Steps for Successful DataServer Development and Deployment

Best Practices Series: Progress® DataServer Technologies

Simon Epps, DataServers Product Manager

PROGRESS
SOFTWARE

Contents

Steps for Successful DataServer Development and Deployment 1
Best Practices Series: Progress® DataServer Technologies Simon Eps, DataServers Product Manager .. 1
Contents 2
Introduction 4
DataServers in a Nutshell 4
Steps for a Successful Adoption of DataServer Technology 5
1) Understand the DataServer Technology 5
2) Eliminate Unsupported 4GL and DBMS Features 5
3) Set Goals and Expectations 5
The DataServer Elements and Their Relationships 6
The Application Layer 6
The DataServer Layer 6
The Target DBMS 6
Checklist for Optimizing DataServer Elements 7
Optimize the Application 7
Designing Applications to Allow Access to More Than One Data Source 7
Low Risk, High Gain 8
Higher Risk, Higher Gain 8
Optimize the DataServer 9
Optimize the DBMS 9
Test and Analyze Functionality 9
Test and Analyze Performance 9
Appendix A 10
Eliminate Unsupported 4GL and DBMS features 10
Three steps for success 10
Step 1 for Implementing the DataServer Technology 10
Step 2 for Tuning the DataServer Technology 10
Step 3 for Final Adoption of the DataServer Technology 10
Unique Progress RDBMS Features 11
Word Indexes 11
Record Locking: Progress vs. the Rest of the World 11
Exclusive Lock 11
Share Lock 11
No Lock 12
Oracle Locking 12
Automatic Optimistic Locking Support 13
Microsoft SQL Server & ODBC Locking 13
Record Scoping / Availability 15
Transaction Boundaries 16
Appendix B 18
Optimization – the Application, the DataServer, the DBMS 18
The Application 18
Expected Behavior of the Application 18
Constraint Violations / Trigger Execution 18
Error Messaging 18
Records 19
Word Indexes 19
Descending Indexes 19
Assign Unique Index Fields on Create 20
Setting ROWID/RECID 20
RECIDs and ROWIDs 20
RELEASE and VALIDATE Statements 20

Application Performance 21
Reduce Network Traffic and the Read / Write Access to the Database 21
Action Segment Overflow 21
Cursors 22
Standard Cursors 22
Lookahead Cursors 22
Cursor Repositioning 22
Influencing Query/Browse Performance 23
RECORD Retrieval 23
Field Lists 23
INDEXED-REPOSITION Support 24
Descending Indexes on a FIND Statement 24
USE-INDEX 24
Join-By-SQLDB 25
FOR FIRST in Place of FIND FIRST 26
Progress 4GL vs. SQL 26
DBMS Stored Procedures 27
DBMS Stored Procedures and Transaction Scoping 27
Native SQL Syntax Support 27
Distributed and Batch Processing 27
Mass Database Modifications 28
The DataServer 28
The Progress Schema Holder 28
Effects of Database Changes 28
Code Recompilation Due to Schema Changes 28
Startup Parameters 30
Client Connection Parameters 30
Progress AppServer 31
Skipping Schema Verification 31
The DBMS 32
Hardware 32
Network Configuration 33
Naming Conventions 33
Database Limitations 33
Meta-Schema References 34
Data Types 34
Additional Database Objects Required 34
Arrays 34
Case-Insensitive Indexes 35
RECIDs & ROWIDs 35
Unknown Values / NULLs / Zero-Length Character Strings 35
Fixed vs. Variable Length Character Strings 35
Trailing Blanks 36

Introduction

This white paper is a guide to help project managers and application developers understand and estimate the work involved in adopting the Progress DataServer technology. This document contains generic advice for the ODBC, Oracle, and MS SQL Server Progress DataServer products, (Progress Version 8 and higher). After reading this document you will be able to determine how the Progress DataServer technology will be able to fulfill your particular business needs.
For a more in-depth technical explanation on 4GL and non-Progress DBMS requirements, please read the following white papers:

Progress DataServers at a Glance

Building High Performance Applications with the Progress® Oracle DataServer

Configuration and Coding Techniques: Performance and Migration: Progress DataServer for Microsoft SQL Server

DataServers in a Nutshell

The DataServer technology allows you to execute your existing Progress application against a non Progress database. The DataServer allows for the develop and maintenance of Progress source code using all the Progress development tools such as ProVision™, WebSpeed®, WebClient™ and the AppServer™. Database migration tools and utilities are included for migration and tuning.

The DataServer products convert Progress 4GL into SQL, which is then automatically executed against a non-Progress database, thus allowing your Progress 4GL application to run directly against any certified SQL-compliant DBMS, such as Oracle or MS SQL Server, (please review the *Progress Product Availability Guide* for other supported SQL-compliant DBMSs). Native Progress record locking and transactional processing logic are preserved, thus allowing you to migrate your application to a non-Progress DBMS with little or no code alterations. The potential amount of code alterations depends upon a number of elements. This document has been designed to highlight and provide practical advice on these elements.

The DBMS component of the DataServer solution also falls outside of Progress Software's control: hardware requirements and database administration/tuning are the responsibility of the DBMS vendor. The Progress DataServer can be visualized as just another SQL client making SQL calls against the target DBMS. There are two main steps when adopting the Progress DataServer technology. Step one is to tune the application for the target DBMS (for instance the Oracle and MS SQL Server Databases do not natively support the Progress SHARE-LOCK or the functionality of Progress Word Indexing). Step two is tuning the environment and application for anticipated performance.

- Step One: Eliminate DBMS differences (Appendix A)
- Step Two: Optimize the Application, the DataServer, and the DBMS (Appendix B)

There is an inherent performance degradation when translating 4GL to ODBC or the Oracle Call Interface (OCI) and then back again from ODBC or OCI to the 4GL. An example of this occurs when the DataServer generates 4 SQL calls to execute one 4GL call. This is a weakness of SQL (and a strength of 4GL), not a weakness of the DataServer.

The rest of this white paper concentrates on understanding and implementing these two phases of successful DataServer adoption.

Steps for a Successful Adoption of DataServer Technology

The following mandatory steps are to ensure the efficient adoption of the DataServer technology.

1) Understand the DataServer Technology

Understanding how DataServer technology works is an essential step in ensuring that your application executes and performs correctly and efficiently. There are three main DataServer components: the 4GL source code, (your application), the DataServer product, and the target DBMS.

The DataServer technology automatically converts the Progress 4GL (your application) into SQL; this SQL is then executed in real time by the foreign DBMS. The target DBMS returns a SQL results set back to the DataServer layer which is then translated back into the Progress 4GL.

To support the differences between the Progress RDBMS and the target DBMS, the DataServer technology employs a *schema holder*, which is a Progress database that contains all of the necessary data type mappings between the Progress and target databases. The DataServer product provides real time 4GL to SQL to 4GL translations and native DBMS connectivity.

2) Eliminate Unsupported 4GL and DBMS Features

Not all target DBMS Databases can support certain native Progress constructs such as word indexing, SHARE-LOCKS, and transaction boundaries. To handle these 'unsupported 4GL' features, your application needs to be made flexible enough through the use of Progress preprocessors or conditional programming logic. If not you will see compilation and runtime errors. (See Appendix A for a detailed explanation of these unsupported 4GL features.)

3) Set Goals and Expectations

The source 4GL and the target DBMS are the two major 'unknowns' in the DataServer equation, the source 4GL could resort in a single record or a multi table join read and the database may have 10 or 10 million rows (not to mention the possible differences in hardware configurations).
If we assume the performance of the Progress and target DBMS to be equal, then running your application via the DataServer will generally result in a slower execution time. It is almost impossible to estimate how much slower due to the many factors involved, such as the complexity of the 4GL source and the performance of the target DBMS. It is essential that you test your application for performance and scalability.

Commit to testing your application internally. If you plan to deploy your application to 200 users, then be prepared to test with 200 users. Performance and scalability are substantially impacted by how you've written your application. These factors are outside the ability of Progress to anticipate or correct. Due to the unknown nature of how the application is written, there is a substantial amount of RISK until actual testing has been done. ***Commit to being an expert on the back-end database***. It may be necessary to write database-independent code (that is, DBMS-specific features such as stored procedures). The need for performance and scalability may outweigh the need for a single source code base. DataServers may not be a 'load and go' solution for your particular application, instead, they can require substantial application changes. How much? Progress doesn't know how you wrote your application, so we cannot provide this answer. DataServer performance has an inherent performance degradation because of the process of translating 4GL to SQL (Oracle or ODBC) and then SQL results sets back to 4GL. An example of this occurs when the DataServer must generate four SQL calls to execute one 4GL call. This is a weakness of SQL (and a strength of the 4GL), not a weakness of the DataServer. The BEST performance possible for Progress-based applications is with the Progress RDBMS.

The DataServer Elements and Their Relationships

The three main components of the DataServer technology consist of your application, the DataServer technology, and the target DBMS (Oracle, MS SQL Server, or ODBC-compliant system).

The Application Layer

The Progress 4GL is a very rich language and is tuned to work with the Progress RDBMS, some 4GL constructs are very difficult to simulate in SQL, and so the resulting SQL the DataServer generates may not look as efficient as if you were writing a SQL application from scratch. Constructs such as the default Progress block and transactional (undo, retry) properties need to be simulated to ensure that your applications business logic is not compromised. Generally, the more complex the applications 4GL logic, the more complex the resulting SQL. At a very simplistic level, the DataServer technology converts a Progress 4GL application into a SQL application, so any independent third-party suggestions for SQL tuning for both client and server processes is a helpful and useful resource. Most SQL applications leverage DBMS stored procedures for load balancing and performance, but this does mean source code changes, the balance of performance against database independence code needs to be evaluated, which can only be done if you have clear goals and expectations on how you expect your application to work with the DataServer technology.

The DataServer Layer

The Progress DataServer translates Progress 4GL into SQL and then translates the SQL results set back into Progress 4GL This translation is not instantaneous and the more complicated the Progress 4GL or untuned the target database the less instantaneous the translation can become. This analogy is not just true for the DataServer world. It also holds true when running Progress 4GL against a Progress database, or running SQL against Oracle or MS SQL Server. It is easy to translate and understand individual words but sentences take a bit more to translate especially as the target language does not naturally support the grammar. When comparing a purely Progress solution against Oracle or MS SQL Server, you must not only appreciate the additional translation process but also the transport layer and the target DBMS being used. SQL access for Oracle is achieved via the Oracle Call Interface (OCI), while ODBC is used against MS SQL Server and other certified ODBC databases. One should not assume that these access layers and target DBMSs are the same when it comes to performance and functionality. The most efficient database for the Progress 4GL is the Progress RDBMS, even when the application is complicated and the database untuned.

The Target DBMS

Not all DBMSs are equal. The DataServer layer makes every effort to tailor efficient SQL for the particular data source but the resulting system loading and query-optimization routes fall under the domain of the target DBMS. OLTP or OLAP applications place differing demands on the target DBMS, and you will need specialized knowledge in analyzing and tuning these DBMSs. For system requirements, the DBMS vendor or third party SQL application suppliers (for example, http://www.sap.com/benchmark/) are a useful resource for information.

Checklist for Optimizing DataServer Elements

Unfortunately, there is no single solution for performance optimization. In actuality, optimization is the sum of all of the other topics discussed in this whiter paper. However, there are some areas that have the potential for large gains in performance. For most environments, the area that yields the greatest performance benefits is through programming techniques. That is not to say that the other areas, such as networking and hardware, are not important. You should make sure that your hardware and software configurations are set up properly. Once this is done, the majority of your time for performance optimization should be spent on programming issues.

The search for performance improvement begins by understanding all of the software and hardware components involved in the DataServer environment. This means learning about the DataServer, the Progress 4GL, and DBMS. The following section lists a few suggestions on potential performance improvements not covered elsewhere in this document.

The following sections give a high level view of what needs to be done, while Appendix B covers optimization techniques available for all the DataServer components.

Optimize the Application

The normal Progress 4GL tuning techniques can be applied here; the use of the Progress Debugger as well as the Progress Profiler tool are recommended. In addition, Progress is developing source scanning tools to help evaluate your application's 4GL as to the potential coding changes needed. (Contact your local Progress representative for more details.)

Designing Applications to Allow Access to More Than One Data Source

It is common for developers to design their applications so that they can substitute different data sources. This is also known as making the code database independent. This goal allows the code base to be the same wherever possible for connections to a Progress database, a SQL Serve database, or any other data source. For example, one set of source code would be able to access ether a Progress database or a SQL Server database. To allow this type of data source substitution requires some planning in the design phase so that you need to maintain only a single source.

Applications already designed to access a DataServer are likely to need very few changes, if any, to allow them to access other vendors' databases. Keep in mind that if the application references DBTYPE anywhere within the code, you must modify it to work with this database type.
As with all DataServer products, there are two phases to the migration process. In the first phase, you get the application to work against the new data source. In the second phase, you evaluate and modify the application as needed to obtain the desired performance. Changes made to optimize the application for a particular data source environment might not work, or might not work well, for a different data source. For performance reasons, it is likely that you will need to make specific modifications to an application when adding support for a new data source. The number and scope of the modifications depend on the target performance criteria.

The vast majority of the Progress 4GL code does not require any changes, regardless of the data source. Programming changes that may need to be done can be categorized into two types. The first type is code that, although it needs to be modified to work with the DataServer, can be changed in a way that allows the same source to run against any data source. The second type of change requires unique source code based on the connected data source. For example only the Progress RDBMS supports word indexing, so compiling against other data sources will fail if 4GL code specific for word indexing is referenced (i.e. CONTAINS).

Almost all of the changes will be of the first type that allow for code that will run against any data source. There are coding techniques that are valid for a Progress database that are not valid or optimal for the DataServer environments. However, almost all of the coding techniques that are

valid for DataServer environments are also valid for the Progress database environment. Therefore, when making coding changes you should use options that are valid for both environments whenever possible. All changes should be tested for all environments that will be supported to confirm compatibility.

In the few cases where a particular code segment cannot be made the same for all environments, you can use preprocessors to designate the data source. Using a preprocessor allows the code for the correct data source to be selected at compile time.

You must set the definition for the preprocessors prior to compile time. One way to simplify setting these types of preprocessor is by putting them into an include file that is included globally in all source code; or, if there are only a few programs that need the preprocessor definitions, you might choose to place the include file only in those programs.

For example a preprocessor might be defined like: &GLOBAL-DEFINE DB-TYPE MSSQLS

The preprocessors could then be referenced as: &IF DEFINED ({&MSSQLS}) &THEN

Or : &IF NOT DEFINED ({&MSSQLS}) &THEN

A benefit to using this method is that it allows you identify all of the DataServer-specific areas in the code quickly.

NOTE: Although you can design and maintain one set of source code for multiple data sources, the compiled code, r-code, is unique to each data source. This means that the source code must be compiled against each data source. Once you have modified your source code so that it is database-independent, Progress Software Corporation recommends that you implement a testing strategy for all of your supported database types on a regular basis regardless of the type of coding changes.

Low Risk, High Gain

The easiest performance gains are those that don't involve code changes, so deployment startup options and hardware configurations should be your main focus. Query analyses tools and the Progress Profiler will help with determining where the major bottlenecks are.

Coding changes that are high gain and low risk are the use of Query field lists and changing any FIND FIRST statements to FOR FIRST. These will not only speed up record access for the DataServer but for the Progress RDBMS as well. Appendix B details the optimization techniques available to you.

Both these deployment and development suggestions will improve application performance for both Progress and non-Progress DBMSs, making your code database-independent.

Higher Risk, Higher Gain

Technologies like DBMS stored procedures, can greatly improve query performance, but implementing stored procedures is not without risk. First, you must learn the SQL necessary to write stored procedures and then morph the technology into your application, once this is done your database-independent source code may be compromised. These code changes will improve performance, but will also make your source code database-dependent
To reduce the database-dependent code it is recommended that you use Progress preprocessors or IF THEN constructs.

Programming issues fall primarily into the area of the Progress 4GL. Depending on your environment, you might also be using other languages. For example, you might use Transact SQL or PL/SQL in areas such as stored procedures or when submitting a query directly to SQL Server, Oracle or ODBC DBMSs using the SEND-SQL-STATEMENT syntax from the Progress client. If you are familiar with the code of the application you are migrating, you can make a rough estimate of the number of changes by reviewing the specific coding issues in Chapter 2, "Programming Considerations" of the *Progress DataServer Guides* and by reviewing the techniques shown in Appendix B of this document.

Optimize the DataServer

The main area for optimization is the application and the target DBMS: the DataServer layer is influenced by both. The only specific optimization that can be done is in the way the DataServer is deployed. For a complete list of deployment tips please read the 'Optimization - The DataServer' topics in Appendix B.

Optimize the DBMS

When it comes to performance tuning and scalability issues, the target DBMS plays a large part in the equation. Tuning a Progress 4GL application running against a Progress database generally starts with tuning the database for the particular hardware configuration and then tuning the application in those areas that need an extra boost. Tuning applications using the Progress DataServer technology is exactly the same but you need to remember that the 4GL is being converted to SQL, so the rules change a bit. Appendix B gives programming and design tips in making the 4GL create more efficient SQL statements. When it comes to DBMS sizing for data and scalability you must ensure you have the necessary expertise for the target DBMS, because at the end of the day, Progress is just another ODBC or SQLNet client connecting to your database.

Test and Analyze Functionality

After any coding changes your application needs to be fully tested against the DataServer technology and the Progress database to ensure that the original functionality has not been compromised, (sort order of Browsers etc). The functionality differences may arise when the unsupported 4GL features that the DataServer cannot emulate are compensated for. Other areas to concentrate on are related to data-retrieval error handling because the target DBMS may not behave exactly the same as Progress'. For instance The CREATE statement under Progress immediately creates a record buffer which can be queried be for the end of the transaction boundary that the CREATE is associated with. Non-Progress DBMSs typically don't allow access to the record buffer until after the transaction boundary (Appendix A, "Record Scoping / Availability").

Test and Analyze Performance

The translation technology of the DataServer tends to emphasize 'bad' coding that runs adequately against the Progress database, so when looking at performance it is essential to know where you are today so you know whether any changes have made improvements.
The DataServer can be seen as any other SQL client to the target DBMS that needs to be tuned and scaled according to the supplier's recommendations. 4GL applications can vary greatly in performance and scalability depending on the coding practices used. Simple 4GL debugging statements can be used to capture the execution profile of a known query or process or the Progress Profiler shareware can be used to capture the whole applications profile . Once you have a known benchmark to work against, it is easy to determine the effect of any code changes. Be careful in how these benchmarks are set up, as record caching can lead to surprising results.

Appendix A

Eliminate Unsupported 4GL and DBMS features

Most application migration issues are due to the target DBMS not being able to handle the Progress 4GL functionality. This appendix highlights these differences. The first section gives a three-step guide to the main areas that need to be addressed.

Three Steps for Success

Step 1 for Implementing the DataServer Technology

You need to establish if your existing code violates the few Progress concepts that are not recognized by non-Progress databases.

- If your application relies on the Progress SHARE-LOCK and the related weak/strong block scooping rules, re-coding is needed. Read "Locking Progress vs. DataServers & Records" in this Appendix.
- If your code relies on a record being able to reread within the same transaction block as the CREATE statement that made it, re-coding is needed. Read "Record Scoping / Availability" in this Appendix.
- If your application leverages Progress arrays and the RECID feature, read the "Database Design Considerations" section in the *Progress DataServer Guides*. This chapter is also important if your target database is also accessed and updated by non-Progress clients (see the sections on NULLs, trailing blanks and case-sensitive searching).

Fully test your application against a known record set and in multi-user mode, the reason being that record sort order due to index selection and record locking may not be quite what you would expect.

Step 2 for Tuning the DataServer Technology

Step 1 allows your Progress 4GL application to successfully run against a database supported by a Progress DataServer. The easiest re-coding gains can be achieved by using Progress FIELD LISTS and substituting the Progress FOR FIRST command instead of FIND FIRST. For noncoding performance gains, the –Dsrv skip-schema-check startup option is recommended for deployment situations. There are also a number of record buffer/query options that can be configured at run-time connect. These are described in greater detail in the relevant DataServer manuals. Performance gain may vary depending on application record requirements (average length, bulk updates). For intensive batch record processing it is recommended that you use DBMS stored procedures.

Step 3 for Final Adoption of the DataServer Technology

Retest your application to confirm expected behavior.

Unique Progress RDBMS Features

The following topics highlight the main differences between Progress and foreign DBMSs. The most noticeable conflicts are around record access and retrieval.

Word Indexes

Most target DBMSs don't support word indexing natively and the DataServer is unable to mimic it. If your application uses the 4GL CONTAINS phrase you will receive a compilation error. One possible workaround for this problem involves keeping the column to be word indexed in a separate Progress database with links to the originating record in the data manager. Word indexing is a Progress-specific feature. At the very least, any code that uses the CONTAINS clause in a query must be removed or eliminated from the code at compile time using preprocessor statements.

SQL Server has a feature that is similar to word indexing called Full Text Search. Though these features are similar, they are not equivalent. Full Text Search indexes are not updated automatically. Administrative functions must be implemented to update Full Text indexes. Due to the performance of the update, there will always be a period of time between when the database change is made and when it is reflected in the Full Text index. Using Full Text Search in place of word indexing requires a considerable amount of design and coding. Support for Full Text Search is not currently part of the DataServer for MS SQL Server product. If you would like help in designing and/or implementing this type of feature, contact Progress Global Professional Services.

Record Locking: Progress vs. the Rest of the World

For simplicity, Progress Software Corporation recommends that you avoid using the SHARE-LOCK if possible. If you use SHARE-LOCK in your code unnecessarily, you should remove it.

If a lock type is not specified, Progress often defaults to a SHARE-LOCK. For this reason, Progress Software Corporation recommends that each query should always explicitly specify a lock. For example specify either an EXCLUSIVE-LOCK or a NO-LOCK on each database query statement. Specifying a specific lock removes ambiguity so that it is clear to anyone that needs to work with the code which lock is being selected.

Also, you should review your code to see if it relies on the automatic downgrading of EXCLUSIVE-LOCK to SHARE-LOCK. If you find any sections that rely on this functionality, you might need to remove the SHARE-LOCK dependency.

Exclusive Lock

The Progress exclusive lock and DataServer exclusive lock are virtually the same thing. There can be subtle data-source dependencies that evoke slightly different behavior and/or possibilities for the exclusive lock condition. For instance, Microsoft SQL Server places an intent-to-update lock on a record before an exclusive lock is allowed. Multiple users can have intent-to-update locks but exclusive locks are applied serially and singularly. These subtleties may affect wait times, record contention, and the probability of a dead lock condition in unique ways even though the ultimate aim of the exclusive lock operation amongst all data sources is effectively the same. Oracle releases the lock at commit/rollback time.

Share Lock

The Progress share lock is somewhat unique from most DBMSs in that if multiple users have a share lock on a specific record at the same time, no user session is allowed to upgrade that lock to exclusive. This prevents all the share lock users from reading dirty

data. Most DBMSs have various flavors of share lock which may be used in tandem with selected isolation levels. The more typical scenario is that share locks held by multiple users would allow one user at a time to upgrade their lock to exclusive. In most cases, the level of isolation and granularity of share locks is configurable within the DBMS. Progress DataServers intentionally leave such control in the hands of the DBMS and therefore do no try to directly emulate Progress SHARE-LOCK behavior.

No Lock

The Progress NO-LOCK is just that – no lock. A NO-LOCK places absolutely no guarantee of integrity upon the records from which data is retrieved. Therefore, dirty reads, phantom records, and non-repeatable reads are all quite possible. This behavior is generally understood among the Progress community at large. For most DataServer applications, a "Progress no-lock" condition can be emulated or configured in some fashion. In Oracle, there is no exact equivalent because uncommitted data cannot be read by other users. However, in so far as Oracle does not place any lock per se on the data when NO-LOCK is specified, it performs the same as Progress. For many data sources, if a record is requested without any lock information, some form of share lock will take place. The equivalent form of a Progress NO-LOCK is typically achieved either by setting a level of isolation that allows for it or by explicitly setting some kind of NO-LOCK condition within a specific data-retrieval request sent to the database engine.

The following 4GL code illustrates an example of optimistic locking performed manually within an application.

```
DEFINE TEMP-TABLE changecust LIKE customer.
FIND FIRST customer /* add WHERE criteria here */.
    IF AVAILABLE customer
    THEN DO:
        /* copy the info to a work area */
        BUFFER-COPY customer TO changecust NO-ERROR.
        /* All update processing should occur to changecust, not
customer.
         Then, at commit time, re-read the record. */
        FIND CURRENT customer EXCLUSIVE-LOCK.
        /* Compare to the customer value before any changes were
made. */
            IF CURRENT-CHANGED customer
            THEN DO:
                MESSAGE "Record changed by another user.".
                UNDO, RETRY.
            END.
    ELSE BUFFER-COPY changecust TO customer.
    END.
```

Oracle Locking

Progress SHARE lock is not supported. Oracle does not support SHARE-LOCK capability. Progress-equivalent EXCLUSIVE-LOCK and NO-LOCK are supported.

Progress NO-LOCK and Oracle NO-LOCK do behave differently. A Progress NO-LOCK condition will read uncommitted changes from other users. So dirty reads, phantom records and non-repeatable reads are all possible. In Oracle, uncommitted changes cannot be read. If a user selects a particular record without a lock and other users have locked and committed changes after that select took place, the database engine will appear to rollback all changes for the select. If the user who performed the select subsequently attempts to fetch those records, the roll back will

provide a view of the data that is consistent with the original select. This NO-LOCK behavior of Oracle is a feature and is non-configurable. NO-LOCK can be specified at the statement level and is passed through from the Progress 4GL to Oracle.

If you are using SHARE-LOCK within your application, Progress cannot guarantee that a record in your Oracle database is actually locked at all. You should implement optimistic locking techniques to reduce data concurrency problems. Specify NO-LOCK on record reads wherever possible, as this gives the most scope for internal optimization and allows you to take advantage of field lists in Version 8 and above of the Progress DataServer for Oracle. At time of record update, reread the record with an EXCLUSIVE-LOCK, and if the two record values are the same, you can proceed to update the record. If you need to use an EXCLUSIVE-LOCK, you should hold the lock for as short a time as possible to reduce locking contention.

Automatic Optimistic Locking Support

The Progress Oracle DataServer does not perform optimistic locking by default. Progress Version 8 of the Progress Oracle DataServer introduced a DataServer startup parameter (-Dsrv optimistic) that forces the DataServer to use optimistic locking as the default when modifying records.

Microsoft SQL Server & ODBC Locking

Progress SHARE-LOCK is not supported. SHARE-LOCK dependencies are database specific. Progress EXCLUSIVE-LOCK and NO-LOCK are supported. You can achieve Progress compatible results when you use the DataServer startup parameter –Dsrv TXN_ISOLATION,1 (ODBC DBMSs only). This causes SQL Server to run with an isolation level of Read Uncommitted. Read Uncommitted is the closest match to the NO-LOCK/EXCLUSIVE-LOCK lock types that the Progress database uses. However, SHARE-LOCK is treated like NO-LOCK at this isolation level.

If you depend on SHARE-LOCK in some parts of your code, you must program your software to achieve a similar type of locking in the SQL Server DataServer environment. To achieve this, you can set your transaction isolation level in accordance with your desired share lock behavior as defined by ODBC and its implementation on Microsoft SQL Server. The unfortunate side effect of changing the isolation level from read-uncommitted is that a Progress NO-LOCK will now behave in conformance with the share lock conformance of your selected isolation level.
The DataServer for MS SQL Server can support the NO-LOCK condition in Progress by virtue of the Read Uncommitted isolation level. This can be set using the –Dsrv connection parameter as follows: "-Dsrv TXN_ISOLATION,1". Unfortunately, this is a connection-level switch only. Therefore if you set read-uncommitted for your session, all non-EXCLUSIVE-LOCK statements (statements w/SHARE-LOCK, NO-LOCK, or unspecified statements) will perform as NO-LOCK. This is the wild, wild west of transaction management and cannot currently be obtained on a statement level as it can in Progress and other DataServers.

The default isolation level for most ODBC data sources, including MS SQL Server is read-committed. This isolation level typically puts some form of database-dependent share lock on records, pages, and/or tables that it reads. Under this isolation level, only committed changes can be read from existing data. Therefore the possibility of dirty reads is removed although phantom records and non-repeatable reads could still occur. Setting isolation one level higher to repeatable read ensures that the integrity of data that is read from a record is preserved from one attempted read to the next, so the possibility of non-repeatable reads is removed . This is comparable behavior to the scenario of multiple Progress users all having SHARE-LOCKS on their data. Again, it is important to note that this capability in ODBC and SQL Server is configurable as a session option and cannot be regulated at the statement level. The most restrictive isolation level in ODBC is called serializable. In this session configuration, the possibility of dirty reads, non-repeatable reads and phantom records are all removed but record contention probability is very high. In MS SQL Server, repeatable read and serializable isolation levels operate the same – both

of which provide protection against phantom records. Other DBMSs may support one, some or all of the predefined ODBC-defined isolations and each may introduce subtle differences in NO-LOCK and SHARE-LOCK behavior.

Below is a table describing the expected locking behavior for the current pessimistic locking model without the availability of NO-LOCK at the statement level.

Isolation Level	Lock Type	Locking Behavior for Progress
Read Uncommitted		
	NO-LOCK	Progress NO-LOCK behavior
	Unspecified	Progress NO-LOCK behavior
	SHARE-LOCK	Progress NO-LOCK behavior
	EXCL-LOCK	Progress EXCL-LOCK behavior
Read Committed		
	NO-LOCK	MSSQL share lock behavior
	Unspecified	MSSQL share lock behavior
	SHARE-LOCK	MSSQL share lock behavior
	EXCL-LOCK	Progress EXCL-LOCK behavior
Repeatable Read		
	NO-LOCK	MSSQL share lock behavior
	Unspecified	MSSQL share lock behavior
	SHARE-LOCK	MSSQL share lock behavior
	EXCL-LOCK	Progress EXCL-LOCK behavior
Serializable		Same as Repeatable Read

Below is a table describing the expected locking behavior for the current pessimistic locking model WITH the availability of NO-LOCK at the statement level. An X marked in the following columns means: DR=allows dirty reads, NRR=allows non-repeatable reads, PH=allows phantom records.

Isolation Level	Lock Type	Locking Behavior for Progress	DR	NRR	PH
Read Uncommitted					
	NO-LOCK	Progress NO-LOCK behavior	X	X	X
	Unspecified	Progress NO-LOCK behavior	X	X	X
	SHARE-LOCK	Progress NO-LOCK behavior	X	X	X
	EXCL-LOCK	Progress EXCL-LOCK behavior	NA	NA	NA
Read Committed					
	NO-LOCK	Progress NO-LOCK behavior	X	X	X
	Unspecified	MSSQL share lock behavior		X	X
	SHARE-LOCK	MSSQL share lock behavior		X	X
	EXCL-LOCK	Progress EXCL-LOCK behavior	NA	NA	NA

Isolation Level	Lock Type	Locking Behavior for Progress	DR	NRR	PH
Repeatable Read					
	NO-LOCK	Progress NO-LOCK behavior	X	X	X
	Unspecified	MSSQL share lock behavior			
	SHARE-LOCK	MSSQL share lock behavior			
	EXCL-LOCK	Progress EXCL-LOCK behavior	NA	NA	NA
Serializable		Same as Repeatable Read			

Record Scoping / Availability

When using a Progress RDBMS a new record becomes available to users as soon as it is supplied with index information. This differs from the many other DBMSs, where a record is not written to the database (and therefore not available to users) until the end of the record scope. This difference can be worked around by forcing the record to be written to disk as soon as the index information is supplied. This can be done by using either the RELEASE or the VALIDATE statement. To illustrate this behavior, look at the following Progress 4GL code:

```
DEFINE BUFFER xcust FOR cust.
CREATE cust.
cust-num = 111.
FIND xcust WHERE xcust.cust-num = 111.
DISPLAY xcust.
```

When the previous example is run against a Progress database, customer number 111 will be displayed. The DataServer, however, will return an error, as the new record does not get written back to the database until the end of the record scope (in this case after the DISPLAY xcust statement). This problem can be corrected in the following manner:

```
DEFINE BUFFER xcust FOR customer.
CREATE customer.
cust-num = 111.
VALIDATE customer. /* or RELEASE or GET RECID/ROWID.  */
FIND xcust WHERE xcust.cust-num = 111.
DISPLAY xcust.
```

Note. The constraint rules that fire by default when the record is normally committed still apply when using the VALIDATE or REALEASE commands.

If you are using the NO-ERROR option and the SESSION-STATUS:ERROR function to do your own error handling, you should use the VALIDATE statement after you have created or updated the record. This will enable you to trap and handle errors returned from the underlying DBMS. To execute on a Progress database, the previous code might be written as follows:

```
DEFINE BUFFER xcust FOR customer.
CREATE customer.
ASSIGN cust-num = 111 NO-ERROR.
        IF ERROR-STATUS:ERROR THEN DO:
                MESSAGE "Error Creating Record".
        END.
        ELSE DO:
                FIND xcust WHERE xcust.cust-num = 111.
                DISPLAY xcust.
        END.
```

If a record with a cust-num of 111 already exists in the database, the Progress database will return the "Error Creating Record" message, otherwise the record is created as per requested, and displayed from the xcust buffer. However, when using the Progress DataServer for Oracle, the customer record will not be visible in the xcust buffer, as the record will not be written back to the Oracle database until the end of this procedure. In order to emulate the behavior expected of a Progress database, the procedure can be modified as follows to run against Oracle:

```
DEFINE BUFFER xcust FOR customer.
CREATE customer.
ASSIGN customer.cust-num = 111 NO-ERROR.
VALIDATE customer NO-ERROR.
        IF ERROR-STATUS:ERROR THEN DO:
                MESSAGE "Error Creating Record".
        END.
        ELSE DO:
                FIND xcust WHERE xcust.cust-num = 111.
                DISPLAY xcust.
        END.
```

The VALIDATE statement forces the record to be written to the DBMS immediately. The NO-ERROR option ensures that user-defined error processing is invoked. The RELEASE customer NO-ERROR statement can also be used here, but it has the side effect of deleting the record from the client buffers. The application would have to refetch the record from the database if it needs to access it again.

Transaction Boundaries

The Progress database has a unique feature in its ability to hold a record lock beyond a transaction boundary. This is in contrast to standard SQL-based data sources, like MS SQL Server, which release all current locks on a given connection at the end of a transaction. This difference in behavior has repercussions with the use of record buffers in your 4GL as illustrated with the following example. The three transactions below are executed in sequence. The first and third are marked (internal) meaning they are executed from this session. The second is marked (external) meaning a separate session issues this transaction sequentially after transaction 1.

```
DEFINE BUFFER b1_state FOR state.
DEFINE BUFFER b2_state FOR state.

/* Transaction 1 (internal) */
DO TRANSACTION:
FIND b1_state WHERE b1_state.State = "CA" EXCLUSIVE-LOCK.
        DISPLAY b1_state.State-Name.
END.

/* Transaction 2 (external) */
PAUSE MESSAGE "The state name is California – Now somebody in
another        session updates the state name associated with "CA" to
"Carolina" and commits the new name change."

/* Transaction 3 (internal) */
DO TRANSACTION:
FIND b2_state WHERE b2_state.State = "CA" EXCLUSIVE-LOCK.
        DISPLAY b1_state.State_Name b2_state.State_Name.
END.
```

The value displayed for State_Name in transaction 3 is “California” in both buffers even though the name was externally changed to “Carolina” in transaction 2. This is because Progress down graded to SHARE-LOCK and did keep the record: the lock on the state record was held between transaction 1 and transaction 3 and therefore the buffer information in b1_state was shareable with buffer b2_state. Since SQL-based transaction scoping does not allow for lock retention beyond the transaction boundary, the second buffer, b2_state, is actually stale at the time of transaction 3. To prevent the use of stale buffers, they should be explicitly released at the end of a transaction using the Progress RELEASE syntax. After transaction 1 was completed, the following syntax should have followed:

```
RELEASE b1_state.
```

The non-Progress DBMS handles transaction rollback and recovery through its own internal mechanisms, however, Progress 4GL transaction scoping rules still apply. In Progress, transactions end at the end of the outermost block where an update takes place. When a transaction that updates a non-Progress DBMS ends successfully, the Progress DataServer sends a COMMIT message to the target DBMS. If the transaction is interrupted, Progress sends a ROLLBACK message to the target DBMS. If you modify data in more than one database in a single transaction (for example, a Progress database and an Oracle database), Progress uses a two-phase commit protocol to minimize the chance of database corruption wherever possible.

Appendix B

Optimization – the Application, the DataServer, the DBMS

This appendix covers details on optimizing the three DataServer components: the application, the DataServer, and the target DBMS.

The Application

There are two main elements to look into when adopting your application to use the Progress DataServer technology, which are performance and behavior.

Expected Behavior of the Application

Appendix A covers the mandatory steps in ensuring application behavior. This Appendix takes those recommendations one step further by describing techniques that can be employed to ensure your Progress code is database-independent.

Constraint Violations / Trigger Execution

The non-Progress DBMS will ensure that any target database constraints and database triggers will not be executed until the record is written to the database. If this isn't handled before the transaction end (i.e. using either the VALIDATE or RELEASE Progress 4GL statements), the user may not be able to gain control of the user interface. If a user attempts to insert or update a column in a unique index, other users attempting to perform a similar update may have to wait until the first user either commits the transaction (in which case, the second user will receive a constraint violation) or rolls back the changes (in which case the second user will proceed). This is native DBMS behavior and cannot be circumvented. If applications outside of the Progress environment are accessing the same foreign tables as a Progress-based application through the Progress DataServer, then these native applications will only be affected by the non-Progress database triggers. If implemented, Progress client triggers will fire in addition to these database triggers only for Progress-based applications. If there are both Progress client triggers and target database triggers, the Progress client triggers will fire before the target database triggers. You can restrict Progress session and database triggers from executing with the DBTYPE function. (The triggers must be defined as Overridable, this can be done via the Progress Data Dictionary.) For example, the following code would only fire a trigger when running on a Progress database:

```
IF DBTYPE(dbname) = "PROGRESS" THEN DO:
        {triggers.i}
END.
```

Error Messaging

Many non-Progress DBMS error messages do not provide clear descriptions of the problems being encountered. If your application will run against both Progress and other DBMSs, you may want to consider handling error messages through Progress to provide consistent behavior across all databases. Error handling is fully covered in the DataServer manuals.

Records

Progress creates a record at a different time in the sequence of events in the Progress database than does a non-Progress database. Specifically, Progress creates records after all of the key fields are assigned, or when a RECID/ROWID is assigned to a variable or at the end of the record scope. The DataServer creates records in the target DBMS at the end of the record scope.
The following techniques can be used when creating database records to avoid unexpected results.

Word Indexes

Due to technical differences between Progress and non-Progress databases, word indexes are not supported. If your application uses the CONTAINS 4GL phrase against a DataServer schema you will receive a compilation error. One possible workaround for this problem involves keeping the column to be word indexed in a separate Progress database with links to the originating record in the DBMS.

Word indexing is a Progress-specific feature. At the very least, any code that uses the CONTAINS clause in a query must be removed or eliminated from the code at compile time using preprocessor statements.

SQL Server has a feature that is similar to Word Indexing called Full Text Search. Though these features are similar, they are not equivalent. Full Text Search indexes are not updated automatically. Administrative functions must be implemented to update Full Text indexes. Due to the performance of the update, there will always be a period of time between when the database change is made and when it is reflected in the Full Text index. Using Full Text Search in place of word indexing requires a considerable amount of design and coding. Support for Full Text Search is not currently part of the DataServer for MS SQL Server product. If you would like help in designing and/or implementing this type of feature, contact Progress Global Professional Services.

Descending Indexes

Note that Microsoft SQL Server and Oracle versions prior to 8i do not support descending indexes.

If you are migrating a Progress database that has descending indexes, you should choose the ProToMss/ProToOracle option to add the descending indexes to the database. When the Progress descending index is selected, the DataServer will use ORDER BY with the DESCENDING keyword when generating the SQL statements.

Selecting the option to add the descending indexes provides an alternative to changing all of the queries that specify a descending index with USE-INDEX or that select a descending index indirectly. This option is selected in the Data Administration tool using the “Create Desc Index” toggle on the screen available from the “Progress DB to MS SQL Server” or Progress DB to Oracle” menu options. These options retain descending index definitions in the schema holder while creating ascending indexes in the target database.

If you do not used the Progress DataServer migration tools to build your target database with the create descending indexes selected, you may need to review the following sections for changes that might need to be done to your code.

Assign Unique Index Fields on Create

When possible, assign all index key fields right after the record is created; more specifically, right after the Create statement. Check the definitions in the Data Dictionary to make sure that all of the fields in each unique are being assigned. In addition to the unique index, other constraints, such as NULL columns and foreign key columns need to be satisfied.

Setting ROWID/RECID

Using the RECID or the ROWID function causes Progress to write the record to the database. Do not reference the RECID/ROWID until all of the target DBMS constraints are set.

RECIDs and ROWIDs

The Progress DataServer will support RECID functionality in the target database through the use of an additional unique indexed integer column on the table (often called PROGRESS_RECID). These additional columns also require indexes and sequences to populate them. A far better strategy for Progress-based applications (Version 8 and above), is to use the ROWID function. The ROWID function performs in a more consistent manner across all databases, as well as eliminating the need for a PROGRESS_RECID column (although it will use it if one is available). Note, however, that the ROWID function causes a newly created record to be written to the target database earlier than it would to a Progress database. If you do not assign values to all fields that are defined as mandatory for a record, the ROWID function will fail.

By default, the DataServer designates a column to support the ROWID function. It evaluates the indexes available for a table and selects one in the following order:

1. PROGRESS_RECID column. If a PROGRESS_RECID column is selected, then both the RECID and ROWID Progress 4GL functions can be used in a Progress 4GL program.

2. Unique index on a single, mandatory, NUMBER column with precision <10 or undefined and scale 0. If a unique indexed column is selected, then both the RECID and ROWID Progress 4GL functions can be used in a Progress 4GL program.

3. Native ROWID. If a native ROWID column is selected, then only the ROWID Progress 4GL function can be used in a Progress 4GL program (the RECID function will not work). The native ROWID typically provides the fastest access to a record. However, the native ROWID does not support the Progress FIND PREV/LAST statement or find cursor repositioning.

RELEASE and VALIDATE Statements

The RELEASE and VALIDATE statements cause Progress to write the record to the database. Do not use the RELEASE and VALIDATE statements until your code has set all of the unique key fields and other column constraints. You can use the RELEASE and VALIDATE statements to force Progress to write a record to the database so that it is available if the record needs to be reread prior to the end of a transaction. Remember, if you use RELEASE, the record is no longer available in the record buffer.

Application Performance

Unfortunately, there is no single solution for performance optimization. In actuality, optimization is the sum of all of the other topics discussed in this document. However, there are some areas that have the potential for large gains in performance. For most environments, the area that yields the greatest performance benefits is through programming techniques. That is not to say that the other areas are not important. You should make sure that your hardware and software configurations are set up properly. Once this is done, the majority of your time for performance optimization should be spent on programming issues.

The search for performance improvement begins by understanding all of the software and hardware components involved in the DataServer environment. This means learning about the DataServer, the Progress 4GL, and target DBMS. The following section lists a few suggestions on potential performance improvements not covered elsewhere in this document.

Reduce Network Traffic and the Read / Write Access to the Database

Almost all performance issues can be traced to reading and writing records to the database. The bottleneck in reading and writing these records is the network, the DataServer, or the database engine itself. It may be obvious to many, but it is worth documenting this important point. Anytime you reduce the number of reads and writes to the database you will increase the performance of the application.

There are two ways reduce reads and writes to the database. First, find unneeded queries and updates of database records. Second, improve how the reads and writes take place.

Unneeded queries and updates of database records — To solve this problem, you must review the types of queries and the available indexes. Some types of queries read many records and then discard all but a few. Modifying the query or adding an index can reduce the number of records that need to be found and read by the database and the number of records that must be sent over the network. Temporary tables provide another method to reduce the network traffic and database reads.

Improve how the reads and writes take place— The 4GL provides options such as field lists. Field lists can be used to return only the needed fields. Using field lists reduces the size of the record that Progress returns from the server. The reduced record size allows more records to be sent over the network in a single packet.

Action Segment Overflow

Progress 4GL code compiled against a DataServer uses more space in the action segment than code that is compiled against a Progress database. There might be action segment errors when compiling existing programs against a DataServer that compile successfully against a Progress database if the programs are near the allowed maximum action segment size.

To resolve this problem, the code must be modified to reduce the size of the action segment. One way to solve this is to move code from the main block into internal procedures. See the Progress Knowledge Base for more information on resolving action segment issues.

Progress 9.1C introduces a new Action Segment size that is four times larger than previous Progress versions.

Cursors

The Progress DataServer caches results sets from the target database to enhance performance. It caches as much data as fits in its allocated cache size. Depending on what kind of cursor (standard or lookahead) a query is using the DataServer caches row identifiers or entire records.

Standard Cursors

The DataServer caches row identifiers for the results set. When a record needs to be read from the results set, its row identifier is retrieved from the cache and a new database query is generated to access the entire record.

Lookahead Cursors

The DataServer caches complete records (or partial records as specified by a field list). Lookahead cursors fetch as many records as can fit in the allocated cache, limiting the number of database accesses, and thereby improving performance. When a record is required, its values are simply retrieved from the cache. This can lead to data consistency problems if a record's value in the database was changed by another user after a lookahead cursor was used to read its value.

Using lookahead cursors results in behavior that is different from Progress because they do not see any changes made to the records in the cache. Specify QUERY-TUNING NO-LOOKAHEAD for behavior that is consistent with Progress (but be aware that performance may suffer as a consequence).

Cursor Repositioning

Progress uses cursors to keep track of where it is within a table. A cursor is like a pointer to consecutive records in a table. For example, Progress uses cursors when it processes statements that return a set of records (such as the FOR EACH or OPEN QUERY statements). Progress maintains cursor positioning across queries. The Progress DataServer supports this behavior for tables that have a unique index on a mandatory integer column or tables that contain the PROGRESS_RECID column. When a FIND FIRST/LAST statement is issued, the DataServer builds a result set that might include every record in a table. Performance will improve if the statement is qualified with a WHERE clause to reduce the size of the result set (however this still will not perform as well as a FOR EACH or OPEN QUERY statement).

Cursor repositioning behaves with the Progress DataServer as it does with Progress databases, except when a target database fails to find a record. In this case, the cursor in a Progress database is located after the last record that was read. In the target database, a failed search does not affect the cursor if a single index is used. For example, assume a result is made up of customer records 1, 2, 3, 4 and 5, and the cursor is positioned at the last record (i.e. customer number 5). If a FIND NEXT customer statement is issued (which uses a single index), the cursor moves as follows:

Progress Database: Target Database:

Regardless of which data source is used, the Progress 4GL functionality behaves consistently when this occurs. The following statement, for example:

```
IF AVAILABLE customer
THEN DISPLAY customer.
ELSE MESSAGE "No customer record found".
```

will display the message “No customer record found” in each of the above situations, regardless of where the cursor is positioned.

Note: However, a failed FIND may reuse an existing cursor which then may invalidate the context of other cursors that assumed the FIND to succeed.

Influencing Query/Browse Performance

The DEFINE BROWSE statement relies on a unique record identifier for forward and backward scrolling, hence a valid ROWID or RECID should be made available for that table.

Progress Version 7 The largest area of concern surrounding performance involves querying large sets of data. A GET LAST QUERY statement will cause a complete set of RECIDs to be gathered before repositioning the query at the last record. (Note: this is no different than what happens with a Progress database today.) Depending on the situation, it can be faster to reopen the query in reverse order. Another technique would be to obtain the ROWID of the last row by using a FIND statement, and then repositioning the query to that ROWID. Regardless of the technique used, it is important that all browsers be tested thoroughly.

Progress Version 8 Progress Version 8 introduced tremendous performance enhancements specifically designed to address performance-related concerns. These enhancements include field lists and lookahead cursors which are described in more detail in the “Application Design” section of this white paper.

Progress Version 9 Progress Version 9 has improved performance of the DataServer by adding features such as optimistic locking (Oracle only), skip schema check, and INDEXED-REPOSITION support. Each of these are described in more detail below and in the “Application Design” section of this white paper.

RECORD Retrieval

Understanding the target DBMS and how the Progress 4GL is converted into the target query language is very important to ensure efficient and consistent record retrieval.

Field Lists

A feature introduced in the Version 8 Progress DataServer is the option of specifying field lists for a query. This allows the DBMS to only send back to the client those fields of the record that were requested in the query (rather than every column in the record). This can dramatically decrease the amount of network traffic in a client/server environment when records are read. It can also improve performance (to a lesser extent) in host-based processing environments. Field lists will only work with FOR EACH or OPEN QUERY NO-LOCK statements. In order to take advantage of these performance improvements, it is recommended that all FIND statements be converted to equivalent FOR EACH or OPEN QUERY statements wherever possible.

INDEXED-REPOSITION Support

The INDEXED-REPOSITION keyword is supported through the Progress DataServer. This can significantly improve on-line random access. Use this option in the 4GL and in browsers to reposition randomly within the result set of an open query. However, be aware that using this option causes performance degradation since repositioning will otherwise be performed sequentially through the query result set.

The INDEXED-REPOSITION option of the OPEN QUERY statement is only supported by Version 9 (and above) of the Progress DataServer. In order for applications accessing a non Progress database to use this functionality, the query must be defined as SCROLLING, and the table must support the RECID/ROWID function. The INDEXED-REPOSITION phrase is not supported on a view that does not have a selectable ROWID.

Descending Indexes on a FIND Statement

When writing or converting Progress 4GL code to work with the DataServer, you must modify FIND statements that use indexes with descending components. In a Progress database environment, a FIND statement might select an index with descending components either explicitly or implicitly. The explicit case occurs where the FIND statement uses the USE-INDEX clause. The implicit case occurs when a FIND statement selects an index that has descending components using the Progress index selection algorithm at compile time.

The DataServer supports queries against non-Progress databases that request information based on descending fields. You can do this with the Progress 4GL DESCENDING on the ORDER BY clause.

The DataServer technology supports descending sorting through the DESCENDING keyword option in the ORDER BY clause. The FIND statement does not support the BY clause. When using the DataServer, you cannot use the FIND statement for queries that need to be accessed by a sort order that contains descending components. In these cases, you must change the FIND statement to a FOR FIRST or an OPEN QUERY statement, since both of those statements support the ORDER BY clause. In the case of an OPEN QUERY statement, the DataServer uses the GET statement to access the record.

USE-INDEX

As noted in the DataServer documentation, you must specify USE-INDEX or use the BY clause to return records in a predictable order.

Against a Progress database, USE-INDEX controls which index is used. However, for the non-Progress DBMS, you cannot guarantee which index is selected. The index selected in the 4GL will dictate the ORDER BY clause generated for SQL execution. However, ultimately the DBMS's query optimizer engine makes the actual index determination.

The DataServer has not implemented index hints directly through the 4GL. However, SQL that is entered through the send-sql-stmt procedure may contain hints since the syntax is just passed through directly to the DBMS engine. You could also call a stored procedure from your 4GL which contains your SQL hints. Specifying index hints does not ensure that the DBMS will use the specified index. Keep in mind that SQL Server optimizes index selection based

on the BY clause, WHERE bracketing, table statistics, etc. and in most cases will determine through cost-based analysis the best index selection.

Using the BY clause in your 4GL statement translates into an ORDER BY. Using the ORDER BY clause leaves the DBMS engine flexibility to use the provided information in selecting the best indexing method.

Progress Software Corporation recommends you avoid USE-INDEX when possible since it overrides the Progress compile-time index algorithm selection. The primary reason to avoid USE-INDEX where possible is to keep flexibility for future changes. However, there are exceptions to the recommendation to avoiding the use of USE-INDEX. One advantage to USE-INDEX is that it ensures that the BY clause generated for server execution will correspond exactly to an existing index definition so if your index definitions change your code is not left with a BY clause without a supporting index.

For example, based on performance, indexes may be added or deleted. Code that uses the BY clause requires only a recompile to work with the new indexes. However, if USE-INDEX is explicitly specified, you must modify the code to take advantage of new indexing. If an index is added that can provide better performance for the query, the BY clause is flexible since it allows the Progress index algorithm selection to select the new index over the old one. However, code with USE-INDEX would need to be modified in both the case of a dropped index and in the case of a new index that offered improved performance.

Join-By-SQLDB

Another DataServer query-tuning option is the ability to direct the target DBMS to perform a query's join on the server. This is preferable to the default action of sending two result sets down to the client (very costly in a networked environment) and having that client (often a less powerful machine than the server) perform the join. This JOIN-BY-SQLDB option (in either a FOR EACH or OPEN QUERY statement) should perform better than a send-sql-statement stored procedure join for three major reasons:

1. JOIN-BY-SQLDB will do an array fetch.
2. JOIN-BY-SQLDB will ship multiple result rows in less network traffic.
3. JOIN-BY-SQLDB will eliminate redundant information.

For example, consider a customer table with exactly 2 customers, each with 10,000 orders, and the following 4GL code:

```
FOR EACH customer, EACH order OF customer:
. . . .
END.
```

If the DBMS performs the join, the DataServer will receive the first customer row with each of its potential 10,000 order rows, and then the second customer and its potential 10,000 order rows. If this join is performed with a send-sql-statement stored procedure (or Progress/SQL SELECT pass-through), 20,000 result rows will be shipped to the client. Each of the first 10,000 result rows will contain all of the columns from the customer table with the values for the first customer duplicated 10,000 times. This will be followed by a similar arrangement for the second customer and their orders. The DataServer will eliminate the redundant cursor information and data before shipping the results set to the client, but performance is still impacted by the DBMS returning the redundant information to the DataServer.

FOR FIRST in Place of FIND FIRST

Using FOR FIRST in place of FIND FIRST can improve the performance when retrieving a single record that is being accessed NO-LOCK.

For example, if your application uses the following FIND FIRST code:

```
FIND FIRST <table-name> WHERE <where clause> NO-LOCK.
```

The code can be replaced with:

```
FOR FIRST <table-name> WHERE <where clause> NO-LOCK:
END.
```

In the case of the FOR FIRST, Progress makes the record available beyond the end of the FOR FIRST loop. FIND cursor repositioning will not occur if the FOR FIRST command is used instead of FIND FIRST.

Progress 4GL vs. SQL

The Progress DataServer allows you to use different approaches for querying a target database. Your application might be able to take advantage of these approaches depending on the kind of query you are writing and the kind of data you are accessing. These approaches are:

Progress 4GL – The DataServer generates SQL (or PL/SQL or Transact SQL) for DEFINE QUERY and FOR EACH statements, but you can use the QUERY-TUNING option to customize the queries that the DataServer passes to the DBMS.

Progress SQL SELECT – When you use a SQL SELECT statement in a Progress procedure, the DataServer passes the SQL directly to the DBMS. This approach can improve performance, especially when counting records, and allows you to access certain types of data more effectively, such as aggregates.

ODBC SQL or Oracle PL/SQL (SQL*Plus) – If you want to use specialized query syntax supported only by Oracle's PL/SQL or SQL*Plus of SQL Server's Transact SQL, you can use RUN-STORED-PROC send-sql-statement to send the syntax to Oracle. When retrieving a large number of records using a FOR EACH … NO-LOCK or OPEN QUERY statement, best performance is obtained by using the lookahead cursor and field list options. Progress/SQL SELECT statements are best for more complex queries (those with subqueries, GROUP BY or HAVING clauses) and queries that perform aggregate functions. These Progress/SQL SELECT statements are passed directly to the DBMS using Progress' SEND-SQL functionality on the server. In Progress Version 8.2B and higher, SQL SELECT pass-through also makes use of Oracle bind variables so the SQL code generated by the DataServer is reusable, thereby reducing processing time. The SQL that is sent to Oracle using the send-sql-statement stored procedure does not use bind variables.

FOR EACH and OPEN QUERY statements (when used with the NO-LOCK and LOOKAHEAD options) tend to perform better than the send-sql-statement stored procedure, because they use array fetches to minimize the number of accesses to the database. Also, if Progress

networking is used, the DataServer packs multiple rows per network message The Progress send-sql-statement stored procedure and Progress/SQL SELECT pass-through statements use a single row buffer to fetch records into and a single result row per network message. The send-sql-statement stored procedure is also useful for massive updates, provided they do not occur in a sub transaction that needs to be rolled back.

DBMS Stored Procedures

The Progress DataServer supports the use of stored procedures. These stored procedures can be called directly from the Progress 4GL using the RUN STORED-PROC command. By running stored procedures, processing is moved off the Progress client, onto the database server machine. Stored procedures are useful in reducing network traffic when aggregations of large numbers of records need to be performed, or when performing mass modifications to database records. Migrating the complete functionality of a Progress procedure into a stored procedures format may not be practical, but any record related 4GL command that can be offloaded to a stored procedure, such as CREATE, FIND, and DELETE, will generally result in better performance.

DBMS Stored Procedures and Transaction Scoping

When stored procedures are called from a Progress procedure, different transaction scoping rules apply. A database modification made through a stored procedure in a sub transaction will not be rolled back if the main Progress transaction in which it is run is undone. For example, if you are updating records and run a stored procedure in the middle of the transaction, and then issue a ROLLBACK statement, the ROLLBACK will not affect any of the operations executed by the stored procedure as the stored procedure has its own transaction scope within the DBMS.

Native SQL Syntax Support

The Progress DataServer provides a function called “send-sql-statement” that can be used to send native SQL (utilizing either SQL*Plus or PL/SQL, Transact SQL, or other native ODBC SQL syntax) directly to the underlying SQL database. This may be useful in situations where functions or syntax available to the target DBMS would be beneficial to the application, or when application processing is better suited to being executed on the server. This SQL syntax will be ignored by the Progress compiler in order to support the different SQL dialects, and as such, the SQL string may be an arbitrary expression that can only be evaluated at runtime. Therefore, syntax errors will not be detected until application runtime. Please refer to the *Progress DataServer Guides* for more details on using the send-sql-statement stored procedure.

Distributed and Batch Processing

Progress 4GL batch programs or Progress AppServer™ programs have the same restrictions that apply to interactive client Progress 4GL programs noted throughout this document. If any batch programs are to perform mass updates or deletes, you may wish to investigate the use of stored procedures or SEND-SQL-STATEMENT as this will improve performance (due to reduced messaging). If any batch processing is to occur, careful consideration must be

given to the DataServer configuration, as the Progress DataServer would be required to be located on the batch or application server, instead of on the client machine.

Mass Database Modifications

Due to the amount of network traffic that is involved in ensuring compatibility with the Progress DataServer, mass modifications (e.g. repeated inserts, updates, or deletes) of a target database are best suited to being executed from within the DBMS's stored procedure (using the Progress send-sql-statement stored procedure to execute it).

The DataServer

The techniques for optimizing the DataServer layer fall mainly into the deployment area of the 4GL application and the Progress environment.

The Progress Schema Holder

The schema holder contains information about the data definitions of a target database. This information includes a description of the target database's structure, the tables, the fields within the tables, and the indexes. The schema holder contains no user-application data. The Progress DataServer accesses the schema holder only when it compiles procedures and at the beginning of a run-time session when loading data definitions into memory.

As a schema holder is normally run in read-only mode, (concurrent user access without locks) there is no need to start a Progress Database Broker against it. The schema holder may also be able to store Progress objects required by applications. For example, the RBREPORT table accessed by the Progress Report Builder could be stored within the schema holder without requiring a separate database connection. (In this instance, however, the schema holder cannot be connected to in read-only mode.)

For performance reasons, schema holders are best placed locally on the client rather than centrally on the server (although this difference is much less noticeable when using a fast Ethernet connection as opposed to 10BaseT configuration). If a schema holder is desired on a server (generally for maintenance reasons, since there is only a single schema holder database to maintain), it is possible to use the –cache database connection parameter to keep a cached copy of the schema holder on the client's local disk, for faster connection times. Refer to the *Progress Database Administration Guide and Reference* for further details on the –cache parameter.

Effects of Database Changes

If the structure of the underlying target database schema changes in any way (for example, adding a table or view, or modifying the table by adding a new column), the Progress schema holder must be updated to reflect those changes.
Utilities are provided with the Progress DataServers to verify that the schemas for the target database and the schema holder match. If a match is not found , you can use additional utilities to bring them in line.

Code Recompilation Due to Schema Changes

Often, a central database is accessed by more than one application. These applications will typically "share" the data in the database tables, without concern for other applications. Oracle's concurrency techniques ensure that consistency is maintained at the data level, regardless of the application. However, there may be situations where one application requires changes to be made to an existing table structure that is referenced by other applications (for example, adding,

deleting, or renaming columns). Often, in a Progress environment, such data object changes will force recompilation of all Progress code accessing this object. This is due to a different CRC (cyclic redundancy check) value being calculated on that object after the change. This recompilation phase is an inconvenience, especially when such changes were caused by a different application and do not affect the application requiring recompilation. This issue can be avoided by creating a DataServer schema which accesses Oracle views. By creating an Oracle view of the table containing only those columns required by the application, any changes made to the underlying table*, as long as they do not affect the view's definitio*n, will be transparent to the application. As the view is based on a single table, all normal Progress functionality can be performed on this view as though it were the original table. Assume the following customer table definition as an example:

cust_num name credit_limit sales_rep
(integer) (char) (decimal) (char)
1 Lift Line Skiing 5,000.00 BBB
2 Sails Afloat 2,300.00 ARP
. . . .
. . . .

If we create a view (called, for example, v_customer), based on the above customer table and defined as the following:

cust_num name credit_limit sales_rep
(integer) (char) (decimal) (char)

our application can now access the data through this view as though it were the original table (which, essentially, it is). To mask the differences in object naming (the table's name is customer, and the view, v_customer), we can change the name of the view in the Progress schema holder to customer. The mapping to the view remains the same, and our application will not need to be changed. What happens when a third-party application requires a change to the structure of our base customer table? The answer would depend on the types of changes occurring. The following chart specifies the types of changes possible on a table structure, and whether or not the application needs recompilation under those circumstances.

Change Description	**Referenced in View?**	**Recompilation Needed?**
Adding a column	No	No
Dropping a column	No	No
Dropping a column	Yes	Yes
Adding an index	No	No
Adding an index	Yes	No
Dropping an index	No	No
Dropping an index	Yes	No

Creating a view does not affect indexing and query resolution. Indexes behave just as they would if you were accessing the table directly. Please note that if a column is added to the underlying table (and this column is not reflected in the view), and you attempt to undo the delete of a row in this view from within a sub transaction, the undo will fail. The Progress DataServer will generate an error and rollback the entire transaction.

Startup Parameters

You should be aware of, and test your applications with, the DataServer (-Dsrv) connection parameter and its various options. Many DataServer startup and hint switches are DBMS-specific, so please read the *DataServer Guide* for your particular DBMS. These are some of the available options:

qt_bind_where / qt_no_bind_where (Oracle DataServer Only)
Specifies whether the DataServer uses Oracle bind variables for values in WHERE clauses.

qt_cache_size
Use the parameter –Dsrv qt_cache_size,x to set the cache size by default for data returned from queries submitted with the DataServer. When you use this parameter, x represents the number of bytes. The following is an example of how the parameter is used: –Dsrv qt_cache_size,30000. Generally speaking, if your tables have large record sizes and/or your queries generally contain a large result set, it may make sense to increment the cache size. However, use caution in setting this value since setting it too high can also cause degradation in performance while caching takes place. To determine the optimal value, test the settings in the specific application environment where you plan to set and use the parameter.

qt_debug / qt_no_debug
Specifies whether the DataServer should print debugging information that it generates for the query to the dataserv.lg file.

qt_lookahead / qt_no_lookahead
Specifies whether the DataServer uses lookahead or standard cursors.

Optimistic (Oracle DataServer only)
Specifies whether the DataServer is to use optimistic locking techniques as the default when locking records.

skip-schema-check
Use the parameter –Dsrv skip_schema_check to skip the check of the schema and thus save some processing time when you first access a table. Because this option bypasses the client and server schema handshake when a table is opened, you must ensure that no schema mismatches exist. To prevent exposure to mismatches that may potentially corrupt data, you should perform a final schema pull just prior to deployment with this option turned off. You can confirm if this option is implemented in your environment by reviewing the dataserv.lg file.

Client Connection Parameters

Consider setting the following connection parameters when configuring your DataServer environment. They are not unique to the DataServer environment, but they do play a role in the performance. Though following list of connection parameters does not provide a complete definition of these parameters, you can find a detailed definition of these parameters in the Progress documentation set.

Message Buffer Size (–Mm) — For most environments it is best to set the message buffer size as large as possible. Setting this parameter to a larger value enables more records to be sent across the network in one packet. For example, try a setting of 2048.

Maximum Memory (–mmax) — Set this value to allocate memory on the client. For example, try a setting of 4096.

Quick Request (–q) — Add this parameter to avoid searching the PROPATH after the first time Progress finds a program. This improves performance, but it should only be used in a production environment where the code is not being modified.

Read Only (–RO) — Use this parameter to set Progress schema holder access to read only. In most application environments, you can access the schema holder as read only. Since setting this parameter stops the logging of most of the before-image activity, setting the access to read only can provide a slight improvement in performance. To avoid a startup warning message when using –RO, you must truncate the before-image file.

To truncate a before-image file enter the following command from a system prompt: proutil <database name> –C truncate bi. For more information on using the proutil command, see the *Progress Database Administration Guide and Reference*.

When first using the proutil command, a common problem occurs stating that the command is not found in the DOS environment. You can fix this by adding the Progress bin directory to the system path. For example, if you installed Progress in the default directory on the C drive, then the following would be added to the path 'C:\program files\progress\bin'.

Temporary Directory (–T) — This parameter is used to specify where temporary files are stored. Set the temporary directory to use a local drive for temporary files. Otherwise, if you store the temporary files on the network there might be major performance degradation for the individual user and for the networked system as a whole.

Speed Sort (–TB) and Merge Number (–TM) — Maximize these values to provide additional memory for merging and sorting. For example, set these values to: –TB 31 –TM 32.

Four Digit Year Default (–yr4def) — Add this parameter so that all years are always dumped as 4 digits. This avoids ambiguity when reloading data.

Progress AppServer

Another way to increase performance in a distributed computing environment is to use the Progress AppServer. AppServer goes a step beyond what a DBMS stored procedure provides since it provides the flexibility to run business logic on the same system as the database, or on another system. This flexibility allows the application developer to design an application that can take best advantage of the networking and computing environments available. By designing or modifying an application to take advantage of the AppServer, you can make the decision of where to run the business logic at deployment time.

When implementing a DataServer solution, the AppServer allows for moving the processing of business logic off of the client machine and onto a high-performance computer. This type of design places the logic in an environment that would reduce the network traffic to the client and thus boosts performance.

Skipping Schema Verification

When executing r-code, the Progress DataServer checks the data definitions of the target database to make sure they match the schema definitions in the schema holder. If they do not match, the DataServer returns an error. Unmatched definitions can result in the corruption of your target database. However, verifying the definitions is not needed in a production environment. The -Dsrv skip-schema-check startup parameter bypasses the checking of schema definitions. Use this parameter with caution, as it will not detect discrepancies between the schema in the schema holder and the data definitions in the DataServer database. If it continues to process queries, inserts, and selections, the target database might become corrupted.

The DBMS

This section covers techniques on optimizing the target DBMS, it is recommended only as a guide, and you should consult with your DBMS vendor for further details.

Hardware

Generally, tuning hardware requires working with three different areas. One of the three areas is always at the root of hardware bottleneck. This is not to say that a hardware change is always the solution for any given problem. Once a hardware bottleneck is identified, the other parts of the system, such as your network, should be reviewed to see if they should be modified instead of or in addition to the hardware changes under consideration. The three hardware areas include the:

- CPU
- System memory (RAM)
- Storage system (hard drives and access methods).

If your environment is constrained by the CPU it is an indication that your system is running at or near capacity. The only solution to bottlenecks in the CPU is to add more CPUs or to switch to faster CPUs.

Bottlenecks in system memory are resolved by adding more memory. In evaluating whether or not to add system memory, make sure that the computer system, the operating system, and your software will be able to take advantage of the increased memory.

The most common hardware bottleneck is the access speed of reading and writing to the storage system. Problems in the area of the storage system can be resolved by:

- Adding more hard drives
- Increasing the speed of the access methods
- Optimizing where objects are stored on the hard disks
- Using a combination of these options

The optimal configuration for a DataServer environment depends on the specific demands of the application. However, you can use the following general guidelines as a starting point when configuring a new system or when optimizing an existing system.

When designing a new system or adding to an existing system it is almost always more beneficial to have more smaller capacity hard drives than a few large capacity hard drives. Each additional hard drive adds flexibility in how the system can be configured.

The following logical software components are areas that can be spread across hard drives to maximize access speed:

- The operating system
- Temporary files
- Application source code
- The database
- The database log file

In an ideal environment, each one of these logical software components would be placed on separate hard drives. For the database, it is beneficial to use multiple hard drives. In most environments, though, there are not enough hard drives to place each of these components on a separate one.

The following examples shows how you might configure a given set of hard drives. If you have two hard drives, try placing the database log file on its own disk and all other components on the second disk. The advantage of this configuration is due to the high sequential write activity to the log file. With the exception of the log file all of the other components are often accessed randomly from one read to the next. With the log file, however, the access is usually a series of sequential writes. As the database is processing transactions, it must write to the log file to complete a unit of work. The entire system often waits during these log file writes. If the log file is on its own disk, the write head is almost always in the correct position for the next write. This allows the system to complete transactions faster. If even one other component is placed on the hard disk with the log file, you lose this performance advantage.

For systems that have three hard drives, the database should be placed on its own disk drive. Or as an alternative, and a better configuration for a larger database, a three-disk configuration should be set up to split the database across two disks with the third disk containing the database log file. Place the other components on the two disks that hold the database. If you have additional disks, you can expand this idea further by either moving additional components, such as temporary files to their own disk, or by continuing to spread the database across more disk drives.

Network Configuration

Typically the DataServer runs over a TCP/IP network. Progress Software Corporation provides no recommendations beyond the standard configuration for optimizing the network configuration. As a general rule, you can best utilize the network by minimizing the amount of network traffic. For remote connections you might want to consider locating the schema holder, application source code, and the progress client code on the client machine instead of on the network, or on a local server. Any benefit in performance that may be gained from locating these objects locally needs to be weighed against the increased maintenance that is required to update these objects in multiple locations, as opposed to a single location on a network.

Naming Conventions

All DBMSs have certain naming conventions that must be adhered to for naming database objects such as tables, columns, sequences, indexes, etc. These conventions forbid the use of some special characters that are otherwise acceptable in the Progress database.

As a general rule, avoid the use of hyphens (-) and percent signs (%) in all object names. Be mindful of any limits on the length of object names within the target database (Progress has a maximum of 32 per table name, Oracle only 30) remembering that the Progress DataServer technology may need to append extra characters to an object name to support extended Progress 4GL compatibility (RECID column or ARRAY support). In general, try to limit your object names to 25 characters.

All database management systems have their own set of reserved words that cannot be used to name database objects. Avoid the use of these reserved words (as well as all Progress reserved words) when naming objects. Progress databases and most ODBC data sources contain restrictions against using keywords as database object names. If an object name consists of a Progress keyword, the DataServer appends an underscore character (_) to the name. For example, an object named "each" becomes "each_"

Database Limitations

Depending on the version of the DBMS, there may be certain physical limits imposed by the database that can cause problems when converting a Progress database. One example of a particularly problematic limitation is Oracle's limit of 254 columns in a single table. (This limitation has been increased to 1,000 columns in Oracle8.) There are several instances where an equivalent non Progress table will require more columns than its Progress equivalent, such as to accommodate Progress arrays, case-insensitive indexes, and RECID values.

Meta-Schema References

Existing code meta-schema references qualified with a logical database name, such as <database>_field, must be modified. This type of issue is easily identified in a code base since it produces compile-time errors for each program that uses this type of database reference.

In the Progress database environment, the Progress meta-schema tables reside in the same database as the data. In this environment, database name qualifications to table and field references all use the same logical database name. When using a DataServer, the Progress environment has two connected databases. One database is the target foreign Database, the other is the Progress schema holder. In this environment the data resides in the target database and the Progress meta-schema tables reside in the schema holder database. In this case there are two logical database names.

You can resolve compile errors from meta-schema references that are qualified with a database name in one of two ways depending on the number of databases in your environment. The most common case occurs where your environment uses only one Progress database. This means there is only one set of Progress meta-schema tables. If this is the case, then you can delete the logical database reference from the code. Progress Software Corporation recommends this method since the code change works for each type of data source. However, if you have multiple Progress databases in the environment, the database qualification should be changed to the logical database name of the Progress schema holder. In this case, the code might differ depending on the data source.

Data Types

Non-Progress DBMSs have specific data types that may vary from those of the Progress database. The Progress DataServer translates these data types as closely as possible into Progress equivalents. When a particular data type has more than one valid Progress equivalent, the DataServer supplies a default data type. One particular example is the single digit data type NUMBER(1) in Oracle. This can map to a single digit INTEGER data type (9(1)), or a LOGICAL data type in Progress. The Progress schema holder contains these Progress data type mappings for each underlying column. You can manually change the default data type mapping in the schema holder using the Progress
Data Dictionary.

Additional Database Objects Required

In order to make the Progress 4GL 100% compatible with the target DBMS, the Progress DataServer may require that additional columns, indexes and sequences be created in the target database. These additional columns are to allow support of Arrays, Case-insensitivity and the Progress RECID functionality.

Arrays

Non-Progress DBMSs do not support arrays in the way the Progress RDBMS does. However, the Progress DataServer allows you to extend this array support to the target database. As an example, a Progress array of 12 elements would be implemented in the target database by creating 12 distinct fields in a certain order, and with a specific naming convention. The Progress DataServer will then recognize these fields as forming an array, which the 4GL can access in its normal fashion.

Please note that the creation of multiple fields for array representation demands close investigation because of limits to the number of columns per table imposed by the target DBMS. So for each Progress array element there will be an equivalent field in the target database, the naming convention of this ‘array element’ will depend on the DBMS being used.
The Progress database allows you to define fields as arrays, also called field extents. Progress interprets specially named columns of the same data type as a Progress field with the same number

of array elements. You must name the data-source columns column-name##1, column-name ##2, and so forth. (In Informix, however, you must name these columns column-name__1, column-name __2, and so forth. Informix does not allow the pound sign (hash sign) in object names.) The DataServer creates a single field definition in the schema holder for the field extents.

Case-Insensitive Indexes

By default, all character indexes defined in a Progress database will be case-insensitive (for example, the letter "a" is equivalent to "A" in a sorting algorithm). The Oracle DBMS does not provide case-insensitive indexing prior to Oracle 8i. Therefore, as an example, the letter "a" will sort differently than its uppercase equivalent "A." The Progress DataServer can support case insensitivity in a non-Progress database to mask this difference by supporting the use of a specially named column (storing only uppercase values of the original column) and an index on that column. If this table is to be updated by non-Progress applications, a target database trigger should also be created to populate this shadow column with an uppercase value of the original column. This additional column is know as a 'schema shadow column' and can be removed from the Progress schema holder if case sensitivity is not important to your application. The DataServer administration tools provide the option of creating shadow columns during the migration process.

RECIDs & ROWIDs

The Progress DataServer will support Progress RECID functionality in the target database through the use of an additional unique integer column on the table (called PROGRESS_RECID). This additional column will. also require an index. The Progress DataServer will automatically increment the sequence, populate the PROGRESS_RECID field, and modify the index when new records are added to the table through the Progress 4GL interface. If records are to be added to the table from outside of a Progress environment, the PROGRESS_RECID field will need to be updated manually. Refer to the Application Design section of this document for a further discussion of using ROWIDs and RECIDs. Please note that the creation of PROGRESS_RECID columns demands close investigation because of limits to the number of columns per table imposed by the target DBMS.

Unknown Values / NULLs / Zero-Length Character Strings

The Oracle and ODBC DBMSs do not support the concept of unknown values (represented as the "?" value) as it applies to a Progress database. The Progress DataServer extends functionality by allowing for this unknown value. This is done by mapping the Progress unknown value to a NULL or a zero-length character string in the target database. This could have implications to unique indexing of existing data, as Oracle allows only one NULL value per unique index. An in depth knowledge of your DBMS is required to ensure that your Progress application performs as you would expect expected, index and NULL differences can easily cause data to be sorted and return in a non-Progress manner, thus causing inconsistencies in the application.

Fixed vs. Variable Length Character Strings

One of the more common problems found in migrating a database from Progress to a SQL database is the format size relative to the actual data that is stored in the database. SQL-based databases such as Oracle and MS SQL Server typically have a maximum size for each field that must be specified at the time a table is created. When you use the Progress tools to migrate a database, Progress uses its native format definition to specify how large the fields should be. An error will occur if the format is exceeded when data is stored into the field. This type of error occurs with two Progress data types: character and decimal.

The Progress RDBMS stores character data in variable length format. This means that a 20 character string will always be stored as 20 bytes in the database (when using single byte character sets), while a 16 character string will always be stored in 16 bytes, (regardless of how the column is modified after it is created). SQL-based DBMSs allow both variable and fixed length string storage. For instance, if a column in Oracle is defined as type CHAR with 18 characters, then it

will store 18 characters, no more and no less. Hence, our 20 character string will be truncated to 18 characters, and our 16 character string will contain two trailing blanks. Oracle also has the VARCHAR2 data type that allows a column to be defined as being of variable length up to a user-defined maximum limit. This means that any character strings longer than the specified maximum will be truncated. Therefore, a VARCHAR2 column with a maximum length of 18 characters will truncate our 20 character string to 18 characters when stored in the Oracle database, but the 16 character string will remain 16 characters in length.

Additionally, an option exists on the Progress to Microsoft SQL Server migration tool (ProToMss) that allows you to use the Width field instead of the Format field when specifying the physical storage size. Progress provides the width field so that the physical size of a database field can be specified that is different than the display format (this SQL Width property is also used for the Progress SQL-92 engine). The initial value of the width fields depends on the data type. The character data type will have a value two times the format. The decimal data type will be 15 digits plus the number of decimals declared. The width field can be modified through the Progress Data Dictionary by using the SQL Properties option. If you choose to use the width field instead of the format field when creating a db table, you can control the size of the SQL Server field without changing anything the Progress client will use (i.e. the Format field). Since this method does not change the Progress database version of the application, Progress Software Corporation recommends this method for resolving field size problems.

Errors can occur when loading data. In the case of a character data type, the error Progress generates is "You tried to compare or to update a character field with a value longer than the maximum length (6142.)" There are two ways to resolve this error: you can increase the field size, or you can change the data type to TEXT. In almost all cases, you should increase the field size. See the section, *Large Formats*, in this White Paper for more information on TEXT data types.

If you are migrating a database, you can go back to the Progress Data Dictionary and update the width field. After you update the width field, you can re-migrate the database using the ProToMss tool. In the case where the value of a decimal exceeds the size defined for the decimal data type, the Progress session will crash. Generally these types of errors are less common and harder to track down than problems with the size of character formats.

Trailing Blanks

DBMS's handle trailing blanks differently than Progress. This presents problems, particularly when using a SUBSTRING function to obtain a string to store in the database. This string will be padded with blanks by Oracle and ODBC if it is not as long as the SUBSTRING definition. This is not a problem when running an application against a Progress database. As an example, if we use the SUBSTRING(4,4) function on the character string "YELLOW", Progress will return the three character string "LOW". Oracle however, will return the four character string "LOW " (note one trailing blank space). As a general rule, it is good practice to trim trailing blanks from character strings before assigning them to the database.

Corporate Headquarters
Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA Tel: 781 280 4000 Fax: 781 280 4095

Europe/Middle East/Africa Headquarters
Progress Software Europe B.V. Schorpioenstraat 67 3067 GG Rotterdam, The Netherlands Tel: 31 10 286 5700 Fax: 31 10 286 5777

Latin American Headquarters
Progress Software Corporation, 2255 Glades Road, One Boca Place, Suite 300 E, Boca Raton, FL 33431 USA Tel: 561 998 2244 Fax: 561 998 1573

Asia/Pacific Headquarters
Progress Software Pty. Ltd., 1911 Malvern Road, Malvern East, 3145, Australia Tel: 61 39 885 0544 Fax: 61 39 885 9473

PROGRESS SOFTWARE

www.progress.com