TableLoad.cpp

```
// TableLoad.cpp - GENERATE THE TABLE LOAD PROFILE
//
// MODULE INDEX
// NAME                    CONTENTS
// DayOfWeek               Figure out the day of the week of a particular date
//                         (monday = 0)
// GetTimeSeg              Get the time segment of a timestamp
// GetSizeInd              Get group size index
// LoadIsZero              Test if the load is zero for all group size segments
// EmitTitle               Emit a page title
// main                    Main line
//
// MAINTENANCE HISTORY
// DATE                    PROGRAMMER AND DETAILS
// 01-09-08         JS     Original
//
//-----------------------------------------------------------------------------

#include <cstdlib>         // C-style standard library
#include <cstring>         // C-style string manipulation functions
#include <cctype>          // C-style character typing functions
#include <iostream>        // C++ input/output streams
#include <iomanip>         // C++ input/output manipulators
#include <sstream>         // C++ string stream declarations
#include <set>             // C++ set declarations
#include <map>             // C++ map declarations
using namespace std;       // Expand the standard namespace
#include <unistd.h>        // Unix standard functions
#include "dtime.h"         // Double time declarations
exec sql include sqlca;    // Include SQL communications area

//-----------------------------------------------------------------------------

// DEFINITIONS

static const long   TIME_SEG_CNT = 36;
                           // Number of time segments
static const size_t GROUP_SIZE_CNT = 4;
                           // Number of group size segments
static const int    LINES_PER_PAGE = 55;
                           // Lines per page

//-----------------------------------------------------------------------------

// CODES FOR DAYS OF THE WEEK

static const char *const DAY_CODE_ARR[] = {"MON", "TUE", "WED", "THU", "FRI",
       "SAT", "SUN"};

//-----------------------------------------------------------------------------

// TABLE MAP STRUCTURE

typedef map<string,int> TableMap_t;
                                   // Table map of table number and
                                   // current number of covers
typedef TableMap_t::iterator TableIter_t;
                                   // Table iterator

//-----------------------------------------------------------------------------

// EVENT TYPE CODES

static const short ORDER = 0;
                                   // An order event
static const short SETTLEMENT = 1;
                                   // A settlement event
```

```
//---------------------------------------------------------------------------

// GLOBAL DATA

set<int>     daySet;            // Set of selected days of week
Dtime_t      fromDate;          // From date
Dtime_t      toDate;            // To date
int          pageNo;            // Page number
int          linesRem;          // Lines remaining on the page

//---------------------------------------------------------------------------

// FIGURE OUT THE DAY OF THE WEEK OF A PARTICULAR DATE (MONDAY = 0)

int
DayOfWeek (
      Dtime_t            dtime) // Date stored in double time format
{
      long         dayNo;       // Day number
      static const long REF_DAY_NO = static_cast <long> (
            DtimeFromUserDate("01-09-08") / (24.0*60.0*60.0)) % 7;
                                 // Reference day number

      dayNo = static_cast<long>(dtime / (24.0*60.0*60.0)) % 7;
      return static_cast<int>((dayNo - REF_DAY_NO + 7) % 7);
}

//---------------------------------------------------------------------------

// GET THE TIME SEGMENT OF A TIMESTAMP

long
GetTimeSeg (
      Dtime_t            date,        // Trading day date
      Dtime_t            dtime)       // Double time value
{
      return static_cast<long>((dtime - date) / 3600.0);
}

//---------------------------------------------------------------------------

// GET GROUP SIZE INDEX

size_t
GetSizeInd (
      long         covers)            // Number of guests
{
      size_t       sizeInd;           // Group size index

      if (covers <= 2)
            sizeInd = 0;
      else if (covers <= 4)
            sizeInd = 1;
      else if (covers <= 6)
            sizeInd = 2;
      else
            sizeInd = 3;
      return sizeInd;
}

//---------------------------------------------------------------------------

// TEST IF THE LOAD IS ZERO FOR ALL GROUP SIZE SEGMENTS

bool
LoadIsZero (
      const long   *loadArr)    // Load for each size segment
{
      size_t       i;           // General purpose index
```

```
	i = 0;
	while (i < GROUP_SIZE_CNT && loadArr[i] == 0) i ++ ;
	return i >= GROUP_SIZE_CNT;
}

//------------------------------------------------------------------------------

// EMIT A PAGE TITLE

void
EmitTitle ()
{
	time_t              sysTime;           // Current system time
	struct tm           localTm;           // Local time
	int                 day;               // Day number
	set<int>::iterator  dayIter;           // Day-of-week iterator
	string              dayList;           // Day-of-week list
	char                fromUserDate[8+1]; // From user date string
	char                toUserDate[8+1];   // To user date string

	pageNo ++ ;

	// If not the first page, separate the pages with a form-feed

	if (pageNo != 1) cout << '\f';

	// Display the title

	sysTime = time(0);
	localTm = *localtime(&sysTime);
	cout << right << setfill('0');
	cout << setw(2) << localTm.tm_mday;
	cout << '-';
	cout << setw(2) << localTm.tm_mon + 1;
	cout << '-';
	cout << setw(2) << localTm.tm_year % 100;
	cout << ' ';
	cout << setw(2) << localTm.tm_hour;
	cout << ':';
	cout << setw(2) << localTm.tm_min;
	cout << "      ";
	cout << "TABLE LOAD PROFILE";
	cout << setfill(' ') << setw(13) << ' ';
	cout << "PAGE " << setw(3) << pageNo << '\n';

	// Generate the list of selected days of the week

	day = 0;
	while (day < 7 && daySet.find(day) != daySet.end()) day ++ ;
	if (day >= 7) {
		dayList = "for all days";
	} else {
		dayList = "for ";
		for (
			dayIter = daySet.begin();
			dayIter != daySet.end();
			dayIter ++
		) {
			if (dayList.length() > 4)
				dayList += ", ";
			dayList += DAY_CODE_ARR[*dayIter];
		}
	}
	cout << setw((58-dayList.length())/2) << " " << dayList << '\n';
```

```
	// Display the reporting period

	cout << setw(14) << " "
	     << "between " << UserDateFromDtime (fromUserDate, fromDate)
	     << " and " << UserDateFromDtime (toUserDate, toDate)
	     << '\n' << '\n';

	// Display the column titles

	cout << "Hourly" << setw(15) << ' ';
	cout << "Peak Concurrently Open Tables\n";
	cout << "Segment" << setw(12) << ' ';
	cout << "1-2       3-4       5-6    7-or-more\n";
	cout << setw(17) << ' ' << "Avg Max   Avg Max   Avg Max   Avg Max\n";

	// Reset the number of data lines remaining on the page

	linesRem = LINES_PER_PAGE;
}

//--------------------------------------------------------------------------

// MAIN LINE

int
main (
	int		argc,		// Argument count
	char		*argv[])	// Argument value pointers
{
	size_t		i, j;		// General purpose indices
	int		c;		// Argument character
	int		day;		// Day-of-week
	bool		daysSelected;	// Days selected flag
	bool		periodDefined;	// Reporting period defined flag
	bool		endOfEvents;	// End-of-events flag
	Dtime_t		eventDate;	// Event date
	Dtime_t		eventTime;	// Event time
	Dtime_t		curDate;	// Current trading day date
	TableMap_t	tableMap;	// Table map for the trading day
	TableIter_t	tableIter;	// Table iterator
	long		curTimeSeg;	// Current time segment
	long		timeSeg;	// This event's time segment
	size_t		sizeInd;	// Group size index
	long		dateCnt;	// Number of trading day dates
	long		peakLoad[TIME_SEG_CNT][GROUP_SIZE_CNT];
					// Peek loads for each time segment
					// and group size
	long		curLoad[GROUP_SIZE_CNT];
					// Current load for each group size
	long		totalPeakLoad[TIME_SEG_CNT][GROUP_SIZE_CNT];
					// Total peak load for calculating
					// averages
	long		maxPeakLoad[TIME_SEG_CNT][GROUP_SIZE_CNT];
					// Maximum peak loads
	long		beginTimeSeg;	// Begin listing time segment
	long		endTimeSeg;	// End listing time segment
	double		avgPeakLoad;	// Average peak load

	exec sql begin declare section;
		char		fromSqlDate[10+1];	// From SQL date
		char		toSqlDate[10+1];	// To SQL date
		char		tableNo[4+1];		// Table number
		char		eventSqlDate[10+1];	// Event date
		char		eventSqlTime[26+1];	// Event time
		long		covers;			// Number of guests
		short		eventType;		// Event type code
	exec sql end declare section;

	// Connect to the database

	exec sql connect to tabledb;
```

```
// Jump to DbError whenever an SQL error occurs

exec sql whenever sqlerror goto DbError;

// Decode arguments

periodDefined = 0;
daysSelected = 0;
while ((c = getopt (argc, argv, "D:W:")) != -1) {
	switch (c) {
	case 'D':
		fromDate = DtimeFromUserDate (optarg);
		if (fromDate == NULL_DTIME) {
			cerr << "Error: bad from-date\n";
			exit (1);
		}
		SqlDateFromDtime (fromSqlDate, fromDate);
		if (optind >= argc) {
			cerr << "Error: missing to-date\n";
			exit (1);
		}
		toDate = DtimeFromUserDate (argv[optind]);
		if (toDate == NULL_DTIME) {
			cerr << "Error: bad to-date\n";
			exit (1);
		}
		SqlDateFromDtime (toSqlDate, toDate);
		optind ++ ;
		periodDefined = 1;
		break;
	case 'W':
		for (;;) {
			i = 0;
			while (
				i < 7 &&
				strcmp (DAY_CODE_ARR[i], optarg) != 0
			) i ++ ;
			if (i >= 7) {
				cerr << "Error: bad day-of-week code\n";
				exit (1);
			}
			daySet.insert (i);
			if (optind >= argc || !isalpha(argv[optind][0]))
				break;
			optarg = argv[optind++];
		}
		daysSelected = 1;
		break;
	default:
		cerr << "Error: invalid option\n";
		exit (1);
		// NOTREACHED
	}
}
if (optind < argc) {
	cerr << "Error: superfluous arguments\n";
	exit (1);
}

// If no reporting period was defined, it's a problem

if ( ! periodDefined) {
	cerr << "Error: no reporting period\n";
	exit (1);
}
```

```
// If no specific days were selected, select all days

if ( ! daysSelected) {
	for (day = 0; day < 7; day++)
		daySet.insert (day);
}

// Reset the totals and maxima

for (timeSeg = 0; timeSeg < TIME_SEG_CNT; timeSeg++) {
	for (sizeInd = 0; sizeInd < GROUP_SIZE_CNT; sizeInd++) {
		totalPeakLoad[timeSeg][sizeInd] = 0;
		maxPeakLoad[timeSeg][sizeInd] = 0;
	}
}

// Order the data in the orders and settlement table by
// trading date and event time and select the event types

exec sql declare eventCur cursor for
	select orderDate, orderTime, tableNo, 0, covers
	from   orders
	where  orderDate >= :fromSqlDate and
		orderDate <= :toSqlDate and
		covers <> 0
	union
	select settleDate, settleTime, tableNo, 1, 0
	from   settlements
	where  settleDate >= :fromSqlDate and
		settleDate <= :toSqlDate
	order  by 1, 2;

// Read a look-ahead row, skipping events for
// days of the week that are not selected

exec sql open eventCur;
do {
	exec sql fetch      eventCur
		into   :eventSqlDate, :eventSqlTime,
			:tableNo, :eventType, :covers;
	endOfEvents = SQLCODE != 0;
	if ( ! endOfEvents) {
		eventDate = DtimeFromSql (eventSqlDate);
		eventTime = DtimeFromSql (eventSqlTime);
	}
} while (
	! endOfEvents &&
	daySet.find(DayOfWeek(eventDate)) == daySet.end()
);

// Process dates until the end of the events is found

dateCnt = 0;
while ( ! endOfEvents) {

	// Reset the data structures for the new trading day

	curDate = eventDate;
	tableMap.clear ();
	for (timeSeg = 0; timeSeg < TIME_SEG_CNT; timeSeg++)
		for (sizeInd = 0; sizeInd < GROUP_SIZE_CNT; sizeInd++)
			peakLoad[timeSeg][sizeInd] = 0;
	for (sizeInd = 0; sizeInd < GROUP_SIZE_CNT; sizeInd++)
		curLoad[sizeInd] = 0;
	curTimeSeg = 0;
```

```
// Process the events for the current trading day

while ( ! endOfEvents && eventDate == curDate) {

	// Roll time forward to the new time segment

	timeSeg = GetTimeSeg (curDate, eventTime);
	while (
		curTimeSeg < timeSeg &&
		curTimeSeg < TIME_SEG_CNT
	) {
		curTimeSeg ++ ;
		for (j = 0; j < GROUP_SIZE_CNT; j++)
			peakLoad[curTimeSeg][j] = curLoad[j];
	}

	// Process the event

	if (eventType == ORDER) {
		tableMap[tableNo] = covers;
		sizeInd = GetSizeInd (covers);
		curLoad[sizeInd] ++ ;
	} else {
		tableIter = tableMap.find (tableNo);
		if (tableIter != tableMap.end()) {
			sizeInd = GetSizeInd(tableIter->second);
			curLoad[sizeInd] -- ;
			tableMap.erase(tableIter);
		} else {
			sizeInd = 0;
		}
	}

	// If the time segment is within range, update
	// the peak loads

	if (timeSeg >= 0 && timeSeg < TIME_SEG_CNT) {
		if (curLoad[sizeInd] >
		    peakLoad[timeSeg][sizeInd])
			peakLoad[timeSeg][sizeInd] =
			    curLoad[sizeInd];
	}

	// Read the next event

	do {
		exec sql fetch      eventCur
			into   :eventSqlDate, :eventSqlTime,
			       :tableNo, :eventType, :covers;
		endOfEvents = SQLCODE != 0;
		if ( ! endOfEvents) {
			eventDate = DtimeFromSql (eventSqlDate);
			eventTime = DtimeFromSql (eventSqlTime);
		}
	} while (
		! endOfEvents &&
		daySet.find(DayOfWeek(eventDate))==daySet.end()
	);
}

// Update the totals and maxima

for (timeSeg = 0; timeSeg < TIME_SEG_CNT; timeSeg++) {
	for (sizeInd = 0; sizeInd < GROUP_SIZE_CNT; sizeInd++) {
		totalPeakLoad[timeSeg][sizeInd] +=
			peakLoad[timeSeg][sizeInd];
		if (peakLoad[timeSeg][sizeInd]
		    > maxPeakLoad[timeSeg][sizeInd])
			maxPeakLoad[timeSeg][sizeInd] =
			    peakLoad[timeSeg][sizeInd];
```

```
                }
            }
            dateCnt ++ ;
        }
        exec sql close eventCur;

        // Emit the results

        pageNo = 0;
        EmitTitle ();

        // Skip leading time segments that are all zero

        beginTimeSeg = 0;
        while (
            beginTimeSeg < TIME_SEG_CNT &&
            LoadIsZero (totalPeakLoad[beginTimeSeg]) &&
            LoadIsZero (maxPeakLoad[beginTimeSeg])
        ) beginTimeSeg ++ ;

        // Skip trailing time segments that are all zero

        endTimeSeg = TIME_SEG_CNT;
        while (
            endTimeSeg > 0 &&
            LoadIsZero (totalPeakLoad[endTimeSeg-1]) &&
            LoadIsZero (maxPeakLoad[endTimeSeg-1])
        ) endTimeSeg -- ;

        // List the time segments
        // A non-zero time segments implies that dateCnt is not zero,
        // so there is no need to check for divide by zero.

        for (timeSeg = beginTimeSeg; timeSeg < endTimeSeg; timeSeg ++ ) {

            if (linesRem < 1) EmitTitle ();

            cout << setfill('0');
            cout << setw(2) << (timeSeg % 24) << ":00 to ";
            cout << setw(2) << (timeSeg % 24) << ":59";
            cout << setfill(' ');
            for (j = 0; j < GROUP_SIZE_CNT; j++) {
                avgPeakLoad = totalPeakLoad[timeSeg][j] / dateCnt;
                cout << "   " << setw(3) << fixed << setprecision(0)
                    << avgPeakLoad;
                cout << ' ' << setw(3) << maxPeakLoad[timeSeg][j];
            }
            cout << '\n';
            linesRem -- ;
        }

        return 0;

        // Process database errors

DbError:
        cerr << "Error: SQLCODE=" << SQLCODE << '\n';
        return 1;
}
```