Case Study

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

Cartesian Products and Joins

A Cartesian join, is a query in which every row of one table is, in turn, matched against every row of another table. This means that, if the first table has 'n' rows, and the second table has 'm' rows, then there will be 'n' full table scans of the 'm' row table, or a total of (m x n) disk accesses.

In terms of computational effort, this type of query is prohibitively expensive, and is only acceptable in circumstances where the tables in question are of a trivial size. However, the situation does arise, where it is impossible to extract the data in any other way, perhaps because the database is a production database and, because of space limitations or other considerations, the creation of smaller temporary tables is not practical.

Whatever the practical applications, this type of query provides an excellent vehicle for the illustration of many aspects of Multi-Dimensional Programming, so we will examine its design in some detail.

Consider the case where we have a table, 'ACCOUNTS', of bank customer account details. The table contains an entry for each account owned by a customer, each account being linked to a customer by the Customer_id.
Such a table construction, in fact, would not exist in reality, as it would represent very bad design practice, but we can pretend that it is a legacy table, and use it to illustrate our point.

Customer_Id	Account_number	Account_type
1234	9876	Investment
1234	8765	Savings
1234	5678	Cheque

Let us assume that we are interested in creating a marketing campaign, whose aim would be, to persuade any customers only having a cheque account, to also open a savings account.
In order to achieve this, we would need a list of those customer_id's, which did not have associated with them. both a Savings account and a Cheque account.
The SQL query could be expressed as follows:

```
SELECT customer_id
FROM  accounts
WHERE account_type = 'Cheque'
AND cutomer_id NOT IN (SELECT customer_id
                       FROM accounts
                       WHERE account_type = 'Savings')
```

It can be seen from the above, that this query will index forward by one row in the main SELECT statement, for each iteration of the sub-query, which performs a full

table scan each time. When the main SELECT has performed a full table scan, the query is complete.

For a trivial table, with a few thousand rows, the run time of this query will probably be quite acceptable but, for a realistic case, where the bank has 10 million customers, some of whom have two or three accounts, the table may well contain 25 million rows. This means that we would be faced with (25,000,000 x 25,000,000) disk accesses.

Fortunately, we can achieve the same result, with only a single table scan, and some trivial comparisons on a simple array of structures.

The first thing we need to do, is to rearrange the above query, so that it reduces, as nearly as possible, to simple SELECT statements, with a minimal predicate.

We can separate the query and sub-query to give:

SELECT customer_id
FROM accounts
WHERE account_type = 'Cheque'

and

SELECT customer_id
FROM accounts
WHERE account_type = 'Savings'

These are the queries which we will implement as cursors and, at this stage, we would try to estimate how much memory we would need, in order to store the returned data. The only two columns of interest, are the customer_id and account_type, which are of type integer and varchar, respectively. However, the account_type is only a two-valued flag, so that it could be represented more economically by a single character, the translation being performed on the fly, as the data is read from the cursor. Thus, if we needed to store the whole table, we would need to need to allocate:

(4+1) = 5 bytes per row, or (5 x 25,000,000) = 125 megabytes of RAM.

It seems, at first, that we should double this figure, and have two arrays of structures, each looking like:

```
struct data {
   int customer_id;
   char flag;
};
```

We could then perform our comparisons on the two arrays, in memory, and store the results, but there is a very good reason for not doing this.

With each iteration of our cursor, we send an SQL request to the database engine. This has to be parsed and executed and, even with query caching, this is an extremely inefficient way of extracting data.
Instead, we will perform an array fetch, and fetch an entire array of records per SQL request. This feature is only available with Oracle, and requires that we extend our data structure, from that described above, to the following format:

struct data accts[FCH];

Where FCH is

#define FCH 50000

earlier in the file.

The 50000 is a purely arbitrary figure, based on our knowledge of the normal traffic to the database engine, and how busy it might be, at the time we want our query executed. The larger the FCH parameter, the more we monopolise the database.

Having decided to perform array fetches, we now have to think ahead to how we will perform the comparisons.
If we merely fill up two arrays, and do the comparisons with two nested loops, like:

```
for(I = 0; I < max1; I++){
        for(j = 0; j < max2; j++){
                if(array[I] == array[j]){
                        (do something);
                }
        }
}
```

this will use the maximum amount of CPU power, since both of the loops will be very tight – in other words, the amount of processing within the inner and outer loops is very small. This, in itself, is not a bad thing.
However, what is bad, is that:

1. Since we have only one process, we will only be running on one CPU
2. We will be sitting around, doing absolutely nothing, while our loops complete.

A better scenario, is as follows:

1. Using array fetches, place the smaller of the two tables into an array of structures. Since we are doing both fetches from the same table, we need to estimate the smaller of the two data sets.
2. Again, using array fetches, fill a buffer array of 50,000 elements from the second cursor.
3. Fork a second process – this will guarantee to get us a separate CPU. Keep forking processes till we have read all the data.
4. Within the new process, perform the comparisons.

   If we did just that, then we would gain very little. Although we have parallelised the comparisons by a factor of 50,000, within each process, we are still sitting around, waiting for a comparison of, effectively, a full table with 50,000 rows.

   To overcome this shortcoming, we employ multi-threading.

5. If the reference array, which contains the contents of the smaller of the two tables,is larger than 50,000, we split it into a reasonable number of segments, like 100, or 500, and pass the starting and stopping array indices to our comparison function. Then, after each cursor fetch, we assign each array segment to a separate thread, for comparison.
   If the reference array is less than 50,000, then we split the incoming 50,000 elements, instead.

Because threads are almost smoke and mirrors, we will not achieve an improvement factor equal to the number of threads we launch. However, a factor of half that should be achievable, so 500 threads will probably give us a speed improvement of 250 times.

6. Communicate our data back to the parent process with a shared memory link.

   We establish the link in the manner examined in detail in the chapter on IPC and Shared Memory.
   - As the parent launches each child, it passes to it the SHMID of a 200 byte token segment.
   - The children each connect to the token
   - Each child estimates how much data it will be returning – in our case, the maximum possible amount is the size of the data structure, multiplied by our array size, FCH, which we have predefined as 50000.
   - Each child allocates shared memory, and starts its thread pool working on the comparisons.
   - Since we want to return all matches to our parent, we will need to store these in a thread-safe way.
     A simple, if not particularly imaginative solution, would be to set the flag in the data structure, so that we can indicate the match, on the fly, and use an output routine to transfer to shared memory, only those elements with the flag set. Since each thread is working on a different part of the array, this method is inherently thread-safe and, since all we do is to set a flag, it is also quick, and economical.
     It is not ideal since, when all the threads have finished their task, we have to make one of two decisions:
     - Perform a single-threaded search through the entire 50,000 element array, while deciding which elements should be written to the shared memory, for return to the parent
     - Copy the entire array to shared memory, and let the parent do the work.

The Code

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <sys/wait.h>
#include <ctype.h>
#include <dirent.h>
#include <sys/shm.h>
#include <sys/mode.h>
#include <ulimit.h>
#include <search.h>
#include <sys/statfs.h>
#include <sys/stat.h>
#include <sys/vnode.h>

#include <sqlda.h>
```

```
#include <sqlca.h>
#include <oraca.h>
#include <pthread.h>

extern char *getenv();
void *getdata();
void *match();

#define FCH 50000                    /* how many we fetch at a time
*/
#define XTH 20                       /* how many threads */

EXEC SQL BEGIN DECLARE SECTION;

int RROWS = 0;       /* rows expected to be returned */
int ROWS = 0;        /* rows expected to be returned */

char user[255];
char pswd[255];
char dbase[255];

/************************* We're going to do
this....*****************

 SELECT count(*)
 FROM accounts_table a
 WHERE a.account_code = 'CHQ'
 AND a.cust_id NOT IN (SELECT b.cust_id
                       FROM accounts_table b
                       WHERE b.account_code = 'SAV')

**********************************************************************
***/

char *q1 = "SELECT cust_id \
            FROM accounts_table \
            WHERE account_code = 'CHQ'";

char *q2 = "SELECT cust_id \
            FROM accounts_table\
            WHERE account_code = 'SAV'";


EXEC SQL DECLARE DB_ACT DATABASE;

struct data {         /* generic structure */
    int flag;
    int cust_id;
};

struct data acctx[FCH]; /* main cursor buffer */

struct data *accta;     /* reference data */
struct data *acctb;     /* q2 main data */
struct data *acctm;     /* matches */

EXEC SQL END DECLARE SECTION;

void cleanup(int);       /* interrupt handler */

pthread_t thr[XTH];        /* our workers */
```

```
int th = 0;              /* no of threads */
int mch[XTH];              /* no of matches */

struct xbounds {         /* array bounds for matching */
    int start;
    int end;
    int what;
};
struct xbounds bounds[XTH];

int matches = 0;
int children = 0;

#define CMAX 200            /* how many children we permit */

int shmid_s;
int token;                  /* size of token memory (~200 bytes) */
unsigned char *chptr[CMAX]; /* where the child writes its identity
etc */
pid_t pids[CMAX];           /* where we keep PIDS for children we
wait for */


main(argc, argv)                /* main */

int argc;
char **argv;

{

char *p1;
int lim;           /* for checking memory & stack limits */

    if(argc < 2){
        strcpy(user, "staging");
        strcpy(pswd, "ithotn1mst");
        strcpy(dbase, "stagep");
    } else {
        strcpy(user, argv[1]);
        strcpy(pswd, argv[2]);
        strcpy(dbase, argv[3]);
    }

    if(pthread_setconcurrency(20) != 0){
        printf("Warning: Can't set thread concurrency\n");
    }

    EXEC SQL CONNECT :user IDENTIFIED BY :pswd at DB_ACT using
:dbase;

    if(sqlca.sqlcode != 0){
        printf("Connection refused:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }
    printf("Connected to %s\n", dbase);

    getdata(q1, q2);
```

```
}                          /* main */

/*****************************************************************
*******
 * Performs split-join processing of the query
 * acctb is a dynamically allocated array of structures of type
'data',
 * which holds the reference 'CHQ' data.
 * acctx is an array of data structures of type 'data', FCH elements
in
 * length. It gets loaded and reloaded with each cursor iteration.
 * The 'match()' function compares each element of acctb, with each
element
 * of acctx, divided into XTH segments, each of which is serviced by
a
 * thread.
 * acctm is a dynamically allocated array of structures of type
'data',
 * which holds the matched cust_id's from acctb, as they get matched.
 * mch is an array of XTH int's, one for each thread. As the thread
finds
 * each match, it increments its particular location so that, when
all
 * XTH threads have finished, the sum of the array elements is the
total
 * no of matches found in the 50k acctx array.

*****************************************************************
******/

void *
getdata(pstring, qstring)                /* getdata */

char *pstring;          /* 'CHQ' */
char *qstring;          /* 'SAV' */

{

int i, j, k;
int status = 0;
int flag = 0;
int total = 0;       /* rows returned this pass */
int fetch = 0;       /* cumulative rows returned previous passes */
int ret = 0;         /* number of matches returned by child proc */
int SQLC = 0;
time_t t1, t2;
int mchs = 0;        /* matches on current fetch */

    printf("Preparing cursor 1...\n");
    EXEC SQL AT DB_ACT PREPARE pcur FROM :pstring;
    if(sqlca.sqlcode != 0){
        printf("Can't prepare cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }

    printf("Preparing cursor 2...\n");
    EXEC SQL AT DB_ACT PREPARE qcur FROM :qstring;
    if(sqlca.sqlcode != 0){
        printf("Can't prepare cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }
```

```
    printf("Declaring cursor 1...\n");
    EXEC SQL AT DB_ACT DECLARE xcur CURSOR FOR pcur;
    if(sqlca.sqlcode != 0){
        printf("Can't declare cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }

    printf("Declaring cursor 2...\n");
    EXEC SQL AT DB_ACT DECLARE ycur CURSOR FOR qcur;
    if(sqlca.sqlcode != 0){
        printf("Can't declare cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }

    printf("Setting read-only...\n");
    EXEC SQL SET TRANSSAVION READ ONLY;

    printf("Opening cursor...\n");
    EXEC SQL AT DB_ACT OPEN xcur;
    if(sqlca.sqlcode != 0){
        printf("Can't open cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }

    printf("Opening cursor...\n");
    EXEC SQL AT DB_ACT OPEN ycur;
    if(sqlca.sqlcode != 0){
        printf("Can't open cursor:%s\n", sqlca.sqlerrm.sqlerrmc);
        exit(-1);
    }

    /* Find out how big our reference table is */

    printf("Estimating 'CHQ' rows to be fetched from
person_dmkr...\n");
    EXEC SQL AT DB_ACT
        SELECT COUNT(*) INTO :RROWS
        FROM person_dmkr
        WHERE inference_code = 'CHQ';

    if((acctb = (struct data *)malloc(RROWS * sizeof(struct data)))
== NULL){
        printf("Memory allocation error\n");
        exit(-1);
    }

    /* load table into memory */

    printf("Fetching %d rows from person_dmkr...\n", RROWS);

    i = 0;
    memset(acctx, '\0', FCH * sizeof(struct data));

    mch[0] = mch[1] = mch[2] = mch[3] = total = fetch = sqlca.sqlcode
= 0;
    while(!(sqlca.sqlcode == 1403 || sqlca.sqlcode == 100)){
        EXEC SQL AT DB_ACT FETCH xcur INTO :acctx;
        SQLC = sqlca.sqlcode;

        if(sqlca.sqlcode != 0){
            if(SQLC == 1403 || SQLC == 100){
```

```
                EXEC SQL AT DB_ACT CLOSE xcur;
                if(sqlca.sqlcode != 0){
                    printf("Can't close cursor:%s\n",
sqlca.sqlerrm.sqlerrmc);
                    exit(-1);
                }
                if(sqlca.sqlerrd[2] == RROWS){
                    printf("Final block fetched\n");
                }
                fetch = sqlca.sqlerrd[2] - total;
                total = sqlca.sqlerrd[2];
                printf("Loaded %d rows...\n", total);
                /* add the remainder of the rows */
                memcpy((char *)&acctb[i], acctx, fetch *
sizeof(struct data));
                break;                  /* no more */
            } else {
                printf("SQLcode: %d:%s\n", SQLC,
sqlca.sqlerrm.sqlerrmc);
            }
        }
        /*
         * sqlca.sqlerrd[2] is a running total, so we must
         * subtract the previous total, to get the current
         * number fetched
         */
        fetch = sqlca.sqlerrd[2] - total;
        total = sqlca.sqlerrd[2];
        if(fetch == 0) break;

        memcpy((char *)&acctb[i], acctx, fetch * sizeof(struct
data));
        i = total;
        printf("Loaded %d rows...\n", total);
        memset(acctx, '\0', FCH * sizeof(struct data));
    }
    EXEC SQL AT DB_ACT CLOSE xcur;
    /*
     * We fetch 50000 records at a time into acctx
     * Then, we cycle through acctb checking for a matching
     * code for each entry in acctb. Any that match are ignored,
     * the others are counted
     */

    printf("Estimating 'SAV' rows to be fetched from
person_dmkr...\n");
    EXEC SQL AT DB_ACT
        SELECT count(*) INTO :ROWS
        FROM person_dmkr WHERE inference_code = 'SAV';

    /* ROWS = 21705694 */
    printf("Fetching %d rows from person_dmkr...\n", ROWS);

    /* Space to log matches */
    if((acctm = (struct data *)malloc(RROWS * sizeof(struct data)))
== NULL){
        printf("Memory allocation error\n");
        exit(-1);
    }
    matches = 0;
    children = total = fetch = sqlca.sqlcode = 0;
```

```
    memset((char *)&mch[0], '\0', sizeof(mch));
    memset((char *)acctm, '\0', RROWS * sizeof(struct data));
    memset((char *)&pids[0], '\0', sizeof(pids));

    printf("Fetching main cursor...\n");

    while(!(sqlca.sqlcode == 1403 || sqlca.sqlcode == 100)){
        EXEC SQL AT DB_ACT FETCH ycur INTO :acctx;

        SQLC = sqlca.sqlcode;

        if(SQLC != 0){
            if(SQLC == 1403 || SQLC == 100){
                EXEC SQL AT DB_ACT CLOSE ycur;
                if(sqlca.sqlcode != 0){
                    printf("Can't close cursor:%s\n",
sqlca.sqlerrm.sqlerrmc);
                    exit(-1);
                }
                if(sqlca.sqlerrd[2] == ROWS){
                    printf("Final block fetched\n");
                }
                fetch = sqlca.sqlerrd[2] - total;
                total = sqlca.sqlerrd[2];
                printf("Fetched %d rows. Doing comparisons...\n",
total);

                /* Last pass. We probably have < FCH records */
                compare(fetch);
                if(children > 1){
                    await();
                }

                break;
            } else {
                printf("SQLcode: %d:%s\n", SQLC,
sqlca.sqlerrm.sqlerrmc);
            }
        }

        /*
         * sqlca.sqlerrd[2] is a running total, so we must
         * subtract the previous total, to get the current
         * number fetched
         */
        fetch = sqlca.sqlerrd[2] - total;
        total = sqlca.sqlerrd[2];
        if(fetch == 0) break;

        printf("Fetched %d rows. Doing comparisons...\n", total);

        /* Fork child process with XTH threads */

        compare(fetch);
        /* only wait if we have 2 to XTH children */
        if(children > 1 && children % XTH == 0){
            await();
        }

        memset((char *)acctx, '\0', sizeof(acctx));
        memset((char *)&mch[0], '\0', sizeof(mch));
```

```
    }
    EXEC SQL AT DB_ACT CLOSE ycur;

    EXEC SQL AT DB_ACT COMMIT WORK RELEASE;


}                       /* getdata */

/*****************************************************************
********
 * We split the current number fetched into XTH segments, and send
out XTH
 * threads, to process them in parallel. Each thread is waited for,
and
 * the function returns when all XTH have done.
 * If the number fetched doesn't divide by XTH, we reduce it by the
remainder,
 * which gives us a possible error of +/- 9 in 21,000,000
 * We have our reference array, and the latest 50k buffer from the
last
 * FETCH, set globally.
 * When the process forks, it takes with it these current copies, and
 * sets the 'bounds' criteria, to split the 50k into XTH x 5k
segments.
 * A separate thread is launched to service each segment, and place
the
 * matches into the appropriate acctm array.
 * when all threads are done, we sum the total matches from each
thread,
 * now in the mch[] array, and use the total for our exit return
value.
 * Meanwhile, the parent process, which has been waiting for all
children,
 * decodes the exit status, returned by waitpid(), and sums each
value in
 * the global 'matches', which stores the total matches.

*****************************************************************
*******/

compare(fetch)                      /* compare */

int fetch;    /* number fetched by cursor this time */

{

int j, k;

pid_t pid;
int status = 0;
time_t t1, t2;

int shmid;          /* shm ID used by child */

    /*
     * token memory, for child to write its ID and return value
     */
    token = sizeof(unsigned char) * 200;

    /*
     * shmid_s is global, so it can be viewed by the child process,
```

```
     * and attached.
     */
    if((shmid_s = shmget((key_t)IPC_PRIVATE, token, IPC_CREAT |
0666)) <= 0){
        perror("Server: Error obtaining shared memory");
        return(-1);
    }
    xids[children] = shmid_s;

    /*
     * shmat returns a pointer to the segment defined by shmid
     */
    if((chptr[children] = (unsigned char *)shmat(shmid_s, 0,
SHM_RND)) == (unsigned char *)-1){
        perror("Server: Error attaching to shared memory");
        return(-1);
    }
    memset((char *)chptr[children], '\0', token);

        switch((pid = fork())){
    case -1:
        perror("Fork failed");
    break;
    case 0:

        time(&t1);

        if((chptr[children] = (unsigned char *)shmat(shmid_s, 0,
SHM_PAGEABLE)) == (unsigned char *)-1){
            perror("Client: Error attaching to incoming shared
memory");
            quit(-1);
        }

        /*
         * make sure the total is divisible by XTH
         */
        if((k = (fetch % XTH)) != 0){
            fetch -= k;
        }

        /* initialise our thread pool */
        th = 0;
        memset((char *)&thr, '\0', sizeof(thr));

        for(k = 0; k < XTH; k++){
            bounds[k].start = (k * fetch) / XTH;
            bounds[k].end =  ((k + 1) * (fetch / XTH)) - 1;
            bounds[k].what = k;
            if((pthread_create(&thr[th], NULL, match, (void
*)&bounds[k])) != 0){
                printf("Failed to create thr[%d]\n",th);
            }
            th++;
        }

        /* we can't return until all threads are done */
        for(k = 0; k < th; k++){
            if(pthread_join(thr[k], NULL) != 0){
               printf("Failed to join thr[%d]\n",k);
            }
```

```
        }

        time(&t2);

        /* return the sum of matches for all threads */
        k = 0;
        for(j = 0; j < XTH; j++){
            k += mch[j];
        }
        printf("PID %d finished. Found %d matches in %d sex\n",
getpid(), k, (t2 - t1));
        sprintf((char *)chptr[children], "%d %d %d", children,
getpid(), k);

        exit(k);
    break;
    default:
        pids[children] = pid;
        children++;
        printf("PID %d launched\n", pid);
    break;
    }


}                           /* compare */

/*****************************************************************
*******
 * This routine only runs within one thread of the child process, and
 * writes its result to two global arrays, acctm, which has the
cust_id
 * of the match, and mch[n]. which stores the total number of matches
for
 * the thread, 'n'.
 * The outer loop goes through the reference data (smaller table),
checking
 * each of the rows against the fraction of the current cursor fetch
 * allocated to this thread. As soon as we get a match, we count it
and go
 * on to the next one, since we're only interested in knowing whether
each
 * CHQ has an SAV, or not.
*****************************************************************
*******/

void *
match(where)                /* match */

struct xbounds *where;


{

int start;
int end;
int what;
int i, j;
int vflag;

    start = where->start;
    end = where->end;
```

```
    what = where->what;

    /* clear the total for this thread */
    mch[what] = 0;

    for(j = 0; j < RROWS; j++){  /* check this CHQ against all 50k
SAV's */
        vflag = 0;
        for(i = start; i < end; i++){  /* go through our fraction of
50k rows */
            if(acctx[i].cust_id == acctb[j].cust_id){
                acctm[j].cust_id = acctb[j].cust_id;  /* save the
evidence */
                mch[what]++;
                break;                          /* match? OK, do the
next one */
            }
        }
    }
    pthread_exit(NULL);

}                       /* match */

/*****************************************************************
*******
 * Waits for something useful to appear on the tokens, handed out to
the
 * children. As soon as a child puts its PID, number and return value
on
 * the token, we can update the match total.

*****************************************************************
******/

await()                 /* await */

{

int i;
int one, two, three;
int flag = 0;
int done[CMAX];
int CCH;

    CCH = children;

    printf("Parent waiting for %d children to write to shm
segments...\n", CCH);
    memset((char *)done, '\0', sizeof(done));

    while(1){
        for(i = 0; i < CCH; i++){
            if(sscanf((char *)chptr[i], "%d %d %d", &one, &two,
&three) == 3){
                if(one == 0 || two == 0) continue;
                if(done[i] == 99) continue;
                printf("Child %d returned\n", i);
                children--;
                done[i] = 99;
                matches += three;
```

```
                printf("Child %d returned %d matches. New total:%d
Children: %d\n", i, three, matches, children);
                if(children == 1){
                    flag = 1;
                    break;
                }
            }
        }
        if(flag == 1) break;
    }

}                       /* await */
```