



**hp e3000**  
solution transition  
advisor

**turboimage**  
and databases



**white paper**

# table of contents



executive summary	1
overview	1
turboIMAGE	2
migrating turboIMAGE applications	2
eloquence	3
mainstream relational databases	4
embedded sql	5
programming style differences: chained reads in turboIMAGE	6
cursors in embedded sql	7
re-writing turboIMAGE intrinsics into embedded sql	7
functionality that exists in turboIMAGE, but not in sql	8
image locking	8
locking in a relational dbms	9
locking in an rdbms	10
data types	10
dates	11
special characters	12
database structure	12
advantages of using an rdbms	13
turboIMAGE intrinsics wrappers	14
allbase/sql	16
converting schema	16
converting/moving data	16
converting constraints, procedures and rules	17
converting programs	18
appendix a: example chain read using turboIMAGE	18
appendix b: example cursor read using embedded sql	20
appendix c: comparison of datatypes	22

## executive summary

Because of HP's strategic decision to discontinue support of its HP e3000 platform, HP recognizes its customers are faced with making significant decisions on transitioning their HP e3000 solutions to other platforms. At this time, many questions arise on how to manage this process:

- What is the impact on affected business functions?
- What are the capital and human costs associated with transition efforts, both now and in the future?
- How can risk be mitigated?
- What is technically required for transitioning all or part of the HP e3000 solution?
- What are possible transition strategies?
- What are the appropriate Independent Software Vendors (ISV) migration tools or mix of tools to use in specific components of the HP e3000 software solution?

There are a total of six white papers in the "HP e3000 Solution Transition Advisor" series. Each one addresses the subject of e3000 transitions from a different angle. The subjects of the six white papers are:

- Business Planning Guide
- Technical Planning Guide
- Compilers and User Interfaces
- TurboIMAGE and Databases (this paper)
- KSAM and the MPE File System
- MPE Commands and Networks

## overview

This is the fourth of a series of white papers from Hewlett-Packard, providing HP e3000 customers with a transition process roadmap to assess, analyze, select and manage the appropriate transition strategy for their HP e3000 solutions. It is not intended to handle all HP e3000 transition issues, but a guide to:

- TurboIMAGE: This paper will explore various ways of porting software using TurboIMAGE to alternative databases.
- We'll provide some guidelines for selecting a databases on your target platform.
- We'll explore the different TurboIMAGE migration mythologies.
- Allbase: This white paper will discuss migrating Allbase/SQL databases to other relational databases.

# turboIMAGE

TurboIMAGE/XL is a database management system that operates only on HP e3000 computer systems. HP bundles TurboIMAGE with current HP e3000 systems, and has done so throughout most of the life of the e3000 platform. Consequently, the vast majority of HP e3000 applications are based on the TurboIMAGE DBMS.

TurboIMAGE offers programmatic access to the data through specialized system procedures called *intrinsic*s. This distinguishes TurboIMAGE from relational databases such as Oracle or SQL/Server. TurboIMAGE/XL is not a relational database — it is based on a network data access model. The TurboIMAGE intrinsic interface is very different from the embedded SQL that is used to access relational databases such as Oracle or SQL/Server.

## migrating turboIMAGE applications

When migrating HP e3000 applications to other platforms, it's important to understand the database dependencies that exist in the applications being ported. Most HP e3000 applications use the TurboIMAGE intrinsic interface — making them dependent upon TurboIMAGE. The TurboIMAGE database management system is not available for Linux®, Windows®, HP-UX or any platform other than MPE. Therefore, applications being moved to another platform must use a different DBMS.

In this white paper, we'll explore three options for migrating TurboIMAGE applications to take advantage of another more powerful DBMS:

1. Use a DBMS on the target platform that can be accessed using the TurboIMAGE intrinsic interface. There is one such DBMS; it is called **Eloquence**. Eloquence is supported on all of the recommended target platforms. Using Eloquence allows you to leave the TurboIMAGE intrinsic calls in your application after migrating it to HP-UX, Linux or Windows.
2. Redesign your application to use an RDBMS. Remove all the TurboIMAGE intrinsic calls from your application, and replace them with appropriate programmatic calls to access your chosen RDBMS on your target platform. These calls will most likely take the form of **Embedded SQL** statements. This approach requires a good deal of work because of incompatibilities between TurboIMAGE and mainstream RDBMS products. These incompatibilities include differences in programming style, functionality differences such as locking, data type differences, dates, special characters and database optimization.
3. Use a mainstream relational DBMS such as Oracle or SQL/Server, while leaving the TurboIMAGE intrinsic calls in your application. on the target platform. This can be achieved using a software library that intercepts the TurboIMAGE intrinsic calls from your application, and translates them into SQL calls for use with the relational DBMS. These libraries are sometimes referred to **IMAGE Wrappers**.



# eloquence



## eloquence

Eloquence is a database management system that has been available on the market for over 10 years. It has an active user base and it has gone through a continuous development process. Eloquence is available for the HP-UX, Windows and LINUX platforms.

Internally, the architecture of Eloquence is different from that used by TurboIMAGE. They are not the same product. However, with the TurboIMAGE Compatibility Option, the interface to the Eloquence database is virtually identical to the HP e3000 TurboIMAGE intrinsic interface. The Eloquence database supports all the TurboIMAGE intrinsics and all of the Image data types. Consequently, few if any code changes are required when porting from TurboIMAGE on the e3000 to Eloquence on one of the recommended target platforms. An application that has been ported to a platform with the Eloquence DBMS installed can access Eloquence databases just as if they were TurboIMAGE databases. From a programmer's perspective, Eloquence and TurboIMAGE are almost interchangeable.

From an administrative point of view, there are differences in the way you manage Eloquence databases. But these are comparatively minor. If you currently have people trained in the administration and maintenance of TurboIMAGE databases, these people should be able to continue to do their jobs with Eloquence databases with very little re-training — certainly far less than would be required to learn to manage a 'mainstream' relational DBMS. This feature, combined with the comparatively small licensing fees for Eloquence, make it an attractive option, especially for companies with many employees that have skills and in-depth knowledge of TurboIMAGE.

A growing community of TurboIMAGE tools vendors are enhancing their products to work with Eloquence. For example, Robelle's Supertool now offers Eloquence support. The Eloquence web site at [www.marxmeier.com](http://www.marxmeier.com) has a current list of TurboIMAGE tools that have been enhanced for Eloquence.

Eloquence has a lot to offer to customers who are migrating HP e3000 applications to other platforms. However, that doesn't necessarily mean that it is the best solution for every application:

- **limited choices of tools and applications** — Although Eloquence enjoys support from a number of current and former HP e3000 partners, it cannot match the selection of 3rd party tools and applications that are available which support mainstream DBMSes such as Oracle or SQLServer.
- **company DBMS policy** — During the past decade, many organizations chose to standardize all their application program development efforts around a particular mainstream DBMS product. In that case, Eloquence may not be the best option. For example, if your company uses Oracle everywhere else, then it may make sense to go through the additional effort necessary to port your HP e3000 application to work with Oracle on your target platform.
- **scalability** — Finally, Eloquence has not been scaled to support very large implementations. Although the database is constantly being refined, it cannot be accessed by more than 500 concurrent users. Applications with over 300 concurrent users should probably look at mainstream relational databases. Check the Eloquence web site at [www.marxmeier.com](http://www.marxmeier.com) for the latest information.

Changing from TurboIMAGE to another DBMS can be one of the most complex aspects of an e3000 transition project. Because Eloquence simplifies this part of the project so much, it makes it possible to migrate TurboIMAGE applications very quickly.

## mainstream relational databases



For this reason, some organizations have elected to use Eloquence as a transitional tool — porting to their chosen target platform with Eloquence first, and then porting from Eloquence to another DBMS product over time. This transitional strategy can be especially useful if there is concern about finishing your transition project in time. HP has announced it will no longer support MPE/iX (or TurboIMAGE) after December 31, 2010, so transition projects will need to be completed by that date. If it turns out that porting to a mainstream RDBMS puts that deadline at risk, porting to Eloquence may allow you to get off of the HP e3000 more quickly. In that case, your transition project will have a second phase — that of porting from Eloquence to another DBMS.

### mainstream relational databases

When the HP 3000 was first introduced in the 1970s, the company took the then unprecedented step of bundling the IMAGE DBMS in with every system. This move contributed to the early success of the HP 3000, because the presence of IMAGE (and later, TurboIMAGE) on every e3000 server made it very easy for the development community to create applications for the e3000. On other platforms, developers either had to convince customers to buy a DBMS, or develop one themselves.

In the 1990s, however, the “TurboIMAGE Advantage” became a disadvantage for the HP 3000. By this time, most computer vendors were bundling proprietary DBMS products with their hardware platforms. This meant that customers who were using more than one hardware platform had to deal with multiple, incompatible DBMS products as well. Customers found it difficult (if not impossible) to write applications that spanned multiple databases. Companies like Oracle and IBM responded to this need by bringing to the marketplace relational databases that ran on multiple hardware platforms.

For example, during this period, many customers had heterogeneous datacenters containing hardware platforms from HP, IBM, Intel® or any of several other vendors. Normally, writing an application that could access more than one database meant using a different database interface language for each platform. But those who standardized on a cross-platform relational DBMS, (Oracle, for example), could write application code once, knowing that it would work with Oracle databases on any platform where Oracle was supported.

By the mid-1990s, Oracle was the dominant player in the database market. It was widely used on HP-UX, Solaris, VMS and virtually every other commercial operating system in use - with one exception: MPE/iX. Of all the bundled DBMS products, none had the acceptance and support of TurboIMAGE. On the HP 3000, the strength and popularity of TurboIMAGE worked against the adoption of Oracle. Although there was an Oracle product for the HP 3000, it was never widely accepted and is today no longer available.

On HP-UX, Linux and Windows, however, customers have a choice of commercially available relational databases including Oracle and Microsoft’s SQL Server. In addition, there are relational databases such as MySQL which can be downloaded from the internet at no charge.

If Eloquence is not a solution for your applications, you may want to use one of these DBMS products instead. The chief barrier that stands between a typical HP e3000 application and a relational DBMS is the fact that none of these products supports the TurboIMAGE intrinsic interface. This means that some additional effort will be required to make your HP e3000 application work with a new DBMS.

# embedded sql

## embedded SQL

The most common way of accessing a relational database from a program is through the use of **embedded SQL**. SQL stands for “Structured Query Language”. It is a language that was designed specifically for accessing database management systems.

It’s worth noting in passing that the TurboIMAGE DBMS was enhanced in the 1990s to make it accessible using the SQL language. This feature made it possible to access TurboIMAGE databases using client/server programs such as the Microsoft Office suite. To promote the awareness of this feature, HP renamed TurboIMAGE to IMAGE/SQL.

This feature effectively makes IMAGE/SQL a “bi-lingual” database. It can be accessed using SQL statements, or it can be accessed using the traditional (proprietary) IMAGE intrinsic interface. Unfortunately, this feature is usually little help to customers who are migrating HP e3000 applications to other platforms. Because most of these applications were designed and written long before the IMAGE/SQL enhancement, they use the intrinsic interface exclusively.

One way this feature might be useful to a customer who is migrating off the HP e3000 would be as an interim step in porting the application to the target platform. An HP e3000 application could be modified to use embedded SQL statements instead of TurboIMAGE intrinsics. With the IMAGE/SQL enhancement, this SQL version of the application could continue to run on the e3000 (against the same databases that it had been using all along). When the customer is ready to move the application to another platform, the embedded SQL will have been thoroughly tested and “shaken out” in a production environment.

It’s important to understand, however, that IMAGE/SQL is optimized around the TurboIMAGE intrinsic interface. The SQL interface to IMAGE cannot deliver the same performance as the TurboIMAGE intrinsic interface. Hence, the modification of an HP e3000 application to use embedded SQL can have grave performance repercussions.

As we’ll see, this is not only true of embedded SQL with IMAGE. The redesign of an application that is today based on TurboIMAGE intrinsics to use embedded SQL instead is fraught with potential performance issues, regardless of whether the target database is IMAGE/SQL, or a mainstream dbms such as Oracle or SQL/Server.

Embedded SQL statements are placed in your application source code, in much the same way that TurboIMAGE intrinsics were embedded in your source code when the application was using TurboIMAGE. In order to make embedded SQL work, you must run your code through a pre-processor. This pre-processor is supplied by the database vendor, and it generates the instructions that actually access the database.

There is an ANSI standard for embedded SQL statements. This means that if you write your code to conform to this standard, it will work with most relational databases. However, almost all relational database products have been enhanced beyond the ANSI standard. In order to get the best performance and functionality out of the database, it’s often necessary to write code that steps outside of the ANSI standard. Consequently, most applications with embedded SQL statements are tied to a particular DBMS. For example, a program that was written to work with Oracle cannot typically work with SQL/Server without significant modification.



# programming style differences



Writing a program that uses embedded SQL is very different from writing a program that uses TurboIMAGE Intrinsic. This difference in programming style may be one of the biggest problems that must be overcome when migrating TurboIMAGE code to use a different database. This paper is not intended to provide you with a step by step process on how to migrate a TurboIMAGE database. However, we will identify some of the most common problems that you may encounter.

## programming style differences: chained reads in TurboIMAGE

A chained read is one of the most common ways of accessing data in a TurboIMAGE database. The following example is based on an application that accesses a TurboIMAGE database using TurboIMAGE intrinsic. The database contains, (in addition to other datasets), a master dataset with one record for each customer, and a detail dataset with one record for each order. The orders are indexed by customer number.

In order to perform a typical chained read, the application uses the DBFIND intrinsic to establish the head of the chain for a given index. In this example, that index is the customer number. DBFIND locates the record in the master dataset that matches the specified customer number, and tells the application how many records (if any) exist in the detail dataset for this customer.

If there are one or more records with the specified customer number, the application program will then call DBGET to retrieve the first order. Subsequent calls to DBGET will retrieve the remaining records associated with that index. Each record might represent an order that was placed by the specified customer. In TurboIMAGE, the set of records that share a common index (in this case, a common customer number) is called a **chain**, and the act of using DBGET to access all the records in the chain is called a **chained read**. Using a chained read, an application can select the records it wants from a TurboIMAGE database using a single search-item as a criteria.

Suppose that the application is looking for data using more than one criteria. For example, in order to locate all orders placed by a specific customer *on a specific date*, the application programmer would have to go beyond a simple chained read. In order to locate these records, the application would have to select them using the following 3-step process.

1. Read all the records in that chain, checking the date of each record as you encounter it.
2. Ignore the ones in which the date does not match the desired date.
3. When you reach the end of the chain, you will have the desired records.

## cursors in embedded sql



## re-writing turbolMAGE intrinsics into embedded sql



### cursors in embedded SQL

In order to better understand embedded SQL, let's look at how you'd handle the query we just explored using a chained read in TurbolMAGE. One of the strengths of the SQL language is its ability to handle more complex queries, including those with multiple criteria. Using embedded SQL statements and a relational database, it is possible to access all the records placed by a specific customer on a specific date as follows.

1. Use a SELECT statement with a WHILE clause. This statement will find all the orders for a specified customer on a specific date. Note the difference from the TurbolMAGE intrinsic interface. The SELECT statement only needs to be called once. TurbolMAGE required you to call DBGET repeatedly — once per record.
2. The SELECT statement will return a **cursor** to the calling application program. In relational terminology, a cursor refers to a chain of qualifying records. Each record is called a row.

This example should make it clear that the skills and knowledge required to code embedded SQL statements are very different from that are required to use the TurbolMAGE intrinsic interface. Therefore, re-designing a TurbolMAGE program to use embedded SQL is not a simple process.

### re-writing turbolMAGE intrinsics into embedded SQL

Although this is a challenging task, there are pitfalls that can be avoided. When re-writing TurbolMAGE intrinsics into embedded SQL statements, some programmers will be tempted to simplify the SELECT statements so they only do only as much as the DBFIND and DBGETs that they're replacing do. This reduces the number of changes that must be made to the original program logic and code. However, it can result in embedded SQL which may be very inefficient, and which can cause serious performance problems down the road.

In embedded SQL, the execution of a single SELECT statement typically takes many more computer resources than the execution of a single DBFIND and DBGET. Furthermore, any SELECT statement that you code will use about the same amount of resources, regardless of how many rows it selects.

Therefore, embedded SQL applications should be designed to process as many selection criteria as possible in a single SELECT. The TurbolMAGE-like coding practice of calling a SELECT with a single selection criteria, then retrieving each record and checking it against the remaining criteria would consume additional computer resources unnecessarily.

Coding embedded SQL statements as if they were TurbolMAGE intrinsic calls will almost surely result very inefficient programs that require more system resources to do the same job.

See appendix a and b for TurbolMAGE and embedded SQL COBOL code examples.

## functionality that exists in turbolIMAGE, but not in sql

### functionality that exists in TurbolIMAGE, but not in SQL

For the most part, the functionality provided by TurbolIMAGE maps neatly into the functionality provided by SQL. The syntax is different, and the way that you access your data may need to be changed in order to get the best performance, but ultimately you can do the same things.

There are exceptions to this rule, however. There are some things that can be done in TurbolIMAGE that simply cannot be accomplished using embedded SQL. For example, once you've selected a cursor using SQL, you can only read it in one direction. There is no simple backward chained read with relational databases.

Backward chained reads are supported in TurbolIMAGE. If your application uses this feature, the program logic will have to be changed significantly in order to accommodate the use of embedded SQL statements.

### IMAGE locking

Like virtually all database management systems, the TurbolIMAGE intrinsic interface allows multiple users to access a TurbolIMAGE database at the same time. However, in order to coordinate multi-user access, it makes some very stringent demands on the application program, and on the application programmer in particular.

TurbolIMAGE requires that each process that is accessing a TurbolIMAGE database manage the locking and unlocking of the database. In TurbolIMAGE terminology, an application uses a lock to indicate that it is in the process of updating a particular database (or a particular part of a database - such as a particular record or dataset). This lock prevents other processes from trying to update the locked data at the same time. When the process running the application is finished with the update, it unlocks it. Without some kind of locking mechanism, the database can become corrupt.

TurbolIMAGE provides a locking concept called **predicate locking**. A predicate lock is a way of establishing a lock without locking a physical record. For example, suppose an application is about to make some changes to the data associated with customer number 123 in the customer master dataset. It can use the DBLOCK intrinsic to establish a lock on the specified data. In fact it can do this, even if there is no customer number 123 record in the database. The strength of a predicate locking mechanism is that you can ensure that no one else uses the customer id of 123 while your process is generating the record.



## image locking



# locking in a relational dbms



## locking in a relational DBMS

Relational databases, if they do support locking to row level granularity, can only lock a record that already exists. This is not to suggest that they cannot handle the scenario explained above. You can insert an incomplete customer record number 123 and then read it back. This will cause the database to lock this row and no one else can update it. Once you complete the record and commit the update the lock will be released.

Another difference between locking in SQL and in the TurboIMAGE intrinsic interface is in the way locks are accumulated. In TurboIMAGE, and in SQL, it is possible to hold more than one lock at a time. This practice is necessary for the processing of certain kinds of complex transactions. However, if you use the TurboIMAGE intrinsic interface to obtain multiple locks, it is the programmer's responsibility to guard against the possibility of causing **deadlocks**.

A **deadlock** happens when two processes are collecting locks. Suppose that both processes want to perform a transaction that will affect two records — we'll call them "A" and "B".

- Process number 1 begins by locking record "B". It knows it also needs to lock record "A" before it can apply its transaction, so it asks for a lock on "A". But suppose that record "A" has already been locked by another user. In that case, process number 1 must wait for the lock on "A" to be released.
- Process number 2 is holding lock "A." However, process number 2 knows that in order to apply its transaction, it needs to lock both "A" and "B". So it asks for a lock on record "B" only to find that process number 1 is already holding the lock on "B".

This is the simplest kind of deadlock. Process 1 cannot proceed until it locks "B" - which it cannot do because process 2 is holding the lock on B. Similarly, process 2 cannot proceed until it locks "A" - which it cannot do because process 1 is holding the lock on "A". Neither process can proceed. Neither process can unlock the resources it's holding to allow the other process to proceed.

This is a simple 2-process deadlock. Note that deadlocks can be far more complex than this. (For example, deadlocks may be caused a several processes, each waiting for a resource held by the next process, until the last process tries to lock a resource currently locked by the first.)

The TurboIMAGE intrinsic interface demands that your application contain logic to prevent deadlocks. It does not provide a means of breaking them once they occur. On the e3000, the only way to break a deadlock such as the one we have just described is to literally shut down the entire system and reboot.

In order to prevent deadlocks, your application must collect your locks in a specific order to make sure deadlocks don't happen. Any deviation from this rule will almost surely result in a deadlock - sooner if not later.

# locking in an rdbms

## locking in an RDBMS

With relational databases the programmer does not control the collection of locks. The relational database engine does whatever locking is necessary for you.

Relational database engines contain logic to detect deadlock conditions. When it's determined that a process is involved in a deadlock, the DBMS will roll back the transaction that the process is trying to apply. To achieve this, it will release the locks held by the process, and return an error condition to the deadlocked process.

This effectively aborts the transaction that the process was attempting to apply. If a process becomes deadlocked, the application program must restart the transaction. This means that every application program must contain logic to automatically restart transactions that become deadlocked.

Because properly coded TurboIMAGE programs cannot become deadlocked, HP never provided a mechanism to escape from deadlocking (short of restarting the system). When porting e3000 applications to other databases that do provide such a mechanism, it is necessary to add this logic to the application.

## data types

There is an ANSI standard for the **data types** supported in relational databases. For example, in order to conform to this standard, a relational database management system must allow data items to be stored as binary numeric items. A numeric item is a standard data type. Similarly a string of bytes (i.e., an ASCII strings) is another standard data type supported by virtually every DBMS.

The TurboIMAGE DBMS supports a number of data types that are not part of the ANSI standard for RDBMS products. Similarly, there are some data types in the ANSI standard for RDBMS products which are not supported by TurboIMAGE. Needless to say, this can cause problems when porting e3000 application programs that currently use TurboIMAGE to another DBMS.

Many HP e3000 applications use data that is stored in the "packed decimal" or "zoned decimal" format. Packed decimal numbers are stored in a decimal-encoded format which has the virtue of being easy to read in a memory dump. It is supported by the standard COBOL language, and some older computer system architectures even provided support for it in the machine instruction set.

PA-RISC models of the HP e3000 provide support for packed decimal in millicode. On the older 16-bit "classic" HP 3000 architecture, there were hard-wired machine instructions for doing packed decimal arithmetic.

Today, numeric data is almost always stored in the binary format, which allows the computer to do arithmetic operations on the data without the need for time-consuming conversions. Consequently, the Relational standard does not provide for a packed decimal data type. Most RDBMS products do not allow you to store numeric data items in their data bases unless they are in the standard binary format. If your e3000 application uses packed decimal, that data must be converted to binary before it can be stored in an RDBMS.



# data types



Compound fields (also known as arrays) are another data type supported by TurboIMAGE, but not by the relational standard. TurboIMAGE allows you to define an array of fields in a database, and refer to the whole array using a single identifier. This idea is foreign to the “rows-and-columns” structure of an RDBMS.

If your e3000 application uses array items, they must be converted to other data types when you migrate your application to use an RDBMS database. Exactly what you’ll convert them into depends upon the structure of the array itself, and how it is used in the e3000 application. For example, one technique might be to replace a TurboIMAGE array of numeric items with an entire column in a relational table.

### dates

TurboIMAGE has never supported a data type for dates. Typically, a date in a TurboIMAGE database would be represented by an array of three numeric items representing year, month and date. For example, the date: “October 11, 2002” might be represented as YEAR MONTH DAY “2002”, “0010”, “0011”.

This scheme has advantages and disadvantages. One of the disadvantages is that you cannot do arithmetic with this kind of date. For example, if you wanted to know what date comes 45 days after October 11, 2002, you couldn’t just add 45 to the date field. (If you did, you’d come up with October 56, 2002, which is obviously invalid.) To do date arithmetic in a TurboIMAGE application, your program would have to contain all the logic necessary to do the date manipulation, (including knowing which months have 30 days, which ones have 31 or 28, which years are leap years, etc. etc.) Date functions are available with embedded SQL.

One interesting characteristic of storing dates in this way is that the three numeric items can be used to represent things other than a valid date. For example, a field labeled “Completion-Date” might contain a valid date if the project being referred to is in fact completed. If the project is not yet completed, the “date-completed” field might be loaded with a string such as 99/99/9999. This is a programming technique that is widely used in TurboIMAGE applications, precisely because there is not data type specifically for dates.

In the relational standard, there is a data type specifically for dates. Fields that are defined using this data type must contain a valid date. Any attempt to place an invalid string (such as 99/99/9999) into a field defined as a date will simply generate an error. In this way, RDBMS products protect themselves against the entry of invalid dates by tired data entry clerks or by programming errors.

Many TurboIMAGE applications put non-date information into numeric fields that are otherwise being used to represent dates. This kind of logic must be changed if you want to use fields defined with the date data type when you’re porting to an RDBMS. The alternative is to use ordinary character fields for your dates and forgo the additional protection from errors that a date field would provide you.



## special characters

### special characters

TurboIMAGE/XL data sets and items may contain some characters that are not allowed in RDBMS tables. For example, the hyphen is not allowed in relational table names, where it is one of the most commonly used characters in TurboIMAGE dataset names. Therefore, hyphens must be represented in another way in an RDBMS. The underscore ( `_` ) character is one of the most commonly used substitutes for the hyphen. Specifically, the characters not allowed in RDBMS table names are: `+ - * / ? ' % & .`

It is common for a specific language preprocessor to convert special characters, for example hyphens in COBOL code will automatically be converted to underscore for accessing the database.

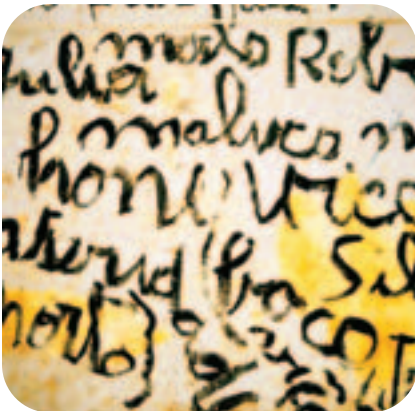
### database structure

For simplicity's sake, it's usually best to keep the basic structure of your database the same when converting from TurboIMAGE to relational.

- TurboIMAGE data Items (aka "fields") become columns in an RDBMS
- TurboIMAGE datasets become tables
- TurboIMAGE data entries (aka "records") become rows
- TurboIMAGE automatic masters become indexes.

This mapping keeps the basic access design of a program consistent, and minimizes the logic changes necessary in the applications that use the database. Unfortunately, a relational database that is created in this way may not be optimized for SQL data access. As with virtually everything else in life, there are trade offs that must be made with the conversion. The optimization of relational database management systems is a complex topic that is beyond the scope of this white paper. It is dependent upon the specific DBMS product that you're using, and the amount of optimization required. For more information, refer to the documentation and training provided with your chosen RDBMS product.

## database structure



## advantages of using an rdbms



### advantages of using an RDBMS

Redesigning an e3000 application to use an RDBMS is not a simple task. There are a lot of issues to be taken into account, and plenty of opportunities for errors to creep into your porting project.

But even with all the porting challenges, RDBMS products offer many advantages over TurboIMAGE.

- Embedded SQL database access is standard and many programmers know how to write to it.
- There is a wealth of 3rd party tools as well as additional application available the run on relational databases and skilled relational database administrators (DBA's) are readily available.
- Relational databases are the most common databases used today. The largest search engines on the internet are based on RDBMS technology. Because RDBMS products are designed to "scale out" as well as "scale up", there is almost no practical limit on the size and speed of the data bases that can be created using RDBMS technology.

# turboimage intrinsic wrappers



## turboIMAGE intrinsic wrappers

Because of the difficulty of migrating TurboIMAGE databases to use the relational model, a technology has been developed to map TurboIMAGE access to relational databases. This technology makes it possible to modify a TurboIMAGE application to use an RDBMS, and yet still leave all the references to the TurboIMAGE intrinsic interface intact.

When the application calls a TurboIMAGE intrinsic, the call is intercepted by a software library that translates the TurboIMAGE intrinsic into a corresponding SQL statement. Because the RDBMS is effectively being “wrapped” in a layer of TurboIMAGE software, this software library is sometimes referred to as an “IMAGE Wrapper.”

TurboIMAGE intrinsic wrappers allow application programs to continue calling TurboIMAGE intrinsics. The migration can be completed much more quickly, because the application programs do not have to be changed to support the RDBMS. All the complexities of accessing the relational database are performed by the wrapper code. Field names are mapped, data types are converted, and locking is automatically handled.

Most wrapper environments come with migration utilities that simplify the job of database migration. These utilities perform tasks such as generating the new database schema and migrating the database data.

With an IMAGE wrapper library and corresponding utilities, the first step in the conversion from TurboIMAGE to an RDBMS is automatic. However, it must be emphasized that the first step is exactly that; a first step. The project is not done when the automated part of the conversion is complete. In most IMAGE wrapper conversions, some modifications to the code must be done to address specific performance or functionality issues that may appear after the initial migration is complete.

In order to work around the differences between TurboIMAGE and the target RDBMS, specialized programming tools and techniques may be used to exercise additional controls over the RDBMS. For example, wrapper libraries and tools often add additional hidden fields to an RDBMS when it is built from a TurboIMAGE database. These products do this to support the TurboIMAGE simulation. Because of this, there may be problems with accessing the database through the wrappers and through conventional SQL access at the same time. Let’s look at an example.

In TurboIMAGE, data is stored chronologically, at least by default. This means that when you read data from a TurboIMAGE database, by default you’ll access the records in the same order in which they were added to the database — first in — first out.

If a TurboIMAGE application depends upon this behavior, then the application must be enhanced before it can be used with an RDBMS. Since an RDBMS does not “serve up” data in chronological sequence, (at least not by default) the wrapper library must do it on behalf of the RDBMS. In this way, the application program can continue to operate as if it were still operating against a TurboIMAGE database.

A common way of doing this is to add a sequence number to the database in order to support the chronological ordering of rows. Each time data is written to the database, the sequence number is incremented by one and written to the same row as the data. If the data is to be retrieved from the database in

chronological sequence, the wrapper library will retrieve it and sort it using the sequence numbers before presenting it to the TurboIMAGE application. In this way, the chronological sequence can be preserved.

The problem arises when another application tries to retrieve the data using conventional SQL. If this second application is not expecting to see the sequence number along with the data, it can result in problems. Users accessing the database without the wrappers must also maintain these hidden fields. Most relational database have stored procedures and triggers that can help with this.

TurboIMAGE wrappers can give you the best of both worlds, ease of migration and running on a mainstream relational database. But easing the database migration comes with a price.

- In order to successfully migrate applications from TurboIMAGE to an RDBMS, you must learn about both worlds: the TurboIMAGE database world and the relational database world.
- When you make changes to your programs you need to understand the TurboIMAGE access and what will be done on the corresponding relational database.
- Depending on what you do in your programs there could be some performance implications. All the additional functionality that's available with mainstream RDBMS products takes up additional computer resources.
- The cost of this solution is the highest because in addition to purchasing the relational database you are also purchasing the wrapper technology.

**table 1. available databases and TurboIMAGE wrappers**

<u>databases</u>	<u>product</u>	<u>comments/contacts</u>	<u>operating system</u>
	Eloquence	A low-cost Image like database <b>www.marxmeier.com</b>	HP-UX, Windows, Linux
	Oracle	<b>www.oracle.com</b>	HP-UX, Windows, Linux
	SQL Server	<b>www.microsoft.com</b>	Windows
	DB2	<b>www.ibm.com</b>	HP-UX, Windows, Linux
	MySQL	Freeware relational database <b>www.mysql.com</b>	Windows, Linux
	SAPdb	<b>www.sap.com</b>	HP-UX, Windows, Linux
	PostgreSQL	Freeware relational database <b>www.postgresql.org</b>	HP-UX, Windows, Linux
<u>wrappers</u>			
	AMXW	Part of their emulation environment <b>www.neartek.com</b>	HP-UX, Windows
	Transport	Part of their emulation environment <b>www.transport.bi-tech.com</b>	HP-UX, Windows
	Open Turbo	A standalone Image to relational database tool <b>www.imaxsoft.com</b>	HP-UX
	Omni Access	Uses their Omniaccess product as interface to relational databases <b>www.disc.com</b>	HP-UX, Windows
	ViaNova 3000	Part of their ViaNova migration package <b>www.ordina-denkart.com</b>	HP-UX, Windows, Linux
	Intelligent Adapters	Part of their migration services <b>www.transoft.com</b>	HP-UX, Windows

# allbase/sql

## Allbase/SQL

Allbase/SQL is a relational database available from HP that runs on the HP e3000. Most relational database support the ANSI standard so migrating between different relational database products is straightforward. This is not to say that relational database migrations are easy. There are many areas in the ANSI standard that allow for different implementations. Most database products support extensions to the ANSI standard.

Allbase/SQL migrations can be divided into four basic steps:

1. converting schema
2. converting /moving data
3. converting constraints, procedures and rules
4. converting programs

### converting schema

A schema is a definition of the structure of a database. To generate a new database the schema that was used to define the Allbase/SQL database must be modified to generate a new target database. Most schema Data Definition Languages (DDL) are similar but there are differences. For example there are variations in the datatypes that are allowed. See appendix A for a comparison of Allbase/SQL and Oracle datatypes. Additionally reserved words uses in the DDL maybe different.

### converting/moving data

All relational databases can dump their data to flat files and they all can load data from flat files. This is the simplest way to migrate data. There maybe data modification necessary because some of the datatypes support different value ranges and data size. There are products that will do automatic data movement and conversions. See the sixth white paper in this series "*HP e3000 Solution Transition Advisor – MPE Commands and Networks*" for more on data movement.

### converting constraints, procedures and rules

In addition to the basic tables and indexes in a DBEnvironment, ALLBASE/SQL lets you create database objects known as constraints, procedures, and rules, which provide for a high degree of data consistency and integrity inside the DBEnvironment without the need for extensive application programming.

**Constraints** define conditions on the rows of a table. **Procedures** define sequences of SQL statements that can be stored in the DBEnvironment and applied as a group either through rules or through execution by specific users. **Rules** let you define complex relationships among tables by tying specific procedures to particular kinds of data manipulation on tables.

Together, these tools let you store many of your organization's business rules in the DBEnvironment itself, reducing the need to develop application code. How constraints, procedures and rules are implemented is different for each relational database (in fact some simpler ones don't have this capability.) The language and syntax used to define the constraints, procedures and rules is different for each relational database. Because of the variations in implementations migrating constraint, procedures and rules is a manual process.



### converting programs

The only programmatic access Allbase/SQL supports is through embedded SQL. In order to make embedded SQL work, you must run your code through a pre-processor. This pre-processor is supplied by the database vendor, and it generates the instructions that actually access the database.

Fortunately this is where there are ANSI standards for embedded SQL statements. The Embedded SQL that is supported by different relational database products can be very similar. As with most implementations of a standard there are variations. Usually when migrating there are only slight syntax changes necessary in the embedded SQL to get the program compiled. Another thing to keep in mind is to make sure that the embedded SQL pre-processor supports the compiler language you are using. For example, C is always supported, but FORTRAN may not be.

Different relational database vendors may have migration services to help with your migration. In addition to the conversion steps discussed above each database is administered differently and have different tricks and techniques for optimizing performance and throughput. Training and possibly consulting will be necessary to utilize your new relational database system.

**table 2. available relational databases**

<b>product</b>	<b>comments/contacts</b>	<b>operating system</b>
Oracle	<a href="http://www.oracle.com">www.oracle.com</a>	HP-UX, Windows, Linux
SQL Server	<a href="http://www.microsoft.com">www.microsoft.com</a>	Windows
DB2	<a href="http://www.ibm.com">www.ibm.com</a>	HP-UX, Windows, Linux
MySQL	Freeware relational database <a href="http://www.mysql.com">www.mysql.com</a>	Windows, Linux
PostgreSQL	Freeware relational database <a href="http://www.postgresql.org">www.postgresql.org</a>	HP-UX, Windows, Linux
SAPdb	Open source database <a href="http://www.sapdb.org">www.sapdb.org</a>	HP-UX, Windows, Linux

# appendix a: example chain read using turboIMAGE



## example chain read using TurboIMAGE

DATA DIVISION.

WORKING-STORAGE SECTION.

```
01 DB-MODE                PIC S9(04) COMP.
01 DB-NAME                PIC X(10) VALUE SPACES.
01 DB-PASSWORD            PIC X(08) VALUE SPACES.
01 SET-NAME               PIC X(16).
01 ITEMS-LIST             PIC X(80) VALUE SPACES.
```

```
10 ORDER-DETAIL-BUFFER.
05 ORDER-NO-VALUE        PIC X(10).
05 CUSTOMER-NO-VALUE    PIC X(8).
05 PART-NO-VALUE        PIC X(12).
05 QUANTITY-VALUE       PIC S9(4) COMP.
```

05 SHIPMENT-DATE-VALUE.

```
10 DATE-YYYY            PIC X(4).
10 DATE-MM              PIC X(2).
10 DATE-DD              PIC X(2).
```

```
01 SEARCH-ITEM-NAME     PIC X(16).
01 SEARCH-ITEM-VALUE   PIC X(12).
01 START-DATE           PIC X(8).
01 END-DATE             PIC X(8).
```

```
01 STATUS-ARRAY.
05 CONDITION-CODE      PIC S9(4) COMP.
05 ENTRY-LENGTH        PIC S9(4) COMP.
05 RECORD-NUMBER      PIC S9(9) COMP.
05 CHAIN-LENGTH        PIC S9(9) COMP.
05 BACKWARD-POINTER   PIC S9(9) COMP.
05 FORWARD-POINTER    PIC S9(9) COMP.
```

PROCEDURE DIVISION.

```
:
```

RETRIEVE-ORDERS.

```
DISPLAY "Please enter the Customer Number to retrieve:"
  WITH NO ADVANCING.
ACCEPT SEARCH-ITEM-VALUE FREE.
DISPLAY SPACE.
DISPLAY "Please enter the Date Range to retrieve:"
DISPLAY "      Start Date (YYYYMMDD):"
  WITH NO ADVANCING.
ACCEPT START-DATE FREE.
DISPLAY "      End Date (YYYYMMDD):"
  WITH NO ADVANCING.
ACCEPT END-DATE FREE.
```

```
MOVE 1          TO DB-MODE.
MOVE "ORDER-DETAIL;" TO SET-NAME.
MOVE "CUSTOMER-NO;" TO SEARCH-ITEM-NAME.
```

```
CALL INTRINSIC "DBFIND" USING DB-NAME,
  SET-NAME,
  DB-MODE,
  STATUS-ARRAY,
  SEARCH-ITEM-NAME,
  SEARCH-ITEM-VALUE.
```

```
IF CONDITION-CODE NOT= 0
  IF CONDITION-CODE = 17
    DISPLAY "The Order does not exist."
  ELSE
    PERFORM IMAGE-ERROR
ELSE
  MOVE "@;" TO ITEMS-LIST
  MOVE 5   TO DB-MODE
```

```
PERFORM DISPLAY-ORDER
  UNTIL CONDITION-CODE = 15.
```

```
DISPLAY-ORDER.
```

```
CALL INTRINSIC "DBGET" USING DB-NAME,
  SET-NAME,
  DB-MODE,
  STATUS-ARRAY,
  ITEMS-LIST,
  ORDER-DETAIL-BUFFER,
  IGNORED-PARM
```

```
IF CONDITION-CODE NOT= 0
  IF CONDITION-CODE = 15
    DISPLAY "There are no more entries for that Order."
  ELSE
    PERFORM IMAGE-ERROR
ELSE
  IF SHIPMENT-DATE-VALUE LESS THAN START-DATE OR
    SHIPMENT-DATE-VALUE GREATER THAN END-DATE
  THEN
    NEXT SENTENCE
  ELSE
    DISPLAY "Purchase Order Number = ",ORDER-NO-VALUE
    DISPLAY "Part Number      = ",PART-NO-VALUE
    DISPLAY "Quantity Ordered  = ",QUANTITY-VALUE
    DISPLAY "Date Part Shipped  = ",SHIPMENT-DATE-VALUE.
```

# appendix b: example cursor read using embedded sql



## example cursor read using embedded sql

Note: The following code example is for comparison with the one in appendix A. No SQL optimization such as adding the SHIPMENT-DATE date check to the SELECT statement was done.

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
10 ORDER-DETAIL.
   05 ORDER-NO           PIC X(10).
   05 CUSTOMER-NO       PIC X(8).
   05 PART-NO           PIC X(12).
   05 QUANTITY          PIC S9(4) COMP.
06 SHIPMENT-DATE.
10 DATE-YYYY           PIC X(4).
10 DATE-MM             PIC X(2).
10 DATE-DD             PIC X(2).
   05 SEQUENCE-NO       PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

01 SEARCH-ITEM         PIC X(12).
01 START-DATE          PIC X(8).
01 END-DATE            PIC X(8).
```

PROCEDURE DIVISION.

```
:
```

RETRIEVE-ORDERS.

```
    DISPLAY "Please enter the Customer Number to retrieve:"
    WITH NO ADVANCING.
    ACCEPT SEARCH-ITEM FREE.
    DISPLAY SPACE.
    DISPLAY "Please enter the Date Range to retrieve:"
    DISPLAY "          Start Date (YYYYMMDD):"
    WITH NO ADVANCING.
    ACCEPT START-DATE FREE.
    DISPLAY "          End Date (YYYYMMDD):"
    WITH NO ADVANCING.
    ACCEPT END-DATE FREE.
```

MOVE SEARCH-ITEM TO CUSTOMER-NO.

```
EXEC SQL DECLARE ORDER-CURSOR
CURSOR FOR
SELECT *
FROM ORDERS.ORDER-DETAIL
WHERE CUSTOMER-NO = :CUSTOMER-NO
ORDER BY SEQUENCE-NO
END-EXEC.
```

EXEC SQL OPEN ORDER-CURSOR END-EXEC.

```
PERFORM DISPLAY-ORDER
UNTIL SQLCODE = 100.
```

DISPLAY-ORDER.

```
EXEC SQL FETCH ORDER-CURSOR
INTO ORDER-DETAIL
END-EXEC.
```

```
IF SQLCODE NOT= 0
IF SQLCODE = 100
  DISPLAY "There are no more entries for that Order."
ELSE
  PERFORM SQL-ERROR
ELSE
IF SHIPMENT-DATE LESS THAN START-DATE OR
  SHIPMENT-DATE GREATER THAN END-DATE
THEN
  NEXT SENTENCE
ELSE
  DISPLAY "Purchase Order Number = ",ORDER-NO
  DISPLAY "Part Number      = ",PART-NO
  DISPLAY "Quantity Ordered  = ",QUANTITY
  DISPLAY "Date Part Shipped  = ",SHIPMENT-DATE.
```

# appendix c: comparison of datatypes



ALLBASE/SQL	Description	Oracle
INTEGER, INT	Four-byte integer, 31 bits and a sign	NUMBER(10)
SMALLINT	Two-byte integer, 15 bits and a sign	NUMBER(6)
DECIMAL(p,[s])	Stores decimal floating point numbers up to a max of 32 significant digits. P is the number of significant digits and s is the number of digits to the right of the decimal point	NUMBER(p,[s])
FLOAT[(p)]	Stores double-precision floating point numbers with up to 16 significant digits. Float corresponds to the Double datatype in C. p specifies the precision of a FLOAT datatype and must be a whole number between 1 and 14	NUMBER or FLOAT or FLOAT(b)
CHAR(n)	A fixed length string of exactly n characters, blank padded. It requires 1 byte per character. 1 <= n <= 3996	VARCHAR2(n) 1 <= n <= 2000
VARCHAR(n,r)	Varying length character string. N is the maximum size of the column and r is the minimum amount of space reserved for that column. 1 <= n <= 3996	VARCHAR2(n)
BINARY (n)	Stores any kind of binary data in an undifferentiated stream. 1 <= n <= 3996	LONG RAW
VARBINARY (n)	Variable length BINARY datatype	LONG RAW
LONG BINARY (n)	Stores any kind of binary data in an undifferentiated stream. 1 <= n <= 2 <sup>31</sup> - 1	LONG RAW
LONG VARBINARY	Variable length LONG BINARY data type	LONG RAW
DATETIME	Stores an instance in time expressed as a calendar date and time of day. It can be defined with qualifiers to specify the precision	DATE
DATE	DATE is stored internally as yyyy-mm-dd	DATE
TIME	TIME is stored internally as HH-MM-SS	DATE
INTERVAL	Stores a value that represents a span of time	NUMBER



**All brand and product names are trademarks or registered trademarks of their respective companies.**

**notice**

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

**Restricted Rights Legend**

Use, duplication, or disclosure is subject to restrictions as set forth in contract subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause 52.227-FAR14.

Hewlett-Packard Company  
3000 Hanover Street  
Palo Alto, CA 94304, USA

© Copyright Hewlett-Packard Company 2006.  
All rights reserved. Reproduction, adaptation or translation of this document is prohibited without prior written permission of Hewlett-Packard Company, except as allowed under the copyright laws

5981-3062EN rev. 3 (9/07)