1.5 Support Databases

Although event data itself will be stored in files in Root format and not in a database, CDF will use databases to store a large variety of information. This information includes calibration constants, alignment and geometric information, run conditions, trigger tables, production configuration, etc. Some of the information is used only in the online system (for example calibration constants that are downloaded to front-end scanners); other data are used both online and offline (for example constants that are used by the reconstruction software both in Level 3 and production) and still others are of offline interest only. CDF has decided to develop a unified approach to support databases that provides for the needs of the online, the offline and remote sites. This approach allows different databases to share schema where appropriate, but does not require that the full schema be common everywhere.

A discussion of CDF database management requirements can be found in Reference [12]. Based on these requirements, CDF has decided to develop the majority of its support databases using relational tables (while retaining the possibility of using embedding binary data for the very largest database objects, if required to get adequate speed performance). The advantages of this approach are that relational databases support a well know standard query language (SQL), that commercial and freeware tools exist for such databases and that import/export between database implementations is a well defined and understood problem. For onsite applications, CDF will use Oracle as its database management system (DBMS). Offsite applications will use a freeware DBMS (at the moment, we are using mSQL).

The advantages of Oracle for onsite use are that it is robust, supports concurrent write access and lock management, supports database replication, includes backup and restore utilities and provides good access speed. Typically, offsite databases support a much smaller number of users, are largely accessed by readonly applications and in general contain only a subset of the available database scheme and therefore can be adequately handled with a freeware relational database.

In the past six months, significant effort has gone into the design and implmentation of support database database software. The focus of the activity has been to prototype the complete system. The decision was made for this prototype to be done for the Calibration database and to be integrated with the online group's program of characterizing the calorimeter front end electronics. Code that now exists includes front-end scanner applications that produce calibration data in Trybos format, as well as consumer process software that fills the calibration database with data read from the online system. This consumer process allows browsing software to perform fits and displays of calibration data both before database constants are written and after reading constants back from the database. The infrastructure code developed to perform these tasks include a C++ API to access the database, automatic code generation tools to define the database schema and to bind the C++ applications to the database, database tables and Root-base browsing tools.

Joint project activities have had important contributions to this effort. Support activities provided by CD computing professionals include: installation of Oracle servers on two CDF SGI machines, providing general consulting help on database access with Oracle, optimizating the database table design, installing Oracle clients on several SGI machines and providing system administration support for the Oracle database. Dennis Box has written the OCI template code being used for automatic code generation binding to Oracle. The overall design and first implementation of the C++ API was done by Jim Kowalkowski.

Database software issues we expect to address within the Joint Project in the next few months include design of a database object server that connects to the Oracle or mSQL database, development of backup and restore utilities for the database and development of administration and management tools for database support.

1.5.1 Database Table Design

Calibration database objects are fetched by access key. This compound key defines the calibration **attribute** (which specifies the table format) and **component** (which specifies the the relevant detector name) as well as management information such as run number and calibration version number. These keys are used to map an object to a set relational tables. Calibration database entries currently exist for all calorimeter components for two attributes (electronic pedestal and electronic gain). In addition to data tables, the database contains tables that map ChannelID to crate location in the counting house and to physical position on the detector. See [13] for details.

Preliminary studies of database performance with the current table design, C++ API and code generated via the templates described below have been performed both for Oracle and for mSQL. With Oracle. the insert rate is about 3000-4000 rows/sec while the select rate is about 6000-8000 rows/sec. For mSQL, insert and select rates are both 600-1000 rows/sec.

1.5.2 C++ API for Database Access

Access to database constants from C++ applications occurs via a C++ API that has been defined and implemented. This interface hides the application programmer from the fundamental database representation (as relational tables) by presenting the database constants as transient C++ objects. One goal of the API is to insulate reconstruction code from changes in the database storage mechanism (such as database implementation). Another goal is to provide mechanisms for caching constants that may be used in may parts of the code (for example, geometry or alignment data) and for automatically updating the constants when necessary (for example when a run number changes). The design of the code shares many features with the Babar ProxyDict mechanism, although it contains additional management tools.

High level user access to calibration occurs via two templated C++ classes:

DataPointer<calibrationDataObject> provides a proxy to a real calibration object. Every user that specifies the same query key will automatically get the same instance of the object. The calibration object is returned as a proxy to the real thing, implying that the system owns and maintains the real instance of CalibrationDataObject

DataMgr<calibrationDataObject> is used by the application programmer to do gets and puts for objects of class calibrationDataObject. The DataMgr keeps track of which version of the database is being accessed, as well as additional management information, such as whether the user wants a calibration for a specified run or whether automatic updating should occur when the run number changes.

This API allows for the possibility that a given calibration object can be initialized from more than one source (each source is called an IO package). It is possible at run time to specify from a configuration file what IO package should be used for each database class. For example, code to initialize an object from Oracle, from mSQL or from an ascii text file could all be supported in parallel. In order to allow this flexibility, filling of calibration objects is done using a low level base access class **IoPkg<calibrationDataObject>**.

1.5.3 Code Generation Tools

The Run 1 CDF support databases contained $\sim$ 200 different types of data object. We expect the number of classes in Run 2 to be comparable. Management of the database schema and ancilliary code is therefore a significant issue. Each database class requires a number of pieces of binding code:

- The definition (data members and access methods) of each class must be defined in a C++ header file.
- For the class data members to be stored in a database, there must be an SQL script that defines a a corresponding set of columns in the relational database.
- The mapping between the class definition and the relational table is done using C++ code. This code must be DBMS specific (speed optimization of the access calls means that Oracle and mSQL calls are not the same).
- Data browsing, histogramming and fitting are done in Root. This requires that the Root data dictionary and ROOTCINT binding code be created.

- Because the number of server connections to Oracle is limited, and because we may wish to cache commonly accessed calibrations in a transient form, we expect to develop (as part of the support database Joint Project) a database object server. Binding code to this server will be needed for each database class. One possible implementation of this server would use Corba to transfer the object to the clients.

The large number of classes and of bindings argues strongly for the development of automatic tools to create all the bindings from a single class description. CDF has developed a prototype automatic code generation tool for use with the calibration database. This prototype creates the C++ header file, the SQL script and the Oracle and mSQL bindings. Corba stubs can also be produced, although we have not yet tested the Corba interface. The use of this tool to generate Root dictionaries is still being investigated.

Automatic code generation requires the following steps: A class is described using a well defined grammer, the class description is parsed to produce an abstract representaion of that class in a convenient internal structure, and a backend uses either this representation directly or an API supplied by the front end parser to generate the required code. Root itself follows such a model with C++ header files used for the class description. The complexity of the C++ language and the difficulty of finding a good C++ parser (see comments on CINT in Section 1.4) have led us to choose a different path. For the prototype, the class definition is done using the Corba Interface Definition Language (IDL). This language is adequate to describe our rather straightforward calibration classes.

Parsing of the IDL class descriptions is done using the Sun CFE parser. This parser is free and reliable, although it has only a rudimentary interface available for use by the backend implementations. Our backend code generator is based on the Fidle package developed by Dave Quarrie during the Babar prototyping phase. This backend is used to emit code tokens that contain the class dependent pieces required for each of the bindings described above. The bindings are then placed in template files using a Perl script.

The automatic code generation tools have been successfully used for several calibration classes. Integration of these tools into SRT has been done in a rather awkward manner. A proper implementation within SRT requires some modifications to that package. These modifications are already on the SRT developers' "to-do" list.

The prototype project has convinced us that automatic code generation is possible and desirable for CDF. We will certainly continue to develop such tools for the database. We are also exploring the possibility of extending this approach to areas other than support databases. In particular, CDF has always considered the possibility of generating Trybos class headers from a definition language (A prototype generator was developed by Rob Kennedy and a summer student in 1997. This generator was driven from a Python/TK GUI.) Because the current Trybos bank structures are so simple, it is likely that IDL would be suffient to describe the current persistent classes. We are, however, currently exploring possible enhancements to our data model (see Section 1.3) and a final requirement on the grammer for class definition must be part of that process.

1.5.4 Database Browsing and Analysis

Once the Run 2 hardware and software configurations are stable, the calculation of calibration constants will be done in the front-end processors. The results of these calculations will be used within the front-end scanners to remove channel-to-channel variations in response. The results will also uploaded and stored in the Oracle database to provide a temporal record. During the commissioning phase, however, the situation is likely to be different. The front-end scanners do not provide a friendly enviroment for algorithm optimization or hardware debugging. We expect that for many components of CDF, debugging and development will be done by uploading intermediate results from the crate and performing fits and analysis from a process running on the online host computer. The interactive fitting, histogramming and analysis features of Root make it an ideal tool for this task.

In the CDF design, data from the front-end is uploaded to AC++ in Trybos format using a special purpose input module. These data are then transfered to Root (using a general purpose utility that maps an STL vector container to a Root TClonesArray) and the Root interpreter is invoked from inside AC++. The user is then free to use all the interactive Root features to study his data. If the user generates calibration constants interactively in Root, these constants will then be written to the Oracle database from AC++ using the standard C++ API.

We also intend to use Root from within AC++ as part of the monitoring software for the calibration database. An AC++ module will be used to query the Oracle database to extract calibration constants for a given set of channels and a given set of runs. Because the database tables contain a mapping of channels to hardware location this query for example can request all channels from a given crate or card. Once the data are extracted from the database, interactive Root will be used for analysis and visualization. For example, the user can histograms of changes in constants between two runs or produce strip charts of the mean calibration for a specified set of channels as a function of time. Prebuilt Cint macros will be used to provide the most commonly needed analysis features.

2 Reconstruction Software

A major integration goal for this Fall was to assemble a first pass production reconstruction executable and exercise that executable on Monte Carlo data. The