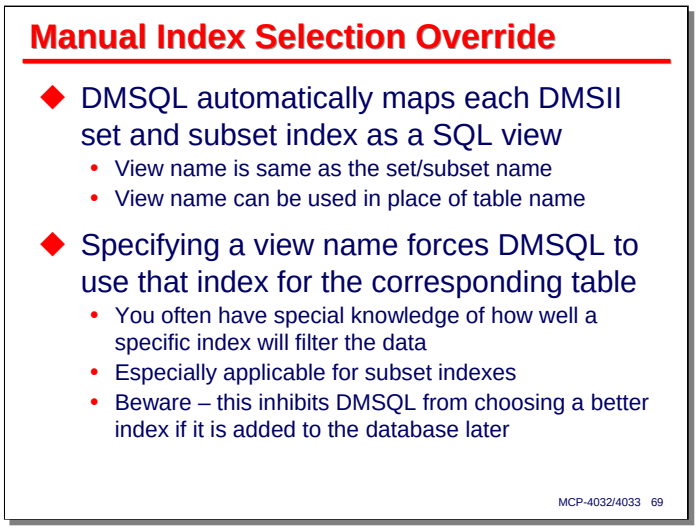


Determining an appropriate index to be used with a table and the order in which indexes should be accessed is one of the most important functions of the query optimizer, and the one that often yields the greatest performance advantage.

In general, the optimizer tries to drive the query from the table that it thinks will have the fewest hits, or from which it will end up selecting the fewest records. Starting from the smallest number of records is usually important when joins are involved, as the number of records that must be accessed often grows geometrically with the number of joins. It is in attempting to determine which table will yield the smallest number of hits that SQLVIEW statistics are applied to the optimization process. Where the optimizer cannot discern a cost advantage, it will tend to favor tables specified earlier in the **FROM** clause over those specified later.

Something to keep in mind is that the DMSQL optimizer will never choose an index for a DMSII subset. The next slide, however, describes how you can force a subset index to be used in a query.

In addition, DMSQL does not do index scanning (**FIND KEY OF** operations) in executing a query unless the index has key data and predicates in the query refer to that key data.

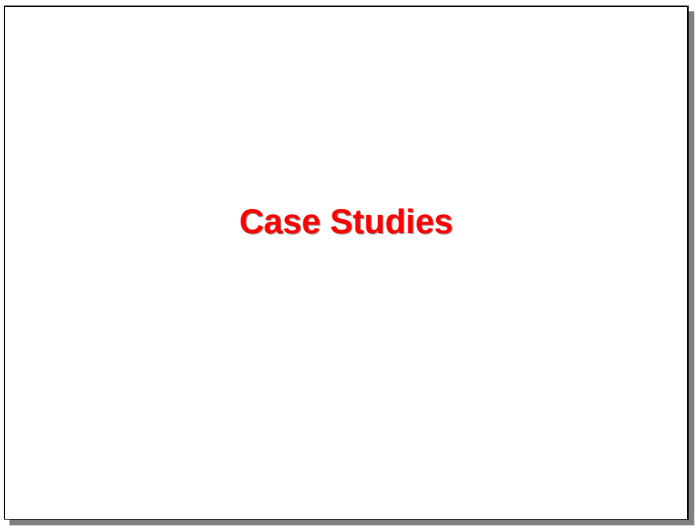


When DMSQL creates a relational mapping for a database, either through RDC or the **`SYSTEM/SQL/ADMIN`** utility, it automatically maps each DMSII set and subset index structure as a SQL view. The name of the set or subset index becomes that of the view. As with all SQL views, a view name can be used wherever a table name can be used in SQL syntax.

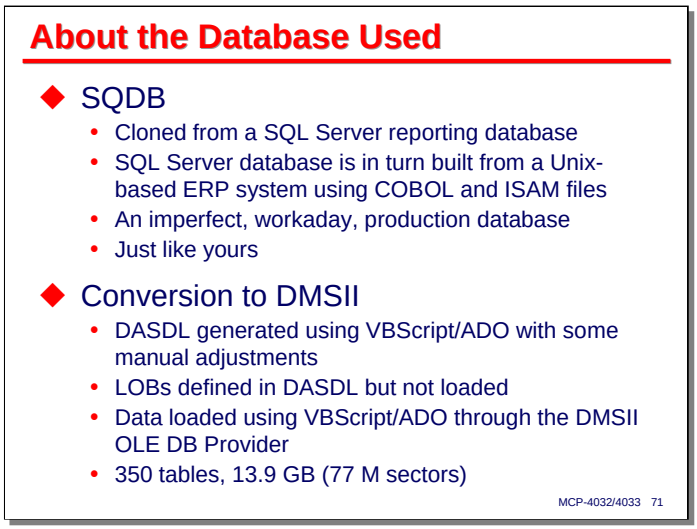
When you specify the view name of a DMSII index in a SQL query, that forces the query processor to access the associated table through that index. In other words, using these view names overrides the automatic index selection process in the query optimizer.

This can be an important optimization technique. You often have special knowledge of how well a specific index will filter the data, and by specifying the view name for that index, can force DMSQL to access the data through that index. This is especially true for subset indexes, which may span only a small fraction of the rows in the full table.

Be aware, though, that by manually specifying the index to be used, you inhibit DMSQL from choosing what could be a better index, if one should be added to the database at a later time.



With that general background on DMSQL performance, let us now examine a few test cases.



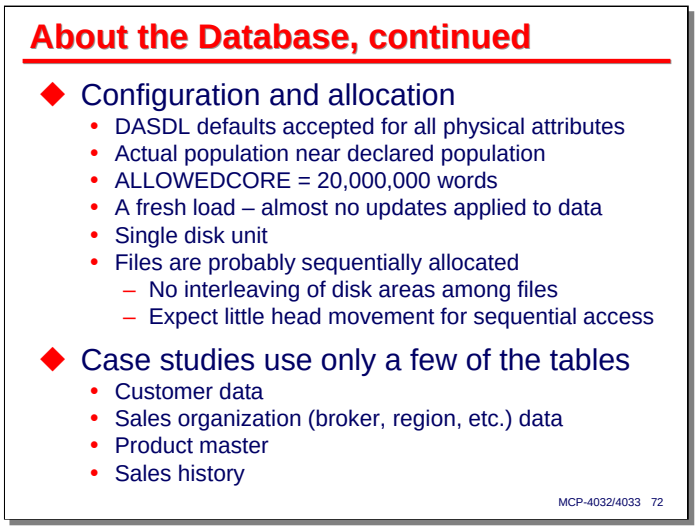
In order to attempt some realistic performance testing, I needed a realistic database, so I cloned one from a Microsoft SQL Server reporting database that was available. This SQL Server database in turn had been built from a Unix-based ERP system, which was implemented using COBOL and ISAM files. The structure of the data, while fairly well normalized, was closer to what you would normally see with a DMSII database rather than one specifically designed for a relational environment. For the purpose of this performance study, the database was named **`SQDB`**.

This is an imperfect, workaday, production database. It has its share of design anomalies, dumb ideas, and things that aren't needed anymore but still manage to get in the way. In that respect, its probably just like the databases you work with.

I converted this database to DMSII in a couple of stages.

- First, I generated DASDL using a VBScript/ADO program that mined the SQL schema for information on tables, columns, and indexes. That program attempted to translate those schema elements to reasonable DASDL equivalents. A number of hand adjustments were necessary in order to get the DASDL to compile.
- The SQL Server database had a number of LOBs, both for text and binary image data. These were carried over into the DASDL version, but were never used. DMSQL does not presently support LOBS in DMSII databases.
- The DMSII database was loaded using another VBScript/ADO program to extract data a row at a time from the SQL Server database and do inserts into the DMSII database through the OLE DB Provider for DMSII. This process took many days.

The result was a DMSII database with approximately 350 tables, occupying 13.9 GB, or 77 million 180-byte disk sectors.



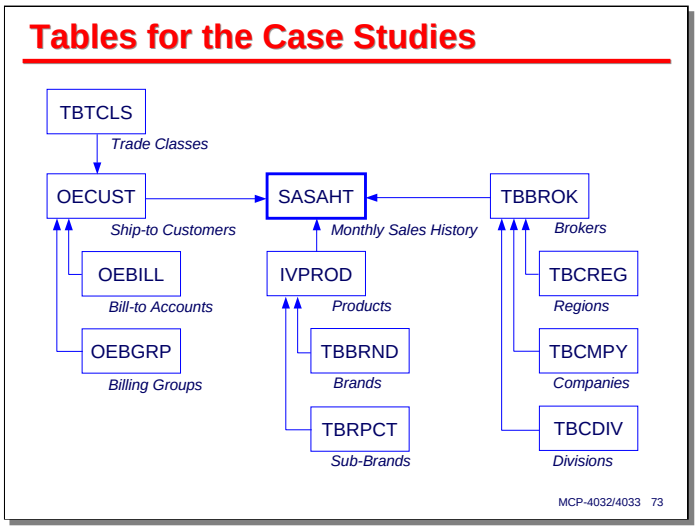
In generating the DASDL source, I left the assignment of all physical attributes except population to their defaults. The declared population for each table is in most cases just slightly more than the actual population (this was actually a bug in the DASDL generator program – the declared population was intended to be about 50% greater). The DASDL parameter **`ALLOWEDCORE`** was set to 20,000,000 words.

In viewing the performance results from this database, there are a few things you should keep in mind:

- Test runs were made against what is essentially a fresh load of the data. Almost no updates had been applied to the database since it was originally loaded from its SQL Server origin. This means that there are no holes created by deleted records, tables are physically ordered by their primary key, and most of the indexes should be, if not optimal, at least in a fairly good condition.
- The database occupied a single disk unit. All of the timings displayed in this presentation are from an LX170 laptop SDK system, so that disk unit is actually a virtual disk emulated with a large Windows file. Windows will have done some caching in addition to whatever the MCP and DMSII were caching.
- Because of the way the data was loaded, individual files are probably sequentially allocated on the disk. That is, all of the areas for a data set or set should be contiguous, or nearly so. This is not a typical allocation pattern for a production database, where file areas tend to be much more scattered on the physical disk surface. You should expect to see little disk head movement for sequential access within a data set or index.

These are all considerations related to the efficiency of physical I/O. In the end, these aspects of the physical file organization probably did not have much of an effect, due to the large amount of caching capacity and the fact that DMSQL tends to be CPU bound during execution of a query.

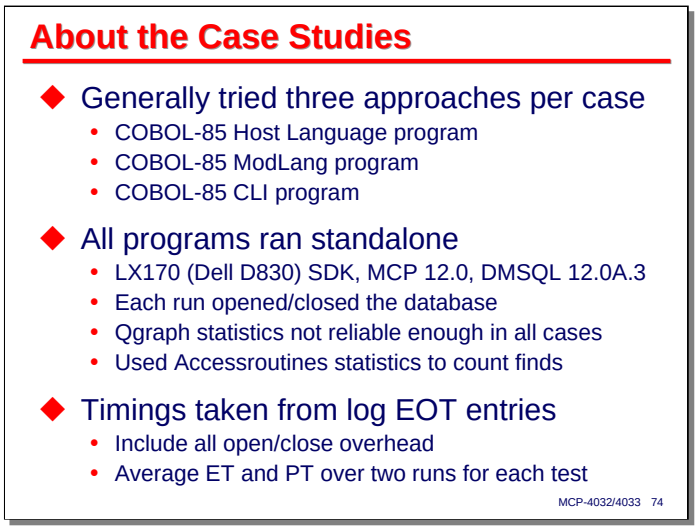
Another thing to consider is that the case studies that will be discussed shortly used only a small number of the tables in the database. Of the approximately 350 tables loaded, the case studies made use of a little over a dozen, primarily in the areas of customer data, sales organization data (brokers, sales regions, etc.), the product master file, and summary sales history data.



This slide shows a schematic diagram of the tables used in the case studies, indicating the relationships among them. Most of the larger tables had numerous indexes – too many to show and discuss here. A simplified version of these tables and relationships was used in many of the examples in the discussion of DMSQL dialect earlier in the presentation. The complete DASDL source is available with the other resources from the study from our web site, as cited in the references at the end of the presentation.

As we will see when we look at the queries for the test cases, the database uses a naming convention for its columns. The first four characters of a column name indicate the table, and come from the last four characters of the table name (the first two columns of a table name indicate the portion of the application it belongs to – **`OE`** for Order Entry, **`IV`** for Inventory, **`TB`** for general tables, etc.). The rest of the column name indicates the specific type of column. This convention is applied quite consistently throughout the tables.

For example, **`custcustomer`** is the customer identifier in the **`OECUST`** table. Similarly, **`sahtcustomer`** is the matching customer identifier in the **`SASAHT`** table. You will particularly notice this naming convention in the join conditions of the queries.



The focus of the performance study was primarily towards comparing DMSQL with the traditional DMSII host language interface. Therefore, the case studies involved writing a COBOL-85 program that implemented a query using the host language interface, a COBOL-85 program that implemented the same retrieval using ModLang, and usually (but not always) a COBOL-85 program that implemented the retrieval using CLI. In a few cases a retrieval was also done using SQL Server 2000 through the OLE DB provider.

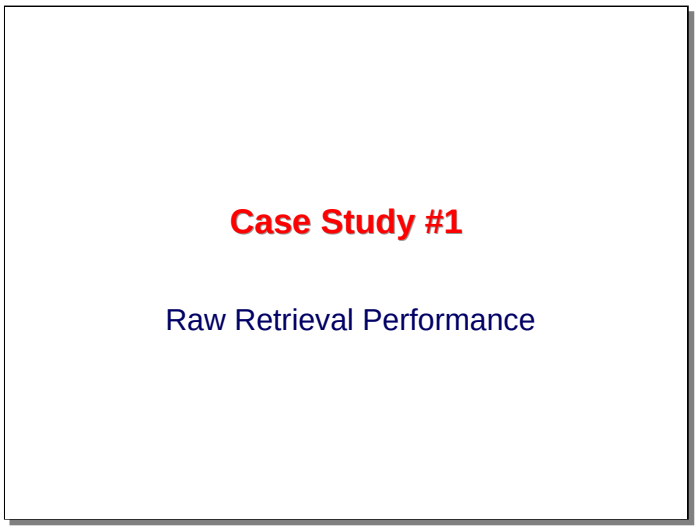
All test cases were run standalone on an LX170 SDK system. This was a Dell Latitude D830 laptop running Windows Vista SP1. The SDK was MCP 12.0, running the base release of DMSII and Interim Correction 012.0A.3 of DMSQL.

There were two main reasons for running the test programs standalone.

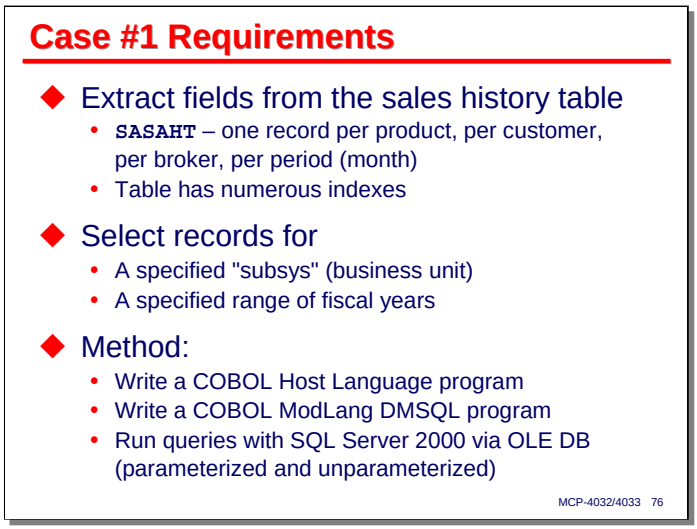
- This put all runs on an equal footing in terms of DMSII cache effects. It also charged all of the overhead for initiating and terminating the database and DMSQL software to the individual programs.
- Qgraph statistics exhibited too many problems for them to be used to determine either timings or the number of finds. Accessroutines statistics were used instead. In order for the Accessroutines statistics to accurately reflect each test run's activity, the runs had to be done standalone.

Timings for the test runs were taken from the system log EOT entries, except the elapsed times for the SQL Server/OLE DB runs, which were obtained manually using the time reported by SQL Server's Query Analyzer utility. Processor times for these runs were taken from the EOT entries for the associated **`SYSTEM/OLEDB/WORKER`** tasks. Each test run was executed twice, and the average elapsed and processor times for the two runs reported. There were some minor variations in the elapsed times, but processor times varied over only a narrow range.

Something to keep in mind is that the LX170, like all of the emulated MCP environments, relies on the underlying Windows operating system for time services. The Windows clock available to the MCP has a period of about 15 milliseconds, which is quite coarse, especially for the accumulation of processor times. The effect that this coarse granularity may have had on the precision of the timings is unknown.



For the first case study, let's simply retrieve some data from a table and measure the raw performance.

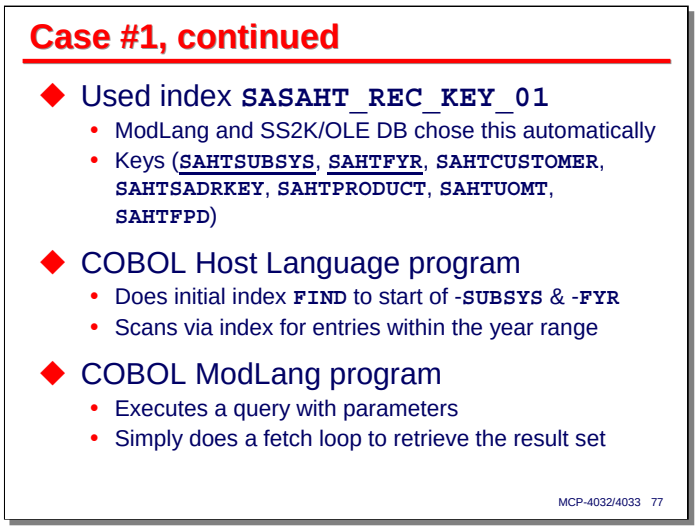


This case study uses a simple query to retrieve a large number of records from the database's **SASAHT** table. This is a table of summary sales history data. It has one record for each product, ship-to customer, broker (salesperson), and fiscal period (month). This table has numerous indexes, including at least one that is suitable for this type of retrieval.

This database is designed to support multiple business units, so the first column in almost every table row and index key is a "subsys" item – a small integer that identifies the business unit with which the row is identified. As we will see, this item is used in a predicate in almost every **WHERE** and join condition in the test cases.

For this first case, the query will retrieve all of the **SASAHT** rows for a specified "subsys" value and a specified *range* of fiscal years. There is no sorting or grouping of the data. The test programs will simply retrieve the data but not process it further.

To implement this case, I wrote a COBOL-85 program that used the host language interface and another COBOL-85 program that used the ModLang to define an equivalent SQL query. I also ran the same query using the Query Analyzer utility for Microsoft SQL Server 2000, connecting to the DMSII database through a linked server for the DMSII OLE DB provider. The SQL Server queries were run in two forms: parameterized and unparameterized (where the subsys and year query parameters were embedded literally in the SQL query text).

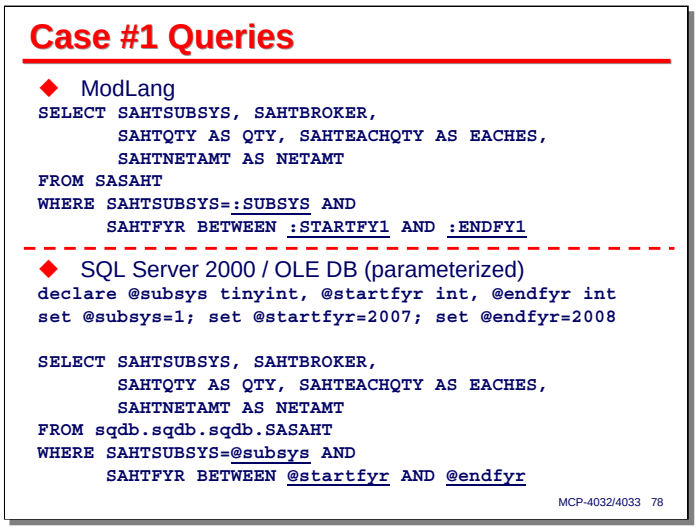


The host language program used the index **SASAHT_REC_KEY_01**, which has the key items shown on the slide. This is the most appropriate index for this problem, as its most-major key items are the subsys and fiscal year columns. Both the ModLang compiler and SQL Server automatically recognized this as the appropriate index to use, as well.

The COBOL host language program implemented this query in a straightforward manner. It did an initial find on the index to the first entry matching the subsys and starting fiscal year value. It then entered a loop, doing a find-next on the index until it encountered a record where the subsys changed or the fiscal year was greater than the ending year of the range.

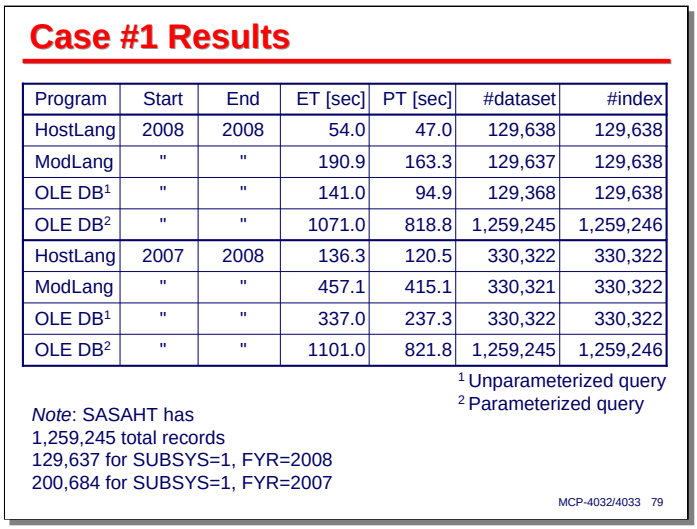
The COBOL ModLang program was also straightforward. It simply executed the query using a ModLang cursor, passing the cursor's open procedure the subsys and starting/ending year values. It then entered a loop, calling the cursor's fetch procedure to retrieve rows of data until the end of the cursor was reached.

For the SQL Server/OLE DB combination, the text of the query was entered in Query Analyzer and executed. For all of the test runs shown here, SQL Server and Query Analyzer were running on an entirely separate system, not on the Windows side of the LX170.



This slide shows the parameterized queries for the ModLang and SQL Server/OLE DB tests. Parameters in the ModLang are prefixed by colons (**:**) and in SQL Server by at-signs (**@**)

For the unparameterized SQL Server/OLE DB test, the "**@**" parameters in the query text were simply replaced with their corresponding literal values.



Case #1 Results

Program	Start	End	ET [sec]	PT [sec]	#dataset	#index
HostLang	2008	2008	54.0	47.0	129,638	129,638
ModLang	"	"	190.9	163.3	129,637	129,638
OLE DB[1]	"	"	141.0	94.9	129,368	129,638
OLE DB[2]	"	"	1071.0	818.8	1,259,245	1,259,246
HostLang	2007	2008	136.3	120.5	330,322	330,322
ModLang	"	"	457.1	415.1	330,321	330,322
OLE DB[1]	"	"	337.0	237.3	330,322	330,322
OLE DB[2]	"	"	1101.0	821.8	1,259,245	1,259,246

[1] Unparameterized query
[2] Parameterized query

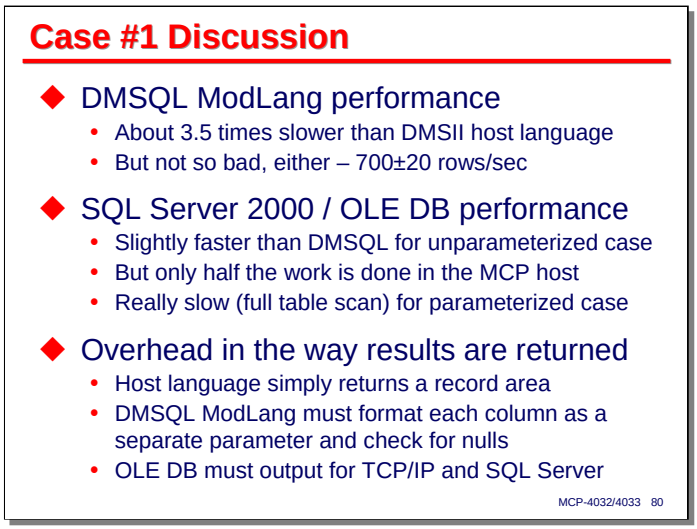
Note: SASAHT has
1,259,245 total records
129,637 for SUBSYS=1, FYR=2008
200,684 for SUBSYS=1, FYR=2007

MCP-4032/4033 79

This slide tabulates the results from the test runs for this case. Two sets of runs were performed – one where the starting and ending year were both 2008, and one for the range of years 2007-2008. Both sets of runs were for subsys=1, which is associated with about 80% of the data in the **`SASAHT`** table.

The **`SASAHT`** table contains a total of 1.26 million records, of which about 10% is for subsys=1, fiscal year=2008, and about 16% for subsys=1, fiscal year=2007.

- The program for the host language interface does exactly what you would expect. For each of the 129,638 records for subsys=1 and fiscal year=2008, it does a find on the index and a corresponding find on the data set. The results are equivalent for the 2007-2008 range of years.
- The ModLang program does the same thing, finding essentially the same number of index and data set records. The elapsed and processor times are substantially larger, however – about 3.5 times over that of the host language program.
- The SQL Server 2000/OLE DB results are very interesting
  - For the unparameterized case, the results are comparable to the ModLang results.The times are actually somewhat lower, but this reflects only the OLE DB portion of the work, as the SQL query processor is running in a separate Windows environment.
  - For the parameterized case, however, both the number of finds and the times are much higher than any of the other methods. Apparently using parameters in the query causes SQL Server to simply read the entire table and do all of the filtering of data itself, rather than pushing the filtering down to OLE DB, as obviously happened in the unparameterized case.



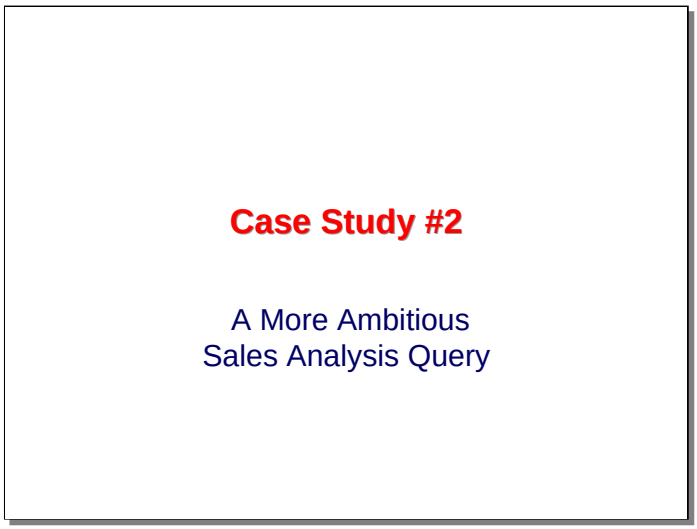
As we saw, the performance of the DMSQL ModLang program was about 3.5 times greater than that for the DMSII host language program. On the other hand, ModLang is returning an average of 700 rows per second to the application program – in absolute terms, that seems fairly impressive.

The SQL Server 2000/OLE DB performance is either somewhat better than ModLang or considerably worse, depending on whether a parameterized query was used or not. Clearly, SQL Server 2000 is not as smart about using the OLE DB linked server with a parameterized query as it is with an unparameterized one, reverting to a simple table scan (through an index, no less) in the parameterized case. Another consideration, as pointed out on the prior slide, is that with OLE DB, only half the work of the query is being done in the MCP environment, as the query engine is running in a separate Windows environment.

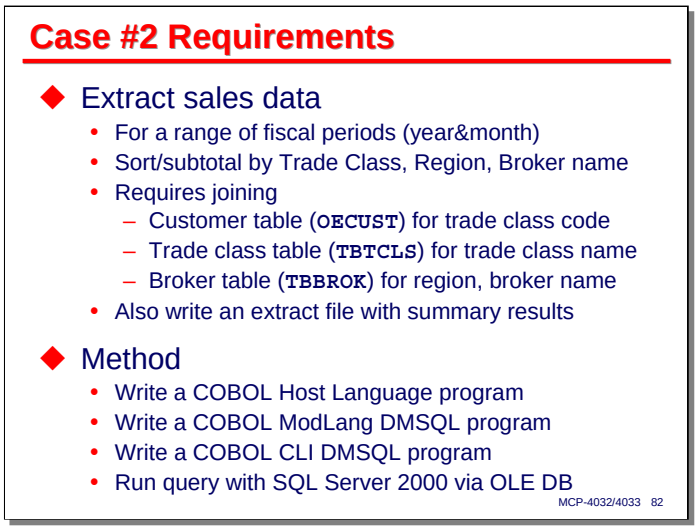
Since, with the exception of the parameterized SQL Server/OLE DB tests, the number of DMSII finds is the same, the difference in performance is probably not in I/O.

At least a partial explanation for the difference in elapsed and processor times is in the overhead that the different methods undergo. For the host language interface program, there is almost no overhead -- once the physical I/O buffer is present in memory, getting the data to the user program involves little move than a memory-to-memory move. For the ModLang interface, however, each column of the result set must be formatted as a separate parameter, including conversion to the specific data format specified in the ModLang procedure interface. Even with the compiled remap code, this is much more work than is required for the host language interface. Similarly, the OLE DB provider must format its results for transmission back to SQL Server over a TCP/IP connection.

Clearly, from this experiment, DMSQL does not come close to the DMSII host language interface in terms of raw retrieval performance.



The next case study builds on the first one to solve a more typical business problem – creation of a summary report with data from multiple tables.



For this case study, we will extract summary sales data, again from **SASAHT**, for a range of fiscal periods. A fiscal period is a combination of a fiscal year and a month. In this database, fiscal years are offset from calendar years by starting four months earlier.

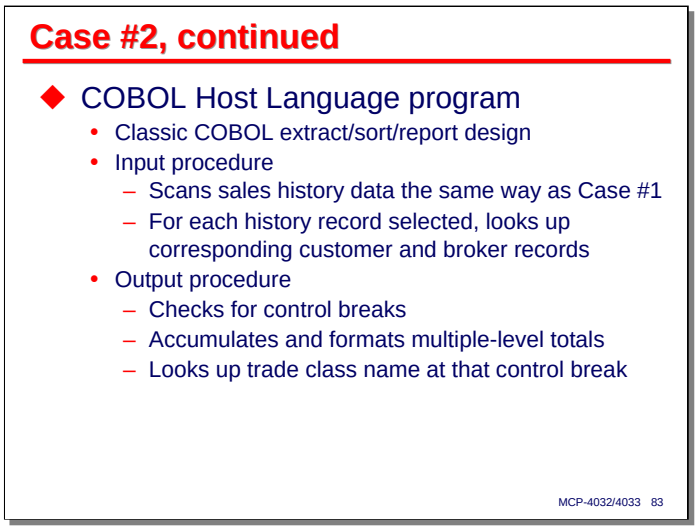
Once the sales data is extracted for the specified range of periods, we want it to be sorted and subtotaled by trade class (a customer attribute indicating what type of business they are in), the sales region, and the broker responsible for the sale. In order to do this, we need to join the sales history data with:

- The ship-to customer table, **OECUST**, in order to obtain the customer's trade class code, **custtradclas**.
- The trade class table, **TBTCLS**, in order to obtain the descriptive text, **tclsdesc**, that corresponds to a trade class code.
- The broker table, **TBBROK**, in order to obtain the sales region identifier, **brokregion**, and the name of the broker, **brokname**.

The COBOL programs will print a report and also write an extract file with the broker-level summary results.

For this case there are three COBOL-85 programs – one for the host language interface, one for the ModLang, and one for CLI. The same query will be executed using SQL Server 2000 through OLE DB, although for this no report or extract file will be generated, as the results will be produced by Query Analyzer on a separate Windows system.

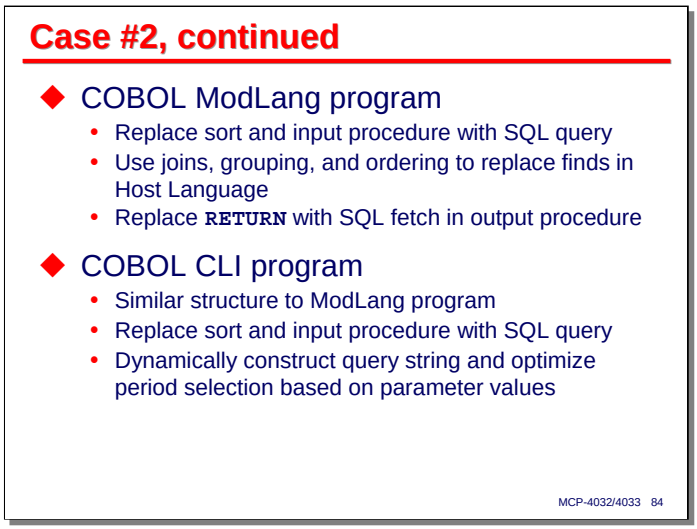
As the CLI program constructs and prepares the SQL query at run time, it embeds the query parameter values in the text of the query rather than binding parameter objects to the prepared query. It also examines the fiscal period values and optimizes the query's **WHERE** clause based on those values (e.g., reducing a **BETWEEN** predicate to an equality when the starting and ending years are the same).



The COBOL-85 host language interface program is written in the classic COBOL extract/sort/report form. The program is driven by a sort with an input procedure and an output procedure. The input procedure scans the sales history table, using the **`SASAHT_REC_KEY_01`** index as best it can, selecting records that fall within the specified range of fiscal periods. For each such record, it accesses the **`OECUST`** table and **`TBBROK`** table for the customer and broker associated with the sale, extracting a couple of fields from each and storing them in the sort record.

In the output procedure for the sort, the program retrieves the sorted records from within a series of nested **`PERFORM`**s, one **`PERFORM`** routine for each level of control break – trade class, region, broker, and detail sort record. These **`PERFORM`** routines are responsible for formatting headings, computing totals for their level, rolling those totals to the next higher level, and formatting those totals. They loop internally, calling the routine for the next lower level of control break, and exit back to the next higher level when the keys for their level of control change.

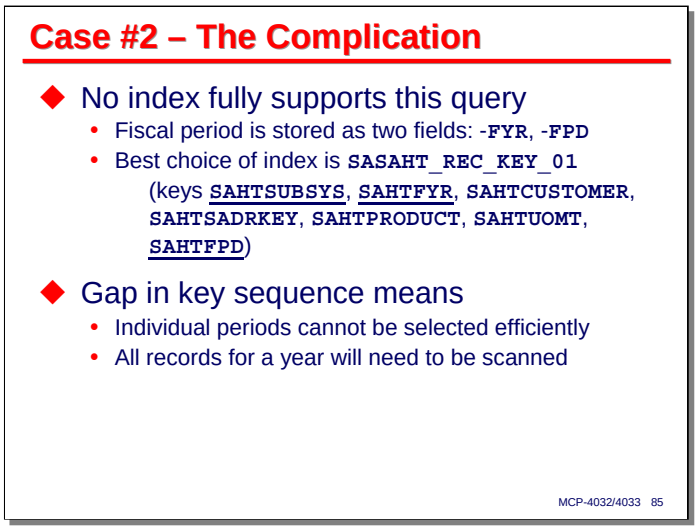
At the beginning of the **`PERFORM`** routine for the trade class level of control break, the routine accesses the **`TBTCLS`** table to translate the trade class code into it description. Thus, this table is accessed only once for each unique trade class code.



Both the ModLang and CLI programs are similar, and both were derived from the host language interface program. The sort and its input procedure were replaced with a single SQL query. That query is designed to return the same data items as the original sort did, and in the same order, with the exception that the grouping and accumulation of sales for each unique combination of trade class, region, and broker is done within the query rather than within the COBOL program. The query does joins to replace the separate lookups done in the host language interface program.

The output procedure hardly changed compared to the host language interface program, with the exception of replacing the sort's **`RETURN`** statement with a fetch to retrieve the next row from the query's result set. Some additional processing also had to be done to handle null values that could have resulted from the left joins in the query, and slightly different error handling and reporting checks, due to the differences between DMSQL error reporting and that of the host language interface.

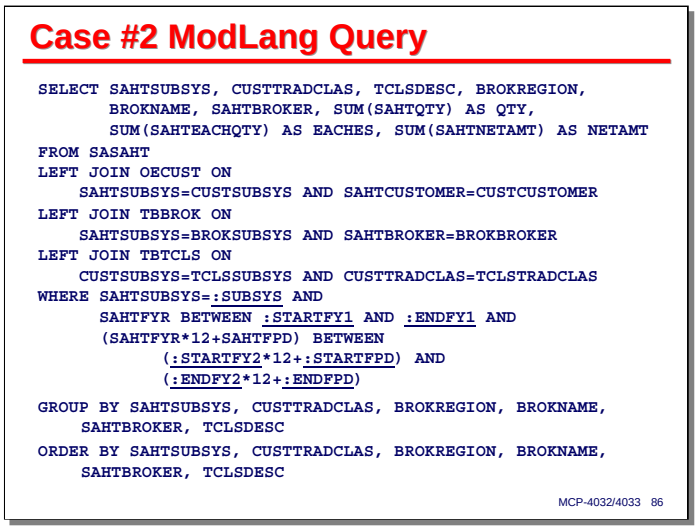
One intentional difference in the way the ModLang and CLI programs worked compared to the host language interface program was in the way the trade class code was translated to its description. Rather than do a separate query as the output is being formatted, the ModLang and CLI queries included an additional join, which effectively moved this lookup to the input phase. This approach is more in keeping with the "give the whole problem to the query processor" philosophy, but in fact increased the number of finds by about one-third, noticeably diminishing the performance of the DMSQL runs.



There is a complication in this case that affects both the host language interface program and the SQL-based versions – there is no index on **SASAHT** that fully supports this query. In addition, the fiscal period is represented as two fields, **-FYR** and **-FPD**, which makes testing for a range of fiscal periods both messy and difficult to optimize. Rather than try to fix this by adding a better index to the table, I left it alone, as it is a good example of the kind of imperfection in database design that we have to deal with all the time. Also, you usually cannot afford to have a custom index optimized for every type of query you want to do. Dealing with that issue is one of the reasons we want to use a query processor.

The best choice of index (in my opinion, at least) is the same one that was used for Case Study #1, **SASAHT_REC_KEY_01**. Notice from the keys illustrated on the slide that it has subsys and fiscal year as the most-major items. That is very good as those are the primary things we want to filter the data by, but fiscal period is the *most-minor* key item.

That gap in the key sequence means we will be able to use this index to efficiently filter out records only at the fiscal year level, and will need to scan all records for a year to see which ones fall in the monthly period range we seek. As will will see, this gap in the key sequence is also a problem for the DMSQL query optimizer.



This slide shows the text of the query as used with the ModLang. An equivalent query (differing only in the linked server convention for table names and the way that parameters are identified) was used for the SQL Server 2000/OLE DB runs. This query is somewhat more complex than others we have seen before, so I will step through its components one at a time:

- Starting with the **FROM** clause, we will be extracting data from the **SASAHT** sales history table.
- To that we need to join the **OECUST** customer table on the **-subsys** and **-customer** number fields to determine the associated customer's trade class code, **custtradclas**. I chose to do a left join here, so that if there are sales with a customer number that does not match a row in **OECUST**, the sales will still be reported. An inner join would have excluded such sales in the case of a customer mismatch.
- We also join the **TBBROK** broker table on the **-subsys** and **-broker** fields to obtain the broker's sales region and their name.
- We also join the **TBTCLS** trade class table on the **-subsys** and **-tradclas** fields to look up the trade class description using its code from the customer table.
- The **WHERE** clause is complex due to the way that fiscal periods are represented as two separate columns. First we restrict the result to sales for a specified subsys. We further restrict the result to rows with a specified range of fiscal years (this strictly isn't necessary, but was done in the hope that the query optimizer would be able to use this predicate to optimize access to the **SASAHT_REC_KEY_01** index – that turned out not to do much good). The final set of predicates compute a fiscal period by multiplying the year times 12 and adding the month, testing the result of this computation against the range of a similar computation on the parameters.
- The **SELECT** clause projects the subsys, trade class, trade class description, broker region, broker name, broker code, the sum of the sales quantity (in cases), sum of the sales quantity (in eaches – retail units), and the sum of the sales net invoice amount.
- The aggregates are computed over groups of rows for each unique combination of subsys, trade class code, broker region, broker name, broker code, and trade class description. Note that all of the non-aggregate columns in the **SELECT** clause have been specified in the **GROUP BY** clause, as this is required.
- Finally, the aggregated groups are sorted by subsys (this wasn't really necessary, since there can only be one), trade class code, broker region, broker name, broker code, and trade class description.

In the CLI program, the routine that built the SQL query text examined the range of years and fiscal periods, and attempted to simplify the **WHERE** condition. For example, if the starting and ending years were the same, the **BETWEEN** predicate was changed to an equality on the one year. If the range of months was for a whole year, the messy computation of a pseudo-period value was eliminated.

Case #2 Results

Program	Start	End	ET [sec]	PT [sec]	#dataset	#index
HostLang	2008/01	2008/02	86.2	78.2	178,121	178,123
ModLang	"	"	636.5	582.9	202,299	96,885
CLI	"	"	627.2	560.6	96,883	96,885
OLEDB[1]	"	"	171.0	118.1	147,932	147,932
OLEDB[2]	"	"	1122.0	858.3	1,280,315	1,280,317
HostLang	2008/01	2008/12	209.0	194.2	388,961	388,963
ModLang	"	"	1082.2	993.7	518,547	518,549
CLI	"	"	1127.0	1000.3	518,547	518,549
OLEDB[1]	"	"	204.0	118.7	147,932	147,932
OLEDB[2]	"	"	1148.0	857.9	1,280,315	1,280,317

Note: SASAHT has 1,259,245 total records
129,637 for SUBSYS=1, FYR=2008
24,221 for SUBSYS=1, FYR=2008, FPD=1-2

[1] Unparameterized query
[2] Parameterized query

MCP-4032/4033 87

This slide tabulates the test results for two sets of runs for subsys=1, one for fiscal 2008, periods 1-2, and one for all of fiscal 2008. As the slide indicates, the **`SASAHT`** table has a total of 1.26 million records, of which 129,637 are associated with subsys=1, fiscal year=2008, and 24,221 are associated with just the first two periods of that year.

The numbers to focus on for this slide are the times, as more detailed data for finds appears on the next slide.

The ModLang and CLI times for the runs involving just periods 1-2 are fairly close, although the CLI does significantly fewer overall finds, probably due to the run-time optimization of its **`WHERE`** clause. Both required a little over seven times the elapsed and processor time of the host language interface program. At least a part of that is due to the additional join on the **`TBTCLS`** trade class table.

The ModLang and CLI times for the runs involving the full 2008 fiscal year do a little better – about five times slower than the host language interface. In this case, ModLang performance was somewhat better than the CLI, even though it did the same number of finds.

With the SQL Server/OLE DB combination, we see much the same phenomenon we did for Case Study #1. Unparameterized queries perform significantly better than either ModLang or CLI, but the parameterized queries appear to revert to full table scans and perform either nearly the same or much worse.

Case #2 Per-Table Statistics

Program	SASAHT	index	OECUST	index	TBBROK	index	TBTCLS	index
HostLang	129639	129639	24220	24221	24221	24221	41	42
ModLang	129637	24222	24220	24221	24221	24221	24221	24221
CLI	24221	24222	24220	24221	24221	24221	24221	24221
OLEDB[1]	129638	129638	17981	17981	228	228	85	85
OLEDB[2]	1259245	1295246	20736	20736	249	249	85	85
HostLang	129639	129639	129636	129637	129637	129637	49	50
ModLang	129637	129638	129636	129637	129637	129637	129637	129637
CLI	129637	129638	129636	129637	129637	129637	129637	129637
OLEDB[1]	129638	129638	17981	17981	228	228	85	85
OLEDB[2]	1259245	1295246	20736	20736	249	249	85	85

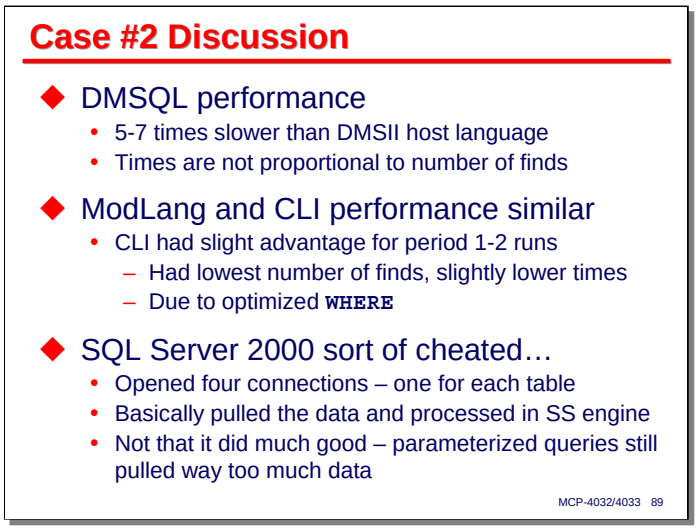
Note: SASAHT has 1,259,245 total records
129,637 for SUBSYS=1, FYR=2008
24,221 for SUBSYS=1, FYR=2008, FPD=1-2

[1] Unparameterized query
[2] Parameterized query

MCP-4032/4033 88

Looking at the number of finds on a per-table and per-index basis for this case is even more interesting.

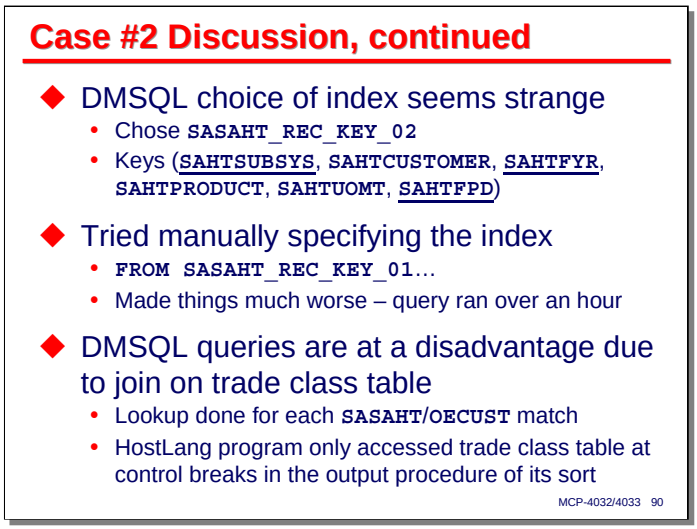
- The host language interface program needs to scan all of the **SASAHT** records for fiscal year 2008, as we would expect from the key structure for the **SASAHT_REC_KEY_01** index. It selects 24,221 records for periods 1-2 and all 129,637 records for the full year, so it has to do one **OECUST** and one **TBBROK** lookup for each of these. Only 41 **TBTLCS** lookups are necessary, since these are done in the sort's output procedure just once for each unique trade class value.
- Except for the number of **TBTLCS** lookups (which are done by a join in the main query instead of lookups at control breaks in the output phase), the number of finds done by the ModLang program is approximately the same as the host language interface program. There is an anomaly for the **SASAHT** table, however. While the index statistics show that only 24,222 finds were done for this table, the data set statistics show that 129,637 finds were done. This suggests that DMSQL is doing multiple data set fetches for each selected row. The thing that is not apparent from this table is that the index for **SASAHT** is not **SASAHT_REC_KEY_01**, as you might expect, but **SASAHT_REC_KEY_02**, which is primarily an index on subsys and customer. Why DMSQL chose this index is a mystery. This abnormality in the number of finds does not show up for the full year run, even though it uses that same **REC_KEY_02** index.
- Optimization of the **WHERE** clause gave the CLI program a real advantage in the number of finds for the period 1-2 case, requiring substantially fewer than even the host language interface program. This was offset somewhat by the additional finds caused by the join to the trade class table. Optimization of the **WHERE** clause generated no benefit in terms of the number of finds for the full-year case.
- The SQL Server 2000/OLE DB runs appear to be working by simply extracting all of the records for each table and doing all of the finer selection in SQL Server on the Windows side. For unparameterized queries, SQL Server requests all records for the specified subsys; for parameterized queries, it appears to request all records in tables **SASAHT** and **OECUST**, but only those for subsys=1 from **TBBROK** and **TBTCLS**.



As seen from the prior slides, DMSQL performance for this case is 5-7 times slower than the host language interface program, but the differences in times are not proportional to the differences in numbers of finds. This suggests that substantial amounts of processor time are being invested in grouping, aggregation, and possibly sorting and internal control of the query process.

The ModLang and CLI performance are similar, except where CLI was able to do fewer finds due to optimization of its **`WHERE`** clause, which was effective only for the period 1-2 case and not the full year. In the full year case, ModLang ran slightly faster than CLI, which is to be expected, since it is normally the more efficient of the two interfaces, and the run time does not include query prepare time.

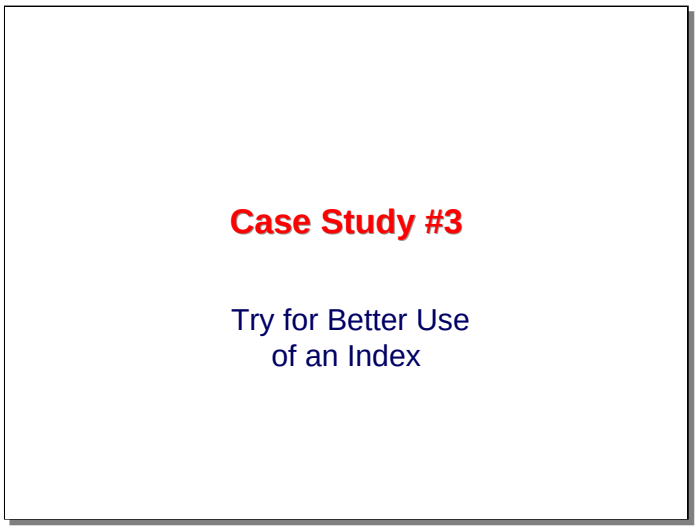
Interestingly, the SQL Server/OLE DB combination appears to have achieved the level of performance it did by opening four connections to the database – one for each table, and probably using four separate threads in SQL Server. SQL Server 2000 does not appear to have made very good use of the capabilities of the OLE DB provider, at least for this query, and especially in the case of the unparameterized queries.



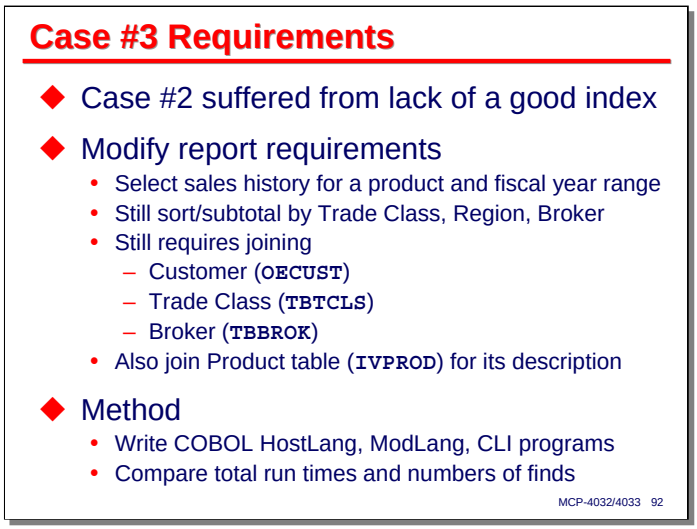
As noted earlier, the choice of index DMSQL made for this query is very strange. **SASAHT_REC_KEY_02** has the **-SUBSYS**, **-FYR**, and **-FPD** items as key values, but only **-SUBSYS** is a most-major key. This means the index cannot be used to optimize finds below the **-SUBSYS** level. It would seem that the lack of an index that fully supports the predicate items is really confusing the query optimizer, although why it chose this index is something of a mystery.

Since I didn't like the index that DMSQL chose, I tried to force the issue by specifying the view name for **SASAHT_REC_KEY_01** in the **FROM** clause of the query. Surprisingly, this made things much, much worse. The query ran about seven times slower than before, requiring over an hour to finish.

As mentioned before, the DMSQL queries were at a (somewhat intentional) disadvantage compared to the host language interface program, due to the join on the trade class table, **TBTCLS**. This caused a lookup on the table for every **SASAHT**/**OECUST** row selected, instead of one lookup for each unique trade class value. This inflated the number of finds by about one third, and probably inflated the overall time by about as much.



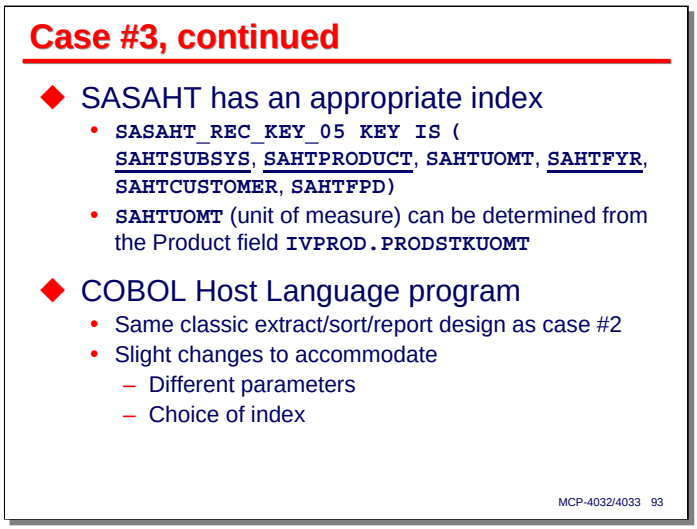
Having gotten somewhat disappointing results with the relatively complex query in the prior case, this next case study is designed to make better use of an index on the sales history table.



The last case study suffered from the lack of a good index with which to retrieve sales history data by fiscal period. This could have been helped somewhat by creating a new index that had fiscal year (**FYR**) and fiscal period (**FPD**, month) adjacent in the most-major portion of the key, but having the period represented as separate year and month fields would still be difficult to handle.

This case study modifies the requirements to select sales history for a specific product and range of fiscal years. We will still sort and subtotal sales data by trade class, region, and broker, and that will continue to require us to join the customer, trade class, and broker tables. Since we are selecting by product, we will also join the product table, **IVPROD**, to get the product description.

As before, we will employ COBOL-85 programs to implement these requirements, one for the host language interface, one for the ModLang, and one for CLI. These programs are slightly modified versions of the ones used in Case Study #2. We will not attempt a SQL Server 2000/OLE DB in this case.

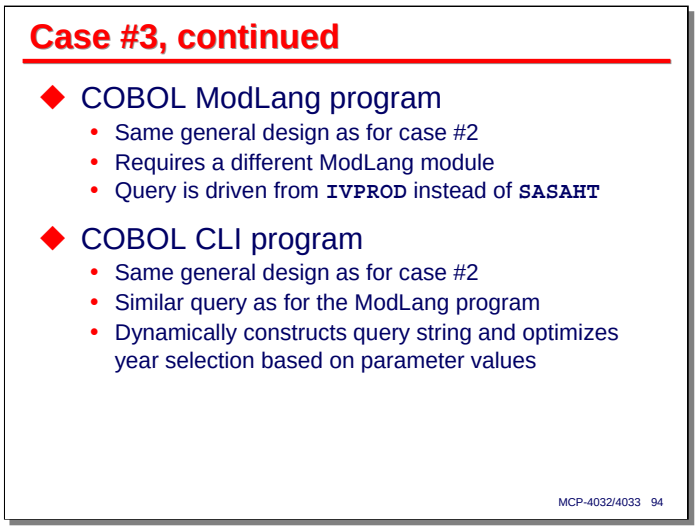


There is an index for the sales history table, **SASAHT**, that supports this query, but it requires a little explanation. **SASAHT_REC_KEY_05** has the keys shown on the slide. It has the subsys and product codes as most-major keys, an item named **SAHTUOMT** (the transactional unit of measure – cases, drums, pallets, etc.), and then the fiscal year key item. Normally this would be a gap in the major key sequence and would inhibit optimization of the query down to the fiscal year level.

As it happens, in this particular database, the presence of the unit of measure item in the key is a historical artifact in the system. At one time, individual product codes could have multiple units of measure, but this is no longer the case. **SAHTUOMT** is actually dependent on **SAHTPRODUCT**, and the associated unit of measure value can be obtained from the product master table, **IVPROD**, in field **PRODSTKUOMT**. This is obviously not good normalization practice, but this is a real database, and not one that was originally design to be a relational database, so we have to take it as it is.

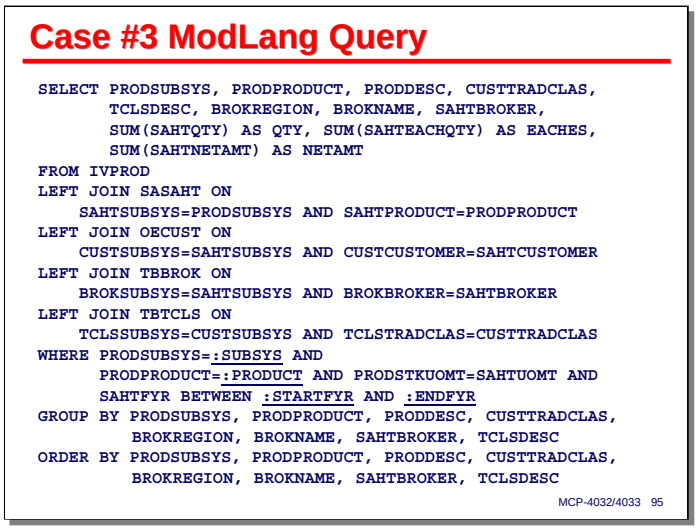
We will see how the unit of measure key is dealt with when we look at the query text shortly.

The COBOL host language interface program has the same classic extract/sort/report design as the one in Case Study #2. The primary changes were due to processing different parameters and using a different index to extract data from the sales history table.



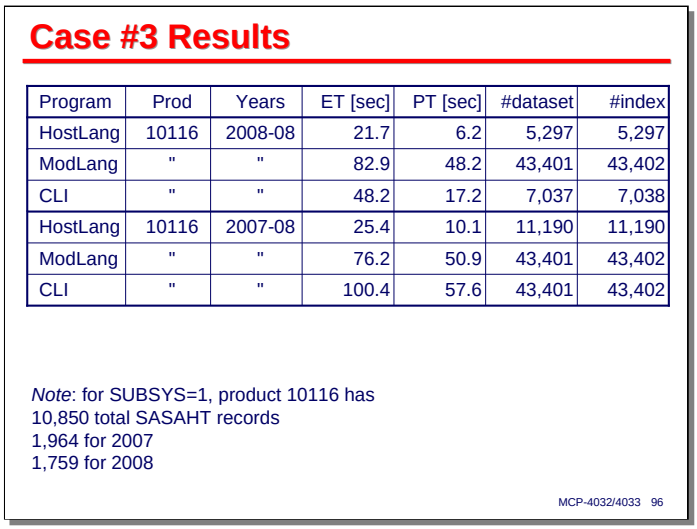
The ModLang and CLI programs are also very similar to their counterparts in Case Study #2 and were derived from them. The ModLang program required a different cursor definition and procedures to manipulate that cursor. These could have been put in the same module as the cursor and procedures for the prior case study, but I chose to define them in a separate module instead. Because the query is based on a product code, I decided to "drive" the query from the product table, **`IVPROD`**, instead of the sales history table, **`SASAHT`**.

The query for the CLI program is basically the same, except that once again I took advantage of the CLI's ability to dynamically tailor the query text to the task at hand. If the fiscal year parameters span only one year, the CLI program will use an equality (**`=`**) predicate instead of a **`BETWEEN`** predicate in its **`WHERE`** clause.



This slide shows the text of the query as it is defined for the ModLang cursor. It is similar to the query for Case Study #2, but with these differences:

- The **SELECT** clause includes the product code and description.
- The **FROM** clause has the product master, **IVPROD**, as the first table, and then joins the sales history table, **SASAHT**, to it based on the matching subsys and product values. The rest of the joins are the same as before.
- The **WHERE** clause is substantially different. The subsys and product predicates are based on fields in **IVPROD** instead of **SASAHT**. The predicate for the date range is much simpler – just a **BETWEEN** on the starting and ending years. In the CLI program, this will be replaced by an equality (**=**) predicate if both years are the same.
- Also note the additional predicate **PRODSTKUOMT=SAHTUOMT**. The intent of this is to have the predicates match a contiguous sequence of most-major key items in the **SASAHT_REC_KEY_05** index – subsys, product, unit of measure, and fiscal year. This predicate could also have been specified in the **ON** phrase of the join for **SASAHT**, and in fact the query ran in the same amount of time when I tried this.
- The product code and description have been added to the **GROUP BY** and **ORDER BY** clauses. Strictly speaking, subsys, product code, and product description should not need to be part of the **ORDER BY** clause, since they will have the same values for all rows of the result set. However, these fields are required in the **GROUP BY** clause, since they are non-aggregate items in the **SELECT** list. By having the **GROUP BY** and **ORDER BY** clauses the same, DMSQL should be able to use the same sort for grouping and ordering.



Case #3 Results

Program	Prod	Years	ET [sec]	PT [sec]	#dataset	#index
HostLang	10116	2008-08	21.7	6.2	5,297	5,297
ModLang	"	"	82.9	48.2	43,401	43,402
CLI	"	"	48.2	17.2	7,037	7,038
HostLang	10116	2007-08	25.4	10.1	11,190	11,190
ModLang	"	"	76.2	50.9	43,401	43,402
CLI	"	"	100.4	57.6	43,401	43,402

Note: for SUBSYS=1, product 10116 has
10,850 total SASAHT records
1,964 for 2007
1,759 for 2008

MCP-4032/4033 96

This case study was run with two sets of parameters. The first for product 10116 in just fiscal 2008. The second was for the same product, but spanned 2007-2008.

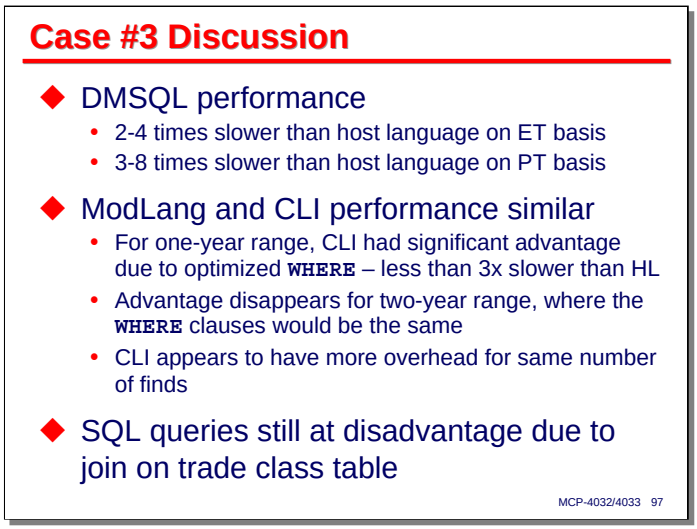
For the first run, the ModLang program was almost four times slower than the host language interface program on an elapsed time basis, but the CLI program was only slightly more than twice as slow. This is most likely due to the CLI program optimizing its **WHERE** clause to use an equality rather than a **BETWEEN**. The CLI program also had a significantly lower number of finds against both the data set and index than the ModLang program, but still about one-third more than the host language interface program. The difference between ModLang and CLI is again probably due to the optimization of the year predicate.

The additional finds the CLI program did over those of the host language interface program are once again due to the join on the trade class table, as with the ModLang and CLI programs that must be done for every **SASAHT** row selected, whereas with the host language interface program, the trade class table must be accessed only for unique values in the program's output procedure.

For the second run, the ModLang program performed better than the CLI program, as the CLI program could not optimize its year predicate and used the same one as the ModLang program. The ModLang program is 3:1 slower than the host language interface program on an elapsed time basis, and about 5:1 slower on a processor time basis. The CLI program is about 4:1 slower than the host language interface program on an elapsed time basis and a little more than 5:1 slower on a processor time basis.

The number of finds for the two DMSQL programs are the same, and almost four times as many as the host language interface program. This difference is due to additional finds against **SASAHT**, along with the joins to **OECUST**, **TBTCLS**, and **TBBROK** that selecting those additional **SASAHT** records generate. From the Accessroutines statistics it can be seen that all **SASAHT** records for subsys=1, product=10116 are being accessed, regardless of the year, which means that query optimization is not able to use the index down to the year level.

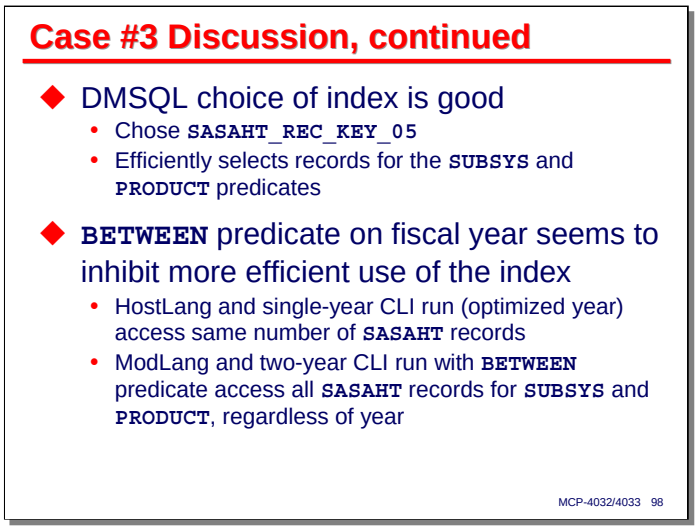
It's clear from the results for the first CLI run (2008 only), which used an equality predicate on year, that the query optimizer was able to use the index to get down to the year level (1,759 rows **SASAHT** rows for subsys=1, product=10116, fiscal year=2008, plus equal numbers of finds for the **OECUST**, **TBTCLS**, and **TBBROK** tables, plus one find for the **IVPROD** table equals the 7,037 data set finds for that case). For the 2008-only ModLang run, and both ModLang and CLI runs for 2007-2008, which all use **BETWEEN** predicates for the year, the number of finds is four times 10,850 (the number of **SASAHT** records for subsys=1, product=10116) plus one, or 43,401. Thus it appears that DMSQL is not able to optimize the **BETWEEN** predicate on fiscal years.



DMSQL fares mostly better with this test case than the prior one, although there is much more variance among the runs than in the earlier cases. It is approximately 2-4 times slower on an elapsed time basis and approximately 3-8 slower on a processor time basis than the host language interface program.

ModLang and CLI performance are similar, but the first run for CLI (covering only one year) was substantially better – only slightly more than twice as slow as the host language interface program. This appears to be entirely due to the optimization of the **WHERE** clause the CLI program does when it constructs the query text, collapsing the **BETWEEN** predicate into an equality predicate. The time improvement disappears in the second run (covering two years) since both ModLang and CLI use the same **WHERE** clause with a **BETWEEN** predicate. As expected, the CLI program appears to have slightly more overhead than the ModLang program when doing the same number of finds.

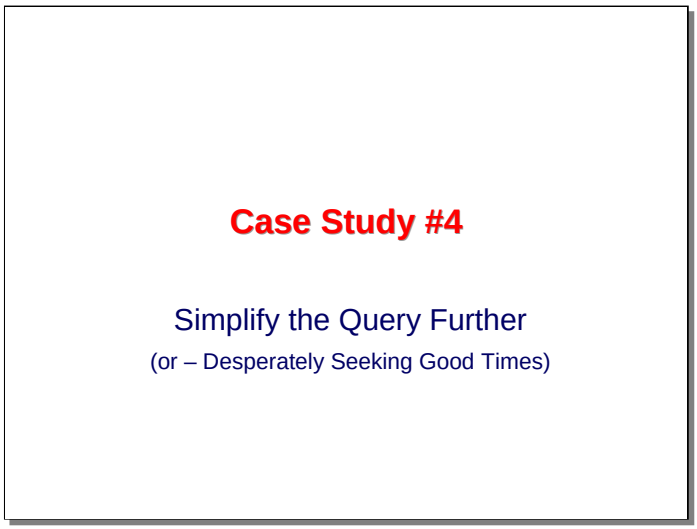
Both the DMSQL programs are still at a disadvantage compared to the host language interface program, as they have a join on the trade class table, **TBTCLS**, and that requires them to do an additional find for each selected row of the sales history table, **SASAHT**. This increases the number of finds they must do over over those the host language interface program must do by about one third.



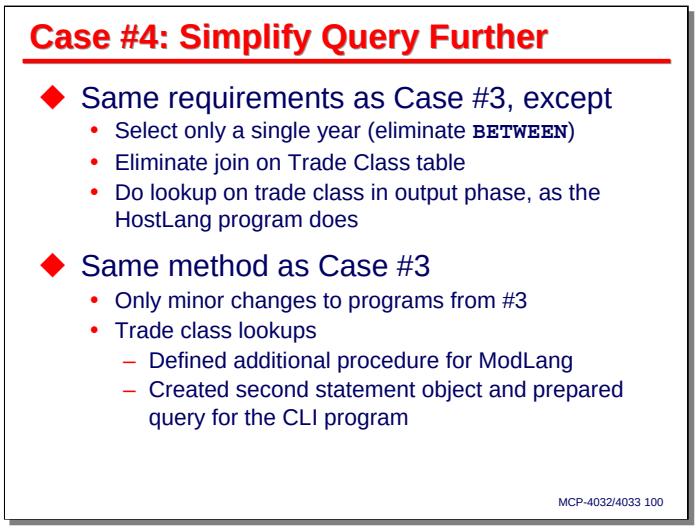
One piece of good news with this case study is that the query optimizer picked a good index – the same one as used in the host language interface program. This index efficiently selects sales history records for a given subsys and product code.

The thing that seems to make a big difference in performance in these runs is the use of the **BETWEEN** predicate for the fiscal year parameters. In the case where the CLI program can reduce the **BETWEEN** to an equality predicate, the query runs much faster, and (except for the extra finds due to the join on the trade class table) does very nearly the same number of finds as the host language interface program.

It also appears that the attempt to deal with the goofy unit of measure key item (**SAHTUOMT**) in the **SASAHT_REC_KEY_05** index does not work that well when there is a **BETWEEN** predicate for the year range in the **WHERE** condition.



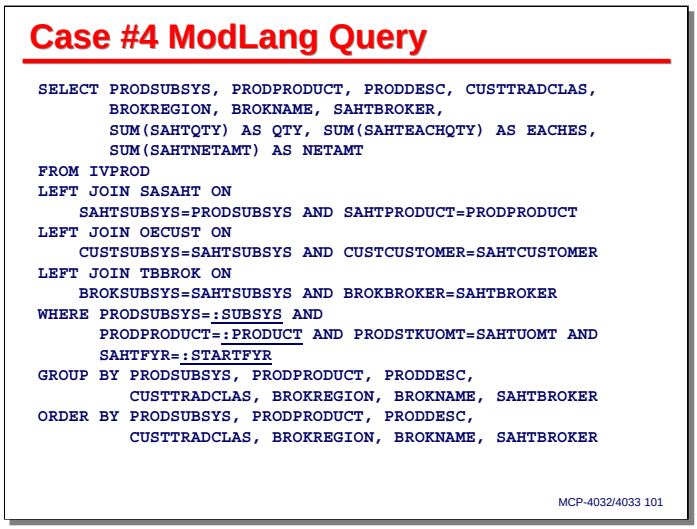
For the last case study, we will simplify the query from Case Study #3 a little further, to see if DMSQL can perform better with a more straightforward set of selection criteria.



This case study is identical to that in Case Study #3, except for two things:

- We will select product sales history only for a single fiscal year. This will eliminate the **BETWEEN** predicate that seems to have inhibited better optimization in the prior case.
- We will eliminate the lookup of trade class description (and the corresponding join on **TBTCLS**) from the query. We will define a second, simple query to do just the trade class lookup in the output phase of the ModLang and CLI programs. This will put the DMSQL and host language interface programs on a more equal footing in terms of the number of finds they will be required to do.

Only minor changes are required to the programs from Case Study #3 to adapt them to the new requirements here. The biggest change is in doing the additional query in the output phase for the ModLang and CLI programs.



This slide shows the text of the query as it is used to define the cursor in the ModLang program. The same text is used for the CLI program, except that, as in all the other prior cases for CLI, the predicate values are embedded literally in the query text instead of being passed in as parameters. Note that all references to the **TBTCLS** table and its columns have been removed from the query.

Case #4 Results

Program	Prod	Year	ET [sec]	PT [sec]	#dataset	#index
HostLang	10116	2008	11.8	6.1	5,297	5,297
ModLang	"	"	43.2	10.8	5,295	5,296
CLI	"	"	52.2	15.2	5,295	5,296
HostLang	10110	2008	32.5	21.0	31,311	31,311
ModLang	"	"	79.3	47.9	31,309	31,310
CLI	"	"	101.9	53.2	31,309	31,310

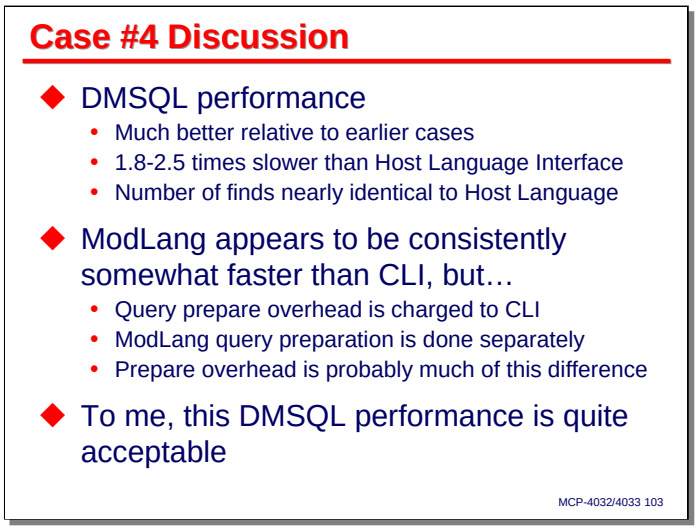
Note: for SUBSYS=1,
product 10116 has 10,850 total records, 1,759 for 2008
product 10110 has 91,828 total records, 10,427 for 2008

MCP-4032/4033 102

This case study was run for two different product codes in the same year. Product code 10116 has 10,850 total sales history records on file, of which 1,759 are in fiscal 2008. Product code 10110 has 91,828 total sales history records on file, of which 10,427 are in fiscal 2008.

As we can see from the results, the performance difference between the host language interface program and the two DMSQL programs has narrowed quite a bit. On an elapsed time basis, the DMSQL programs are running about 2.5-4.5 times slower, but on a processor time basis, they are only 1.8-2.5 times slower. At least some of the elapsed time difference is due to startup delays for the DMSQL libraries, which must be opened as well as the DMSII database.

The good news is that both DMSQL programs do essentially the same number of finds as the host language interface program. This means that DMSII is doing the same amount of work for each of the interfaces. The difference in the times between the DMSQL programs and the host language interface program are most likely due to the more general nature of DMSQL operations and the overhead of formatting individual result set columns rather than just returning a record area to a program.

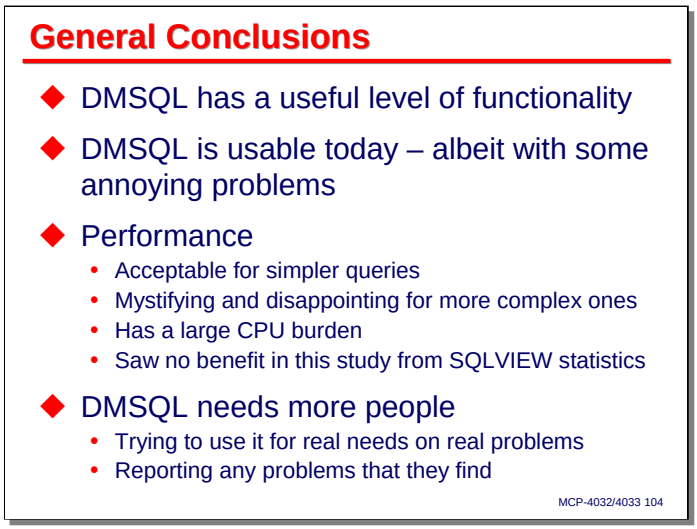


To summarize the findings from the prior slide, the relative DMSQL performance is much better in this case study – only 1.8-2.5 times slower than the host language interface program. All interfaces are doing the same numbers of finds.

The ModLang interface appears to be consistently somewhat faster than CLI. This is to be expected. Also note, however, that the query preparation time (which averages 2-5 seconds on the LX170) is being charged to the CLI program, whereas query preparation for the ModLang was done separately and not charged to its execution time. The prepare time is probably a good portion of the difference between the two, at least for processor time.

To me, the performance DMSQL exhibited in this case study, while not spectacular, seems quite acceptable.

The other thing to note is that, even though this query is quite a bit more complex than the simple data retrieval one for Case Study #1, the relative time difference between host language interface and DMSQL in these results is significantly smaller than those for the results of that first case study. This indicates that there actually is some advantage to doing more work inside the query processor, so the potential for SQL to perform better than record-at-a-time retrieval interfaces is certainly there. Whether that potential can be realized with the existing design of DMSQL running on top of the existing design of DMSII is another question, and one which will require some time to resolve.



I can draw a few general conclusions from the limited work I've done thus far on DMSQL performance.

First, DMSQL has a useful level of functionality, at least for queries. There are a few features I'd like to see, especially conditional expressions (e.g., **`ifnull`**, **`case`**) and derived tables (using the results of one **`SELECT`** statement as a table in the **`FROM`** list of an outer **`SELECT`** statement). For most business-oriented data retrieval tasks, however, what it currently has is good enough.

I think DMSQL is usable today, even though it has a few annoying problems. The current problems with Qgraphs were particularly frustrating – they seem work great in the simpler cases when you don't need them, and don't work in the more complex cases where the type if information they report would be very useful. In general, I found DMSQL to be very opaque – it's essentially impossible to find out what is happening when it is not doing things the way you think it should. Perhaps some of the information in Qdumps would be useful along that line, but there are currently no resources to help you understand what a Qdump is reporting.

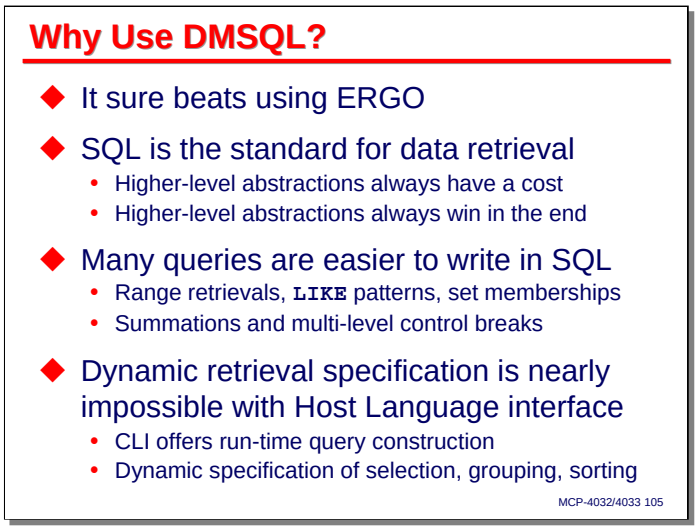
DMSQL is at a substantial performance disadvantage compared to the traditional DMSII host language interface, but considering how little the host language interface does, that should not be surprising. For a couple of the case studies, I think the performance of DMSQL is acceptable. For the time being, a 2:1 difference between DMSQL and DMSII host language is probably about the best we can expect.

The performance difference between DMSQL and the host language interface for the larger and more complex queries is both disappointing and mystifying. I think DMSQL should be able to do much better with these queries, but SQL query optimization is difficult and complex. Hopefully, this is something that can be improved upon over time.

One thing is very clear from the performance studies – DMSQL has a large CPU burden. Query speed seems primarily to be a function of the speed of the processor.

I saw no benefit from enabling SQLVIEW statistics in the performance studies. That probably does not mean much about the value of SQLVIEW statistics – I suspect that the queries I used simply were not of the type that could benefit from knowing actual table populations.

Finally, I think what DMSQL really needs is more people using it for real needs on real problems. The only way a large, complex piece of software gets to be any good is if it has a relatively large user population that tries to apply it to a wide variety of problems in a wide variety of ways, and then expresses dissatisfaction (i.e., files UCFs) when the results are not up to their expectations. That is how DMSII got to the point where it is today. If we want DMSQL to be a better tool, we need to start using it.



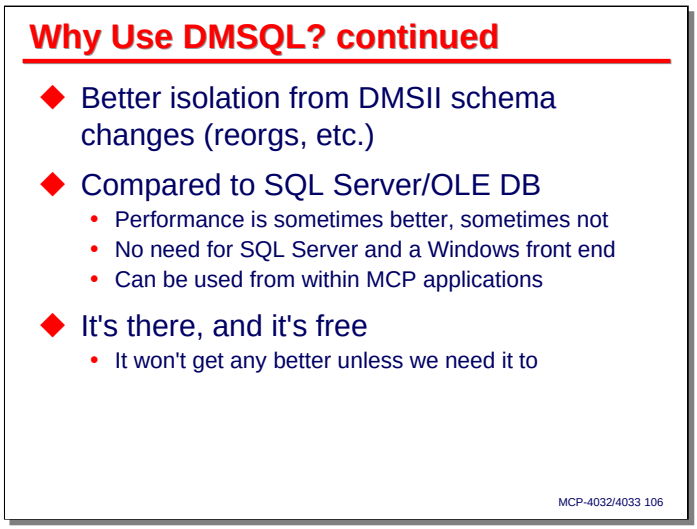
Even if the performance were better, I doubt that anyone would be interested in going through their applications and replace the host language interface code with DMSQL code. So, with all of the issues that have been discussed over the last couple of dozen slides, why should we use DMSQL?

Well, for starters, it's a lot better than ERGO. While ERGO has some output capabilities that the DMSQL user interface tools do not, in terms of ad hoc data retrieval capability (the **TAB** command in ERGO), SQL is a much better data retrieval language. In my experience, DMSQL is much more efficient as well.

Second, SQL has become the standard for data retrieval. DMSQL may not be as efficient as the host language interface at present, but it is a higher-level abstraction of the problem. The whole history of computing has been filled with these performance vs. level of abstraction issues – among them assembly language vs. compilers, files vs. databases, and host language vs. SQL – and the higher level abstraction always wins in the end. The reason for this is that the limit to what we can accomplish in software is our ability to handle complexity, and using higher-level abstractions reduces the amount of complexity we must handle, leaving us free to think about bigger problems.

Third, many queries are easier to write in SQL, often much, much easier, especially if you need to retrieve a range of records, use the pattern matching of **LIKE** predicates, or retrieve data based on set memberships. Summations with multi-level control breaks are also quite easy. In doing the case studies, I would design the query and then write the host language interface code to implement it. I found that I had to keep going back to the host language code and fix minor problems with the record selection logic. Once you start thinking about data retrieval in terms of sets, it's difficult to go back to thinking about it in terms of loops and **IF** statements.

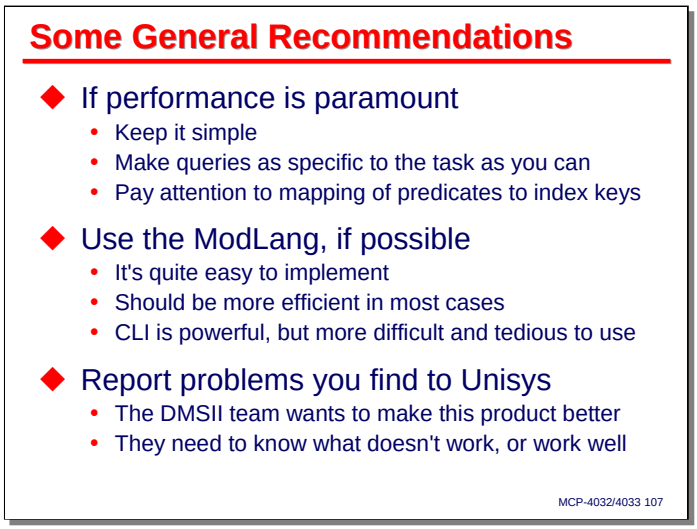
Fourth, if you need to decide at run time what the selection criteria, grouping, or ordering for the data will be, doing that with the host language interface is difficult and often impossible. With DMSQL this is quite easy. You need to use CLI to do this with DMSQL, but the alternative is DMINTERPRETER, which is no less difficult and nowhere near as efficient.



Fifth, because the composition of a SQL result set is completely divorced from the table structures it is generated from, using DMSQL provides better isolation for application programs from DMSII schema changes. Changes to record formats generally require you to recompile host language interface programs. That is not the case with DMSQL. In fact, with the CLI, you do not compile anything for a specific database. The name of the database resource your program opens is passed as a parameter to the CLI open routine. It is easy to write a CLI program that will switch among various databases, as long as you know the table and column names involved. You can do this to some degree with ModLang, by compiling separate but equivalent module libraries for different databases, and then dynamically selecting the library you want to call at run time.

Sixth, compared to doing SQL queries through the OLE DB provider with Microsoft SQL Server as a front-end query engine, DMSQL sometimes performs better and sometimes not. With DMSQL, however, there is no need for SQL Server and a separate Windows system. What is more, there is no straightforward way to use SQL Server/OLE DB from within MCP applications, as you can with DMSQL.

Finally, you should try using DMSQL simply because it's there. It is bundled with every release since MCP 10.1. As I mentioned a couple of slides ago, DMSQL may have some problems at present, but it won't get better unless we need it to, and the way to address that issue is by using the product.

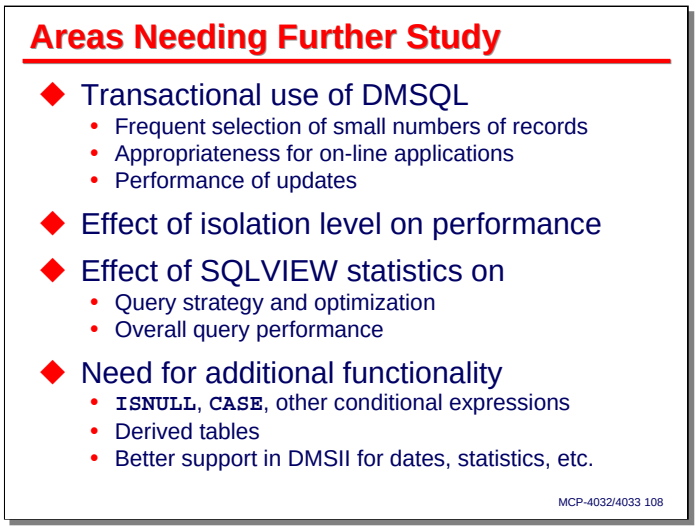


Here are a few recommendations I'll offer based on my experience thus far with DMSQL. These may not be universally applicable, but I think you will find them generally useful.

First, if performance is paramount, keep the query as simple as you can. It's tempting to exploit the power of SQL, but at least at present, query complexity seems to carry a significant performance cost. Make your queries as specific to the task as you can, either by writing multiple queries for multiple types of retrievals, or using the CLI to retrieve data based on dynamically generated query text. Pay particular attention to the mapping of your predicates to key items of indexes. The better your predicate items map to the most-major keys of an index, the fewer records DMSQL will need to access.

Second, use the ModLang with your application programs if possible, especially when you are first starting out. It is quite easy to set up and use, and in most cases should be more efficient than CLI. The CLI is very flexible and powerful, but it is much more difficult to learn and actually quite tedious to code, at least in COBOL.

Third, it is not unlikely that you will encounter some problems with DMSQL, either in terms of functionality or performance. Please report those to Unisys. The DMSII Data Access team wants to make this product better, and they need to know from us what isn't working, or what isn't working well.



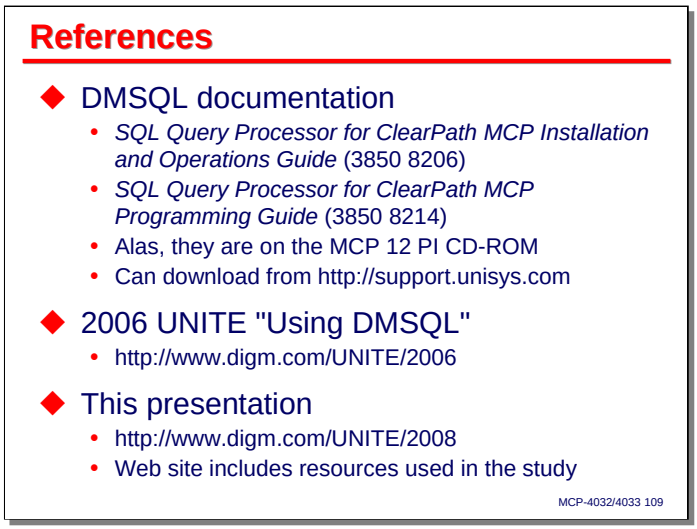
The performance studies I've done for this presentation have concentrated on fairly large data retrievals, such as you would use in reporting applications. There are a number of other facets to DMSQL performance that need to be looked at.

Probably the most important one is transactional performance. By that I mean doing many frequent, time-critical queries that touch at most a few tables and usually return only a few rows. This is the type of database access you generally see in on-line applications. For this type of query, the overhead of initiating the query and managing it internally will be much more pronounced, and it will be interesting to see how DMSQL stacks up against more traditional techniques.

Another aspect of performance is the effect that transaction isolation level has on it. DMSQL supports three levels: read uncommitted (corresponding to a DMSII **`FIND`** statement), read committed (corresponding to a DMSII **`SECURE`** statement), and serializable (corresponding to transactions that use **`SECURE STRUCTURE`** statements on all of their tables).

In my work with DMSQL, I did not see any benefit from enabling SQLVIEW statistics. I suspect that the queries I was using simply don't benefit much from knowing actual table populations. This is an interesting area for further research.

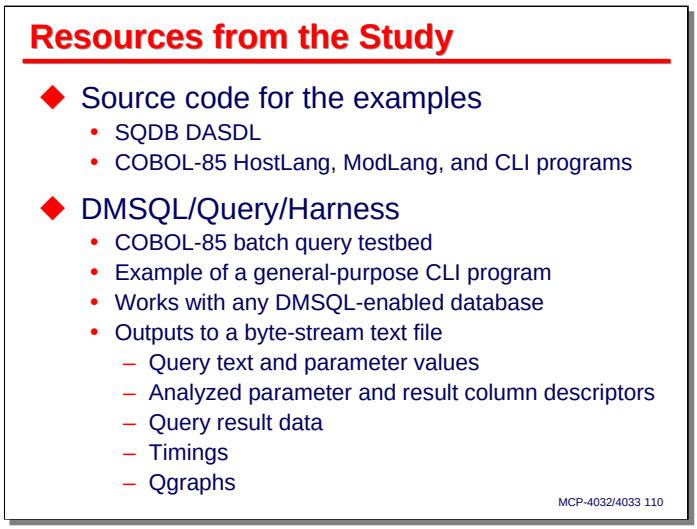
It would also be interesting to know how useful some additional functionality for DMSQL would be and what effect that additional functionality would have on performance. I have already mentioned a desire for conditional expressions and derived tables. DMSQL could also use some support in DMSII for dates and times, and some persistent statistics, such as key value distributions, which could be used by the query optimizer in choosing indexes and other aspects of query strategy.



There are two manuals for DMSQL, an Installation and Operations Guide and a Programming Guide. Unfortunately, they were omitted from the Product Information CD-ROM for MCP 12. You can download them from the Unisys support web site, however.

My DMSQL presentation from the 2006 UNITE conference is still available on our web site. You may find that useful for more detailed information on setting up DMSQL and using the ModLang and CLI interfaces.

Finally, this presentation will also be on our web site. The source code for all of the case studies and some other tools I developed will be there as well.



As I mentioned on the prior slide, I will make all of the source code from the study available on our web site. That includes the DASDL for the SQDB database, and all the COBOL-85 test case programs.

I wrote a CLI-based query testbed for some initial studies that were not discussed in this program. None of the existing tools could give a consolidated output containing query text, parameter and result set descriptors, query results, timings, and Qgraphs. The **`DMSQL/Query/Harness`** program does all of that, and is an example of writing a general-purpose data retrieval tool using the CLI. This program is also included in the source code. You are welcome to use and adapt this program in any way you wish.

END

DMSQL Query Capabilities and Performance

2008 UNITE Conference

Session MCP-4032/4033