Using an SGI CHALLENGE Server to Build a 200 GB Data Warehouse

Bradley R. Eacker
Member of Technical Staff
Silicon Graphics Inc.

Abstract

Large databases are becoming more prevalent as companies collect information about their customers, such as buying habits or marketing responses—so much so that we are seeing requests for databases that exceed hundreds of gigabytes of information. This paper describes an implementation of one of these data sets—sometimes referred to as a data warehouse—covering some of the traps and pitfalls encountered along the way.

Introduction

The objective is to show that the SGI CHALLENGE™ product line can support major database environments that contain more than 200 gigabytes (GB) of information. The 200 GB target came at the request of one of our customers, EDS, whose application contains the history information and buying habits of a large group of customers.

The following table, which is described in greater detail later in this paper, shows the final sizes of the database tables and their indices.

TABLE 1. 200 GB Data Warehouse Facts and Figures

Table Name	Number of Rows	Size (GB)	Index Size(GB)
Address	243,145,504	15.9	4.2
Consumer_hist	821,088,480	34.2	16.6
Customer	273,223,704	22.8	4.7
Household	37,459,307	1.2	0.7
Orders	561,999,025	32.2	13.7
Order_item	1,356,965,750	38.8	27.6
Total sizes		145.1	67.5

1.0 Resource Requirements

At the outset, it was estimated that the 200 GB target would require an SGI CHALLENGE XL system. Previous experience with the TPC A/B at SGI used 32 processors, a disk farm containing 106 drives, and 24 fast and wide SCSI buses.

The TPC A/B used ORACLE™ 7.0.15 as its Relational DataBase Management System (RDBMS), along with some IRIX® database accelerators. EDS wanted to continue using the ORACLE environment, although they requested a migration to 7.1. They felt that, because of the size of this database, the parallel query option in 7.1 would enhance performance.

Based on the knowledge gained from the prior TPC A/B work, and aiming for the target of a 200 GB or larger database, it was determined that 150 2-GB disks would satisfy the storage requirements for this warehouse. This final set of 150 was pared down from a set of 250 originally desired to maintain an online backup copy of the database in the event of a catastrophic failure.

To accommodate EDS's desire to characterize their tools, their database schema was used in the creation of this data warehouse. Their product is a set of direct marketing tools and associated decision support elements. A follow-on to this exercise was to gauge how the EDS tools would scale in a data warehouse environment of this size.

Another aspect of their environment was some data synthesis programs made available for our use. These programs were able to create arbitrarily large sets of rows, which would then be loaded into the tables that defined this particular data warehouse. One problem that arose while this filling process progressed was the original program's inability to fill the address table beyond 20 million rows. This address generation routine was putting only one address per PostalSoft[1] location. There were provisions for more in the sources, but these provisions had not been necessary until the address table exceeded 20 million rows.

1.1 Prototypes and Logistics

In the midst of laying out this large data warehouse, it was determined that another system, containing a smaller (~10 GB), prototype database, would help to get the EDS dbIntellect development progressing. The smaller prototype helped determine how much time would be required for the table filling process on the larger warehouse. This second system, with seven 2-GB drives, also helped to determine the time needed to create the required indices, as well as allowing some experimentation on the warehouse environment using different database setups, such as disk layout, initialization parameters, and temporary tablespace configurations.

Putting together the disk farm described above (150 drives) was quite an effort, and supporting it required help from other areas of the company. Aspects of this effort included items such as the logistics associated with acquiring the necessary drives, making sure that all the drives were connected to the system properly, and setting up the

[1]PostalSoft is a third-party package for validating addresses within the United States.

necessary backup facilities for a data set of this magnitude. To that end, a group of three DLT stackers was used for the backup once the creation process had completed. Each of the stackers was capable of storing approximately 63 GB of data. Filling each stacker's set of seven tapes required about 16 hours of unattended operation. This operation could be done on each of the stackers in parallel. The final chunk of 50 GB required one tape refill cycle.

2.0 Creation of the Data Warehouse

Simple arithmetic says that 200 GB of storage requires 100 2-GB drives. For some conservative leeway on the construction, including the necessary temporary space for building indices, a system with 150 2-GB drives attached to 32 SCSI buses was built. In the final configuration, 24 SCSI buses would probably have been able to handle the necessary group of disk drives, with eight drives attached to each of the 24 SCSI buses.

2.1 Database Creation

Using ORACLE, we found a number of constraints in dealing with a database configuration of this magnitude. The first is a basic limitation: 2 GB is the largest piece of storage (datafile) that the SGI port of ORACLE can handle. This constraint, coupled with the maximum number of datafiles (255) that a particular table space can handle, meant that each table needed to be in a different tablespace. This decision probably helped, because we could then determine the storage usage of each table from the free space available in each tablespace.

Each of the 150 disks was split into a 1.6 GB element and a second piece of 400 MB. The first 1.6 GB of each disk was allotted to ORACLE, to become a separate datafile in the basic 200 GB, and the remaining small pieces were consolidated into logical volumes (LV) to support the smaller prototype. This prototype was used to determine how much time it would take to create the elements of the larger database. The prototype also helped to get an estimate for the amount of storage required for each of the final tables and indices, along with the time needed to complete construction of each element.

2.2 Demo Database Tables

The data warehouse consisted of six primary tables: *address*, *customer*, *household*, *consumer_hist*, *orders*, and *order_item*. (See Appendix A for the complete DDL [Data Definition Language] descriptions for each of these tables.)

Table 1 shows the final sizes of the tables and their indices. Associated with each of these tables is an index used to provide data integrity in the form of constraints on the tables themselves. Data integrity in this context applies to maintaining an environment of unique ID's in each table. The space associated with each index had to be accounted for in the construction of the data warehouse.

In the first table, *address*, each record describes a particular mail location, indexed by a unique *address_id*. Along with this ID, the table held elements describing street information, city, state, zip, and so on. The second table, *customer*, is used to define a particular *person*, with elements such as *first_name*, *last_name*, *address_id*, *household_id*, and so on. To further characterize the database information, the *household* table describes a set of elements covering one or more customers, containing items such as *household_size*, *type_of_residence*, *pool*, and *pets*.

The *consumer_hist* table holds the past purchasing history as records matching customers with products, along with the first and last dates associated with this particular customer purchasing a particular item. Another element of the data warehouse, the *orders* table, contains the information regarding specific orders that have been placed. Items within each *orders* record hold payment method, how the person was contacted, promotional identification, and so on. *Order_item* is a table containing all the separate elements of each order, holding information regarding each particular ordered item with a reference back to a particular order.

2.3 Polk Database Tables

Another aspect of the EDS environment is a database of Polk information that holds approximately 1 GB of household demographic data. This Polk database contains a pair of tables, *household* and *person*, that are filled with real demographic information rather than the constructed information associated with the synthetic Demo tables.

2.4 Parallel Elements for Table Creation

When the initial steps were taken to create the tablespaces for this environment, some elements had to be done serially, while others could be done in parallel. The creation of the different tablespaces was one of the items that could only be done serially. ORACLE seems to have a latch, or *mutual exclusion element*, that it uses when it creates a tablespace.

Adding a data file to a particular tablespace did not seem to suffer from serialization. Since each data file took about 15 minutes to initialize, adding a list of files to each tablespace in parallel saved quite a bit of time over doing each of the tablespaces serially, especially when the 150 different data files were spread over the nine different tablespaces involved in this endeavor.

3.0 Loading Tables

When this project was started, the objective was to generate a database that contained over 200 GB of data. The data generation routines from EDS were capable of generating a large number of records used to populate the previously described tables. These routines generate flat files of information temporarily stored in the UNIX® file system. The first of these programs was the address generator, which not only generated reasonable addresses within a given zip code range, but also had a second part that validated these random addresses. This validation was done with a third-party package that EDS had licensed from PostalSoft. Once a flat file containing a large set of addresses had been created, Oracle SQL*Loader® was used, with the appropriate control file, to load this synthetic address data into the address table. To fill this table to the appropriate size, new sets of addresses from all over the country were created and loaded into the database.

The second generator from EDS created separate flat files for each of the *customer*, *consumer_hist*, *household*, *orders*, and *order_item* tables. These flat files were fed into separate SQL*Loader instances so that each of the tables could be loaded in parallel. Once again, this operation was repeated until the target size for this database was achieved. This goal was less than the 200 GB target, since space was needed for the indices associated with each of the tables.

Once the smaller prototype database was completed, the indices were found to occupy roughly 40% of the space that the data itself occupied. As you can see from Table 1, approximately 145 GB was necessary for the data, so this was the target for the data portion of the warehouse. When this target had been reached, the primary tables, *address* and *customer*, each contained roughly 250 million records. As the filling progressed on these tables, it was noted the run rate for the *customer* table seemed to be about two million records per hour. This figure gave an estimate of approximately 125 hours to do the complete load from scratch, or about five days of fill activity. The procedure for loading was primarily constrained by the amount of filesystem space available for creation of the flat files.

Given more filesystem space, it is very likely that the fill rate could be increased by adding more SQL*Loader instances. Another worthwhile alternative would be to use the parallel load option that ORACLE provides. But the time to load the data was similar to the time to create the flat files, so this may not have saved a substantial amount of time.

4.0 Indexing

One of the effects of using SQL*Loader in direct mode is that the index is not built on the fly. This direct mode load is necessary to optimize the rate at which rows can be loaded into the tables. Once the loading phase had completed, it was necessary to build the indices for each of the tables. The smaller prototype database showed that the *order_item* table and its index were of comparable size. The *order_item* table in the large warehouse filled 38 GB with an index of about 28 GB. In the process of building this index, ORACLE needed an extra 28 GB of disk storage for temporary space and about 45 hours of elapsed run time. Total build time for all the indices was calculated to be just over 100 hours, or about four days.

The estimates for building these indices were derived from previous experiments with the smaller prototype database. Each attempt on the prototype taught some lessons on how to use the parallel option on index creation, along with the best ways to lay out the temporary space on the available disk drives. Parallel index builds seem to proceed along a particular sequence of steps for ORACLE 7. As the first step, ORACLE scans the entire table, handing off different pieces of the table to each of the parallel elements. Meanwhile, the parallel elements are building their portion of the sorted index set in temporary space. This sorting is done in memory unless the data set exceeds the sort area's available memory. Once this occurs, the data is transferred to the SGA (shared memory region) and, subsequently, to the temporary tablespace.

Next there is a consolidation step to build the final index from the parallel sets. The massive set of IOPS (I/Os per second) associated with this amount of data moving through the SGA exposed the DB writer process as a bottleneck. This same behavior had been encountered previously in the TPC A and B efforts. The solution to this bottleneck was to spread the temporary space over as many drives as possible in order to increase the available IOPS count. In fact, when the temporary space was specified, the files were symlinked to logical volumes. These logical volumes were spread over 40 different disk spindles to provide the necessary aggregate IOPS for building the indices in a timely fashion.

5.0 Optimizations

This logical volume configuration was just one of the optimizations that SGI has available for use with the ORACLE database environment. There are other accelerators that have been added to the SGI-dependent portion of the ORACLE installation when the dba.sw.* package for IRIX has also been installed. The first of these is the *postwait* driver. To use this accelerator, it is necessary to add the following 2 lines to the ORACLE configuration file(*init.ora)*:

```
use_post_wait_driver = true
post_wait_device = /dev/postwait
```

ORACLE and SGI use the *postwait* driver to facilitate the suspend/resume primitives ORACLE relies upon. This set of primitives is used to do the process switching necessary for an ORACLE instance for each of the elements of work.

Another accelerator is the *list_io* environment that has been added to the SGI port. Acceleration in this area is accomplished by sending a list of device and block requests into the kernel, rather than having each I/O request presented to IRIX independently. Once inside the kernel, these I/O requests are farmed out to each of the devices in parallel. *List_io* is enabled by the following line in the *init.ora* file:

```
use_async_io = true
```

This amount of optimization is dependent on several other elements. First, it is necessary to ensure that ORACLE is attempting to do a large number of IOPS so that *list_io* can be asked to do multiple operations at the same time. Once *init.ora* was configured for *list_i*o, the following lines were added to get ORACLE to do more I/Os.

```
_db_block_write_batch = 256
db_block_checkpoint_batch = 256
```

Another aspect of this optimization is that *list_i*o is dependent on the number of spindles that will be active when a particular I/O is attempted. The use of logical volumes increases the number of active spindles by spreading the data out over the different disks. But this distribution must be tempered with the number of IOPS that a particular drive can accomplish within a set time. For the 2GB drives used in this experiment, 60 IOPS is about the best that can be done when doing random seeks spanning the entire drive.

6.0 Accessing the Database

Having the database and indices built is only half of the solution. The real benefit of having this massive amount of data available is to try to garner some interesting information from the it. One of the queries that was of interest to EDS was:

```
SELECT
 /*+ FULL(CUSTOMER) ORDERED PARALLEL(CUSTOMER,16)
 INDEX(ADDRESS ADDRESS_PK) USE_NL(ADDRESS)
 INDEX(HOUSEHOLD HOUSEHOLD_PK) USE_NL(HOUSEHOLD) */
 CUSTOMER.CUSTOMER_ID, ADDRESS.ADDRESS_ID,
 HOUSEHOLD.HOUSEHOLD_ID
 FROM CUSTOMER, ADDRESS, HOUSEHOLD
 WHERE (CUSTOMER.HOUSEHOLD_ID=HOUSEHOLD.HOUSEHOLD_ID)
 AND CUSTOMER.HOUSEHOLD_MEMBER_TYPE = 'HEAD'
 AND HOUSEHOLD.TYPE_OF_RESIDENCE = 'M'
 AND HOUSEHOLD.LTH_OF_RESIDENCE > 36
 AND HOUSEHOLD.PETS = 'Y'
 AND (ADDRESS.ADDRESS_ID=CUSTOMER.ADDRESS_ID)
 AND (ADDRESS.STATE = 'MA' OR ADDRESS.STATE = 'CO')
```

Table 2 summarizes the observed run times when this query was submitted to ORACLE against the 200 GB database while varying the number of processes:

TABLE 2. Parallel Query Results

# of CPUs Active	Elapsed Time
1	9:30 hours
4	2:17 hours
8	1:21 hours
12	1:08 hours
16	0:50 hours

To gather these times, each of the values associated with the number of CPUs in Table 2 was substituted for the value of ‘16’ in the line

```
PARALLEL(CUSTOMER,16)
```

shown in the select statement above.

With the increase in the amount of parallelism, the time to complete the operation dropped off rapidly. Table 2 also shows there is a limit to the parallelism available based on the number of available CPUs in the system. In this case, the number of CPUs equated with the number of parallel processes that ORACLE was requested to use. This actually works in practice due because each of these processes was CPU bound.

An attempt to increase the parallelism beyond 16 did not cause a decrease in time, as expected; instead, a bottleneck was observed. This situation is currently under investigation and has been brought to Oracle’s attention.

7.0 Conclusions

This project was put together to show that the CHALLENGE Server is well suited to a massive database environment, also known as a “data warehouse”. The CHALLENGE Server product line is easily capable of handling almost 1 terabyte (TB) of raw disk using current technology drives (2 GB each) and about 3.6 TB when using Raids. Up to this point, an SGI CHALLENGE Server had not been utilized to put together anything larger than the 40 GB database necessary for the TPC A and B environments. As demonstrated in this paper, the SGI CHALLENGE product line is quite capable of creating and using these massive data sets, delivering exceptional performance when dealing with this quantity of information.

This “proof of concept”, building a greater than 200 GB database, confirms that SGI is well positioned to be a major player in the commercial database market with the CHALLENGE Server line, especially when you see SGI using this massive data warehouse capability with Data Mining and Data Visualization (See the Silicon Graphics White Paper, *Data Mining with Silicon Graphics Technology*, by Arun Swami).

Appendix

```
create table demo.address
       (address_id          number(12)  not null,
       constraint address_pk
                 primary key (address_id),
       address_line         varchar2(60),
       carrier_route        char(04),
       city                 varchar2(28),
       county               char(4),
       state                char(2),
       usps_unit            varchar2(9),
       zip                  number(5),
       zip_plus_four        number(4)
       )
       tablespace data1_tbspc
       pctfree 0
       storage ( initial 174M next 174M minextents 6 pctincrease 0
)
       disable primary key;

create table demo.customer
       (customer_id         number(12)  not null,
       constraint customer_pk
                 primary key (customer_id),
       first_name           varchar2(30),
       middle_initial       char(01),
       last_name            varchar2(30),
       title                char(04),
       original_source      char(04),
       acquisition_date     date,
       phone_nbr            number(10),
       address_id           number(12),
       household_id         number(12),
       household_member_type char(4),
       suppr_flag           char(01),
       sex_cd               char(01),
       income_cd            char(01),
       age                  number(03),
       marital_status       char(01),
       ethnicity            char(01),
       education_lvl        char(02),
       mobility_index       char(01),
       nbr_of_dependents    number(02),
       homeowner            char(01)
       )
       tablespace customer_tbspc
       pctfree 0
       storage ( initial 210M next 210M minextents 1 pctincrease 0
)
       disable primary key;
```

```
create table demo.consumer_hist
       (customer_id          number(12)  not null,
       product_id            number(12)  not null,
       constraint consumer_hist_pk
                 primary key (customer_id, product_id),
       first_purchase_date   date,
       last_purchase_date    date,
       total_orders          number(12),
       total_quantity_orderednumber(12),
       total_orders_amount   number(10,2)
       )
       tablespace data2_tbspc
       pctfree 0
       /* pctused 40 */
       storage (
                 initial 281M
                 next 281M
                 minextents 4
                 pctincrease 0 )
       disable primary key;


create table demo.household
       (household_id         number(12)  not null,
       constraint household_pk
                 primary key (household_id),
       household_size        number(02),
       income_cd             char(01),
       type_of_residence     char(01),
       lth_of_residence      number(03),
       nbr_vehicles          number(02),
       nbr_tvs               number(02),
       pool                  char(01),
       pets                  char(01)
       )
       tablespace data3_tbspc
       pctfree 0
       /* pctused 40 */
       storage (
                 initial 16M
                 next 16M
                 minextents 6
                 pctincrease 0 )
       disable primary key;
```

```
create table demo.order_item
       (order_id            number(12)  not null,
       order_item_id        number(12)  not null,
       constraint order_item_pk
                 primary key (order_id, order_item_id),
       product_id           number(12)  not null,
       quantity             number(6),
       status               char(04),
       amount               number(10,2)
       )
       tablespace data5_tbspc
       pctfree 0
       /* pctused 40 */
       storage (
                 initial 337M
                 next 337M
                 minextents 4
                 pctincrease 0)
       disable primary key;


create table demo.orders
       (customer_id         number(12)  not null,
       order_id             number(12)  not null,
       constraint orders_pk
                 primary key (customer_id, order_id),
       order_date           date,
       amount               number(10,2),
       region_id            number(12),
       bill_address_id      number(12),
       ship_address_id      number(12),
       payment_mthd         char(03),
       contact_medium       char(04),
       promo_id             number(12)
       )
       tablespace data4_tbspc
       pctfree 0
       /* pctused 40 */
       storage (
                 initial 270M
                 next 270M
                 minextents 4
                 pctincrease 0
       )
       disable primary key;
```