Retrievals from DB2 BLOB (Binary Large Objects) Data Warehouse Using SAS®

Tracy L. Lord / Robert C. Pratt
IBM Microelectronics Division
Essex Junction, Vermont

Abstract

This paper describes a method of retrieving data from a DB2 BLOB (Binary Large Object) data warehouse using SAS as the data analysis tool. Our current Data Warehouse architecture involves storing summary data in traditional DB2 relational databases and storing raw chip data in a DB2 BLOB data type. With this BLOB data type many opportunities have opened up for our team to experiment with various methods of retrieving data. This paper details the choices we made and the technical reasons why.

Background

IBM's semiconductor production facility in Essex Junction, Vermont, has a large engineering community and three manufacturing fabricators. With the increase in speed, density, and sophistication of semiconductors the business of manufacturing integrated circuits has become a very competitive business. Improvements in yield, cycle time, and reliability are crucial to maintaining a competitive edge. Integrated data collection, storage, and analysis are the key tools needed to improve one's position in this competitive market world.

A correctly modeled Data Warehouse is an integral part of this analysis solution. Semiconductor data collection happens through many different methods, but the storage must be standardized for a warehouse to function. Data types can be classified into three major types :

1) Raw Chip Data
2) Wafer Summary Data
3) Lot Summary Data

where a wafer is a collection of chips on a flat circular slice of silicon and a lot is a collection of wafers that flow together through the process line.

Wafer and Lot Summary data can easily be stored in traditional DB2 tables because they are orthogonal, but Raw Chip Data has many forms. Determining the best architecture for these databases and the tool to analyze this data was the task at hand.

The Challenge

Our challenge was to build a RISC based Data Warehouse with a friendly user interface for analysis.

The Problems

In semiconductor manufacturing, the demand for test data on parts moving through the production line is insatiable. There seems to be no limit to the number of ways a chip can be tested, and engineers and scientists exploit this test data in as many ways as possible. With the enormous quantities of parts being manufactured, this creates a virtual glut of information. Along with the need for this information is the inevitable requirement for long term storage and fast retrieval. All these elements combine to present a unique challenge to the information systems professionals in a semiconductor manufacturing enterprise.

To add to the challenge, the information being stored is not necessarily well organized, does not follow traditional relational models, and is heterogeneous in type. By this we mean that for a given chip there may be several different types of data represented in the database. The key phrases in this sentence are “different” and “may be”.

Different refers to the type of data stored. A typical test of a memory chip, for example, will have parametric testing to determine such things as poly line widths, the resistance of materials, and power consumption, along with bit mapping of the device to indicate passing and failing cells within the memory array. The parametrics are typically floating point and integer (fixed point) values, while the bit map is an array of bits. Storing these two widely varying types of data would certainly be possible in a traditional relational model by defining FLOAT columns for the parametrics and INTEGER columns to store a representation of a bit map, except for two things.

1) The numbers of columns would exceed the limits of the Relational Database Management System (RDBMS) without ever coming close to the data storage requirements.

2) Not all data may be present all the time. There are many different products in the manufacturing line, each of which is tested a little differently. In a traditional relational model this would lead to much wasted space.

This leads us to the need for a different model to store our data. The model we chose is the extended relational, or object-relational model. Our use of the extended relational model allowed us to create objects to store the data, while using the relational side of the model to maintain the index structure to access the data objects. Unfortunately the problem does note stop there.

To the RDBMS, even an extended relational RDBMS, all BLOBs are created equal and remain a mystery as to their content. The RDBMS also has functions to read/write the BLOB. This presented us with yet another challenge, particularly considering the requirements that subsets of the BLOBs needed to be accessed and returned to the user. To meet these needs we devised a registry structure to define the contents of the BLOBs. We also developed User Defined Functions (UDFs) to access the BLOBs inner parts based on a query into the registry and return subsets of the BLOBs as columns in a view.

One final requirement was the database location. The database did not have to be local to the user accessing it or the applications writing data to it. This network requirement dictated that we choose an RDBMS that would be easily integrated into a networked architecture, and also to be cognizant of the amount of data flowing around the network.

The Architecture

The architecture of the database system consists of four major parts :

1) The local area network
2) The database server
3) The client
4) The application director

Clients consist of RISC and OS/2 workstations running SAS. They are connected to the database and application servers via a Local Area Network (LAN). SQL queries pass from the client through the LAN to the database server / application director machine to the extended relational database. Data retrieved passes back in the opposite direction to the client.

The Network

The network in this case is a Token Ring network running 16Megabit/Sec. The protocol for communications is TCP/IP. Clients and servers communicate via remotely catalogued databases using TCP/IP as provided by the RDBMS remote connectivity support.

The Database Server

The database servers are RS/6000 model R30 multiprocessors running AIX 4.2.1. The RDBMS is DB2/AIX V2.1.2. The database servers authenticate clients locally and support remote cataloging of their databases via TCP/IP from DB2 clients running DB2 Client Application Enabler (CAE).

The Clients

Clients can exist on any platform that support the following :

- O TCP/IP
- O DB2 CAE
- O SAS

In our case, we have clients on both AIX and OS/2. The client machine is connected to the local area network and has TCP/IP running. The client must also have DB2 CAE installed locally. The client catalogs the remote database to make it appear as a local database on the client machine. In this way, the client can access the database with standard Structured Query Language (SQL), unaware that the database is physically on another machine in the network.

The Application Director

Because the databases are distributed among several machines (due to the large data volumes), and to facilitate the registry structure required to access the inner parts of the BLOBs, a machine is set aside specifically to direct the client applications and SQL users to the machine their data is on. This machine also provides the information necessary to extract parts of the BLOBs. We refer to this machine as the application director.

The application director is a RS/6000 running AIX and DB2. The application director could be any platform capable of running DB2 Common Server and TCP/IP.

The Application Director does not contain any BLOB data records and does not engage in the transfer of these large objects, but rather is queried by the clients to determine which database server to talk with to access large objects. The application director contains all the tables necessary to run the network of database servers and direct clients to where their data resides.

The DB2 Design

An extended relational database consists of two parts :

1) The relational data space
2) The object extensions

The traditional relational data space consists of tables, columns, and references between like columns in different tables (referential integrity).

The object extensions consist of object columns attached to normal relational tables in which data that does not fit the relational model are stored. Classical examples of objects are images, audio and video clips, and graphical constructs such as icons. In our environment the objects represent semiconductor test records. These test records are collections of test measurements made against the chip and stored together as a logical unit. At face value, the object as stored in the data tables looks like a simple collection of bytes that appear to have no form. Without some external information to decode the objects, they would be meaningless.

The BLOBs give us value in the database design for several reasons. First, the data provider does not have to insert special descriptive information with the data record which would slow down the process of testing the chips. Along this same vein is the increase in network traffic associated with the naturally larger data records containing descriptive information. At the database end, the objects would be too large to be practical. With data volumes in the hundreds of gigabytes, adding descriptive information into the records would explode the data storage requirements beyond reasonable limits. Objects also allow us to store large numbers of data values. Well beyond

what the relational database system would allow. It also allows us to store varying length records without wasting space since objects can be variable in length and only use storage that is necessary to store the object. Using objects also creates an opportunity to store records of aggregate types where an object is actually comprised of many smaller objects that have different data types. The RDBMS does not know or care about what we store in a binary object, only that it takes up this much space and is stored in this row of the table. This has the additional benefit of reducing the number of rows in the table.

Now that the object has been stored in the database, we need to determine what makes up the object and how to access parts of the object. This is where the registry comes in. The registry is a set of tables that define the type of object (type in this case is defined by the application, not necessarily a DB2 data type) and the contents of the object. Each object is comprised of elements that have a name, type, and length. All of this information is stored in the registry. The type is also stored within the row the object is attached to, so the User Defined Function (UDF) can determine what information in the registry to use to decode the object.

With registry information, a UDF can be written to accept an element name and some criteria to collect a set of BLOB rows from the table. The registry will also be used to drill down into the objects and extract the specific element in the BLOB. The UDF then returns to the SQL user a table view. Other UDFs can be written to extract multiple elements from BLOBs, insert new information into BLOB columns, and update elements in any object. With a UDF and DB2's ability to define abstract data types, the application can build up data structures that are unique to the business - data structures that DB2 could never define natively. An example of which might be an object that contains a map of a memory chip, some speed measurements (integers), and measurements of power usage and heat dissipation (floats).

To complete the package, UDFs must be written to support the objects and the new data types they represent. These UDFs are C programs written, compiled, and stored in the DB2 database. The UDF acts as a function call in an SQL statement that will operate on the rows retrieved by the WHERE clause in the SQL statement. Since the UDF is a C program running on the database server, it can do most anything. In our case the UDF reads a flattened version of the registry (because the only thing a UDF can't do is SQL). With the element name passed, it looks into the objects and extract that element, returning it as a column in the resultant view.

An example of the use of a UDF called GETELEMENT in a query follows :

```
SELECT
LOT,WAFER,CHIP,GETELEMENT(OBJECT1,D_VAL1)
   FROM  DB.TABLE1
     WHERE LOT='123456789' AND
              WAFER='ABCDEF'
```

This query scans the OBJECT1 BLOB in the DB.TABLE1 table and finds the D_VAL1 element in each object, returning it as a column in the table.

Our example returns a view like this:

LOT	WAFER	CHIP	D_VAL1
123456789	ABCDEF	001001	1.234
123456789	ABCDEF	001002	2.32
123456789	ABCDEF	001003	1.234
123456789	ABCDEF	002001	1.325
123456789	ABCDEF	002002	1.234
123456789	ABCDEF	002003	2.343

The Analysis Tools

The analysis tool had to be flexible enough to handle many types of data. Our most predominant data types would be relational DB2 databases and extended relational DB2 databases using BLOBs.

The tool also needed to output many type of reports, graphs, and statistics. With a customer base that extends from manufacturing operations personnel with very little statistics background all the way to chemist, physicist, and engineers, the output options had to be extensive and complete to satisfy everyone.

The analysis tool also had to be non-(i.e. no IBM 390 mainframe) based. With a large push from IS to remove HOST

dependencies, our analysis tool had to be workstation based. Our customer set is 70% OS/2 based and 30% Power PC based. We planned to use a “RISC Farm” of CPU servers to help with the analysis workload of our project. This removed the dependency for everyone to have a powerful WINTEL box on their desk. With a RISC based analysis tool the user could telnet into a RISC box and use its CPU cycles even with a lower powered X486 box. This decision pushed us toward using a RISC based analysis tool.

Once all the requirements came in, the decision was clear to use SAS as our analysis tool. It has flexible input capabilities and extended output functions. We developed a GUI front end using SAS/AF that allowed our users push button access to their data and generic forms of analysis. An added bonus to using SAS is the wealth of tools available for detailed analysis like SAS/INSIGHT, SAS/STATS, and SAS/GRAPH.

Conclusions

Semiconductor manufacturing has become a competitive business where improvements in yield and production can give a company the leading edge necessary for success. A properly designed Data Warehouse and analysis tool can make the difference in this high tech industry. Our requirements led us to a mixture of DB2 database models, both traditional and extended relational. This allowed us to use User Defined Functions for retrieval of the data and SAS as the analysis tool. This solution provided us with a client-server based warehouse / analysis tool thus saving HOST mainframe CPU costs.

Tracy L. Lord
TLORD@US.IBM.COM
c/o IBM Microelectronics
Dept. LCWV/967-1
1000 River Street
Essex Junction, VT. 05452
(802)769-8734

Robert C. Pratt
RPRATT@US.IBM.COM
c/o IBM Microelectronics
Dept. LCWV/967-1
1000 River Street
Essex Junction, VT. 05452
(802)769-9663