SLAC PUB-5504
March 1991
(E/I)

Jazelle

History, Status and Future Plans[*]

A.S. Johnson
Dept. of Physics, Boston University, 590 Commonwealth Avenue, Boston, MA 02215, U.S.A.

H. Hissen
Department of Physics, University of Illinois, 1110 W. Green Street, Urbana, IL 61801, U.S.A.

G.B. Word
Department of Physics, Rutgers University, Piscataway, NJ 08855-0849, U.S.A.

M. Breidenbach, P.F. Kunz and D.J. Sherden
Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94309, U.S.A.

ABSTRACT

The data management system Jazelle has been created as a successor to earlier HEP data managers such as YBOS and ZEBRA. While it has many similarities with these systems, it also has many enhancements such as self-documenting data descriptions, mnemonic access to all data, relational data structures, powerful machine-independent IO facilities, including network IO, and many mechanisms for presenting data to the physicist in an intuitive manner. The emphasis has been on producing a powerful, user friendly, data management system which can be accessed from many languages as a natural extension of those languages.

Invited talk presented at the INFN Eloisatron Project 14th Workshop: Data Structures for Particle Physics Experiments, Erice, Italy, Novermber 11-18, 1990.

[*]Work supported in part by the Department of Energy, contracts DE-AC03-76SF00515 and DE-AC02-89ER40509 and by the National Science Foundation.

1 Introduction

1.1 Why was Jazelle created?

The data obtained from a typical High Energy Physics experiment represents the result of many thousands of man years of effort and the expenditure of tens of thousands of dollars. The experiment's ability to manipulate and exploit these data is essential to the timely production of physics results. The data include both raw and reconstructed event data, calibration data such as detector gains, drift velocities, etc., and data which describe the experiment, such as detector geometry descriptions. During analysis and reconstruction, the data must typically be processed through several programs, often moved between several different types of computers, and at all stages the experimenters must have easy and efficient access to the data. The various types of data may differ dramatically in size, from several hundred kilobtyes of raw data per event, to several bytes per event in the final stages of analysis. Therefore, an essential part of any High Energy Physics experiment's software should be a data management system capable of handling these varied needs.

When the SLD experiment began, we looked for a system which would be able to handle all of these requirements. We looked at many different languages, but, while many existing languages have very powerful mechanisms for defining and manipulating data structures within a program, they do not, in general, have good facilities for moving data between programs or for providing easy interactive access to the data. Similarly, although there are many commercial database programs available, they are in general not well suited to handle the range of different data types and sizes needed and do not provide sufficiently efficient and easy to use program interfaces.

Finally, we looked at other data management systems developed specifically for HEP applications, specifically ZBOOK, ZEBRA[1] and YBOS.[2] While these systems are well suited to handle the data processing needs of HEP experiments, they generally fall short in the area of user friendliness, forcing users to memorize numeric offsets within data structures, thus making programs rather hard to read and understand.

For these reasons it was decided to develop an entirely new data management system for the SLD experiment, called Jazelle. Jazelle has been in use by SLD now for over four years and currently runs on both IBM/VM/XA and VAX/VMS operating systems. In addition, work is underway to port it to several UNIX-based workstations.

1.2 Main Features of Jazelle

Jazelle has many similarities with the earlier HEP data managers mentioned above and has adopted many of the best features of these systems. Data are stored in structures called *banks*. Banks may be dynamically created, modified and destroyed during program execution and may be read in or written out from programs. Some of the more novel features of Jazelle include:

- The structure of each type of Jazelle bank is defined in a *template file*. In the file each element is typed (real, integer, etc.) and named. The template file allows a description to be attached to each element and therefore fulfills an important documentation function as well. Elements of different types can be freely mixed within banks.
- Jazelle has been interfaced to several languages including Fortran and C. All access to data in Jazelle banks is by name. Particular care has been taken to produce easy to use and efficient language interfaces.
- Jazelle contains utilities for presenting data to the user in many different formats. The name of each element and the description from the template can optionally be included along with the data itself.
- Jazelle is fully integrated into SLAC's interactive data analysis program, IDA.[3] A complete set of interactive commands are available for manipulating and viewing Jazelle structures, and data stored in Jazelle banks can be freely accessed from IDA's interactive data analysis language, IDAL.[4]
- Debugging facilities are provided by making the entire set of interactive commands also available from within the VMS and VM debuggers.
- Jazelle contains facilities for describing data structures using a relational technique, in addition to the more common hierarchical data descriptions.
- Jazelle obviates the need for the large pre-assigned common block found in earlier systems by using the virtual memory services of the host operating system. Data can be partitioned into virtual memory zones (called *contexts* in Jazelle).
- Jazelle includes facilities to handle input and output of data to both sequential and indexed files. Data written to disk or tape on VM or VMS can be read on either system.
- The VMS version of Jazelle includes features, such as asynchronous inter-process I/O, which are useful in connection with online systems.

• INTEGER [*4|*2]	• STRING [*1|*4|*8]	• ENUM [*4|*2]
• HEX [*4|*2	• COMPLEX	• BITS [*4|*2]
• REAL [*4|*8]	• POINTER	
• LOGICAL [*4|*1]	• KEY	

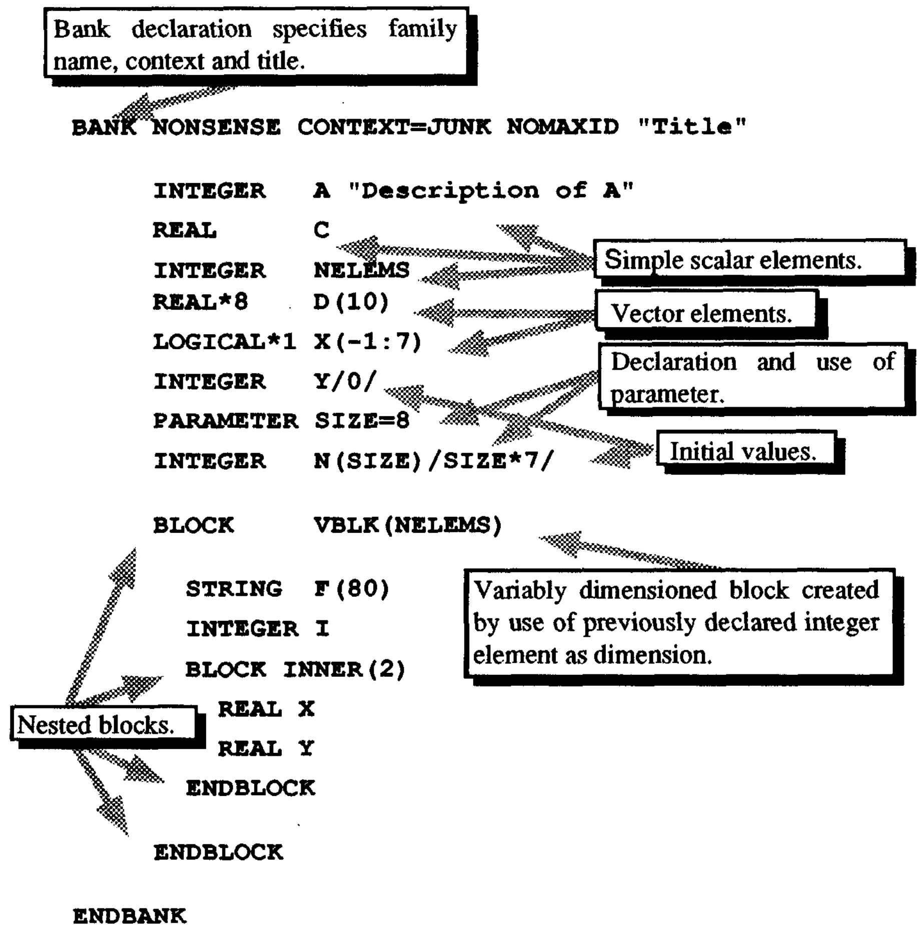
Figure 1. Element Types Supported by Jazelle
All types may be used as scalars or vectors (fixed or variably dimensioned). Elements of different types may be free mixed within banks. Jazelle also supports user-defined types. The brackets are used to denote optional size modifiers.

2 Using Jazelle banks

2.1 Defining Data Structures - Jazelle Templates

Jazelle data structures are composed of banks. A bank is a contiguous piece of memory consisting of two parts: a 16-byte header section containing Jazelle system data and a user area where the user data associated with the bank are stored. The user area contains a sequence of elements which is defined in a user supplied template file. The various types of elements supported by Jazelle are listed in Figure 1.

The syntax of the template file allows each element within the bank to be named and typed, and also allows for initial values and descriptions to be attached to each element. Well written templates contain all the information needed to document the data structure. Figure 2 shows an example template.


Bank declaration specifies family name, context and title.
Simple scalar elements.
Vector elements.
Declaration and use of parameter.
Initial values.
Variably dimensioned block created by use of previously declared integer element as dimension.
Nested blocks.

```
BANK NONSENSE CONTEXT=JUNK NOMAXID "Title"

     INTEGER   A "Description of A"
     REAL      C
     INTEGER   NELEMS
     REAL*8    D(10)
     LOGICAL*1 X(-1:7)
     INTEGER   Y/0/
     PARAMETER SIZE=8
     INTEGER   N(SIZE)/SIZE*7/

     BLOCK     VBLK(NELEMS)

       STRING  F(80)
       INTEGER I
       BLOCK INNER(2)
         REAL X
         REAL Y
       ENDBLOCK

     ENDBLOCK

ENDBANK
```

Figure 2. Example Template
An example template file demonstrating many of the different syntaxes that may be used.

Banks may contain arbitrary combinations of data types in any order, including both scaler and vector elements. Banks may also contain constants (declared using a *parameter* statement). In addition, elements within a bank may be grouped into *blocks*. Blocks themselves may be dimensioned to create an arbitrary number of repetitions of the elements within the block. Blocks can also be nested. In addition to constant dimensions, the last element or block in each bank may be given a variable dimension. The template in Figure 2 contains an example of a variably dimensioned block, VBLK. Variable dimensions are created by specifying the dimension of a block or element as an integer element declared previously in the bank (NELEMS in this example). The amount of memory allocated for variably dimensioned elements can be increased or decreased dynamically. This last feature is extremely useful since it allows, for example, an arbitrary number of hits to be stored in a bank describing a track, and for the bank to be expanded indefinitely as new hits are added to the track.

Templates support data types beyond those found in similar systems. "Pointer" and "Key" data types are used to form links between banks and are discussed further in Section 3. "Enum" and "Bits" data types allow mnemonic names to be assigned to each value (Enum) or each bit (Bits) thus allowing code manipulating bit masks or lists of values to be coded in a clear way, with no need for memorizing arbitrary values. In addition, users can define their own data types. Examples of data types defined by SLD include "Time" (8-byte absolute data and time) and "Partid" (LUND particale code).

2.2 Families of banks

Jazelle banks are grouped into *families*, each of which has a unique *family name*. All banks in a family share a single template, and hence a single data structure. If the template includes a variable dimension then the variable dimension can have a different value for each bank within the family. To distinguish banks within a family, each bank is assigned a unique ID in the range 0-65535.

Banks can be referenced either by a family-name/ID pair or, more commonly, by means of a *pointer*. Jazelle pointers are simply variables which point to the memory location at which the

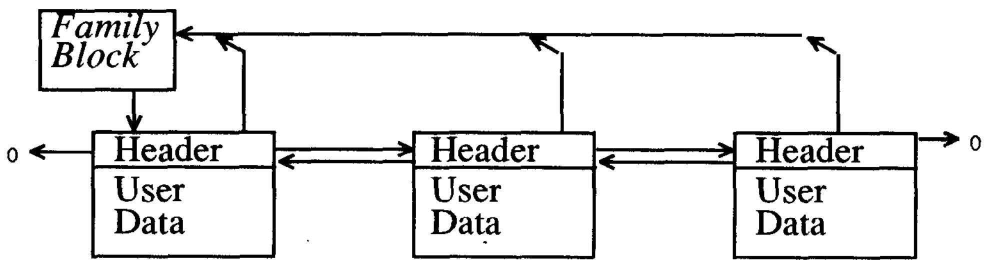


Figure 3. Family

Pointers inside each Jazelle bank's header are used to link the banks within a family into a doubly linked list. Further pointers are used to link each bank back to its family block where information on the banks structure is kept. The bank headers make looping over all of the banks in a family an easy and efficient operation.

bank starts, and are used whenever it is desired to access data from or store data into a bank. A unique feature of the way in which Jazelle pointers are implemented is that their values remain constant so long as a bank remains in existence. In particular, the value does not change when the bank is expanded or deleted, and is never changed due to memory cleanup operations (garbage collections).

Pointers within banks can be used to form links from one bank to another. For example Jazelle automatically keeps pointers inside the bank header updated in such a way as to chain together all of the banks within a family, as illustrated in Figure 3. Similarly, pointers within the user section of the bank can be used to form arbitrary links between banks (discussed further in Section 3).

2.3 Creating and Manipulating Banks

Jazelle provides a powerful set of routines to allow users to create and manipulate banks, as well as to write out and read in banks. Routines can operate on individual banks, families of

JZBADD	Create a bank	**Bank Manipulation Routines**
JZBDEL	Delete a bank	
JZBEXP	Expand/contract a bank	
JZBFND	Find an existing bank	
JZBLOC	Find a family of banks	
JZBCPY	Create a copy of a bank	
JZBDMP	Output (dump) bank(s)	**Routines for Examining Banks**
JZBTBL	Tabulate bank(s)	
JZTDEF	Modify or create tabulation format	
JZIOPN	Open a file for Jazelle IO	**Sequential and Indexed IO Routines**
JZIOCL	Close a file	
JZIOWR	Write a record using a list	
JZIOWC	Write a record using a context	
JZIORD	Read a record	
JZINDX	List existing banks	**Miscellaneous Routines**
JZSTAT	Summarize memory usage	
JZTSCN	Scan a relational table	
JZXWIP	Delete an entire context	
JZPCMP	Compare two banks	

Figure 4. Commonly Used Jazelle Routines

banks, or on arbitrary collections of banks. Some of the more common routines are summarized in Figure 4.

In addition, there are many routines which allow the contents of banks to be examined. In designing these latter routines, considerable attention has been paid to producing easy-to-read dumps based on the information given in the bank's template. Each routine allows the data to be dumped in several different levels of detail: full dumps being suitable for users unfamiliar with the contents of the bank being dumped, with progressively more abbreviated dumps for users with greater familiarity with the bank.

Fortran

```
Subroutine FILLPTOT(MCPART)
Pointer MCPART-->MCPART
....
PTSQ=0

Do I=1,3
  PTSQ=PTSQ+MCPART%(P(I))**2
Enddo

MCPART%(PTOT) = SQRT(PTSQ)

Return
End
```

Mortran

```
$sld_integer_function FILLPTOT
    (Pointer MCPART-->MCPART);
....
PTSQ=0;

Do I=1,3
[
  PTSQ=PTSQ+MCPART%(P(I))**2;
]

MCPART%(PTOT) = SQRT(PTSQ);

$Return;
End;
```

C

```
FillPtot(Mcpart)
jpointer Mcpart;
...
{
  int ptsq=0;
  int i;

  for (i=0 ; i<3 ; i++ )
    ptsq += _Mcpart->P(i) *
            _Mcpart->P(i) ;

  _Mcpart->PTOT = sqrt(ptsq);

}
```

IDAL

```
Def FILLPTOT(MCPART)
  Pointer MCPART-->MCPART

  PTSQ=0

  Do I=1 to 3
    PTSQ=PTSQ+MCPART%(P(I))**2
  Enddo

  MCPART%(PTOT) = SQRT(PTSQ)

Enddef
```

Figure 5. Example Program In Four Languages using Jazelle.

2.4 Accessing Data in Jazelle Banks

One of the main aims in designing Jazelle was to make it fit elegantly into the programming language(s) used to write code for the experiment: in effect to give the user the impression that access to Jazelle data is a natural extension of the language. SLD chose to use an existing pre-processor, MORTRAN, and to extend it to provide access to Jazelle[5]. However, the same features can be made available in almost any language, and Jazelle has so far been interfaced to Fortran-77, C, and IDAL, a language designed specifically for physics analysis. A simple program making use of Jazelle from four different languages is shown in Figure 5.

The interface to Fortran is by way of a pre-processor that converts the Jazelle language references into standard Fortran code. In the case of C, no pre-processor is required since the Jazelle structures can be mapped directly onto C-structures and the native C syntax can be used to access data from Jazelle banks. It is hoped that in the future it will be possible to design a Fortran-90 interface similar to the current C interface.

Note that in all language interfaces, access to Jazelle data is achieved using purely in-line code. No function or subroutine calls are necessary to access the data. Care has been taken to generate target language expressions which do not inhibit the normal ability of the compiler to optimize the code. Real Fortran applications making extensive use of Jazelle show that the typical CPU time overhead incurred by using Jazelle as opposed to common blocks is only about 5%.

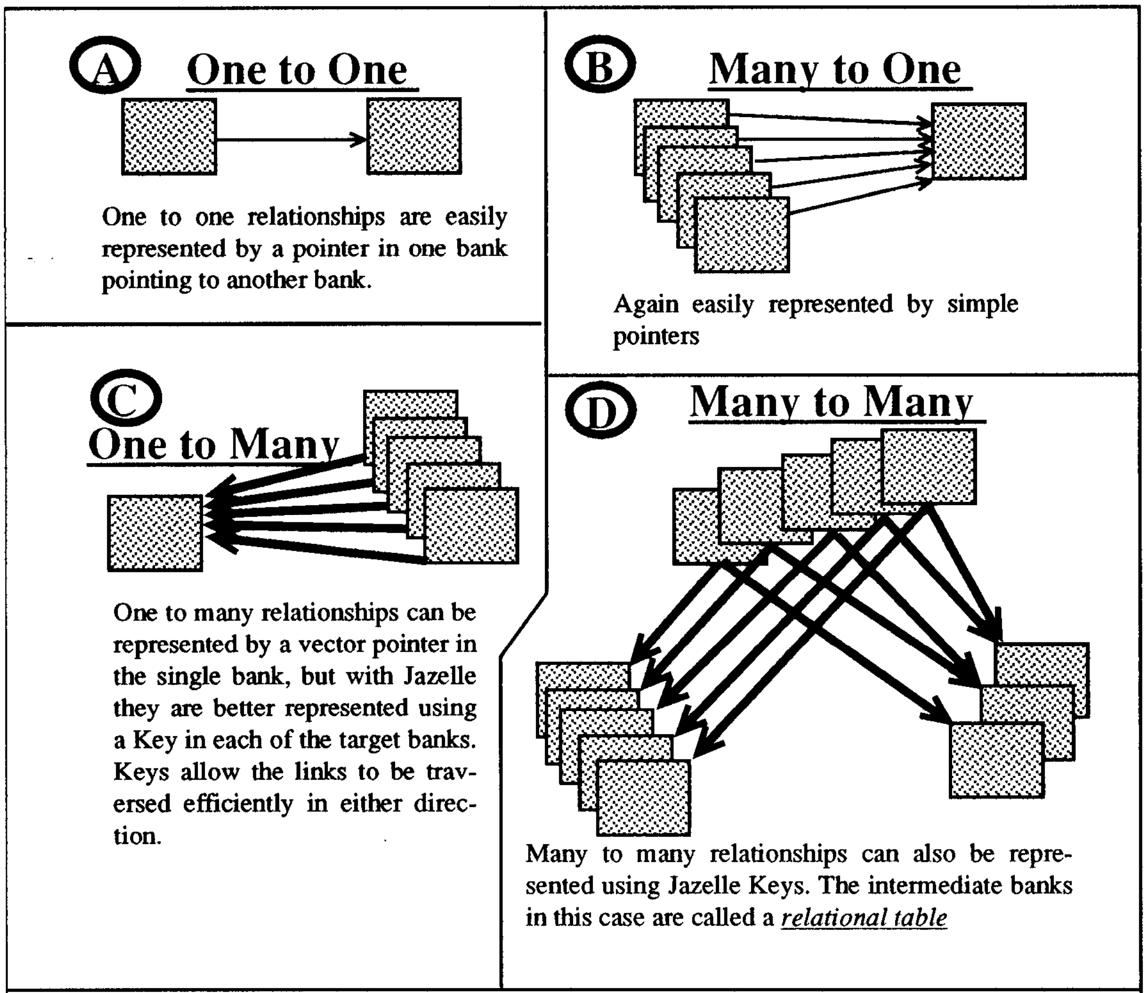
For those language interfaces that make use of a pre-processor, one concern is that the pre-processor will form a barrier to the use of symbolic debuggers. There are two aspects to this problem. Firstly, the code seen by the user in the debugger will not correspond exactly to the code written by the user and, secondly, the debugger will not be able to access data stored in Jazelle banks. To overcome the first problem we have been careful to restrict the Jazelle pre-processor to making only local (within one line) changes to the code, and to always produce a comment in the output code indicating the original source expression. In addition, it has been possible with both the VM and VMS debuggers to interface Jazelle to the debugger so as to allow the direct examination and modification of Jazelle data structures and pointers from within the debugger.

3 Using Jazelle to Build Complex Data Structures

There are many reasons to keep individual banks small and to compose complex data structures by linking together many small banks. Jazelle provides mechanisms for linking banks together to make arbitrarily complex data structures, as illustrated in Figure 6.

The simplest of these mechanisms is to imbed pointers to banks inside other banks. A typical use of pointers is to create hierarchies of banks with a top level bank pointing to other banks which contain more detailed information. For example, in the SLD DST structure (Figure 7) a particle summary bank (PHPART) contains pointers to lower level banks which contain detailed information from each of the detector elements in which the particle was reconstructed. Some of these in turn point to still lower level banks containing even greater detail. Arranging the data in this way, rather than in one huge bank, allows efficient representation of the fact that not all particles are seen in all detector elements; unnecessary banks are simply not created and the pointer which would otherwise point to them is assigned a null value. In addition, such a structure allows the user to drop any information which is not required for a particular analysis.

A second and more innovative way of representing relationships between Jazelle banks is by the use of relational tables. A relational table is a family of Jazelle banks whose structure contains one or more elements of type *Key*. Within a relational table, each bank represents a relationship, with the key(s) pointing to the bank(s) being related. The bank containing the key(s) is also able to store additional information about the relationship. For example, the SLD DST structure includes a relational table which relates vertices and particles. Given that vertices are not reconstructed unambiguously, each vertex can be associated with any number of particle, and each particle can be attached to an arbitrary number of vertices. Thus it was required to



One to one relationships are easily represented by a pointer in one bank pointing to another bank.

Again easily represented by simple pointers

One to many relationships can be represented by a vector pointer in the single bank, but with Jazelle they are better represented using a Key in each of the target banks. Keys allow the links to be traversed efficiently in either direction.

Many to many relationships can also be represented using Jazelle Keys. The intermediate banks in this case are called a *relational table*

Figure 6. Types of Relationship Supported by Jazelle

represent a completely arbitrary N by M relationship. The family of banks used to create the particle-vertex relational table is called PHPTVX, as is shown in Figure 7. Properties of the relationship, such as the distance of closest approach of the track to the vertex, are stored in the bank along with the keys which point to the track and vertex banks.

Unlike all other Jazelle element types, keys can not be assigned values directly. Their values are set when the bank containing them is first created and can only be modified by calling a special key-modification routine. This enables Jazelle to create additional hidden links between banks containing keys. One link interconnects all keys which have the same value, while a second link interconnects the first occurrence of each value within a table. These two sets of hidden links make very efficient scans of tables possible. Using Jazelle tools for handling relational tables,

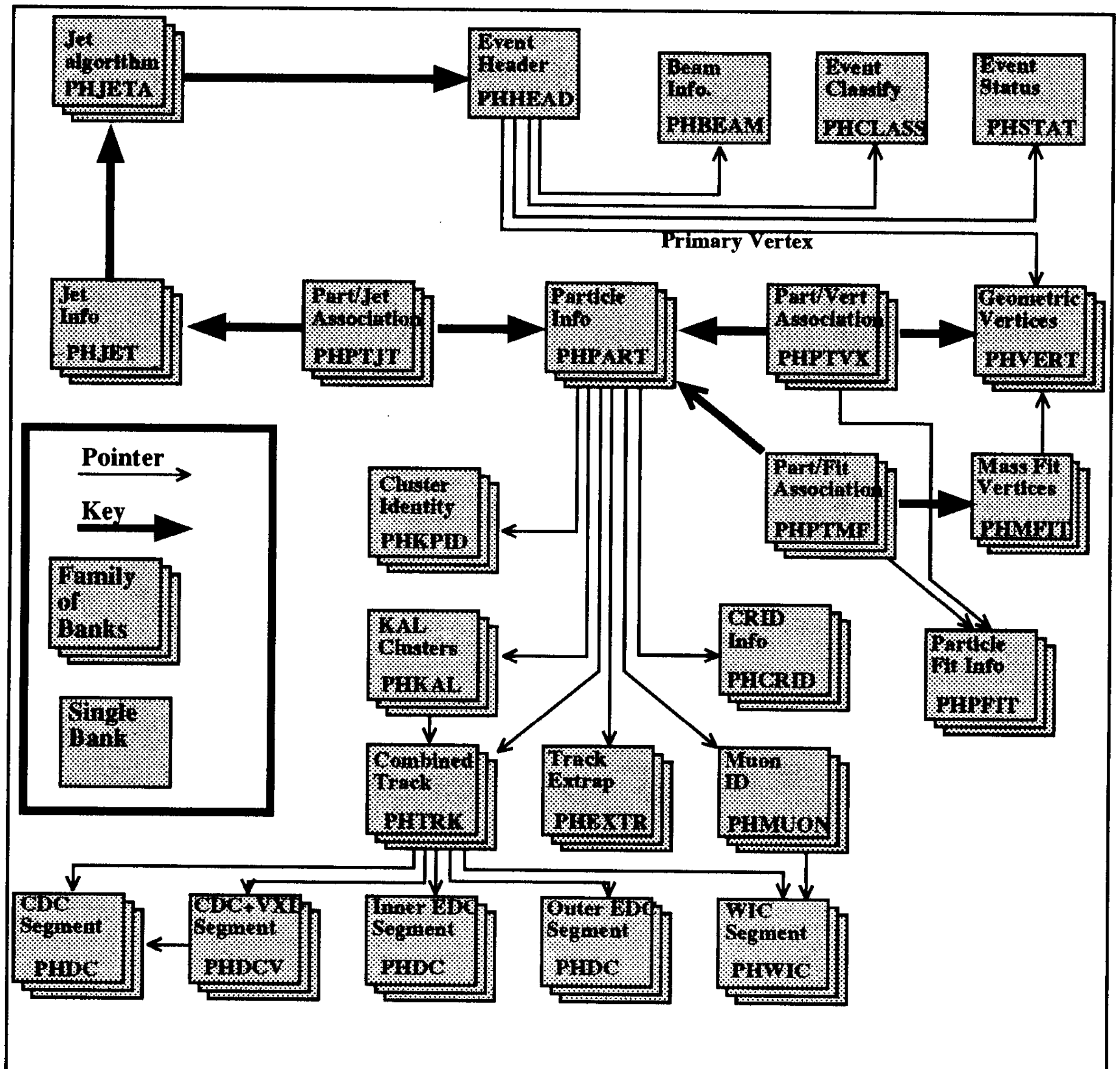


Figure 7. The SLD DST Data Structure

questions such as, "Which particles are attached to this vertex?" as well as the symmetric question, "Which vertices is this particle attached too?" can be answered very easily and efficiently.

By describing relationships with relational tables it is very easy to add additional relationships by simply creating new banks, or to remove existing relationships by deleting banks. Again, this mechanism is well suited to the case of tracks and vertices, where the precise relationship is often built up slowly, with frequent changes and ambiguities.

4 Conclusions and Future Plans

Data managers have in the past been viewed by many people as only necessary to compensate for the inadequacies of older languages such as Fortran. During the development of Jazelle it has become apparent that it is possible to develop tools which have considerable advantages over those provided by any programming language alone. Examples of these advantages are the ability to transport complex data-structures easily and flexibly between programs and between different machines, tools to allow easy examination of the data, and perhaps most importantly the ability to access and manipulate the data interactively from data analysis programs such as IDA and PAW[1].

A further advantage of Jazelle is that it can be used as a tool in multi-language environments. One of the major problems in mixing languages is transporting complex data structures between different languages, since the internal representation of the data is often very different. Jazelle provides a language independent data repository and makes transporting data between different languages much easier.

Jazelle currently consists of approximately 50,000 lines of code, and represents almost 10 man years of effort. Less than 25% of this represents work which could be eliminated in producing an equally functional system based on a more recent high level language. For these reasons some of the authors would like to continue the development of Jazelle by adding support for other languages such as Fortran-90, by porting the system to new environments such as UNIX and by continuing to add new features and improve existing ones. We would particularly like to find new experiments interested in using Jazelle in the upcoming SSC/LHC era.

5 Acknowledgments

We would like to thank all the members of the SLD collaboration for their ideas and patience during Jazelle's development and especially those who have contributed code, in particular D.Aston, W.Ballentyne, C.Boeheim, M.Gravina, Y.Lu, L.J.Moss and S.Sterner.

[1] It has only been possible to cover a few of these topics here, for further discussion the reader is encouraged to consult the Jazelle User Guide, reference 6.

6 References

1. R.Brun, M.Goosens, J.Zoll, CERN DD/EE/85-6 and papers presented at this conference.
2. V.Blobel, DESY-R1-88-01 and papers presented at this conference.
3. T.H. Burnett, Comput. Phys. Commun. **45** (1987) 195-199.
4. A.S.Johnson et al., AIP conference proceedings **209**, 285.
5. A.S. Johnson, Comput. Phys. Commun. **45** (1987) 275-281.
6. A.S.Johnson and D.J.Sherden, SLAC-PUB-263.