TIBCO® Patterns

Programmer's Guide

Version 6.1.2 | June 2024

Contents

Getting Started

This section provides details about the TIBCO® Patterns development kit.

What is TIBCO Patterns Developer’s ToolKit (DevKit)?

TIBCO Patterns DevKit is a high level "C" programming interface to the TIBCO Patterns server.

The heart of the TIBCO Patterns DevKit is a set of about a dozen high-level command functions that load records into memory, update them in real time, and perform queries on them.

The TIBCO Patterns DevKit functions are kept simple. This applies not only to basic matching but also to advanced operations like field selection, document search, and match visualization.

All functions are optimized for efficiency. You can use the TIBCO Patterns DevKit to update records in real time, even in multi-threaded applications. This makes it easy to integrate matching and machine learning technology with your existing infrastructure.

For efficiency, load into the TIBCO Patterns servers only records and fields that you search. It is often useful to include identifying keys. For example, if a record is a paragraph of text on a web page, you can use the page's URL as a key.

A Simple Example

Here is a listing of source file `lgrep.c`, a simple, fully functional program that loads a text file of one-field records into the DevKit, then looks up records for a query. If you are familiar with Unix systems, consider it as an TIBCO Patterns analogue of the grep utility. While simple, this program demonstrates most of the DevKit's essential functions.

DevKit type names and function names appear in boldface. DevKit functions fall into three categories:

- Command functions perform administrative tasks, as well as matching and machine learning scoring operations. These are the functions that do real work. `lgrep.c` invokes two command functions: `lkt_dbload` to load data, and `lkt_dbsearch` to search the data.

- Parameter functions create, manipulate, and destroy the input and output parameters that are passed to and from command functions. Most of the functions shown in boldface are parameter functions (for example, `lpar_create_lst`, `lpar_create_record`, `lpar_create_int`, `lpar_destroy`).
- System functions perform DevKit system tasks. The most important are for startup (`lkt_devkit_init`) and shutdown (`lkt_devkit_shutdown`).

In outline, `lgrep.c` performs the following steps:

1. DevKit system initialization.
2. Read the data file,

   Packaging each line as a DevKit record object and

   Appending it to a DevKit list object.
3. Define the field structure for the table.
4. Call the `lkt_dbload` command function with the list of records and a string that names the new database. The `lkt_dbload` command creates and populates the database.
5. Build a list of search parameters (srchpars).
6. Call the function `lkt_dbsearch` with the query, the database name, and the list of search parameters. `lkt_dbsearch` returns a ranked list of record objects (matches).
7. Print some statistics, including the search time, followed by the text of each record in the ranked list.
8. Call the system shutdown function `lkt_devkit_shutdown`.

lgrep.c

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "netrics/devkit.h"
#define TRUE                1
#define FALSE               0
typedef unsigned char       UCHAR;
#define MAX_RECORD_LEN      2048
char    *ProgName;
static void     checkerr( dvkerr_t err, char *function );
int main( int argc, char *argv[] )
{
    int qlen, matchesret, matchnum, numrecs, reclen ;
    char *query, *datafile, dataline[MAX_RECORD_LEN], reckey[10];
    FILE *fp;
    int fldlens[1];
    /* our database will have one field named "text" */
    static const UCHAR *fldnames[1] = { "text" };
    const UCHAR    *rectxt;
    /* Simple lpars */
lpar_t      dbname, qpar, lpar;
    /* Record lpars */
lpar_t      record;
    /* List lpars */
lpar_t      dbpars, reclist, names, srchpars, stats, matches, minfo;
dvkerr_t    err;
    ProgName = argv[0];
    if (argc != 3) {
            fprintf(stderr, "Usage: %s <query> <datafile>\n", ProgName);
            exit(-1);
    }
    /* 1. INITIALIZE THE DEVKIT */
    err = lkt_devkit_init();
    checkerr(err, "lkt_devkit_init");
    query = argv[1];
    qlen  = strlen(query);
    datafile = argv[2];
    if (!(fp = fopen(datafile, "r"))) {
            fprintf(stderr, "%s: Can't open file `%s'\n", ProgName,
                    datafile);
            exit(-1);
    }
    /* 2. READ THE DATA FILE */
    numrecs = 0;
    reclist = lpar_create_lst(LPAR_LST_GENERIC);
    while (fgets(dataline, MAX_RECORD_LEN, fp)) {
            /* 2A: CREATE A RECORD WITH `dataline' AS SEARCHABLE TEXT */
            record = lpar_create_record();
            sprintf(reckey, "%06d", ++numrecs);
lpar_set_record_key(record, (UCHAR *)reckey);
            fldlens[0] = strlen(dataline);
            if (dataline[fldlens[0]-1] == '\n') {  /* Chop newline */
                    fldlens[0]--;
            }
            rectxt = (UCHAR *)dataline;
lpar_set_record_srchtxt(record, rectxt, fldlens[0]);
            /* Record field structure must match database field structure */
lpar_set_record_srchflds(record, fldlens, 1);
            /* 2B: ADD RECORD TO THE RECORD LIST */
lpar_append_lst(reclist, record);
    }
    fclose(fp);
    /* Package file name as the name for the database */
    dbname  = lpar_create_str(LPAR_STR_DBDESCRIPTOR, (UCHAR *)datafile);
    /* 3. DEFINE THE FIELDS */
    dbpars = lpar_create_lst(LPAR_LST_GENERIC);
    lpar = lpar_create_strarr(LPAR_STRARR_FIELDNAMES, fldnames, 1);
lpar_append_lst(dbpars, lpar);
    /* 4. LOAD THE DATABASE */
    err = lkt_dbload(LPAR_NULL, dbname, dbpars, reclist, &stats);
    checkerr(err, "lkt_dbload");
    /* Clean up */
lpar_destroy(reclist);
lpar_destroy(stats);
lpar_destroy(dbpars);
    /* 5. SET UP INPUT PARAMETERS FOR LOOKUP */
    srchpars = lpar_create_lst(LPAR_LST_GENERIC);
    lpar = lpar_create_int(LPAR_INT_MATCHESREQ, 10);
lpar_append_lst(srchpars, lpar);
    /* Add name to a list of names, and package query */
    names = lpar_create_lst(LPAR_LST_GENERIC);
lpar_append_lst(names, dbname);
    qpar = lpar_create_blk(LPAR_BLK_SEARCHQUERY, (UCHAR *)query, qlen);
    /* 6. DO THE LOOKUP */
    err = lkt_dbsearch(LPAR_NULL, names, LPAR_NULL, qpar, srchpars, &stats,
                       &matches, &minfo);
    checkerr(err, "lkt_dbsearch");
    /* 7. PRINT THE RESULTS */
    printf("Query: `%s'\n", query);
    matchesret = lpar_get_lst_num_items(matches);
    printf("Matches returned: %d\n", matchesret);
    lpar = lpar_find_lst_lpar(stats, LPAR_DBL_SEARCHTIME);
    printf("Lookup time: %.2f seconds\n", lpar_get_dbl(lpar));
    printf("\n");
    /* Print the ranked list of records */
    for (matchnum=0 ; matchnum < matchesret ; matchnum++) {
            record = lpar_get_lst_item(matches, matchnum);
            rectxt = lpar_get_record_srchtxt(record, &reclen);
            printf("[%d] `%.*s'\n", matchnum+1, reclen, rectxt);
    }
    /* Clean up */
lpar_destroy(names);
lpar_destroy(qpar);
lpar_destroy(srchpars);
lpar_destroy(stats);
lpar_destroy(matches);
lpar_destroy(minfo);
    /* 8. SYSTEM TERMINATION */
    err = lkt_devkit_shutdown();
    checkerr(err, "lkt_devkit_shutdown");
    exit(0);
}
static void
checkerr( dvkerr_t err, char *function )
{
lpar_t  erritem;
    if (DVKERR(err)) {
            fprintf(stderr, "%s() returned %d (%s)\n", function,
DVKERR(err), dvkerr_get_string(err));
            if ((erritem = dvkerr_get_item(err))) {
lpar_fprintf(stderr, erritem, -1);
            }
dvkerr_clear(err);
            exit(-1);
    }
}
```

On a Unix system, after compiling `lgrep.c` and linking it with the DevKit, run the program as follows:

```
lgrep dictioneryprblembounds cspapers.lst
```

where `dictioneryprblembounds` is an inexact search query, and `cspapers.lst` is a text file of single-line records. The output looks like this, where some long lines are abbreviated:

```
Query: 'dictioneryprblembounds'
Matches returned: 10
Lookup time: 0.01 seconds
[1] `Andersson, Optimal Bounds on the Dictionary Problem|LNCS|401|1989'
[2] `..., Upper and Lower Bounds for the Dictionary Problem|LNCS|318|1988'
[3] `Sundar, A Lower Bound for the Dictionary Problem ...|FOCS|32|1991'
[4] `... Concurrency Methods for the Dictionary Problem: A Survey|...'
[5] `Fich, Lower Bounds for the Cycle Detection Problem|STOC||1981'
[6] `Fich, Lower Bounds for the Cycle Detection Problem|JCSS|26|1983'
[7] `... Bounds for Communication Complexity Problems|ACTAINF|28|1991'
[8] `... Dictionary-Matching on Unbounded Alphabets ...|LIBTR||1993'
[9] `... Dictionary-Matching on Unbounded Alphabets...|ALGORITHMS|18|1995'
[10] `Willard, Lower Bounds for Dynamic Range Query Problems ...|ICALP||1986'
```

The `lgrep` program loads data from a file to service a single query. Most DevKit applications run as services that maintain searchable data in memory, concurrently processing matching or machine learning scoring requests and database administration requests. In either case, the steps in assembling and passing parameters for the `lkt_dbload` and `lkt_dbsearch` functions are the same.

That is how simple it is to use DevKit.

The DevKit Header File devkit.h

The header file `devkit.h` has all the type definitions, macros, and function prototypes you need for the DevKit. It is normally installed in the netrics subdirectory of an include directory on your standard include path.

Compiling and Linking Programs With the DevKit

The DevKit can be used only on systems that fully support ANSI-compliant C and the POSIX threads interface. Depending on your system, you might need to specify libraries explicitly when linking a program with the DevKit. On a Linux system, for example, here is how you would link lgrep:

```
cc -o lgrep lgrep.c -lnetricsdvk -lm -lpthread
```

The argument `-lnetricsdvk` works only if the library `libnetricsdvk.a` is installed in a directory on your standard library search path.

Input and Output Parameter Types

When you looked at the source code for the `lgrep` program in A Simple Example, you probably noticed the important role of the generic parameter type `lpar_t`. This type provides a clean, extensible interface for DevKit command functions. A value of type `lpar_t` is called an lpar, and is pronounced `ell-par`. Most DevKit command functions take lpars as arguments. An `lpar` might be a list of other `lpars`, so you can pass arbitrarily many values as each argument.

This section describes how to create, build, and destroy lpars. Read this section to know everything about passing input parameters to DevKit commands and processing their output.

Building the Input: lpars

An lpar is a value and an integer code, called the lpar ID, that identifies the value's data type. Lpar ID's are predefined in the header file `devkit.h`. Here is a partial listing of that file:

```
enum lparid_en {
        /* ... */
        LPAR_STR_DBDESCRIPTOR,     /* Database name */
        LPAR_STR_LAYOUT,           /* Layout name - deprecated*/
        /* ... */
        LPAR_INTARR_SELECTFLDS,     /* Indexes of fields to search */
        LPAR_INT_ALIGNFIELD,        /* Initial alignment field IN FIELDSET */
        LPAR_BLK_SEARCHQUERY,       /* Search query */
        LPAR_LST_MULTIQUERY,        /* Multi-query */
        /* ... */
};
```

In this example, you created a string lpar, a Boolean lpar, a signed integer lpar, a list lpar, and a byte block lpar. Lpar creator functions always have the form `lpar_create_< >`, where <*type*> is the value type of the lpar. See the header file listing in Table of DevKit Error Codes and Strings for prototypes of the creator functions for the lpar types.

Each of the enum identifiers shown above is an lpar ID. The second component of the identifier part between the two underscores represents the type of the value. For example, the ID LPAR_INT_ALIGNFIELD takes an integer value; the ID LPAR_STR_DBDESCRIPTOR takes a null-terminated character string as a value. Creating an lpar with a value that does not agree with its lpar id is a programming error.

Here is a list of all lpar value types used in the DevKit, with the corresponding enum prefixes (additional types used internally that are not used in the interface):

LPAR value types used in the DevKit

Lpar value type	Id macro prefix
Boolean (true/false)	LPAR_BOOL_<*name*>
signed integer	LPAR_INT_<*name*>
array of signed integers	LPAR_INTARR_<*name*>
double-precision oat	LPAR_DBL_<*name*>
array of double-precision oats	LPAR_DBLARR_<*name*>
Null-terminated character string	LPAR_STR_<*name*>

Lpar value type	Id macro prefix
array of null-terminated character strings	LPAR_STRARR_*<name>*
byte block	LPAR_BLK_*<name>*
array of byte blocks	LPAR_BLKARR_*<name>*
list of more lpars	LPAR_LST_*<name>*

You assign a value to an lpar when you create it. Call the lpar creator function for the value type, passing it the lpar id and the value.

Note: The null-terminated string values of LPAR_STR_ and LPAR_STRARR must never contain a new-line character. A byte block however might contain any character, including new-line characters.

Note: You cannot initialize the value of a list type lpar. It is created empty and must be filled later.

The creator function returns a new lpar containing the value you specified. The following is an example of this:

```
typedef unsigned char UCHAR;
/* ... */
{
    int        querylen, nfields;
    UCHAR      *query;
    lpar_t     lpar1, lpar2, lpar3, lpar4;
    /* ... */
    query = malloc(querylen);
    /* ... initialize the query here ... */
    lpar1 = lpar_create_str(LPAR_STR_DBDESCRIPTOR, (UCHAR *)"My Database");
    lpar2 = lpar_create_int(LPAR_INT_DBNUMFIELDS, nfields);
    lpar3 = lpar_create_lst(LPAR_LST_QUERYLET);
    lpar4 = lpar_create_blk(LPAR_BLK_SEARCHQUERY,query,querylen);
    /* ... pass lpars as input to a command function ... */
lpar_destroy(lpar1);
lpar_destroy(lpar2);
lpar_destroy(lpar3);
lpar_destroy(lpar4);
```

```
    free(query);
}
```

In this example, you created a string lpar, a Boolean lpar, a signed integer lpar, a list lpar, and a byte block lpar. Lpar creator functions always have the form `lpar_create_<type>`, where <type> is the value type of the lpar. See the devkit.h file in TIBCO Patterns installation for prototypes of the creator functions for the lpar types.

After creating an lpar, you can access its integer id with the function lpar_id. See examples of this in the section on DevKit output. The function lpar_value_type returns the value type of an lpar:

```
      lpar_t         lpar;
      lpar_type_t    type;
      /* ... */
      type = lpar_value_type(lpar);
```

The type returned by `lpar_value_type` can be any one of the following:

- LPAR_TYPE_BOOL (boolean)
- LPAR_TYPE_INT (32-bit, integer)
- LPAR_TYPE_INTARR (array of integers)
- LPAR_TYPE_LONG (64-bit, integer)
- LPAR_TYPE_DBL (double)
- LPAR_TYPE_DBLARR (array of doubles)
- LPAR_TYPE_STR (nul-terminated, string)
- LPAR_TYPE_STRARR (array of strings)
- LPAR_TYPE_LST (list of lpars)
- LPAR_TYPE_BLK (byte block)
- LPAR_TYPE_BLKARR (byte block array)
- LPAR_TYPE_RN

Since lpar objects are allocated by the DevKit dynamically, their storage must be released after use. The function `lpar_destroy` releases an lpar of any value type. When you create `lpar4`, it is initialized with a copy of the byte block query.

Note: The same holds for lpars containing integer arrays, double-precision arrays, and strings.

Using `lpar_destroy` to destroy `lpar4` does not remove the need to free query.

Most DevKit commands expect list type lpars as parameters. Non-list lpars are usually created to be attached to one or more lists, which are then passed to DevKit commands.

Typically, create a list lpar first, and then the member lpars, appending each lpar to the list. The following is an example of this:

```
typedef unsigned char UCHAR;
* ... */
{
    int        querylen;
    UCHAR      *query;
    lpar_t     lpar, lpar2;
    lpar_t    list1, list2;
    /* ... */
    list1 = lpar_create_lst(LPAR_LST_GENERIC);
    lpar = lpar_create_str(LPAR_STR_DBDESCRIPTOR, (UCHAR *)"My Database");
lpar_append_lst(list1, lpar);
    lpar = lpar_create_int(LPAR_INT_DBNUMFIELDS, 4);
lpar_append_lst(list1, lpar);
    list2 = lpar_create_lst(LPAR_LST_GENERIC);
    lpar = lpar_create_lst(LPAR_LST_QUERYLET);
    lpar2 = lpar_create_blk(LPAR_BLK_SEARCHQUERY,query);
lpar_append_lst(lpar, lpar2); /* Put the query in the querylet */
lpar_append_lst(list2, lpar);
    /* ... Pass the lists to a command function ... */
lpar_destroy(list1);
lpar_destroy(list2);
    free(query);
                                    }
```

In this realistic example, the code is compact and simpler. Rather than declaring lpar variables one by one, declare two temporary lpars and reuse them after adding each lpar to a list. Then pass the lists to a command function.

Note: If an lpar is appended to a list that already contains one or more lpars with the same lpar ID, and only one lpar of that ID is expected, the value of the most recently added lpar supersedes that of previous instances of that lpar ID.

To clean up, destroy the lists and free query, if it was dynamically allocated. Destroying a list destroys all lpars in that list. When you `lpar_destroy` a list, all its elements are destroyed recursively. Do not add an lpar to more than one list, or destroy it while it is a member of a list.

When you need to add an lpar to more than one list, copy it with the function `lpar_copy`(lpar) and add the copy.

Building the Input: Data Records

There is a special list type of lpar used to construct and manipulate data records:

The record is a child record if the parent key string is present. The parent key string is a null-terminated character string that is used to identify the parent record of the child record. The value is the key string value of the parent record. The record is a child record if, and only if, the parent key string is present. The value of the parent key string must not be null or empty.

A record is a list object possessing the following entries:

- Key string (mandatory)
- Searchable text (mandatory)
- Record field structure (mandatory)
- Parent key string (optional)

The key string is a null-terminated character string used to identify the record in record update, delete, or fetch operations. It must be at least one-character in length without counting the terminating null character.

The searchable text must be Unicode UTF-8 encoded text data. It is the data for all fields of the record given as a single concatenated text block.

The field structure defines how the search text block is to be split into fields by defining the length of each field within the searchable text in bytes.

Every data record passed to the DevKit must have the three mandatory entries. If the optional parent key entry is given, it must be populated.

Each record entry is itself an lpar which is a member of the record. Do not use `lpar_append_lst` to fill a record list! For efficiency, the DevKit expects record entries to be stored in a predefined order. The DevKit has functions for setting record entries that create the entry lpar and insert it in its proper place in the record.

The following segment of the code creates and populates either a standard record or a child record, and then adds it to a list. It assumes that the input records have the structure shown, and that all data fields are null-terminated UTF-8 encoded strings.

```
typedef unsigned char UCHAR ;
...
{
    int srchtxtlen;
    int fldlens[2];
    int srchtxt[1024];
    lpar_t reclst;
    lpar_t record;
    struct parent_child_rec {
        const char *key; /* record key */
        int is_parent_record; /* 1 if parent, 0 if child */
        /* for parent records. */
        const char *first_name;
        const char *last_name;
        /* for child records. */
        const char *parent_key;
        const char *street;
        const char *city;
    } cur_record;
    reclst = lpar_create_lst(LPAR_LST_GENERIC);
    for ( /* all records */ ) {
        /* ... */
        if ( cur_record.is_parent_record ) {
            /* Get parent record data. */
            /* get field lengths and create searchable text block */
            fldlens[0] = strlen(cur_record.first_name);
            fldlens[1] = strlen(cur_record.middle_name);
            strcpy(srchtxt, cur_record.first_name);
            strcat(srchtxt, cur_record.last_name);
            srchtxtlen = fldlens[0] + fldlens[1];
/* Create the standard record. */
            record = lpar_create_record();
            lpar_set_record_key(record, cur_record.key);
            lpar_set_record_srchtxt(record, srchtxt, srchtxtlen);
            lpar_set_record_srchflds(record, fldlens, 2);
        } else {
            /* Get child record data. */
            /* get field lengths and create searchable text block */
            fldlens[0] = strlen(cur_record.street);
            fldlens[1] = strlen(cur_record.city);
            strcpy(srchtxt, cur_record.street);
            strcat(srchtxt, cur_record.city);
            srchtxtlen = fldlens[0]+fldlens[1] ;
/* Create the child record. */
            record = lpar_create_child_record();
            lpar_set_record_key(record, cur_record.key);
            lpar_set_record_parent_key(record, cur_record.parent_key);
            lpar_set_record_srchtxt(record, srchtxt, srchtxtlen);
            lpar_set_record_srchflds(record, fldlens, 2);
        }
        lpar_append_lst(reclst, record);
    }
    /* ... Pass list of records to a command function ... */
    lpar_destroy(reclst);
}
```

You create a record list and add record objects to it in the same simple way as for other types of lpars. And when you destroy the list, you thereby destroy its elements.

Note that although the searchable text data is in UTF-8, the length values are in bytes, not characters. UTF-8 is a convenient encoding of Unicode data to use in "C" as all of the standard "C" string functions work with UTF-8 data as it has no embedded \0- (null) bytes. But the functions still work in terms of bytes and not characters.

Note: Another reason UTF-8 is a convenient encoding is that the standard 7-bit ASCII character set is also valid UTF-8. This means applications that were dealing with ASCII data are also effectively UTF-8. Note this is not true for latin-1 (ISO-8859-1) encoded data that contains characters in the 8 Bit range. These must be converted to UTF-8 encoding.

Since the key and searchable text are copied by the `lpar_set_record_`*<entry>* functions, the data passed to them is not destroyed by `lpar_destroy`. If it is dynamically allocated, you must free it explicitly.

Invoking TIBCO Patterns Matching (lkt_dbsearch) shows how to specify which fields to search and which to ignore.

Processing the Output of DevKit Commands

Most DevKit command functions return lists. The following example illustrates two ways to access them:

```
typedef unsigned char UCHAR;
/* ... */
{
    int       i, item, numitems, recnum, numrecs, arrlen;
    int       srchtxtlen;
    lpar_t    query, lpar;
    lpar_t    name, srchpars, stats, matches, minfo;
    dvkerr_t  dvkerr;
    const int    *pvals, *dvals;
    const double *vvals;
    const UCHAR  *srchtxt;
    const lpar_t *records;
    const lpar_t *recinfo;
    /* ... Build table-name list, query, and search parameter list ... */
    dvkerr = lkt_dbsearch(LPAR_NULL, names, LPAR_NULL, query, srchpars,
&stats,
                          &matches, &minfo);
    /* ... Check error return from lkt_dbsearch, and clear dvkerr ... */
    numitems = lpar_get_lst_num_items(stats);
    for (item=0 ; item < numitems ; item++) {
            lpar = lpar_get_lst_item(stats, item);
       switch (lpar_id(lpar)) {
         case LPAR_INT_MATCHESRET:
           printf("%d matching records\n", lpar_get_int(lpar));
           break;
         case LPAR_DBL_SEARCHTIME:
           printf("Lookup took %.2f seconds\n",lpar_get_dbl(lpar));
           break;
         default: /* (We're not interested in other statistics) */
           break;
            }
    }
    records = lpar_get_lst_item_array(matches, &numrecs);
    for (recnum=0 ; recnum < numrecs ; recnum++) {
            printf("[%2d] ", recnum+1);
            srchtxt = lpar_get_record_srchtxt(records[recnum], &srchtxtlen);
            printf("`%.*s'\n", srchtxtlen, (char *)srchtxt);
            printf("Key: `%s'\n", lpar_get_record_key(records[recnum]);
            if (lpar_get_record_parent_key(records[recnum]) != NULL) {
                   printf("Parent Key: `%s'\n",
                       lpar_get_record_parent_key(records[recnum]));
            }
            recinfo  = lpar_get_lst_item(minfo, recnum);
            numitems = lpar_get_lst_num_items(recinfo);
            for (item=0 ; item < numitems ; item++) {
                    lpar = lpar_get_lst_item(recinfo, item);
                    switch (lpar_id(lpar)) {
                    case LPAR_DBL_MATCHSCORE:
                            printf("Match score: %.4f\n", lpar_get_dbl
(lpar));
                            break;
                    case LPAR_DBLARR_V2V:
                            vvals = lpar_get_dblarr(lpar, &arrlen);
                            printf("V array:");
                            for (i=0 ; i < arrlen ; i++) {
                                    printf(" %f", vvals[i]);
```

```
                                }
                                printf("\n");
                                break;
                        case LPAR_INTARR_V2P:
                                pvals = lpar_get_intarr(lpar, &arrlen);
                                printf("P array:");
                                for (i=0 ; i < arrlen ; i++) {
                                        printf(" %d", pvals[i]);
                                }
                                printf("\n");
                                break;
                        case LPAR_INTARR_V2D:
                                dvals = (lpar, &arrlen);
                                printf("D array:");
                                for (i=0 ; i < arrlen ; i++) {
                                        printf(" %d", dvals[i]);
                                }
                                printf("\n");
                                break;
                        /* ... other cases ... */
                        }
                }
        }
        lpar_destroy(names);
        lpar_destroy(query);
        lpar_destroy(srchpars);
        lpar_destroy(stats);
        lpar_destroy(matches);
        lpar_destroy(minfo);
}
```

The command function `lkt_dbsearch` returns output in three lists. The first (stats) is a list of lpars, the second (matches) is a list of records, and the third (minfo) is a list of lpar lists (one list per record returned in matches).

Study the way in which the items in the stats list and in the minfo list are accessed. The function `lpar_get_lst_num_items` returns the number of items in a list. The list items can then be accessed with calls to the function `lpar_get_lst_item`, which returns a valid list item whenever its second argument falls in the interval [0, N - 1], where N is the number of items in the list, and the value `LPAR_NULL` otherwise.

Note that the minfo list is a list of lists - one list for each record returned in the matches list.

The second way of accessing all items in a list is illustrated by our handling of the matches list, which is a list of record lpars. We use the function `lpar_get_lst_item_array` to fetch a pointer directly to the array of list items. The nice thing about this method is that once you've gotten the pointer to the actual list item array, you can use normal C square-bracket array access to get the individual items. Study the way in which we access the items in the matches list.

Note: There is still another way of accessing all items in a list, not shown in our example. This is a loop similar to the following:
for (i=0 ; (lpar = lpar_get_lst_item(list, i)) != LPAR_NULL ; i++) {
 /* process lpar here... */
 }

Once you've gotten a list item, by whatever method, you can access the value or values contained in it with the appropriate `get` or selector function. Switching on the lpar ID (returned by the function `lpar_id`) is the normal method for determining which selector function must be used for an lpar. The selector functions for lpar values parallel exactly the functions that create them: an lpar whose value is an integer (ID `LPAR_INT_`<*name*>) is created with the function `lpar_create_int`, which also assigns it its value. That value can be accessed at any time with the function `lpar_get_int`. Other lpar selectors work similarly. When an lpar contains an array (of integers or doubles), array items might be either accessed singly or, as illustrated above, with a function such as `lpar_get_intarr` (`lpar_get_dblarr` for doubles) that returns an array pointer that can then be dereferenced with square brackets. Study how the array items in `vvals`, `pvals` and `dvals` are accessed in the example.

Retrieving the components of a record works in a similar way. The selector functions for record components parallel exactly the functions that set those components. See the devkit.h include file from your TIBCO Patterns installation for the prototypes of all lpar and record selector functions.

Note: Unlike the `lpar_create_`<*type*> functions and the `lpar_set_record_`<*attribute*> functions, all of which copy data, the get functions for DevKit objects do not give you a copy of the contents of the object, but instead a pointer to the actual data inside the object. You can neither modify this data directly, nor free it with a call to free. This is why the types of the return values of object selector functions are qualified with the const type modifier. See the devkit.h include file in TIBCO Patterns installation.

Note: You can only free the object itself by calling `lpar_destroy`.

The lpar_find_lst_lpar Function

There is still another way of accessing items in a list lpar. You can call the function `lpar_find_lst_lpar` with a specific lpar id, and the function return the lpar in the list that has that id, or the value `LPAR_NULL` if such an lpar was not found.

Note: If there is more than one lpar in a list with the same id, `lpar_find_lst_lpar` returns the last instance of that id in the list.

This is especially convenient when only some lpars in a list (for example, a list of statistics) are of interest to you. The loop that accessed the stats list in our example could be replaced with these four lines:

```
lpar = lpar_find_lst_lpar(stats, LPAR_INT_MATCHESRET);
printf("%d matching records\n", lpar_get_int(lpar));
lpar = lpar_find_lst_lpar(stats, LPAR_DBL_SEARCHTIME);
printf("Lookup took %.2f seconds\n", lpar_get_dbl(lpar));
```

We chose not to test the return value against `LPAR_NULL`, since the stats list returned by `lkt_dbsearch` (in the non-error case) always contains these lpars.

The lpar returned by `lpar_find_lst_lpar` is not a copy of the list lpar, it is the same lpar, using the same memory, as the lpar in the list. That means that this lpar must not be destroyed with `lpar_destroy`. It is automatically destroyed when the list is destroyed. This also means that any lpar returned by `lpar_find_lst_lpar` must not be used after the associated list is destroyed.

Destroying DevKit Command Output

The application that uses the DevKit is responsible for destroying all DevKit command output. Since all command output is in the form of lists, simply calling `lpar_destroy` for each list is sufficient. Note that even a list of lists (for more information, see Processing the Output of DevKit Commands) can be destroyed with a single call to `lpar_destroy`.

Printing Input and Output Objects

For debugging purposes, it's useful to be able to print the contents of an lpar. The DevKit provides the following function that prints to a file stream:

```
void lpar_fprintf( FILE *stream, lpar_t lpar, int nicechar );
```

The object parameter can be any type of lpar. The function operates recursively, printing the contents of any nested list structures. Indentation is used to depict nesting relationships.

The `nicechar` parameter controls how the content of byte blocks is printed. If nicechar is nonnegative, every non-printable byte (as defined by the C standard library function `isprint`) is replaced with (`unsigned char`) `nicechar` in the printed output. If `nicechar` is negative, this replacement is not done, which can cause trouble for some terminal emulation programs. However, if you are outputting UTF-8 encoded data `nicechar` must always be negative as the `isprint` function might interpret bytes of the UTF-8 character as non-printable.

Note: To view the output your terminal or viewer must be configured to handle UTF-8 encoded text.

An Important Word (and a Warning) About Error Conditions

Now that you know everything there is need to know about DevKit input and output, it's time to discuss DevKit command functions in detail. But first, we need to have a brief but very important discussion of the way in which DevKit commands return error-related information.

As you'll see from the command function prototypes, all DevKit command functions return an error handle of type `dvkerr_t`. You can declare variables of this type statically, and use them as follows:

```
    {
        dvkerr_t err;
        lpar_t erritem;
        /* ... */
        err = lkt_dbsearch( /* parameters */ );
        if (DVKERR(err)) {
                fprintf(stderr, "dbsearch returned error %d (%s)\n",
                        DVKERR(err), dvkerr_get_string(err));
                if ((erritem = dvkerr_get_item(err))) {
                        lpar_fprintf(stderr, erritem, -1);
                }
                dvkerr_clear(err);
        }
        /* ... */
        err = lkt_dbdelete( /* parameter */ );
        if (DVKERR(err)) {
                fprintf(stderr, "dbdelete returned error %d (%s)\n",
                        DVKERR(err), dvkerr_get_string(err));
                if ((erritem = dvkerr_get_item(err))) {
                        lpar_fprintf(stderr, erritem, -1);
                }
                dvkerr_clear(err);
        }
        /* ... */
    }
```

Note the static declaration of err in this example and its assignment to the return value of two DevKit command functions.

Note: You can also allocate an error handle dynamically, but you are then responsible for eventually freeing the storage.

As the example shows, an error handle can be reused after being cleared with the function `dvkerr_clear`.

When a DevKit command returns, the error handle must be tested with the macro DVKERR, which returns a nonnegative integer error code.

Note: All error codes returnable by DevKit commands begin with the prefix DVK_ERR_, and are defined in the header file devkit.h. See your copy of `devkit.h` for a complete list of error codes, and their general meanings.

The zero code (DVK_OK) is reserved for successful command termination, thus, the first invocation of DVKERR in our example works as a true/false test: was there an error, or not. If there was an error, our example prints the error code, and then an explanatory string which is the return value of the function `dvkerr_get_string`.

The string returned by `dvkerr_get_string` is a static string explaining the general nature of the error type. To supply you with as much useful information as possible, an error item might also be returned in the error handle. This error item (if it exists) is a copy of the lpar that caused or occasioned the error: an lpar with an illegal or out-of-range value, a record containing an empty searchable text attribute, and so on.

Since the error item is a dynamically allocated object, you must call the function `dvkerr_clear` to release the storage associated with the item, whether or not you choose to access the item with `dvkerr_get_item`. It is good practice to call `dvkerr_clear` after every DevKit command invocation, or at least after all those that return a non-zero error code.

Note: Calling `dvkerr_clear` is unnecessary, but harmless, in the case of command success.

Warning: Reusing error handles without clearing them causes memory leak in your program. That memory leak is equal to the storage associated with any dynamically allocated error items. This storage can easily build up to a critical level in long-running applications or servers.

The TIBCO Patterns Table

Before describing the commands to load and search a TIBCO Patterns database, you must be clear about the logical structure of the database. For historical reasons, throughout the document the term *database* is used as a synonym for table. But in the following section the term *database* refers to the entire collection of tables, with the term *table* referring to a single table in that collection

Note: The TIBCO Patterns database is not a persistent data store, it must be reloaded each time the server is restarted. It is designed to work with your current Database Management System (DBMS) or other persistent store, and not to replace it.

The TIBCO Patterns database is similar to a DBMS. It consists of a set of tables. Each table consists of fielded records, every record in the same table having the same number of fields. Every field in a table has a field name associated with it. In TIBCO Patterns the number of fields, the field names, and the field types are established at the time the table is created, fields or field types cannot be changed after a table is created.

TIBCO Patterns has the following three types of tables:

- *Standard Tables:* A standard table is not associated with any other table.
- *Parent Tables:* A parent table can be associated with any number of child tables.

- *Child Tables:* A child table is always associated with a single parent table. Records in a child table can be linked to records in the associated parent table.

The association of parent and child tables forms a star schema. TIBCO Patterns supports only a star schema, it does not support many to many or multi-level schema.

The table type is established when a table is created and cannot be changed. The association between a child table and a parent table is established when the child table is created and cannot be changed after the table is created. Parent tables and child tables must use the Gram Index Prefilter (GIP) (see The GIP Prefilter - Usage Details).

Every record in a standard or parent table must have a unique key value. The key value is a null-terminated string value. This key value is distinct from the field values for the record. Records in a standard or parent table cannot contain a parent key field.

A child table contains two record types:

- Standard Records: These are identical to records in a Parent or Standard table.
- Child Records: These records are linked to a parent record in the associated parent table. They contain a parent key field in addition to the record key field. The parent key field links the child record to its parent record. It contains the record key value of the parent record (in Relational Database Management System (RDBMS) terms it is the foreign key field). The parent key field is distinct from the field values for the record. For child records, the combination of the record key and parent record key must be unique.

Every field in the TIBCO Patterns table has a field type. The field type must be one of the following types.

- LKT_FLD_TYPE_KEY (2) This is only used in the field types header line of record files. It is the field type for the key field. Currently the key field cannot appear as a data field in the table, so a table field never has a field type of LKT_FLD_TYPE_KEY. See Reading a Database From a File (lkt_dbread) for more on record files.
- LKT_FLD_TYPE_SRCHTEXT (5) This is the default field type and by far the most commonly used. This holds UTF-8 encoded text data. Fields of this type can be used in both inexact matching and exact matching. Predicates are logical expressions similar to SQL select clauses. See Predicate Expressions for a description of predicates and their use
- LKT_FLD_TYPE_TEXT (4) These fields contain UTF-8 encoded text data that is NOT searched. You cannot specify this field in inexact matching, but it can be used in predicate tests. Although it is not used in matching it is returned with the record data in searches. This field type is useful for fields that are only used in predicate tests, such as a gender indicator field, or as a means of returning foreign key values with the record that might be useful in linking the data returned by a search to additional data in the permanent data store. Another use is to save extra calls to the permanent data store by storing data which needs to use the results of a search, but is not needed in the search. Using this field type

for such fields as opposed to LKT_FLD_TYPE_SRCHTEXT can reduce memory usage and database load times.

- LKT_FLD_TYPE_INT (6) This field type holds an integer value. The range of valid integers is as defined for the "C" int type on the machine the TIBCO Patterns servers are running on. Note that the value of this field is still transmitted as part of the searchable text data of the LPAR record. This text must be an integer string as recognized by the "C" atoi(3) function. Following two special values are there for this field type and all of the other non-text field types:
  - **empty** indicates there was no data (field length was zero) in the field.
  - **invalid** indicates there was data in the field, but it could not be translated to the indicated field type. For example, a string value of apple in a field of type LKT_FLD_TYPE_INT. When records are returned the invalid value for non-text fields is encoded as: *** for int and float fields, ***invalid date*** for date fields and ***invalid date time*** for date time fields.

  Fields of this type are primarily useful in predicate expressions. They are more efficient in predicate expressions and use less space to store than keeping the value as a text string.
- LKT_FLD_TYPE_FLOAT (8) This field type holds a floating point value. The range of valid values is as defined for the "C" double type on the machine the TIBCO Patterns server is running on. As for the integer field type the value is sent as text in the searchable text data of the LPAR record. The format is as recognized by the "C" atof(3) function. It might take on the two special field values: empty and invalid.
- LKT_FLD_TYPE_DATE (10) This field type holds a date value. That is a month, day, year. The value is transmitted as text. A wide array of formats is recognized including:
  - mm/dd/yyyy
  - mm/dd/yy
  - yyyy/mm/dd
  - yyyy-mm-dd
  - yyyymmdd
  - mm/dd (year default to current year)
  - dd-MON-yyyy
  - dd MON yyyy
  - MON dd, yyyy
  - MON dd (year defaults to current year)

  where mm is the month number 1-12, dd is the day of the month 1-31, yyyy is the four-digit year, yy is a two-digit year. Values greater than the current two digit year are assumed to refer to the previous century, otherwise the current century is assumed. MON is a month's name. The following month names are recognized (letter case insensitive): january, jan, february, feb, march, mar, april, apr, might, june, jun, july, jul, august, aug, september, sep, october, oct, november, nov, december, dec.

When returned date values are always encoded in the format: yyyy/mm/dd.

- LKT_FLD_TYPE_SRCHDATE (11) This is the same as LKT_FLD_TYPE_DATE with the important difference that the data is indexed by the GIP prefilter (see The GIP Prefilter - Usage Details). This improves matching accuracy where a custom date query is used in a search (see Simple Query, the LPAR_INT_CUSTOMSCORE parameter). This also allows queries that consist only of a custom date query.
- LKT_FLD_TYPE_DATET (12) The Date-Time field type is just like the date field type except it also stores the time. It accepts all the same date formats as date fields but also accepts an appended time value in the form: hh:mm:ss where hh is the 24 hour value (00-23), mm is the minute value 00-59 and ss is the second value (00-59). When returned date-time values are always encoded in the format: yyyy/mm/dd hh:mm:ss.nnn where nnn is milliseconds.
- LKT_FLD_TYPE_ATTRIBUTES (15) This is the container for the Variable Attributes associated with a record. If a table is to hold Variable Attributes the last field in the field set must be of this type. Only the last field of a field set might have this type. For more information on Variable Attributes and how to use fields of type LKT_FLD_TYPE_ ATTRIBUTES see Variable Attributes - Usage Details.

The default field type is LKT_FLD_TYPE_SRCHTEXT.

Associated with every field of type LKT_FLD_TYPE_SRCHTEXT and LKT_FLD_TYPE_ATTRIBUTES is a character map. This defines how characters are mapped before inexact matching is performed. Each field can have a different character map. Character maps do things such as letter case folding, removal of punctuation, translating the various types of white space to a common value, normalizing characters as defined by the Unicode standard and any special character to character mapping desired. For a more complete description of character maps see Character Mapping. There is a default character map that is applied if one has not been specified. The character map associated with a field is established at the time the table is created and cannot be changed.

Matching Accelerators (prefilters)

A Matching Accelerator speeds matching by quickly filtering out many table records from consideration before performing a full match calculation. Thus they are generally referred to as prefilters.

GIP, SORT, and PSI are the three prefilters that provide a large boost in search time performance.

It is strongly recommended that for any large table one of these prefilters be used. Large being a somewhat vague term, being dependent on many different factors such as number of fields, size of fields, type of queries, and performance constraints, but can be roughly construed as 100,000 records or more.

Each has certain advantages: the GIP prefilter is far more flexible and easier to use, SORT takes less memory, and can give a greater speed boost, especially on very large tables. PSI is similar to SORT, but can be more accurate in situations where SORT might not perform well. Generally, the SORT and PSI prefilters are designed for use in record matching where complete records are to be matched on a field by field basis. The SORT filter is not intended for situations where you are doing general queries that might have far less information in the query than contained in the complete record. The PSI prefilter might work well where a limited number of query types are in use. Both SORT and PSI have slightly slower record operations than GIP, so they might be inappropriate when real-time updates on large tables are required. SORT and PSI do not index variable attributes, and therefore, might not be suitable for use with tables containing variable attributes. The GIP prefilter works well for almost any kind of query and is completely transparent in use. It also performs well for record updates on any size table, but query performance on very large tables is significantly lower than the SORT and PSI prefilters. GIP uses more memory than SORT, and PSI uses more memory than GIP.

The default prefilter can be selected via the command line argument. See *TIBCO Patterns Installation* for details on command line arguments. If no prefilter is specified on the command line, the default is the GIP prefilter. This default prefilter applies only when a table is initially created. The default prefilter for queries is the prefilter used by the table being queried.

The GIP Prefilter - Usage Details

If the GIP prefilter is the default prefilter, any table you load is automatically configured for matching acceleration with the GIP prefilter at load time. If for any reason you do not wish a particular table to be so configured simply include the Boolean parameter `LPAR_BOOL_GIPSEARCH` with a value of false in the `dbpars` parameter list for the command `lkt_dbload`. The GIP prefilter must be created when the table is created, it cannot be added to or removed from an existing table.

Whenever database statistics are returned for a GIP prefilter enabled table, the value (`LPAR_INT_DBIDXTOTALKBYTES`) gives the total size of the GIP prefilter related data structures associated with this table.

Note: If predicate indexes also exist for the table this value also includes the space used by the predicate indexes (see Partition Indexes).

If the GIP prefilter is enabled for a table, by default searches on that table use the GIP prefilter. You can explicitly turn acceleration off for all tables by including the `LPAR_BOOL_GIPSEARCH` parameter with a value of false in the srchpars list you pass to `lkt_dbsearch`. This disabling of acceleration is effective for the current query only.

Whenever a `lkt_dbsearch` is done, the same Boolean parameter LPAR_BOOL_GIPSEARCH is returned in the stats list. This parameter is true if the GIP prefilter was employed, false otherwise.

If you are running Patterns Server on a Linux platform, and your hardware has qualified GPU units installed, you can configure your GIP prefilter to take advantage of the GPU processors to speed processing. See `lkt_dbload` command for a description of the related option arguments. For more information about GPU processing, see the *TIBCO® Patterns Concepts Guide.* For more information about enabling GPU processing when starting a server, see the *TIBCO® Patterns Installation Guide*.

The SORT Prefilter - Usage Details

A SORT prefilter uses a collection of sorted indexes. A single sorted index is called an encoding. When creating a table that is to use a SORT prefilter, the collection of encodings to be used might be defined. If you do not specify them explicitly, a default set is created. The default set is reasonable for many applications. If it is not appropriate for your application, contact your TIBCO representative for advice on which encodings to use for your specific application.

To create the sort filter, pass an LPAR_LST_ENCODINGS in the dbpars parameter of the `lkt_dbload` function. Each element in the LPAR_LST_ENCODINGS lpar must be an LPAR_LST_ENCODING lpar. Within the LPAR_LST_ENCODING, there are currently two allowable lpars.

The first is LPAR_STRARR_FIELDNAMES. Each string array is a list of field names which is used for a given sort encoding. The first field name is the primary sort key, the second field is secondary, and so on. In general, you must have as many encodings as the number of fields in the table and use all important fields as the primary sort key in at least one encoding. Contact your TIBCO representative for further information on which sort of encodings to use for your specific application.

The second is LPAR_INTARR_BKWDSFLDS. Fields can also be sorted backwards (from the last characters in the field to the first). Important fields can and must be used as primary sort keys in both their forward and reverse directions. The backwards fields array, if given, must be the same length as the field names array. The integer is set to one of the field in the corresponding position in the field names array is to be sorted in the backwards direction and zero for the forward direction. If this value is not given all fields are sorted in the forward direction.

To use the SORT prefilter in a search, you might pass either LPAR_BLKARR_SORTLOOKUPFIELDS in the srchpars parameter or the LPAR_LST_DEDUPQUERY parameter in the dbpars of the `lkt_dbsearch` function. If neither parameter is passed, the field values is inferred from the query. If the field value inference cannot be done, an error is returned. As the SORT prefilter is predominantly used in a record linking scenario it expects a set of field values corresponding to the fields of the table. The LPAR_BLKARR_SORTLOOKUPFIELD parameter specifies these field values explicitly. The length of the array must be the same as the number of fields in the table and the order of the values must correspond to the field order when the table was defined. The LPAR_LST_DEDUPQUERY parameter in the dbpars can be used as a shortcut when a record in the same table is being matched against other records in the table. The list must contain a single LPAR_STR_RECKEY value. The fields of the record within the table to be searched identified by the key are used as the SORT prefilter look up fields.

If the SORT prefilter is enabled for a table, searches on that table use the SORT prefilter unless it is disabled using the `LPAR_BOOL_SORTSEARCH` option. This disabling of acceleration is effective for the current query only.

Whenever a `lkt_dbsearch` is done, the same Boolean parameter, LPAR_BOOL_SORTSEARCH, is returned in the stats list. This parameter is true if the SORT prefilter was employed, false otherwise.

Whenever database statistics are returned for a SORT prefilter enabled database, the value (LPAR_INT_DBIDXTOTALKBYTES) gives the total size of the SORT prefilter related data structures associated with this database.

Note: If predicate indexes also exist for the table this value also includes the space used by the predicate indexes (see Partition Indexes).

The PSI Prefilter - Usage Details

Like the SORT prefilter, a PSI prefilter uses a collection of sorted indexes. The terminology, defaults, and restrictions of the SORT prefilter apply to the PSI prefilter.

Creating a PSI prefilter is the same as creating a SORT prefilter: pass an `LPAR_LST_ENCODINGS` in the dbpars parameter of the lkt_dbload function. Each element in the `LPAR_LST_ENCODINGS` lpar must be an `LPAR_LST_ENCODING` lpar. Within each `LPAR_LST_ENCODING`, PSI allows five lpars (SORT only allows two).

- The first is `LPAR_STRARR_FIELDNAMES`. Each string array is a list of field names which is used for a given encoding. The first field name is the primary sort key, the second field is secondary, and so on.
- The second is `LPAR_INTARR_BKWDSFLDS`. Fields can also be sorted backwards (from the last characters in the field to the first). Important fields can and must be used as primary sort keys in both their forward and reverse directions. The backward fields array, if given, must be the same length as the field names array. The integer is set to one if the field in the

corresponding position in the field names array is to be sorted in the backward direction and zero for the forward direction. If this value is not given, all fields are sorted in the forward direction.

These first two parameters, LPAR_STRARR_FIELDNAMES and LPAR_INTARR_BKWDSFLDS, operate exactly as in SORT.

- The third is LPAR_INT_ENCODING_SFX_CNT, which holds the number of fields to which suffixing is applied. This is optional. If present, it must be 0 or 1. Default is 1. Passing 0 turns off suffixing for the encoding, making it behave much like a SORT encoding.
- The fourth is LPAR_INTARR_PSIMINMATCHSIZES. This is also optional. If present, it must be the same length as LPAR_STRARR_FIELDNAMES. It is used for improving search speed. Larger values improve speed, but might degrade accuracy. Contact your TIBCO representative for assistance in balancing this parameter. Each entry controls the minimum amount of data that must match within a specific field. If no field meets the minimum, the prefilter rejects the record and prunes the scan of the encoding.
- The fifth is LPAR_INT_PSI_DENSITY. This is optional. The allowable values are 0 (high density), 1 (standard density – the default), and 2 (low density). High density might improve accuracy, but increase memory usage and might lower throughput. Low density lowers memory usage and might increase throughput, but might lower accuracy.

Like the SORT prefilter, if no encodings are given when a PSI enabled table is created, a default set of encodings is automatically generated. This default set must provide good accuracy for record-matching operations but might take more memory than a carefully crafted set of encodings.

Contact your TIBCO representative for further information on which encodings to use for your specific application.

To use PSI prefilter in a search, the table must be created with PSI enabled.

Note: Unlike SORT, LPAR_LST_DEDUPQUERY in the dbpars is not supported by PSI.

Like SORT, the PSI prefilter accepts a set of field values corresponding to the fields of the table. The LPAR_BLKARR_PSILOOKUPFIELD parameter specifies these field values explicitly. The length of the array must be the same as the number of fields in the table, and the order of the values must correspond to the field order when the table was defined.

If the look-up fields are not specified, the TIBCO Patterns server attempts to infer them from the query given. If it cannot do so, an error is returned and no query is performed.

If the PSI prefilter is enabled for a table, the server uses PSI with queries on that table unless LPAR_BOOL_PSISEARCH parameter is sent with value false.

Whenever a lkt_dbsearch is done, the same Boolean parameter, `LPAR_BOOL_PSISEARCH`, is returned in the stats list. This parameter is true if the PSI prefilter was employed, false otherwise.

Whenever database statistics are returned for a PSI prefilter enabled database, the value of `LPAR_INT_DBIDXTOTALKBYTES` gives the total size of the PSI prefilter related data structures associated with this database. The value of LPAR_INTARR_PSI_DENSITIES lists the suffix densities of the PSI encodings.

Note: If predicate indexes also exist for the table, this value also includes the space used by the predicate indexes. See Partition Indexes for more details.

Variable Attributes - Usage Details

The Variable Attributes feature provides the ability to associate with a record a variable number of labeled pieces of data and then search for records based on these labeled values. These labeled pieces of information are called Attributes of the record. The feature is called Variable Attributes because any particular record is allowed to have anywhere from zero to 1,000 Attributes associated with it. Each record is allowed to have a completely different set of Attributes. Labels are added dynamically as they are received, there is no need to predefine them, thus it is not necessary to know the entire set of possible Attributes when a table is created.

While searching, an Attribute value can be specified and used anywhere a standard field value is used. This includes all the query types and predicate expressions. However, note that an Attribute value is always considered to be of type `LKT_FLD_TYPE_SRCHTEXT` and can be used only where a text field is used. For instance, it cannot be the target of a custom date search.

An example used for the Variable Attributes feature might be a products table. There can be a very large number of different characteristics for a product (e.g. height, weight, color, power requirements, operating temperature) the particular set of characteristics being dependent on the type of product. To provide a separate column in a table for each possible characteristic is impractical from several points of view: the number of fields needed could easily be in the thousands or even tens of thousands, too large to be practical, the set of characteristics is likely to change frequently as new products are introduced, adding and removing columns in a large table on a frequent basis is not practical. With the Variable Attributes feature handling this kind of situation becomes straightforward.

All Variable Attribute values are kept in a special field of the table that is of type `LKT_FLD_TYPE_ATTRIBUTES`. A table might have only one field of type `LKT_FLD_TYPE_ATTRIBUTES` and that field must be the last field of the table. The presence of such a field in a table enables Variable Attributes to be attached to the records of the table. This field is referred to as the Variable

Attributes field. If a table does not have a Variable Attributes field, variable attributes cannot be loaded into records of that table.

A record might never have two attribute values with the same name.

The ordering of attributes is maintained. Attributes are always returned in the same order they were added.

Attributes are referenced by using a "field name" with the format:

```
"Variable-Attribute-Field:attribute-name"
```

where Variable-Attribute-Field is the name assigned to the Variable Attribute Field for the table, a colon is a literal colon character and attribute-name is the name or label for the attribute being referenced. It is an error if Variable-Attribute-Field does not exist in the table or is not a field of type `LKT_FLD_TYPE_ATTRIBUTES`.

The above convention implies that the name of the Variable Attributes Field might not contain the colon character. Other sizes and limits to keep in mind:

- There is a limit of 1,000 Attributes per record.
- Variable Attribute data must be UTF-8 encoded characters.
- The total Attribute data for a particular record is limited to 50,000 bytes.
- There might be at most 60,000 different Attribute names in one table.
- The variable attribute name is case sensitive and allows letters, digits, the underscore, and the hyphen character, other characters are not supported.
- An Attribute name is a null-terminated string of from 1 to 2048 characters.

When referencing a Variable Attribute in a query or predicate, if the named attribute does not exist in the record it is treated exactly as if the record had a zero length (empty) value for that Attribute. No distinction is made between an Attribute with a zero length value and an Attribute that doesn't exist in the record.

If you reference a Variable Attribute field without an Attribute name qualifier (i.e. reference it as a regular searchable text field) the value of the field is the values of each of the Attributes in the field concatenated together with a single space character between each value. The order of the values is maintained as originally given. Thus if you had a Variable Attribute set:

```
street="123 Albany St"
city="Boston"
state="MA"
zip="02201"
```

the value of the Variable Attribute field as a whole (with no Attribute name qualifier) would be "123 Albany St Boston MA 02201". Thus it is possible to do a simple query against all attribute values by querying the field as if it were a normal searchable text field.

In addition to the previously described means of referencing Variable Attributes in a table, a new query structure also provides a convenient means of performing a query across a set of attributes. This query structure is described in Query Construction along with the other query constructs.

Adding and Accessing Variable Attributes in a Record

As we saw in Building the Input: Data Records all of the field data for a record is loaded as a single block of text. Thus all fields, regardless of field type, must have a text representation. This is true of the Variable Attributes Field just as it is true for integer, float, date and date-time fields. A set of Attributes is encoded as a text string using the following syntax:

Attribute-Name **:** *Attribute-Value* [**;** *Attribute-Name* **:** *Attribute-Value*] *

I.e. A list of Attribute-Name - Attribute-Value pairs, where the name and value are separated by a colon character and pairs in the list are separated by semicolon characters. If an Attribute-Value contains a semicolon character it must be escaped by giving it twice. An example:

For address: "123; Suite 107 Main St" Split into street-name and street-number.

```
    "street-number:123;; Suite 107;street-name:Main St"
```

A zero length string is the string encoded form of an empty Attributes set.

There is no need to worry too much about the details of this format because a set of functions is provided to encode and decode a set of attribute values to and from this format. These functions work with an opaque object that represents a set of Attributes. These functions provide a means of creating one of these objects from either a list of attribute names and values, or an encoded text form of an attribute set. You can then retrieve attribute values in a number of ways and generate the encoded text string for the attribute set. The opaque object representing an Attribute set is declared as type:

```
        dvk_va_t
```

The special constant `dvk_va_null` represents a non-existent attribute set (as opposed to an empty attribute set). This value might be assigned to a variable of type `dvk_va_t` to indicate that it is not currently set. For example,

```
        dvk_va_t our_va_set = dvk_va_null ;
```

The functions available are:

dvk_varattr_create

```
dvk_va_t
dvk_varattr_create(int num_attrs, const char **attr_names,
                   const char **attr_values, int *val_lens)
```

This function creates a Variable Attributes object from a list of Attribute names and values. It returns a new Variable Attributes object. It returns `dvk_va_null` if invalid arguments are given. The returned item must be cleaned up by the caller using the `dvk_varattr_destroy` function.

The arguments are:

- **num_attrs** The count of the number of attributes. All of the following arrays must be of this length.
- **attr_names** The list of attribute names. Each name is a null-terminated string. Names are subject to the restrictions for attribute names given above. All names in the list must be unique.
- **attr_values** The list of attribute values. Each value is a text string whose length in bytes, is given by the corresponding entry in `val_lens`. Values must be UTF-8 encoded character strings, but need not be null-terminated strings.
- **val_lens** This is a list of integer values that give the length of the Attribute values in bytes.

dvk_varattr_decode

```
dvk_va_t
dvk_varattr_decode(const char *enc_str, int enc_len)
```

This function creates a new Variable Attributes set object from the encoded text string format for an Attribute set. It returns a new Attributes set object if successful. It returns `dvk_va_null` if the given string is not a properly encoded Attribute set. The returned item must be cleaned up by the caller using the `dvk_varattr_destroy` function.

The arguments are:

- **enc_str** This is the encoded text string version of the attribute set. This might not be NULL. To represent an empty set, use a zero length string.
- **enc_len** This is the length in bytes of the encoded text string.

dvk_varattr_destroy

```
void
dvk_varattr_destroy(dvk_va_t va)
```

This function destroys a Variable Attributes set object. If you do not call this function for each dvk_va_t object you create, a major memory leak occurs. Note that this also destroys all data that might have been returned from this Attribute set, so all such data is invalid after this call is made.

The argument is:

va The Attribute set to be destroyed.

dvk_varattr_num_attrs

```
int
dvk_varattr_num_attrs(dvk_va_t va)
```

This function returns the number of attributes in an Attribute set. It returns a value less than zero if the given set is not valid.

The argument is:

va The Attribute set.

dvk_varattr_get_names

```
const char **
dvk_varattr_get_names(dvk_va_t va, int *num_attrs)
```

This function returns a list of the names of all of the Attributes in the given Variable Attributes set. The names are null-terminated strings. The names appear in the order they were originally given when the set was created. If `va` is invalid a NULL pointer is returned.

Note that the list returned and the names in the list become invalid when the given Variable Attribute set is destroyed. If they are to be used after the set is destroyed the caller must make a copy of the list and all values in the list.

The arguments are:

- **va** The Attribute set.
- **num_attrs** If this is a non-null pointer the number of items in the returned list is placed in the integer pointed to by this value.

dvk_varattr_get_values

```
const char **
dvk_varattr_get_values(dvk_va_t va, int *num_attrs)
```

This function returns a list of the values of all of the Attributes in the given Variable Attributes set. The values are not null-terminated strings. To get the lengths of the values `dvk_varattr_get_lengths` must be used. The values appear in the order they were originally given when the set was created. This is the same order names are returned in by the `dvk_varattr_get_names` function, so names and values can be matched up by index position. If `va` is invalid a NULL pointer is returned.

Note that the list returned and the values in the list become invalid when the given Variable Attribute set is destroyed. If they are to be used after the set is destroyed the caller must make a copy of the list and all values in the list.

The arguments are:

- **va** The Attribute set.
- **num_attrs** If this is a non-null pointer the number of items in the returned list is placed in the integer pointed to by this value.

dvk_varattr_get_lengths

```
const int *
dvk_varattr_get_lengths(dvk_va_t va, int *num_attrs)
```

This function returns a list of the lengths of the values of all of the Attributes in the given Variable Attributes set. The values appear in the order they were originally given when the set was created. This is the same order names are returned in by the `dvk_varattr_get_names` function, so names, values and value lengths can be matched up by index position. If `va` is invalid a NULL pointer is returned.

Note that the list returned becomes invalid when the given Variable Attribute set is destroyed. If the list is to be used after the set is destroyed the caller must make a copy of the list. The caller must not attempt to update any value in the list returned.

The arguments are:

- **va** The Attribute set.
- **num_attrs** If this is a non-null pointer the number of items in the returned list is placed in the integer pointed to by this value.

dvk_varattr_get_attr

```
const char *
dvk_varattr_get_attr(dvk_va_t va, const char *attr_name,
                     int *val_len)
```

This function returns the value of an Attribute by Attribute name. Given an Attribute set and an Attribute name a pointer to the value of that attribute is returned. If the set has no Attribute with the given value a pointer to a zero length string is returned. If the Attribute set is invalid or null, a null pointer is returned. The value returned is a static value that must not be updated. It is valid only until the Attribute set is destroyed.

Note that with this function you can't distinguish between the case of an Attribute that is in the set and has a zero length value and an attribute that is not in the set at all. If you wish to distinguish these cases, you should retrieve the list of Attribute names, values and lengths. An Attribute that is not in the set of course is not returned in the list, but an Attribute with a zero length value is in the list, with the corresponding value being of length zero.

The arguments are:

- **va:** The Attribute set.
- **attr_name**: The null-terminated Attribute name. It must not be NULL.
- **val_len:** If this is non-null the length of the attribute value is placed in the integer pointed to by this value.

dvk_varattr_encode

```
const char *
dvk_varattr_encode(dvk_va_t va, int *enc_len)
```

This function is used to translate an Attribute set into the text string encoded form for the set. This can be used to encode an attribute set into a record to be sent to the TIBCO Patterns server. If the Attribute set is invalid a NULL pointer is returned. The value returned is a static value that must not be updated. It is valid only until the Attribute set is destroyed.

The arguments are:

- **va** The Attribute set.
- **enc_len** If this is non-null the length of the encoded string in bytes is placed in the integer pointed to by this value.

Let's look back at the example of building records in Building the Input: Data Records, only this time let's suppose the input record has four fields: "street", "city", "state", and "zip" instead of a single "addr" field, and the table has an attributes field instead of a text address field that we wish to encode these values into. This could be done as shown below.

```
typedef unsigned char UCHAR;
/* ... */
{
    int       srchtxtlen;
    int       fldlens[2];
    int       srchtxt[1024];
    UCHAR     *keystr;
    static const char *addr_names[4] = { "street","city","state","zip"};
    const char *addr_values[4];
    int       addr_lens[4];
    const char *enc_str ;
dvk_va_t va ;
    lpar_t    reclst;
    lpar_t    record;
    reclst = lpar_create_lst(LPAR_LST_GENERIC);
    for ( /* all records */ ) {
        /* ... */
        /* set standard field */*/
        fldlens[0] = strlen(cur_record.name);
        strcpy(srchtxt, cur_record.name);
        /* create attribute set for record attributes */
        addr_values[0] = cur_record.street;
        addr_lens[0]   = strlen(addr_values[0]);
        addr_values[1] = cur_record.city;
        addr_lens[1]   = strlen(addr_values[1]);
        addr_values[2] = cur_record.state;
        addr_lens[2]   = strlen(addr_values[2]);
        addr_values[3] = cur_record.zip;
        addr_lens[3]   = strlen(addr_values[3]);
        va = dvk_varattr_create(4, addr_names, addr_values, addr_lens);
        if (va == dvk_va_null) {
                /* handle error here */
        }
        /* get the encoded text string for it. */
        enc_str = dvk_varattr_encode(va, &(fldlens[1]));
        if (enc_str == NULL) {
                /* handle error here */
        }
        /* append it to search text, note it is not null terminated string */
        memcpy(srchtext+fldlens[0], enc_str, fldlens[1]);
        /* we've copied what we need - clean up the attribute set */
dvk_varattr_destroy(va);
        va = dvk_va_null;
        /* now create record as before */
        srchtxtlen = fldlens[0] + fldlens[1];
        record = lpar_create_record();
lpar_set_record_key(record, keystr);
lpar_set_record_srchtxt(record, srchtxt, srchtxtlen);
lpar_set_record_srchflds(record, fldlens, 2);
        lpar_append_lst(reclst, record);
    }
    /* ... Pass list of records to a command function ... */
    lpar_destroy(reclst);
}
```

For fetching attribute values from a returned record let's look at the example from Processing the Output of DevKit Commands again. This time assuming the record has the above structure and we wish to print out each attribute value in the returned record.

```
/* ... */
{
    int          i, item, numitems, recnum, numrecs, arrlen;
    int          namelen;
    lpar_t       query, lpar;
    lpar_t       names, srchpars, stats, matches, minfo;
    dvkerr_t     dvkerr;
    const int    *pvals, *dvals;
    const double *vvals;
    const UCHAR  *name;
    const lpar_t *records;
    const lpar_t *recinfo;
    int          numattrs;
    const char   **attrnames = {"street","city","state","zip",NULL};
    const char   *attrval;
    int          attrlen;
    int          attridx;
    const UCHAR  *enc_str;
    int          enc_len;
    dvk_va_t     va;
    /* ... Build names list, query, and search parameter list ... */
    /* .... as before .... */
    records = lpar_get_lst_item_array(matches, &numrecs);
    for (recnum=0 ; recnum < numrecs ; recnum++) {
        printf("[%2d] ", recnum+1);
        name = lpar_get_record_field(records[recnum], 0, &namelen);
        printf("Name field `%.*s'\n", namelen, (char *)name);
        /* pull the Attributes encoded string and decode it */
        enc_str = lpar_get_record_field(records[recnum],1,&enc_len);
        va = dvk_varattr_decode(enc_str, enc_len);
        if (va == (dvk_va_t *)0) {
            /* handle error here */
            printf("Invalid Attributes\n");
        } else {
            /* successfully decoded, print our name-values */
            for (attridx = 0; attrnames[attridx] != NULL; attridx++) {
                attrval = dvk_varattr_get_attr(va,
                                                 attrnames[attridx],
                                                 &attrlen);
                if ((attrval == NULL)||(attrlen < 0)) {
                    /* handle error here */
                    printf("Error retrieving value for: %s.",
                           attrnames[attridx]);
                } else {
                    /* Output the value. */
                    printf("%s `%.*s'\n", attrnames[attridx],
                                      attrlen,
                                      attrval) ;
                }
            }
            /* clean up, done with values now */
            dvk_varattr_destroy(va);
            va = dvk_va_null;
        }
        /* the rest is as before.... */
    }
    lpar_destroy(names);
    lpar_destroy(query);
    lpar_destroy(srchpars);
    lpar_destroy(stats);
    lpar_destroy(matches);
    lpar_destroy(minfo);
}
```

In the example above we have assumed we know the names of all of the Attributes of interest and pulled them out by name. If we wanted to retrieve all attributes in the record the above could be modified as follows:

```
/* ... */
{
    int      i, item, numitems, recnum, numrecs, arrlen;
    int      namelen;
    lpar_t   query, lpar;
    lpar_t   names, srchpars, stats, matches, minfo;
    dvkerr_t dvkerr;
    const int    *pvals, *dvals;
    const double *vvals;
    const UCHAR  *name;
    const lpar_t *records;
    const lpar_t *recinfo;
    int numattrs;
    const char  **attrnames;
    const char  **attrvals;
    const int   *attrlens;
    int         attridx;
    const UCHAR *enc_str;
    int         enc_len;
    dvk_va_t    va;
    /* ... Build names list, query, and search parameter list ... */
    /* .... as before .... */
    records = lpar_get_lst_item_array(matches, &numrecs);
    for (recnum=0 ; recnum < numrecs ; recnum++) {
        printf("[%2d] ", recnum+1);
        name = lpar_get_record_field(records[recnum], 0, &namelen);
        printf("Name field `%.*s'\n", namelen, (char *)name);
        /* pull the Attributes encoded string and decode it */
        enc_str = lpar_get_record_field(records[recnum],1,&enc_len);
        va = dvk_varattr_decode(enc_str, enc_len);
        if (va == (dvk_va_t *)0) {
            /* handle error here */
            printf("Invalid Attributes\n");
        } else {
            /* successfully decoded, print all name-values */
            attrnames = dvk_varattr_get_names(va, &numattrs);
            attrvals = dvk_varattr_get_values(va, &numattrs);
            attrlens = dvk_varattr_get_lengths(va, &numattrs);
            if ((attrnames == NULL)
            ||  (attrvals == NULL)
            ||  (attrlens == NULL)) {
                /* handle error here */
                printf("Could not retrieve attribute values.\n");
            } else {
                for (attridx = 0; attridx < numattrs; attridx++) {
                    printf("%s `%.*s'\n", attrnames[attridx],
                                           attrlens[attridx],
                                           attrvals[attridx]);
                }
            }
            /* clean up, done with values now */
            dvk_varattr_destroy(va);
            va = dvk_va_null;
        }
        /* the rest is as before.... */
    }
    lpar_destroy(names);
    lpar_destroy(query);
    lpar_destroy(srchpars);
    lpar_destroy(stats);
    lpar_destroy(matches);
    lpar_destroy(minfo);
}
```

Federated Tables

When working with federated tables through a gateway there are a number of additional error codes that might be returned. These codes might be returned by almost any command, so they are described once here, and not mentioned further in the individual command descriptions. All of these codes start with the prefix: SVR_ERR_ which is omitted from the listing below.

- NODE_CONFLICT: Two or more nodes reported two or more different errors. If all nodes report the same error that error is returned. If two or more different errors are reported then this error code is generated and the individual errors are returned as a list in the error item.
- NODE_DOWN: One or more nodes could not be reached. The TIBCO Patterns server process might be down, the host machine might be down, or communications might be down or blocked.
- NODE_DATA_SYNC: This object is not properly synchronized on the nodes. This usually means the referenced object (table, thesaurus or character map) exists on some nodes, but not others.
- OTHER_NODE: Operation was aborted due to an error on another node. This is an internal error code that should never be returned by the gateway.

Federated Tables and Gateways

TIBCO Patterns directly supports federated tables through a gateway server. Although this feature has little impact on the API described in this document some understanding of federated tables and gateway servers are useful. A brief description of the important points is given here.

For a full description see the TIBCO® Patterns Installation guide and the TIBCO® Patterns Concepts.

A federated table is a table that is physically split into multiple tables residing on separate TIBCO Patterns servers (usually on separate machines). This splitting is intended for those cases where a table is simply too large to fit on a single machine. It might also be done to reduce latency times when performing a search. The records of the logical table are divided among the physical tables that comprise the logical table.

A *gateway* is a special TIBCO Patterns server process that manages one or more federated tables. It maps operations on the logical table into operations on the individual physical tables in the respective TIBCO Patterns server processes.

More specifically a gateway manages a cluster of TIBCO Patterns server processes, each such process being referred to as a *node*.

All operations performed through a gateway on federated tables are done in an identical manner to operations on a standard table on a standard TIBCO Patterns server process.

Gateways also manage thesauri, character maps, and Learn Models.

Communicating with TIBCO Patterns Servers

In all of the examples so far the tables have been assumed to be local, that is within the same process. However, you are probably communicating with a TIBCO Patterns server process, in which case the host must be identified. This is done using the `host` parameter. The `host` parameter is always the first parameter of any function that must communicate with a TIBCO Patterns server. A host parameter of LPAR_NULL indicates the local process. To construct a host parameter for a remote TIBCO Patterns server, use:

```
lpar_t lkt_host_lpar( const char *address, int port );
```

The input address identifies the machine on which the server is running. It can be a network (IP) address, or a network (DNS) name.

The port identifies the port the server is listening on.

The output returns an lpar that holds the host information. It can be used in any lkt function that requires a host parameter.

Field Names – Referencing a Field in a Table

The field names used in a query or predicate expression have the following form:

`[table-name.]field-name[:attribute-name]`

An optional table name followed by a period, a field name and an optional variable attribute name preceded by a colon.

Never enter the period without a table name, or a colon without an attribute name.

- **Table-name**: is used only in join searches. It is required only if the *field-name* is not unique in the set of tables included in the join.
- **Field-name**: is always required. It identifies the field in the table.
- **Attribute-name**: is allowed only if the field is a variable attributes field. It specifies which attribute value in the field is desired. If *field-name* specifies a variable attributes field, and an `attribute-name` is not specified the reference is to the concatenation of all attribute values in the field.

Restrictions on Names

The names of in-memory objects (tables, thesauri, character maps, and Learn Models) must not contain any of the following characters:

- at sign (@)
- backslash (\)
- forward slash (/)
- colon (:)
- newline character

In addition, the name of a table must not contain a period (.).

The field names must not contain a period (.), colon (:) or new line character.

When using Checkpoint / Restore, the length of an object name is restricted to the maximum file name length minus 6 bytes. On most modern file systems, the maximum file name length is 255 bytes, so the maximum object name length would be 249 bytes.

Essential DevKit Commands

This section explains the basic DevKit command functions that perform the following tasks: system initialization and cleanup, creating and destroying DevKit tables and checking their status, and TIBCO Patterns matching and machine learning scoring operations.

For ease of reference, we follow a common format in our descriptions of DevKit commands.

Each description consists of the following elements:

- The command function prototype
- A brief description of what the command does (usually one or two sentences in length)
- A detailed description of the command's input parameters
- A detailed description of the command's output parameters
- An alphabetical table of error codes and error items returnable by the command.

Note: In these tables, the error code prefix DVK_ERR_ is omitted, but should be understood. If a command is operating on a remote database, it might also return server related errors (prefixed with SVR_ERR_). In general, these errors are not listed in the error tables following each command description. Server related errors never contain an attached object. For the list of error codes and their explanatory strings, see Table of DevKit Error Codes and Strings.

Each input and output parameter is classified as either required or optional. A null value (LPAR_NULL) might be specified for any optional parameter, but is illegal for any required parameter. When a parameter is a simple lpar or a list lpar, our documentation includes explanations of all lpar id's that might be supplied as input or returned as output.

Remember that the error item (if any) returned along with an error code is a copy of an offending input item, and must be freed with a call to `dvkerr_clear` (see An Important Word (and a Warning) About Error Conditions).

DevKit System Initialization (lkt_devkit_init)

```
dvkerr_t  lkt_devkit_init( void );
```

Before calling any other functions, all DevKit applications must:

- Make any desired changes to DevKit parameters, by calling `lkt_devkit_get_parameters` and `lkt_devkit_set_parameters`.
- Call `lkt_devkit_init`.

This applies to the input-building functions discussed in Input and Output Parameter Types as well as other command functions.

```
dvk_params_t lkt_get_devkit_parameters( void )
```

Obtains the current DevKit global settings.

Output

A structure containing the current devkit settings.

Note: The returned `trusted_store` pointer can be changed, but the memory pointed to by the returned `trusted_store` member should not be modified.

```
int lkt_set_devkit_parameters( const dvk_params_t* params )
```

Sets the DevKit global parameters.

This function must only be called before `lkt_devkit_init`. Calling it after `lkt_devkit_init` might produce unpredictable results.

Input

The new global parameters for DevKit.

Output

0: Success

Non-zero: error (invalid parameter).

```
dvkerr_t lkt_devkit_init( void );
```

Initializes DevKit.

Error codes

This command never returns an error. If it cannot initialize devkit it terminates the process.

dvk_params_t

This structure holds settings that affect DevKit behavior globally. It contains the following members:

- int connect_protocols

  Controls which protocols DevKit uses to connect to servers. It must be one of

DVK_PROTO_DEFAULT: Use the default setting

DVK_PROTO_IPV4: Use only IPv4

DVK_PROTO_IPV6: Use only IPv6

DVK_PROTO_MIXED: Use either. Actual usage is determined by the system network configuration.

Default: DVK_PROTO_DEFAULT

- int persist_connections

  Controls whether outgoing connections are persistent.

  0: outgoing connections are not persistent

  non-0: outgoing connections are persistent

  Default: 1.

- int ssl_enabled

  Controls if SSL/TLS is used to encrypt data on outgoing connections.

  0: data on outgoing connections is plaintext.

  non-0: data on outgoing connections is encrypted.

  Default: 0

- const char *trusted_store

  Specifies the store of trusted certificates for SSL connections.

  For details, see the "--trusted-store command line option" in the TIBCO® Patterns *Installation Guide.*

Loading a Database (lkt_dbload)

```
This command is the only DevKit command that creates a new database.
dvkerr_t lkt_dbload( lpar_t host, lpar_t name, lpar_t dbpars, lpar_t reclist,
 lpar_t *dbstats );
dvkerr_t lkt_dbloadT(lpar_t host, lpar_t name, lpar_t tran, lpar_t dbpars,
 lpar_t reclist, lpar_t *dbstats );
```

Note: The `lkt_restore` command can also create a database, but as it is a database that must have been, at some point, created with `lkt_dbload` it is not strictly speaking a new database.

In the normal case a database is created and loaded with an initial set of records (hence the command name, `lkt_dbload`). Records might then be added, deleted, or replaced in real-time using commands described later on in Updating Loaded Databases. Applications that do not require continual data updates might find it simpler to replace an entire database with a new version. We'll describe the best way to do this when we discuss the `lkt_dbmove` command (Renaming a Database (lkt_dbmove)).

Starting with release 4.4.1 loading is optimized based on the number of records to be loaded.

Passing the LPAR_INT_DBNUMRECORDS value in the dbpars list informs the TIBCO Patterns servers of the approximate number of records to be loaded so that it can choose the best loading strategy. The loading takes place even if this value is not given or is incorrect, the only consequence is that loading performance might not be optimal.

A batch of over 40,000 records is needed to achieve the best load performance. With batches of this size or larger the TIBCO Patterns servers employs multiple processors to speed up the loading.

Note: If you are running the TIBCO Patterns servers on an older single processor system then the multi-threaded fast load only slows down the process and should be suppressed by passing a LPAR_INT_DBNUMRECORDS value of 0.

For smaller numbers of records, a low overhead single stream load is performed. The best strategy for loading depends on your hardware and the number and nature of the tables to be loaded. If optimal load times are critical consult your TIBCO representative for the best means of optimizing loads for your particular case.

Note that if a multi-threaded fast load is performed the memory cap is checked only once, before the load is started. Thus the memory cap can be greatly exceeded. Therefore, if you are using memory caps it is not recommended that you use fast loading.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

name (required)

This is a null-terminated character string that is unique to the table. Every database must have a non-zero length.

dbpars (optional)

is a list of lpars specifying parameters that in some way govern or affect the database over its entire lifetime. Every one of these parameters has a default value, so it is not strictly necessary to pass any dbpars list to `lkt_dbload`. If the defaults are satisfactory, simply pass a value of LPAR_NULL instead.

Here is the list of database parameters and their default values:

- LPAR_BOOL_DBGIPFILTER is a Boolean parameter specifying whether or not to enable the GIP prefilter. See The GIP Prefilter - Usage Details for a full discussion.

  Default value: true (if GIP prefilter is the default prefilter. It is False if it is not.)

- LPAR_BOOL_DBSORTFILTER is a Boolean parameter specifying whether or not to enable the SORT prefilter. See The SORT Prefilter - Usage Details for a full discussion.

  Default value: true (if SORT prefilter is the default prefilter. It is False if it is not.)

- LPAR_BOOL_DBPSIFILTER is a Boolean parameter specifying whether or not to enable the PSI prefilter. See The PSI Prefilter - Usage Details for a full discussion.

  Default value: true (if PSI prefilter is the default prefilter. It is False if it is not.)

- LPAR_LST_ENCODINGS is only used if the SORT or PSI prefilter is enabled. See The SORT Prefilter - Usage Details and The PSI Prefilter - Usage Details for a full discussion of this parameter.

  Default value: Every searchable field is indexed forward and backward.

- LPAR_INT_PSI_DENSITY overrides the default encoding density for PSI-indexed tables. See The PSI Prefilter - Usage Details for a full discussion of this parameter.

- LPAR_INT_DBNUMFIELDS specifies the number of fields that each record in the database has. If specified, every record that is ever loaded into this database must have exactly this number of fields (see Building the Input: Data Records for how this field structure is specified in terms of field lengths).

  Default value: If the number of fields is not specified here it is taken from the number of fields in the LPAR_STRARR_FIELDNAMES parameter.

  If that is not given it is taken from the number of fields in the LPAR_INTARR_FIELDTYPES parameter. If neither is given it is taken from the number of fields in the record source. It is an error if the number of fields is defined in none of these places.

- LPAR_STRARR_FIELDNAMES is an array of strings containing the names of each field. The number of field names must be equal to the number of fields. Field names are required, however if records are being loaded from a CSV file the names might be defined in the CSV file. If the names are defined both here and in the CSV file the names must be identical.

  Default value: If data is not being loaded from a CSV file with LPAR_BOOL_FNAMESFIRST set then this is required. Otherwise they default to the names as defined in the CSV file.

Note: See `Reading a Database From a File (lkt_dbread)Reading a Database From a File (lkt_dbread)`Reading a Database From a File (lkt_dbread) function for a description of this parameter.

- LPAR_INTARR_FIELDTYPES is an array of integers containing the field type of each field. The length of the field type array must be equal to the number of fields. For a description of field types and a list of valid field types see The TIBCO Patterns Table.

  Default value: All fields are of type LKT_FLD_TYPE_SRCHTEXT by default.

- LPAR_BLK_DBINFO is simply a block of information that you can attach to a database. It can contain any text data you wish. Its contents are returned any time database statistics are returned (see the following Output).

  Default value: Databases have no database info unless you supply it.

- LPAR_STRARR_CHARMAPS is an array of names of character maps. The length of the array must be the same as the number of fields in the database. Even though character maps apply only to fields of type LKT_FLD_TYPE_SRCHTEXT or LKT_FLD_TYPE_ATTRIBUTES a valid character map name must be supplied for every field. For a description of character maps see Character Mapping.

  Default value: The default character map DVK_CMAP_STDNAME is assigned to every field.

- LPAR_LST_PIDXDEF defines a partition index for this table. Any number of these values might be given. For a description of this option and partition indexes see Partition Indexes.

  Default value: No indexes are created.

- LPAR_INT_DBNUMRECORDS gives an estimate of the number of records in the record list. It is not necessary that this value be exact or even close.

  This is used by the TIBCO Patterns servers to determine the most efficient means of loading the records.

  Default value: 0 (It assumes a small batch of records). The exception is if loading using an LPAR_LST_REMOTEFILE file specification the assumption is that it is a large batch of records.

- LPAR_BOOL_DOMAXWORK applies when loading records. If this flag is true invalid records are quietly ignored and loading of other records continues. If this flag is not given or is false an invalid record causes the entire table create and load operation to fail.

  Some conditions that are considered to be an "invalid record" include: invalid UTF-8 data, duplicate record key, wrong number of fields, a completely empty record, an empty key value (if keys are not being generated automatically). It is not considered invalid for a record to have invalid or empty data values in non-text fields. These are simply stored as having the special empty or invalid value.

  Default value: false.
- LPAR_STR_PARENT_TBL must name an existing parent table. This table is created as a child table of the named parent table. If not given this is not a child table. A child table cannot also be a parent table, therefore it is not valid to give this argument and the LPAR_BOOL_PARENT_TBL argument with a value of true.
- LPAR_BOOL_PARENT_TBL indicates whether this table is a parent table. If given and set to true, this table is created as a table that can act as a parent in a join relationship. If false, or not given this table is a standard or child table and cannot act as a parent table in a joint relationship. A parent table cannot also be a child table.
- LPAR_BOOL_DBGIPGPU is a Boolean parameter specifying whether GPU acceleration is to be employed when searching this table. If GPU is not enabled on this server a value of true causes a FEATURESET error. This argument is ignored if this table is not GIP indexed. Set this value to true to enable GPU acceleration for this table.
- LPAR_INTARR_GPU_DEVICE_IDS is used to specify which GPU devices is to be used when employing the GPU accelerator. A PARAMVAL error is returned if a given ID is not a valid GPU device ID. A GPU_DEVICE_NOT_ALLOWED error is returned if a given ID is a valid GPU device, but the device can’t be used. If GPU acceleration is not enabled and this is given, a FEATURESET error is returned. If this argument is not given, the default GPU devices are used.

If you are using the SORT or PSI prefilter it might be necessary to set additional database parameters (see the sections The SORT Prefilter - Usage Details andThe PSI Prefilter - Usage Details).

reclist (optional)

must be a list of records or a file specification. If a record list is given every record must contain a unique key.

File specifications take the form of an LPAR_LST_LOCALFILE or an LPAR_LST_REMOTEFILE. This list type lpar must contain either an LPAR_STR_CSVFILE or LPAR_STR_FWFFILE as well as any options appropriate for the `lkt_dbread` command. If no host is specified (so that the command runs in the local process), there is no distinction made between the LPAR_LST_LOCALFILE and

LPAR_LST_REMOTEFILE forms. If a host is specified, an LPAR_LST_LOCALFILE indicates a file that is read by the client, the contents of which are sent to the server. With an LPAR_LST_ REMOTEFILE, only the filename and dbread options are sent to the server. The file must exist within the server's loadable-data directory. Its contents are read directly by the server.

It is permitted to create a database with zero records by passing `lkt_dbload` either LPAR_NULL or a zero-item list as the value of reclist.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the table load operation is to be performed.

Output

dbstats (optional)

is a list of lpars giving some statistics on the newly-created database. These include the effective values of the following database parameters described previously:

- LPAR_STR_DBDESCRIPTOR
- LPAR_BLK_DBINFO
- LPAR_INT_DBNUMFIELDS
- LPAR_STRARR_FIELDNAMES
- LPAR_INTARR_FIELDTYPES
- LPAR_STRARR_CHARMAPS
- LPAR_BOOL_DBGIPFILTER
- LPAR_BOOL_DBSORTFILTER
- LPAR_BOOL_DBPSIFILTER
- LPAP_BOOL_GIPGPU
- LPAR_INTARR_GPU_DEVICE_IDS (returned only if the default set is not used)
- LPAR_STR_PARENT_TBL (returned only for child tables)

as well as the following:

- LPAR_STR_CHECKPOINT_STATUS describes the data-persistence status for this database. For checkpoint/restore it is one of: "Last Checkpoint: yyyy/mm/dd-hh:mm:ss", "Never

Checkpointed", "Checkpoints Disabled", "No Checkpoint Info", "Gateway generated checkpoint id: yyyy/mm/dd-hh:mm:ss". For durable-data, it is "Automatic".

- LPAR_LONGINTARR_FIELDBYTECOUNTS the total number of bytes of data stored in the table broken out by field.
- LPAR_LONGINTARR_FIELDCHARCOUNTS the total number of characters of data stored in the table broken out by field after character mapping is applied.
- LPAR_INT_DBNUMRECORDS is the number of records in the database.
- LPAR_INT_DBKEYTREEBYTES is the number kilobytes used to index the record keys.
- LPAR_INT_DBRECFIELDKBYTES is the number of kilobytes used to store the record data.
- LPAR_INT_DBHEADERKBYTES is the number of kilobytes of memory used in storing various header data. It is the overhead.
- LPAR_INT_DBIDXTOTALKBYTES is the number of kilobytes used to store index data. This includes either GIP, PSI, or SORT prefilter index memory needs and the memory used by any predicate partitioned indexes on the table.
- LPAR_INT_DBTOTALKBYTES is the total memory occupied by the database in kilobytes.
- LPAR_INT_PROCTOTALKBYTES is the total memory occupied by the process. This is returned only for servers built for the Linux platform.
- LPAR_LONGINT_TRAN_ID is the transaction id of the transaction that owns or claims the object at that instance.
- LPAR_INT_TRAN_OBJ_STATE is the state of an object when a transaction owns or claims it; if there is no such transaction, it returns LKT_TRAN_OBJ_EXISTS.
- LPAR_INTARR_PSI_DENSITIES (PSI-indexed tables only) lists the densities of the PSI encodings.
- LPAR_STR_TABLE_TYPE is one of: “Standard”, “Child”, “Parent” or “Unknown”. This indicates the type of table. “Unknown” is returned only if there was some kind of error in creating the table.
- LPAR_STRARR_CHILD_TBLS is a list of the child tables for this table. This is returned only for a parent table. This list contains zero entries if the parent table has no child tables.

If you do not care to have any of these statistics, just pass `lkt_dbload` a pointer value of NULL for dbstats.

Error codes and items returned by lkt_dbload

ARRAYLEN	array lpar with wrong number of values

CHARCONV	record that contains illegally encoded character
DBEXISTS	name of already-existing database
DBPARAM	inappropriate item in a parameter list
DUPFIELDNAM	ES field names array with duplicate names
EXPECTDBDESC	item that should have been a table name
EXPECTLIST	item that should have been a list type lpar
FEATURESET	lpar applicable to a premium feature not available
FILEFORMAT	LPAR_STR_FILEFORMATERROR description of error
GPU_DEVICE_NOT_ALLOWED	Bad GPU device id list
INTERNAL	(none) or the record causing the error or LPAR_STR_ERRORDETAILS giving more details
IOERROR	LPAR_STR_SYSERROR description of error reading records
JOINSET	Name of table
NOCHARMAP	dbpars list containing invalid character map name
NODBDESC	(none)
NOMEM	record at which database memory cap was exceeded.
NORECKEY	record that lacks a record key
NOSRCHTXT	record that lacks searchable text
NOSYSINIT	(none)
NUMFIELDS	record that contains the wrong number of fields

PARAMCONFLICT	lpar that contains a conflicting value
PARAMTYPE	Invalid name
PARAMMISSING	list in which a required item is missing
PARAMVAL	lpar that contains an illegal value
RECEXISTS	a record containing a duplicated record key
TBLNOTCHILD	(none)
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ ERRORDETAILS
UNKFIELD	Encoding field names list with bad field name

Reading a Database From a File (lkt_dbread)

The dbread command is used to facilitate database loading. It's designed to read records from a file and produce a record list and database parameter list suitable for input to lkt_dbload.

```
dvkerr_t lkt_dbread( lpar_t file, lpar_t parameters,
                     lpar_t *reclist, lpar_t *dbpars );
```

The following is a sample CSV file and the code that would be used to parse it.

```
first,middle,last,order
5,5,5,6
key1,William,Jefferson,Clinton,42
key2,George,W,Bush,43
lpar_t filepar,configpars,reclist,dbpars,dbname;
filepar=lpar_create_str(LPAR_STR_CSVFILE,"filename.csv");
configpars=lpar_create_lst(LPAR_LST_GENERIC);
lpar_append_list(configpars,lpar_create_bool(LPAR_BOOL_FNAMESFIRST,TRUE));
lpar_append_list(configpars,lpar_create_bool(LPAR_BOOL_FTYPESFIRST,TRUE));
lpar_append_list(configpars,lpar_create_bool(LPAR_BOOL_LEADINGKEY,TRUE));
lpar_append_list(configpars,lpar_create_int(LPAR_INT_DBNUMFIELDS,4));
lkt_dbread(filepar,configpars,&reclist,&dbpars);
dbname = lpar_create_str(LPAR_STR_DBDESCRIPTOR, "dbname");
lkt_dbload(LPAR_NULL,dbname,dbpars,reclist,LPAR_NULL);
```

Two file formats are supported: Fixed Width text files and CSV files. For Fixed Width files a width is defined for each field of the record, no separators exist between fields. Every record is exactly one line, each line has exactly the same number of bytes. CSV files are standard format comma separated values files. Microsoft Excel format is used: fields containing commas, double quotes, or new-lines must be enclosed in double quotes, a double quote within a quoted field is given as two double quotes.

In addition to the data records, files might contain one or both of the special header lines: field names—the names of the fields, field types—the types of the fields given as integer values or field type names. If both are specified, the field names always appear first.

The field types are specified using any of the names or numbers below (the letters in the names are not case sensitive):

Field Types

Field Type	Recognized Names or Numbers
key field in CSV	2, key
parent key field	16, parent-key, parent_key, parentkey
non-searchable text	4, text
searchable text	5, searchtext, srchtext
integer	6, integer, int
floating point	8, float, double
date	10, date
Searchable Date	11, srchdate, searchdate
date and time	12, datetime, datet
variable attributes	15, attributes, attrs

The record key can be specified in one of 4 ways:

- **Leading Key** Each record starts with an unnamed, untyped field that holds the record key. Thus, each data record has one more field than any field names or field types header record. This is the format that was used up to release 4.4.3.
- **Field Index** One of the record fields is identified via field index position as the key field. This field is included in any field name or field type header. Thus each data record has the same number of fields as the header records. However, this field is NOT loaded as a field in the table.

  Thus, the record as loaded into the table has one fewer field than the file.
- **Field Name** This is similar to Field Index except the key field is specified by field name instead of index position. To use this, "the field names first" option must be used.
- **No Key** The record key is not given, instead keys are generated automatically when the records are read.

If the file contains child records, the parent key can be specified by field index or by field name.

An example file for each method of specifying the key is given below. Each example has both field names and field types header records. In all cases, the records have 4 fields: `first,middle,last,order`.

Leading Key
First,middle,last,order
5,5,5,6
key1,William,Jefferson,Clinton,42
key2,George,W,Bush,43

Key Index (with key index being 4)
first,middle,last,order,key
searchtext,searchtext,searchtext,int,key
William,Jefferson,Clinton,42,key1
George,W,Bush,43,key2

Key Name (with key name being "record key")
First,middle,last,record key,order
5,5,5,2,6
William,Jefferson,Clinton,key1,42
George,W,Bush,key2,43

No Key
First,middle,last,order
5,5,5,6
William,Jefferson,Clinton,42
George,W,Bush,43

Input

host (required)

is the name of a file from which the database must be read. This must be either an LPAR_STR_CSVFILE or an LPAR_STR_FWFFILE for comma separated value or fixed width field files respectively. For more information, see Communicating with TIBCO Patterns Servers.

parameters (optional)

is a list of parameters which describe the file contents. It might contain any of the following lpars:

- LPAR_STR_ENCODING defines the character encoding used in the file to be read. It should be one of "latin-1" or "UTF-8".

  Default value: "latin-1"

- LPAR_INTARR_RECSRCHFLDS contains field widths measured in bytes. This lpar is required for fixed width field files and it is ignored if used with comma separated value files. If LPAR_BOOL_LEADINGKEY is set a width must be given for the key field in addition to each record field.

- LPAR_INT_DBNUMFIELDS is the number of fields in the file. If this lpar is not provided, lkt_dbread attempt to guess this value. This count includes only actual data fields, it does not include the key field as part of the field count.
- LPAR_BOOL_LEADINGKEY is true if the first field in the file should be used as the record key.

  Default value: false
- LPAR_INT_KEYFIELD gives the index (zero based) of the field that is to be used as the key field.

  Default value: no field is used as the key field
- LPAR_STR_KEYFIELD gives the name of the field that is to be used as the key field.

  Default value: no field is used as the key field.
- LPAR_INT_INITIALKEY is the number used to generate the key for the first record in the file when no key field was specified. Each successive record is numbered one higher than the previous.

  Default value: 0
- LPAR_BOOL_FNAMESFIRST is true if the first line of the file contains field names rather than a record.

  Default value: false
- LPAR_BOOL_FTYPESFIRST is true if the first line of the file (after field names, if they exist) are field type codes. See The TIBCO Patterns Table for a description of field types.

  Default value: false
- LPAR_INT_BATCHSIZE indicates that the records read should be broken into multiple lists of at most this size. This is useful if you wish to implement an interactive database load with a periodic status update. If this lpar is not given, all of the records are placed in a single list.
- LPAR_BOOL_DOMAXWORK indicates that errors should be ignored whenever possible. For instance, if there is no record content for a given line (i.e. all the fields are empty), a DVK_ERR_FILEFORMAT is usually returned. If LPAR_BOOL_DOMAXWORK is set to true, this line is skipped and parsing begins on the next line.

  Default value: false
- LPAR_INT_PARENT_KEYFIELD gives the index (zero based) of the field that is used as the parent key field.

  Default value: no field is used as the parent key field

- LPAR_STR_PARENT_KEYFIELD gives the name of the field that is used as the parent key field.

  Default value: no field is used as the parent key field.

Only one of LPAR_BOOL_LEADINGKEY, LPAR_INT_KEYFIELD, LPAR_STR_KEYFIELD can be specified. If none of these are specified then this is considered the "No Key" case and keys are generated automatically for each record.

Output

reclist (required)

is either a list of records, or a list of lists of records, depending on whether a batch size is specified. Each list of records can be passed to the `lkt_dbload`, `lkt_dbrecadd`, `lkt_dbrecreplace` or `lkt_dbrecupdate` commands to add or update them in a database.

dbpars (optional)

is a list of database parameters determined from reading the file. These parameters are intended to be passed to the `lkt_dbload` command as the dbpars parameter. Extra parameters can be added to the list as necessary.

Error codes and items returned by lkt_dbread

ARRAYLEN	item that has the wrong length
EXPECTLIST	item that should have been a list type lpar
FILEFORMAT	a line from the file which could not be parsed
IOERROR	LPAR_STR_SYSERROR describing error
PARAMCONFLICT	lpar in conflict with an earlier lpar
	FILEFORMATERROR describing what is missing

PARAMVAL	lpar that contains an illegal value

Reading a Database From a File in Small Chunks (lkt_open_dbfile)

```
The dbread command reads an entire database file, but there might be times
when it is desirable to open a file and only read a handful of records at a
time.
dvkerr_t lkt_open_dbfile( lpar_t file, lpar_t parameters,
                          lpar_t *filehandle, lpar_t *dbpars );
```

`lkt_open_dbfile` is called like `lkt_dbread` but instead of reading records, it creates a special block lpar called a file handle. The contents of this lpar must never be altered directly, but can be passed to lkt_read_records to do the actual reading.

The input lpars for `lkt_open_dbfile` are identical to those of lkt_dbread, with the exception is that LPAR_INT_BATCHSIZE is not supported.

Open files are closed when the associated file handle lpar is destroyed with `lpar_destroy`.

Input

host file (required)

is the name of a file from which the database should be read. This must be either an LPAR_STR_CSVFILE or an LPAR_STR_FWFFILE for comma separated value or fixed width field files respectively.

parameters (optional)

is a list of parameters which describe the file contents. It might contain any of the following lpars (see Reading a Database From a File (lkt_dbread) for descriptions):

- LPAR_STR_ENCODING
- LPAR_INTARR_RECSRCHFLDS
- LPAR_INT_DBNUMFIELDS
- LPAR_BOOL_LEADINGKEY

- LPAR_INT_KEYFIELD
- LPAR_STR_KEYFIELD
- LPAR_INT_INITIALKEY
- LPAR_BOOL_FNAMESFIRST
- LPAR_BOOL_FTYPESFIRST
- LPAR_BOOL_DOMAXWORK
- LPAR_INT_PARENT_KEYFIELD
- LPAR_STR_PARENT_KEYFIELD

Output

filehandle (required)

is an opaque block lpar to be used with `lkt_read_records`.

dbpars (optional)

is a list of database parameters determined from reading the file. These parameters are intended to be passed to the `lkt_dbload` command as the dbpars parameter. Extra parameters can be added to the list as necessary.

Error codes and items returned by lkt_open_dbfile

ARRAYLEN	item that has the wrong length
EXPECTLIST	item that should have been a list type lpar
FILEFORMAT	a line from the file which could not be parsed
IOERROR	LPAR_STR_SYSERROR describing error
PARAMVAL	lpar that contains an illegal value
PARAMCONFLICT	lpar in conflict with an earlier lpar

Reading a Database From a File in Small Chunks (lkt_read_records)

The following command is used to read records from the database.

```
dvkerr_t lkt_read_records(lpar_t host, lpar_t filehandle, lpar_t *reclist,
                             int numrecords );
```

`lkt_read_records` accepts a filehandle lpar returned by `lkt_open_dbfile` and reads up to numrecords from the file and returns them in reclist.

The output of `lkt_read_records` is a list of records or LPAR_NULL if the end of the file has been reached.

Input:

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

filehandle (required)

is the open file handle returned by `lkt_open_dbfile`.

numrecords (required)

the number of records to read.

Output:

reclist (required)

is a list of records read, or LPAR_NULL if the end of the file has been reached.

Error codes and items returned by lkt_read_records

FILEFORMAT	a line from the file which could not be parsed
PARAMVAL	

Deleting a Database (lkt_dbdelete)

This command deletes a database, releasing all storage associated with it and with its records.

```
dvkerr_t lkt_dbdelete(lpar_t host, lpar_t name );
dvkerr_t lkt_dbdeleteT(lpar_t host, lpar_t name, lpar_t tran );
```

If the database has been checkpointed (see Checkpointing and Restoring a Database) the checkpoint is also deleted when the transaction is committed, and thus the database can no longer be restored with the `lkt_restore` command.

If durable-data is enabled, all of the files backing the table are removed when the transaction is committed.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

name (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) to be destroyed.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the table delete operation is to be performed.

Error codes and items returned by lkt_dbdelete

CHKPT_CORRUPT	Name of table that has corrupt checkpoint information

DBNOTFOUND	name of nonexistent database
EXPECTDBDESC	item that should have been a table name
INTERNAL	details of unexpected error encountered
JOINSET	Name of parent table that has child tables linked to it
NODBDESC	(none)
NOSYSINIT	(none)
PARAMVAL	invalid table name
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONG_INT_TRAN_ID and LPAR_STR_ERRORDETAILS

Renaming a Database (lkt_dbmove)

```
This command renames a table.
dvkerr_t lkt_dbmove( lpar_t host, lpar_t srcname, lpar_t dstname );
dvkerr_t lkt_dbmoveT( lpar_t host, lpar_t srcname, lpar_t tran, lpar_t
dstname );
```

The database's new name is dstdesc. If there already exists a database with name dstdesc, that database is deleted.

Note: If you are familiar with UNIX systems, the semantics of lkt_dbmove are exactly parallel to those of the shell command mv.

Note that this includes deleting any checkpoint or durable data of the table with name dstdesc (see Checkpointing and Restoring a Database for a description of checkpointing databases).

Any checkpoint (see Checkpointing and Restoring a Database) of the table is also renamed by this command. Thus after the lkt_dbmove command is executed an `lkt_restore` of dstdesc restore the table that was checkpointed under the name srcdesc. An `lkt_restore` of srcdesc return a not found error.

The `lkt_dbmove` command is especially useful for doing updates of entire databases without appreciable down-time. Simply load the new version of the database dstdesc under the name srcdesc, and then do a `lkt_dbmove`.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

srcname (required)

is the current name (LPAR_STR_DBDESCRIPTOR) of the database. There must be an existing loaded database with this name.

dstname (required)

is the new name (LPAR_STR_DBDESCRIPTOR). It must differ from srcdesc.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the table move operation is to be performed.

Error codes and items returned by lkt_dbmove

DBMOVESAME	names supplied as both source and destination
DBNOTFOUND	name of nonexistent source database
EXPECTDBDESC	item that should have been a database name
INTERNAL	(none) or item being processed when error occurred
JOINSET	(none) Destination overwrites parent table with

	children
NODBDESC	(none)
NOMEM	(none)
NOSYSINIT	(none)
PARAMVAL	invalid name
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONG_INT_TRAN_ID and LPAR_STR_ERRORDETAILS

Checking the Status of Loaded Databases (lkt_dblist)

```
The following command returns a list of lists of database statistics one list
per database specified.
dvkerr_t lkt_dblist( lpar_t host, lpar_t names, lpar_t *dbstatlists );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

names (optional)

is a list of database names (LPAR_STR_DBDESCRIPTOR's) of existing loaded databases. It is an error (DVK_ERR_DBNOTFOUND) if any of these databases do not exist.

A value of LPAR_NULL might be supplied for names. This causes statistics lists for all currently loaded databases to be returned.

Output

dbstatlists (required)

is the list of statistics lists. Each list is similar to that returned by the lkt_dbload command.
Error codes and items returned by lkt_dblist

DBNOTFOUND	list of names of nonexistent databases
EXPECTDBDESC	item that should have been a database name
EXPECTLIST	item that should have been a list
FEATURESET	(none)
INTERNAL	Varies
NODBDESC	(none)
NOSYSINIT	(none)

Invoking TIBCO Patterns Matching (lkt_dbsearch)

```
The following command finds matches for a given query over a list of
databases. It returns statistics, matching records, and per-match
information.
dvkerr_t lkt_dbsearch( lpar_t host, lpar_t names, lpar_t dbparlists,
                        lpar_t query, lpar_t srchpars, lpar_t *stats,
                   lpar_t *matches, lpar_t *minfo );
```

The dbsearch command supports many different types of queries, from simple single text block queries to advanced query expressions with complex specification of scoping and weighting.

In its simplest form, a query is a single block of text which is used to search all selected fields of all databases in names. Query expressions allow different blocks of query text to be specified for different fields of each record.

For a full discussion of query building, see Query Construction.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

names (required)

is a database name or a list of database names (LPAR_STR_DBDESCRIPTOR's) of those databases that is included. These databases must all be currently loaded, and not have searches locked out (see Blocking Operations On a Database (lkt_dblock, lkt_dbunlock)). There must be at least one name.

dbparlists (optional)

are database-specific search parameters. The dimensionality of this list must be one higher than that of the names. If names is a single database (not a list), dbparlists must be a single list of database parameters.

If names is a list of databases, dbparlists must be a list of lists of database parameters, one for each database in desclist. The entries in names and the parameter lists in dbparlists are associated in one-to-one fashion. If no such parameters are to be passed for any database, the value of dbparlists can be LPAR_NULL. If names is a list and no parameters are needed for a particular database the corresponding list element can be LPAR_NULL.

Note: It is perfectly legal to append the value LPAR_NULL to the following list:

Note: lpar_append_lst(dbparlists, LPAR_NULL);

Possible database specific parameters are listed as follows:

- LPAR_LST_JOIN specifies the type of join to be used for this query. The content of this list is described in the specifying a join section.

  Default Value: No join is used.

- LPAR_INTARR_SELECTFLDS controls field selection for this query. All query field references in fuzzy text matching queries are restricted to using these fields. This also defines the default set of fields for all fuzzy matching text queries. This affects only the current query.

Note: The use of integer field selections is discouraged. The fields must be selected by field names.

Each of the integer values is interpreted as a field number, with fields numbered starting from zero. For a joined search fields are numbered starting with the parent table and continuing across each child table in the order given in the child tables list. For child tables the key and parent key count as fields in the numbering order (but cannot be used in a search). They appear as the first and second field of the child table respectively. For example, given a join across a parent table with three data fields, a child table with two data fields and a second child table with three data fields, the array [0, 5, 11] would select the first data field from the parent table, the first data field from the first child table, and the third data field from the second child table.

Note: It is an error to give both LPAR_INTARR_SELECTFLDS and LPAR_STRARR_FIELDNAMES.

Note: All fields selected must be of type LKT_FLD_TYPE_SRCHTEXT or LKT_FLD_TYPE_ATTRIBUTES.

Default value: All fields of type LKT_FLD_TYPE_SRCHTEXT or LKT_FLD_TYPE_ATTRIBUTES are selected.

- LPAR_STRARR_FIELDNAMES controls field selection for this query. This is similar to the LPAR_INTARR_SELECTFLDS value, except that it identifies fields by name. Attribute names are not used. This is preferred over the use of LPAR_INTARR_SELECTFLDS. It is an error to give both LPAR_INTARR_SELECTFLDS and LPAR_STRARR_FIELDNAMES.
- LPAR_LST_DEDUPQUERY is deprecated and is not be used in new code. This list must contain a single LPAR_STR_RECKEY value. This defines the record to use as the search criteria for the SORT prefilter. If using the SORT prefilter, this value replaces the LPAR_BLKARR_SORTLOOKUPFIELDS in the search parameters. This value must not be specified if LPAR_BLKARR_SORTLOOKUPFIELDS is specified. This value is ignored if the GIP or PSI prefilter is used.
- LPAR_LST_PREDICATE, LPAR_STR_PREDICATE or LPAR_BLK_PREDICATE defines a predicate expression used to select records to be searched. Records that fail to pass the predicate expression test are not considered for the query. Because of the complexity of predicate expressions, they are discussed separately in section Predicate Expressions.

Default value: all records are considered.

- LPAR_INT_LOCKKEY enables a search to go through on a database containing a keyed lock if the key value matches the lock value. Locks are deprecated and are not to be used in new code.

specifying a join

To query across a joined set of tables the parent table is specified in names and a join specification is added to the dbparlists. A join specification is an LPAR_LST_JOIN list containing the following values:

- LPAR_STR_PARENT_TBL is the name of the parent table. This is optional and if it is given it must be the same table as specified in the names parameter.
- LPAR_INT_JOINTYPE specifies the type of join operation. This is required and is one of the following values:

  LKT_JOIN_SEARCH_OFF is a standard search against a single table.

  LKT_JOIN_SEARCH_ON[|LKT_JOIN_ALLOW_PARTIALS][|LKT_JOIN_ALLOW_ORPHANS] is a join search across a parent table and one or more child tables.

  The LKT_JOIN_SEARCH_ON value can be or'ed with either LKT_JOIN_ALLOW_PARTIALS or LKT_JOIN_ALLOW_ORPHANS.

  If LKT_JOIN_ALLOW_PARTIALS is given records missing one or more child tables are included in the output, otherwise only full records are included. If LKT_JOIN_ALLOW_ORPHANS is given orphan child records, not associated with any parent record, are included in the output, otherwise orphan child records are not considered.
- LPAR_BOOL_MULTI_PARENT_JOIN if true each qualifying combination of parent and child records is returned, otherwise only the single best combination for each parent record is returned. This is optional and its default value is false.
- LPAR_STRARR_CHILD_TBLS specifies the list of one or more child tables to be included in the join. All tables must be a child table of the table specified in the LPAR_STR_PARENT_TBL. The same child table can be specified multiple times. If a child table is specified multiple times, a unique alias must be assigned for each occurrence.
- LPAR_STRARR_TBL_ALIASES specifies an alias name for each child table. This is required only if the same child table is specified multiple times in the LPAR_STRARR_CHILD_TBLS list. If this list is given, all table name references in the query and predicate expressions must use this alias value instead of the actual table name. The length of this list must be the same as the length of the LPAR_STRARR_CHILD_TBLS list. All entries must be non-null, non-empty values. All entries in this list must be unique. An entry in this list cannot be the same as the name of a table in the LPAR_STRARR_CHILD_TBLS list at a different index position. The default value for this is the same as the LPAR_STRARR_CHILD_TBLS entries.

query (required)

is the query. This might be either a single query, or a list of queries. When this parameter is a list of queries, one query is expected for each database in name. When this is a single query, the same query is used for every database in names. Details of query construction are covered in Query Construction.

srchpars (optional)

is a list of lpars specifying general (non-database specific) search parameters. Every one of these parameters has a default value, so it is not strictly necessary to pass a srchpars list to lkt_ dbsearch. If the default values are satisfactory, simply pass the value LPAR_NULL instead.

A number of the values here set default values for query options. These defaults can be overridden on a per query basis by setting them in the query.

Here is the list of search parameters and their default values:

- LPAR_INT_MATCHESREQ sets the maximum number of matching records to be returned. The `lkt_dbsearch` command returns this number, if possible; but it might return less.

Note: Fewer records are returned if the number of records in the searched databases satisfying the selection predicate (if any) is smaller than the number of matches requested (also taking into account the value of LPAR_INT_STARTMATCH) or if a cutoff has been specified.

  Default value: 25

- LPAR_INT_STARTMATCH sets the rank (1-based) of the first match in the list of matching records to be returned (for example, a value of 11 asks for a match list beginning with the eleventh-best match).

  Default value: 1

- LPAR_INT_SORTSCORE sets the score type to use for record ranking. See Score Types for more on score types.

  Default value: LKT_SCORE_NORMAL

- LPAR_BOOL_FAILBADPRED controls the behavior of predicate evaluation failure in filtering predicates. When this value is true, records which cannot be evaluated are excluded from the search. When this value is false, records are searched if predicate evaluation fails. See Controlling Predicate Error Handling for more discussion of the meaning of this lpar.

  Default value: true

- LPAR_BOOL_FAILEMPTYPRED controls the behavior of predicate evaluation on empty field values in filtering predicates. When this value is true, tests on empty record field values are considered to be invalid and the predicate test fails with a "empty value" error. The behavior is then controlled by the value of the LPAR_BOOL_FAILBADPRED setting. When this value is false empty field values are treated as any other value and the predicate test is performed. See Controlling Predicate Error Handling for more discussion of the meaning of this lpar.

Default value: true

- LPAR_DBL_INVALID_DATA_SCORE is the score returned by predicate and custom scoring queries if the record or query contains invalid data. You should consult your TIBCO representative before setting this parameter.

  Default value: 0.0

- LPAR_DBL_EMPTY_DATA_SCORE is the score returned by a query if the record or query contains empty data. You should consult your TIBCO representative before setting this parameter.

  Default value: 0.0

- LPAR_BOOL_MATCH_EMPTY controls behavior when a query string and the data it is being matched to are both empty. If true, a 1.0 score is generated. If false, the empty-score is used.

  Default: false

- LPAR_INT_VISUALSTYLE selects the visualization style for matching records. Currently, the only legal values for this parameter are 0, 2, and 256. A value of 0 is the null style (no visualization information are returned). See Match Visualization for a detailed discussion of visualization style 2 and visualization style 256 and other search options associated with them.

  Default value: 0 (no visualization)

- LPAR_LST_TIEBREAKS selects custom tie breaking rules. See Tie Breaking for more information on custom tie breaks.

  Default value: Tie break first on symmetric score, then on record Key.

- LPAR_LST_CUTOFF selects the dynamic score cutoff method. See Dynamic Score Cutoffs for more information on dynamic score cutoffs.

  Default value: No cutoff is applied.

- LPAR_STR_THESAURUSNAME selects the default thesaurus used to compare records. See Thesaurus Matching for more information about TIBCO Patterns servers' thesaurus support. Only one of LPAR_STR_THESAURUSNAME or LPAR_LST_THESAURUS might be specified.

  Default value: none (no thesaurus)

- LPAR_LST_THESAURUS defines a default thesaurus used to compare records. The content of the list consists of two lpars, these being exactly the same as the `thesaurus_options` and `thesaurus_data` arguments for the `lkt_create_thesaurus` command. See Thesaurus Matching for more information about these arguments and TIBCO Patterns servers' thesaurus support in general. The thesaurus defined exists only for the duration of the query and is local to the query. See Ephemeral Thesauri for more information on these ephemeral thesauri in particular. Only one of LPAR_STR_THESAURUSNAME or LPAR_LST_THESAURUS might be specified.

  Default value: none (no thesaurus)

- LPAR_DBL_THESAURUSWEIGHT sets the default weight given to a thesaurus match compared to a non-thesaurus match of equal quality. See Thesaurus Matching for more information about TIBCO Patterns servers' thesaurus support.

  Default value: 1.0 (equal weight)

- LPAR_STR_CHARMAP is the name of a character map to be used for all queries. It might be necessary to set this value if you need to perform a single simple or cognate query across multiple fields with different character maps.

  Extreme caution must be used when setting this value. You should not set this value without consulting your TIBCO representative. See Character Mapping for more information on character maps.

  Default value: Character map for the field being queried.

- LPAR_STR_RLMODELNAME is the default Learn model to use for all Learn queries. See Using a Learn Model in a Search for a description of this parameter.

  Default value: none, if using Learn queries it must be given either here or with the Learn query.

- LPAR_LST_RLPREDLIST (Deprecated) is an obsolete method of adding Predicate queries to a Learn query. Use Predicate queries instead.

  Default value: No Predicate queries added.

- LPAR_BOOL_RLUSESSYMSCORES (Deprecated) when given and true, the default match score type is Symmetric instead of Normal.

  Default value: false, use Normal score as the default.

- LPAR_DBL_NONCOG_WGT is the default non-cognate field weight for all cognate queries. See Cognate Query for a description of the non-cognate weight.

- LPAR_LST_QOPTS is the default set of options for all queries. Most of these options provide for highly detailed control over how the TIBCO Patterns server performs matching allowing it to be tuned for unusual situations. Nothing should be changed without consulting your TIBCO representative.

- LPAR_BOOL_DOMAXWORK effects the return in the case of a multi-database search if some but not all of the databases are not found. With a value of true, results are returned from whatever databases are found. The search returns a DVK_ERR_PARTIAL error code, along with a list of unavailable databases. With a value of false, results are returned only if all databases are present.

  When working through a gateway on a federated table setting this flag to true cause any search to return partial success if one or more nodes are unreachable. A result of partial success means only a subset of the entire table was searched when searching a federated table. For more information about gateways and federated tables, see Federated Tables and Gateways.

  Partial results are not valid results as they represent a search over an unknown portion of the total data. Therefore, use of this flag is extremely rare and should be done only after careful consideration.

  Default value: false

- LPAR_BOOL_GIPSEARCH controls whether the GIP prefilter is enabled for this search.

  Default value: true if GIP prefilter is enabled for the table being searched.

- LPAR_BOOL_SORTSEARCH Controls whether the SORT prefilter is used. See The SORT Prefilter - Usage Details for details on this and other parameters associated with the SORT prefilter.

  Default value: true if SORT prefilter is enabled for the table being searched.

- LPAR_BOOL_PSISEARCH Controls whether the PSI prefilter is used. See The PSI Prefilter - Usage Details for details on this and other parameters associated with the PSI prefilter.

  Default value: true if PSI prefilter is enabled for the table being searched.

- LPAR_INT_PSI_DENSITY Controls PSI suffix density for queries on PSI-indexed tables. See The PSI Prefilter - Usage Details for a full discussion of this parameter.

- LPAR_BLKARR_SORTLOOKUPFIELDS Sets the search criteria for the SORT prefilter. See The SORT Prefilter - Usage Details for details of this and other parameters associated with the sort filter. If using the SORT prefilter either this value or LPAR_LST_DEDUPQUERY in the database parameters might be required. If it is not given, the TIBCO Patterns server attempts to infer the lookup fields from the query. If it cannot do so, an error is returned.

- LPAR_BLKARR_PSILOOKUPFIELDS Sets the search criteria for the PSI prefilter. See The PSI Prefilter - Usage Details for details of this and other parameters associated with the PSI filter. If using the PSI prefilter this value might be required. If this value is not given, the TIBCO Patterns server attempts to infer the lookup fields from the query. If it cannot do so, an error is returned.

- LPAR_INT_FETCH_SIZE sets the size of the candidate set selected by the GIP, SORT, or PSI prefilter. This value should not be set unless directed to do so by your TIBCO technical representative.

  Default value: dynamically calculated
- LPAR_INT_PC_SIZE this value is used for tuning query performance, it shouldn't be set unless you are directed to do so by your TIBCO technical representative.
- LPAR_INT_QC_SIZE value is used for tuning query performance, it is not to be set unless you are directed to do so by your TIBCO technical representative.
- LPAR_BOOL_DETAILEDSTATS request more detailed statistics information. If set true more detailed information on internal performance data is returned. This should not be set unless instructed to do so by your TIBCO representative as it has significant impact on the performance.

  Default value: false
- LPAR_INT_GPU_PARALLELISM is the number of parallel GPU channels to use per query. If the value is outside the range accepted by the server, it is clamped to the nearest acceptable value.

  Minimum value: 1, Maximum value: 6, Default value: 4
- LPAR_INT_INT_GPU_B_P_K is the number of memory blocks to use per GPU unit. If the value is outside the range accepted by the server, it is clamped to the nearest acceptable value. Minimum value: 4, Maximum value: 192. The default is calculated by the server based on the GPU hardware. Overriding the server's default calculation is likely to reduce performance. Ignored for Join queries.
- LPAR_BOOL_DBGIPGPU if true employ GPU units to speed GIP processing, if false, disables GPU processing for this query. A FEATURESET error is returned if GPU is not enabled for all of the queried tables.

Output

stats (required)

is a list of lpars giving statistics related to the just-completed matching. It includes the following lpars (other advanced search statistics might also be present):

- LPAR_INT_MATCHESRET is the actual number of matches returned in the matches list.
- LPAR_INT_VISUALSTYLE is the visualization style. See Match Visualization for more information on visualization

- LPAR_BLK_HTMLLEGEND is an HTML segment that can be used as a key to interpret the HTML-formatted text of visualization style 256. See Match Visualization for more information regarding visualization. Returned only if visualization style 256 is selected.
- LPAR_BOOL_GIPSEARCH tells whether the GIP prefilter was used for this query.
- LPAR_BOOL_DBGIPGPU true if GPU acceleration was used in the search. This does not appear if GPU acceleration is not available with this server.
- LPAR_BOOL_SORTSEARCH tells whether the SORT prefilter was used for this query.
- LPAR_BOOL_PSISEARCH tells whether the PSI prefilter was used for this query.
- LPAR_STRARR_FIELDNAMES is a list of the field names for the table or tables queried, in the order they appear in the returned records. For joined searches, the key and parent key of each child record is included in the list as the first two fields of the child record. The key appears first and has the name "*table-name*". Where *table-name* is the name of the child table. It is followed by a period. The parent key appears second and has the name “*table-name.***^parent**”, where *table-name* is the name of the child table. If aliases for the child tables were specified the alias name is used for *table-name*.

  For example:

  A parent table with fields: “first”, “last”

  A child table (addresses) with fields: “street”, “city”, “state”

  A second child table (phones) with field: “phone”

  An alias list “a”, “p”

  The output field name list is:

  first, last, a., a.^parent, a.street, a.city, a.state, p., p.^parent, p.phone

- LPAR_INTARR_FIELDTYPES is a list of the field types for the database(s) given in the same order that they appear in the returned records. See The TIBCO Patterns Table for information on field types.
- LPAR_DBL_SEARCHTIME is the number of seconds of CPU time used by the process during matching. On most platforms this value is meaningful only if no other commands were being run in parallel with this query.
- LPAR_DBLARR_FILTERTIMES is a breakdown of seconds of CPU time by different steps of the search.
- LPAR_STRARR_FILTERTIME_NAMES are the names of the search steps reported in LPAR_ DBLARR_FILTERTIMES.
- LPAR_INTARR_FILTEROUTSIZES are the number of records output by each filter step.

- LPAR_STRARR_FILTEROUT_NAMES are the names of the search steps reported in LPAR_DBLARR_FILTEROUTSIZES.
- LPAR_STRARR_QLETNAMES is a list of names of all of the named querylets in the query. These are given in the same order as the scores in the LPAR_DBLARR_NAMEDQLETSCORES entry in the minfo values.
- LPAR_DBL_MIN_CONFIDENCE is the minimum confidence value seen during the query processing. This might be the confidence value of a record that is not returned in the result set. Confidence is relevant only if a TIBCO Patterns Learn Model is used.
- LPAR_DBL_RESULT_CONFIDENCE is the minimum confidence value of any of the records returned in the return set. Confidence is relevant only if a TIBCO Patterns Learn Model is used.

matches (required)

is a list of records, ranked by the TIBCO Patterns servers according to their likeness to the query. Each record is returned with its key and complete data. All of this information might be retrieved using the record selector functions described above in Processing the Output of DevKit Commands.

minfo (optional)

is a list of lists of lpars, one lpar list per record returned in matches. Each lpar list can include the following lpars:

- LPAR_INT_MATCHRANK is the rank (1-based) of this matching record.

Note: A rank of *n* does not necessarily mean that this is the nth item in the matches list. If you set the value of LPAR_INT_STARTMATCH to something other than its default of 1, for example 21, then the first item in matches have a match rank of 21.

- LPAR_STR_SRCDATABASE is the name of the source data base for this record. For a join, this is also the name of the parent table in the joined set of tables.
- LPAR_BLK_DBINFO is the database info block associated with the source database, if such a block was defined at database load time.
- LPAR_DBL_MATCHSCORE is the match score of this matching record, which is always a value between 0.0 and 1.0 (larger values indicate better matches). A full discussion of match scores is in An Explanation of Match Scores.

- LPAR_DBL_NORMMATCHSCORE is the normal match score of this matching record, which is always a value between 0.0 and 1.0 (larger values indicate better matches). A discussion of different types of match scores is in Score Types.
- LPAR_DBL_SYMMATCHSCORE is the symmetric match score of this matching record, which is always a value between 0.0 and 1.0 (larger values indicate better matches).
- LPAR_DBL_REVMATCHSCORE is the reverse match score of this matching record, which is always a value between 0.0 and 1.0 (larger values indicate better matches).
- LPAR_DBL_MINMATCHSCORE is the minimum match score of this matching record, which is always a value between 0.0 and 1.0 (larger values indicate better matches).
- LPAR_DBL_MAXMATCHSCORE is the maximum match score of this matching record, which is always a value between 0.0 and 1.0 (larger values indicate better matches).
- LPAR_DBL_ITMATCHSCORE is a score from 0 to 1.0 indicating the information theoretic score for this record. Contact your TIBCO representative if you're interested in learning how IT scores are computed.
- LPAR_DBL_RLRLINKSCORE is a score from 0 to 1.0 indicating the TIBCO Patterns machine learning model score for this record. This is outputted only for Learn queries.
- LPAR_DBL_RLRECORDSCORE is a score from 0 to 1.0 indicating the raw match strength of the record before the TIBCO Patterns machine learning model was applied. This is outputted only for Learn queries.
- LPAR_DBLARR_RLFEATUREVEC feature scores the TIBCO Patterns machine learning model used in determining an overall score.
- LPAR_DBLARR_RLSIGNIFICANCES is a relative measure of importance that the Patterns machine learning model assigned to each feature in the input feature list. Each value is a number in the range of 0 (least important) - 1.0 (most important). This is returned only if a Learn Model is employed and the model determined that this record is a matching record.
- LPAR_DBL_RLCONFIDENCE is the confidence the TIBCO Patterns Learn Model has in the score it generated. Values are in the range of 0.0 (no confidence at all) to 1.0 (fully confident).
- LPAR_DBLARR_MATCHSCORE_QLT is an array of match scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall match score (LPAR_DBL_MATCHSCORE). Only query combiners output querylet scores.
- LPAR_DBLARR_NORMMATCHSCORE_QLT is an array of normal match scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall normal match score (LPAR_DBL_NORMMATCHSCORE). Only query combiners output querylet scores.

- LPAR_DBLARR_SYMMATCHSCORE_QLT is an array of symmetric match scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall symmetric match score (LPAR_DBL_SYMMATCHSCORE). Only query combiners output querylet scores.
- LPAR_DBLARR_REVMATCHSCORE_QLT is an array of reverse match scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall reverse match score (LPAR_DBL_REVMATCHSCORE). Only query combiners output querylet scores.
- LPAR_DBLARR_MINMATCHSCORE_QLT is an array of minimum match scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall minimum match score (LPAR_DBL_MINMATCHSCORE). Only query combiners output querylet scores.
- LPAR_DBLARR_MAXMATCHSCORE_QLT is an array of maximum match scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall maximum match score (LPAR_DBL_MAXMATCHSCORE). Only query combiners output querylet scores.
- LPAR_DBLARR_ITMATCHSCORE_QLT is an array of information theoretic scores, one for each querylet used in the search. Each element in the array is the quality of a match between that querylet and the record. These scores are combined to give the overall reverse match score (LPAR_DBL_ITMATCHSCORE). Only query combiners output querylet scores.
- LPAR_INT_QALIGN is the alignment offset for simple and cognate queries. This defines the number of character positions the query was offset when matching the record value. This is output only if the top level query was a simple or cognate query. No meaningful value exists for query combiners. (See Query Construction for a discussion of simple, cognate and query combiners.)
- LPAR_DBLARR_NAMEDQLETSCORES contains the sort score for all the named querylets in the query. The scores are given in the same order as the names in the LPAR_STRARR_ QLETNAMES entry in the returned stats.
- LPAR_DBLARR_NAMEDQLETCONFIDENCES contains the confidence measure output by all named querylets in the query. The measures are given in the same order as the names in the LPAR_STRARR_QLETNAMES entry in the returned stats.
- LPAR_DBLARR_V2V is an array of character match strengths. It gives an overall match strength for each character in the searchable text of this record. Output only if visualization style 2 is selected. See Match Visualization for more information on match visualization.

- LPAR_INTARR_V2P is an array of match block size for each character in the searchable text of the record. It gives the number of characters in the matched block of characters containing this character. See Match Visualization for more information on match visualization.
- LPAR_INTARR_V2D is an array of displacements for each character in the searchable text of the record. It gives the displacement of the record character relative to the query character. See Match Visualization for more information on match visualization.
- LPAR_INTARR_V2N is an array of noise flags for each character in the searchable text of the record. It is 1 if this character was deemed to be noise (a randomly matched character of no value), otherwise zero. See Match Visualization for more information on match visualization.
- LPAR_BLK_HTML is the matching record's searchable text containing HTML tags for visualization. It is returned if the visualization style is 256, see Match Visualization for customization options.
- LPAR_INTARR_HTMLFLDS is the lengths of the record fields, in bytes, in the LPAR_BLK_HTML value. It can be used to split this value into individual field values. It is returned if the visualization style is 256. See Match Visualization for more information on visualization.
- LPAR_STR_MATCHTYPE is a string containing the match type. Currently this is always typographic.
- LPAR_BOOL_TRAN_DIRTY is set to true if this record is modified by an open transaction.

In addition to those listed here there might be additional items that are used internally by the TIBCO Patterns servers.

If you wish to ignore this extra match info, you can pass a NULL pointer as minfo to `lkt_dbsearch`.

Error codes and items returned by lkt_dbsearch

ARRAYLEN	array lpar with wrong number of values
DBNOTFOUND	name of non-existent database
EXPECTDBDESC	item that should have been a table name
EXPECTLIST	item that should have been a list lpar
EXPECTQUERY	item that should have been a search query

EXPIRED	(none)
FEATURESET	lpar applicable to a premium feature not available
INTERNAL	item being processed at time of error
JOINSET	lpar containing bad set of joined tables.
LISTLEN	list (dbparlists) that has the wrong number of items
LOOKUP	(none) (error in search operation)
LOOKUP_FIELDS_NEEDED	search parameters missing lookup fields
MODEL_UNSUPPORTED	(none)
NOCHARMAP	character map that doesn't exist
NODBDESC	(none)
NOQUERY	(none)
NORLINKMODEL	Learn model name lpar
NOSYSINIT	(none)
NOWGTFLD	query expression missing the LPAR_STR_WGTFLD_NAME
NSFIELD	field selection lpar
NUMFEATURESMISMATCH	Learn expression lpar
PARAMCONFLICT	lpar that contains a conflicting parameter
PARAMMISSING	the list missing a required item
PARAMTYPE	lpar with invalid ID
PARAMVAL	

PARTIAL	list of unavailable databases
PREDSTRING	LPAR_STR_ERRORDETAILS details on syntax error
PREDTYPE	lpar that contains a non-Boolean predicate
QLETREFBROKEN	LPAR_STR_QLETNAME item containing name of querylet that could not be found.
QLETREFLOOP	(none)
QUERYEXPR	lpar that contains the query specification
RECNOTFOUND	list that was missing a valid record key
SRCHPARAM	item that should have been a search parameter
THESNOTFOUND	name of thesaurus not found
UNKFIELD	A field name lpar containing an unknown field name

Online Cluster Configuration Modification (lkt_cl_cfg_reload)

This command is used to load a new configuration into a running gateway.

```
lkt_cl_cfg_reload(lpar_t host, lpar_t filedesc , lpar_t options);
```

It has following restrictions:

- There can be no open transactions when the configuration is reloaded.
- Loading a new configuration, blocks all modify commands until the new configuration is loaded.
- The new configuration might only differ from the existing configuration in the following ways:
  - Node host names and ports might change
  - The order of existing node definitions must be preserved

Input

host (required)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

filedesc (required)

is the name of the new file. This must be either a full path name of a file, readable by the gateway server process or a relative path based on the working directory of the gateway server process.

options

for future use.

Error codes and items returned by lkt_dbsearch

IOERROR	lpar that contains the bad file desc
CL_CONFIG	lpar that contains the descriptor of the bad config file
CL_CFG_CONFLICT	LPAR_STR_LAYOUT with name of the layout that has an issue
TRAN_EXISTS	(none)
GATEWAY	(none)

Termination and Cleanup (lkt_devkit_shutdown)

The following command shuts down the DevKit, and releases all storage associated with all loaded databases.

```
dvkerr_t lkt_devkit_shutdown( void );
```

It does not release any storage allocated by calls to the creator functions for lpars, records or lists documented in Input and Output Parameter Types. The number of calls to this function must match the number of calls to `lkt_devkit_init` in order for storage to actually be deallocated.

If the DevKit is initialized three times, it must be shut down three times as well. This behavior allows multiple threads of execution or independent blocks of code to each initialize and shut down the DevKit and the DevKit is guaranteed to be available to each thread or routine.

Error codes and items returned by lkt_devkit_shutdown

NOSYSINIT	(none)

Getting Version Information (lkt_devkit_version)

This function returns a string lpar with id LPAR_STR_DEVKITINFO. The lpar value is a printable string giving version numbers and compilation information for this version of the DevKit.

```
lpar_t lkt_devkit_version( void );
```

The output of this function should be destroyed with a call to `lpar_destroy` when you're finished with it.

An Explanation of Match Scores

To fully understand how to use the TIBCO Patterns servers' search technology, it's important to have a basic understanding of what the match score returned by the search engine does and does not mean.

What the match score is not is a percentage of the query found in the record. In other words, a match score of 0.2 does not mean that only 20% of the text in the query string was found in the record text.

The match score is an estimate of the likelihood (or probability) that a given record matches the query, based solely on the record and the query.

Note: This is not to be confused with probabilistic matching techniques used by other search engines. Probabilistic matching produces a match score based on a relative probability distribution function built from the database contents, meaning that the score of a single record also depends on the contents of all other records. The TIBCO Patterns servers uses an absolute probability distribution function which does not change as database contents change.

This means that it is not a good idea to arbitrarily cut off search results at a fixed match score value as is commonly done with percent-of-query match scores. A match score of 0.2 means that it is 20% likely that the record relates to the query, which is still a fairly high probability and records such as this should not be discarded unless there are large numbers of records with higher match scores.

This is why the TIBCO Patterns servers returns a fixed number of matches by default rather than returning matches above a static score threshold. There are several dynamic score cutoffs that are used while searching. See Dynamic Score Cutoffs for more information.

Score Types

A search computes several types of scores, each of which is appropriate for a separate problem domain. Score types are defined by the `lscore_t` enumerated type in the DevKit header.

By passing an LPAR_INT_SORTSCORE parameter at search time, you can select which type of score is used for ranking records.

Score types include the following:

LKT_SCORE_NORMAL	This is the default search which looks for the query text inside the record text. The presence of extra information in the record not found in the query does not penalize the record score. Use this score type for a substring or keyword search.
LKT_SCORE_ SYMMETRIC	A symmetric search compares the full texts of both the query and record and evaluates their similarity. If the record contains information not present in the query, the score are lower than if that information had not been present. Use this score only when the query represents the entirety of the text expected to be found.

LKT_SCORE_REVERSE	A reverse search compares the full text of the record against the value of the query. This type of search looks for the record text inside the query text. The presence of extra information not found in the query does not penalize the record score. Possible uses for this type of search would be retrieving lists of standard keywords, locations or names embedded within a body of text.
LKT_SCORE_MIN	This is the minimum of the normal and reverse scores. In some situations when matching record to record this might be a better indicator of match quality than the symmetric score. This puts a higher penalty on extra data in either the record or the query than a symmetric search would. Note that as combining query expressions such as AND compute each score type independently the final minimum score might not be equal to either the final normal or reverse scores.
LKT_SCORE_MAX	This is the maximum of the normal and reverse scores. This is useful in the fairly rare situation where either the query or the record might be a subset of the other. Therefore you do not want to be penalized for extraneous data in either the record or the query. Examples might be when you are matching fields where there is a core set of relevant data in the field along with extra information that might or might not appear. Note that as combining query expressions such as AND compute each score type independently the final maximum score might not be equal to either the final normal or reverse scores.
LKT_SCORE_IT	scores the record using a special "information theoretic" method proprietary to TIBCO. The method should only be used if your TIBCO technical representative requires you to do so. Details on this scoring method can be provided at your request.

All six score types are computed and returned for each record and placed in lpars called LPAR_DBL_NORMMATCHSCORE, LPAR_DBL_SYMMATCHSCORE, LPAR_DBL_REVMATCHSCORE, LPAR_DBL_MINMATCHSCORE, LPAR_DBL_MAXMATCHSCORE and LPAR_DBL_ITMATCHSCORE respectively. In addition to these, the lpar LPAR_DBL_MATCHSCORE holds a copy of whichever match score was used to select and sort the records in the list.

Tie Breaking

Custom tie breaking rules allow you to control how records are ordered in the event that multiple records receive the same score. Tie breaks are specified by attaching an LPAR_LST_TIEBREAKS list lpar to the search parameters list. This is an ordered list of tie breaking rules. The first rule in the list is used as the primary tie breaking rule. If this rule also results in a tie, the next rule is used. If all tie breaking rules are exhausted, the resulting ordering is not defined. The tie breaking rules available are:

LKT_TIEBREAK_FIELDVAL	Records are ordered based on the comparison of the value of a field in the records. The field to compare is specified by adding an LPAR_INT_TIEBREAKPARAM to the LPAR_LST_TIEBREAK list with the value being the field number. Text fields are ordered lowest first (that is alphabetically), other fields are ordered greatest first.
LKT_TIEBREAK_ALIGNMENT	Records where most of the matching text occurs near the front of the record are ordered before records where most of the matching text occurs near the end of the record. As alignment is only meaningful for cognate and simple queries this tie breaking rule is ignored if query expressions are used.
LKT_TIEBREAK_RECID	Records are ordered in sorted order by their record key. The ordering is based on the raw code values of the bytes and does not take into consideration any locale ordering rules for characters. As record keys must be unique this is guaranteed to break a tie and produce a fixed, stable, ordering for the records.
LKT_TIEBREAK_RECLEN	This is deprecated. This is equivalent to LKT_TIEBREAK_SCORETYPE with a score type of symmetric.
LKT_TIEBREAK_SCORETYPE	Records are ordered first by the primary sort score and then ties are broken based on a secondary score type. For instance, you can sort primarily by normal score and then tie break by symmetric score. The score type is specified by adding an LPAR_INT_TIEBREAKPARAM to the LPAR_LST_TIEBREAK list with a value of: LKT_SCORE_NORMAL, LKT_SCORE_REVERSE, LKT_SCORE_SYMMETRIC, LKT_SCORE_MIN, LKT_SCORE_MAX or LKT_SCORE_IT.

Each tie break is specified by adding an LPAR_INT_TIEBREAK or LPAR_LST_TIEBREAK to the tie break list.

The following sample code creates a tie breaking rule set which orders records first by symmetric score, then by record key. This is in fact the default tie breaking rule used if no tie breaking rule is specified.

```
lpar_t tiebreaks;
lpar_t tb;
tiebreaks = lpar_create_lst(LPAR_LST_TIEBREAKS);
tb = lpar_create_lst(LPAR_LST_TIEBREAK);
lpar_append_lst(tb, lpar_create_int(LPAR_INT_TIEBREAK,LKT_TIEBREAK_
SCORETYPE));
lpar_append_lst(tb, lpar_create_int(LPAR_INT_TIEBREAKPARAM,LKT_SCORE_
SYMMETRIC));
lpar_append_lst(tiebreaks, tb);
tb = lpar_create_int(LPAR_INT_TIEBREAK,LKT_TIEBREAK_RECID);
lpar_append_lst(tiebreaks,tb);
```

The lpar tie breaks can be appended to the search parameter list to use these tie breaking rules in a search.

Dynamic Score Cutoffs

Because it is inappropriate to cut off search results at a static score threshold, there are several configurable dynamic methods which can be used. Cutoff types are defined by the lcutoff_t enumerated type in the DevKit header. A cutoff is specified by creating an LPAR_LST_CUTOFF which contains an LPAR_INT_CUTOFFTYPE and optional parameters specific to that type of cutoff.

All cutoffs use the score type specified in LPAR_INT_SORTSCORE (LKT_SCORE_NORMAL by default) to perform the cutoff.

Cutoff types are shown below with their associated parameters.

LKT_CUTOFF_NONE	A cutoff of this type behaves the same way as if no LPAR_ LST_CUTOFF is specified at all.
LKT_CUTOFF_EXACTPLUS	A cutoff of this type returns exact matches plus a specified

	number of additional records. The number of additional records is given by adding an LPAR_INT_DBNUMRECORDS to the LPAR_LST_CUTOFF. This parameter must be supplied.
LKT_CUTOFF_ PERCENTOFTOP	A cutoff of this type returns all records with a match score greater than or equal to a specified percentage of the match score of the first record returned. The percentage is given by adding an LPAR_DBL_PERCENTAGE to the LPAR_ LST_CUTOFF. Valid values for this parameter are doubles from 0 to 100. This parameter must be supplied.
LKT_CUTOFF_SIMPLEGAP	A cutoff of this type returns all records until a gap between the match scores of consecutive records is greater than a specified percentage of the match score of the first record returned. The percentage is given by adding an LPAR_DBL_ PERCENTAGE to the LPAR_LST_CUTOFF. Valid values for this parameter are doubles from 0 to 100. This parameter must be supplied.
LKT_CUTOFF_ABSOLUTE	A cutoff of this type returns all records above a fixed absolute score. The score is given by adding an LPAR_DBL_ MATCHSCORE to the LPAR_LST_CUTOFF. Valid values for this parameter are doubles from 0.0 to 1.0. The parameter must be supplied. This cutoff type is not dynamic and should be used with caution, as it might very easily cut good records.

Score cutoffs only reduce the number of records returned from a search. Under no circumstances you ever receive more records than the number specified by the LPAR_INT_MATCHESREQ search parameter (or its default value, if none is specified). For example, if you request a cutoff type of LKT_CUTOFF_EXACTPLUS, this is not guarantee that you receive all exact matches. If there are more exact matches in your database than your requested number of records, you only receive a portion of the exact matches.

In addition to specifying a cut off with the LPAR_LST_CUTOFF parameter in the search options, a cutoff can be specified by passing a true value for the LPAR_BOOL_USEMODELTHRESH parameter in the LPAR_LST_QOPTS list of an Learn score combiner. If this is done and the Learn model contains a threshold value, that value is used as the cutoff value to the LKT_CUTOFF_ABSOLUTE cutoff method. This supersedes any cutoff specified in the search options.

Partition Indexes

Partition Indexes are used in conjunction with record filtering predicates to speed the selection of a subset of records based on some exact matching criteria, e.g. search only those patient records with a date of birth within two years of May 8, 1964. See dbparlists in Invoking TIBCO Patterns Matching (lkt_dbsearch) for how to specify a filtering predicate, and Predicate Expressions for a description of predicates in general.

Partition indexes are defined for a table when the table is created. Indexes can neither be added nor removed after a table is created. A partition index is only effective with the GIP prefilter. A partition index is defined by adding a LPAR_LST_PIDXDEF LPAR to the dbpars parameter of the lkt_dbload command (see Loading a Database (lkt_dbload) containing the following items:

- LPAR_INT_PIDXTYPE: Partition indexes are created as either primary or secondary indexes. A primary index provides greater improvements in speed especially for large tables and highly selective filtering predicates, but at the cost of significantly higher memory usage to store the index. Set this LPAR to 1 for a primary index, 0 for a secondary index. All other values are invalid and are rejected.

  Default value: None. This LPAR is required.

- LPAR_STR_PIDXFIELD: Each index must be associated with exactly one field in the table. The field might be of any data type, it need not be a searchable text field.

  Default value: None. This LPAR is required.

- LPAR_BLKARR_PIDXPARTITIONS: A partition index works by partitioning records into groups based on the value of a selected field. This LPAR is used to define partitions for the field selected by the LPAR_STR_PIDXFIELD LPAR. Each value in this block array defines the upper limit value of a partition in the index.

  The values given must be valid for the field type of the selected field. For example, if the field type is integer, all values in this block array must be valid American Standard Code for Information Interchange (ASCII) representations of integers.

  In addition to being valid values for the field type they must be given in sort order from lowest to highest. The sort order is based on the field type. Thus if the field type is integer a valid ordering would be "20", "100", "1000". But if the field type was searchable text this would be an invalid ordering as 20 would sort after the other values in a lexical sort.

  The number of partitions in the index are equal to the number of elements in this block array plus three. There is an additional partition to hold all values greater than the last element in this array. In addition there are always two special partitions for records with empty values in the selected field and records with invalid values in the selected field. There is a limit on the number of partitions that might be defined. Currently this limit is set to 1026, therefore there might be no more than 1023 entries in this block array.

An example of a partition set for the state field of an address would be the ordered list of state codes: AL, AK, AZ, ...WY.

Default value: None. This LPAR is required.

- LPAR_BOOL_PIDXNORMALIED: When comparing against text data predicate comparisons might be against either the raw field value or the normalized data. Normalization consists of mapping all letters to a common case (thus normalized comparisons are also referred to as case insensitive), removing diacritic marks, and a number of other operations. If indexing a text field you must specify whether you wish to index for the normalized, i.e. case insensitive, comparisons or the raw data comparisons. Include this LPAR and set it to true to index for normalized comparisons. Do not include it or set it to false to index for raw comparisons. If the indexed field is not a text field this LPAR is ignored.

  Default value: False.

Any number of indexes might be defined for a table. A field might have multiple indexes. For example, you might create both a normalized and a raw index on a text field.

You might define multiple primary indexes, however each additional primary index after the first one incur a very large overhead in memory usage. The first primary index incurs a moderate amount of overhead, secondary indexes have low overhead.

When used correctly primary indexes can provide a very large speed up in query through put. Primary indexes provide the best advantage on very large tables where the filtering predicate selects only a small percentage of the total number of records. Generally, if a predicate expression filters out over 95 percent of all records a primary index provides a large advantage over secondary or no index. If the predicate expression generally filters out less than 90 percent of records a secondary index might be suitable. If the predicate expression filters out only a few percent of records it is probably not worth creating an index.

It is important to note that not all predicate expressions that involve an indexed field can take advantage of the index. Only comparisons based on the native field value can use the index. For example, if there is an index on the text field: "birthday" a predicate expression of the form:

```
DATE $"birthday" > DATE "12/7/1986"
```

is not able to take advantage of the index as the value is converted to DATE before being compared.

The operators IN, I_IN, SUBSET, SUPERSET and predicate functions do not work with predicate indexes. So even if the address field is indexed, the predicate expression:

```
"Main" IN $"address"
```

is not be able to take advantage of the index and gets no improvement in performance.

Additionally, combining expressions with a mix of AND and OR operators might make it impossible to take advantage of the indexes on the table. Consult your TIBCO representative for help in selecting the indexes and predicate expressions that are best for your particular situation.

Detailed Description

This section describes additional commands to the core set described in the previous section, and provides more detailed information on using the TIBCO Patterns server.

Updating Loaded Databases

The commands described in Essential DevKit Commands are sufficient for many applications that do not do real-time data updates. By using the commands described in this section you can update databases in real-time.

While a command like `lkt_dbrecreplace` or `lkt_dbrecadd` is modifying a database, all other commands that would either read or modify that database are automatically blocked. They are automatically unblocked as soon as the database-modifying command has completed. See Multithreaded Programming With the DevKit for more details on multi-threaded programming with the DevKit.

Adding Records to a Database (lkt_dbrecadd)

This command adds one or more records to a database.

```
dvkerr_t lkt_dbrecadd(lpar_t host, lpar_t dbname, lpar_t reclist,
                       lpar_t params,lpar_t *dbstats );
dvkerr_t lkt_dbrecaddT(lpar_t host, lpar_t dbname, lpar_t tran, lpar_t
reclist,
                       lpar_t params, lpar_t *dbstats );
```

Statistics are returned for the modified database. The existing records are not replaced; the commands `lkt_dbrecreplace` and `lkt_dbrecupdate` provide that functionality (see Replacing Records in a Database (lkt_dbrecreplace) and Updating Records in a Database (lkt_dbrecupdate)).

Starting with release 4.4.1 adding of records is optimized based on the number of records to be added. Passing the LPAR_INT_DBNUMRECORDS value in the params list informs the TIBCO Patterns servers of the approximate number of records in reclist so that it can choose the best loading strategy. If this value is not given or incorrect all records are still added, the only consequence is that loading performance might not be optimal.

In general larger record batches give better overall throughput. To take advantage of the new multi-threaded faster loads the record batches should be over 40,000 records. For smaller numbers of records, a low overhead single threaded load is performed.

The best strategy for adding records depends on your hardware and the number and nature of the databases involved. If optimal load times are critical consult your TIBCO representative for the best means of optimizing loads for your particular case.

Note however that the large batch loading strategy checks the memory cap only once at the start of the load, thus memory cap limits can be greatly exceeded when using the faster loads for large batches. If you are using memory caps and it is critical to stay within the cap you should force the server to use the small batch load strategy by giving an estimated size of 0 records.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

dbname (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) to which records are added.

reclist (required)

is a list of the records to be added. The record keys of the records in this list must be unique (no duplicates), and must not already exist in database dbname.

If some, but not all, of the record keys already exist in database dbname (error DVK_ERR_RECEXISTS), failure of the whole command can be prevented by setting the parameter LPAR_BOOL_DOMAXWORK (see next item).

Like the `lkt_dbload` command, a file specification might be used in place of an explicitly constructed record list. When using files which do not contain record keys, it is important to set the initial key in the file to a value sufficiently large to not conflict with records already in the database.

params (optional)

This is a list of parameters:

- LPAR_BOOL_DOMAXWORK: If this parameter is false, any error (including the error DVK_ERR_RECEXISTS) causes the complete failure of the command (For example, no records are added to database dbname).

  If this parameter is true, maximum work is done. Records in reclist that can't be loaded due to errors such as duplicate keys or character conversion errors, are quietly ignored and all other records are loaded.

  Default value: false
- LPAR_INT_LOCKKEY allows this update to go through on a database containing a keyed lock if the key value matches the lock value.

  Default value: N/A
- LPAR_INT_DBNUMRECORDS gives an estimate of the number of records in the record list. It is not necessary that this value be exact or even close. This is used by the TIBCO Patterns servers to determine the most efficient means of loading the records.

  Default value: 0 (It assumes a small batch of records). The exception is if loading using an LPAR_LST_REMOTEFILE file specification the assumption is that it is a large batch of records.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the add record operation is to be performed.

Output

dbstats (optional)

is a list of lpars that give statistics on the modified database similar to those returned by lkt_dbload.

If you do not care to have these statistics, just pass lkt_dbrecadd a pointer value of NULL for dbstats.

Error codes and items returned by lkt_dbrecadd

CHARCONV	record that contains improperly encoded characters

DBNOTFOUND	name of nonexistent database
EXPECTDBDESC	item that should have been a database name
EXPECTLIST	item that should have been a list lpar
EXPECTRECORD	item that should have been a record
INTERNAL	item that was being processed at time of error
NODBDESC	(none)
NOMEM	record being processed when memory cap was hit
NORECKEY	record that lacks a record key
NOSRCHTXT	record that lacks searchable text
NOSYSINIT	(none)
NUMFIELDS	record that contains the wrong number of fields
PARAMTYPE	lpar that has invalid ID
RECEXISTS	record with duplicate key value
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONG_INT_TRAN_ID and LPAR_STR_ERRORDETAILS
UPDPARAM	item that should have been an update parameter

Deleting Records From a Database (lkt_dbrecdelete)

The following command deletes one or more records from a database. Statistics are returned for the modified database.

```
dvkerr_t lkt_dbrecdelete(lpar_t host, lpar_t dbname, lpar_t reckeylist,
                          lpar_t params, lpar_t *dbstats );
dvkerr_t lkt_dbrecdeleteT(lpar_t host, lpar_t dbname, lpar_t tran,
                          lpar_t reckeylist, lpar_t params,
                          lpar_t *dbstats );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

dbname (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) from which records are deleted.

reckeylist (required)

A record key list has the following possible formats:

- **Standard:** For a standard table or parent table a record list consists of a generic list (LPAR_LST_GENERIC) containing zero or more LPAR_STR_RECKEY lpars. The key values in the list must be unique and exist in the data base (see the DOMAXWORK flag).
- **Specific Child:** To specify specific records in a child table a generic list (LPAR_LST_GENERIC) containing two sub-lists is passed. The first sub-list must be LPAR_LST_PARENT_KEYS. It must contain a list of LPAR_STR_RECKEY values, each containing the parent key value of a child record. The second sub-list must be LPAR_LST_CHILD_KEYS. It must contain a list of LPAR_STR_RECKEY values, each containing the key value of a child record. Both lists must be the same length, the keys must be in the same order (the nth item must contain the key for the nth record in both lists). The parent key list can contain a null entry, or an entry with a zero length key value, to specify a child record that does not have a parent key. For every parent key, pair of child record must exist in the data base with that parent key and record key (see the DOMAXWORK flag).
- **All Children:** To specify all child records for a particular parent key a generic list (LPAR_LST_GENERIC) containing a single sub-list must be passed. The sub-list must be LPAR_LST_PARENT_KEYS. It must contain a list of LPAR_STR_RECKEY values. The values must be unique. There must be at least one child record in the database with the given parent key

(see the DOMAXWORK flag). All child records with the given parent keys are processed (as modified by any orphan status flags).

params (optional)

is a list of updating parameters:

- LPAR_BOOL_DOMAXWORK: If this parameter is false, any error (including the error DVK_ ERR_RECNOTFOUND) causes the complete failure of the command (i.e., no records are deleted from database dbname).

  If this parameter is true, maximum work is done even if some record keys in reckeylist are not found (error DVK_ERR_RECNOTFOUND) provided that this is the only error that occurs. In such a case the command returns DVK_OK and all of the records that were found are deleted.

  Default value: false
- LPAR_INT_LOCKKEY allows this update to go through on a database containing a keyed lock if the key value matches the lock value.

  Default value: N/A
- LPAR_BOOL_NO_ORPHANS if this flag is given and is set to true, orphan child records are not allowed. For the "**All Children"** record key list format orphan child records with the given parent key are quietly ignored. For the "**Specific Child"** record key list format orphan child records are considered to be a non-existing record, and triggers a record not found error unless the DOMAXWORK flag is set. It is an error if both this value is true and LPAR_BOOL_ORPHANS_ONLY is given and set true.
- LPAR_BOOL_ORPHANS_ONLY if this flag is given and is set to true, only orphan child records are allowed. For the "**All Children"** record key list format non-orphan child records with the given parent key are quietly ignored. For the "**Specific Child"** record key list format non-orphan child records are considered to be a non-existing record, and triggers a record not found error unless the DOMAXWORK flag is set. It is an error if both this value is true, and LPAR_BOOL_NO_ORPHANS is given and set to true.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the delete records operation is to be performed.

Output

dbstats (optional)

is a list of lpars that give statistics on the modified database similar to those returned by `lkt_dbload`.

If you do not care to have these statistics, just pass lkt_dbrecdelete a pointer value of NULL for dbstats.

Error codes and items returned by lkt_dbrecdelete

ARRAYLEN	(none) returned if too many child records for given parent keys
DBNOTFOUND	database name
DUPRECKEYS	list containing duplicate keys
EXPECTDBDESC	item that should have been a database name
EXPECTLIST	item that should have been a list lpar
EXPECTRECKEY	item that should have been a record key
INTERNAL	item being processed when error occurred
NODBDESC	(none)
NORECKEY	key with zero-length string, or (none)
NOSYSINIT	(none)
PARAMCONFLICT	conflicting parameter
PARAMTYPE	lpar that has invalid id
PARAMVAL	Invalid key list
RECNOTFOUND	list of record keys in same format as reckeylist

TBLNOTCHILD	name of table that should be a child table
TRAN_IN_USE	lpar that contains the transaction id
TRAN_UNKNOWN	
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ ERRORDETAILS
UPDPARAM	lpar that should have been an update parameter

Replacing Records in a Database (lkt_dbrecreplace)

The following command replaces one or more records in a database. Statistics are returned for the modified database.

```
dvkerr_t lkt_dbrecreplace(lpar_t host, lpar_t dbname, lpar_t reclist,
                           lpar_t params, lpar_t *dbstats );
dvkerr_t lkt_dbrecreplaceT(lpar_t host, lpar_t dbname, lpar_t tran,
                           lpar_t reclist, lpar_t params,
                           lpar_t *dbstats );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

dbname (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) in which records are replaced.

reclist (required)

is a list of records to be replaced. Each record in reclist replaces, in its totality, the record in dbname having the same record key.

To put it another way, the replace operation is exactly equivalent to a **lkt_dbrecdelete** followed by a lkt_dbrecadd using the same record key. The record keys of the records in reclist must already exist in database dbname.

If some, but not all, of the record keys do not exist in database dbname (error DVK_ERR_ RECNOTFOUND), failure of the whole command can be prevented by setting the parameter LPAR_BOOL_DOMAXWORK (see next item).

Like the `lkt_dbload` command, a file specification might be used in place of an explicitly constructed record list. When using files which do not contain record keys, it is important to set the initial key in the file to a value sufficiently large to not conflict with records already in the database.

params (optional)

is a list of updating parameters:

- LPAR_BOOL_DOMAXWORK: If this parameter is false, any error causes the complete failure of the command (For example, no records are replaced in database dbname).

  If this parameter is true, maximum work is done. Records in reclist that can't be loaded due to errors such as record not found or character conversion errors, are quietly ignored and all other records are loaded.

  Default value: false

- LPAR_INT_LOCKKEY allows this update to go through on a database containing a keyed lock if the key value matches the lock value.

  Default value: N/A

- LPAR_INT_DBNUMRECORDS gives an estimate of the number of records in the record list. It is not necessary that this value be exact or even close. This is used by the TIBCO Patterns server to determine the most efficient means of loading records.

  Default value: 0 (It assumes a small batch of records). The exception is if loading using an LPAR_LST_REMOTEFILE file specification, the assumption is that it is a large batch of records.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the replace records operation is to be performed.

Output

dbstats (optional)

is a list of lpars that give statistics on the modified database similar to those returned by `lkt_dbload`.

If you do not care to have these statistics, just pass `lkt_dbrecreplace` a pointer value of NULL for dbstats.

Error codes and items returned by lkt_dbrecreplace

CHARCONV	record that contains improperly encoded characters
DBNOTFOUND	name of nonexistent database
EXPECTDBDESC	item that should have been a database name
EXPECTLIST	item that should have been a list lpar
EXPECTRECORD	item that should have been a record
INTERNAL	item that was being processed at time of error
NODBDESC	(none)
NOMEM	record being processed when memory cap was hit
NORECKEY	record that lacks a record key
NOSRCHTXT	record that lacks searchable text
NOSYSINIT	(none)
NUMFIELDS	record that contains the wrong number of fields
PARAMTYPE	lpar that has invalid ID
RECNOTFOUND	first record not in the database

TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ERRORDETAILS
UPDPARAM	item that should have been an update parameter

Updating Records in a Database (lkt_dbrecupdate)

The lkt_dbrecupdate command combines the functionality of lkt_dbrecadd, lkt_dbrecreplace and lkt_dbrecdelete. If a simple record is passed to the command, it adds the record if a record with the same key does not exist; if a record with the same key exists, the existing record is replaced with the record that was passed in. If it is passed in a record operation list, it performs the indicated action for the record: adding, deleting or replacing the record.

```
dvkerr_t lkt_dbrecupdate(lpar_t host, lpar_t dbname, lpar_t reclist,
                          lpar_t params, lpar_t *dbstats );
dvkerr_t lkt_dbrecupdateT(lpar_t host, lpar_t dbname, lpar_t tran,
                          lpar_t reclist, lpar_t params,
                          lpar_t *dbstats );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers.

dbname (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) in which records are added or replaced.

reclist (required)

is a generic list of records or record operations. The list might contain a mix of these. If a list item is a record and a record with the same key exists in the table, the record in the table is replaced with the given record. If the list item is a record and a record with the same key does not exist in the table, the given record is added to the table.

A list item might also be a record operation. A record operation is an **LPAR_LST_REC_OP** list. This list must contain exactly two elements (in any order):

- **LPAR_LST_RECORD** An LPAR record.
- **LPAR_STR_REC_OP** This LPAR defines the record operation to be performed. It must be one of the following string values. Any other value results in a DVK_ERR_PARAMVAL error. The values are letter case sensitive.
  - "**ADD**" The record is added to the table. If a record with the same key already exists in the table, the operation fails with a `DVK_ERR_RECEXISTS` error, unless the "Do Max Work" flag is set, in which case the error is quietly ignored.
  - "**DEL**" The record is deleted from the table. If a record with the same key does not exist in the table, the operation fails with a `DVK_ERR_RECNOTFOUND` error, unless the "Do Max Work" flag is set, in which case the error is quietly ignored. For this operation the given record only needs to contain the key data. The field data might be left empty. It is not required it have the correct number of fields.
  - "**RPL**" The record in the table is replaced in its entirety with the given record. If a record with the same key does not exist in the table, the operation fails with a `DVK_ERR_RECNOTFOUND` error, unless the "Do Max Work" flag is set, in which case the error is quietly ignored.
  - "**UPD**" If a record with the same key does not exist in the table, the given record is added; otherwise the record in the table is replaced in its entirety with the given record. This is the same operation that is performed when a record instead of a record operation appears in the list.

Like the lkt_dbload command, a file specification might be used in place of the record or record operation list. The file contains records, not record operations. Therefore, the update operation is applied to all records in the file.

params (optional)

is a list of updating parameters:

- LPAR_BOOL_DOMAXWORK: If this parameter is false, any error causes the complete failure of the command (For example, no records are replaced in database dbname).

If this parameter is true, maximum work is done. Records in reclist that can't be loaded due to errors such as character conversion errors, are quietly ignored and all other records are loaded.

Default value: false

- LPAR_INT_LOCKKEY allows this update to go through on a database containing a keyed lock if the key value matches the lock value.

  Default value: N/A

- LPAR_INT_DBNUMRECORDS gives an estimate of the number of records in the record list. It is not necessary that this value be exact or even close. This is used by the TIBCO Patterns server to determine the most efficient means of loading records.

  Default value: 0 (It assumes a small batch of records). The exception is if loading using an LPAR_LST_REMOTEFILE file specification, the assumption is that it is a large batch of records.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the update records operation is to be performed.

Output

dbstats (optional)

is a list of lpars that give statistics on the modified database similar to those returned by `lkt_dbload`.

If you do not care to have these statistics, just pass `lkt_dbrecreplace` a pointer value of NULL for dbstats.

Error codes and items returned by lkt_dbrecreplace

CHARCONV	record that contains improperly encoded characters
DBLOCKED	descriptor of locked database
EXPECTDBDESC	item that should have been a database name

EXPECTLIST	item that should have been a list lpar
EXPECTRECORD	item that should have been a name
INTERNAL	item that was being processed at time of error
NODBDESC	(none)
NOMEM	record being processed when memory cap was hit
NORECKEY	record that lacks a record key
NOSRCHTXT	record that lacks searchable text
NOSYSINIT	(none)
NUMFIELDS	record that contains the wrong number of fields
PARAMTYPE	lpar that has invalid id
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ERRORDETAILS
UPDPARAM	

Looking Up Records in a Database (lkt_dbrecget)

The following command retrieves (without modifying or deleting) one or more records from a database.

```
dvkerr_t lkt_dbrecget(lpar_t host, lpar_t dbname, lpar_t reckeylist,
                      lpar_t params, lpar_t *reclist );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

dbname (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) from which records are retrieved.

reckeylist (required)

The record key list for the lkt_dbrecget command is the same as described for the lkt_ dbrecdelete command, see reckeylist (required) of lkt_dbrecdelete).

params (optional)

The parameters for the lkt_dbrecget command are the same as described for the lkt_dbrecdelete command, see reckeylist (required) of lkt_dbrecdelete.

Output

reclist (required)

is the list of records requested.

Error codes and items returned by lkt_dbrecget

ARRAYLEN	(none) returned if too many child records found
DBNOTFOUND	database name
DUPRECKEYS	list of duplicate record keys
EXPECTDBDESC	item that should have been a database name

EXPECTLIST	item that should have been a list lpar
EXPECTRECKEY	item that should have been a record key
INTERNAL	item that was being processed at time of error
NODBDESC	(none)
NORECKEY	key with zero-length string, or (none)
NOSYSINIT	(none)
PARAMCONFLICT	parameter in conflict
PARAMTYPE	lpar that has invalid ID
PARAMVAL	Invalid key list
RECNOTFOUND	list of record keys not found
TBLNOTCHILD	Name of table that should be a child table
UPDPARAM	lpar that should have been an update parameter

Scanning Records in a Table

The following command is provided for scanning through records of a table:

- lkt_dbrecnext - scans through records using a cursor position token

Scanning Records Using a Cursor (lkt_dbrecnext)

This function is used to retrieve all records, or a subset of records in a table.

```
int cursor_at_EOT(lpar_t next_pt);
int cursor_skipped_records(lpar_t next_pt);
dvkerr_t lkt_dbrecnext(lpar_t host, lpar_t dbname, lpar_t tran,
                       lpar_t start_pt,lpar_t params,
                       lpar_t *reclst, lpar_t *next_pt);
```

This function is useful whenever the records in a table must be scanned. This might be to perform some operation on each record, or to dump the contents of a table to another storage medium. It uses a cursor object to maintain the current position in the scan so that records can be fetched in a series of calls.

The number of records to be returned with each call can be specified. When scanning through a large number of records larger batch sizes are more efficient. You can specify that an initial number of records are to be skipped. This might be useful when resuming a partially completed scan.

The returned cursor position is valid only as long as no records are added or deleted from the table. The cursor position `lpar` returned in `next_pt` should be treated as an opaque item, the contents are subject to change.

The functions `cursor_at_EOT` and `cursor_skipped_records` are used to test for the end of the table being reached and to fetch the actual number of records that were skipped. The actual number of records skipped can be less than the requested number of records to skip if the table has fewer records than the requested skip count.

The following code example shows how to scan through all the records in a table skipping the first 300 records. A batch size of 100 records is used.

```
lpar_t dbname = lpar_create_str(LPAR_STR_DBDESCRIPTOR, "table-name");
lpar_t params     = lpar_create_lst(LPAR_LST_GENERIC);
int    skip_count = 300 ;
lpar_t start_pt   = LPAR_NULL ;
lpar_t reclist    = LPAR_NULL ;
lpar_t next_pt    = LPAR_NULL ;
dvkerr_t err ;
/* Create the initial params setting */
lpar_append_lst(params, lpar_create_int(LPAR_INT_BATCHSIZE, 100));
lpar_append_lst(params, lpar_create_int(LPAR_INT_SKIPCOUNT, skip_count));
/* Get first batch */
err = lkt_dbrecnext( LPAR_NULL,      /* local process */
        dbname,     /* Our table name*/
        LPAR_NULL,  /* Transactions always NULL */
        start_pt,   /* start_pt is NULL for first call */
        params,     /* parameters with initial skip count */
        &reclist,   /* records returned in here */
        &next_pt);  /* next batch location returned in here */
if (DVKERR(err)) {
    fprintf(stderr, "First call failed: %d (%s)\n",
            DVKERR(err), dvkerr_get_string(err));
    /* clean up and return error */
    dvkerr_clear(err);
    if (start_pt != LPAR_NULL) { lpar_destroy(start_pt) ; }
    if (next_pt != LPAR_NULL)  { lpar_destroy(next_pt) ; }
```

```
        if (params != LPAR_NULL)   { lpar_destroy(params) ; }
        if (reclist != LPAR_NULL)  { lpar_destroy(reclist) ; }
        lpar_destroy(dbname) ;
        /* return error here */
    }
    dvkerr_clear(err);
    /* process initial batch of records */
    process_record_list(reclist);
    /* clean up the records */
    if (reclist != LPAR_NULL) {
        lpar_destroy(reclist) ;
        reclist = LPAR_NULL ;
    }
    /* retrieve more records until cursor hits EOT */
    while ( ! cursor_at_EOT(next_pt) ) {
        /* reset skip count if we skipped records on previous batch */
        if (skip_count > 0) {
            skip_count -= cursor_skipped_records(next_pt);
            /* recreate parameters with new skip count */
            lpar_destroy(params);
            params = lpar_create_lst(LPAR_LST_GENERIC);
            lpar_append_lst(params,
                           lpar_create_int(LPAR_INT_BATCHSIZE,100));
            if (skip_count > 0) {
                lpar_append_list(params,
                                 lpar_create_int(LPAR_INT_SKIPCOUNT,
                               skip_count));
              }
        }
        /* Set start point to previous call's next_pt */
        if (start_pt != LPAR_NULL) {
            /* must destroy old start point! */
            lpar_destroy(start_pt) ;
        }
        start_pt = next_pt ;
        next_pt = LPAR_NULL ;
        /* Following batches use next_pt as starting point */
        err = lkt_dbrecnext(LPAR_NULL,      /* local process */
                dbname,     /* our table name*/
                LPAR_NULL,  /* transactions always NULL */
                start_pt,   /* now the previous next point */
                params,     /* params with skip count updated */
                &reclist,   /* records returned in here */
                &next_pt);  /* next batch location returned here */
        if (DVKERR(err)) {
            fprintf(stderr, "Next call failed: %d (%s)\n",
                    DVKERR(err), dvkerr_get_string(err));
            /* clean up and return error */
            dvkerr_clear(err);
            if (start_pt != LPAR_NULL) { lpar_destroy(start_pt) ; }
            if (next_pt != LPAR_NULL)  { lpar_destroy(next_pt) ; }
            if (params != LPAR_NULL)   { lpar_destroy(params) ; }
            if (reclist != LPAR_NULL)  { lpar_destroy(reclist) ; }
            lpar_destroy(dbname) ;
            /* return error here */
        }
        dvkerr_clear(err);
        /* process the records from this batch */
        process_record_list(reclist);
        /* clean up the processed record list */
        if (reclist != LPAR_NULL) {
            lpar_destroy(reclist);
        }
    }
    /* clean up */
    if (start_pt != LPAR_NULL) { lpar_destroy(start_pt) ; }
    if (next_pt != LPAR_NULL)  { lpar_destroy(next_pt) ; }
    if (params != LPAR_NULL)   { lpar_destroy(params) ; }
    if (reclist != LPAR_NULL)  { lpar_destroy(reclist) ; }
    lpar_destroy(dbname) ;
    /* return success */
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

name (required)

is the name of the database (LPAR_STR_DBDESCRIPTOR) to be scanned.

tran

this must always be LPAR_NULL. It is a place holder for potential future functionality.

start_pt

This is either LPAR_NULL, or the next_pt value returned by a previous call. Remember to clean up this value before resetting it.

params (optional)

This might be LPAR_INT_BATCHSIZE, LPAR_INT_SKIPCOUNT or a generic list containing one or both of these values.

- LPAR_INT_BATCHSIZE
  This specifies the number of records to be returned. The actual number of records returned might be fewer if the end of the table is reached.
  Default value: 1
- LPAR_INT_SKIPCOUNT
  This specifies the number of records to be skipped, starting at start_pt. This effectively sets the starting cursor position to the SKIPCOUNT record position after start_pt.
  Default value: 0.

Output

reclist (required)

is a generic list of records. If the cursor position was at the end of the table this is set to LPAR_ NULL.

next_pt (required)

is an opaque cursor position value. It is set to the next record in the table after the last record in reclist. If there are no more records in the table, the call `cursort_at_EOT(next_pt)` returns true. If params contained LPAR_INT_SKIPCOUNT with a value greater than zero, this also contains the actual number of records skipped. This can be retrieved using the function: `cursor_skipped_ records(next_pt)`.
Error codes and items returned by lkt_dbrecnext

NOSYSINIT	(LPAR_NULL) Devkit was not initialized
PARAMTYPE	(dbname) invalid lpar type for dbname
NODBDESC	(LPAR_NULL) dbname value was null

Checkpointing and Restoring a Database

With TIBCO Patterns server, you can write any or all of its in-memory data objects (database tables, thesauri, Learn Models, and custom character maps) to a permanent data store.

The manual version of this process is called checkpointing. The saved objects can be restored at a later time. This provides a faster means of restoring the state of a server after a planned shutdown than reloading all of the objects from the original data sources.

For a full overview of the checkpoint/restore feature, see the "Checkpoints of In-Memory Tables" and "Durable Data" sections in the *TIBCO® Patterns Concepts Guide.* For more information about enabling and managing this feature, see the section "Running the TIBCO Patterns Server" in the *TIBCO® Patterns Installation guide*.

Note: The –R, -B, and –A options and the sub-section on Checkpoint - Restore Directories.

Checkpoint (lkt_checkpoint)

Use the following functions to checkpoint all, or a selected set of, in-memory objects:

```
dvkerr_t lkt_checkpoint(lpar_t host, lpar_t names, lpar_t *dblist);
dvkerr_t lkt_checkpointT(lpar_t host, lpar_t names, lpar_t tran,
                         lpar_t options, lpar_t *dblist);
```

If a checkpoint of multiple objects is requested, all objects that can be checkpointed are checkpointed. A list of all objects that could not be checkpointed is returned as the error item of the DVK_ERR_PARTIALCHPT error.

If a checkpoint of a database table is requested, all tables in the same joined set of tables are automatically included in the tables checkpointed. The joined set of tables for a child table is its parent table and all child tables of its parent table. The joined set of tables for a parent table is all of its child tables. If one or more tables in the joined set of tables cannot be checkpointed, the entire joined set of tables is not checkpointed. In other words, a joined set of tables is always checkpointed as a unit. If the durable-data feature is enabled, lkt_checkpoint has no effect and returns successfully.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

names (required)

This entry must be one of the following:

- LPAR_NULL use this to specify all database tables.
- LPAR_STR_DBDESCRIPTOR use this to specify a single database table.
- A generic list (LPAR_LST_GENERIC) of LPAR_STR_DBDESCRIPTORs. Use this to specify a set of specific tables. The list might consist of a single entry, in which case the entry is treated as if a single LPAR_STR_DBDESCRIPTOR was passed.
- A multi descriptor list (LPAR_LST_MULTIDESC) of lists of objects. Use this to checkpoint any combination of object types. The LPAR_LST_MULTIDESC must contain exactly four generic lists. The first list must be a list of LPAR_STR_DBDESCRIPTOR values, specifying the database tables to be processed. The second list must be a list of LPAR_STR_CHARMAP values, specifying the custom character maps to be processed. The third list must be a list of LPAR_STR_THESAURUSNAME values, specifying the thesauri to be processed. The fourth list must be a list of LPAR_STR_RLMODELNAME values, specifying the Learn Models to be processed.

The following holds for all four lists in a multi descriptor list:

- If the list value is LPAR_NULL instead of LPAR_LST_GENERIC, all objects of the type (as determined by the position of the list in the multi descriptor list) are processed.
- If the list is an empty generic list (LPAR_LST_GENERIC), no objects of the type (as determined by the position of the list in the multi descriptor list) are processed.
- If the list is a generic list (LPAR_LST_GENERIC) with more than one entry, the object specified by each entry is processed.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the checkpoint operation is to be performed.

options (optional)

for future use. Value should be LPAR_NULL

Output

dblist (optional)

If non-null a list of the objects successfully checkpointed is returned. The list is a generic list (LPAR_LST_GENERIC) of object names. The names can be a mix of LPAR_STR_DBDESCRIPTOR, LPAR_STR_CHARMAP, LPAR_STR_THESAURUSNAME, and LPAR_STR_RLMODELNAME. The entries are not returned in any particular order.

Error codes and items returned by lkt_checkpoint

CHPTFAIL	list of databases that failed to checkpoint
DBNOTFOUND	list of objects not found
DBPARAM	invalid lpar from object name
DUPDBDESCS	duplicate object name
EXPECTDBDESC	item that should have been a n object name
LPAR_NULL	if checkpointing not enabled
INTERNAL	description of error, or LPAR_NULL
NODBDESC	Object name with empty value or LPAR_LST_GENERIC with no entries
NOSYSINIT	(none)
PARAMVAL	The invalid object name
PARMTYPE	The object name that is invalid
PARTIALCHPT	list of databases that failed to checkpoint

TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ERRORDETAILS

Restore (lkt_restore)

Use the following functions to restore all, or a selected set of, in-memory objects:

```
dvkerr_t lkt_restore(lpar_t host, lpar_t names, lpar_t *dblist);
dvkerr_t lkt_restoreT(lpar_t host, lpar_t names, lpar_t tran, lpar_t
*dblist);
```

If a restore of multiple objects is requested all objects that can be restored are restored, even if the restore of one or more other objects fail. The exception is if there is a failure because an incomplete join set was specified. In this case the entire request fails. If some objects were restored successfully and others were not, DVK_ERR_PARTIALRESTORE is returned and the error item is a list of objects that failed to be restored.

When restoring a database table the entire joined set of tables must be specified. Unlike the `lkt_checkpoint` command, the restore command does not automatically fill out the join set. A join set is a parent table and all child tables of that parent table.

Note that the restore command locks the checkpoint files of all objects to be restored. This might block other commands, such as `lkt_delete_thesaurus`, `lkt_dbmove`, `lkt_checkpoint`, any command that creates, deletes or renames one of the objects to be restored. As a restore can take a significant amount of time, this can seriously impact performance. If the durable-data feature is enabled, `lkt_restore` has no effect and returns successfully.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

names (required)

The name parameter for restore is the same as that for the checkpoint command, see names (required). name (required).

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the restore operation is to be performed.

Output

dblist (optional)

If non-null a list of the objects successfully restored is returned. The list is a generic list (LPAR_LST_GENERIC) of object names. The entries can be a mix of LPAR_STR_DBDESCRIPTOR, LPAR_STR_CHARMAP, LPAR_STR_THESAURUSNAME and LPAR_STR_RLMODELNAME. The entries are not returned in any particular order.

Error codes and items returned by lkt_restore

DBNOTFOUND	list of objects not found (i.e. not previously checkpointed)
DBPARAM	invalid lpar from object list
DUPDBDESCS	duplicate object name
EXPECTDBDESC	
FEATURESET	Checkpoint/restore is not enabled on the server.
JOINSET	The table name list with incomplete joined set of tables
LPAR_NULL	if checkpointing not enabled
INTERNAL	description of error, or LPAR_NULL
NODBDESC	LPAR_STR_DBDESCRIPTR with empty value, or LPAR_LST_GENERIC with no entries

NOSYSINIT	(none)
PARMTYPE	The object name that is invalid
PARAMVAL	The invalid object name
PARTIALRESTORE	list of databases that failed to restore
RESTOREFAIL	list of databases that failed to restore 2.6 Server control
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ ERRORDETAILS

Checking Server Status

These functions are used to query or control the state of the server process itself. When sent to a gateway these operations act only on the gateway itself, they don't change or reflect the status of any nodes controlled by the gateway. For more information on gateways and nodes see Federated Tables and Gateways.

Checking Server Status (lkt_svrnoop)

The server noop command literally does nothing. If the server is alive, it always returns success. This can be used to test connectivity.

```
dvkerr_t lkt_svrnoop( lpar_t host);
```

Input

host (required)

For more information, see Communicating with TIBCO Patterns Servers.
Error codes and items returned by lkt_svrnoop

IPCERR	(none)
IPCTIMEOUT	(none)
IPCEOF	(none)
HANDSHAKE	(none)

Getting TIBCO Patterns Servers Version Information (lkt_svrversion)

The following command is used to get the server version and configuration information:

```
dvkerr_t lkt_svrversion( lpar_t host, lpar_t *version );
```

`lkt_svrversion` returns information that can identify the particular version of the server that is running and information on its current configuration.

Input

host (required)

For more information, see Communicating with TIBCO Patterns Servers.

Output

version (required)

is a list containing the following lpars:

- SVR_LPAR_BLK_DEVKITINFO this holds the version information on the DevKit used to build the server. It is a free form string containing build and feature selection information.
- SVR_LPAR_BLK_SERVERINFO this holds version information on the server. It is structured as an XML snippet with the following form:

```
<server_info>
     <product_name>official product name</product_name>
     <version>full version number</version>
     <build_date>time and date built</build_date>
     <who_built>ID of who ran build</who_built>
</server_info>
```

  Where *official product name* is the official name for this version of the server, for example, TIBCO Patterns, *full version number* is the 3 place version number, for example, 5.3.0, *time and date built* is what it says, a date stamp for when the build for this server was performed, and *ID of who ran build* is also what is says, the ID of who ran the build.
- LPAR_INT_PLAINTEXT_PORT The port the server is listening on for plaintext communications. If plaintext communications are disabled, -1 is reported.
- LPAR_INT_ENCRYPTED_PORT The port the server is listening on for encrypted communications. If encrypted communications are disabled, -1 is reported.
- LPAR_BOOL_DBGIPFILTER True if the GIP prefilter is enabled.
- LPAR_BOOL_DBSORTFILTER True if the SORT prefilter is enabled.
- LPAR_BOOL_DBPSIFILTER True if the PSI prefilter is enabled.
- LPAR_INT_MAXTHREADS Maximum number of command processing threads that runs in parallel.
- LPAR_STR_CHECKPOINTDIR The path name of the directory used for checkpoint/restore. If checkpoint/restore is not enabled, this is the string “(not-enabled)”.
- LPAR_BOOL_AUTORESTORE True if the server was started with the auto-restore flag.
- LPAR_STR_GATEWAYCFG The path name of the cluster configuration file for a gateway process. If this is not a gateway server, this is the string "(not a gateway)".
- LPAR_INT_MAXMEMKB The memory cap in K-Bytes. This is zero if no memory cap was set.

- LPAR_BOOL_DBRLINKFILTER This is true if the TIBCO Patterns Machine Learning Platform is enabled on this server.
- LPAR_INT_TRAN_IDLETIME the maximum transaction idle time out value, in seconds.
- LPAR_STR_TRAN_CLOSE_ACT is the action taken against transactions that exceed the time out limit. One of: “a” abort it, “e” abort if it has errors, commit otherwise, “c” forcibly commit the transaction, even if it has errors, “n” do nothing, “ “ no change.
- LPAR_BOOL_DBGIPGPU is true if GIP GPU acceleration is enabled, false otherwise.
- LPAR_LST_GPU_DEVICE_INFOS is output if LPAR_BOOL_DBGIPGPU is true and there is at least one known GPU device. This list contains one or more LPAR_LST_GPU_DEVICE_INFO lists. Each of these contains the following:
  - LPAR_INT_GPU_DEVICE_ID the numeric id for the GPU device.
  - LPAR_STR_GPU_DEVICE_NAME the name for the GPU device.
  - LPAR_INT_GPU_DEVICE_USAGE a set of bit flags indicating the state of the device. The flags used are: (1) This is a recommended device, (2) This device does not meet minimum specifications, (4) this is a default device, (8) this device is used only on explicit request.

For more information about the configuration settings, see the TIBCO® Patterns Installation guide.

Error codes and items returned by lkt_svrversion

IPCERR	(none)
IPCTIMEOUT	(none)
IPCEOF	(none)
HANDSHAKE	(none)

Getting command performance statistics (lkt_ svrcmdstats)

This command is used to retrieve summary reports on performance statistics for one or more commands. It is also used to reset the statistics back to a zero state, that is, delete all existing statistics history for one or more commands.

```
dvkerr_t lkt_svrcmdstats( lpar_t host, lpar_t iargs,
                          lpar_t *cmdstats );
```

Summary reports might be requested for a particular time interval by specifying an end time (most recent) and a duration. However the actual time covered is based on when previous reports were generated. Summary reports are created by merging previous summary reports. Currently the server maintains a history of all summary reports requested over the last 17 minutes plus a final summary report that covers the rest of the time back to the start of the server or the last reset issued on the command. When you request a report starting from "now" a new summary report is generated and stored into this "history" of reports. This new report covers the time from "now" back to when the last report was issued. The returned report is then generated by merging those "history" reports that most closely cover the requested time period as defined by the duration. This has a number of implications:

- The first report generated is always for the entire time period since the server was started.
- Durations greater than 17 minutes covers the last 17 minutes or the entire time period since the server was started, or the last reset.
- Actual durations for summary reports are based on when previous summary reports were requested.

In practice if reports over specific time periods are wanted summary reports should be requested at regular intervals representing the shortest time period. For example if you want reports for the last 1 minute, 5 minutes and 15 minutes you should request a report every 1 minute. The system has a history of reports at one minute intervals available and is able to generate summary reports for 5 minute and 15 minute intervals.

Note that new summary reports are generated into the "history" only if you request an end time of "now" for your report (this is the default). This essentially creates a time tic-mark. Requesting a specific end time does not create a new tic-mark, it uses only existing "history" reports to generate the output report. So in the example used above where reports for the last 1, 5 and 15 minutes are desired, the recommended procedure is to first request a report from "now" with a duration of 1 minute. Then for the 5 and 15 minute reports the end time should be taken from the returned end time of the 1 minute report. In this way you create a tic-mark every one minute, which can be used to generate the 5 and 15 minute reports over the desired time period. The following example shows how this could be done:

```
lpar_t iargs ;       /* arguments to define report */
lpar_t tlpar ;       /* temp lpar */
lpar_t host;  /*  identify server */
lpar_t cmdstats ;    /* the report is placed in here */
double end_time ;    /* end time for 5 & 15 minute report */
/* create the one minute report */
```

```
iargs = lpar_create_list(LPAR_LST_GENERIC) ;
/* for this exampled we retrieve search command stats */
lpar_append_list(iargs,
                 lpar_create_str(LPAR_STR_CMDNAME,DVK_CMD_SEARCH));
/* duration is 1 minute - 60 seconds */
lpar_append_list(iargs, lpar_create_dbl(LPAR_DBL_SUM_END, 60.0));
/* end time of now is the default */
/* get the report */
dvkerr = lkt_svrcmdstats(host, iargs, &cmdstats);
if (DVKERR(dvkerr)) {
        /* handle error here */
}
/* output report */
...
/* save our end time */
tlpar = lpar_find_lst_lpar_recursive(cmdstats, LPAR_DBL_SUM_END);
if (tlpar == LPAR_NULL) {
       /* this would be an error, end time should be returned */
       ....
}
end_time = lpar_get_dbl(tlpar) ;
/* clean up */
lpar_destroy(iargs) ;
lpar_destroy(cmdstats) ;
/* get 5 minute report */
iargs = lpar_create_list(LPAR_LST_GENERIC) ;
/* set command name */
lpar_append_list(iargs,
                 lpar_create_str(LPAR_STR_CMDNAME,DVK_CMD_SEARCH));
/* duration is 5 minutes - 5 * 60 seconds */
lpar_append_list(iargs, lpar_create_dbl(LPAR_DBL_SUM_DURATION, 5 * 60.0));
/* end time is end time of previous report, i.e. "now" */
lpar_append_list(iargs, lpar_create_dbl(LPAR_DBL_SUM_END, end_time));
/* get the report */
dvkerr = lkt_svrcmdstats(host, iargs, &cmdstats);
if (DVKERR(dvkerr)) {
        /* handle error here */
}
/* output report */
...
/* clean up */
lpar_destroy(iargs) ;
lpar_destroy(cmdstats) ;
/* 15 minute report, similar to the 5 minute report */
...
```

An alternative approach is to have a process or thread that requests a report starting at "now" every *N* seconds, and records the end time of the latest such report. Reports for any desired duration (up to 17 minutes) can then be generated with a resolution of *N* seconds by requesting the desired duration using the end time from the latest tic-mark request.

Input

host (required)

For more information, see Communicating with TIBCO Patterns Servers.

iargs (required)

is a generic list of the following parameters.

- LPAR_STR_CMDNAME the name of a command for which statistics should be returned. This must appear at least once, and might appear multiple times. The command names are defined in the `devkit.h` include file. There is a special command name: DVK_CMD_ALL that can be used to indicate that statistics on all commands should be returned.
- LPAR_DBL_SUM_DURATION sets the time span to be covered by this report. The time is given in seconds. If this is not given, or a value less than 0.0 is given the report covers the entire time span from the "end time" back to the start of the server or the last time the statistics for this command were last reset.

  The actual duration covered by the report depends on when previous reports were generated as described above.

  This parameter might be given at most once. It is not valid to give both this parameter and LKT_DBL_SUM_START.
- LPAR_DBL_SUM_END sets the ending time of the report (most recent time included in the report). If this is not given or is less than 0.0 the end time is considered to be "now", the present moment, and this request creates a new tic-mark (see discussion above).

  The end time is given in milliseconds from an arbitrary epoch point. Therefore the only valid way to get a specific end time is from the returned end time of a previous report.

  This parameter might be given at most once. It is not valid to give both this parameter and LKT_DBL_SUM_START.
- LPAR_DBL_SUM_START If specified this is a reset request and not a summary report request. All statistics for times at or before the given start time is deleted for the indicated commands. If the indicated time is less than zero all statistics are deleted (i.e. revert to zero).

The start time is given in milliseconds from an arbitrary epoch point. Therefore the only valid way to get a specific start time is from the returned start time of a previous report.

This parameter might be given at most once. It is not valid to give this parameter in combination with either LKT_DBL_SUM_END or LKT_DBL_SUM_DURATION.

Output

cmdstats (required)

if a reset was requested (LKT_DBL_SUM_START was given) this is set to LPAR_NULL. Otherwise a list of lists is returned, one list for each command requested. Each list contains the following lpars. Note that all times given are elapsed time in milliseconds, not CPU time.

- LPAR_STR_CMDNAME The name of the command this report is for.
- LPAR_DBL_SUM_START The actual start time for the report. This is given in milliseconds from an arbitrary epoch point.
- LPAR_DBL_SUM_END The actual end time for the report. This is given in milliseconds from an arbitrary epoch point. Note that the actual duration of the report in seconds can be calculated as:

  "(LKT_DBL_SUM_END - LKT_DBL_SUM_START) / 1000.0".
- LPAR_INT_NUMCOMMANDS gives the number of commands that were completed in the time covered by this report.
- LPAR_DBL_SUM_MINPROC gives the minimum time in milliseconds taken to process one of the commands completed during the report time period. Processing time does not include time spent reading or sending data nor time spent waiting for a command processing thread to become available.
- LPAR_DBL_SUM_MAXPROC gives the maximum time in milliseconds taken to process one of the commands completed during the report time period. Processing time does not include time spent reading or sending data nor time spent waiting for a command processing thread to become available.
- LPAR_DBL_SUM_TOTPROC gives the sum of the processing times in milliseconds for all of the commands that were completed during the report time period. Processing time does not include time spent reading or sending data nor time spent waiting for a command processing thread to become available.
- LPAR_DBL_SUM_MINOVERHEAD gives the minimum time in milliseconds spent on overhead for one of the commands completed during the report time period. Overhead

time includes time spent reading or sending data and time spent waiting for a command processing thread to become available.

- LPAR_DBL_SUM_MAXOVERHEAD gives the maximum time in milliseconds spent on overhead for one of the commands completed during the report time period. Overhead time includes time spent reading or sending data and time spent waiting for a command processing thread to become available.
- LPAR_DBL_SUM_TOTOVERHEAD gives the sum of the overhead times in milliseconds for all of the commands that were completed during the report time period. Overhead time includes time spent reading or sending data and time spent waiting for a command processing thread to become available.
- LPAR_DBL_SUM_MINFULL gives the minimum time in milliseconds taken to handle one of the commands completed during the report time period. Full time includes both processing and overhead time.
- LPAR_DBL_SUM_MAXFULL gives the maximum time in milliseconds taken to handle one of the commands completed during the report time period. Full time includes both processing and overhead time.
- LPAR_DBL_SUM_TOTFULL gives the sum of the full times in milliseconds for all of the commands that were completed during the report time period. Full time includes both processing and overhead time.
- LPAR_DBL_SUM_MAXTHRU gives the highest average throughput seen, in commands per second, during any one minute time period covered by this report. Note that the one minute time period is subject to the same duration adjustments as for the reports as a whole. Thus if reports are requested only every 5 minutes this is actually over a five minute time period.

Error codes and items returned by lkt_svrcmdstats

EXPECTLIST	iargs (iargs was not a list)
PARAMVAL	LPAR_STR_CMDNAME (unknown command name given)
PARAMCONFLICT	Value in conflict. (start time with end time or duration.)
DBPARAM	The unrecognized parameter.
INTERNAL	None. (unexpected internal error)

Server Shutdown

The following function shuts down the server.

```
dvkerr_t lkt_svrshutdown( lpar_t host );
```

It is normally not necessary to use this command. A client should not shut down the server as a normal part of disconnecting.

Input

host (required)

For more information, see Communicating with TIBCO Patterns Servers.
Error codes and items returned by lkt_svrshutdownex

IPCERR	(none)
IPCTIMEOUT	(none)
IPCEOF	(none)
HANDSHAKE	(none)
LOGFILE	(none)
PARAMVAL	Parameter that should have been a SVR_LPAR_BLK_LOGFORMAT

Server Logging On

The following function turns on server query logging and/or alters the log format being used.

```
dvkerr_t lkt_svrlogon( lpar_t hostc, lpar_t format );
```

In order for this command to have any effect, the server must have been started with the –l command line option to specify the name of the query log file. For further details, see the TIBCO® Patterns Installation guide.

The format lpar, if not LPAR_NULL, specifies the new log format to use. The format string is copied literally to the log file after the following conversions:

%c	(where c is any character) is substituted as in the ANSI C library function strftime.
$c	
$d	database(s) accessed by command
$g	GIP prefilter state
$i	IP address of connection
$n	number of matching records returned
$P	predicate
$q	query
$s	numeric status code returned by server
$S	string representation of status returned by server
$t	CPU time needed by command
$$	literal $

The default format is:

$c %d/%b/%Y-%H:%M:%S $i $s $t $d $q $n $p $g\n

Input

host (required)

For more information, see Communicating with TIBCO Patterns Servers.

format (optional)

is the log file format (SVR_LPAR_BLK_LOGFORMAT).
Error codes and items returned by lkt_svrlogon

IPCERR	(none)
IPCTIMEOUT	(none)
IPCEOF	(none)
HANDSHAKE	(none)
LOGFILE	(none)
PARAMVAL	Parameter that should have been a SVR_LPAR_BLK_LOGFORMAT.

Server Logging Off

This function turns off server query logging.

```
dvkerr_t lkt_svrlogoff( lpar_t host );
```

Input

host (required)

For more information, see Communicating with TIBCO Patterns Servers.
Error codes and items returned by lkt_svrlogoff

IPCERR	(none)
IPCTIMEOUT	(none)
IPCEOF	(none)
HANDSHAKE	(none)

Scoring a Set of Records (lkt_scorerecords)

This command is similar to the lkt_dbsearch command except that it applies the query to a given set of records instead of to an in-memory table of records.

```
dvkerr_t lkt_scorerecords( lpar_t host, lpar_t rec_def,
                           lpar_t querypar, lpar_t srchpars,
                           lpar_t rec_list, lpar_t *stats,
                           lpar_t *matches, lpar_t *minfo );
```

This command runs a query against a set of records, returning the query results for each record in the given set. Unlike the lkt_dbsearch command it does not find and return the best matches. Instead it scores every record given returning the query match information for each record. The key differences between lkt_scorerecords and lkt_dbsearch are:

- Instead of passing the name of a table or set of tables to be searched the definition of the record layout and a set of records is passed in.
- Joined searches are not supported by the lkt_scorerecords command. As no tables are involved the concept of joined tables does not apply.
- All records passed in are always scored and returned. lkt_scorerecords is explicitly a request to calculate and return the query score for a set of records. It is NOT a request to find records. Therefore all records are always returned, even if the record has been assigned a "reject" score.
- Filtering predicates do not apply. There is no way to specify a filtering predicate.
- prefilters are never used, so all prefilter options are ignored.
- Cutoffs do not apply. So all cutoff options are ignored.
- A query that has no query data is not considered an error when used with the lkt_scorerecords command. The lkt_dbsearch command must have some query data to provide a basis for selecting a set records. The lkt_scorerecords command does not select records, so this restriction does not apply.

Generally, the lkt_scorerecords command ignores options that do not apply rather than reject them. This helps the exact same query and search parameters to be used with lkt_dbsearch and lkt_scorerecords.

Input

host (optional)

is used to identify the server.

For more information, see Communicating with TIBCO Patterns Servers

rec_def (required)

is used to define the field names and field types of the records to be searched. This parameter accepts the same lpar structure as the dbpars parameter of the lkt_dbload command except the LPAR_STRARR_FIELDNAMES lpar is required in all cases. All values except LPAR_STRARR_FIELDNAMES, LPAR_STRARR_CHARMAPS and LPAR_INTARR_FIELDTYPES are ignored. As with the lkt_dbload command the character maps default to DVK_CMAP_STDNAME and the field types default to LKT_FLD_TYPE_SRCHTEXT.

query (required)

This is the query to be applied. It is identical to the query parameter of the lkt_dbsearch command. Details on query construction are covered in Query Construction.

srchpars (optional)

This is the same as the srchpars parameter to the lkt_dbsearch command. The following parameter values have no effect: LPAR_INT_STARTMATCH, LPAR_LST_CUTOFF (and all associated entries), LPAR_BOOL_DOMAXWORK, LPAR_BOOL_GIPSEARCH, LPAR_BOOL_SORTSEARCH, LPAR_BOOL_PSISEARCH, LPAR_BLKARR_SORTLOOKUPFIELDS, LPAR_BLKARR_PSILOOKUPFIELDS, LPAR_INT_FETCH_SIZE, LPAR_INT_PC_SIZE, LPAR_INT_PSI_DENSITY.

rec_list (required)

This is a generic list of records to be scored. All records must have the same number of fields as defined by LPAR_STRARR_FIELDNAMES in the rec_def parameter.

Output

stats (required)

is identical to the stats output parameter of the lkt_dbsearch command. The filter times and filter sizes output lists only the last, filter stage 2, filter step as the other filter steps are not run for the lkt_scorerecords command.

matches (required)

is identical to the matches output parameter of the lkt_dbsearch command.

minfo (required)

is identical to the minfo output parameter of the lkt_dbsearch command, except that it is required for the lkt_scorerecords command. The entire purpose of this command is to return this scoring information, so it makes no sense to run the command without this output parameter.

Error codes and items returned by lkt_scorerecords

ARRAYLEN	array lpar with wrong number of values
CHARCONV	Record containing invalid UTF-8 character
DBPARAM	Record definition item that is not valid
EXPECTDBDESC	
EXPECTLIST	item that should have been a list lpar
EXPECTQUERY	item that should have been a search query
FEATURESET	item specifying an unsupported feature
INTERNAL	item being processed at the time of the error, or a detailed description of the error
LOOKUP	(none) error in scoring operation
MODEL_UNSUPPORTED	(none) issued when the Learn Model version is not

	supported
NOCHARMAP	specified character map doesn't exist
NOQUERY	(none) A query parameter was missing
NORLINKMODEL	Learn Model name - the model is not loaded, query parameter - no Learn Model name is specified
NOSYSINIT	(none) DevKit is not initialized
NOWGTFLD	query expression missing the LPAR_STR_WGTFLD_NAME
NSFIELD	field selection lpar with name of non-existant field
NUMFEATURESMISMATCH	Learn expression lpar with wrong number of entries for the specified model.
NUMFIELDS	Record containing incorrect number of fields
PARAMCONFLICT	lpar that contains a parameter conflicting with an earlier parameter.
PARAMMISSING	the list missing a required item
PARAMVAL	lpar that contains an illegal value
PREDSTRING	LPAR_STR_ERRORDETAILS describing predicate syntax error
QUERYEXPR	lpar that contains the invalid query specification
SRCHPARAM	item that should have been a search parameter
THESNOTFOUND	name of thesaurus not found
UNKFIELD	A field name lpar containing a field not listed in the rec_def LPAR_STRARR_FIELDNAMES list.

Query Construction

With the advent of version 4.1 of the TIBCO Patterns servers, query construction has simplified dramatically. This section outlines the type of construction that is now suggested and is supported in the future. A query consists of a tree of "score generators" and "score combiners." A score generator takes data from a table and computes a score (between 0.0 and 1.0). The currently available score generators are Simple, Cognate, Attribute, and Predicate. There is a special case of the simple score generator, called a custom scorer, which is used to compare two dates. A score combiner is used to combine previously computed scores, the simplest case of which is an arithmetic mean. The available score combiners are AND, OR, RECWGT, and Record Linkage (RLINK).

The user must construct a query tree such that all the leaves in the tree are score generators and all the branches of the tree are score combiners. In practice most score trees are just one level deep - with a set of score generators combined using a single score combiner.

For a concrete example, consider a database with the following fields:

- First Name
- Last Name
- Social Security Number
- House Number
- Street Name
- City

and a search application which provides the user three text entry boxes labeled Name, Address, and SSN.

```
lpar_t ANDquery,ANDqueryargs,simpquery,sq,sf;
unsigned char *name,*address,*ssn;
unsigned char *name_fields[] =    { "First Name",
                                    "Last Name" };
unsigned char *ssn_fields[] =     { "Social Security Number" };
unsigned char *address_fields[] = { "House Number",
                                    "Street Name",
                                    "City" };
name    = get_name_box();
address = get_address_box();
ssn     = get_ssn_box();
ANDqueryargs=lpar_create_lst(LPAR_LST_QEXPR_ARGS);
/* Name simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
```

```
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,name,strlen(name));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,name_fields,2);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
/* SSN simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,ssn,strlen(ssn));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,ssn_fields,1);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
/* Address simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
ANDquery=lpar_create_lst(LPAR_LST_QEXPR);
lpar_append_lst(ANDquery,lpar_create_int(LPAR_INT_QEXPR_TYPE,LKT_QEXPR_AND));
lpar_append_lst(ANDquery,ANDqueryargs);
```

Here an AND combiner containing three simple queries has been built. Each querylet contains a SEARCHQUERY text block and a field selection array. Notice that there does not need to be a one to one mapping between record fields and querylets. When one querylet lists multiple fields in its field set, those fields are concatenated and the querylet text is compared against this concatenated field string.

We now outline all the different score generators and combiners and the parameters they take before giving a more complicated example.

Simple Query

A simple query is a comparison of an input query string and one or more fields in the table. The TIBCO Patterns servers' intelligent string matching algorithm is used to score the comparison and tolerate errors.

This is the most common score generator and the TIBCO Patterns servers raison d'etre. A simple query is specified as a LPAR_LST_SIMPLEQUERY which contains the following parameters.

- LPAR_BLK_SEARCHQUERY specifies the query string to be used for this comparison.

- LPAR_INTARR_SELECTFLDS selects the record fields against which this querylet is compared. Fields are specified by number with this option. Note that it is impossible to reference a Variable Attribute value using field numbers.
- LPAR_STRARR_FIELDNAMES selects the record fields against which this querylet is compared. Fields are specified by name with this option. The field name might include a Variable Attribute qualifier (see Variable Attributes - Usage Details for details).
- LPAR_DBLARR_QFIELDWEIGHTS (optional) specifies the weight for matched text against each field in the LPAR_INTARR_SELECTFLDS or LPAR_STRARR_FIELDNAMES array. The maximum weight is 1.0, with values less than 1.0 representing penalized matches.
- LPAR_DBL_INVALID_DATA_SCORE (optional) is the score to return for a record if either a query or a record field has invalid data. This supersedes on a query by query basis the value set in the search parameters. There is no such thing as invalid text data so this pertains to custom date matching and predicate scorers only.
- LPAR_DBL_EMPTY_DATA_SCORE (optional) is the score to return for a record if either a query or a record field set is empty. This supersedes on a query by query basis the value set in the search parameters. If a Variable Attribute doesn't exist for a record it is considered to be empty.
- LPAR_BOOL_MATCH_EMPTY controls behavior when a query string and the data it is being matched to are both empty. If true, a 1.0 score is generated. If false, the empty-score is used.

  Default: false
- LPAR_INT_SORTSCORE (optional) sets the score type that is considered to be the match score for this query. This supersedes on a query by query basis the value set in the search parameters.
- LPAR_STR_THESAURUSNAME (optional) is the thesaurus to use for just this querylet. If LPAR_STR_THESAURUSNAME is specified LPAR_LST_THESAURUS might not be.
- LPAR_LST_THESAURUS (optional) defines a thesaurus to use for just this querylet. If LPAR_LST_THESAURUS is specified LPAR_STR_THESAURUSNAME might not be. The list must consist of two lpars, they being the thesaurus_options and thesaurus_data arguments to the `lkt_create_thesaurus` command. The thesaurus defined by this argument is created and exists only for the duration of this query and is local to this query. See Thesaurus Matching for more information about defining a thesaurus and TIBCO Patterns servers' thesaurus support in general and Ephemeral Thesauri for more information on these ephemeral thesauri in particular.
- LPAR_DBL_THESAURUSWEIGHT (optional) is a penalty factor that is applied to every thesaurus substitution. This penalty factor is applied for all thesaurus types, but only if there was a thesaurus match between query and record of two different terms (for example, a match of Peggy to Margaret but it does not apply to a match of Peggy to

Peggy). If a combined thesaurus is being used the penalty factor defined in the thesaurus for the class is multiplied by the factor provided here to obtain the final penalty factor.

- LPAR_STR_CHARMAP (optional) is the name of the character map to be applied to the query values. By default all fields referenced in a simple query must have the same character map. If a query is to be performed across fields that use two or more different character maps then this option must be supplied to specify which map is to be used.

Note: Setting LPAR_STR_CHARMAP option might result in invalid or incorrect results. Consult your TIBCO representative before using this option.

- LPAR_LST_QOPTS (optional) is a list of detailed tuning parameters which control how the TIBCO Patterns server's scores matches. It is extremely rare that any of these values need to be changed and should be changed only in consultation with your TIBCO representative.
- LPAR_INT_CUSTOMSCORE (optional) specifies which custom score function to use for this comparison. The currently defined values for this parameter are:
  - LKT_CS_NONE specifies that no custom scoring is to be applied. This is the default.
  - LKT_CS_DATE_DEDUP specifies that a special date comparison scoring method should be used. This is tuned for detecting likely transformations or incomplete entries for dates. If this custom scoring method is specified the field list for this query must consist of exactly one field and that field must have a field type of LKT_FLD_TYPE_DATE. See The TIBCO Patterns Table for an explanation of field types.
- LPAR_STR_QLETGROUP (optional) assigns a group name to this querylet. Group names are used by the GIP prefilter to improve the selection of child records in joined searches. See the "Matching Compound Records and Querylet Grouping” section in the *TIBCO® Patterns Concepts Guide* for more information on querylet grouping and when to use it.

Cognate Query

A cognate query is a more complex version of a simple query. It uses the same string comparison algorithm but performs the match with multiple query strings. For instance, a first and last name can be searched against the first and last name fields of the database.

This aids in finding transposed fields (where a first name occurs in the last name field or vice versa).

The difference between two or more simple queries combined with an AND and a cognate query is most clear when there are blocks of repeated text in the query. Assume that the query is for the name John, Johnson, and a record contains the name Frank, Johnson.

With two simple queries, both matched against first and last name fields, first we would consider matching John against Frank, Johnson and find a reasonable match since Johnson resembles John. Then we would consider matching Johnson against Frank, Johnson and find a perfect match. But when computed as a cognate query, the first name John is permitted to match against Johnson since this section of the record is already claimed by the better match with Johnson. In the case of the combined simple queries the last name field of the record is matched twice, creating a higher record score, but one that is not valid in this context. Cognate queries are used when you are matching a set of fields that comprise a single value against a similar set of fields, but have no assurance that data has been placed in the correct field.

This is called a cognate query because every query string has a unique cognate or corresponding field. Thus there is a one to one correspondence between query strings and the selected field set for the query. This implies the length of the LPAR_BLKARR_SEARCHQUERY array must always equal the length of the LPAR_STRARR_FIELDNAMES array. The LPAR_DBLARR_QFIELDWEIGHTS array must also be the same length. Query strings are preferentially matched against their cognate field but might be matched against any field in the field set. LPAR_DBL_NONCOG_WGT is a weighting factor or penalty applied when a value from a query string is matched to a non-cognate field. It must be a value between 0.0 and 1.0 inclusive.

Thus in the example of the first and last name match above if a LPAR_DBL_NONCOG_WGT value of 0.8 was given and a query of John, Smith was matched against a record of Smitty, Johnson there would be a nearly perfect (normal score) match of John to Johnson and Smith to Smitty, but because the matches are against non-cognate fields (first to last, last to first) the score would be penalized by a factor of 0.8.

In some cases a field in a cognate match is empty. The most common example is the middle name when matching first, middle, and last name fields in English-speaking countries where middle names are often not used. When matching one name against another, if one has a middle name and the other does not, the match should not be penalized for the unmatched middle name. Because the cognate query allows for the misfielding of data, it is not valid to look only at the middle name field and adjust scores for not matching that field. The key thing to focus on is a difference in the number of populated fields. When there is a difference in the number of populated fields, the LPAR_DBL_COGNATE_EMPTY_PENALTY LPAR adjusts the penalty applied to the unmatched data in the “extra” populated fields. The *penalty* is multiplied by the LPAR_DBL_COGNATE_EMPTY_PENALTY value. A value of 1.0 applies the full penalty for unmatched data in the extra populated fields; a value of 0.0 applies no penalty for the unmatched data, and a value of 0.1 applies only one tenth as much of a penalty. For examples and more detailed information, see the *TIBCO Patterns Concepts Guide.*

A cognate query is specified as a LPAR_LST_COGQUERY which contains the following parameters:

- LPAR_BLKARR_SEARCHQUERY specifies the query strings to be used for this comparison.
- LPAR_INTARR_SELECTFLDS selects the record fields against which this querylet is compared. Fields are specified by number with this option. Note that it is impossible to reference a Variable Attribute value using field numbers.

- LPAR_STRARR_FIELDNAMES selects the record fields against which this querylet is compared. Fields are specified by name with this option. The field name might include a Variable Attribute qualifier (see Variable Attributes - Usage Details for details).
- LPAR_DBLARR_QFIELDWEIGHTS (optional) specifies the weight for matched text against each field in the LPAR_INTARR_SELECTFLDS or LPAR_STRARR_FIELDNAMES array. The maximum weight is 1.0, with values less than 1.0 representing penalized matches.
- LPAR_DBL_INVALID_DATA_SCORE (optional) is the score to return for a record if either a query or a record field has invalid data. This supersedes on a query by query basis the value set in the search parameters. There is no such thing as invalid text data so this option is not currently relevant to cognate queries and appears only to maintain consistency with other query forms.
- LPAR_DBL_EMPTY_DATA_SCORE (optional) is the score to return for a record if either a query or a record field set is empty. This supersedes on a query by query basis the value set in the search parameters.
- LPAR_BOOL_MATCH_EMPTY controls behavior when a query string and the data it is being matched to are both empty. If true, a 1.0 score is generated. If false, the empty-score is used.

  Default: false
- LPAR_INT_SORTSCORE (optional) sets the score type that is considered to be the match score for this query. This supersedes on a query by query basis the value set in the search parameters.
- LPAR_STR_THESAURUSNAME (optional) is the thesaurus to use for just this querylet. If LPAR_STR_THESAURUSNAME is specified LPAR_LST_THESAURUS might not be.
- LPAR_LST_THESAURUS (optional) defines a thesaurus to use for just this querylet. If LPAR_LST_THESAURUS is specified LPAR_STR_THESAURUSNAME might not be. The list must consist of two lpars, they being the thesaurus_options and thesaurus_data arguments to the lkt_create_thesaurus command. The thesaurus defined by this argument is created and exists only for the duration of this query and is local to this query. See Thesaurus Matching for more information about defining a thesaurus and TIBCO Patterns servers' thesaurus support in general and Ephemeral Thesauri for more information on these ephemeral thesauri in particular.
- LPAR_DBL_THESAURUSWEIGHT (optional) is a penalty factor that is applied to every thesaurus substitution. This penalty factor is applied for all thesaurus types, but only if there was a thesaurus match between query and record of two different terms (e.g. a match of Peggy to Margaret, but it does not apply to a match of Peggy to Peggy). If a combined thesaurus is being used the penalty factor defined in the thesaurus for the class is multiplied by the factor provided here to obtain the final penalty factor.

- LPAR_DBL_NONCOG_WGT (optional) is a penalty factor to be applied to transposed field matches.
- LPAR_DBL_COGNATE_EMPTY_PENALTY (optional) The adjustment to the penalty applied for unmatched data in extra fields. The default value is 1.0 (full penalty).
- LPAR_LST_QOPTS (optional) is a list of detailed tuning parameters which control how the TIBCO Patterns servers scores matches. It is extremely rare that any of these values need to be changed and should be changed only in consultation with your TIBCO representative.
- LPAR_STR_CHARMAP (optional) is the name of the character map to be applied to the query values. By default all fields referenced in a simple query must have the same character map. If a query is to be performed across fields that use two or more different character maps then this option must be supplied to specify which map is to be used.
- LPAR_STR_QLETGROUP (optional) assigns a group name to this querylet. Group names are used by the GIP prefilter to improve the selection of child records in joined searches. See the “Matching Compound Records and Querylet Grouping” section in the *TIBCO Patterns Concepts Guide* for more information on querylet grouping and when to use it.

Note: Setting LPAR_STR_CHARMAP option might result in invalid or incorrect results. Consult your TIBCO representative before using this option.

Attributes Query

An attributes query is a comparison of a set of input query strings against a set of attribute values. This query is analogous to an AND of simple queries, each simple query being a comparison of one value against one attribute. It is provided as a shortcut convenience when querying a set of Variable Attributes. Instead of needing to construct a simple query for each attribute and then an AND score combiner for the simple queries you need to supply only the list of values and the list of attribute names to build a single query.

By default if an attribute doesn't exist in a record, or is empty (has zero length value) it is ignored. This can be changed by setting the empty score option: LKT_DBL_EMPTY_DATA_SCORE. If the empty score is set instead of being ignored the match of this attribute is considered to have the given score. For example consider a query against 2 attributes color and size. Suppose one record has a perfect match on size but does not have an attribute named color. By default this record would get a score of 1.0 because the non-existent attribute is ignored, so effectively the query is only against the perfectly matching size attribute. If the empty score is set to 0.2 then this record would get a score of (1.0 + 0.2) / 2 = 0.6 (assuming field weights were both the default 1.0).

- LPAR_BLKARR_SEARCHQUERY specifies the query strings to be matched by the attribute values.
- LPAR_STRARR_ATTRNAMES selects the attributes to be matched. Note that these values are just the attribute name, not a full field name with attribute qualifier. The length of this array must be the same as the length of the LPAR_BLKARR_SEARCHQUERY array. There is a one to one correspondence with each search query value being matched against the attribute with the name in the corresponding location in this array. Note there is no requirement that attribute names be unique. The same attribute might be matched by different search query values.
- LPAR_DBLARR_QFIELDWEIGHTS (optional) specifies the weight for matched text against each attribute in the LPAR_STRARR_ATTRNAMES array. The weight must be between 0.0 and 1.0 inclusive. Note that these are relative weights, as in the cognate query, not penalizing weights as in the simple query.
- LPAR_DBL_EMPTY_DATA_SCORE (optional) this sets the score assigned to empty or non-existent attributes. See discussion above.
- LPAR_BOOL_MATCH_EMPTY controls behavior when a query string and the data it is being matched to are both empty. If true, a 1.0 score is generated. If false, the empty-score is used.

  Default: false
- LPAR_INT_SORTSCORE (optional) sets the score type that is considered to be the "match" score for this query. This supersedes on a query by query basis the value set in the search parameters.
- LPAR_STR_THESAURUSNAME (optional) is the thesaurus to use for this query. It applies to all attributes and query values listed. If LPAR_STR_THESAURUSNAME is specified LPAR_LST_THESAURUS might not be.
- LPAR_LST_THESAURUS (optional) defines a thesaurus to use for this query. It applies to all attributes and query values listed. If LPAR_LST_THESAURUS is specified LPAR_STR_THESAURUSNAME might not be. The list must consist of two lpars, they being the *thesaurus_options* and *thesaurus_data* arguments to the `lkt_create_thesaurus` command. The thesaurus defined by this argument is created and exists only for the duration of this query and is local to this query. See Thesaurus Matching for more information about defining a thesaurus and TIBCO Patterns server thesaurus support in general and Ephemeral Thesauri for more information on these ephemeral thesauri in particular.
- LPAR_DBL_THESAURUSWEIGHT (optional) is a penalty factor that is applied to every thesaurus substitution. This penalty factor is applied for all thesaurus types, but only if there was a thesaurus match between query and record of two different terms (e.g. a match of Peggy to Margaret, but it does not apply to a match of Peggy to Peggy). If a

combined thesaurus is being used the penalty factor defined in the thesaurus for the class is multiplied by the factor provided here to obtain the final penalty factor.

- LPAR_LST_QOPTS (optional) is a list of detailed tuning parameters which control how the TIBCO Patterns server scores matches. It is extremely rare that any of these values need to be changed and should be changed only in consultation with your TIBCO representative.
- LPAR_STR_QLETGROUP (optional) assigns a group name to this querylet. Group names are used by the GIP prefilter to improve the selection of child records in joined searches. See the “Matching Compound Records and Querylet Grouping” section in the *TIBCO Patterns Concepts Guide* for more information on querylet grouping and when to use it.

Predicate Query

The predicate is evaluated on the record and the result used as the score assigned by this score generator. Generally, the Boolean tests are used, in which case if it's true the score is 1.0, if it's false the score is 0.0. It is possible to have the predicate return a double value, which is used directly as the score. If this is done the score must be in the range 0.0 to 1.0 inclusive.

A predicate query is a simple comparison for use when other queries are both slower and more cumbersome (like a gender comparison). It can also be used when inexact matching is not required and speed and precision are preferred. A predicate is specified as either an LPAR_LST_PREDICATE or an LPAR_STR_PREDICATE or LPAR_BLK_PREDICATE described in Predicate Expressions or as a list LPAR_LST_PREDICATE_QRY which contains the following parameters.

- LPAR_LST_PREDICATE specifies the predicate test. Either this parameter or LPAR_STR_PREDICATE or LPAR_BLK_PREDICATE must appear exactly once.
- LPAR_DBL_INVALID_DATA_SCORE (optional) is the score returned if a record field used in the query expression has invalid data or if the predicate evaluation fails. This supersedes on a query by query basis the value set in the search parameters.
- LPAR_DBL_EMPTY_DATA_SCORE (optional) is the score to return for a record if a record field used in the predicate expression is empty. This supersedes on a query by query basis the value set in the search parameters.
- LPAR_STR_QLETGROUP (optional) assigns a group name to this querylet. Group names are used by the GIP prefilter to improve the selection of child records in joined searches. See the “Matching Compound Records and Querylet Grouping” section in the *TIBCO Patterns Concepts Guide* for more information on querylet grouping and when to use it.

Query Combiners

Query combiners, also known as query expressions, are used to combine multiple input scores into a single score. LKT_QEXPR_AND takes the simple average of the input scores, LKT_QEXPR_OR takes the maximum score, and LKT_QEXPR_RLINK performs a much more sophisticated weighted combination under the control of a TIBCO Patterns machine learning model. The query structure used must be the same as was used to train the TIBCO Patterns Learn model. Your TIBCO technical representative can provide more information on how to set up a query for the RLINK score combiner.

With release 4.2 the special query expression LKT_QEXPR_RECWGT was added. Unlike the other expressions this modifies a single score rather than combining multiple scores. It multiplies the single score by the value of a field in the record, which must be a Float field type, to produce a new score for the record (resulting scores are constrained to the range 0.0 to 1.0). This provides a way of increasing or decreasing record priorities in matching.

A query expression is specified as a LPAR_LST_QEXPR which contains the following parameters.

- LPAR_INT_QEXPR_TYPE specifies the type of the query combiner (e.g. LKT_QEXPR_AND).
- LPAR_LST_QEXPR_ARGS is a list of the score generators (or combiners), the output of which is combined or modified by this expression. The elements of this list is typically the Simple, Cognate, and Predicate score generators described above.
- LPAR_DBLARR_QUERYLETWEIGHTS (optional) specifies the weight for each score being combined in this expression. The maximum weight is 1.0, with values less than 1.0 representing penalized matches. For instance, using the weights in an AND combiner is equivalent to a weighted average.
- LPAR_STR_QLETGROUP (optional) assigns a group name to this querylet. Group names are used by the GIP prefilter to improve the selection of child records in joined searches. See the “Matching Compound Records and Querylet Grouping” section in the *TIBCO Patterns Concepts Guide* for more information on querylet grouping and when to use it.
- LPAR_LST_QOPTS (options) contains any general query options for this expression.

  Currently the available members of this list by expression type are:

  — LKT_QEXPR_AND: LPAR_DBLARR_IGNORE_SCORES is an array of cutoff scores for each sub-query of this query expression. The length of the array must equal the number of sub-queries in the expression. The values must be either -1.0 (the special reject record score), -2.0 (a special value to indicate no ignore cutoff processing is to be performed for this sub-query) or be in the range 0.0 - 1.0. This provides a means of ignoring sub-queries with poor matches or sub-queries with empty or invalid data. Any sub-query with a score at or below the corresponding score in this array is ignored completely. So if you have three sub-queries with scores: 1.0, 0.7, 0.1 and an ignore scores array of: 0.1, 0.1, 0.1. Then the score would be 0.85 instead of 0.6. To ignore sub queries on empty or invalid data set the LPAR_DBL_EMPTY_DATA_SCORE and LPAR_DBL_INVALID_DATA_SCORE for the sub-query to -1.0 and set the associated ignore scores array value to -1.0.

Warning: Ignore scores should be used with extreme caution. Ignoring a low score in one sub-query would likely boost the score of a record with a poor match on that sub-query (For example, a score below the threshold) above the score of a similar record with a good match on that sub-query (For example, above the threshold). Thus when used inappropriately ignore score thresholds tend to push poorer matches above better matches. This is especially true if ignore thresholds are applied to more than one sub-query. Thus ignore thresholds should only be applied in the rare cases where a low score indicates the sub-query is completely irrelevant and the record should score above close but imperfect matches on the sub-query. Another valid use would be to ignore empty fields as described above, although normally it is better to just let the default score of 0.0 be averaged into the output score.

— LKT_QEXPR_AND: LPAR_DBLARR_REJECT_SCORES is an array of cutoff scores similar to LPAR_DBLARR_IGNORE_SCORES except that instead of ignoring the sub-query the record is rejected. (More precisely the output score for the AND expression is set to -1.0, which normally causes the record to be rejected, but could be trapped by a higher level AND expression and ignored as described above for LPAR_DBLARR_IGNORE_SCORES.) LPAR_DBLARR_REJECT_SCORES accepts the same score values with the same meaning as LPAR_DBLARR_IGNORE_SCORES.

  If a sub-query is assigned both an ignore score and a reject score then the lesser value takes precedence over its range, the greater value is applied to the range from the lesser to the greater. E.g. if the reject score for sub-query 1 is 0.2 and the ignore score for sub-query 1 is 0.4 then records with a sub-query 1 score less than 0.2 are rejected and for records with a sub-query 1 score between 0.2 and 0.4 sub-query 1 is ignored. Conversely if the ignore score is 0.2 and the reject score 0.4 then scores less than 0.2 causes sub-query 1 to be ignored and scores from 0.2 to 0.4 causes the record to be rejected.

Warning: If using both ignore scores and reject scores on the same set of sub-queries it should be noted that ignore thresholds tend to favor records with high scores for one sub-query and very low scores for the others. But reject scores pass a record only if all sub-queries are above their given reject threshold. Thus combining ignore scores and reject scores can result in few or no records being returned.

— LKT_QEXPR_RLINK: LPAR_STR_RLMODELNAME, which specifies the model name to be used.

— LKT_QEXPR_RLINK: LPAR_BOOL_USEMODELTHRESH indicates whether the cutoff threshold in the TIBCO Patterns Learn model should be used. If this is set to true, and the named model contains a threshold value, the value is applied as an absolute cutoff score for cutoff processing. This supersedes any cutoff specified in the search options. If the model does not contain a threshold value, this option is ignored. The default value for this option is false. See Dynamic Score Cutoffs for details on cut off processing.

— LKT_QEXPR_RECWGT: LPAR_STR_WGTFLD_NAME specifies the name of the field to be used to provide the weight value to be applied to the score or LPAR_INT_WGTFLD_NUM specifies the field using a field number instead of a name.

— LKT_QEXPR_MATCH: The LPARs LPAR_DBL_MATCHSTRENGTH, LPAR_DBLARR_MATCHTHRESHOLDS, LPAR_DBLARR_MATCHREWARDS and LPAR_DBLARR_MATCHPENALTIES are accepted. For a description of these see MatchCase Score Combiner .

— LKT_QEXPR_FIRSTVALID: LPARs: LPAR_DBLARR_CONFIDENCE_CUTOFFS, LPAR_DBLARR_MATCH_CUTOFFS and LPAR_BOOL_INVALID_ONLY are accepted. For a description of these see FirstValid Score Combiner.

Consider the previous example, but with the additional complication that the user now has five text boxes. The first and last names have been isolated into separate entries, but we still want to allow for queries or records where the first and last name have been accidentally transposed. In addition, there is now a gender box which is compared to the record using a predicate. The following code sets up the described query.

```
lpar_t        ANDquery,ANDqueryargs,simpquery,cogquery,sq,sf,ncw,predicate;
unsigned char *fname,*lname,*address,*ssn,*gender;
unsigned char *name_fields[]    = { "First Name", "Last Name" };
unsigned char *ssn_fields[]     = { "Social Security Number" };
unsigned char *address_fields[] = { "House Number", "Street Name", "City" };
unsigned char *gender_field     = "Gender";
char   *names[2];
int    namelens[2];
double qweights[4] = { 1.0, 0.9, 0.75, 1.0 };
fname   = get_fname_box();
lname   = get_lname_box();
address = get_address_box();
ssn     = get_ssn_box();
gender  = get_gender_box();
ANDqueryargs=lpar_create_lst(LPAR_LST_QEXPR_ARGS);
/* Name cogquery */
cogquery    =lpar_create_lst(LPAR_LST_COGQUERY);
names[0]    = fname;
namelens[0] = strlen(fname);
```

```
names[1]    = lname;
namelens[1] = strlen(lname);
sq=lpar_create_blkarr(LPAR_BLKARR_SEARCHQUERY,names,namelens);
lpar_append_lst(cogquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,name_fields,2);
lpar_append_lst(cogquery,sf);
ncw=lpar_create_int(LPAR_DBL_NONCOG_WGT,0.8);
lpar_append_lst(cogquery,ncw);
lpar_append_lst(ANDqueryargs,cogquery);
/* SSN simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,ssn,strlen(ssn));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,ssn_fields,1);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
/* Address simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
/* Gender predicate */
predicate = lpar_create_predicate3(
                    lpar_create_str(LPAR_STR_PREDFIELD,gender_field)),
                    PRED_OP_EQUALS,
                    lpar_create_str(LPAR_STR_PREDVALUE,gender));
lpar_append_lst(ANDqueryargs,predicate);
ANDquery=lpar_create_lst(LPAR_LST_QEXPR);
lpar_append_lst(ANDquery,
                lpar_create_int(LPAR_INT_QEXPR_TYPE,LKT_QEXPR_AND));
lpar_append_lst(ANDquery,
                lpar_create_dblarr(LPAR_DBLARR_QUERYLETWEIGHTS,qweights,4));
```

First we create the list of queries to be combined. The first two querylets are part of a single cognate query, so the record score is calculated using a single bipartite graph using the two named record fields. The optional parameter, LPAR_DBL_NONCOG_WGT, sets up a penalty factor for non-cognate field matches as described in Cognate Query. The second and third queries are simple queries on the SSN and address fields respectively. Finally the predicate query tests the gender value.

The query combiner is then created and set as an AND combiner. We add the querylet weight array to the query combiner. Here we have decided to rank the name and gender as most important, followed by the SSN field with the address being least important. Finally we add the list of queries that are to be combined. This query is now ready to be used in a call to `lkt_dbsearch`.

Now suppose we have a music download site. Music is identified by song title and artist. For promotional reasons we wish to favor certain records over others. To do this we add a float field we'll call promotional ranking that contains a value between 0.5 and 1.5. We can set up a query that returns the matches adjusted by the promotional ranking as shown in the following code segment:

```
lpar_t ANDquery,ANDqueryargs,simpquery,sq,sf,opts;
lpar_t WGTquery,WGTqueryargs;
unsigned char *fname,*lname,*address,*ssn,*gender;
unsigned char *title_field[] =  { "song title" } ;
unsigned char *artist_field[] = { "artist" };
unsigned char *ranking_field[] = { "promotional ranking" };
unsigned char *title ;
unsigned char *artist ;
title   = get_title_box();
artist  = get_artist_box();
ANDqueryargs=lpar_create_lst(LPAR_LST_QEXPR_ARGS);
/* Title simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,title,strlen(title));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,title_field,1);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
/* Artist simpquery */
simpquery=lpar_create_lst(LPAR_LST_SIMPLEQUERY);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,artist,strlen(artist));
lpar_append_lst(simpquery,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,artist_field,1);
lpar_append_lst(simpquery,sf);
lpar_append_lst(ANDqueryargs,simpquery);
/* Create And of artist and title simple queries */
ANDquery=lpar_create_lst(LPAR_LST_QEXPR);
lpar_append_lst(ANDquery,lpar_create_int(LPAR_INT_QEXPR_TYPE,LKT_QEXPR_AND));
lpar_append_lst(ANDquery,ANDqueryargs);
/* Now apply promotional rating to combined scores */
WGTquery=lpar_create_lst(LPAR_LST_QEXPR);
lpar_append_lst(WGTquery, lpar_create_int(LPAR_INT_QEXPR_TYPE,LKT_QEXPR_
RECWGT));
opts = lpar_create_lst(LPAR_LST_QOPTS);
lpar_append_lst(opts, lpar_create_str(LPAR_STR_WGTFLD_NAME,ranking_field);
lpar_append_lst(WGTquery, opts);
WGTqueryargs = lpar_create_lst(LPAR_LST_QEXPR_ARGS);
```

```
lpar_append_lst(WGTqueryargs,ANDquery);
lpar_append_lst(WGTquery, WGTqueryargs);
```

Now WGTquery is ready to be passed to `dvk_dbsearch`. The raw score of each record is adjusted by multiplying the score by the value of the promotional ranking field to produce the final score used in selecting and ranking the records for return. The score returned is the score after adjustment. The original score is available in the LPAR_DBLARR_MATCHSCORE_QLT value.

MatchCase Score Combiner

A MatchCase query combiner, like other combiners, merges a set of scores into a single output score. Like all other combiners, it can appear as any non-leaf node in a query tree. There is one important subtle difference between the MatchCase combiner and other combiners. Other combiners output a score that represents a degree of similarity between the query and the record. The MatchCase Score combiner outputs a score that represents the likelihood that the query and the record represent the same entity. Like the similarity score, this is not a probability, it is a relative measure. Also, like the similarity score, it is a value between 0.0 and 1.0.

The distinction between a similarity score and the match likelihood score output by the MatchCase combiner is an important one. Two records might be very similar but do not represent the same entity. On the other hand, two records might have a low similarity score but represent the same entity. In determining whether two records represent the same entity, knowing which portions matched, and to what degree is critical. By using the MatchCase Score combiner, you can directly express which portions of the query must match, and to what degree. For more information on the MatchCase Score combiner and examples of how it is used, see the "MatchCase Score Combiner" section in the *TIBCO Patterns Concept Guide.*

The MatchCase Score combiner is a query expression and accepts the same items in the LPAR_ LST_QEXPR list as all other query expressions. The MatchCase Score combiner accepts the following additional LPAR values in the LPAR_LST_QOPTS list. These LPARs are valid only within the LPAR_LST_QOPTS list of a query expression of type LKT_QEXPR_MATCH.

- **LPAR_DBL_MATCHSTRENGTH** This holds the match strength for the MatchCase. This is required for all LKT_QEXPR_MATCH type query expressions.
- **LPAR_DBLARR_MATCHTHRESHOLDS** This contains the list of threshold values. Entries corresponding to core querylets must be negative. The absolute value of the entry represents the minimum score allowed for the core querylet. Entries corresponding to secondary querylets must be positive. Positive entries are the match/non-match threshold value for the secondary querylet. This is required for all LKT_QEXPR_MATCH type query expressions.

- **LPAR_DBLARR_MATCHREWARDS** This is the reward weight factors. Values corresponding to core querylets are ignored. This is required for all LKT_QEXPR_MATCH type query expressions.
- **LPAR_DBLARR_MATCHPENALTIES** This is the penalty weight factors. Values corresponding to core querylets are ignored. This is optional. If this LPAR is not given, penalty factors default to the reward factors.

The following is the example from the *TIBCO® Patterns Concepts Guide* rewritten using the "C" API:

```
/* Querylet LPARs for each category (setting not shown) */
lpar_t    name_cat_qry ;      /* name category querylet */
lpar_t    street_cat_qry ;    /* street address category querylet */
lpar_t    location_cat_qry ;  /* location category querylet */
lpar_t    phone_cat_qry ;     /* phone number category querylet */
lpar_t    dob_cat_qry ;       /* date of birth category querylet */
/* Our Intermediate Querylets */
lpar_t    cat_qrys ;          /* category queries for match rules */
lpar_t    rule_1_qry ;        /* match case query for rule 1 */
lpar_t    rule_1_qopts ;      /* match case settings for rule 1 */
lpar_t    rule_2_qry ;        /* match case query for rule 2 */
lpar_t    rule_2_qopts ;      /* match case settings for rule 2 */
lpar_t    final_query_args ;  /* arguments for full query */
/* The final query. */
lpar_t    final_query ;
/* For example we define rule parameters as static values */
/* These could be read in from a configuration file. */
/* Rule 1 */
/* ------ */
/* Weights for core categories, others are left as 0.0 */
static double rule_1_weights[] = { 1.0, 0.80, 0.80, 0.0, 0.0 } ;
/* Thresholds for all categories, core categories are negative */
static double rule_1_thresholds[] = { -0.70, -0.80, -0.75, 0.85, 0.60 } ;
/* Rewards for secondary categories (core categories left 0.0) */
static double rule_1_rewards[] = { 0.0, 0.0, 0.0, 0.15, 0.40 } ;
/* Penalty factors for secondary categories */
static double rule_1_penalties[] = { 0.0, 0.0, 0.0, 0.05, 0.50 } ;
/* Rule 2 */
/* ------ */
/* Weights for core categories, others are left as 0.0 */
static double rule_2_weights[] = { 1.0, 0.0, 0.0, 0.60, 0.90 } ;
/* Thresholds for all categories, core categories are negative */
static double rule_2_thresholds[] = { -0.70, 0.85, 0.80, -0.80, -0.50 } ;
/* Rewards for secondary categories (core categories left 0.0) */
static double rule_2_rewards[] = { 0.0, 0.35, 0.35, 0.0, 0.0 } ;
/* Penalty factors for secondary categories */
```

```
static double rule_2_penalties[] = { 0.0, 0.10, 0.10, 0.0, 0.0 } ;
/* Create the query expression arguments list. */
cat_qrys = lpar_create_lst(LPAR_LST_QEXPR_ARGS) ;
lpar_append_lst(cat_qrys, name_cat_qry) ;
lpar_append_lst(cat_qrys, street_cat_qry) ;
lpar_append_lst(cat_qrys, location_cat_qry) ;
lpar_append_lst(cat_qrys, phone_cat_qry) ;
lpar_append_lst(cat_qrys, dob_cat_qry) ;
/* Define Rule 1 Query */
/* ------------------- */
/* Create the Query. */
rule_1_qry = lpar_create_lst(LPAR_LST_QEXPR) ;
/* Mark it as a Match Case Query. */
lpar_append_lst(rule_1_qry,
                lpar_create_int(LPAR_INT_QEXPR_TYPE,
                                LKT_QEXPR_MATCH)) ;
/* Add the arguments (category querylets) */
lpar_append_lst(rule_1_qry, cat_qrys) ;
/* Core query weights are added as querylet weights. */
lpar_append_lst(rule_1_qry,
            lpar_create_dblarr(LPAR_DBLARR_QUERYLETWEIGHTS,
                               rule_1_weights,
                               5)) ;
/* All other Match Case parameters go in the Query Options list. */
rule_1_qopts = lpar_create_lst(LPAR_LST_QOPTS) ;
/* Add match strength. */
lpar_append_lst(rule_1_qopts,
            lpar_create_int(LPAR_DBL_MATCHSTRENGTH, 0.8)) ;
/* Add thresholds */
lpar_append_lst(rule_1_qopts,
            lpar_create_dblarr(LPAR_DBLARR_MATCHTHRESHOLDS,
                               rule_1_thresholds,
                               5)) ;
/* Add rewards */
lpar_append_lst(rule_1_qopts,
            lpar_create_dblarr(LPAR_DBLARR_MATCHREWARDS,
                               rule_1_rewards,
                               5)) ;
/* Add Penalties */
lpar_append_lst(rule_1_qopts,
            lpar_create_dblarr(LPAR_DBLARR_MATCHPENALTIES,
                               rule_1_penalties,
                               5)) ;
/* Add Query Options to Rule 1 query. */
lpar_append_lst(rule_1_qry, rule_1_qopts) ;
/* Define Rule 2 Query */
/* ------------------- */
/* Create the Query. */
```

```
rule_2_qry = lpar_create_lst(LPAR_LST_QEXPR) ;
/* Mark it as a Match Case Query. */
lpar_append_lst(rule_2_qry,
                lpar_create_int(LPAR_INT_QEXPR_TYPE,
                                LKT_QEXPR_MATCH)) ;
/* Add the arguments (category querylets, we make a copy) */
lpar_append_lst(rule_2_qry, lpar_copy(cat_qrys)) ;
/* Core query weights are added as querylet weights. */
lpar_append_lst(rule_2_qry,
            lpar_create_dblarr(LPAR_DBLARR_QUERYLETWEIGHTS,
                               rule_2_weights,
                               5)) ;
/* All other Match Case parameters go in the Query Options list. */
rule_2_qopts = lpar_create_lst(LPAR_LST_QOPTS) ;
/* Add match strength. */
lpar_append_lst(rule_2_qopts,
            lpar_create_int(LPAR_DBL_MATCHSTRENGTH, 0.85)) ;
/* Add thresholds */
lpar_append_lst(rule_2_qopts,
            lpar_create_dblarr(LPAR_DBLARR_MATCHTHRESHOLDS,
                               rule_2_thresholds,
                               5)) ;
/* Add rewards */
lpar_append_lst(rule_2_qopts,
            lpar_create_dblarr(LPAR_DBLARR_MATCHREWARDS,
                               rule_2_rewards,
                               5)) ;
/* Add Penalties */
lpar_append_lst(rule_2_qopts,
            lpar_create_dblarr(LPAR_DBLARR_MATCHPENALTIES,
                               rule_2_penalties,
                               5)) ;
/* Add Query Options to Rule 2 query. */
lpar_append_lst(rule_2_qry, rule_2_qopts) ;
/* Final Query - Is OR of two Match Case Queries */
final_query = lpar_create_lst(LPAR_LST_QEXPR) ;
/* Mark it as an OR Query. */
lpar_append_lst(final_query,
                lpar_create_int(LPAR_INT_QEXPR_TYPE,
                                LKT_QEXPR_OR)) ;
/* create arguments list. */
final_query_args = lpar_create_lst(LPAR_LST_QEXPR_ARGS) ;
lpar_append_lst(final_query_args, rule_1_qry) ;
lpar_append_lst(final_query_args, rule_2_qry) ;
/* add it to OR query */
lpar_append_lst(final_query, final_query_args) ;
/* The query is now completed. */
return final_query ;
```

First Valid Score Combiner

The First Valid score combiner is used in conjunction with the RLINK score combiner (see TIBCO Patterns Machine Learning Platform and Using a Learn Model in a Search). It is used to select a querylet based on its confidence value. For more information on the use cases for this score combiner, see the section "TIBCO Patterns Learn Features" in the *TIBCO® Patterns Concepts Guide* that explains handling and finding low confidence pairs.

This combiner takes the standard set of options for query combiners. In addition it takes the following values in the LPAR_LST_QOPTS list:

- LPAR_DBLARR_CONFIDENCE_CUTOFFS is the cutoff value for the confidence value of the querylets. This value is required and must have the same number of elements as there are querylets for this combiner. If the LPAR_BOOL_INVALID_ONLY flag is present and true this value represents a maximum confidence value; if the associated querylet has a confidence value greater than this value it is skipped. If LPAR_BOOL_INVALID_ONLY is not given, or is false, and the associated querylet has a confidence value less than this value, it is skipped. All values must be in the range 0.0 to 1.0 inclusive.
- LPAR_DBLARR_MATCH_CUTOFFS is the cutoff threshold value for the child queries. This value is required and must have the same number of elements as there are querylets for this combiner. This value is used to normalize the scores of the querylets. The intent is to allow each querylet to use the same cutoff point for scores considered a match verses those considered a non-match.

  The score for each querylet is normalized so that a score equal to its match cutoff score defined here is set equal to the match cutoff score of the first querylet. Scores greater or less than the match cutoff score are adjusted using linear interpolation. Note this means the score of the first querylet is not changed. A negative score means the absolute cutoff score for the query should be used as the match cutoff score. If no absolute cut off score is defined for this query a negative score is rejected with a DVK_ERR_PARAMVAL error. Match cutoff scores above 1.0 are rejected with a DVK_ERR_PARAMVAL error.
- LPAR_BOOL_INVALID_ONLY indicates whether querylets with confidence values greater than or less than the confidence cutoff score are kept. This value is optional. The default value is false. If this value is true, the first querylet with a confidence value less than or equal to the confidence cutoff score is used. If false or not given, the first querylet with a confidence value greater than or equal to the cutoff score is used.

This combiner scans its querylets in the order given. The first querylet that meets its confidence cutoff test is used to supply the scores and the confidence value returned by this combiner. If no querylet meets its confidence cutoff test the special -1.0 score is returned to reject this record.

Named Querylets

Complex query trees can be constructed using the score generator and score combiner querylets described in the Query Combiners section. By default only the scores from the top level querylet are returned. Usually this is all that is needed, but in some cases you might need to know the score returned by a lower level querylet. For example, consider a query to find a person where records contain several alternate names:

```
    And( Or( Simple("tom jones", { "first", "middle", "last" }),
             Simple("tom jones", { "aka_1" }),
             Simple("tom jones", { "aka_2" })
           ),
         Simple("123 Walrus St", { "street" }),
         Simple("Santa Cruz", { "city" }),
         Simple("CA", { "state" })
       )
```

In addition to the overall score, it would be good to know how well it matched on the primary "first", "middle", and "last" name fields. By default only the top level scores are returned. So there is the overall score and the top level querylet scores, but because the simple query for the names appears two levels down, you do not have access to the scores returned by these querylets. By assigning a name to these querylets you can retrieve the desired scores. The following example shows how to construct the above query assigning a name to each of the simple querylets for the name:

```
/* field names used by querylets */
unsigned char *name_fields[]   = { "first", "middle", "last" };
unsigned char *aka1_fields[]   = { "aka1" } ;
unsigned char *aka2_fields[]   = { "aka2" } ;
unsigned char *street_fields[] = { "street" } ;
unsigned char *city_fields[]   = { "city" } ;
unsigned char *state_fields[]  = { "state" } ;
/* our search values */
const unsigned char *name_value = (const unsigned char *)"tom jones" ;
const unsigned char *street_value = (const unsigned char *)"123 Walrus St" ;
const unsigned char *city_value = (const unsigned char *)"Santa Cruz" ;
const unsigned char *state_value = (const unsigned char *)"CA" ;
/* our query lpar's */
lpar_t name_query ;
lpar_t street_query ;
lpar_t city_query ;
lpar_t state_query ;
lpar_t or_queryargs ;
lpar_t and_queryargs ;
lpar_t or_query ;
lpar_t and_query ;
and_queryargs = lpar_create_lst(LPAR_LST_QEXPR_ARGS) ;
or_queryargs = lpar_create_lst(LPAR_LST_QEXPR_ARGS) ;
/* create name simple query */
name_query = lpar_create_lst(LPAR_LST_SIMPLEQUERY) ;
lpar_append_lst(name_query, lpar_create_blk(LPAR_BLK_SEARCHQUERY,
                                            name_value,
                                            strlen(name_value))) ;
lpar_append_lst(name_query,
                lpar_create_strarr(LPAR_STRARR_FIELDNAMES, name_fields, 3)) ;
/* assign name to this querylet */
lpar_append_lst(name_query, lpar_create_str(LPAR_STR_QLETNAME, "primary")) ;
/* add to Or query args. */
lpar_append_lst(or_queryargs, name_query) ;
/* create first aka simple query */
name_query = lpar_create_lst(LPAR_LST_SIMPLEQUERY) ;
lpar_append_lst(name_query, lpar_create_blk(LPAR_BLK_SEARCHQUERY,
                                            name_value,
                                            strlen(name_value))) ;
lpar_append_lst(name_query,
                lpar_create_strarr(LPAR_STRARR_FIELDNAMES, aka1_fields, 1)) ;
/* assign name to this querylet */
lpar_append_lst(name_query, lpar_create_str(LPAR_STR_QLETNAME, "aka1")) ;
/* add to Or query args. */
lpar_append_lst(or_queryargs, name_query) ;
/* create second aka simple query */
name_query = lpar_create_lst(LPAR_LST_SIMPLEQUERY) ;
lpar_append_lst(name_query, lpar_create_blk(LPAR_BLK_SEARCHQUERY,
                                            name_value,
                                            strlen(name_value))) ;
lpar_append_lst(name_query,
                lpar_create_strarr(LPAR_STRARR_FIELDNAMES, aka2_fields, 1)) ;
/* assign name to this querylet */
lpar_append_lst(name_query, lpar_create_str(LPAR_STR_QLETNAME, "aka2")) ;
/* add to Or query args. */
lpar_append_lst(or_queryargs, name_query) ;
/* create OR query */
or_query = lpar_create_lst(LPAR_LST_QEXPR) ;
lpar_append_lst(or_query,
                lpar_create_int(LPAR_INT_QEXPR_TYPE, LKT_QEXPR_OR)) ;
/* add to And query arguments. */
lpar_append_lst(and_queryargs, or_query) ;
/* create simple queries for other items */
street_query = lpar_create_lst(LPAR_LST_SIMPLEQUERY) ;
lpar_append_lst(street_query, lpar_create_blk(LPAR_BLK_SEARCHQUERY,
                                              street_value,
                                              strlen(street_value))) ;
```

```
lpar_append_lst(street_query,
                lpar_create_strarr(LPAR_STRARR_FIELDNAMES, street_fields, 1))
;
/* add to And query args. */
lpar_append_lst(and_queryargs, street_query) ;
city_query = lpar_create_lst(LPAR_LST_SIMPLEQUERY) ;
lpar_append_lst(city_query, lpar_create_blk(LPAR_BLK_SEARCHQUERY,
                                              city_value,
                                              strlen(city_value))) ;
lpar_append_lst(city_query,
                lpar_create_strarr(LPAR_STRARR_FIELDNAMES, city_fields, 1)) ;
/* add to And query args. */
lpar_append_lst(and_queryargs, city_query) ;
state_query = lpar_create_lst(LPAR_LST_SIMPLEQUERY) ;
lpar_append_lst(state_query, lpar_create_blk(LPAR_BLK_SEARCHQUERY,
                                              state_value,
                                              strlen(state_value))) ;
lpar_append_lst(state_query,
                lpar_create_strarr(LPAR_STRARR_FIELDNAMES, state_fields, 1))
;
/* add to And query args. */
lpar_append_lst(and_queryargs, state_query) ;
```

When named querylets are included in a query, the returned stats list contains an LPAR_STRARR_QLETNAMES value. This lists the names of all of the named querylets. Each minfo record contains an LPAR_DBLARR_NAMEDQLETSCORES value. This contains the score for each of the named querylets. The scores are presented in the same order as the names in the LPAR_STRARR_QLETNAMES lpar. The following example shows how the match score for the primary set of names can be retrieved:

```
lpar_t        named_qlet_fields ;
unsigned char **qlet_names ;
int           num_named_qlets ;
int           qlet_idx ;
lpar_t        *minfo_items ;
int           numrecs ;
int           recnum ;
lpar_t        named_qlet_scores_lpar ;
double        *named_qlet_scores ;
int           num_named_qlet_scores ;
/* Get our list of named querylets. */
named_qlet_fields = lpar_find_lst_lpar(stats, LPAR_STRARR_QLETNAMES) ;
if (named_qlet_fields == LPAR_NULL) {
    /* Handle Error. */
    /* abort processing */
}
qlet_names = lpar_get_strarr(named_qlet_fields, &num_named_qlets) ;
/* find the position of the querylet named "primary". */
for (qlet_idx = 0 ;
     (    (qlet_idx < num_named_qlets)
       && (strcmp("primary",qlet_names[qlet_idx]) != 0)) ;
     qlet_idx++ ) {
}
if (qlet_idx >= num_named_qlets) {
    /* It is an unexpected error if it is not in the list. */
    /* abort processing */
}
/* now scan all the minfo records, displaying the primary querylet score */
minfo_items = lpar_get_lst_item_array(minfo, &numrecs) ;
for (recnum = 0; recnum < numrecs; recnum++) {
    /* find the named querylet scores for this record */
    named_qlet_scores_lpar = lpar_find_lst_lpar(minfo_items[recnum],
                                                LPAR_DBLARR_NAMEDQLETSCORES)
;
    if (named_qlet_scores_lpar == LPAR_NULL) {
        /* It is an unexpected error if this is not in the list. */
        /* abort processing. */
    }
    named_qlet_scores = lpar_get_dblarr(named_qlet_scores_lpar,
                                        num_named_qlet_scores) ;
    if (num_named_qlet_scores != num_named_qlets) {
        /* these lists are always the same length, so this is an error! */
        /* abort processing */
    }
    /* display the score of the named querylet named "primary" */
    printf("Primary score for record %d = %f\n",
           recnum, named_qlet_scores[qlet_idx]) ;
}
```

The following are some points to remember about named querylets:

- Querylet names are letter case sensitive. "Name" is different than "name".
- Although the order of querylet names within LPAR_STRARR_QLETNAMES and scores within LPAR_DBLARR_NAMEDQLETSCORES are guaranteed to be the same, there is no guaranteed ordering of querylet names within LPAR_STRARR_QLETNAMES relative to the placement of the associated querylet in the query structure. In other words, you cannot infer the location of a named querylet score based on the position of the querylet in the query. You must use the LPAR_STRARR_QLETNAMES array to find the location.

- A querylet name can be assigned to any type of querylet. All score generators and score combiner query types can be assigned a name.

Note: All querylet names must be unique. The same name cannot be assigned to different querylets.

Querylet References

In some complex query situations, the identical querylet must be repeated two or more times at different locations in the query tree. The example in the Match Case score combiner section shows one of these queries. In this example the querylets for each category are repeated twice, one time for each match rule. Some situations might require far more match rules. Because the category querylets are repeated for each match rule, this can result in many copies of each category querylet appearing in the query tree, and the querylet being reevaluated many times. This incurs a substantial performance penalty. The result is the same every time it is evaluated, so it is not necessary to reevaluate each category querylet each time it appears. Instead a querylet reference can be used.

A querylet reference can be used anyplace an **LPAR_LST_QEXPR** can be used. The result output by a querylet reference is the same as the result of the querylet it references. No reevaluation is performed; it uses the same output.

A querylet reference is specified as an **LPAR_LST_REFQUERY**. This list contains exactly one item: **LPAR_STR_QLETNAME**. The name in this item must match a named querylet on the query tree (see First Valid Score Combiner for more information on this). There is no order dependency between the reference and the referenced querylet. The referenced querylet only needs to be somewhere on the query tree. If there is no querylet with the given name on the query tree, a **DVK_ERR_QLETREFBROKEN** error is returned.

A querylet reference can reference any named querylet. This can be a leaf, or a query combiner with a complex sub-tree beneath it.

A querylet reference cannot be named. References to references are not allowed.

Circular references are not allowed. A querylet reference is not allowed to reference a querylet that has a node on its sub-tree that is a reference to an ancestor of the querylet reference. If a circular reference is detected, a **DVK_ERR_QLETREFLOOP** error is returned.

The following example shows how the MatchCase example can be modified to use querylet references. New or changed lines are in bold face font.

```
/* Querylet LPARs for each category (setting not shown) */
lpar_t    name_cat_qry ;      /* name category querylet */
lpar_t    street_cat_qry ;    /* street address category querylet */
lpar_t    location_cat_qry ;  /* location category querylet */
lpar_t    phone_cat_qry ;     /* phone number category querylet */
lpar_t    dob_cat_qry ;       /* date of birth category querylet */
/* References to the category queries */
lpar_t    name_ref ;
lpar_t    street_ref ;
lpar_t    location_ref ;
lpar_t    phone_ref ;
lpar_t    dob_ref ;
/* Our Intermediate Querylets */
lpar_t    cat_qrys ;          /* category queries for match rules */
lpar_t    cat_ref_qrys ;      /* category queries as references */
lpar_t    rule_1_qry ;        /* Match Case query for rule 1 */
lpar_t    rule_1_qopts ;      /* Match Case settings for rule 1 */
lpar_t    rule_2_qry ;        /* Match Case query for rule 2 */
lpar_t    rule_2_qopts ;      /* Match Case settings for rule 2 */
lpar_t    final_query_args ;  /* arguments for full query */
/* The final query. */
lpar_t    final_query ;
/* For example we define rule parameters as static values */
/* These could be read in from a configuration file. */
/* Rule 1 */
/* ------ */
/* ... as before ... */
/* Rule 2 */
/* ------ */
/* ... as before ... */
/* We must name the category queries to reference them. */
lpar_append_lst(name_cat_qry,
                lpar_create_str(LPAR_STR_QLETNAME, "Name Cat")) ;
lpar_append_lst(street_cat_qry,
                lpar_create_str(LPAR_STR_QLETNAME, "Street Cat")) ;
lpar_append_lst(location_cat_qry,
                lpar_create_str(LPAR_STR_QLETNAME, "Location Cat")) ;
lpar_append_lst(phone_cat_qry,
                lpar_create_str(LPAR_STR_QLETNAME, "Phone Cat")) ;
lpar_append_lst(dob_cat_qry,
                lpar_create_str(LPAR_STR_QLETNAME, "DOB Cat")) ;
/* Create the query expression arguments list. */
cat_qrys = lpar_create_lst(LPAR_LST_QEXPR_ARGS) ;
lpar_append_lst(cat_qrys, name_cat_qry) ;
lpar_append_lst(cat_qrys, street_cat_qry) ;
lpar_append_lst(cat_qrys, location_cat_qry) ;
lpar_append_lst(cat_qrys, phone_cat_qry) ;
lpar_append_lst(cat_qrys, dob_cat_qry) ;
```

```
/* Create Reference Querylets to the category querylets. */
name_ref = lpar_create_lst(LPAR_LST_REFQUERY);
lpar_append_lst(name_ref,
                lpar_create_str(LPAR_STR_QLETNAME, "Name Cat")) ;
street_ref = lpar_create_lst(LPAR_LST_REFQUERY);
lpar_append_lst(street_ref,
                lpar_create_str(LPAR_STR_QLETNAME, "Street Cat")) ;
location_ref = lpar_create_lst(LPAR_LST_REFQUERY);
lpar_append_lst(location_ref,
                lpar_create_str(LPAR_STR_QLETNAME, "Location Cat")) ;
phone_ref = lpar_create_lst(LPAR_LST_REFQUERY);
lpar_append_lst(phone_ref,
                lpar_create_str(LPAR_STR_QLETNAME, "Phone Cat")) ;
dob_ref = lpar_create_lst(LPAR_LST_REFQUERY);
lpar_append_lst(dob_ref,
                lpar_create_str(LPAR_STR_QLETNAME, "DOB Cat")) ;
/* And a query expression argument list using references. */
cat_ref_qrys = lpar_create_lst(LPAR_LST_QEXPR_ARGS) ;
lpar_append_lst(cat_qrys, name_ref) ;
lpar_append_lst(cat_qrys, street_ref) ;
lpar_append_lst(cat_qrys, location_ref) ;
lpar_append_lst(cat_qrys, phone_ref) ;
lpar_append_lst(cat_qrys, dob_ref) ;
/* Define Rule 1 Query */
/* ------------------- */
/* ... as before ... */
/* Define Rule 2 Query */
/* ------------------- */
/* Create the Query. */
rule_2_qry = lpar_create_lst(LPAR_LST_QEXPR) ;
/* Mark it as a Match Case Query. */
lpar_append_lst(rule_2_qry,
                lpar_create_int(LPAR_INT_QEXPR_TYPE,
                                LKT_QEXPR_MATCH)) ;
/* Add the arguments (we use the reference querylets) */
lpar_append_lst(rule_2_qry, cat_ref_qrys) ;
/* Core query weights are added as querylet weights. */
/* ... the rest as before ... */
```

Old-style Query Construction

This style of query construction is deprecated as of version 4.1. It remains only for the sake of reverse compatibility and to document functionality which previous users might still be using.

See Query Construction for more information on the currently recommended query formulation.

Note that not all of the pre version 4.1 forms and options are supported in version 4.1 and later. If you are upgrading from a pre-4.1 version it is likely that your old-style queries work as before. However, in some cases it might be necessary to modify your queries to meet the forms required by this version of the TIBCO Patterns servers.

In its simplest form, a query is a simple, contiguous block of text, held in an LPAR_BLK_SEARCHQUERY. Records are compared against this text and evaluated for similarity.

For more advanced searches, it might be desirable to have multiple, independent blocks of query text, with different search options for each block. When a query text block is combined with search options that pertain only to that text block it is referred to as a querylet. A querylet is a list type lpar, LPAR_LST_QUERYLET, which must contain exactly one LPAR_BLK_SEARCHQUERY, but might also contain any of the following additional options:

- LPAR_INTARR_SELECTFLDS selects the record fields against which this querylet is compared. Fields are specified by number with this option.
- LPAR_STRARR_FIELDNAMES selects the record fields against which this querylet is compared. Fields are specified by name with this option.
- LPAR_DBL_QUERYLETWEIGHT specifies a weighting factor for this querylet. The raw score is multiplied by this factor to obtain the final score for this querylet. This can be used to adjust the relative importance of each querylet.
- LPAR_DBLARR_QFIELDWEIGHTS specifies the weight for matched text against each field in the LPAR_INTARR_SELECTFLDS or LPAR_STRARR_FIELDNAMES array.

  The maximum weight is 1.0, with values less than 1.0 representing penalized matches.
- LPAR_INT_ALIGNFIELD The meaning and use of this field changed somewhat as of the 4.1 release. Previous to 4.1 it defined the starting point for alignment of the query string, that is which field of the merged field set the querylet string should be aligned over. As of 4.1 this defines which field is to be considered the cognate field for this querylet (see Cognate Query). Thus this value is only relevant to interdependent (described below) multi-queries or multiquerylets (which correspond to cognate queries in 4.1). This still indicates the field within the querylet, not in the database. In other words, if the field set consists of fields 1, 3, and 9, an ALIGNFIELD value of 1 is assigned field 3 as the cognate field for this querylet.
- LPAR_STR_THESAURUSNAME is the thesaurus to use for just this querylet. For example, this makes it possible to use one thesaurus for mapping St. to Street for an address field querylet, and St. to Saint for a City field querylet, allowing the correct recognition of Main St., St. Louis. In version 4.1 and later there is a restriction that all querylets of an interdependent multi-query or multiquerylet must use the same thesaurus. An attempt to define a different thesaurus for such interdependent querylets is rejected with a DVK_ERR_PARAMCONFLICT error.

Querylets must be combined into a multi-part query, or multi-query by placing them in an LPAR_LST_MULTIQUERY. This multi-query can then be used as the querypar for the dbsearch command.

For a concrete example, consider a database with the following fields:

- First Name
- Last Name
- Social Security Number
- House Number
- Street Name
- City

and a search application which provides the user three text entry boxes labeled Name, Address, and SSN.

```
lpar_t multiquery,querylet,sq,sf;
unsigned char *name,*address,*ssn;
unsigned char *name_fields[] =    { "First Name",
                                    "Last Name" };
unsigned char *ssn_fields[] =     { "Social Security Number" };
unsigned char *address_fields[] = { "House Number",
                                    "Street Name",
                                    "City" };
name    = get_name_box();
address = get_address_box();
ssn     = get_ssn_box();
multiquery=lpar_create_lst(LPAR_LST_MULTIQUERY);
/* Name querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,name,strlen(name));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,name_fields,2);
lpar_append_lst(querylet,sf);
lpar_append_lst(multiquery,querylet);
/* SSN querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,ssn,strlen(ssn));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,ssn_fields,1);
lpar_append_lst(querylet,sf);
lpar_append_lst(multiquery,querylet);
/* Address querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
```

```
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(querylet,sf);
lpar_append_lst(multiquery,querylet);
```

Here a multiquery containing three querylets has been built. Each querylet contains a SEARCHQUERY text block and a field selection array. Notice that there does not need to be a one to one mapping between record fields and querylets. When one querylet lists multiple fields in its field set, those fields are concatenated and the querylet text is compared against this concatenated field string.

In the above example, the querylets are said to be independent, because no two querylets share a common record field. When two querylets have the same or overlapping field sets, the querylets become interdependent. The match scores for interdependent querylets must be computed simultaneously, since these querylets belongs to the same bipartite graph.

When a multiquery contains interdependent querylets, the field sets for all querylets is joined and the query as a whole is compared with the union of all selected fields. (In other words, when there are any interdependent querylets, all querylets are considered to be interdependent, even if their specified field sets do not actually overlap the field sets of other querylets.) In version 4.1 and later this corresponds to a cognate query (see Cognate Query) and must conform to the cognate model. That is there must be a one to one mapping of querylets to fields in the combined field set.

Consider the previous example, but with the additional complication that the user now has four text boxes. The first and last names have been isolated into separate entries, but we still want to allow for queries or records where the first and last name have been accidentally transposed. The following code sets up a new query with interdependent querylets.

```
lpar_t multiquery,querylet,sq,sf,af,qfw;
unsigned char *fname,*lname,*address,*ssn;
unsigned char *name_fields[] =    { "First Name",
                                    "Last Name" };
double name_weights[2][2] = { { 1.0 , 0.8 } ,
                              { 0.8 , 1.0 } };
unsigned char *ssn_fields[] =     { "Social Security Number" };
unsigned char *address_fields[] = { "House Number",
                                    "Street Name",
                                    "City" };
double house_weights[3] = { 1.0, 0.8, 0.8 } ;
double street_weights[3] = { 0.8, 1.0, 0.8 } ;
double city_weights[3] = { 0.8, 0.8, 1.0 } ;
fname    = get_fname_box();
lname    = get_lname_box();
```

```
address = get_address_box();
ssn     = get_ssn_box();
multiquery=lpar_create_lst(LPAR_LST_MULTIQUERY);
/* First name querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,fname,strlen(fname));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,name_fields,2);
lpar_append_lst(querylet,sf);
as=lpar_create_int(LPAR_INT_ALIGNFIELD,0);
lpar_append_lst(querylet,af);
qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,name_weights[0],2);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquery,querylet);
/* Last name querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,fname,strlen(fname));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,name_fields,2);
lpar_append_lst(querylet,sf);
af=lpar_create_int(LPAR_INT_ALIGNFIELD,1);
lpar_append_lst(querylet,af);
qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,name_weights[1],2);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquery,querylet);
/* SSN querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,ssn,strlen(ssn));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,ssn_fields,1);
lpar_append_lst(querylet,sf);
lpar_append_lst(multiquery,querylet);
/* Address querylets */
/* house */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(querylet,sf);
qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,house_weights,3);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquery,querylet);
/* street */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(querylet,sf);
```

```
qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,street_weights,3);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquery,querylet);
/* city */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(querylet,sf);
qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,city_weights,3);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquery,querylet);
```

The first two querylets have the same field set, so the record score is calculated using a single bipartite graph using the six named record fields. These two querylets also include two additional optional parameters. The ALIGNFIELD specifies the cognate field for the querylet (see Cognate Query). In version 4.1 or later of the TIBCO Patterns servers the cognate field for a querylet must be specified either by the ALIGNFIELD parameter or by providing a QFIELDWEIGHTS parameter that has a single unique maximum field weight. (By definition a field set that consists of a single field has a single maximum field weight so it is not necessary to specify either value in that case.) The next parameter, QFIELDWEIGHTS, sets up a penalty factor for field transpositions. In version 4.1 or later these are not applied directly, but are combined to determine the non-cognate field weight. The non-cognate weight is taken as the average of all field weights for non-cognate fields in the field set. As in this example all non-cognate fields have a field weight of 0.8 the average is 0.8.

Note that the address value is matched three times. As there must be a one to one correspondence of querylets to fields there must be 3 address querylets as there are 3 address fields.

The field weights are needed to determine which querylet matches which field. Versions previous to 4.1 did not have this one to one correspondence restriction on interdependent querylets.

Even though the SSN and address querylets do not appear to be matched against the name fields they are as all querylets of the multi-query are merged into a single cognate query. This is probably not what is intended. In reality what is probably desired is a means of combining the scores of a cross match on the name fields with the scores of simple direct matches of the SSN querylet to the SSN field and the address querylet to the address fields. In current versions this can be expressed directly and flexibly using the AND score combiner. Previous to version 4.1 there was a more limited method for doing this using an additional layer of querylet grouping called a multiquerylet. A multiquerylet is used to explicitly specify querylet dependency. A multiquerylet is another list type lpar, LPAR_LST_MULTIQUERYLET, which contains a field set specification (either by name or number) and a collection of interdependent querylets.

Here is the same search done with multiquerylets.

```
lpar_t multiquery,multiquerylet,querylet,sq,sf,af,qfw;
unsigned char *fname,*lname,*address,*ssn;
unsigned char *name_fields[] =    { "First Name",
                                    "Last Name" };
unsigned char *ssn_fields[] =     { "Social Security Number" };
unsigned char *address_fields[] = { "House Number",
                                    "Street Name",
                                    "City" };
double name_weights[2][2] = { { 1.0 , 0.8 } ,
                              { 0.8 , 1.0 } };
fname    = get_fname_box();
lname    = get_lname_box();
address = get_address_box();
ssn     = get_ssn_box();
multiquery=lpar_create_lst(LPAR_LST_MULTIQUERY);
/* Name Multiquerlet */
multiquerylet=lpar_create_lst(LPAR_LST_MULTIQUERYLET);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,name_fields,2);
lpar_append_lst(multiquerylet,sf);
/* First name querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,fname,strlen(fname));
lpar_append_lst(querylet,sq);
as=lpar_create_int(LPAR_INT_ALIGNFIELD,0);
lpar_append_lst(querylet,af);
  qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,name_weights[0],2);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquerylet,querylet);
/* Last name querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,fname,strlen(fname));
lpar_append_lst(querylet,sq);
as=lpar_create_int(LPAR_INT_ALIGNFIELD,1);
lpar_append_lst(querylet,af);
  qfw=lpar_create_dblarr(LPAR_DBLARR_QFIELDWEIGHTS,name_weights[1],2);
lpar_append_lst(querylet,qfw);
lpar_append_lst(multiquerylet,querylet);
lpar_append_lst(multiquery,multiquerylet);
/* SSN querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,ssn,strlen(ssn));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,ssn_fields,1);
lpar_append_lst(querylet,sf);
lpar_append_lst(multiquery,querylet);
```

```
/* Address querylet */
querylet=lpar_create_lst(LPAR_LST_QUERYLET);
sq=lpar_create_blk(LPAR_BLK_SEARCHQUERY,address,strlen(address));
lpar_append_lst(querylet,sq);
sf=lpar_create_strarr(LPAR_STRARR_FIELDNAMES,address_fields,3);
lpar_append_lst(querylet,sf);
lpar_append_lst(multiquery,querylet);;
```

In this case, the explicit listing of the interdependency of the first and last name querylets by defining them as part of a single multiquerylet sets them apart from the SSN and address querylets, which are now computed independently. As the address querylet is now independent it is no longer bound by the one to one querylet to database field restriction so we need only specify a single querylet for it.

Predicate Expressions

Predicate expressions are used in two contexts: to filter records from a database while doing a search or to generate scores when evaluating the match strength of a record. For example, if each record has a field which contains a date, and you only want to search records with dates after a certain point in time, you could use a predicate expression to select these records.

Two methods are available for defining predicate expressions: through the use of an LPAR list or, starting with the 4.0 release of the TIBCO Patterns servers, via the use of an SQL like string format. The LPAR list format is described first.

Predicate lpar Expressions

The list-type lpar LPAR_LST_PREDICATE contains a predicate expression consisting of one, two, or three elements. Single-element predicates consist solely of a predicate component, which is either a fixed value, a reference to a record field, or another (nested) predicate expression. The following lpars are valid predicate components:

Lpar Id	Lpar contents
LPAR_BOOL_PREDVALUE	Fixed boolean value
LPAR_INT_PREDVALUE	Fixed integer value

Lpar Id	Lpar contents
LPAR_DBL_PREDVALUE	Fixed double value
LPAR_STR_PREDVALUE	Fixed string value
LPAR_BLK_PREDVALUE	Fixed byte block value
LPAR_BLKARR_PREDVALUE	Array of fixed byte block values
LPAR_INT_PREDFIELD	Reference to a record field by number
LPAR_STR_PREDFIELD	Reference to a record field by name
LPAR_LST_PREDICATE	Another predicate expression

Two-element predicate expressions consist of an LPAR_INT_PREDOP as the first element, and any predicate component as the second element. The value of the LPAR_INT_PREDOP must be one of the following:

LPAR_INT_PREDOP value	
PRED_OP_ABS	Absolute value of int or dbl
PRED_OP_MINUS	Arithmetic inverse of int or dbl
PRED_OP_NOT	Logical inverse of bool
PRED_OP_TOINT	Convert string/block/int/dbl/date/date-time to int
PRED_OP_TOBLK	Convert any type to blk
PRED_OP_TODBL	Convert string/block/int/dbl/date/date-time to dbl
PRED_OP_TODATE	Convert string/block/int/dbl/date/date-time to date
PRED_OP_TODATEEU	as above except it expects European day/month/year format

LPAR_INT_PREDOP value	
PRED_OP_TODATET	Convert string/block/int/dbl/date/date-time to date-time
PRED_OP_TODATEEUT	as above except it expects European day/month/year format
PRED_OP_TOKENIZE	Convert a blk or string into a block array of white space separated words.
PRED_OP_ARGS_CREATE	Create an argument list with one element. For information on argument lists, see Predicate Functions and Argument Lists.
PRED_OP_FUNC_IF	Conditional expression. For more details on this predicate function, see Predicate Functions and Argument Lists.
PRED_OP_FUNC_GEOD	Compute the distance between two points. For more details on this predicate function, see Predicate Functions and Argument Lists.
PRED_OP_FUNC_TOSCORE	Normalize a value into a 0.0 to 1.0 score value. For more details on this predicate function, see Predicate Functions and Argument Lists.

Three-element predicate expressions consist of any predicate component as the first element, an LPAR_INT_PREDOP as the second element, and any predicate component as the third element. The value of the LPAR_INT_PREDOP must be one of the following:

LPAR_INT_PREDOP value	
PRED_OP_PLUS	Arithmetic addition of ints, doubles, concatenate strings, blocks
PRED_OP_MINUS	Arithmetic subtraction of ints or doubles
PRED_OP_TIMES	Arithmetic multiplication of ints or doubles

LPAR_INT_PREDOP value	
PRED_OP_DIVIDEDBY	Arithmetic division of ints or doubles
PRED_OP_TOTHE	Arithmetic exponentiation of ints or doubles
PRED_OP_AND	Logical and of bools
PRED_OP_OR	Logical or of bools
PRED_OP_EQUALS	Arithmetic or string comparison of ints, dbls, strs, blks, dates
PRED_OP_iEQUALS	Letter case insensitive comparison of strs or blks
PRED_OP_LESSTHAN	Arithmetic or string comparison of ints, dbls, strs, blks, dates
PRED_OP_iLESSTHAN	Letter case insensitive comparison of strs or blks
PRED_OP_LESSTHANOREQ	Arithmetic or string comparison of ints, dbls, strs, blks, dates
PRED_OP_iLESSTHANOREQ	Letter case insensitive comparison of strs or blks
PRED_OP_GREATERTHAN	
PRED_OP_iGREATERTHAN	Letter case insensitive comparison of strs or blks
PRED_OP_ GREATERTHANOREQ	Arithmetic or string comparison of ints, dbls, strs, blks, dates
PRED_OP_ iGREATERTHANOREQ	Letter case insensitive comparison of strs or blks
PRED_OP_ISIN	Substring match of strings or blks
PRED_OP_iISIN	Letter case insensitive substring match

LPAR_INT_PREDOP value	
PRED_OP_SUPERSET	Set comparison of blkarrs
PRED_OP_SUBSET	Set comparison of blkarrs
PRED_OP_TOKENIZE	Split a block or string into words using a specified separator
PRED_OP_ARGS_APPEND	Append a value to an argument list. See Predicate Functions and Argument Lists for more details.
PRED_OP_SW	Check if one string starts with another
PRED_OP_iSW	Check if one string starts with another (case-insensitive)
PRED_OP_EW	Check if one string ends with another
PRED_OP_iEW	Check if one string ends with another (case-insensitive)
PRED_OP_LIKE	Check if one string is “like” another, similar to SQL, takes percent (%) as wildcard matching zero or more characters, and underscore (_) as matching exactly one character.
PRED_OP_iLIKE	As PRED_OP_LIKE, but case-insensitive.
PRED_OP_REGEX_MATCH	Check if one string “matches” another as a regular expression.
PRED_OP_iREGEX_MATCH	As PRED_OP_REGEX_MATCH, but case-insensitive.
PRED_OP_ISBLANK	Check if a value consists only white-space and punctuation.

Each of these operators expects specific value types as arguments; these are validated when the TIBCO Patterns server receives the expression as part of search command. A DVK_ERR_PARAMVAL error is returned if the expression is invalid because the arguments are of an invalid type. But note that many operators accept a number of different argument types. For instance, the PRED_OP_PLUS operator accepts an integer and double values.

To aid the construction of two and three component predicate lpars, two additional DevKit functions are available:

```
lpar_t lpar_create_predicate2( int operator, lpar_t predicate );
lpar_t lpar_create_predicate3( lpar_t left, int operator,
                               lpar_t right );
```

To create a predicate to search only those records in a database dated more recently than January 3, 2002, where the date is stored in a field named "`Creation Date`" you could use the following:

```
lpar_t predicate;
predicate = lpar_create_predicate3(
              lpar_create_predicate2(
                PRED_OP_TODATE,
                lpar_create_str(LPAR_STR_PREDFIELD,"Creation Date")),
              PRED_OP_GREATERTHAN,
              lpar_create_predicate2(
                PRED_OP_TODATE,
                lpar_create_str(LPAR_STR_PREDVALUE,"January 3, 2002")));
```

The TODATE operator accepts the same date formats as the DATE field type (see The TIBCO Patterns Table) Dates hold only date values, you might specify a time value but it is ignored. If you wish to have resolution down to the second you can use the TODATET operator which keeps both time and date. Dates of the form xx/xx/xx are interpreted in the U.S. style (mm/dd/yy), not the European style (dd/mm/yy) unless the TODATEEU or TODATEEUT operator is used.

The range of valid dates is from 100 AD (years below 100 are interpreted as the last two digits of the current century if less than or equal to the current year, or the last two digits of the previous century if greater than the current year) to 9999 AD.

If the Creation Date field was specified as a LKT_FLD_TYPE_DATE field when the database was created then it is not necessary to convert the field to a date value as it is already stored as a date. The above example then becomes:

```
lpar_t predicate;
predicate = lpar_create_predicate3(
                         lpar_create_str(LPAR_STR_PREDFIELD, "Creation
Date"),
                         PRED_OP_GREATERTHAN,
                         lpar_create_predicate2(
                                  PRED_OP_TODATE,
                                     lpar_create_str(LPAR_STR_
PREDVALUE,"1/3/2002")));
```

See The TIBCO Patterns Table for an explanation of field types.

Text comparisons can be performed against the raw string values or using the letter case insensitive operators. These are new with release 4.2. They replace the old TOUPPER and TOLOWER operators. With Unicode support TOUPPER and TOLOWER are both insufficient and ambiguous. The case insensitive comparison operators apply the Unicode standard case folding and normalization rules before comparing the strings. These include stripping diacritic marks and other letter normalization operations. Note that the LESSTHAN, GREATERTHAN operators are based on the Unicode code point values. They DO NOT apply any language specific lexical ordering rules.

Predicate Functions and Argument Lists

As described in the previous section, Predicate lpar Expressions, predicate expressions consist of constant or field values, unary operators, and binary operators. There are some operations that need more than just one or two inputs. Consider determining the distance between two points on the Earth's surface. To do so we need the latitude and longitude of the first point, plus the latitude and longitude of the second point; a total of four values. Predicate expressions handle this by assembling the four values into a special type called an *argument list*. The distance operation is then implemented as a unary operator that accepts a value of the special type "argument list". Unary operators that accept a value of type argument list are called *predicate functions*.

An argument list is created using the PRED_OP_ARGS_CREATE unary operator. This creates an argument list with one value. Additional values are appended to the argument list using the PRED_OP_ARGS_APPEND binary operator. The left side of this operator is an argument list, the right side is any value, and the result is a new argument list with the value added to the end of the argument list. An example of creating an argument list for the distance function is given below. The distance function actually takes 5 arguments, the fifth argument is the units of measure for the output value.

```
/*
 * Create call to geo-distance function.
 */
/* create the argument list with the first argument. */
lpar_t geo_args = lpar_create_predicate2(PRED_OP_ARGS_CREATE,
                                                                                lpar
PREDVALUE,
                                                          45.0)) ;
/* append each of the next arguments */
geo_args = lpar_create_predicate3(geo_args, PRED_OP_ARGS_APPEND,
                                  lpar_create_dbl(LPAR_DBL_PREDVALUE, 75.0));
geo_args = lpar_create_predicate3(geo_args, PRED_OP_ARGS_APPEND,
                                  lpar_create_str(LPAR_STR_PREDFIELD,
                                                  "latitude")) ;
geo_args = lpar_create_predicate3(geo_args, PRED_OP_ARGS_APPEND,
                                  lpar_create_str(LPAR_STR_PREDFIELD,
                                                  "longitude")) ;
geo_args = lpar_create_predicate3(geo_args, PRED_OP_ARGS_APPEND,
                                  lpar_create_str(LPAR_STR_PREDVALUE,
                                                  "miles")) ;
/* now create the geo-distance call. */
lpar_t geo_distance = lpar_create_predicate2(PRED_OP_FUNC_GEOD, geo_args) ;
```

The order in which arguments are appended is critical. Each predicate function expects a specific set of arguments in its argument list in a specific order.

In the example above all of the arguments are simple values, but they can be any predicate expression. For example, to use the distance value returned by the above predicate expression to score records, you need to convert it from a distance measure to a score in the range of 0.0 to 1.0. This can be done using the PRED_OP_FUNC_TOSCORE predicate function. This expects 3 values: the raw value to be converted, a value that represents the zero score, and a value that represents the 1.0 score. Using the geo_distance value from the above example, this call is created as follows:

```
/*
 * Now convert distance to a score value.
 */
/* create the argument list for the to score function. */
lpar_t toscore_args = lpar_create_predicate2(PRED_OP_ARGS_CREATE,
                                             geod_distance) ;
toscore_args = lpar_create_predicate3(toscore_args, PRED_OP_ARGS_APPEND,
                                      lpar_create_dbl(LPAR_DBL_PREDVALUE,
                                                      20.0)) ;
toscore_args = lpar_create_predicate3(toscore_args, PRED_OP_ARGS_APPEND,
                                      lpar_create_dbl(LPAR_DBL_PREDVALUE,
                                                      0.0)) ;
/* call the function. */
lpar_t toscore_func = lpar_create_predicate2(PRED_OP_FUNC_TOSCORE,
toscore_args) ;
```

As mentioned previously, a predicate function is a unary operator that accepts an argument list as its operand. That it has received an argument list, and not, for example, an integer value, is a syntax check that is performed when the expression is compiled; this triggers a DVK_ERR_PARAMVAL error. In addition, each predicate function operator verifies that its argument list contains the number and type of arguments it expects. A DVK_ERR_PARAMVAL error is returned if this validation fails.

The following are descriptions of each of the predicate functions:

PRED_OP_FUNC_GEOD

This function computes the distance between two points on the Earth's surface. It takes the following five arguments:

1. The first argument is the latitude of the first point. It is expressed in degrees as a floating point number.
2. The second argument is the longitude of the first point. It is expressed in degrees as a floating point number.
3. The third argument is the latitude of the second point. It is expressed in degrees as a floating point number.
4. The fourth argument is the longitude of the second point. It is expressed in degrees as a floating point number.
5. The fifth argument defines the units to be used. The units supported are either miles or kilometers. The argument value is a string value and must be one of the following recognized strings (letters are not case sensitive): "**km**", "**mi**", "**mile**", "**miles**", "**kilometer**", "**kilometers**".

PRED_OP_FUNC_TOSCORE

This function is used to convert a raw value to a value in the valid score range of 0.0 to 1.0. A raw value that is to be considered the 0.0 score and a raw value that is to be considered a 1.0 score are identified. Raw values in between the 0.0 score and the 1.0 score are converted using linear interpolation. Raw values outside the range are converted to either 0.0 or 1.0. The 0.0 score value might be greater than the 1.0 score value. In this case, the interpolation is reversed. Lower values get higher scores than higher values. For example, if you wish to convert a distance to a score, where anything more than 20 miles away is a non-match, and anything 0.0 miles away is a perfect match, the zero score value would be 20.0 and the 1.0 score value would be 0.0. All arguments must be of type double. The following are the arguments:

6. The raw value.
7. The zero score value.
8. The 1.0 score value

PRED_OP_FUNC_IF

This function can be thought of as an extended version of the “C” ternary operator. It is used to select one of a set of possible values based on conditional expressions.

Unlike the other functions, this one accepts a variable number of arguments, but the arguments must follow a specific set of rules. The arguments come in pairs with a final singleton argument. The first argument of the pair must evaluate to a Boolean value. The second argument of the pair might have any value type with the restriction that all second arguments, and the final singleton argument, must have the same value type. Zero or more argument pairs are allowed.

The value returned by this expression is the value of the second argument of the first pair for which the first argument evaluates to true. If no such pair exists, the value of the last, singleton, argument is returned. For example:

The following example returns “good comparison”:

{ 0 > 10, “bad comparison”, 10 > 0, “good comparison”, “no comparison” }

But this example returns "ok math":

{ 0 > 10, “bad comparison”, 2*5 > 3*5, “bad math”, “ok math” }

Controlling Predicate Error Handling

To be used as a filtering predicate in a search a predicate expression must evaluate to a Boolean value (true or false). For a query predicate the requirements are much more lenient, the predicate value need only be convertible to a Double. This includes Boolean values (by far the most common case) where true equals 1.0 and false equals 0.0, but can be any other value except for the array values.

If a predicate expression which does not produce a Boolean value is passed to the lkt_dbsearch command as a filtering predicate in the search options, or a query predicate is given which cannot be converted to a Double value, the DevKit responds with a PREDTYPE error.

For example, the predicate expression below is not a valid expression for use as a filtering predicate in a search:

```
predicate = lpar_create_predicate2(
              PRED_OP_TODATE,
              lpar_create_str(LPAR_STR_PREDFIELD,"Creation Date"));
```

This predicate expression never produces a Boolean result, so it is an error.

For a given record, even well-formed predicates might fail to produce any value at all depending on the record's contents. Consider the following predicate expression where the field Clicks is an integer field:

```
predicate = lpar_create_predicate3(
              lpar_create_str(LPAR_STR_PREDFIELD,"Clicks"),
              PRED_OP_GREATERTHAN,
              lpar_create_str(LPAR_INT_PREDVALUE,50));
```

This expression has the potential to evaluate to a Boolean value, so it is not rejected by lkt_dbsearch, but to evaluate correctly there must be an integer value in the field Clicks. If a record was loaded with the value 123 in this field, it evaluates to true, if it was loaded with 21 it evaluates to false, but if it was loaded with the value "orange" it does not evaluate to anything as the field value is invalid. Likewise if the field contains no value at all it has a special value "empty" and a valid comparison cannot be made.

By default, if a predicate expression fails to evaluate for a given record, that record is rejected from the search. In other words, the predicate expression selects only those records which evaluate to true, and the search is performed on only those records.

In some cases for a filtering predicate it might be desirable to include records which fail evaluation in the search. In this case, the search parameters LPAR_BOOL_FAILBADPRED and LPAR_BOOL_FAILEMPTYPRED can be used. Setting LPAR_BOOL_FAILBADPRED to false causes all evaluations which encounter an invalid value to be considered true. That is any records that contain fields that are tested by the predicate with invalid values are returned.

Setting LPAR_BOOL_FAILEMPTYPRED to false causes empty field values to be treated as valid. Predicate evaluation continues and depending on the results of the evaluation the record might or might not be included. Text fields are simply used as is (a zero length string or text block), numeric fields are given a value of zero, and date or date-time field have a special empty value. Empty dates compare less than all valid dates, greater than all invalid dates and equal to all empty dates.

Note that LPAR_BOOL_FAILBADPRED controls the behavior of predicates that fail for any reason. Thus if LPAR_BOOL_FAILEMPTYPRED is false and LPAR_BOOL_FAILBADPRED is true (the default), then any record where the predicate encounters an empty field value is returned.

For a query predicate, a predicate that fails to evaluate is assigned a score of 0.0 by default (in Learn queries the default value is -1.0 which has special meaning to the TIBCO Patterns Learn Model). This can be changed using the LPAR_DBL_INVALID_DATA_SCORE or LPAR_DBL_EMPTY_DATA_SCORE search parameters. Queries that encounter an empty field value is assigned the score given by the LPAR_DBL_EMPTY_DATA_SCORE parameter.

Predicates that fail evaluation for any other reason is assigned the score given by the LPAR_DBL_INVALID_DATA_SCORE parameter. In both cases the value must be in the range: 0.0 - 1.0 inclusive or the special value -1.0. The -1.0 special value indicates the record should be rejected from further consideration regardless of the overall record score (in Learn queries the -1.0 score does not cause the record to be rejected, instead it is an indicator that this data was not available).

Predicate String Expressions

For predicates constructed on the fly the LPAR form described in the previous section might be appropriate, but the string format described here might be more convenient to use and easier understand in other cases. The string form is passed in using the string type lpars LPAR_STR_PREDICATE or block type LPAR_BLK_PREDICATE. These lpars can be used anywhere an LPAR_LST_PREDICATE lpar can be used. So the above example of a predicate to restrict searches to records dated after January 3, 2002 could also be expressed as:

```
lpar_t predicate;
predicate = lpar_create_str(LPAR_STR_PREDICATE,
            "$\"Creation Date\" > DATE \"January 3, 2002\""
);
```

The predicate block form (LPAR_BLK_PREDICATE) is identical to the string form. It is provided as a means of handling very long expressions. An LPAR string is limited to about 1,000 characters in length, a block has no length restrictions. The null-terminated string form is easier to work with however so makes more sense for most cases. The block form can be used anywhere the string form is used even if it is not explicitly mentioned in the documentation. So anywhere in this documentation where LPAR_STR_PREDICATE is mentioned it should be understood that LPAR_BLK_PREDICATE can also be used.

A predicate string is similar to an SQL where clause expression. There are unary and binary operators and data elements. A data element is a field of a record or a string, block, numeric or Boolean constant. The data elements are:

Examples	Description
?TRUE?, ?FALSE?	Boolean constant values are true and false enclosed in question marks. Letter case insensitive.
123, 0777, 0x8FFF	Integer constant values. They follow the standard "C" conventions for decimal, octal and hexadecimal integers. Values must be within the defined range for integers.
123.45, 0.17e-10	Floating point values. They follow the standard "C" conventions for fixed and scientific notation values. Values must be within the defined range for "C" doubles.
"string value"	String constants are enclosed in double quotes. The basic XML/HTML entity encoding scheme is used to represent the double quote character

Examples	Description
	itself (i.e. `&quot;`) and other special characters. The numeric conventions: `&#ddd;`, `&#xhh;` are recognized and the entity names: quot, amp, lt, gt and apos are recognized. No other entity names are recognized. Note that you must NOT insert an encoded NULL character into the string. The string is converted to a standard "C" NULL terminated string, inserting a NULL effectively terminates the string at that point.
:"byte block"	A byte block is specified by preceding a quoted string value with a colon character. The length of the block is computed automatically. As with strings all non-valid string characters must be encoded using the XML/HTML like encodings.
#2	Table fields can be specified by numeric position. An integer value preceded with the pound (#) character is used to represent the record field at the indicated column position. Like array indexes in "C" these field numbers are zero based. That is #0 refers to the first field of the record.
$"first name"	Table fields can be specified by field name. A quoted string value (see string description above) preceded by the dollar sign ($) is used to specify the field name. Variable Attribute qualifiers and table name qualifiers are allowed.
[:"block 1", :"block 2"] or [:]	A comma separated list of blocks enclosed in square brackets is a block array. To specify a block array with zero entries, use an open bracket followed by a colon close bracket (:]) no space between the colon and close bracket.

The table below shows the unary operators and the equivalent LPAR_INT_PREDOP operator described above. In many cases two or more synonyms are provided for the same operator. All alphabetic characters in operator names are not letter case sensitive for example, DATE, date, and Date are considered to be the same operator.

string value	LPAR_INT_PREDOP
int	PRED_OP_TOINT

string value	LPAR_INT_PREDOP
dbl, double, float	PRED_OP_TODBL
date	PRED_OP_TODATE
date_time, datet	PRED_OP_TODATET
eudate, dateeu	PRED_OP_TODATEEU
eudate_time, dateeu_time, eudatet, dateeut	PRED_OP_TODATEEUT
blk, block	PRED_OP_TOBLK
-	PRED_OP_MINUS
+	none, this does nothing
not	PRED_OP_NOT
split, tokenize	PRED_OP_TOKENIZE
abs	PRED_OP_ABS
geod, geodistance, geo_distance	PRED_OP_FUNC_GEOD
if	PRED_OP_FUNC_IF
to_score, toscore	PRED_OP_FUNC_TOSCORE

The binary operators are listed below. As with the unary operators named operators are letter case insensitive.

string value	LPAR_INT_PREDOP
+	PRED_OP_PLUS
-	PRED_OP_MINUS

string value	LPAR_INT_PREDOP
*	PRED_OP_TIMES
/	PRED_OP_DIVIDEDBY
**	PRED_OP_TOTHE
and	PRED_OP_AND
or	PRED_OP_OR
=, ==	PRED_OP_EQUALS
~=, ~==	PRED_OP_iEQUALS
<	PRED_OP_LESSTHAN
~<	PRED_OP_iLESSTHAN
<=	PRED_OP_LESSTHANOREQ
~<=	PRED_OP_iLESSTHANOREQ
>	PRED_OP_GREATERTHAN
~>	PRED_OP_iGREATERTHAN
>=	PRED_OP_GREATERTHANOREQ
~>=	PRED_OP_iGREATERTHANOREQ
in	PRED_OP_ISIN
i_in	PRED_OP_iISIN
superset	PRED_OP_SUPERSET
subset	PRED_OP_SUBSET

string value	LPAR_INT_PREDOP
split	PRED_OP_TOKENIZE
tokenize	PRED_OP_TOKENIZE
startswith	PRED_OP_SW
i_startswith	PRED_OP_iSW
endswith	PRED_OP_EW
i_endswith	PRED_OP_iEW
like	PRED_OP_LIKE
i_like	PRED_OP_iLIKE
matches	PRED_OP_REGEX_MATCH
i_matches	PRED_OP_iREGEX_MATCH
isblank, is_blank	PRED_OP_ISBLANK

All unary operators have higher precedence than binary operators. That is: BLOCK ABS $"count1" - $"count2" is equivalent to:

```
( BLOCK ( ABS $"count1" ) ) - $"count2"
```

probably not what was intended. Parenthesis can be used to alter the default precedence relations:

```
BLOCK ABS ( $"count1" - $"count2" )
```

The binary operators are left associative, e.g.

```
1 + 2 + 3
```

is implemented as:

```
( 1 + 2 ) + 3
```

They have the standard precedence relations with the following caveats:

1. + has slightly higher precedence than - (minus), thus a - b + c is a - (b + c) not (a - b) + c
2. similarly * has slightly higher precedence than / (divide)
3. All comparison operators have the same precedence.

The binary operators listed by precedence from highest to lowest are:

4. **
5. *
6. /
7. +
8. -
9. tokenize, split
10. =, ~=, ==, ~==, <, ~<, <=, ~<=, >, ~>, >=, ~>=, in, i_in, superset, subset
11. and
12. or

There are no operators for constructing argument lists in the string format. This is because the string format provides a syntactic means of specifying argument lists directly. An argument list is specified as an open curly brace ({) followed by a comma separated list of expressions, followed by a close curly brace (}). For example, the following is how the geo-distance function can be expressed:

geod { $"latitude", $"longitude", 45.0, 75.0, "miles" }

As function predicates are unary operators, they bind to their argument list at the highest precedence. Thus the expression:

geod { $"latitude", $"longitude", 45.0, 75.0, "miles" } ** 2

is the distance squared, and not the argument list squared.

Examples:

- Include only records of males born on or after January 1st, 1980:

```
( $"sex" ~= "m" OR $"sex" ~= "male" ) AND $"birth date" >= date "1/1/1980"
```

- Include only parts of category "tool" and an average price less than $10:

```
$"category" ~= "tool" AND ( $"max price" + $"min price" ) / 2.0 < 10.0
```

- Include only those people whose previous and current weight differ by less than 5 lbs.

```
ABS ( $"cur weight" - $"prev weight" ) < 5.0
```

- Return a score for records within 20 miles of 45.0 degrees latitude and 75.0 degrees longitude.

```
toscore { geod { $"latitude", $"longitude", 45.0, 75.0, "miles" },
20.0, 0.0 }
```

Predicate Value Data-type Conversions

Predicates do not support implicit data-type conversions. You must perform an explicit conversion while performing a binary operation with operands of different types. For example, if the `DOB` field is of type `DATETTIME` then the predicate expression `DATE \"2001/01/01\" <= $\"DOB\` produces a PARAMVAL error. An explicit conversion `DATE \"2001/01/01\" <= DATE $\"DOB\` is required.

This applies to both predicate-expressions and predicate-string-expressions.

Weighting Fields

Field weights allow you to alter the importance of fields in a record or querylet.

Field weights can be attached to an LPAR_LST_SIMPLEQUERY, LPAR_LST_COGQUERY, LPAR_LST_ATTRQUERY or LPAR_LST_QUERYLET inside an old style multiquery. These field weights, contained in an LPAR_DBLARR_QFIELDWEIGHTS, are double precision floating point numbers in the range of 0.0 (no weight) to 1.0 (maximum weight). Each element of this array corresponds one to one to the selected fields for that individual querylet. Weights for non-participating fields should not be included.

Querylet specific field weights allow you to adjust the importance of fields on a querylet by querylet basis. For old style multi-queries this was generally used to set the cognate vs. non-cognate field weighting (see Cognate Query and Old-style Query Construction for more details on this). For new style simple and cognate queries this provides a more flexible means of setting the importance of fields if you should need it.

Field weights behave somewhat differently depending on which query structure they are applied to. For a simple query field weights are penalizing, that is they reduce the overall score of the match. For cognate and attribute queries field weights are structural, that is they change the relative importance of matches within the fields, but do not actually penalize the final score. For a simple query if a query value matches in a field with a weight of 0.8, then the highest score that match can have is 0.8, even if it is a perfect match. With a cognate query or an attributes query a perfect match in a field with a weight of 0.8 still receives a final score of 1.0. The following example might help in understanding what the different kinds of weights do.

We run a query against the record:

first	last
Bob	Taf

We use both a simple query (penalizing weights) and a cognate query (structural weights). The results for two different queries with various combinations of weights are:

Query	Query Type	Weights (first,last)	Score
Bob Taf	Simple	1.0, 1.0	1.0
Bob Taf	Simple	0.5, 1.0	0.75
Bob Taf	Simple	1.0, 0.5	0.75
Bob Taf	Cognate	1.0, 1.0	1.0
Bob Taf	Cognate	0.5, 1.0	1.0
Bob Taf	Cognate	1.0, 0.5	1.0
Jim Taf	Simple	1.0, 1.0	0.5
Jim Taf	Simple	0.5, 1.0	0.5

Query	Query Type	Weights (first,last)	Score
Jim Taf	Simple	1.0, 0.5	0.25
Jim Taf	Cognate	1.0, 1.0	0.5
Jim Taf	Cognate	0.5, 1.0	0.66667
Jim Taf	Cognate	1.0, 0.5	0.33333

The first thing to note is that even with an exact match penalty weight as used by a simple query lowers the score, but with structural weights as used by the cognate query an exact match always has a 1.0 score.

In the second example, we have a complete mismatch on the first name field and a perfect match on the last name field. With equal weights both simple and cognate queries have a score of 0.5.

Note: Remember scores are not a measure of the amount of the query that has been matched, this example is carefully constructed to make scores come out even to make the effects of weights clearer. Do not expect scores on real data to be so even.

With the penalty weights of the simple query lowering the weight of the first name field has no effect on the score as it is already zero. But with the structural weights of the cognate query we see that the score is raised when the weight of the first name field is lowered. With a weight of 0.5 on the first name and 1.0 on the last name the last name now represents two thirds of the query, so the score is now two thirds of the perfect score of 1.0.

If we reverse the weighting, in the simple query the perfect match on last name is now getting penalized by a factor of 0.5, and the score on the first name is still zero, so the final score is reduced to 0.25. For the structural weights the last name now represents only one third of the query so the score is reduced to just one third of a perfect match.

Why do we have different types of weights? With a cognate query we fully expect that every field is matched. The field set represents parts of a whole. By using weights we adjust the relative importance of the parts, but we do not want to penalize a score for matching a part. Simple queries over multiple fields are often used where it is unclear which field is to be matched. The fields do not necessarily represent parts of a whole, they might represent alternatives, each field potentially being a whole in itself. In this case we might prefer matches in one field over another, thus we penalize the less preferred fields. The same arguments hold for the AND vs. the OR score combiners. The AND is parts of a whole, so the weights are structural weights, the OR is alternatives, so penalty weights apply.

Below is a table of the different weights and whether they are structural or penalizing.

Weight	Type
LPAR_DBLARR_QUERYLETWEIGHTS (AND)	Structural
LPAR_DBLARR_QUERYLETWEIGHTS (OR)	Penalizing
LPAR_DBL_THESAURUSWEIGHT	Penalizing
LPAR_DBLARR_QFIELDWEIGHTS (Simple Query)	Penalizing
LPAR_DBLARR_QFIELDWEIGHTS (Cognate Query)	Structural
LPAR_DBLARR_QFIELDWEIGHTS (Attributes Query)	Structural
LPAR_DBL_NONCOG_WGT	Penalizing

Character Mapping

Character maps are used to normalize text data before comparisons are performed. Normalization consists of one or more of:

- **Fold letter case:** This applies the Unicode Consortium defined rules for letter case folding to map all alphabetic characters to a common letter case.
- **Fold diacritics:** This applies the Unicode Consortium defined rules for stripping letters of their diacritic marks and other character normalization.
- **Special Character mappings:** Using this a particular character or class of characters to be mapped to another character. Currently there are two character classes defined: whitespace and punctuation. The definition of these classes is as specified by the Unicode Consortium with the exception of characters in the ASCII range, where all non-alphanumeric characters except the standard white space characters are considered punctuation characters.

The precedence of mappings, from lowest to highest is:

1. letter case foldings
2. diacritic folding and character normalization
3. character class mapping
4. explicitly defined mappings.

Thus it is possible to override letter folding and normalization or character class mappings using explicit character maps. For example, if we wish to create a character map that folds all letters to a common letter case except for 'A', and maps all punctuation to blank except for ampersand we can do this by adding a special mapping that maps ’A’ to ’A’ and ’&’ to ’&’.

Character maps must be created and assigned a name before they can be used. Once created they cannot be deleted or updated.

The predefined character maps are:

- DVK_CMAP_STDNAME is the standard character mapping applied by default. It maps all forms of whitespace to the blank character (code point 0x0020), all punctuation characters except the ampersand to the blank character (code point 0x0020), folds letter case and folds diacritics (normalizes characters). For a specific list of the punctuation code-points mapped to the blank character, see Punctuation and Whitespace Code Points Mapped by Built-in Character Maps.
- DVK_CMAP_PUNCTNAME is a punctuation sensitive character map. It is the same as DVK_CMAP_STDNAME except that punctuation characters are not mapped (remain as themselves).

To create your own custom character map use the `lkt_create_charmap` function. You can list all existing character maps, or check for the existence of a particular character map using the `lkt_charmap_list` function.

Creating a Custom Character Map (lkt_create_charmap)

The following functions create a new custom character map. Character maps created under a user transaction might not be used in table creation outside that transaction until the transaction is committed.

```
dvkerr_t lkt_create_charmap(lpar_t host, lpar_t mapname, lpar_t mapdef );
dvkerr_t lkt_create_charmapT(lpar_t host, lpar_t mapname, lpar_t tran,
                             lpar_t mapdef );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

mapname (required)

is the name of the character map. It is specified using (LPAR_STR_CHARMAP).

mapdesc (optional)

If given this must be a list containing one or more of:

- LPAR_INT_FOLDCLASS must be one of the following:
  - LKT_CCLASS_CASE fold letter case
  - LKT_CCLASS_DIACRITICS fold diacritic marks and normalize characters.
- LPAR_LST_MAP defines a special character mapping. This list consists of exactly two items: a from character set and a to character set. The from character set is defined using one of three LPARs:
- LPAR_INT_FROM_CLASS specifies that the from character set is a predefined class of characters. The currently defined classes that might be used are: LKT_CCLASS_ WHITESPACE all white space characters, and LKT_CCLASS_PUNCTUATION all punctuation characters.
- LPAR_BLK_FROM_CHARS contains a list of UTF-8 encoded characters that is the from character set.
- LPAR_INTARR_FROM_UCODES contains a list of characters specified as Unicode Code Points.

The to character set is defined using one of two LPARs:

- LPAR_BLK_TO_CHARS contains a list of UTF-8 encoded characters that is the to character set.
- LPAR_INTARR_TO_UCODES contains a list of characters specified as Unicode Code Points.

  If the from character set is defined as a class of characters the to character set must be a single character, otherwise it must be either a single character or have exactly the same number of characters as the from set. If the to character set is a single character all characters in the from set are mapped to that character, otherwise each character in the from set is mapped to the corresponding character in the to set.

The special value: LKT_CHAR_DELETE might be included as a Unicode Code Point in the to character set. This indicates that the from character is to be deleted instead of mapped.

Any number of entries might be given in the mapdesc list.

Default value: No mapping is performed.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the charmap is to be created

Error codes and items returned by lkt_create_charmap

ARRAYLEN	array item with incorrect number of entries
CHARCONV	item with invalid UTF-8 encoded data
DBEXISTS	name of character map that already exists
HANDSHAKE	(none)
INTERNAL	(none)
IPCERR	(none)
IPCTIMEOUT	(none)
IPCEOF	(none)
PARAMCONFLICT	item which conflicts with an earlier item
PARAMMISSING	(none) or list with missing entry
PARAMVAL	item with invalid value
SRCHPARAM	unrecognized LPAR in map definition parameters
TRAN_UNKNOWN	lpar that contains the unknown transaction id

TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ERRORDETAILS

Checking the Status of Character Maps (lkt_ charmap_list)

```
The following command returns a list of the names of all existing character
maps or checks for the existence of a particular character map or maps.
dvkerr_t lkt_charmap_list(lpar_t host, lpar_t names, lpar_t *ret_stats );
```

Input

names (optional)

is a single character map name (LPAR_STR_CHARMAP) or a list of character map names of existing character maps. It is an error (DVK_ERR_DBNOTFOUND) if any of these character maps do not exist.

A value of LPAR_NULL might be supplied for names. This causes all of the names of the currently created character maps to be returned.

Output

ret_stats (optional)

is a list of lists, one list for each character map. Each list contains the character map name as an LPAR_STR_CHARMAP.

- LPAR_LONGINT_TRAN_ID is the transaction id of the transaction that owns or claims the object at that instance.
- LPAR_INT_TRAN_OBJ_STATE is the state of an object when a transaction owns or claims it; if there is no such transaction, it returns LKT_TRAN_OBJ_EXISTS.

Error codes and items returned by lkt_charmap_list

DBNOTFOUND	first item that isn't an existing character map
EXPECTDBDESC	item that should have been a character map name
EXPECTLIST	item that should have been a list
FEATURESET	list with unsupported item or combination of items
NODBDESC	the list or item that was missing a character map name
NOSYSINIT	(none)

Creating a Custom Character Map: Example

The following code sample creates a character map similar to the standard character map except that the characters ’&’ and ’#’ are not mapped, the dash character ’-’ is deleted and the default case folding for the letter ’I’ is changed to that for Turkish. We start by checking to see if our character map is already created.

```
static char map_name = "our-turkic-map" ;
static char from_set = "&#-I" ;
/* ampersand, pound,  delete it,       lower i no dot */
static int  to_set = { 0x0026,  0x0023, LKT_CHAR_DELETE, 0x0131 } ;
lpar_t      mapname ;     /* name of our character map */
lpar_t      mapdef ;      /* definition parameters for creating map */
lpar_t      mapping ;     /* one mapping clause in mapdef */
dvkerr_t    err ;         /* return status of devkit function calls */
/* create descriptor for our character map */
mapname = lpar_create_str(LPAR_STR_CHARMAP, map_name) ;
/* create only if doesn't already exist */
err = lkt_charmap_list(mapdesc, NULL) ;
if (DVKERR(err) == DVK_ERR_DBNOTFOUND) {
    /* create parameter list */
    mapdef = lkt_list_create();
    /* fold case */
    lkt_list_append(mapdef, lpar_create_int(LPAR_INT_FOLDCLASS, LKT_CCLASS_
CASE));
    /* fold diacritics and normalize */
    lkt_list_append(mapdef,
                    lpar_create_int(LPAR_INT_FOLDCLASS, LKT_CCLASS_
DIACRITICS));
    /* map all whitespace to blank */
    mapping = lpar_create_list(LPAR_LST_MAP);
    lkt_list_append(mapping, lpar_create_int(LPAR_INT_FROM_CLASS,
                                             LKT_CCLASS_WHITESPACE));
    lkt_list_append(mapping, lpar_create_blk(LPAR_BLK_TO_CHARS, " ", 1));
    lkt_list_append(mapdef, mapping);
    /* map all punctuation to blank */
    mapping = lpar_create_list(LPAR_LST_MAP);
    lkt_list_append(mapping, lpar_create_int(LPAR_INT_FROM_CLASS,
                                             LKT_CCLASS_PUNCTUATION));
    lkt_list_append(mapping, lpar_create_blk(LPAR_BLK_TO_CHARS, " ", 1));
    lkt_list_append(mapdef, mapping);
    /* now apply our special mappings */
    mapping = lpar_create_list(LPAR_LST_MAP);
    lkt_list_append(mapping, lpar_create_blk(LPAR_BLK_FROM_CHARS, from_set,
4));
    lkt_list_append(mapping, lpar_create_intarr(LPAR_INTARR_TO_UCODES, to_
set, 4));
    lkt_list_append(mapdef, mapping);
    /* create the character map */
    err = lkt_create_charmap(mapname, mapdef);
    if (DVKERR(err)) {
            /* handle error here */
    }
}
```

Thesaurus Matching

The TIBCO Patterns string matching algorithm does an excellent job of finding similar strings, but sometimes it's desirable to equate strings which are textually dissimilar. For example, a computer hardware vendor might want customers searching for notebook to be able to find items listed under laptop. Because these two strings bear very little resemblance to each other, the standard matching probably won't find these items, but with a thesaurus that lists the laptop as a synonym for notebook, it does.

The available thesaurus types are substitution, weighted term, and combined. All of them share the following attributes.

A thesaurus defines a set of classes, where a class is a list of terms that are to be considered equivalent. A term is a list of one or more words (more correctly tokens, a white space separated group of characters). As a term might have more than one word it is possible to set up equivalences between phrases and words or phrases and phrases, e.g. hypertension and high blood pressure. All terms within a class are considered equivalent, but equivalences are not transitive across classes. That is if we have two equivalence classes:

```
john,jonathan,johnny
jean,john
```

Then john is equivalent to jonathan, johnny and jean, but neither jonathan nor johnny is equivalent to jean.

By default, thesaurus matching supports a small degree of error tolerance. Thus given the above classes john is also match jonathin, just one letter off, but would not match johnithon.

Error tolerance is relative to the length of the term in characters, thus short terms must be match exactly, long terms might allow up to two character differences. If error tolerance is not desired, you can specify when creating a thesaurus that it allow only exact matches.

Standard Substitution

This is used for the typical use case described above. If it finds two equivalent, but not identical, terms in query and record the two terms are linked together as being a match. A substitution penalty might be defined in the query which is applied to all such matched thesaurus terms. Essentially, instead of being considered a perfect 1.0 match, the score for the matched term is considered to be the penalty factor. Thus with a query of john and a record of jean, with no penalty the score is 1.0, with a 0.9 penalty the score is 0.9 and with a 0.1 penalty the score is 0.1. This supports terms that match perfectly without the thesaurus to be considered better matches than those matched by the thesaurus.

Weighted Term

This is a special kind of thesaurus. Its primary purpose is not to link textually dissimilar terms, but to change the importance within the match of certain terms by applying a weighting factor to the term. A value below 1.0 indicates a lower importance, while a value above 1.0 is indicative of a higher importance. The special value -1.0 can be used to indicate a stop term, this is a term that is ignored completely, matching being done as if the term did not appear. For instance for business name data the terms company and incorporated are very common and thus of little relevance to a match.

Without weighted terms a search for abc incorporated would match more strongly on a2z incorporated than on abc company. By defining terms such as incorporated and company as weighted terms with very low weights, we tell the TIBCO Patterns servers that even though there is a perfect match on the long string incorporated this is of little importance, thus the match of abc to abc dominates the score and brings the desired record to the top.

Unlike the penalty factor of the substitution thesaurus the weight on a weighted term does not lower the overall score, it essentially shrinks or enlarges the effective size of the term within the match. Also unlike a substitution thesaurus a weighted term dictionary does not necessarily create an equivalence between record and query, it applies the weighting factor to the term

Whenever and wherever it is found, even if there is no match found for it. As with a substitution thesaurus a weighted dictionary can be used to create matches between dissimilar items. For example "inc" and "incorporated" can be specified as equivalent terms. If equivalent but non-equal terms are matched between query and record the weighting factor is applied to the average of the lengths of the terms, and, as with the substitution thesaurus, any penalty factor specified in the query is also applied as a penalty which results in lower final score.

Each class within a weighted term thesaurus has its own weighting factor. This factor is a positive floating point number or the special stop token value -1.0. It is given as the first term within the class. Because weighted terms do not necessarily define synonyms it is perfectly legitimate to define classes with only a single term in them (in addition to the weight).

Combined

A combined thesaurus combines the features of a substitution thesaurus and a weighted term thesaurus. Each class is given both a weight and a penalty. It behaves like a weighted term thesaurus in that the weighting factor is applied to all terms found in query or record, even if not matched with equivalent terms. It behaves like a substitution thesaurus in that when equivalent, but non identical, terms are matched between query and record those terms are linked as matched and the penalty is applied. To continue the example of business names, in addition to reducing the weight for common terms like incorporated we might want to add equivalences for common names and abbreviations such as American Broadcasting Company and ABC. These could be entered with a weighting factor of 1.0 and whatever penalty is desired. A different penalty factor might be desired for common company nicknames such as IBM and Big Blue. The combined thesaurus gives you the flexibility to do so.

For combined thesauri the first entry for each class is the weighting factor as defined for a weighted term thesaurus, the second term is the penalty, a floating point number between 0.0 and 1.0 inclusive. If a query thesaurus penalty is given with a combined thesaurus that penalty is multiplied with the class penalty to get the final penalty. Generally there should never be a need to use a query penalty with a combined thesaurus however.

Conflict Resolution

It is possible that the thesaurus matching might find more than one possible thesaurus match for a particular word in query or record. Although the exact resolution rules are complex the general rule is such conflicts are resolved to maximize the overall score of the thesaurus match.

Creating a Thesaurus (lkt_create_thesaurus)

The following command is used to create a thesaurus either locally or on a remote server.

```
dvkerr_t lkt_create_thesaurus(lpar_t host, lpar_t thesaurus_options,
                              lpar_t thesaurus_data );
dvkerr_t lkt_create_thesaurusT(lpar_t host, lpar_t thesaurus_options,
                              lpar_t tran,
                              lpar_t thesaurus_data );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

thesaurus_options (required)

is a list containing thesaurus settings. It must contain one thesaurus name lpar (LPAR_STR_THESAURUSNAME). Like the database name, this is a null-terminated character string that is unique to the thesaurus.

The list might also contain the following:

- LPAR_STR_THESAURUSTYPE specifies the type of thesaurus that is being created. Set this parameter to "weighted term" for a weighted term thesaurus, "substitution" for a standard thesaurus or "combined" for a combined thesaurus. The default value is "substitution".
- LPAR_STR_CHARMAP The character map used in loading the thesaurus class items. Character maps are described in Character Mapping.
- LPAR_INT_THESMATCHMODE Indicate whether error tolerant (0) or exact only (1) matching is to be used by this thesaurus. Default value is (0), error tolerant.

If the thesaurus name is the only desired parameter, it might be passed directly without being encapsulated in a list.

thesaurus_data (required)

is a list elements of type LPAR_BLKARR_THESAURUSCLASS. A standard substitution thesaurus class is an array of byte blocks which form an equivalence class (each element in the array is equivalent to each other element in the array). The elements of the block array must be UTF-8 encoded characters.

If this thesaurus is a weighted term thesaurus, the first item of the LPAR_BLKARR_THESAURUSCLASS array must be a floating point number in string format (as recognized by the C `sscanf(3)` function). This is the weight for the terms of the class. If the first item is not a valid floating point value the load of the thesaurus fails and a file format error is returned. All following elements are the terms. As in the standard substitution thesaurus these terms are considered equivalent.

If this thesaurus is a combined thesaurus the first item of the LPAR_BLKARR_THESAURUSCLASS array is the term weight as described for weighted term thesauri. The second term is the substitution penalty. It has the same format as the weighted term with the restriction that it must be a value between 0.0 and 1.0 inclusive.

Multi-token thesaurus matching is allowed. This means you can equate hypertension with high blood pressure. Empty terms are not allowed and causes the create to fail.

A thesaurus can also be read directly from a file by passing as the contents of this list a single LPAR_STR_CSVFILE and optionally an LPAR_STR_ENCODING value instead of the LPAR_BLKARR_THESAURUSCLASS values. If a host is specified, the file must exist within the host's loadable-data directory. In this case, each line in the file is considered an equivalence class and must follow the rules for a class for the thesaurus type. The character encoding used in the file can be specified with a LPAR_STR_ENCODING parameter. The recognized encodings are: UTF-8 and latin-1 case-sensitive. The default is latin-1. If only an LPAR_STR_CSVFILE value is desired, it can be given directly as the thesaurus_data value, instead of packaging it in a list. If the same token occurs in multiple equivalence classes, the tokens in a given class are not equivalent with the tokens of the other classes. For example, if given the two equivalence classes (X,Y) and (X,Z), X would be a synonym for Y and Z, but Y and Z would not be synonyms for each other.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the thesaurus is to be created.

If a thesaurus with the specified thesaurus name already exists, it is replaced.

Warning: Thesaurus Character Maps: An Explanation And Caution

Warning: The thesaurus character map is used when loading the thesaurus class entries. The character map is applied to all of the class entries before being stored.

Warning: Although there is no validation that forces you to do so the thesaurus character map should always match the map used in the database fields to which the thesaurus is applied.

Warning: If the thesaurus character map differs from the character map for the field data the thesaurus might be unable to find any matches. The default map for both the thesaurus and the field data is the same. In the vast majority of cases this is the mapping that should be used in both cases.

Error codes and items returned by lkt_create_thesaurus

NOCHARMAP	list containing character map that doesn't exist
EXPECTTHESNAME	item that was expected to be a thesaurus name
EXPECTLIST	item that was expected to be a list
FEATURESET	(none)
FILEFORMAT	LPAR_STR_FILEFORMATERROR description of error
INTERNAL	LPAR_STR_ERRORDETAILS description of error
IOERROR	LPAR_STR_SYSERROR description of error
PARAMCONFLICT	item in conflict with earlier item
PARAMMISSING	(none)
PARAMVAL	item that had an invalid value
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ERRORDETAILS

Checking the Status of Loaded Thesauri (lkt_thlist)

The following command returns a list of lists of thesaurus statistics - one list per thesaurus - specified.

```
dvkerr_t lkt_thlist(lpar_t host, lpar_t names, lpar_t *thstatlists );
```

At this time only the thesaurus name, type and number of unique terms is returned.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

names (optional)

is a list of thesaurus names (LPAR_STR_THESAURUSNAME's) of existing loaded thesauri. It is an error (DVK_ERR_THESNOTFOUND) if any of these thesauri do not exist.

A value of LPAR_NULL might be supplied for names. This causes statistics lists for all currently loaded thesauri to be returned.

Output

thstatlists (required)

is the list of statistics lists. Each item in the list is a generic lpar list containing the following values:

- LPAR_STR_THESAURUSNAME the name of the thesaurus.
- LPAR_STR_THESAURUSTYPE one of: substitution, weighted term, combined.
- LPAR_INT_NUMTOKENS a count of the total number of tokens (words) in the thesaurus. This is mostly useful as a rough check of whether two versions of a thesaurus are the same.
- LPAR_LONGINT_TRAN_ID is the transaction id of the transaction that owns or claims the object at that instance.
- LPAR_INT_TRAN_OBJ_STATE is the state of an object when a transaction owns or claims it; if there is no such transaction, it returns LKT_TRAN_OBJ_EXISTS.

Error codes and items returned by lkt_thlist

THESNOTFOUND	list of names of nonexistent thesauri
EXPECTLIST	item that was expected to be a list
PARAMVAL	item that had an invalid value
NODBDESC	(none)
NOSYSINIT	(none)

Deleting Thesauri (lkt_delete_thesaurus)

This function deletes one thesaurus, a list of thesauri, or all thesauri.

```
dvkerr_t lkt_delete_thesaurus(lpar_t host, lpar_t names);
dvkerr_t lkt_delete_thesaurusT(lpar_t host, lpar_t names, lpar_t tran);
```

If a delete of multiple thesauri is requested all thesauri that can be deleted are deleted even if one or more deletes should fail. If one or more deletes fail a list of those that failed are returned as the error item.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

name (required)

identifies the thesaurus or thesauri to be deleted. If this is LPAR_NULL all currently defined thesauri are deleted. If this is LPAR_STR_THESAURUSNAME the thesaurus named is deleted. If this is a generic list (LPAR_LST_GENERIC) then each item of the list must be an LPAR_STR_THESAURUSNAME that names a thesaurus to be deleted. It is invalid to name the same thesaurus more than once.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the thesaurus is to be deleted
Error codes and items returned by lkt_delete_thesaurus

DBNOTFOUND	list of database names not found
DBPARAM	invalid lpar from names parameter
DUPDBDESCS	duplicate thesaurus name
EXPECTDBDESC	item that should have been a thesaurus name
EXPECTLIST	item that should be a generic list of thesaurus names
FEATURESET	LPAR_NULL
INTERNAL	list of thesauri that failed to delete
NODBDESC	(none)
NOSYSINIT	(none)
PARAMVAL	item that should be a LPAR_STR_THESAURUSNAME value or zero length LPAR_STR_THESAURUSNAME value
THESNOTFOUND	list of thesauri from names that did not exist
TRAN_UNKNOWN	lpar that contains the unknown transaction id
TRAN_IN_USE	lpar that contains the transaction id
TRANCONFLICT	list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ ERRORDETAILS

Using a Thesaurus in a Search

To use a thesaurus in a search, simply pass an LPAR_STR_THESAURUSNAME as one of the search parameters (using the srchpars argument to `lkt_dbsearch`). A different thesaurus can be specified per querylet by adding an LPAR_STR_THESAURUSNAME to each LPAR_LST_

SIMPLEQUERY, LPAR_LST_COGQUERY or LPAR_LST_QUERYLET. If a thesaurus is specified in both places the thesaurus specified in the query specification is used.

Each thesaurus must exist on the same server as the database.

To set the penalty factor of a substitution thesaurus match, pass an LPAR_DBL_THESAURUSWEIGHT as a search parameter. It might also be added to the parameters list for simple, cognate and attributes queries to set the penalty for individual querylets. If set in both places the value on the individual querylet takes precedence.

Ephemeral Thesauri

As an alternative to passing an LPAR_STR_THESAURUSNAME argument specifying a thesaurus previously created with the `lkt_create_thesaurus` command an LPAR_LST_THESAURUS can be passed in the search options, in the LPAR_LST_SIMPLEQUERY or in the LPAR_LST_COGQUERY (but not in the old style LPAR_LST_QUERYLET). The LPAR_LST_THESAURUS should contain the two argument values, thesaurus_options and thesaurus_data, as passed to the `lkt_create_thesaurus` command. That is one element of the list is a generic list containing the thesaurus name and, optionally, type, match mode and character map. The second element of the list is another LPAR_LST_THESAURUS containing the thesaurus class data for the thesaurus.

A thesaurus so defined is created and exists only for the duration of the query, and is thus known as an ephemeral thesaurus. Although when defining ephemeral thesauri the thesaurus name is required the name is ignored. It is required only to maintain consistency with the format of the thesaurus definition for the command. Thus ephemeral thesauri might have the same name as thesauri created with the `lkt_create_thesaurus` command or even other ephemeral thesauri in the same query without danger of interfering with or overriding these definitions.

Ephemeral thesauri are intended for those cases where possible synonyms or term weightings for a query are generated dynamically based on the query contents and cannot be predefined for all possible queries. Although it is possible to create any size ephemeral thesaurus or even create them from a CSV file it is strongly recommended that they be limited to a few classes in size. Creating ephemeral thesauri from a CSV file is NOT recommended for obvious performance reasons.

Match Visualization

In most cases, users of TIBCO Patterns servers profit greatly from the ability to see which portions of a matching record matched the query, and with what intensity. We call this match visualization. Typically, different colors and/or type sizes are used to highlight characters of matching records according to their relative match intensity.

When you include the LPAR_INT_VISUALSTYLE parameter set to a value of 2 in the `srchpars` parameter list that you pass to the `lkt_dbsearch` command, `lkt_dbsearch` will output visualization information. You'll find it, along with other match-specific information, in each of the match information lists contained in the minfo output list (see Invoking TIBCO Patterns Matching (lkt_dbsearch)).

Each minfo list contains four array lpars, LPAR_DBLARR_V2V, LPAR_INTARR_V2P, LPAR_INTARR_N, and LPAR_INTARR_V2D (known respectively as the V array, the P array, the N array, and the D array), whose lengths are equal to the length of all searchable text fields of the matching record.

Note: In the case of queries subject to field selection, this length includes the lengths of all searchable text fields, those selected for searching and those not selected. Non-searchable text fields, variable attributes fields, integer, double and date fields are NOT included.

The first element of each array applies to the first character of text, the second element of the arrays applies to the second character of text, and so on. In other words, there is a V value, P value, N value, and a D value for each character of searchable text in a record.

It is important to note that the visualization vectors are based on character positions, not byte positions. If the original data was UTF-8 encoded the lengths of the visualization vectors corresponds to the number of characters in the original data, which might differ from the number of bytes. In all of the discussion below references to character position mean exactly that, character and not byte.

Each (V, P,N,D) value set provides a measure of the match intensity at that character position in the searchable text. The P value measures how big a segment of matching text this character position is part of. The D value measures how far away this segment is from the corresponding segment in the query, given an optimal superposition of query over searchable text. If the P value for a character position is 0, this means that character remained unmatched in the match calculation. The N value is set to one if the character is considered to be noise by the matching algorithm. Noise characters are considered much less important than non-noise characters. They contribute much less to the match score than non-noise characters. The V value is a summary score for the match strength on that character computed from the P, D and N values plus various weighting factors defined for the query. It is a value between 0.0 and 1.0 inclusive. For most applications the V value should provide all of the visualization information needed.

The typical color visualization scheme utilizing these values is one that associates six shades of some hue with V values, with brighter shades corresponding to higher V values. The V values are split evenly into 7 score ranges. The lowest range is not highlighted, the higher ranges are assigned the six shades with the higher score ranges getting the brighter shades.

When you include the LPAR_INT_VISUALSTYLE parameter set to a value of 256, `lkt_dbsearch` outputs a block in HTML format based on the V array. The blocks are returned in the minfo lpar list with each match, and are of type LPAR_BLK_HTML. This style accepts several lpars via the srchpars list for customization:

- LPAR_BOOL_USECOLOR specifies whether or not colors are used to distinguish match strength. If true, it creates a gradient between LPAR_STR_BASECOLOR and LPAR_STR_MATCHCOLOR for different strengths of matches within the searchable text. This parameter must be true for LPAR_STR_BASECOLOR and LPAR_STR_MATCHCOLOR to be valid parameters.

  Default value: true

- LPAR_STR_BASECOLOR specifies the end of the color gradient for weak matches.

> **Note:** The base color does not specify unmatched text color; it specifies a gradient so that matches fit well with the default text color of a webpage.

  It must be a string of length six containing six hexadecimal digits corresponding to an RGB color. The first two digits correspond to the amount of red, the third and fourth to the amount of green, and the fifth and sixth to the amount of blue. Note that this is one of the standard formats for specifying colors in HTML.

  Default value: "0000ff"(blue)

- LPAR_STR_MATCHCOLOR specifies the end of the color gradient for strong matches.

  The string's format is the same as that of LPAR_STR_BASECOLOR.

  Default value: "ff0000" (red)

- LPAR_BOOL_USEBGCOLOR specifies whether or not background colors (text highlighting) are used to distinguish match strength. This functions like LPAR_BOOL_USECOLOR except it uses LPAR_STR_BASEBGCOLOR and LPAR_STR_MATCHBGCOLOR to create the gradient.

  Default value: false

- LPAR_STR_BASEBGCOLOR specifies the end of the background color gradient for weak matches. The string's format is the same as that of LPAR_STR_BASECOLOR.

  Default value: "0000ff" (blue)

- LPAR_STR_MATCHBGCOLOR specifies the end of the background color gradient for strong matches. The string's format is the same as that of LPAR_STR_BASECOLOR.

  Default value: "ff0000" (red)

- LPAR_INT_MAXFONTSIZE creates a gradient between this value and the html document's default font size. It must be between 0 and 4, where values of 1 to 4 increase the font size for stronger matches and 0 disables the feature.

  Default value: 0 (disabled)

- LPAR_INT_BOLDTHRESH is the threshold for displaying matching text in bold. It accepts a value between 0 and 6. Characters in the searchable text are bold if the corresponding V is greater than or equal to this threshold. A threshold value of 0 disables this feature. The integer values 1 through 6 correspond to the 6 upper ranges of the evenly divided V value range of 0.0 - 1.0.

  Default value: 0 (disabled)
- LPAR_INT_ITALICSTHRESH is identical to LPAR_INT_BOLDTHRESH except that it specifies the threshold for italicized text.

  Default value: 0 (disabled)
- LPAR_INT_UNDERLINETHRESH is identical to LPAR_INT_BOLDTHRESH except that it specifies the threshold for underlined text.

  Default value: 0 (disabled)

  Visualization style 256 also returns LPAR_BLK_HTMLLEGEND for each search via the stats lpar list. It is an HTML segment that contains a strong to weak indicator to help interpret the match strengths in the LPAR_BLK_HTML lpars.

Multithreaded Programming With the DevKit

The DevKit is designed to be used in a multithreaded environment. It takes care of ensuring that all of the in-memory data objects are accessed and updated in thread safe manner. Applications using the DevKit need not worry about controlling access to these objects. However, it might be useful to understand the locking that goes on when the DevKit commands are called. This section describes that locking at a high level.

Thesauri, Character Maps, and Learn Models are all immutable objects. They cannot be updated or modified after they are created. This simplifies their management. The locks associated with adding, deleting, and accessing these objects are held so briefly that they have no impact on an application. An exception might be when checkpointing these objects, the tracking data for checkpoints might have to be held locked for a noticeable period of time. This can block some other operations that must access the checkpoint data.

If checkpointing is enabled, any operation that adds, deletes or renames an in-memory object (database table, thesaurus, character map, Learn Model) must write lock the checkpoint data while the operation is performed. In addition the checkpoint operation must write lock the checkpoint data. The restore operation must read lock the checkpoint data. The duration of the locks are kept to a minimum, but any of these operations can create contention issues in a heavily loaded system.

Database tables can be modified and therefore must be locked while being used. Locking is at the table level, not the individual record level. Any number of threads can read a table in parallel, but update operations require an exclusive lock on the table; they must lock out all other operations, both read and update, while the update is performed. In applications that have multiple threads both reading and updating a table, this can create contention issues.

Operations that add, delete, or rename a table must update the information on the set of tables in memory. All other operations trying to access a table are locked out while that information is being updated. These locks are held briefly (not for the duration of the entire command), but might still create contention issues as they block all other commands that access tables.

The following important classes of locks are involved:

- **Locks on the Checkpoint Data**: conceptually there is a single lock. This comes into play only when checkpoint or restore is enabled. Generally, commands hold this lock only briefly.
- **Locks on the index of Database Tables**: this controls operations adding, deleting or renaming tables. Conceptually there is a single lock. Generally, commands hold this lock only briefly.
- **Locks on individual Database Tables**: this controls access to an individual table. There is a separate lock for each table. Commands must hold this lock for the duration of the command.

Additional locks are used for the indexes of the thesauri, Learn Models, and custom character maps that are loaded on the server. These locks are held only briefly, and in most applications are used rarely, hence they have negligible impact on the performance.

Each of these locks can be read locked or write locked. Any number of threads can hold a read lock, but a write lock is exclusive, no other thread can have a read lock or write lock.

In the following table the entry:

- "**-**" indicates no lock is used
- "**R**" indicates a read lock is used
- "**W**" indicates a write lock is used
- "**V**" indicates the lock used varies - this pertains to the transaction commit and abort commands, the locks needed depend on the operations within the transaction.

The table lists all of the DevKit commands that use one of the classes of locks described above.

DevKit commands with locks and their types

Command	Checkpoint Lock	Database Index Lock	Database Table Lock
lkt_dbload	R	W	-

Command	Checkpoint Lock	Database Index Lock	Database Table Lock
lkt_dbdelete	W	W	R
lkt_dbmove	W	W	W (source) R (dest)
lkt_dbrecget	-	R	R
lkt_dbrecadd	-	R	W
lkt_dbrecdelete	-	R	W
lkt_dbrecreplace	-	R	W
lkt_dbrecupdate	-	R	W
lkt_dblist	-	R	R
lkt_idxlist	-	R	R
lkt_dbdump	-	R	R
lkt_dblock	-	R	W
lkt_dbunlock	-	R	W
lkt_dbsearch	-	R	R
lkt_delete_thesaurus	W	-	-
lkt_delete_rlmodel	W	-	-
lkt_checkpoint	W	R	R
lkt_restore	R	W	R
lkt_tran_commit	V	V	V

Command	Checkpoint Lock	Database Index Lock	Database Table Lock
lkt_tran_abort	V	V	V
lkt_dbrecnext	-	R	R

Transactions

A *transaction* is a group of write operations that can be committed or rolled back as a unit. A *write* operation is any operation that changes the existence or content of data on the Patterns server. The following is a complete list of write operations:

- Table load, delete, move, checkpoint, and restore
- Record add, delete, update, and replace
- Thesaurus create and delete
- Character Map create
- Learn Model create and delete

Each DevKit function associated with a write operation has two versions: the backwards-compatible non-transactional version and a new transactional version that takes an additional transaction ID parameter. The transactional functions are named by the convention of suffixing "T" to the non-transactional name. For example, lkt_dbload for transactional would be lkt_dbloadT.

Calling a transactional function with an LPAR_NULL transaction parameter is the same as calling the non-transactional version.

If the application uses transactions, or if a cluster is in use, DevKit functions might return error codes which include: DVK_ERR_TRAN_UNKNOWN, DVK_ERR_TRAN_IN_USE, and DVK_ERR_TRANCONFLICT. For more information, see the *TIBCO Patterns Concepts* guide.

Read operations are not affected by the use of transactions.

Transactions Usage

To use transactions, do the following:

1. Create a transaction on the server using lkt_tran_create.
2. Perform operations under the transaction using the lkt_xxxT functions, passing the transaction parameters created in step 1.
3. Commit or roll back the transaction using lkt_tran_commit or lkt_tran_abort.

The application must roll back or commit a transaction. The application can consider conditions outside the TIBCO Patterns server (for example a failed SQL operation) when choosing between committing or rolling back a transaction.

If an operation encounters an error on the TIBCO Patterns server, it is recommended that the application roll back the containing transaction. Committing such transactions might leave data in an unknown and incorrect state.

Transaction Information in Object Statistics

Object statistics (table stats, character-map stats, thesaurus stats, and Learn model stats) have additional information about their transactional status. The transaction which has modified or locked the object for use is reported in LPAR_LONGINT_TRAN_ID. The transactional state of the object is reported in LPAR_INT_TRAN_OBJ_STATE.

LPAR_INT_TRAN_OBJ_STATE values

LKT_TRAN_OBJ_NULL	a special internal value indicating an object that does not exist. This value should not be returned outside the Devkit. It is exclusive of all other values.
LKT_TRAN_OBJ_EXISTS	the value for an object that exists. All objects have this state.
LKT_TRAN_OBJ_UPDATED	an object whose content has been updated by an open transaction
LKT_TRAN_OBJ_HELD	an object on which an open transaction has placed a hold without modifying it
LKT_TRAN_OBJ_ADDED	an object that was added by an open transaction
LKT_TRAN_OBJ_RENAMED	an object that was renamed to its current name by an open transaction

LKT_TRAN_OBJ_REPLACED	all the contents of this object were replaced by an open transaction. Checkpoint and restore operations are considered to replace the table.
LKT_TRAN_OBJ_DELETED	the object that was deleted by an open transaction. This state cannot be experienced as this object was deleted.
LKT_TRAN_OBJ_MIXED	a special state value that is reported in object status reports when the nodes of a cluster report two or more different states. No object ever actually has this state.

Transaction Conflicts

When you attempt to modify the same object under two transactions, the second attempt returns a DVK_ERR_TRANCONFLICT error code, and the error item is a list of two lpars:

- LPAR_LONGINT_TRAN_ID: the ID of the conflicting transaction
- LPAR_STR_ERRORDETAILS: a message string with the format, "Tran. <ID> Action <int> Conflicts with <ID> on object: <object name>"

Creating a New Transaction (lkt_tran_create)

The command initiates a new transaction on the TIBCO Patterns server. The current limit to the number of open transactions on a TIBCO Patterns server is 255.

```
dvkerr_t lkt_tran_create(lpar_t host, lpar_t options,
                         lpar_t *tran_lpar );
```

Input

host (optional)

For more information, see Communicating with TIBCO Patterns Servers.

options (optional)

are options for future use.

Output

tran_lpar (required)

where the ID of the transaction is returned.

Error codes and items returned by lkt_tran_create

PARAMMISSING	(none)
TRAN_LIMIT	(none)

Committing a Transaction (lkt_tran_commit)

The following command commits all the work done under the specified transaction.

```
dvkerr_t lkt_tran_commit( lpar_t host, lpar_t tran_lpar,
                          lpar_t options );
```

Input

host (optional)

For more information, see Communicating with TIBCO Patterns Servers.

tran_lpar (required)

is an LPAR_LONGINT_TRAN_ID holding the ID of the transaction to commit.

options (optional)

might contain a single option or a list of allowable options.

LPAR_BOOL_DOMAXWORK is the only option that is allowed. If its value is `true` then the transaction is committed in spite of the errors occurred during work under this transaction. Using this option is not recommended as it might cause inconsistency. Consult TIBCO support before using this option.

Error codes and items returned by lkt_tran_create

PARAMMISSING	(none)
PARAMTYPE	item with invalid lpar type.
HASERRORS	(none)
TRAN_UNKNOWN	The lpar containing the unknown transaction id.

Rolling Back a Transaction (lkt_tran_abort)

The following command rolls back all the work done under the specified transaction.

```
dvkerr_t lkt_tran_abort( lpar_t host, lpar_t tran_lpar,
                         lpar_t options );
```

Input

host (optional)

For more information, see Communicating with TIBCO Patterns Servers.

tran_lpar (required)

where the id of the transaction is returned.

Error codes and items returned by lkt_tran_create

PARAMMISSING	(none)
PARAMTYPE	an item with invalid lpar type.
TRAN_UNKNOWN	the lpar containing the unknown transaction id.

List Transactions (lkt_tran_list)

The following command returns statistics for transactions on the Patterns server.

```
dvkerr_t lkt_tran_list(lpar_t host, lpar_t tran_list,
                       lpar_t options, lpar_t *tran_stats );
```

Only transactions that have been created but not committed or rolled back are reported. The listing can be restricted to specific transaction IDs or by transaction idle time or both.

Input

host (optional)

For more information, see Communicating with TIBCO Patterns Servers.

tran_list (optional)

is a list of LPAR_LONGINT_TRAN_ID. If present, only transactions in the list are reported.

options (optional)

might contain a single option or a list of allowable options.

Output

tran_stats (required)

where the list of transaction statistics is returned.

LPAR_INT_TRAN_IDLETIME is the only option that is allowed. If its value > 0, only transactions that have been idle for more than the specified number of seconds are reported.

Each transaction statistics lpar is a list containing the following lpars:

- LPAR_LONGINT_TRAN_ID
- LPAR_INT_TRAN_TYPE
- LPAR_BOOL_TRAN_GATEWAY
- LPAR_INT_TRAN_STATE
- LPAR_INT_TRAN_IDLETIME
- LPAR_INT_TRAN_ERRORS
- LPAR_INT_TRAN_LOGSIZE

Error codes and items returned by lkt_tran_create

PARAMMISSING	(none)
PARAMTYPE	an item with invalid lpar type

Transactions Example

Create a character map and a table, and commit or roll them back together.

```
lpar_t   tran ;   /* the transaction id parameters. */
dvkerr_t err ;    /* return status of Devkit function calls */
/* parameters for character map creation */
lpar_t host ;
lpar_t mapname ;
lpar_t mapdef ;
/* parameters for table creation */
lpar_t dbname;
lpar_t dbpars;
lpar_t dbstats;
dbname = lpar_create_str(LPAR_STR_DBDESCRIPTOR, "dbname");
host = lkt_host_lpar( "localhost", 5051);
/* population of other parameters omitted */
/* create transaction, storing tran id in tran variable */
err = lkt_tran_create(host, LPAR_NULL, &tran) ;
if ( ! DVKERR(err) ) {
    /* create character map */
    err = lkt_create_charmapT(host, mapname, tran, mapdef);
    if ( ! DVKERR(err)) {
        /* create table */
        err =lkt_dbloadT(host, dbname, tran, dbpars, reclist, &dbstats);
    }
    /* commit or abort */
    if ( DVKERR(err) ) {
        err = lkt_tran_abort(host, tran, LPAR_NULL);
    } else {
        err = lkt_tran_commit(host, tran, LPAR_NULL);
    }
}
```

The TIBCO Patterns Machine Learning Platform

You can combine individual scores in an intelligent way using TIBCO Patterns Machine Learning Platform.

TIBCO Patterns Machine Learning Platform

The TIBCO Patterns Machine Learning Platform maximizes the difference between a match (good set of scores) and a non-match (bad set of scores). It does this using a Learn model that has been created and trained for the particular application. A model can be created and trained using the Learn UI (see the *TIBCO Patterns Learn UI Guide*), the Learn API library (see the *TIBCO Patterns Learn API reference* documentation), or TIBCO Patterns can create a Learn model for you. The model can be used to evaluate input feature vectors. It can also be used to predict the match score in a search operation. To use the model you must first load the model into memory with a given name. The model is then specified by name as a parameter for the search operation or feature vector evaluation operation. The structure of the query or the input feature vector used with the model must be identical to those used to train the model.

Loading a Learn Model (lkt_create_rlmodel)

Using the following command you can load a TIBCO Patterns Learn Model either locally or on a remote server.

```
dvkerr_t lkt_create_rlmodel(lpar_t host, lpar_t name, lpar_t model );
dvkerr_t lkt_create_rlmodelT(lpar_t host, lpar_t name, lpar_t tran,
                             lpar_t model );
dvkerr_t lkt_create_rlmodelX(lpar_t host, lpar_t name, lpar_t tran,
                             lpar_t options, lpar_t model);
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

name (required)

This must be an lpar of type LPAR_STR_RLMODELNAME. Like the database name, this is a null-terminated character string that is unique to the model.

model (required)

This is the location of the model file on disk. If the model exists on the local file system and a host is specified, use an LPAR of type LPAR_RN_FILE. Otherwise, use an LPAR of type LPAR_STR_RLMODELFILE. If no host is specified, the interface will read from the local file system. If a host is specified, the host will read the file, and the file must exist within the host's loadable-data directory.

If a model with the specified model name has already been loaded, it is replaced.

tran (optional)

This identifies the user transaction (LPAR_LONG_TRAN_ID) under which the Learn model is to be created.

options (optional)

This is either a supported option or a list of supported options.

Currently the only supported option is LPAR_BOOL_USE_MEMBLOCK.

LPAR_BOOL_USE_MEMBLOCK is only used by gateway engines (non-gateway engines ignore it). This option enables the transfer of very large Learn models through the gateway without the use of a temporary file; this might consume large amounts of memory on the gateway engine. This option is off by default. For more information, see the -G command line option in the *TIBCO Patterns Installation Guide.*

Error codes and items returned by lkt_create_rlmodel

NORLINKMODEL	(none)
PARAMVAL	the item that had an invalid value
PARAMMISSING	(none)
TRAN_UNKNOWN	the lpar that contains the unknown transaction id
TRAN_IN_USE	the lpar that contains the transaction id
TRANCONFLICT	the list that contains LPAR_LONGINT_TRAN_ID and LPAR_STR_ERRORDETAILS

Checking the Status of Loaded Learn Models (lkt_rllist)

```
The following command returns a list of lists of TIBCO Patterns Learn Model
information, one list per model specified.
dvkerr_t lkt_rllist(lpar_t host, lpar_t names, lpar_t *rlstatlists );
```

The information returned includes the name of the model, the metadata associated with the model and the number of features in the model.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

names

specifies one or more Learn models to be listed. If the value of LPAR_NULL is given information on all currently loaded models is returned. If a single lpar is given information on the named model, LPAR_STR_RLMODELNAME is returned. Otherwise this must be a generic list (LPAR_LST_

GENERIC) of LPAR_STR_RLMODELNAME values and information, on each of the named models, is returned.

Output

rlstatlists (required)

is the list of statistics lists. The value returned is a list of generic lists (LKT_LST_GENERIC) containing the following values:

- LPAR_STR_RLMODELNAME the name of the model
- LPAR_INT_RLNUMFEATURES the number of features in the model, that is, the number of input feature scores the model expects.
- LPAR_STR_RLMETADATA the metadata for the model (arbitrary text data associated with the model when it was saved to file)
- LPAR_STR_RLTAGSINFO shows all of the tags for this model. The values are encoded as: '**<**'*tag-name*'**=**'*tag-value*'**>**' The currently defined *tag-names* are: THRESHOLD, ID, and VERSION. If there are no tags for this model "**=NONE=**" is returned.
- LPAR_STR_RLMODEL_EXT_ID if an external ID has been assigned to this model this value is returned with the external ID value. Generally external IDs are only used in a clustered environment and are generated by the Gateway server.
- LPAR_LONGINT_TRAN_ID is the transaction ID of the transaction that owns or claims the object at that instance.
- LPAR_INT_TRAN_OBJ_STATE is the state of an object when a transaction owns or claims it; if there is no such transaction, returns LKT_TRAN_OBJ_EXISTS.

Error codes and items returned by lkt_rllist

EXPECTLIST	the item that should have been a list of model names
FEATURESET	(none)
INTERNAL	the details on the error that occurred
LOOKUP	(none)

NODBDESC	(none)
NOSYSINIT	(none)
PARAMVAL	an item that should be a LPAR_STR_RLMODELNAME or zero length LPAR_STR_RLMODELNAME value
RLINKMODELNOTFOUND	the name of the model not found

Deleting Learn Models (lkt_delete_rlmodel)

```
The following function deletes one Learn model, a list of models, or all
models.
dvkerr_t lkt_delete_rlmodel(lpar_t host, lpar_t names);
dvkerr_t lkt_delete_rlmodelT(lpar_t host, lpar_t names, lpar_t tran);
```

If a delete of multiple models is requested, all models that can be deleted are deleted, even if a single delete operation fails. If one or more deletes fail, a list of those that failed are returned as the error item.

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

This is used to identify the server. For more information, see Communicating with Servers.

name (required)

identifies the model or models to be deleted. If this is LPAR_NULL, all currently loaded models are deleted. If a single LPAR_STR_RLMODELNAME is given, the named model is deleted. If this is a generic list (LPAR_LST_GENERIC) then each item of the list must be an LPAR_STR_

RLMODELNAME that names a model to be deleted. It is invalid to name the same model more than once.

tran (optional)

identifies the user transaction (LPAR_LONG_TRAN_ID) under which the Learn model is to be deleted.

Error codes and items returned by lkt_delete_rlmodel

EXPECTLIST	an item that should have been a list of model names
FEATURESET	(none)
INTERNAL	a list of models that were not deleted
LOOKUP	(none)
NODBDESC	(none)
NOSYSINIT	(none)
PARAMVAL	an item that should be a LPAR_STR_RLMODELNAME or zero length LPAR_STR_RLMODELNAME value
RLINKMODELNOTFOUND	a list of models not deleted because they didn't exist
TRAN_UNKNOWN	an lpar that contains the unknown transaction id
TRAN_IN_USE	an lpar that contains the transaction id
TRANCONFLICT	a list that contains LPAR_LONGINT_TRAN_ID and LPAR_ STR_ERRORDETAILS

Evaluating Feature Scores Using a Learn Model (lkt_rleval)

The following command is used to evaluate a feature vector on a TIBCO Patterns Learn Model.

```
dvkerr_t lkt_rleval(lpar_t host, lpar_t modelname, lpar_t indata,
                    lpar_t *outdata );
```

Input

host (optional)

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

This is used to identify the server. For more information, see Communicating with TIBCO Patterns Servers

This is used to identify the server. For more information, see Communicating with Servers.

modelname (required)

is the name of the model (LPAR_STR_RLMODELNAME) to be used.

indata (required)

must be a generic list containing an lpar of type LPAR_DBLARR_RLFEATUREVEC, which is an array of double feature values (from 0.0 to 1.0, or -1.0 for a missing feature value) which are evaluated by the model. The number and meaning of features in this array must be identical to the features used when training the model.

Output

outdata (required)

is a generic list (LPAR_LST_GENERIC) containing one value of type LPAR_DBL_RLRLINKSCORE in the range of 0.0 to 1.0. This value is the evaluation result produced by the model for the given feature vector.

Error codes and items returned by lkt_rleval

NORLINKMODEL
PARAMVAL
PARAMMISSING
RLINKMODELNOTFOUND
NOFEATUREVEC
NUMFEATURESMISMATCH

Using a Learn Model in a Search

To use a trained Learn model in a search query use the LKT_QEXPR_RLINK query combiner. The name of the TIBCO Patterns Learn model to be used should be passed as an LPAR_STR_RLMODELNAME parameter to the LPAR_LST_QEXPR.

For a cutoff to be employed, the cutoff threshold can be encoded in the TIBCO Patterns Learn model and the LPAR_BOOL_USEMODELTHRESH lpar can be added to the LPAR_LST_QEXPR with a value of true. The proper cutoff value is likely to change each time the model is retrained; this ensures the proper cutoff for the model in use is always applied.

The number of sub-queries to the LKT_QEXPR_RLINK combiner must be the same as the number of features in the model. The query structure under the LKT_QEXPR_RLINK combiner must be the same as the query used to train the model.

Table of DevKit Error Codes and Strings

This appendix includes the following error codes from the `devkit.h` header file:

1. DVK_ERR_PARTIAL - Partial search results (sort of success)
2. DVK_ERR_NOSYSINIT - System not initialized
3. DVK_ERR_SYSINIT - System already initialized
4. DVK_ERR_FEATURESET - Feature not present
5. DVK_ERR_FEATUREOFF - Feature present but disabled
6. DVK_ERR_EXPECTLIST - List was expected
7. DVK_ERR_EXPECTDBDESC - Database descriptor was expected
8. DVK_ERR_EXPECTRDBDESC - Remote database descriptor was expected
9. DVK_ERR_EXPECTRECORD - Record was expected
10. DVK_ERR_EXPECTRECKEY - Record key was expected
11. DVK_ERR_EXPECTQUERY - Query was expected
12. DVK_ERR_EXPECTTHESNAME - Thesaurus name expected
13. DVK_ERR_EXPECTCHARMAP - Character map name expected
14. DVK_ERR_DBPARAM - Illegal database parameter
15. DVK_ERR_CHARCONV - Error in character conversion
16. DVK_ERR_CHARMAP - Invalid character mapping
17. DVK_ERR_NORECKEY - Record key is required
18. DVK_ERR_NOSRCHTXT - Searchable text is required
19. DVK_ERR_NUMFIELDS - Wrong number of fields in record
20. DVK_ERR_FIELDLENSUM - Field lengths do not sum to record length
21. DVK_ERR_FIELDLEN - Single field length greater than 2^16
22. DVK_ERR_UNKFIELD - Unknown field name
23. DVK_ERR_NSFIELD - Attempt to search non-searchable field
24. DVK_ERR_NOWGTFLD - Weight field not specified in RECWGT expr
25. DVK_ERR_TBLNOTCHILD - Specified table is not a child table
26. DVK_ERR_QUERYEXPR - Badly constructed query expression
27. DVK_ERR_SRCHPARAM - Illegal search parameter

28. DVK_ERR_UPDPARAM - Illegal update parameter
29. DVK_ERR_PARAMVAL - Illegal parameter value
30. DVK_ERR_PARAMCONFLICT - Conflicting parameters
31. DVK_ERR_TRAN_EXISTS - A transaction was detected.
32. DVK_ERR_PARAMMISSING - Parameter missing from list
33. DVK_ERR_PARAMTYPE - Parameter type is invalid for operation
34. DVK_ERR_ARRAYLEN - Wrong number of values in array parameter
35. DVK_ERR_LISTLEN - Wrong number of items in list
36. DVK_ERR_PREDTYPE - Predicate does not evaluate to Boolean
37. DVK_ERR_PREDSTRING - Error in predicate string
38. DVK_ERR_NODBDESC - No database descriptor
39. DVK_ERR_DBNOTFOUND - Database not found
40. DVK_ERR_DBEXISTS - Database already exists
41. DVK_ERR_DBLOCKED - Database locked
42. DVK_ERR_DBUNLOCKED - Database unlocked
43. DVK_ERR_DBMOVESAME - Source and destination are the same
44. DVK_ERR_DBNOFIELDS - Field support not requested for this db
45. DVK_ERR_NORECORDS - No records
46. DVK_ERR_RECEXISTS - Record already exists
47. DVK_ERR_RECNOTFOUND - Record not found
48. DVK_ERR_PARTIALCHPT - Some of the tables failed to checkpoint
49. DVK_ERR_CHPTFAIL - All of the tables failed to checkpoint
50. DVK_ERR_PARTIALRESTORE - Some of the tables failed to restore
51. DVK_ERR_RESTOREFAIL - All of the tables failed to restore
52. DVK_ERR_DUPRECKEYS - Duplicate keys in a batch
53. DVK_ERR_DUPDBDESCS - Duplicate dbdescs given
54. DVK_ERR_DUPFIELDNAMES - Duplicate field names given
55. DVK_ERR_THESNOTFOUND - Thesaurus not found
56. DVK_ERR_NORLINKMODEL - No Learn model
57. DVK_ERR_RLINKMODELNOTFOUND - Learn model not found
58. DVK_ERR_MODEL_DATA - Binary Learn model data is corrupt or invalid

59. DVK_ERR_NOFEATUREVEC - No feature vector given to the Learn model

60. DVK_ERR_NUMFEATURESMISMATCH - The number of features in the given feature vector does not match the number of features used to train the Learn model

61. DVK_ERR_NOCHARMAP - No character map with the given name

62. DVK_ERR_NOFUNCTION - No callback function

63. DVK_ERR_NOQUERY - No search query

64. DVK_ERR_FILEFORMAT - File format is invalid

65. DVK_ERR_IOERROR - I/O Error

66. DVK_ERR_EXPIRED - Trial expired

67. DVK_ERR_LOOKUP - Lookup error

68. DVK_ERR_NOSTATUS - The NULL error

69. DVK_ERR_NOMEM - Out of memory

70. DVK_ERR_INTERNAL - Unexpected internal error

71. DVK_ERR_EXPECTKEYFIELD - Key field not identified in file load

72. DVK_ERR_EXPECTPARENTKEYFIELD - Parent key field not identified in file load

73. DVK_ERR_CL_CONFIG - The configuration file contains errors.

74. DVK_ERR_CL_CFG_CONFLICT - The new configuration conflicts with the current configuration.

75. DVK_ERR_GATEWAY - The engine is not a cluster gateway.

76. DVK_ERR_JOINSET - An incomplete or invalid set of joined tables was detected

77. DVK_ERR_TRAN_UNKNOWN - An explicit transaction ID was given, but it is not an existing open transaction.

78. DVK_ERR_TRAN_IN_USE - This operation was rejected because another operation using this transaction ID is running

79. DVK_ERR_TRAN_LIMIT - Too many open transactions

80. DVK_ERR_TRANCONFLICT - An attempt was made to modify an object that is being modified by another transaction

81. DVK_ERR_ACTCONFLICT - An attempt was made to perform an action that conflicts with an earlier action in this transaction

82. DVK_ERR_HASERRORS - An attempt was made to commit a transaction that has errors on it

83. DVK_ERR_CHKPT_CORRUPT - The file system containing the checkpoint system is damaged.

84. DVK_ERR_OUTPUTLIMIT - Query output size limits were exceeded.

85. DVK_ERR_FILE_NOT_FOUND - Expected file was not found.

86. DVK_ERR_LOOKUP_FLDS_NEEDED - SORT or PSI lookup fields are required for this query.

87. DVK_ERR_QLETREFBROKEN - The querylet referenced does not exist.

88. DVK_ERR_QLETREFLOOP - A querylet reference loop exists.

89. DVK_ERR_MODEL_UNSUPPORTED - The model version does not support the operation.

90. DVK_ERR_QUERYLET_LIMIT - Number of leaf querylets exceeds the limit.

91. DVK_ERR_GPU_DEVICE_NOT_ALLOWED - The GPU device is not supported, or is disallowed on the command line.

92. DVK_ERR_REMOTE_DISABLED - Remote commands are disabled.

93. DVK_ERR_SSL - Error performing SSL/TLS operations.

94. DVK_ERR_CMD_CANCELLED - Command was canceled.

95. DVK_ERR_JOURNAL - Error performing journal operations

96. DVK_ERR_GPU_DEVICE_NOT_ALLOWED - The GPU device is not supported, or is disallowed on the command line.

97. SVR_ERR_IPCEOF - Unexpected EOF

98. SVR_ERR_IPCTIMEOUT - Socket I/O operation timed out

99. SVR_ERR_IPCERR - I/O error on socket

100. SVR_ERR_HANDSHAKE - Protocol handshake failed

101. SVR_ERR_PROTOCMD - Protocol command not recognized

102. SVR_ERR_REQUESTFMT - Protocol request format error

103. SVR_ERR_BYTECOUNT - Protocol byte count out of range

104. SVR_ERR_LINECOUNT - Protocol line count out of range

105. SVR_ERR_LINELENGTH - Protocol line too long

106. SVR_ERR_PARAM - Protocol parameter not recognized

107. SVR_ERR_PARAMFMT - Protocol parameter format error

108. SVR_ERR_PARAMVAL - Protocol parameter value illegal

109. SVR_ERR_PARAMTYPE - Protocol parameter type mismatch

110. SVR_ERR_NODATABASES - No databases to search

111. SVR_ERR_NOQUERY - No search query

112. SVR_ERR_EXTRAINPUT - Unexpected additional input

113. SVR_ERR_LOGFILE - Log file could not be opened/written

114. SVR_ERR_ZEROLEN - Zero length searchable text

115. SVR_ERR_NOSTATUS - Unparsable status line

116. SVR_ERR_LAYOUT_CONFLICT - Object exists on an overlapping layout.

117. SVR_ERR_NODE_CONFLICT - Conflicting errors from multiple nodes.

118. SVR_ERR_NODE_DOWN - Node communications failed.

119. SVR_ERR_NODE_DATA_SYNC - Objects out of sync between nodes.

120. SVR_ERR_OTHER_NODE - Node operation aborted due to failure on an operational partner node.

121. SVR_ERR_NO_DEFAULT_LAYOUT - No default layout and no layout specified.

122. SVR_ERR_UNKNOWN_LAYOUT - Layout does not exist.

123. SVR_ERR_DEFERRED - An error occurs, but the specific code is not known yet.

124. SVR_ERR_SHUTDOWN - Performs server shutdown

Punctuation and Whitespace Code Points Mapped by Built-in Character Maps

This appendix includes the following DVK_CMP_STDNAME maps several punctuation code points to the blank character (0x20):

Values in the ASCII Range:

Decimal Value	Hex Value	Character
33	0x21	!
34	0x22	“
35	0x23	#
36	0x24	$
37	0x25	%
39	0x27	‘
40	0x28	(
41	0x29	)
42	0x2A	*
43	0x2B	+
44	0x2C	,
45	0x2D	-
46	0x2E	.
47	0x2F	/

Decimal Value	Hex Value	Character
58	0x3A	:
59	0x3B	;
60	0x3C	<
61	0x3D	=
62	0x3E	>
63	0x3F	?
64	0x40	@
91	0x5B	[
92	0x5C	\
93	0x5D	]
94	0x5E	^
95	0x5F	_
96	0x60	`
123	0x 7B	{
124	0x7C	|
125	0x7D	}
126	0x7E	~

Values (hexadecimal) above the ASCII range:

Decimal Value	Hex Value	Unicode Character Name
894	0x037E	GREEK QUESTION MARK
903	0x0387	GREEK ANO TELEIA
1417	0x0589	ARMENIAN FULL STOP
1418	0x058A	ARMENIAN HYPHEN
1470	0x05BE	HEBREW PUNCTUATION MAQAF
1475	0x05C3	HEBREW PUNCTUATION SOF PASUQ
1548	0x060C	ARABIC COMMA
1563	0x061B	ARABIC SEMICOLON
1567	0x061F	ARABIC QUESTION MARK
1748	0x06D4	ARABIC FULL STOP
1792	0x0700	SYRIAC END OF PARAGRAPH
1793	0x0701	SYRIAC SUPRALINEAR FULL STOP
1794	0x0702	SYRIAC SUBLINEAR FULL STOP
1795	0x0703	SYRIAC SUPRALINEAR COLON
1796	0x0704	SYRIAC SUBLINEAR COLON
1797	0x0705	SYRIAC HORIZONTAL COLON
1798	0x0706	SYRIAC COLON SKEWED LEFT
1799	0x0707	SYRIAC COLON SKEWED RIGHT

Decimal Value	Hex Value	Unicode Character Name
1800	0x0708	SYRIAC SUPRALINEAR COLON SKEWED LEFT
1801	0x0709	SYRIAC SUBLINEAR COLON SKEWED RIGHT
1802	0x070A	SYRIAC CONTRACTION
1804	0x070C	SYRIAC HARKLEAN METOBELUS
2040	0x07F8	NKO COMMA
2041	0x07F9	NKO EXCLAMATION MARK
2404	0x0964	DEVANAGARI DANDA
2405	0x0965	DEVANAGARI DOUBLE DANDA
2053	0x0805	SAMARITAN LETTER BAA
3674	0x0E5A	THAI CHARACTER ANGKHANKHU
3675	0x0E5B	THAI CHARACTER KHOMUT
3848	0x0F08	TIBETAN MARK SBRUL SHAD
3853	0x0F0D	TIBETAN MARK SHAD
3854	0x0F0E	TIBETAN MARK NYIS SHAD
3855	0x0F0F	TIBETAN MARK TSHEG SHAD
3856	0x0F10	TIBETAN MARK NYIS TSHEG SHAD
3857	0x0F11	TIBETAN MARK RIN CHEN SPUNGS SHAD
3858	0x0F12	TIBETAN MARK RGYA GRAM SHAD
4170	0x104A	MYANMAR SIGN LITTLE SECTION

Decimal Value	Hex Value	Unicode Character Name
4171	0x104B	MYANMAR SIGN SECTION
4961	0x1361	ETHIOPIC WORDSPACE
4962	0x1362	ETHIOPIC FULL STOP
4963	0x1363	ETHIOPIC COMMA
4964	0x1364	ETHIOPIC SEMICOLON
4965	0x1365	ETHIOPIC COLON
4966	0x1366	ETHIOPIC PREFACE COLON
4967	0x1367	ETHIOPIC QUESTION MARK
4968	0x1368	ETHIOPIC PARAGRAPH SEPARATOR
5741	0x166D	CANADIAN SYLLABICS CHI SIGN
5742	0x166E	CANADIAN SYLLABICS FULL STOP
5867	0x16EB	RUNIC SINGLE PUNCTUATION
5868	0x16EC	RUNIC MULTIPLE PUNCTUATION
5869	0x16ED	RUNIC CROSS PUNCTUATION
6100	0x17D4	KHMER SIGN KHAN
6101	0x17D5	KHMER SIGN BARIYOOSAN
6102	0x17D6	KHMER SIGN CAMNUC PII KUUH
6106	0x17DA	KHMER SIGN KOOMUUT
6146	0x1802	MONGOLIAN COMMA

Decimal Value	Hex Value	Unicode Character Name
6147	0x1803	MONGOLIAN FULL STOP
6148	0x1804	MONGOLIAN COLON
6150	0x1806	MONGOLIAN TODO SOFT HYPHEN
6152	0x1808	MONGOLIAN MANCHU COMMA
6153	0x1809	MONGOLIAN MANCHU FULL STOP
6468	0x1944	LIMBU EXCLAMATION MARK
6469	0x1945	LIMBU QUESTION MARK
7002	0x1B5A	BALINESE PANTI
7003	0x1B5B	BALINESE PAMADA
7005	0x1B5D	BALINESE CARIK PAMUNGKAH
7006	0x1B5E	BALINESE CARIK SIKI
7007	0x1B5F	BALINESE CARIK PAREREN
7227	0x1C3B	LEPCHA PUNCTUATION TA-ROL
7228	0x1C3C	LEPCHA PUNCTUATION NYET THYOOM TA-ROL
7229	0x1C3D	LEPCHA PUNCTUATION CER-WA
7230	0x1C3E	LEPCHA PUNCTUATION TSHOOK CER-WA
7231	0x1C3F	LEPCHA PUNCTUATION TSHOOK
7294	0x1C7E	OL CHIKI PUNCTUATION MUCAAD
7295	0x1C7F	OL CHIKI PUNCTUATION DOUBLE MUCAAD

Decimal Value	Hex Value	Unicode Character Name
8315	0x207B	SUPERSCRIPT MINUS
8331	0x208B	SUBSCRIPT MINUS
8722	0x2212	MINUS SIGN
11799	0x2E17	DOUBLE OBLIQUE HYPHEN
11802	0x2E1A	HYPHEN WITH DIAERESIS
11822	0x2E2E	REVERSED QUESTION MARK
12448	0x30A0	KATAKANA-HIRAGANA DOUBLE HYPHEN
12539	0x30FB	KATAKANA MIDDLE DOT
42509	0xA60D	VAI COMMA
42510	0xA60E	VAI FULL STOP
42511	0xA60F	VAI QUESTION MARK
43126	0xA876	PHAGS-PA MARK SHAD
43127	0xA877	PHAGS-PA MARK DOUBLE SHAD
43214	0xA8CE	SAURASHTRA DANDA
43215	0xA8CF	SAURASHTRA DOUBLE DANDA
43311	0xA92F	KAYAH LI SIGN SHYA
43613	0xAA5D	CHAM PUNCTUAT
65293	0xFF0D	FULLWIDTH HYPHEN-MINUS
65381	0xFF65	HALFWIDTH KATAKANA MIDDLE DOT

Decimal Value	Hex Value	Unicode Character Name
65282	0xFF02	FULLWIDTH QUOTATION MARK
65287	0xFF07	FULLWIDTH APOSTROPHE
65378	0xFF62	HALFWIDTH LEFT CORNER BRACKET
65379	0xFF63	HALFWIDTH RIGHT CORNER BRACKET

White-Space Code Points Mapped by Built-in Character Maps

Decimal Value	Hex Value	Unicode Character Name
9	0x0009	HORIZONTAL TAB
10	0x000A	LINE FEED
11	0x000B	VERTICAL TAB
12	0x000C	FORM FEED
13	0x000D	CARRIAGE RETURN
32	0x0020	SPACE 133
133	0x0085	NEXT LINE
160	0x00A0	NO-BREAK SPACE
5760	0x1680	OGHAM SPACE MARK
6158	0x180E	MONGOLIAN VOWEL SEPARATOR
8192	0x2000	EN QUAD
8193	0x2001	EM QUAD
8194	0x2002	EN SPACE

Decimal Value	Hex Value	Unicode Character Name
8195	0x2003	EM SPACE
8196	0x2004	THREE-PER-EM SPACE
8197	0x2005	FOUR-PER-EM SPACE
8198	0x2006	SIX-PER-EM SPACE
8199	0x2007	FIGURE SPACE
8200	0x2008	PUNCTUATION SPACE
8201	0x2009	THIN SPACE
8202	0x200A	HAIR SPACE
8232	0x2028	LINE SEPARATOR
8233	0x2029	PARAGRAPH SEPARATOR
8239	0x202F	NARROW NO-BREAK SPACE
8287	0x205F	MEDIUM MATHEMATICAL SPACE
12288	0x3000	IDEOGRAPHIC SPACE

Frequently Asked Questions

Frequently asked questions are listed questions and answers that are supposed to be commonly asked when using TIBCO Patterns.

Is TIBCO Patterns accessible from Node.js?

You can access TIBCO® Patterns from Node.js™ using npm™.
Node.js is a development tool for creating web applications and services based on JavaScript.

Caveat - Application Responsiveness

As with any database-like system, some TIBCO® Patterns calls can take longer to impact the user's perception of application responsiveness. When developing a Node.js application that uses TIBCO® Patterns, the usual measures for long-running functions should be used.

Procedure

1. Install TIBCO® Patterns, npm™, and Node.js.

2. Install npm's `java` package.

3. From the *`TIBCO_HOME`*`/tps/version/java/lib` directory, retrieve `TIB_tps_java_interface.jar`.

4. [Optional]: Create a `wrapper` jar. This must encapsulate `TIB_tps_java_interface.jar` and expose only the functionality required by the application.

5. Access the `wrapper` jar from the JavaScript code by following the npm™ instructions. The wrapper jar is available on the official npmjs website.

The TIBCO Patterns Java API can be accessed directly from JavaScript code using npm™. However, that can lead to complex and hard-to maintain JavaScript code. The Patterns API is complex and has detailed data structures. As a best practice, isolate this complexity by using a `wrapper` jar.

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

Documentation for TIBCO® Patterns is available on the TIBCO® Patterns Product Documentation page.

How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices