# A Comparison of TurboIMAGE and HPSQL

by Larry Kemp, HP Bellevue, WA

This paper is intended as a primer on HPSQL for current users of the IMAGE database management system on HP3000 computers. SQL, which is an acronym for Structured Query Language, is the new relational database management system for the HP3000 family. SQL was originally implemented on IBM mainframes, and has since been implemented on several other computer systems. SQL is an implementation of the original "System R" specification for relational databases. The ANSI committee has accepted SQL as the relational database model.

Users of TurboIMAGE will find that HPSQL provides considerably more flexibility than does TurboIMAGE. IMAGE has probably gained most of its popularity due to its ease-of-use and simplicity of design aspects. SQL should provide even more ease-of-use and simplicity.

IMAGE has gained popularity due to its good performance, predominantly to do with the ease with which the designer can take performance into account. For example, the IMAGE designer can effectively, easily, and accurately utilize blocking factors.

Another area where IMAGE excels is having a considerable knowledge and experience base. IMAGE is installed on all HP3000 computer systems, and IMAGE is the database management system used for most HP3000 applications. Therefore, there is considerable expertise available on good IMAGE design, both from HP and from a large number of third party consultants. The IMAGE handbook exemplifies the public knowledge base. There are a number of well known implementation (and optimization) techniques for IMAGE.

There is a knowledge base for SQL, and for the most part that knowledge focuses on high level design issues. There are well documented logical database design techniques that utilize relational database constructs, one example which is the normalization of databases to "third normal form".

The last, very positive trait of IMAGE has been its reliability. IMAGE databases rarely, if ever have integrity problems. And when some damage does happen, there are accurate, if not time consuming, recovery techniques. Since SQL is new, its reliability remains to be seen. SQL does have automated logging and rollback recovery, so SQL databases should not have integrity problems.

The remainder of this primer will focus on the usage and features of IMAGE and SQL on a sample database and problem. I will focus on data structure and design, query (data manipulation) language, program-and-data independence, security, and transaction management. I feel that these are the reasonings for databases.

## Structure.

IMAGE and HPSQL use different terms to describe database structure. IMAGE uses the term "sets" to describe logical groupings of like described data. A non-database user would call that construct a file, with a restriction that all of the records are of the same record-layout. An SQL user calls that construct a "table". The IMAGE user refers to repetitive occurances in the set as "entries", while the non-database user refers to that construct as records.

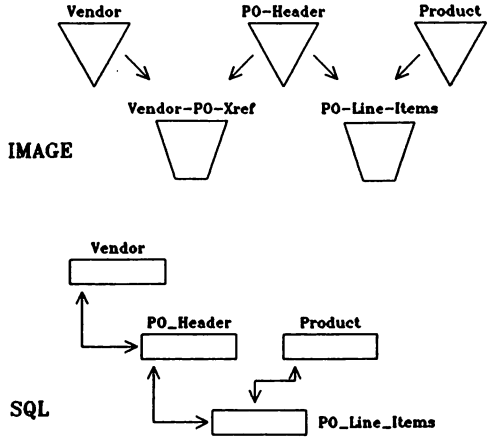
The SQL user refers to those constructs as "rows". And lastly, the IMAGE user refers to the individual components of an entry as "data items", where the non-database user refers to them as fields. The SQL user refers to "columns".

| Non-Database | IMAGE | SQL |
|---|---|---|
| File | Set | Table |
| Record | Entry | Row |
| Field | Item | Column |

IMAGE datasets are defined as one of master datasets, or detail datasets. Master datasets have unique keys and can be accessed by key or sequentially. Entries in a detail dataset are chronologically organized by common key. Entries can be accessed either sequentially, or along the chronological key path. Master datasets can be related to details, and in a logical sense, detail datasets can be related to masters. This results in the definition of IMAGE has an extended two level hierarchy.

SQL makes no distinction of master versus detail datasets. Any two tables can be related, allowing multi-level "Join" operations. And any table can be accessed either by key/path, or sequentially. Furthermore, a given table can have multiple keys, including keys which are formulated from several columns. Generic and approximate searches are allowed.

Here is an example implementation using the two database management systems:




The most noticeable difference between the two implementations is the lack of an artifical connecting dataset between Vendor and PO-header in the SQL database. Just as worthy, is that the SQL database implements the same functionality as the IMAGE version. SQL, as in IMAGE, has the ability to declare unique keys for both the Vendor and PO-header tables. Also, the PO-number index for the PO-line-items dataset can be declared "clustering", which

allows optimized physical placement along that index, in an analogous technique to the "primary path" for IMAGE detail datasets.

Query Language.

Most users of IMAGE were introduced to IMAGE through QUERY. QUERY has a simple, English-like syntax that allows command driven access to IMAGE databases. Query is a good learning tool in that it is easy to learn, and allows exercising of most IMAGE functions. Once the novice has mastered QUERY, he/she next learns how to programmatically access IMAGE. This involves formatting subroutine calls to IMAGE. One record is accessed at a time, with one or two calls necessary to access each record.

SQL, like IMAGE, has an ad-hoc program for accessing databases. (The SQL program is called ISQL, where the I stands for Interactive.) Most SQL users learn SQL through ISQL, in an analagous manner to the IMAGE user with QUERY. But unlike the IMAGE, programmatic access to SQL is nearly identical to ISQL access. In other words, the user codes the same commands programmatically as he/she uses in ISQL. Consequently SQL is easy to learn.

Like Query, SQL allows conditional specifications of rows to be selected. And like Query, SQL uses that specification to determine the access method. The access method is determined by SQL, and not by the application program.

SQL implements the query language in COBOL and PASCAL by using pre-processors. These pre-processors translate the high-level query commands into the appropriate subroutine calls. The only difference between the interactive commands and the programmatic SQL commands are the specification for where the resulting data resides. (Programmatically, the INTO clause is specified which says where in the program to store the result of a command.)

Here are sample interactive and programmatic SQL commands:

```
SELECT NAME FROM VENDOR                    WHERE VENDOR_NUMBER = '0023'

SELECT NAME FROM VENDOR INTO :WS-NAME WHERE VENDOR-NUMBER = :WS-VENDOR-NUM
```

In this example, NAME is a column in the table VENDOR. A row with the VENDOR-NUMBER equal to the value of WS-VENDOR-NUM in the COBOL program is selected. And from the selected row, NAME is delivered to the COBOL item WS-NAME. Notice that the only significant difference is the specification of program data names in the programmatic version. (You might also notice the substitution of '-' for '_'. SQL syntax wants an underscore, while COBOL wants dashes, so the preprocessor converts dashes to underscores.)

Additionally, SQL commands have the ability to retrieve multiple records in a single command. This eliminates the need to code loops in many transaction processing programs. (It also speeds up performance, since it reduces the number of entries and exits from SQL.)

Using the sample database, here are code comparisons for IMAGE versus SQL. These statements display a purchase order. The SQL code is actual code, where the IMAGE code is psuedo-code. Notice that the SQL version takes exactly one SQL command to retrieve all qualifying rows.

```
SQL:  BULK SELECT * INTO :PO-RECORDS
        FROM PO-HEADER,VENDOR,PO-LINE-ITEMS,PRODUCT
        WHERE PO-HEADER.VENDOR-NUMBER=VENDOR.VENDOR-NUMBER AND
              PO-HEADER.PO-NUMBER=PO-LINE-ITEMS.PO-NUMBER AND
              PO-LINE-ITEMS.PROD-NO=PRODUCT.PROD-NO AND
              PO-HEADER.PO-NUMBER = :WS-PO-NUMBER

IMAGE:    DBGET(MODE7,PO-HEADER,PO-NUMBER,PO-RECORD)
          DBGET(MODE7,VENDOR,VENDOR-NUMBER,VENDOR-RECORD)
          DBFIND(PO-LINE-ITEMS,PO-NUMBER)
          REPEAT
            DBGET(MODE5,PO-LINE-ITEMS,LINE(I))
            DBGET(MODE7,PRODUCT,PROD-NO,PROD(I))
            ADD 1 TO I
          UNTIL (END-OF-CHAIN(PO-LINE-ITEMS))
```

This particular example is a complex one, requiring accesses to four different data sets or tables, and locating multiple records from the PO-LINE-ITEMS dataset or table. Note that the SQL user can test out his/her query interactively, using ISQL, before coding the command.

Program and Data Independence.

One of the most significant advantages of a database system is the ability to change the database without affecting the executing programs. All database systems have this characteristic to some extent, really none completely implement it. (One example is where a program accesses a field that has been eliminated from a database.)

IMAGE allows addition and deletion of fields of a database by a database administrator. IMAGE allows changing of field definitions by the database administrator such that programs that do not access the changed fields need no modifications.

The mechanism that IMAGE uses to implement this feature is called access by "item list". Specifically, when a program asks IMAGE for data, it presents a buffer, and a symbolic list of data items that describe the items that should fill the buffer. For example, a program might present IMAGE with the item list "VENDOR-NUMBER,VENDOR-NAME".

IMAGE databases are at least initially created by a text file called a "schema". A database administrator creates the schema which defines all sets, items, relationships, and security. Subsequent structural changes can be made to the database by modifying the schema, and recreating the database, or by use of Adager, or a similar utility which recreates the affected datasets. In all cases, the changes are made offline, and the database administrator will probably want to maintain the schema file.

Like IMAGE, SQL provides item flexibility by having programs request data using an item list. SQL also provides a significantly greater degree of program and data independence through a construct call a "View". A VIEW is a logical window that a program uses to access the database. A VIEW might be construed as a logical 'table' in that a program accesses a view just as it might access a table. A view can contain join operations across multiple files.

Here is an example of a VIEW:

VIEW creation:

```
CREATE VIEW PURCHASE_ORDER (PO_NUMBER,VENDOR,AMOUNT) AS
  SELECT PO_HEADER.PO_NUMBER,VENDOR_NAME,AMOUNT
    FROM PO_HEADER,VENDOR
    WHERE PO_HEADER.VENDOR_NUMBER=VENDOR.VENDOR_NUMBER
```

VIEW access (which could be programmatic):

```
SELECT * FROM PURCHASE_ORDER WHERE PO_NUMBER = '1020'
```

The VIEW facility allows external specification of not only the data elements accessed by a program, but also the access path to the data. It allows a program to retrieve data with no knowledge of the access path. It also allows the access path to be changed without requiring alterations to the program.

SQL databases are maintained by SQL commands. These can be given interactively or programmatically, just as any other command. Physical database structure changes can be made while the database is in use. For example, the following command could be given while the specified table is in use:

```
ALTER TABLE VENDOR ADD CLASSIFICATION CHAR(2)
```

This command would add a new column CLASSIFICATION to the table VENDOR. Currently executing programs would not be affected.

The following command could also be given while the database is in use:

```
CREATE INDEX PO_LINE_ITEM ON PO_LINE_ITEMS(PO_NUMBER,PART_NUMBER)
```

This command creates a combined index for the table PO_LINE_ITEMS using the columns PO_NUMBER and PART_NUMBER. Applications using the original database and selecting on PO_NUMBER and PART_NUMBER would have used the PO_NUMBER index, and then searched sequentially for the PART_NUMBER. Now those applications can use the new index PO_LINE_ITEM to go directly to requested line item. This change in access method is transparent to application programs.

**Security.**

There is little doubt in the industry today that security is an important job of a database management system. Ad-hoc programs, third-party applications, and open computer systems have mandated externally managed security systems.

IMAGE implements security in the form of passwords. Data items and data sets are passworded for a combination of read/update/none access to data items and read/write/none access to data sets. Passwords are specfied by the application program when it opens the database.

Security in SQL is implemented through the granting of access rights to logon user ids. Rather than use a separate password, SQL uses the user logon id, and allows MPE security to be used for passwording. Access is granted against tables or views. Since access to elements can be restricted by using views, data element security is achieved. Hopefully, this will prove to be a simpler technique.

A view, however, is more than simply a subset of data. It can contain not only access specification, but also selection criteria. Since access to data can be granted on views, this allows security to be specified by value. For example:

```
CREATE VIEW PO23 AS
   SELECT * FROM PO_HEADER WHERE VENDOR_NUMBER = '0023';
GRANT SELECT ON PO23 TO VENDOR23@PURCH;
```

This view allows the user VENDOR23 in the account PURCH to look at only his own purchase orders in the PO_HEADER table.

Transaction Management.

One of the functions of a database management system is to coordinate data between concurrent users. There are two issues: (1) protection against "race conditions" where multiple users desire to access and update the same data, and (2) guaranteeing logical integrity of data. A database management system protects against race conditions by serializing access to the same data. And a database management system guarantees logical integrity by ensuring that either all of its database manipulations succeed, or none of it succeeds.

For example, a user is going to make a transaction which adds one part to inventory, and subtracts one part from a purchase order. The increment to inventory includes reading the data and then updating it. No other updating transaction can be allowed to intervene between the read and update. If an intervening transaction did update the inventory count, then this transaction would make its changes to inventory using the old inventory count, effectively undoing the other transactions inventory update. In the case of system or program failure, the transaction must either have completed, or must be backed out. Otherwise, the partially completed transaction might allow artificial inventory growth.

IMAGE has two facilities to address transaction management: Locking and Transaction Logging. Locking allows programs to logically reserve a specified item before making a transaction against it. Locking is done explicitly by the program. Transaction logging allows a program to declare the beginning and ending of a logical transaction. In the case of system failure, the database can be recovered to last consistent (logically complete) point before the failure. Here is an example of inventory receivings using the sample database:

```
DBLOCK(product.prod-no=2666,po-line-items.prod-no=2666)
  DBGET(product,prod-no=2666)
  REPEAT
     DBGET(po-line-items,prod-no=2666)
  UNTIL (po-number=A2345)
  DBBEGIN
    DBUPDATE(product,qty-on-hand)
    DBUPDATE(po-line-items,qty-received)
  DBEND
DBUNLOCK
```

In this example, the program locks the item PROD-NO in both the PRODUCT and PO-LINE-ITEMS datasets. Then it retrieves the requisite entries. Once the

entries are found, then a logical transaction is started which updates quantity fields in both datasets.

SQL implements the same constructs, but using a more automated technique. Locks in SQL are implicit; the programmer never needs to code LOCKs into a program. SQL determines concurrency conflicts by examining the data "pages" (which are similar to blocks) accessed within a transaction. If one transaction conflicts with another transaction, then SQL will either wait for the other transaction to complete, or return an error, allowing the transaction to restart itself.

This technique is not only eaiser to use, but it can also be more efficient. For example, a transaction which updates a bill-of-materials has no idea at the start of the transaction which part-numbers to lock, since the parts-explosion is determined by reading the records to be updated. An explicit locking technique would require either data set locking, or double accesses to the parts dataset. The SQL technique allows maximum concurrency since it does not require pre-determined locking.

Here is the SQL version of the parts receiving problem:

```
BEGIN WORK;
  UPDATE PRODUCT SET qty_on_hand = qty_on_hand + 1
    WHERE prod_no = '2666';
  UPDATE PO_LINE_ITEMS SET qty_received = qty_received + 1
    WHERE prod_no = '2666' AND po_number = 'A2345';
COMMIT WORK;
```

SQL assures logical consistency of data using a similar technique to IMAGE: each transaction is bracketted by the commands BEGIN WORK and COMMIT WORK. After a system failure, the uncompleted transactions are backed out in a manner similar to IMAGE. SQL provides the additional feature that if a program aborts, that any incomplete transactions will be rolled back.

A transaction roll-back can also be programmatically initiated by the ROLLBACK WORK command. This feature can simplfy, and potentially optimize transaction processing programs. For example: a program to fill sales orders might first match requested line items against inventory to see if the order could be filled. If the order can be satisfied, then it would re-read, and update the records from inventory. The SQL version of this program would simply read and update inventory. If a line item could not be satisfied from inventory, then it would request a rollback.

**In Summary.**

SQL provides all of the features of IMAGE, and in most cases in a significantly enhanced fashion. Additionally, SQL allows the user to administer databases at a considerably higher level. With SQL, the database administrator has a high degree of program independant control over data and access paths. In essence, SQL has provided the database adminstrator with many of the tasks that require programming (and debugging) in IMAGE.

The simplicity of IMAGE has resulted in very good performance for well designed IMAGE databases. IMAGE performance is well understood and reasonably consistent.

On the positive side of performance for SQL is that when a database performance issue arises after an application has already been implemented, that the database administrator can take action without involving changes to program logic. In other words, SQL allows the database designer to make mistakes in the initial design, and correct them after the fact.

**References.**

Astraham, M. M., et al, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems 1*, No. 2 (June 1976).

Codd, E. F., "Normalized Data Base Structure: A Brief Tutorial", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control. Available from ACM.

Codd, E. F., "Further Normalization of the Data Base Relational Model", in *Data Base Systems*, Courant Computer Science Symposia Series, Vol 6, Prentice Hall (1972).

Russell, Marguerite (ed.), *The IMAGE Handbook*, Wordware(1984), Seattle, WA.

Also, TurboIMAGE and HPSQL Reference Manuals from Hewlett-Packard Co.

**Comparison of Limits.**

| | TurboImage | HPSQL |
|---|---|---|
| Sets/Tables per Database | 199 | Unlimited |
| Items/Columns per Database | 199 | Unlimited |
| Items/Columns per Dataset/Table | 255 | 64 |
| Item Size | 4094 | 3996 |
| Items/Columns per Path/Index | 1 | 15 |

**Sample SQL/COBOL program.**

This program prompts the user for a product number, and displays all purchase orders against that product, including which vendor that the purchase order was issued to. Each SQL statement is bracketted by EXEC SQL/END-EXEC, to signal to the pre-processor that these are SQL commands. Notice that the pre-processor also knows the data-division elements, allowing it to check on data types and lengths.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. POS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  I PIC S9(4) COMP.
EXEC SQL INCLUDE SQLCA END-EXEC.              <<SQL communication area>>
EXEC SQL BEGIN DECLARE SECTION END-EXEC.      <<SQL data elements>>
01  PURCHASE-ORDERS.
    05  PURCHASE-ORDER OCCURS 20 TIMES.
        10  PO-NUMBER PIC X(6).
        10  VENDOR-NAME PIC X(20).
        10  QUANTITY PIC S9(4) COMP.
01  PROD-NO PIC X(4).
01  ERROR-MSG PIC X(72).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
OPEN-DATABASE.
    EXEC SQL WHENEVER SQLERROR GO TO SQL-ERROR END-EXEC.
    EXEC SQL CONNECT TO 'PURCHDB' END-EXEC.
ASK-FOR-PART-NO.
    DISPLAY 'ENTER PROD-NO FOR INQUIRY, OR CR TO STOP'.
    MOVE SPACES TO PROD-NO.
    ACCEPT PROD-NO.
    IF PROD-NO EQUAL SPACES THEN GO TO CLOSE-DATABASE.
    EXEC SQL BULK SELECT PO-LINE-ITEMS.PO-NUMBER,VENDOR-NAME,QUANTITY
              INTO :PURCHASE-ORDER
              FROM PO-LINE-ITEMS,VENDOR
              WHERE PO-LINE-ITEMS.PO-NUMBER=PO-HEADER.PO-NUMBER AND
                    PO-HEADER.VENDOR-NUMBER=VENDOR.VENDOR-NUMBER AND
                    PO-LINE-ITEMS.PROD-NO = :PROD-NO END-EXEC.
    IF SQLCODE GREATER THAN 0 THEN DISPLAY "NO POS FOR THAT PART-NO"
    ELSE PERFORM DISPLAY-PO VARYING I FROM 1 BY 1 UNTIL I > SQLERRD (3).
    GO TO ASK-FOR-PART-NO.
DISPLAY-PO.
    DISPLAY "PO-NUMBER=" PO-NUMBER (I)
         "   VENDOR-NAME=" VENDOR-NAME (I)
         "   QUANTITY=" QUANTITY (I).
SQL-ERROR.
    EXEC SQL SQLEXPLAIN :ERROR-MSG END-EXEC.
    DISPLAY ERROR-MSG.
CLOSE-DATABASE.
    EXEC SQL RELEASE END-EXEC.
    STOP RUN.
```