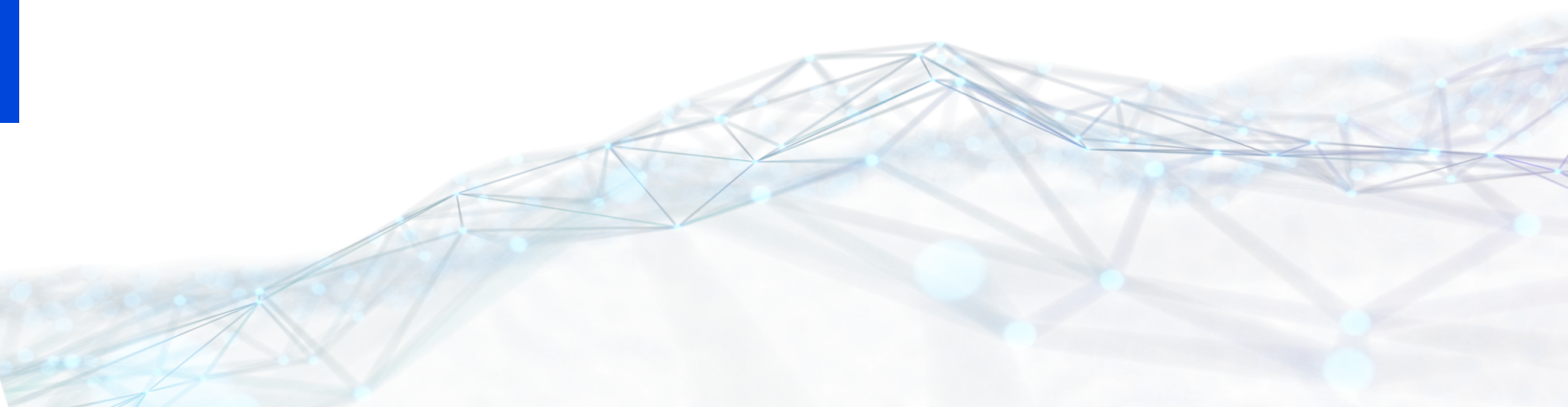




TIBCO® Patterns

Concepts

Version 6.1.2 | June 2024



Contents

Contents	2
Introduction	7
The Importance of “Patterns” to Enterprises	7
TIBCO Patterns	8
A Typical Use Case: Matching People	11
Starting TIBCO Patterns Server	12
Loading a Table of Person Data	13
Loading a Thesaurus of Nicknames	14
Building and Executing Queries	15
TIBCO Patterns Encrypted Communication	22
Maintaining Patterns Tables in Memory	23
Characteristics of In-Memory Tables	23
Field Types	24
Memory Utilization	26
Loading In-Memory Tables	26
Updating In-Memory Tables	27
Checkpoints of In-Memory Data	28
Durable Tables	30
Designing Queries for TIBCO Patterns	32
TIBCO Patterns Queries	32
Simple Queries	33
Cognate Queries	35
Variable Attributes Queries	44
Date Comparisons	46
Predicate Queries	47

Complex Queries	47
Match Case Score Combiner	51
Match Case Score Combiner Rule One: Example 1	52
Match Case Score Combiner Rule One: Example 2	53
Querylet References	58
Weighting Factors	60
Scoring Modes	63
The Score Records Command	65
Thesaurus and Term Weighting	67
Classic Thesaurus Tables	67
Weighted Dictionary	69
Combined Thesaurus Tables	71
Comparing Thesaurus Table Variants	71
Additional Considerations Related to Thesaurus Tables	72
Internationalization and Character Maps	74
Internationalization	74
Character Maps	74
Custom Character Maps	75
Additional Considerations Related to Character Maps	76
Interpreting and Handling Patterns Output	78
The Meaning of a Match Score	78
The Number of Matches Requested	79
The Returned Scores	79
Dynamic Score Cutoffs	80
Tie-breaking Rules	82
Match Visualization	83
Predicates	85
Filtering with Predicates	85

Constructing Predicate Expressions	86
Error Conditions with Predicates	93
Predicates and Performance	94
Predicate Queries	96
Rules to Handle Predicate Errors	98
Transactions	99
Transactions	99
TIBCO Patterns Transactions versus RDBMS Transactions	100
Explicit versus Implicit Transactions	100
Marking versus Locking	101
Marking and Access Rules	103
Dirty Reads and Object Status	104
Transaction Size, Memory, and Performance	105
Joins	106
Overview	106
Joins Example	107
Tables	108
Standard Table	108
Parent Table	108
Child Table	109
Records	110
Standard Records	110
Parent Records	110
Child Records	111
Compound Records	112
Joined Searches	113
Specifying Field Names	117
Joined Records	120
Join Search Modes and Types	124
Matching Compound Records and Querylet Grouping	127

Predicate Indexes and Joined Searches	131
Performance Considerations	132
Special Considerations	136
Working with Mixed Record Types	138
Prefilters and Scaling	140
Prefilters	140
Scaling for Large Data Sets and Workloads	143
Multithreading	143
Horizontal Scaling and Failover Replication	144
Clustering	144
TIBCO Patterns Machine Learning Platform	146
TIBCO Patterns Machine Learning Platform	146
TIBCO Patterns Machine Learning Platform: Basic Concepts	147
Sample Problem: Record Equivalence	147
Supervised Learning without Explicit Rules	151
Characteristics of TIBCO Patterns Machine Learning Models	152
How to Use TIBCO Patterns Learn Models	154
How to Use Learn Models to Evaluate Feature Vectors	154
How to Use Learn Models for Record Matching	155
Handling Low Confidence Predictions	155
How to Use Learn Models on a Cluster	159
Query Builder and Query Builder Platform	160
An Introduction to Query Builders	160
An Introduction to the Query Builder Platform	161
Using the Query Builder Platform to Test a Query	162
Using the Query Builder Platform in an Application	163
Using the Query Builder Platform to Generate Code	165
Integration with Other TIBCO Patterns Tools	165
TIBCO Documentation and Support Services	166

Legal and Third-Party Notices	168
--	------------

Introduction

This section introduces you to the concepts behind TIBCO® Patterns products and a typical use case.

- [The Importance of “Patterns” to Enterprises](#)
- [TIBCO Patterns](#)
- [A Typical Use Case: Matching People](#)
- [TIBCO Patterns Encrypted Communication](#)

The Importance of “Patterns” to Enterprises

In today’s globally networked world, enterprise data comes from many sources and is “used” by so many different people, organizations, and systems. Data is never 100% perfect, never 100% consistent, never 100% complete - and it never can be. Human beings are good at spotting relevant *patterns* embedded in the information and thus making sense out of imperfect data, despite inconsistencies, errors, differences in representation, and so forth.

The last several decades have seen the development of infrastructure to capture, transport, and store enterprise data in ever-growing repositories. Traditionally, SQL queries have been used to retrieve and analyze this data retrospectively. SQL queries can be efficient and function adequately when the data is perfect and you know precisely what data you need. In the real world, where data and queries are never consistent or correct, what happens when exact matching using SQL queries cannot find the data you need?

The underlying problem is that of gauging the similarity of different representations of what is the same entity. What is required is an accurate way of gauging that similarity to deal with imperfect data effectively – as human beings do.

Several techniques – some of them very old – have been developed to deal with imperfect data. All suffer from significant limitations and weaknesses. Some methods, like the century-old Soundex algorithm, are applicable and reliable only for certain kinds of data, like names of people. Some, like wildcard or substring matching, require many queries to be performed to get a result. Some, like string edit distance, are too computationally

expensive to deal with variations outside of a tiny edit window. But all of these methods suffer from the fundamental defect of an inadequate concept of similarity – it is easy to construct plausible variations and misspellings for which these mainstream approaches perform either poorly, or not at all. The ability to deal with imperfect data is long overdue for an overhaul.

TIBCO Patterns provides applications with a much richer, more “human” measure of similarity that enables them to take the appropriate course of action. A high level of performance of true inexact pattern matching is needed to effectively deal with the following scenarios:

- When looking for groups of possible duplicate medical records in a Master Patient Index.
- While detecting fraudulent financial transactions or abuse of entitlement claims in government programs.
- To provide a consumer with a likely list of products in response to an imperfectly formed query.

TIBCO Patterns

From a conceptual viewpoint, TIBCO Patterns functions as an in-memory database that provides inexact matching instead of exact matching.

The data is loaded into tables in-memory with rows and columns similar to a database management system (DBMS). The biggest difference is that a DBMS is optimized for exact matching (by indexing certain columns), TIBCO Patterns is designed for measuring similarity across any chosen set of fields.

With exact matching, a record either matches or does not. However, for inexact matching, all records match to some degree. It is a matter of judgment as to how well one record matches compared to another. The patented technology in TIBCO Patterns is designed to make judgments like the way a human would. In fuzzy matching, however, there is never a guarantee that the selection of best matching records exactly corresponds to the set of records a human would select. There is always a possibility that the returned “best” matches might not include a record a human would judge as a best match (this is called a false negative) or that the “best” matches might include a record a human would judge as a poor match (this is called a false positive). The goal of inexact matching is to reduce the number of false negatives and the number of false positives to a minimum. It should be remembered that the minimum cannot be guaranteed to be zero; this is why TIBCO

Patterns does not replace the exact matching capabilities of a DBMS, it is performing a fundamentally different operation.

TIBCO Patterns technology can compute a meaningful similarity score between a query and any record – the similarity score is quite small when there is little similarity between a given record and the query.

For example, assume that a table is loaded with data about people, and you want to find records matching the following query:

Query		
Results	Score	Record Contents
	1.0	Maria Kristina Cassandra
	0.89	Maria Kristna Cassendra
	0.48	Mary K Casand

	0.05	Bill Bailey

The first record in the result set is an exact match, having the highest possible similarity score of 1.0. The second record is very similar to the query - it probably represents the same person. The third and subsequent records are even less similar. Note that even the final record in this list, the very dissimilar “Bill Bailey,” still has a computable score, even though it is very small.

This kind of similarity computation is very different from a typical database search or select function where the record match for the query is always just “yes” or “no,” and the result set is all records for which the answer is true. With inexact pattern matching, the answer is a score based on a measure of similarity, and a result set is a list of records ranked by score.

This means that with inexact pattern matching, unlike the typical exact-match search, the set of results is not automatically defined, since there is often no clear division between “yes” and “no.” Instead, there is generally a “maybe” region characterized by similarity scores that are neither very high (near-exact matches) nor very low (almost no similarity between the record and the query). Capturing and creating value from this intermediate region is the whole point of inexact matching.

Matching Records

Generally, matching records are those that are similar enough to the query of interest. In practice, “similar enough” corresponds to some chosen threshold value of the similarity score. Once this threshold value is determined – based on the business impact and trade-off between false positives (records higher than the threshold that are not true matches) and false negatives (records lower the threshold that are true matches) – the result set then becomes well-defined. Matching records can then be processed in all the ways that search results typically are processed: consumed by another application, displayed in a Web application, or sorted by attributes.

Here is a second example.

Query:	Peter Sellers Herbert Lohm Aleck Guinness		
Results:	Score	Movie	Actors
	0.68	The Ladykillers	Alec Guinness, Peter Sellers, Cecil Parker, Herbert Lom
	0.49	Revenge of the Pink Panther	Peter Sellers, Herbert Lom, Burt Kwouk
	0.45	A Shot in the Dark	Peter Sellers, Elke Sommer, Herbert Lom
	0.43	Return of the Pink Panther	Peter Sellers, Christopher Plummer, Herbert Lom
	0.35	Murder By Death	Peter Falk, James Coco, Peter Sellers, David Niven, Alec Guinness
	0.22	Lawrence of Arabia	Peter O'Toole, Omar Sharif, Alec Guinness, Jose Ferrer

In this example of a movie database search, one movie is found with matches to all six query terms, corresponding to the names of three actors. The next four movies returned match two out of the three actors for the query terms, and the final movie shown in this list matches only one of the actors.

Note the free-form nature of the query. It consists of a single “phrase” with half a dozen terms. The “best matching records” contain all these terms the “records that may match” contain just some, or only one. Search engines often enforce a Boolean “AND” relationship between terms in a query: all the given terms must match the record. To make the search more flexible, some search engines provide a query language that includes OR and AND operators for specifying these relationships explicitly.

TIBCO Patterns inexact matching uses algorithms that themselves tend to score higher records that closely match more query terms instead of records that match fewer. It is the total amount of similarity that matters, not “hard-coded” relationships between query terms. So if a query contains multiple terms, the top of the results list generally contains those records that contain the best-quality similarity to the most terms. In this respect, the beginning of the result list tends to resemble that of an "AND" type search. Later entries in the result list may more closely resemble an "OR" search as they might have only one or two matching terms. The truth is that TIBCO Patterns's inexact matching is neither AND nor OR, being an improvement on both, selecting records with the best overall match to all of the terms given in the query.

A Typical Use Case: Matching People

In this section, you can learn about TIBCO Patterns concepts by working through a set of examples in connection with a typical use case: finding matches in a table representing people. First load a table and then search it, using both simple queries and more advanced queries. You can also learn how to add capabilities to the search such as a thesaurus to catch nicknames and other variants of a name, and predicate expressions to automatically filter the result set based on the contents of a field.

For our purpose in this section, you must work through these examples using Java™, the Patterns NSC command line tool, and some sample programs. The Patterns NSC command line tool allows you to load data from local files into a running instance of TIBCO Patterns. The sample programs demonstrate how to design and submit queries, and visualize results. Only features that directly pertain to our set of examples are illustrated here. For a complete reference on the use of the NSC command line tool, see *TIBCO Patterns Server Command Line Interface*.

- [Starting TIBCO Patterns Server](#)
- [Loading a Table of Person Data](#)
- [Loading a Thesaurus of Nicknames](#)

- [Building and Executing Queries](#)

Starting TIBCO Patterns Server

On Windows, you can start TIBCO Patterns server as a standalone process, or as a Windows service.

Run as a Standalone Process

To run TIBCO Patterns server as a standalone process,

1. From the command prompt, go to the `PATTERNS_HOME\server\bin` folder.
2. Run the following command:

```
TIB_tps_server.exe *.*.*.*
```

For options, see "Running the TIBCO Patterns Server" in *TIBCO® Patterns Installation*.

This starts the server as a standalone process and instructs it to accept connections from clients at any IP address.

Note:

- The server accepts connections from the local host (IP address 127.0.0.1) and any addresses given on the command line (this is called the authentication list). Addresses given on the command line can include wildcards and subnet mask lengths, for instance 129.48.34.*; 192.168.*.*; or 129.48.34.0/24.
- For additional information about TIBCO Patterns command-line arguments, see *TIBCO® Patterns Installation*.

The newly started TIBCO Patterns server listens on the default port 5051

Windows Service

To run TIBCO Patterns as a Windows service, run the Windows `sc` program as Administrator to install the service. The command is as follows:

```
sc create "TIBCO Patterns Server version" binpath= PATTERNS_
HOME\server\bin\TIB_tps_server.exe
```

Note the space after `binpath=`, and ensure that you specify the correct path to the server.

Optionally, select **Install as a Windows Service** during product installation to start TIBCO Patterns server as a Windows service.

Now use the Services applet to control the service called “TIBCO Patterns Server *version*”. You can find the applet in the Control Panel under Administrative Tools, but the position of the applet might differ depending on the version of Windows on your machine. With the applet, you can start and stop the service. To remove the service, use the following command:

```
sc delete "TIBCO Patterns Server version"
```

Loading a Table of Person Data

Here are the steps to load a sample table containing 10,000 rows of person records:

Procedure

1. Review `$PATTERNS_HOME/java/sample/data/persons.csv` using a text editor. For Patterns installed at `$TIBCO_HOME`, `$PATTERNS_HOME` is `$TIBCO_HOME/tps/6.1.2`.
2. Load `persons.csv` in a Patterns table.

```
cd $PATTERNS_HOME

java -jar NSC/bin/NSC.jar -cmd tblload -table person -file
java/sample/data/persons.csv -fieldnamesfirst
```

This lists the statistics for the loaded table, including the number of records.

Result

The output is as below:

```
$PATTERNS_HOME\tps\6.1.2>java -jar NSC/bin/NSC.jar -cmd tblload -table
person -file java/sample/data/persons.csv -fieldnamesfirst
```

Copyright (c) 2000–2024 Cloud Software Group, Inc. All rights reserved.
 Cloud Software Group, Inc. Confidential & Proprietary Information.
 TIBCO® Patterns Server Commands (NSC): Version – 6.1.2

```
>>> nsc start.
List: GENERIC
      DBDESCRIPTOR: person
      OBJ_ID: a62c5e34-3a52-43c7-90aa-6f2163f43b2b
      CHECKPOINT_STATUS: Automatic
      TRAN_ID: 1037
      TRAN_OBJ_STATE: 10
      DBINFO:
      DBNUMFIELDS: 7
      FIELDNAMES: ["last","first","ssn","street","city","state","zip"]
      FIELDTYPES: [ 5, 5, 5, 5, 5, 5, 5]
      FIELDBYTECOUNTS: [ 63348, 50387, 45108, 170818, 88926, 20028,
49707]
      FIELDCHARCOUNTS: [ 63348, 50387, 45108, 170818, 88926, 20028,
49707]
      CHARMAPS:
      ["=STD=", "=STD=", "=STD=", "=STD=", "=STD=", "=STD=", "=STD="]
      DBGIPFILTER: true
      DBGIPGPU: false
      DBSORTFILTER: false
      DBPSIFILTER: false
      DBNUMRECORDS: 10014
      DBKEYTREEKBYTES: 96
      DBRECFIELDKBYTES: 524
      DBHEADERKBYTES: 246
      DBIDXTOTALKBYTES: 3120
      DBTOTALKBYTES: 5406
      TABLE_TYPE: Standard

>>> nsc end.
```

Loading a Thesaurus of Nicknames

Since the data in a sample table includes names, load a thesaurus containing nicknames or variants of a name. A thesaurus lets us equate certain words, or even phrases, that are textually dissimilar such as the name “Margaret” and the nickname “Peggy”.

The thesaurus capability of TIBCO Patterns works in concert with inexact or fuzzy matching based on textual similarity. A certain level of error-tolerance is even supported in the recognition of terms with thesaurus equivalents.

1. Using a text editor, review PATTERNS_HOME/java/sample/data/nicknames.csv.
2. Load nicknames.csv into a Patterns thesaurus

```
cd $PATTERNS_HOME
java -jar NSC/bin/NSC.jar -cmd thcreate -thesaurus nicknames -file
java/sample/data/nicknames.csv -local
```

3. The output is as follows:

```
$PATTERNS_HOME\tps\6.1.2>java -jar NSC/bin/NSC.jar -cmd thcreate -
thesaurus nicknames -file java/sample/data/nicknames.csv -local

Copyright (c) 2000-2024 Cloud Software Group, Inc. All rights
reserved.

Cloud Software Group, Inc. Confidential & Proprietary Information
TIBCO® Patterns Server Commands (NSC): Version - 6.1.2

>>> nsc start.
Thesaurus created
>>> nsc end.
```

Building and Executing Queries

Scenario 1: A Simple Query Across all Fields

The sample program java\sample\QrySimple.java performs a simple search across the full text of a record. You can review the code if required (Java) or run it. Add the name of the table and search text. For example, search the person table created for “jasoz fitgerlad”.

```
cd $PATTERNS_HOME/
java -cp java\lib\TIB_tps_java_interface.jar java\sample\QrySimple.java
person "jasoz fitgerlad"
```

You can see the following results.

key	score	record text
1395	0.662	FitzgereId,Jasom,151507584, 200 NClasson Str,St.Paul,MN,55101
1394	0.662	Fitzgerald,Jason,151607584, 2000 N Classen Blvd,Saint Paul,MN,55101
1396	0.572	FitzgreId,Jasom,151507584, 200 N Classon Str,St.Paul,MN,55101
1397	0.500	FitzgreId,Jassy,151507584, 200 N Classon Str,St.Paul,MN,55101
7511	0.492	Hull,Jason,570766446,40 Ridge Dr Apt 101,Dallas,TX,75234

Results similar to the input are found, even when the text is scrambled.

This very simple search doesn't distinguish between fields. Searching for "paul fitgerlad" yields the following results.

key	score	record text
1395	0.648	FitzgereId,Jasom,151507584, 200 N Classon Str,St. Paul,MN,55101,RGQY-8946-WPII
5736	0.598	Drooking,Paul,,1 Ford Pl # 2e, Columbus,GA
1642	0.549	Litzinger,Paul,570000884,3808 N Tamiami Trl, San Francisco,CA
40	0.543	Sorenson,Paul,942323408,4 Vilalge Dr, Houston,TX,77210
312	0.543	Bearden,Paul,,860 Ridge Lake Blvd, Rochester,MN,55905

Note that "paul" can appear in any field.

Scenario 2: Using a Thesaurus

The sample program `java/sample/src/QryThes.java` also performs a search across all fields, and adds the use of a thesaurus to handle nicknames. You can review the code if

required (Java), or run it. Give it the name of the table, the name of the thesaurus, and search text.

```
cd $PATTERNS_HOME
java -cp java\lib\TIB_tps_java_interface.jar
java/sample/src/QryThes.java person nicknames "peggy sooder"
```

You can see the following results.

key	score	record text
111	0.754	Souders,Margaret,361363143,621 Nw 53rd St,Naperville,IL,60565
5707	0.642	Smith,Ryder,,832 Marguerite Mine Rd,Knoxville,IA,50138
6377	0.624	Shoults,Margaret,135155146,104 Whippoorwill Dr,Mobile,AL,36619
4569	0.622	Wolf,Margaret,812653366,7420 Grace Dr,Scottsdale,AZ,85260
8550	0.610	Sun,Peggy,,21093 Vinton Lane,Overland Park,KS,66212

For comparison, you can run QrySimple.java on "peggy sooder" to see results without the thesaurus.

```
cd $PATTERNS_HOME
java -cp java\lib\TIB_tps_java_interface.jar
java/sample/src/QrySimple.java person "peggy sooder"
```

You can see the following results.

key	score	record text
8550	0.610	Sun,Peggy,,21093 Vinton Lane,Overland Park,KS,66212
1307	0.603	Beam,Peggy,,11 W Mohawk Dr,Oxford,MS,38655
9741	0.496	Sonderman,Jeff,,Po Box 2900,Miami,FL,33166
6326	0.494	Hehr,Peggy,297933019,5300 Kings Island Dr,North Bend, WA,98045
3344	0.453	Goodpaster,Don,,Po Box 687,Tulsa,OK,74134

The results using the thesaurus are precise.

Scenario 3: Targeting Specific Fields

In many use cases, it is required that the specific portions of a query target different fields of the table.

Terminology:

- A query is atop-level search submitted to the Patterns server.
- A querylet or sub-query is a piece of a query.
- A combiner is formed using querylets. Combiners often have options (such as weighting).

A query can be either a querylet or a (most commonly) combiner.

For example, you want to find a record belonging to David Brown who lives on Bath Street.

The sample program `java/sample/src/QryTargeted.java` performs a field-targeted search. You can review the code if required (Java) or run it. Give it the name of the table, the first name, the last name, and the street.

```
cd $PATTERNS_HOME
java -cp java\lib\TIB_tps_java_interface.jar
java/sample/src/QryTargeted.java person dvid brwn bath
```

You can see the following results.

key	score	record text
6232	0.95	Brown,David,695674330,3161 Bath Pike,Chesapeake,VA,23320
8456	0.70	Brown,David,,931 14th St # 1310,Lakeland,FL,33813
1539	0.63	Brown,David,445772053,4313 Grainwood Cir Ne,Seattle,WA,98124
9973	0.59	Erwin,David,,1232 7th Ave,Kirkland,WA,98083
5616	0.58	Winters,David,323046791,508 Dartmouth Crossing Dr,Palm City,FL,34990

Notice multiple David Browns are available, all of them found despite the multiple misspellings of his name. However, only one of these David Brown lives at an address containing “bath”. This record has a substantially higher score..

The sample program `java/sample/src/QryTargeted.java` makes use of Patterns' "AND" operator. This may remind you of the "AND" operator from other query languages. But unlike the hard-coded "AND" relation that forcibly includes multiple terms, the TIBCO Patterns "AND" operator behaves in a "soft" manner. It is a combiner that computes a weighted average.

Scenario 4: Using a Predicate to Filter Records

At times it is necessary to filter certain kinds of records out of the result set, based on the content of a certain field.

For example, suppose you wish to find all the people with the last name of 'Browne' who live in the state of Oregon ('OR').

```
cd $PATTERNS_HOME
java -cp java\lib\TIB_tps_java_interface.jar
java/sample/src/QryPredFilter.java person browne OR
```

You can see the following results.

key	score	record text
9316	1.00	Browne,Rennel,,4975 Lacross Rd Ste 20,Eugene,OR,97401
3286	0.91	Brown,T,757694401,706 Mission St Fl 8,Milwaukie,OR,97222
1280	0.76	La Brie,Bill,983987453,Technology Division,Terrebonne,OR,97760
1116	0.73	Bristow,Donald,473762663,12030 Pasteur Dr Apt 413,Corvallis,OR,97333
3661	0.72	Rowley,Roger,,5733 Central Ave,Portland,OR,97233

Note how the predicate filter works in concert with inexact matching. The string specified in the predicate filter must match the "state" field exactly. At the same time, the usual inexact matching is applied to the querylet: "Brown" records are returned (only Browns living in California) despite the misspelling of the name.

TIBCO Patterns offers a rich set of options for constructing complex predicate expressions. See [Predicates](#) for more information.

Scenario 5: Alternate Conditions

Sometimes it is necessary to find that match one at least one of several conditions. The sample program `java/sample/src/QryOR.java` does this. Using TIBCO Patterns' OR operator, it searches for records with a particular last name or a particular city. For example, searching for the last name "Xu" or city "Russell".

```
cd $PATTERNS_HOME
java -cp java\lib\TIB_tps_java_interface.jar java/sample/src/QryOr.java
person xu russell
```

You can see the following results.

key	score	record text
1186	1.00	Soltys,John,715187409,1200 Lower River Rd Nw,Russell,KS,67665
6688	1.00	Xu,Brian,585404059,2662 Holcomb Bridge Rd,Mentor,OH,44060
7850	1.00	Suttin,Barry,856722824,334 Tasf,Russell,KY,41169
8765	1.00	Xu,James,,1732 Elmwood Dr,Charlotte,NC,28277
1275	0.56	Ellister,Mark,,2230 E Imperial Hwy,Roswell,GA,30076

Scenario 6: Targeted Queries with Cross-field Matching

Different parts of personal names are frequently entered incorrectly in fields. Some last names sound like first names or vice versa. Sometimes multiple surnames are entered in different name fields. Targeted queries are useful, but with cases like name fields, you might want to target specific fields, while simultaneously permitting selective cross-field matching.

The sample program `java/sample/src/QryCognate.java` does this. Using TIBCO Patterns' Cognate feature, it searches for specifically the last-name and first-name fields, while accounting for data that has a cross between those two fields.

```
cd $PATTERNS_HOME
java -cp java\lib\TIB_tps_java_interface.jar
java/sample/src/QryCognate.java person Sam Sang 1.0
```

You can see the following results.

key	score	record text
1636	0.97	Sangsom,Sam,,2015 Manhattan Beach Blvd,San Diego,CA,92154
6338	0.75	Nam,Sang,,5445 Dtc Pkwy Ste 600,Sunnyvale,CA,94088
903	0.67	Sarmad,Sam,300461920,12479 Research Pkwy,Salt Lake City,UT,84112
1053	0.66	Sabesan,Sriraman,821833572,2933 Bunker Hill Ln,Foster City,CA,94404
6070	0.65	Goldstein,Sam,,16106 La Avenida Dr,Silver City,IA,51571

In the above search, you passed a first-name, last-name, and a cross-field weight. Here, 1.0 indicates full weight was given to cross-fielded data. If you reduce that weight, you see reduced scores on the results with cross-fielded data:

```
java -cp java\lib\TIB_tps_java_interface.jar
java/sample/src/QryCognate.java person Sam Sang 0.8
```

You can see the following results.

key	score	record text
1636	0.97	Sangsom,Sam,,2015 Manhattan Beach Blvd,San Diego,CA,92154
903	0.67	Sarmad,Sam,300461920,12479 Research Pkwy,Salt Lake City,UT,84112
1053	0.66	Sabesan,Sriraman,821833572,2933 Bunker Hill Ln,Foster City,CA,94404
6070	0.65	Goldstein,Sam,,16106 La Avenida Dr,Silver City,IA,51571
6338	0.60	Nam,Sang,,5445 Dtc Pkwy Ste 600,Sunnyvale,CA,94088

TIBCO Patterns Encrypted Communication

TIBCO Patterns supports SSL/TLS encrypted communication between the Server and Client applications. By default, the server accepts both encrypted and unencrypted connections. The server's acceptance of encrypted or unencrypted communications can be disabled by using a command line option. Also, all external interfaces support both encrypted and unencrypted connections.

API Connection

The server can accept plaintext connections from the usual APIs fairly quickly (hundreds or thousands of connections per second). SSL connections are substantially slower to set up (tens per second).

Normally, the connection speed is not a major factor because connections are persisted. Disabling persistent connections in combination with enabling SSL might lead to significant performance loss.



Note: A failed command results in the connection being closed.

Load Balancer Support

The TIBCO Patterns server and all interfaces support SSL/TLS through a load balancer. Consult your load-balancer documentation to configure such features as SSL-pass-through and SSL-session-resume.

Using SSL/TLS through a load-balancer might impact performance, due to the load-balancer performing additional encryption/decryption. For more information, see load-balancer documentation.

Maintaining Patterns Tables in Memory

TIBCO Patterns maintains data in in-memory tables. This enables the TIBCO Patterns algorithms to process data at in-memory speeds, with no round trips to a DBMS, and no impact on existing operational systems.

TIBCO Patterns makes it easy to maintain data in its internal tables. Loading is fast, tables can be updated dynamically, and a checkpointing feature enables quick restoration of tables when a server is restarted after a shutdown.

- [Characteristics of In-Memory Tables](#)
- [Loading In-Memory Tables](#)
- [Updating In-Memory Tables](#)
- [Checkpoints of In-Memory Data](#)

Characteristics of In-Memory Tables

The TIBCO Patterns in-memory table is not a persistent data store; it must be reloaded each time the matching server is restarted. TIBCO Patterns in-memory tables are designed to work with, not to replace, your current DBMS or another persistent data store.

TIBCO Patterns in-memory tables are quite similar to DBMS tables. An in-memory table consists of fielded records, with every record having the same number of named fields. The number of fields and their types are established at the time the table is created, and remain fixed for the lifetime of the table – addition or deletion of fields to an in-memory Patterns table is not supported.

Each record in the table must have a unique key value. Keys can be defined by the application or auto-generated by the TIBCO Patterns server.

A TIBCO Patterns in-memory table has a fixed configuration that is defined at the time of creation. It cannot be changed for the life of the table. The configuration includes the list of fields and the field types, character maps associated with each field, and indexes for the table including the index (prefilter) used for inexact matching and any partition indexes used for predicate tests. It also includes any parent-child relationship between tables.

Field Types

The set of possible field types includes: searchable text, Variable Attributes, non-searchable text, integer, float, date, and date-time.

Searchable Text

Searchable text is the default field type, and by far the most commonly used. Searchable text consists of UTF-8 encoded textual data that is used for inexact matching. Such text can also be used for exact matching within predicate expressions. See [Predicates](#).

Variable Attributes

A field of type Variable Attributes is a container for an arbitrary set of name-value pairs. The value, like searchable text fields, is any UTF-8 encoded text value. The name is treated much as a field name. Each Variable Attribute name-value pair can be referenced by inexact matching queries and exact matching predicates like a standard searchable text field. Unlike standard fields, however, the names of attributes are not predefined and each record can contain a completely different set of attributes.

A table can contain at the most one field of Variable Attributes type. It is an optional field and if present, it is always the last field in the table. This field is named like any other field in the table.

Methods are provided for adding Variable Attributes to records. Methods are also provided for encoding a set of Variable Attributes into a text string that can be included in a standard CSV file. This allows tables containing Variable Attributes to be dumped to and loaded from a CSV file.

A convention for referencing a Variable Attribute value within a table is:

```
<field-name>:<attribute-name>
```

where:

- `field-name` is the name of a field of type Variable Attributes and
- `attribute-name` is the name of a Variable Attribute.

For example, if you define the "**address**" field of type Variable Attributes, it can be defined using many attributes which could be queried using expressions such as: "`address:line1`", "`address:line2`", "`address:city`".

The predicate expression from the example in the [Introduction](#) can be: `$"address:state"`
= `"CA"`

i Note: This convention imposes the restriction that field names cannot contain the ‘:’ (colon) character. This restriction is enforced for all fields.

Although limits are present on the number and size of Variable Attributes within a record and table, you are unlikely to encounter them in common scenarios. For more information, see the "Variable Attributes - Usage Details" section in *TIBCO Patterns Programmer’s Guide*.

Non-searchable Text

Non-searchable text is UTF-8 encoded textual data that cannot be used for inexact matching. However, you can store it in the in-memory table to avoid extra calls to the permanent data store, or use it for exact matching in predicate expressions, or for some other purpose. When inexact matching is not required for textual data, the use of this field type can significantly reduce memory usage and load times.

Integer and Float

Integer and float are integers or floating-point values that are transmitted to the TIBCO Patterns server as text, but stored in the table as binary values. Fields of these types are primarily useful in predicate expressions, in which they are more efficient, and require less storage, than the equivalent text value. Two special values recognized for these field types are:

- empty, which indicates no data in the field;
- invalid, which indicates that the data transmitted for the field could not be successfully translated to the integer or float type. For example, a string value of “apple” is unable to be translated to an integer type, resulting in an invalid value of the field for that record.

Date and Date-time

The date and date-time fields store date and date-time values compactly. You cannot use these fields in the standard inexact queries. However, a specialized inexact matching query for date fields (but not date-time fields) provides better results than matching on the text representation of the date value. You can use both field types in predicate expressions.

A standard date field is not visible to the GIP prefilter (For information about the GIP prefilter, see [Prefilters and Scaling](#)). This can result in a loss of accuracy as the GIP prefilter cannot use the date field information to help select the best matching records. However, a date field can be marked as a searchable date field. A searchable date field is visible to the GIP prefilter. This can improve search accuracy, but increases memory usage. A searchable date field is still a date field, it cannot be used in a standard inexact text matching query.

Memory Utilization

It is a good practice to load only the fields that are required for matching in in-memory Patterns tables. If the table contains additional fields (for example, to avoid extra calls to retrieve records from a DBMS), use the non-searchable text type for these fields to minimize memory usage.

As a general rule of thumb, an in-memory Patterns table consisting entirely of searchable text fields consumes approximately five times the amount of storage represented by those fields. This is because of the specialized indices that are built at load time to support high-performance inexact matching. Non-searchable fields (text, integer, float, date, or date-time) do not utilize any extra memory.

Loading In-Memory Tables

Tables are loaded into TIBCO Patterns products in two ways:

- From a remote source (such as a DBMS) via a TCP socket connection to the TIBCO Patterns server, or
- From files local to the host that is running the TIBCO Patterns server (server-side loading).

This section describes only the common features of loading tables, regardless of the method used to manage this process.

The TIBCO Patterns server optimizes the loading of a table based on the number of records it expects. When more than 40,000 records are to be loaded into a newly created table or added to an existing table – the TIBCO Patterns server employs multiple processors to speed up the loading. (For smaller loads, a low overhead, the single-stream load is performed.)

Loading From Files - Local File System

TIBCO Patterns also comes with easy-to-use utilities for reading client-side data files and preparing the records for transmission to a TIBCO Patterns server. These utilities support comma-separated values .csv files.

The caller can specify an estimate of the number of records to be sent. Based on the estimate, the TIBCO Patterns server uses multiple processors or a lower overhead single processor load. If an estimate is not given, it assumes that a very small number of records are to be loaded and uses the single-processor low-overhead load.

Loading From Files - Server File System

The server can be instructed to read data from files on its file system. The location of these files is restricted and only files in the server's loadable-data directory can be read.

When loading data directly from a .csv file, the TIBCO Patterns server assumes a large number of records are to be loaded and employs multiple processors to hasten the load.

Updating In-Memory Tables

TIBCO Patterns supports addition, lookup, updating, and deletion of records in existing in-memory tables. Like the initial loading of records into a new table, record additions might flow to the server over a socket connection from a remote source or a server-side local file. Updates of the existing records can only be performed over a socket connection and updates direct from a server-side file are not supported.

Any operation that modifies a table causes all other operations that would either read or modify that table to be blocked (table-level locking). Such operations are queued and automatically unblocked when the table-modifying command is completed.

Applications that do not require continual data updates might find it simpler to replace an entire table with a new version according to an appropriate schedule. This is easily accomplished by loading the new version of the table under an alternate name. While this load is progressing, queries remain enabled on the previous version. When the load of the new version is completed, it can be renamed from the alternate name to the standard name of the table. This effectively replaces the previous version of the table with the new version without appreciable down time.

Considerations

It requires enough memory to hold two copies of the table in memory at the same time. Therefore, it is a good option for small to mid-sized tables that are updated by the way of a periodic batch run.

This method cannot be used to update a joined set of tables. For more information about joined tables, see [Joins](#).

This method of swapping tables can also cause memory growth due to fragmentation of memory. Although it is not a memory leak, fragmentation can result in a process accumulating large amounts of unusable memory. Frequently reloading and swapping tables might lead to excessive memory usage due to fragmentation, which might require periodically restarting the TIBCO Patterns server.

Checkpoints of In-Memory Data

You can save in-memory data tables, and other in-memory objects (thesauri, custom character maps, and Learn Models) to a permanent storage device. The saved copies, or snapshots, can be easily restored at a later time. This feature is called *checkpointing*. Its primary purpose is to permit efficient recovery of the data in tables when the TIBCO Patterns server is restarted after a planned shutdown.



Warning: The checkpoint/restore feature is deprecated in favor of the durable-data feature.

A typical usage scenario is to checkpoint all in memory objects after the initial load is complete. If you perform batch updates of a table, the table should be checkpointed after each batch update. A checkpoint is a snapshot of the object as it exists at the time of the checkpoint. No history of updates is kept. Thus, if updates on a table are performed after a checkpoint of the table, or if a thesaurus or Learn Model is replaced after it was last checkpointed, those updates are not reflected when the table, model, or thesaurus is restored from the checkpoint.

If a controlled shutdown of the TIBCO Patterns server is planned, all in-memory objects can be checkpointed before the planned shutdown. On restart, direct the server to restore all checkpointed objects. This method is far more efficient than reloading the content of a large table from the original source.

Here are some additional points to keep in mind about the checkpoint feature:

- Checkpointing and restoring operations are not automatic, but are controlled through explicit commands.
- Checkpointing does not provide a history of versions of a table. There is at most one checkpoint for a table – the latest performed. Again, even with the checkpointing feature, the TIBCO Patterns in-memory tables are not a substitute for a full-fledged DBMS or other persistent data store.
- Checkpointing operations are safe. This means that if the system crashes during a checkpointing operation on a table, the previous checkpoint of that table remains intact (assuming, of course, the physical device on which the checkpoint is stored remains intact). It also means that if an operation such as a deletion or a renaming of a checkpointed table fails, the checkpoint of that table remains intact.
- Checkpoint operations are very fast. They generally operate at or close to the maximum write speed of the device they are writing to. Checkpoints on small or mid-sized tables are generally non-intrusive. However, frequent checkpoints on large tables that have a combination of record update and queries in progress, can cause a significant drop in performance.

Checkpoints of very large tables (approximately 100 million records or more) can impact performance. Checkpoints of such large tables should only be done when the system is inactive.

- Restoring a checkpointed table after a server restart is significantly slower than the checkpointing operation, and thus potentially more intrusive. However, the restore operation generally has no effect on operations performed on other tables.
- Renaming a table renames any associated checkpoint file. For example, if a table named "table_A" is loaded, a checkpoint is performed and then the table is renamed to "table_B". A restore of "table_A" fails because there is no longer any such checkpoint, but a restore of "table_B" succeeds as the checkpoint is now named "table_B".
- Deleting a table deletes all associated checkpoint files. Thus, if you delete a table that was checkpointed, you cannot recover it by doing a restore. Checkpointing is not intended as a backup mechanism, it is designed for reloading tables on a restart.
- If a table uses a custom character map the character map must be loaded before the table is restored. If the character map has been checkpointed, instructing the server to restore all objects loads the checkpointed custom character maps before loading any table that uses it. Therefore, it is a good practice to checkpoint the custom character maps if tables are checkpointed and a restore all operation is used. If the application chooses to restore objects individually, it is up to the application to

ensure that any custom character maps are restored before any table or thesaurus that uses it.

- Checkpointed data is NOT encrypted. The checkpoint directory, and any files in the directory, must be protected from unauthorized access.

Durable Tables

Data tables and other in-memory objects (thesauri, custom character maps, and Learn Models) can be automatically saved to a permanent storage device. This feature is called durable data. Its primary purpose is to permit efficient recovery of up-to-date data when the TIBCO Patterns server is restarted after a planned or unplanned shutdown.

Some additional points to keep in mind about the durable data feature:

- Durable Data is automatic.
- Durable Data supersedes the Checkpoint/Restore feature. When durable data is enabled, the checkpoint and restore commands do nothing.
- Durable data persists in all data: tables, thesauri, custom character maps, and Learn Models.
- Durable Data does not provide a history of data, only the current data.
- Durable Data is safe. This means data changes are handled within a transaction system and are immediately entrusted to the operating system. If the software crashes, the operating system is still able to update the persistent storage device.
- Durable data is fast. It has only a modest impact on the speed of update operations. Very large updates (100 million records or more) must be done only when the system is inactive.
- Durable Data is automatically reloaded when the server is restarted.
- Deleting a table, thesaurus, or Learn Model deletes all associated durable data.
- Durable data is NOT encrypted. The persistence directory, and any files in it, must be protected from unauthorized access.
- On Windows, the Durable-Data feature imposes a limit of 100 tables.
- On Linux, the Durable-Data feature imposes a limit on the number of tables based on the value reported by `ulimit -n`. Divide this value by 100 to obtain the limit on the number of tables.

- If there are large tables (over 10M records), loading the data at server start-up can take noticeable time. Applications may encounter a timeout error if they connect to TIBCO Patterns while it is loading large tables at start-up.

Designing Queries for TIBCO Patterns

This section covers all the details on how to design queries for TIBCO Patterns, all the way from simple unstructured queries to complex targeted queries using score combiners.

- [TIBCO Patterns Queries](#)
- [Simple Queries](#)
- [Cognate Queries](#)
- [Variable Attributes Queries](#)
- [Date Comparisons](#)
- [Predicate Queries](#)
- [Complex Queries](#)
- [Scoring Modes](#)
- [The Score Records Command](#)

TIBCO Patterns Queries

A TIBCO Patterns query performs a fuzzy search for records that are most similar to the query string. A TIBCO Patterns query does not return a specific set of records that meet a specific set of criteria, instead it returns a set of records that are judged to be most similar to the specified query. A TIBCO Patterns query is not an SQL query with a few new operators; its behavior is fundamentally different. An SQL query has a specific set of records that meet the conditions defined by the query. For a TIBCO Patterns query no specific set of records are specified, all records match the query, the only difference lies in the degree of similarity.

When designing a TIBCO Patterns query one should always keep in mind that the purpose of the query is to provide enough information to allow TIBCO Patterns to distinguish the records you want to see from those you do not. A common mistake made when designing a TIBCO Patterns query is to translate a set of SQL queries directly into what appears to be the corresponding TIBCO Patterns constructs. These SQL queries were developed to get

around the lack of fuzzy matching in SQL by doing a series of queries and merging the results using a set of coded rules. This type of direct translation has two major issues:

1. Each of the separate SQL queries probably provides too little information for TIBCO Patterns to make a reasonable selection. A query on just a person's name, followed by a query on just a street address, and then a merge of common records in the returned lists is far less likely to yield the results you want than a single query on both name and address.
2. Issuing multiple queries is far less efficient than issuing a single query with more data. A fuzzy search is generally more expensive than an SQL exact match search. A single well designed TIBCO Patterns query, however, can often replace a whole set of SQL searches and give better results, making it more efficient overall.

As a general rule, one should attempt to use a single query that contains all of the available data, rather than two or more queries that contain subsets of the data available.

Although the TIBCO Patterns query API provides exact matching capabilities via the predicate query type, all queries performed by TIBCO Patterns are fuzzy queries, and therefore must contain fuzzy matching data. A query that consists only of exact match predicate queries is not allowed. Obvious cases are rejected by the server. Although it is possible to circumvent the query structure validations through various means, a query that is effectively based only on a predicate query does not return the desired results. TIBCO Patterns is a fuzzy matching engine.

Simple Queries

A query consists of a fixed data item that one wants to compare with a set of items or records in some collection of data. A query might be a single, unstructured, text string such as the interactive queries that a user types into a search box. It can also be a structured item similar to the fielded records themselves. The parts of a structured query are usually targeted against the corresponding fields of the records. (Consider the “advanced search” function often provided by many search engines.)

An unstructured query consisting of a single text string is called a simple query. With TIBCO Patterns, you can compare a simple query against any or all of the searchable-text attributes of a table.



Note: Here, the term "attribute" refers to a searchable element of the record. It might be a field of the record or it might be a named Variable Attribute.

For each record, the TIBCO Patterns matching algorithm finds the best match across the selected attributes. The resulting match score – a value between 0.0 and 1.0 – reflects the contributions of the whole or partial matches found in each field. You can also apply "weighting" to adjust the score contributions made by the material matched in the specific fields.

For example, suppose an online bookseller provides a catalog search based on three fields: Author, Book Title, and Book Description. By default, a simple, unstructured query against these three fields weight match contributions equally from any of the three. But you would likely consider content matched in the first two fields to be more significant than content matched in the third field. Therefore, retain the default field weight of 1.0 for the first two fields, and assign a lower weight such as 0.8 to the Book Description field. (Field weights, like the match score, are floating point values between 0.0 and 1.0).

Query matching is performed first, uninfluenced by weighting factors. After a matching is complete, the weighting factors are applied in computing the final score.

Consider the following additional points when weighting simple query comparisons against multiple fields:

- Lowering the weight of a field from the default value of 1.0 reduces the score of records with matches in that field, even if they are perfect matches.

All else being equal matches in lower weighted fields will receive lower scores than the same match in a higher weighted field. In general, the highest score a match into a particular field can receive is the field weight for that field.

In terms of our example, a simple query that perfectly matched a phrase in the Book Description field would receive a lesser score for that record than for another record in which it perfectly matched the Book Title.

- A set of weights that appear right for one particular query might not be right for other queries with different data or structures. Hence, when adjusting the field weights, be sure to test the selected set of weights using a variety of queries.
- If a query string is empty, or if all the values to be searched are empty, there is no information on which to base a match score. In this case, the query is assigned the "empty score". The empty score defaults to 0.0, but you might set this to any value between 0.0 and 1.0. In the vast majority of cases, 0.0 is the appropriate empty score, but in some cases it might be more appropriate to set this to some intermediate value. Consider a case when querying against a Variable Attribute where records might or might not contain that attribute. You might not want to penalize records that do not contain the attribute (see the sections on AND score combiner and Attributes queries for other means of dealing with missing data).

Using a special score you can reject a record entirely when the "empty" condition is encountered. This is quite different than a zero score, especially when complex queries are used. It is possible that a record with a zero score might be returned, if enough records with non-zero scores are not available to fill the requested return set size, and if this is part of a complex query, the final score might be quite high even if this component of the complex query has a zero score. However, if the special reject score is used (in the AND combiner) the record is not returned.

A couple of other options are available with simple queries that are described in detail in later sections:

- You can define a thesaurus to be used for a particular simple query, even if the simple query is just one part of a more complex query structure. See the section [Complex Queries](#) for details on complex queries.
- Similarly, you can define the scoring mode (such as normal or symmetric) to be used for a particular simple query, even if the simple query is just one part of a more complex query structure.
- You can query on specific attributes in a Variable Attributes type field. See [Variable Attributes Queries](#) for details.

Cognate Queries

Often, queries are *structured*, that is, the query contains several separate data items that must be matched against different fields of the record. Each of these data items is often referred to as a *querylet*. A common approach to the implementation of a *structured* query is a *targeted query*. This is a set of simple queries, where each simple query matches one querylet of the structured query against a different field or set of fields in the record.

One limitation of a set of simple queries is that the component querylets are restricted to matching only within the field or fields specified. In some cases, however, you might have several querylets that correspond to a group of closely related fields whose values are commonly assigned to a wrong field. In such cases, the *cognate query* provides an option for a more refined comparison of these related querylets and fields. Like simple queries, the cognate query computes a match score between 0.0 and 1.0 for the group of related fields.

For instance, tables containing name data often have several name fields: first name, last name, middle name, second surname, and so on. It is common for data to be entered in one name field that belongs in another. Therefore, the values given for the querylets are

subject to the same confusion. In this situation, construct a cognate query in which each name querylet is targeted at the corresponding (or “cognate”) field.

This sounds like a set of targeted simple queries, but there is a crucial difference. Though each querylet in a cognate query is preferentially matched with data in the cognate field, the cognate query also allows the possibility of cross-matching between a querylet and the other participating non-cognate fields. Cross-matching of this kind occurs without penalty (unless you assign a penalty for it). This makes the cognate query the tool of choice for matching across closely related fields.

A query consists of first, middle, and last name querylets – for example, “Alfred”, “E”, “Newman”. You can construct a cognate query matching these querylets against the corresponding First, Middle, and Last Name fields of the table. The search results might include the following four records:

Score	First	Middle	Last
1.00	E	Alfred	Newman
0.92	Alfred	E	Neuman
0.46	John	E	Neumann
0.38	Albert	E	Noyes

Note that the top record “E Alfred Newman” scores a perfect 1.0, although the query’s first name “Alfred” and middle initial “E” are fielded differently in the record. This causes the “E Alfred Newman” record to outrank the “Alfred E Neuman” record, which does not score a perfect match due to the different spelling of the last name.

You can alter this behavior by lowering the non-cognate weight from its default value of 1.0 to some lesser value. (Like field weights, the non-cognate weight is a floating point value between 0.0 and 1.0.) You might then obtain the following results:

Score	First	Middle	Last
0.92	Alfred	E	Neuman

0.88	E	Alfred	Newman
0.46	John	E	Neumann
0.38	Albert	E	Noyes

Note that the “E Alfred Newman” record now scores comparably to the “Alfred E Neuman” record – the “imperfection” of misfielding, like misspelling, being reflected in a slightly lower match score.

After reading the section on [Complex Queries](#), you might think that the cognate query could be simulated by an AND of simple queries, one simple query for each querylet, and each simple query matching all fields in the set. However, there is a critical difference between the cognate and an AND of simple queries.

For example, a person with the first name "johnny" and the last name "johns".

Simple query "**johnny**" against "first,middle,last" AND Simple query "**Johns**" against "first,middle,last" yields:

Score	First	Middle	Last
0.85	johnny	e	
0.82	john	s	henny

The second record scores almost as high as the first even though it is not a good match. The reason is that the string "john" in the first name field of the record is getting matched twice, once for each querylet. This is where the cognate query comes in, while it allows for cross field matching, it ensures that each item is matched only once. The results of the cognate query on the same records are:

Score	First	Middle	Last
0.92	johhny	e	
0.48	john	s	henny

You can see that with the cognate query the second record now has a much lower, more appropriate score.

Cognate Queries with an Empty Field Penalty

All three fields (First, Middle, and Last) are populated in both query and record. In actual name data, it is very often the case that the middle name is not populated. The following is an example where not all fields are populated:

Query			
First	Middle	Last	
John	Quincy		

Query Results			
Score	First	Middle	Last
1.0		Quincy	
0.84	John	Q	Adams
0.84	John Q		Adams
0.83	John	Quick	Adamson
0.81	John		Adams
0.81		John	Adams
0.58			Adamso

Here, you can see that “John Quick Adamson” scored higher than “John Adams” in the match. Most users would not rank these matches in this order. The reason is that when a name, especially a middle name, is missing, users tend to discount it in the match. Users see “John Adams” as a good match because they discount the unmatched “Quincy” because there is no middle name in the record. The standard cognate query penalizes the match for the completely unmatched “Quincy” in the query. The *empty field penalty* option allows the cognate query to discount unmatched data that can be attributed to an empty field either in the query or the record.

When the number of populated fields in the query and the record differs, the empty field penalty factor adjusts the amount of the penalty applied for the unmatched data in the “*extra*” fields. In this example, when the query “John, Quincy, Adams” is matched against “John, Adams”, “,John, Adams” and “John Q,Adams” the query has one *extra* field. In the match against “,Adamso” the query has two extra fields. Extra fields are available only if the *count* of populated fields is different. For example, in a match of “John, Adams” against “,John, Adams” no extra fields are present. Both have two populated fields and one unpopulated field. It does not matter that different fields are populated in the two records.

The determination of whether a field is empty or not is made after character mapping is applied. If a character map eliminates all characters in a field, the field is considered empty. Beware of fields that contain only punctuation or special characters; under the standard character map, they are considered empty.

But which fields are the “*extra*” fields? The cognate query chooses the fields with the poorest match as the extra fields. If there is one extra field, the field with the poorest match is chosen as the extra field. If two extra fields are present, the two poorest matching fields are chosen, and so on. The empty field penalty adjustment is applied to these fields.

An empty field penalty of 1.0 applies the full penalty for unmatched data. This is the default behavior. An empty field penalty of 0.0 completely discounts all unmatched data in the extra fields; it applies no penalty for unmatched data in the extra fields. Factors in between apply a partial penalty.

Look at the following example with three different empty field penalties applied:

Query Results Empty Penalty = 1.0			
Score	First	Middle	Last
1.0		Quincy	
0.84	John	Q	Adams
0.84	John Q		Adams
0.83	John	Quick	Adamson
0.81	John		Adams

0.81	John	Adams
0.58		Adamso

With an empty penalty of 1.0, the results are unchanged. “John, Adams” is still fully penalized for the unmatched “Quincy” in the query.

Query Results Empty Penalty = 0.0

Score	First	Middle	Last
1.0		Quincy	
1.0	John		Adams
1.0		John	Adams
0.99			Adams
0.89			Adamso
0.84	John	Q	Adams
0.83	John	Quick	Adamson

With an empty penalty of 0.0 “John, Adams” and “John, Adams” are now considered perfect matches. There was no penalty for the unmatched “Quincy” when the first and last names matched perfectly. However, the matches still do not look quite right. The “John, Adamso” record is matching “John, Q, Adams”. This is because there was no penalty for the unmatched “John, Quincy”, but the “John, Q, Adams” record is penalized for the unmatched “Quincy” as no “extra” fields are available in this match. This can be compensated for by applying some penalty.

Query Results Empty Penalty = 0.1

Score	First	Middle	Last
-------	-------	--------	------

1.0	Quincy		
0.97	John		Adams
0.97		John	Adams
0.96	John Q		Adams
0.84	John	Q	Adams
0.83	John	Quick	Adamson
0.82			Adamso

These matches are now much closer to the way most people would judge them. But “„Adamso” score is still too high. When the number of extra fields is a very high proportion of the total number of fields, a low extra field penalty usually results in scores most people would judge as too high. (The other factor here is “Q” vs. “Quincy”, people recognize this as an abbreviation, and so it is judged as a good match; the cognate query does not have such contextual knowledge.)

The check for extra fields works whether the extra fields are in the query or the record. The following are scores for an example where the query is incomplete and the record has extra fields:

Query			
First	Middle	Last	
John			
Query Results Empty Penalty = 0.1			
Score	First	Middle	Last
1.0			

1.0		John	Adams
0.99	John	Q	Adams
0.97	John	Quincy	Adams
0.91	John Q		Adams
0.88	John	Quick	Adamson
0.86			Adamso

Notice that “John Q,,Adams” scores much lower than “John,Q,Adams”. In the case of “John Q,,Adams”, no extra fields are available. So the extra field penalty adjustment is not applied. In the case of “John,Q,Adams”, the record has one extra field. The empty field penalty adjustment is applied to the extra field, reducing the penalty for the unmatched “Q”. The presence of an empty field does not mean the empty field penalty adjustment will be applied. It is only the presence of an *extra* field that causes the empty field penalty to be applied.

In cases where a field is frequently unpopulated, It is a good practice to use a low empty field penalty (such as 0.1) so the query can give better overall results. Name matching, where the middle name is frequently empty, is the most common example of this. However, if it is often the case that the majority of fields are empty, a low empty field penalty might result in scores that appear too high. And of course, where matching on all fields is considered mandatory, the empty field penalty adjustment should not be used.

Additional points on Cognate Queries

The following are some additional facts to note about cognate queries:

- Cognate queries can be used in any scenario where you have a set of querylets matching against a corresponding group of closely related fields when you expect frequent misfielding. Name fields are one example; sets of address fields (lines of a street address, apartment/suite number, city, and postal code) are another. In such situations, you always have the option of constructing a set of simple queries (one per querylet) and combining their scores in a complex query. However, in such scenarios, the cognate query's advantage of the cross-matching behavior is lost.
- Cognate queries assume a one-to-one correspondence between querylets and fields. If the structure of the querylets is not identical to the structure of the fields, you

might need to create this correspondence artificially by adapting the structure of the query. You can concatenate two or more querylets or insert one or more "blank" querylets. For instance, if your query has only a first and last name, but the table has three name fields (First, Middle, Last), allow for the possibility of cross-matching with the Middle Name field by supplying an empty string as a middle name querylet.

- Cognate queries support field weights same as simple queries. One difference is that unlike the field weights for simple queries, cognate field weights specify only the relative importance of the fields. They do not penalize perfect matches in less important fields. For a full description of the behavior of field weights in cognate verse simple queries see the section, [Weighting Factors](#).

i Note: Remember in Simple Queries lowering a field weight lowers the final score even for perfect matches. In Cognate Queries, lowering a field weight lowers the relative importance of the field, but perfect matches still get perfect scores.

If you set the cognate field weights for First, Middle, and Last name fields to 0.8, 0.6, and 1.0, respectively, a match in which first and middle name are transposed in the record still scores just as highly no matter how these fields are weighted (unless, you choose to penalize cross-matching by setting the non-cognate weight to a value less than 1.0).

- Cognate queries support querying on specific attributes in a Variable Attributes type field like Simple queries. See [Variable Attributes Queries](#) for details.
- If ALL query strings are empty or if all attributes to be searched are empty, there is no information on which to base a match score. As with the Simple query in this case, the query is assigned the "empty score". The behavior and usage of the empty score for Cognate queries is identical to that for Simple queries. However, note that all of the querylets of the Cognate must be empty for it to be considered empty. As long as one querylet has some data in it, the Cognate is not considered to have an empty query.

Like simple queries, cognate queries can be combined with simple queries or other cognate queries, in more complex query structures. The complex queries are discussed in detail in the subsequent sections.

Variable Attributes Queries

If your table has a container field for Variable Attributes, a Variable Attributes query is a convenient way to match against the values of such attributes. A Variable Attributes query consists of a set of name-value pairs. The name portion of the name-value pair is the name of a Variable Attribute that might or might not exist in any individual record in the table. The value is the text you search for in the attributes with that name. Weighting might be attached to each name-value pair to adjust its relative significance.

For example, the records in the table might represent products in a catalog. Catalog items can have Variable Attributes such as Color, Size, Style, Weight, and so forth. Not all items have every attribute; some might have none. A Variable Attributes query on such a table might consist of name value pairs such as "Color" = "Red", and "Size" = "X-Large".

With Variable Attributes, it is expected that many records will not have one or more of the named Attributes. The question is how to treat such records? By default, the Attribute is simply ignored if either the query value is empty or the record does not have the attribute, or the attribute value in the record is empty. It is as if the Name-Value pair was not included in the query. This behavior is controlled by setting the empty score value for the Variable Attribute query. If the user sets the empty score to a value between 0.0 and 1.0, the empty attributes are no longer ignored. Instead, if a query value is empty or an Attribute is not present or empty, that portion of the Variable Attribute Query is assigned the "empty score". Note that it is only that portion that receives the empty score, not the entire query. For example, assume a query:

```
Color="Red", Size="X-Large"
```

and two records:

```
Color="Red", Size="Large"  
Color="Red"
```

The first record receives a score somewhat less than 1.0 because it does not have a perfect match on both attributes. By default, the second record receives a perfect 1.0 score as it has a perfect match on the only attribute present in the record, the missing attribute is ignored so the total score is based on the remaining attribute. If you set the empty score to 0.0, the first record has the same high score as before, while the second record has a score of 0.5 as only half the attributes were matched, and the non-existent attribute gets a score of 0.0.

Remember that in all the cases, an attribute that is not present in a record is treated as if it existed but had an empty value. Matching makes no distinction between an empty attribute and a missing attribute.

A Variable Attributes query can be combined with other query types such as simple and cognate queries in more complex query structures. A Variable Attributes query is typically used in this way to qualify a search so that the most desired records (the records matching the most desired attributes) score highest.

i Note: Simple and cognate queries can work with Variable Attributes. When specifying a field name to match against, use the field name of the Variable Attributes container, qualifying it with the name of a Variable Attribute using the colon (":") character as a separator, for example, "varattrs:Color".

i Note: If the name of the Variable Attributes container field is used with no qualifier, the match is performed against the concatenated list of all Variable Attribute *values* in the container field.

Example

If a field "Address" of type Variable Attributes is defined using the attributes:

```
line1:123 Main St;line2:Suite101;city:Gothem;county;;state:NY;zip:07123
```

You can query the entire Variable Attribute field or any specific attribute as specified:

- Two Variable Attributes: {"address:line1"= "123 Main St","address:line2"= "Suite 10"}
- Entire attribute field: {"address"= "123 Main St Suite 101 Gothem NY 07123"}
- One Variable Attribute and one normal field: {"address:line1"= "123 Main St","first"= "John"}
- One entire Variable Attribute and one normal field: {"address"= "123 Main St Suite 101 Gothem NY 07123","last"= "Smith"}

Date Comparisons

A date comparison is a specialized version of a simple query, where the query text string is compared against a single field that must be of type date. Using this specialized comparison for dates is generally preferable to an ordinary simple query comparison since the inexact matching employed in the specialized version is tuned for the particular structure exhibited by date values. In addition, the specialized date comparison recognizes a wide variety of date formats, making the comparison more robust and tolerant of variations. Like the simple and cognate comparisons, the date comparison outputs a match score between 0.0 and 1.0, and it can be combined with other kinds of queries in more complex query structures.

For date comparisons, the query must be a valid date, or the query is rejected. However, if the date in the record is empty, as in the other query types, the "empty score" is returned. As in the other query types, the empty score defaults to 0.0 and can be set to any value between 0.0 and 1.0. For date comparisons, there is an additional possibility, the value might be non-empty but an invalid date value. For example, a value of "None" cannot be interpreted as a date, so no comparison can be made. In this case, the "invalid score" is assigned. Its behavior is the same as the "empty score", it defaults to 0.0 and can be set to any preferred value. Separate scores are provided as an "empty" field might have an entirely different connotation than an "invalid" field. Separate values for these cases allow them to be distinguished and treated differently in matching.

Date comparisons must be made against a field of type date (searchable or non-searchable). For both searchable and non-searchable date fields, the query scoring is the same. A difference in behavior between searchable and non-searchable date fields exists only in the initial, prefilter, record selection phase. If the query is run on a searchable date field, the initial record selection phase can use the query date value to aid in record selection. If the field is non-searchable, the query date value cannot be used in the preliminary record selection. The use of searchable date fields has two important consequences:

- The initial record selection is more accurate. With the extra data from the date value in the query, the initial record selection can do a better job, resulting in fewer missed matches.
- The query date counts as "fuzzy match" data. Every query must have some fuzzy match data. A single date query on a non-searchable date field is rejected because it lacks fuzzy match data for the initial record selection. A single date query on a searchable date field is valid as it provides "fuzzy match" data for the initial record selection phase.

The cost of using searchable date fields is increased memory usage to store the indexing data.

Predicate Queries

Another type of query is a predicate query. Predicate queries are used in those cases where exact matching is required, for instance when testing against a small fixed set of code values (for example, gender, race, and marital status) or when integers or floating point numbers must be matched as numeric values. A predicate query is just a predicate expression that returns a floating point value in the range 0.0 - 1.0 or a value that can be converted to a floating point value in the range 0.0 - 1.0 such as a Boolean value.

You can use variable attributes in predicate queries. A rich language is provided for creating predicate expressions, see [Predicates](#) for a complete description of predicate expressions and more details on their use as predicate queries. It can be combined with other types of queries into more complex query structures.

Predicate queries might encounter records with empty or invalid values. As with other queries, the match score in these cases is the empty score or invalid score respectively. Their behavior is the same as with other queries, they default to 0.0 and can be set to any preferred value.

Complex Queries

The following ways of comparing queries or other “given” data (such as constant values in predicate expressions) with attributes of records in a table were described:

- Simple queries compare an unstructured text string against one or more fields.
- Cognate queries compare a set of querylets (query segments) against a corresponding set of closely related fields subject to frequent misfielding.
- Variable Attribute queries match a list of Variable Attribute values.
- Date comparisons are optimized for inexact comparison of date values.
- Predicate queries evaluate a predicate expression over one or more attributes of a record.

Each of these five comparison methods can be thought of as a score generator. Each of them outputs a value between 0.0 and 1.0 indicating a degree of similarity between the

query and one or more record attributes (or, in the case of most predicate queries, whether or not the record satisfies the predicate).

In a complex query, the scores output from two or more score generators is combined into a single overall score using one or more score combiners. The two most commonly used score combiners are the logical query operators AND and OR, which are illustrated with examples in [Scenario 5: Alternate Conditions](#).

The AND operator combines the scores output by a set of score generators into a single composite score by computing the weighted average of those scores. By default, each input score receives a weight of 1.0, and the output score of the AND operator is just the simple arithmetic mean of the input scores. By decreasing some of these weights to positive values less than 1.0, you can decrease the relevancy of some of the input scores relative to the others.

The OR operator combines the scores output by a set of score generators into a single composite score by computing their maximum.

As described in [Introduction](#), the AND and OR logical query operators, because they work as score combiners, “soften” the characteristics of Boolean connectives of conventional query languages. The TIBCO Patterns AND operator outputs a measure of total amount of similarity, based on weighted contributions from different comparisons. Analogously, the TIBCO Patterns OR operator selects the greatest amount of similarity from a set of comparisons that might all be “inexact.”

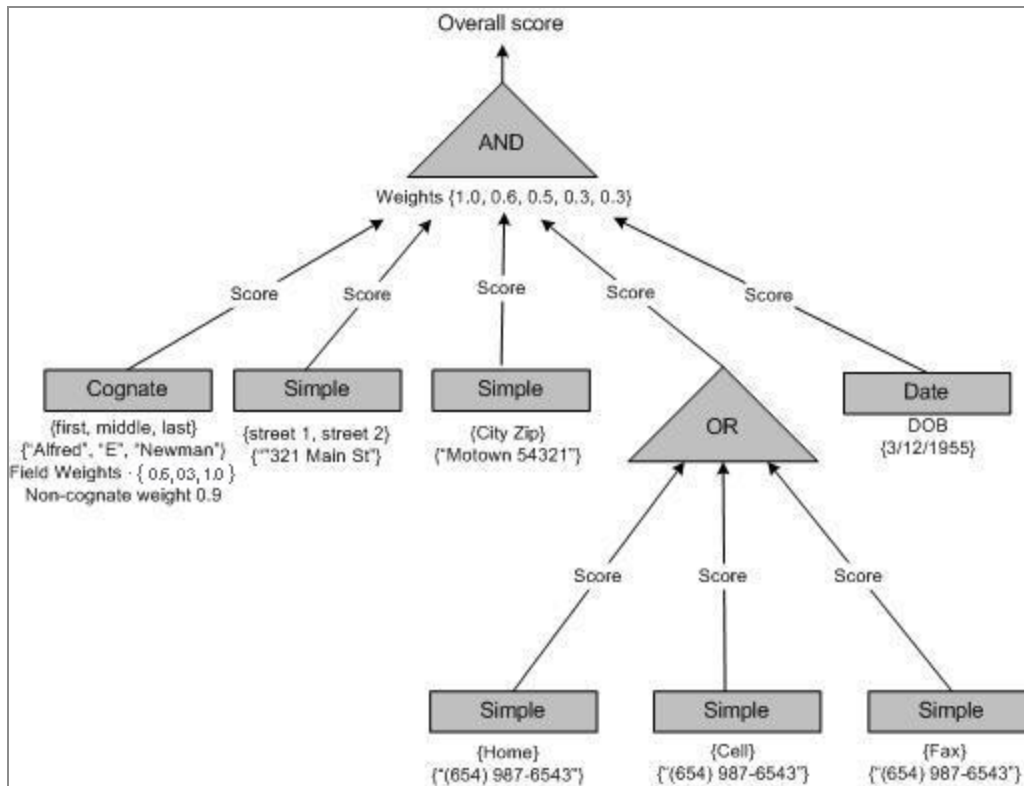
Score “trees”

The output of a score combiner can be one of the inputs to another score combiner. This allows you to combine a set of scores in one way, say with an OR operator, and then combine the resulting score with other scores using a different score combiner, say an AND operator. For example, using the OR operator to select the best score from the comparison of a name with several alternate name or alias fields, and then feed that score to an AND operator along with the output scores from other simple or cognate queries.

A complex query consists of a tree-like set of relationships, in which individual scores output by simple, cognate, Variable Attribute, date, or predicate queries represent the “leaves” of the tree. These are progressively combined using score combiners until an overall score is a final output for the record (this represents the “root” of the tree).

In the following example, an overall query consisting of three name querylets, a street address querylet, a querylet combining city or ZIP code information, a phone number querylet, and a date-of-birth querylet is defined. The table to be searched has fields that

include three name fields, two street address fields, a city field, a ZIP code field, three different phone number fields, and a date-of-birth field.



The figure illustrates one possible design for a complex query that combines seven separate query comparisons.

- There is a cognate query for comparing the three name querylets against the three name fields. Check how the field weights and the non-cognate weight are set.
- Use a simple query to compare the address querylet against the two lines of the street address.
- Use another simple query to compare our city or ZIP querylet against those two fields in the record. (Note that the field weights for either of these two simple queries are not set in the example).
- To handle the phone number, three simple queries to match a single phone number querylet against each of the three phone number fields are used.
- For the date-of-birth the special comparison method for dates is used.
- Finally, the five resulting scores are combined using an AND operator with the specified set of weights to obtain the overall score for the record.

Here are some additional considerations related to complex queries:

- In general, two kinds of structured queries are present that suggest the construction of a complex query:
 - Structured queries over diverse attribute types, especially when these also have diverse relevance for the match. In this case, separate simple queries might be constructed for single field (and cognate queries for groups of closely related fields subject to frequent misfielding), and the resulting scores combined using the AND combiner along with a set of weights to use in computing the weighted average.
 - A querylet that requires comparison against each of several alternative fields where each field represents a possible alternate form of the entire query as opposed to a different portion of the query. For example, a record might contain multiple fields for a phone number: home, work, cell, second line, and a query might contain a single phone number querylet that should be matched against any of the phone number fields in the record. In this case, separate simple or cognate queries might be constructed for each alternative, and then the highest score can be selected using the OR combiner.
- In some complex-query situations, you might have a large number of query inputs (each translating into a querylet), many of which are optional. (A common case might be implementing a search service.) If a record does not have a value for one of these optional fields, or a value that matches very poorly, sometimes you do not want to penalize the record for failing to match that optional value. In cases such as these, you can direct TIBCO Patterns to ignore the querylet's contribution to the score for that record. See the TIBCO® Patterns Programmer's Guide for more details and appropriate cautions relating to this "ignore scores" feature.
- In some complex-query situations, a large number of query inputs are available (each translating into a querylet), of which some are mandatory. (In other words, you do not want any records that do not have very high scores in that field.) In these situations, direct TIBCO Patterns to reject records that have scores less than a threshold value for such a mandatory querylet. See the TIBCO® Patterns Programmer's Guide for more information and appropriate cautions relating to this "reject scores" feature.
- All queries must contain at least one Simple, Cognate, Variable Attributes query, or a Date query against a searchable date field; you cannot have a query that consists of only predicate and date queries against non-searchable date fields, and score combiners.

Match Case Score Combiner

The preceding section on [Complex Queries](#) describes the many different queries that can be combined in complex ways to produce a single overall score that represents the degree of similarity between two records. But sometimes users want to know if the two records represent the same entity. Determining whether two records represent the same entity involves more than just the overall similarity of the two records. It matters a great deal what portions of the record were similar to what degree and what portions were not.

TIBCO Patterns provides two special score combiners to answer this question. The TIBCO Patterns Record Linkage (RLINK) score combiner uses an associated Learn model to determine whether two records represent the same entity. When using TIBCO Patterns Machine Learning Platform, a model can be trained using a set of examples to recognize the difference between matching and non-matching records. See [TIBCO Patterns Machine Learning Platform](#) for a description of this feature. The second alternative is the Match Case Score Combiner feature. This feature uses user defined rules to determine if two records match.

A standard practice in determining if two records match is to formulate a set of “rules” that define what must match to what degree before declaring that two records represent the same entity. The Match Case Score Combiner feature provides a means of directly implementing these match rules that otherwise might need complex post processing or multiple queries to do so.

A standard approach in defining rules is to first divide the information about the entity into a set of *categories*. The determination of categories is a judgment call based on knowledge of the information and the needs of the business. In the Complex Queries example, the following categories are used:

- Name - the first, middle and last name fields form the name category.
- Street Address - the street 1 and street 2 fields form the street address category.
- Location - the City and Zip fields form the location category.
- Phone Number - the Home, Cell, and Fax fields form the phone number category.
- Date of Birth - the DOB field forms the Date of Birth category.

A record that has no match on the name information is probably not a match regardless of how well the other portions of the record match. Match rules typically follow this pattern: they define a minimum set of categories that must match, and other categories then either support or refute the match. There might be multiple rules. These rules are determined based on the nature of the data and the needs of the business. The following is an example

of a set of rules for the categories described previously. These rules are for illustrative purposes and do not represent a recommended set of rules.

- **Rule 1:** Name, Street Address, and Location must match.
- **Rule 2:** Name, Phone Number, and Date of Birth must match.

A match on the name alone is insufficient to determine a match. There might be many people named “John Smith”, but two John Smiths that live at the same address are probably the same person. Similarly two John Smiths that have the same date of birth and a matching phone number are probably the same person, even if the street address and location are different. In this case, “John Smith” probably moved.

The categories that must match are called the *core categories*. The core categories are a set of categories that by themselves are sufficient enough for users to reasonably judge the two records to be the same entity.

But what does “match” mean when stating the core categories must match? If the aforementioned rules were implemented in SQL, or some other exact matching framework, “match” would be a simple equivalence test. But when using inexact matching, there is no equivalence, just degrees of similarity. Therefore, a degree of similarity that represents the boundary between something considered a match and something considered a non-match must be defined. For each category, you must have a querylet that produces a similarity score for that category. You assign a *match threshold* score that defines the boundary between matching and non-matching items for that category. If the category similarity score is lower the threshold, the category does not match. If the category similarity score is at or higher than the threshold, the category matches. Rule one states that name, street address, and location must match, so all three categories must have similarity scores at or higher than their respective match thresholds.

Match Case Score Combiner Rule One: Example 1

Considering the two records in the following table, you might ask if these records are for the same person. With rule 1, the answer would be yes, as there is a near-perfect match on the name, street, and location. But most people would say these are most likely not the same person because the date of birth is completely different. With no date of birth information, most people would likely say these are the same person, but the different date of birth values makes most people think otherwise. The difference in the date of birth values *refutes* the claim made by the match on the core categories.

Match Case Score Combiner Rule One Example 1

Firs t	Middl e	Last	Stree t1	Stree t2	City	ZIP	Hom e	Cell	Fax	DOB
Joh n		Smit h	123 Main St		Trent on	0869 0				1967/12/ 23
Joh n		Smit h	133 Main St		Trent on	0869 0				1993/6/1 4

Match Case Score Combiner Rule One: Example 2

Considering the two records in the following table, you might ask if these records are for the same person. Looking only at the core categories of rule one, these records might not be the same person. Depending on where you have set your match thresholds for the individual core categories, it might be that all three categories are higher than the respective threshold. But because all three are weak matches these two records are likely to be judged a non match by most people. Therefore, in addition to the individual category thresholds, an overall threshold is still needed. For the records to qualify as a match, they must meet both the individual core category match criteria as defined by the rule, and the overall match strength requirement. In TIBCO Patterns, a test for overall match strength is normally supplied by setting a dynamic cutoff. For more information on the dynamic score cutoff, see [Dynamic Score Cutoffs](#).

A typical example is when a perfect match on all of the core categories returns a score higher than the chosen overall threshold. If the match on the core categories is less than perfect, the record might fall below the overall threshold and the record is not a match, even though the individual categories are all higher than their threshold values.

Match Case Score Combiner Rule One Example 2

First	Middl e	Last	Street 1	Street 2	City	ZIP	Hom e	Cell	Fax	DOB
Joh n		Smith	123 Main St		Trento n	0869 0				1967/12/ 23

First	Middle	Last	Street 1	Street 2	City	ZIP	Home	Cell	Fax	DOB
John		Smythe	133 Main Ave		Trenton	08691				1967/12/23

In this example, when looking beyond the core categories for rule one, you see a perfect match on the date of birth category. Instead of the date of birth category refuting the match, the category is supporting it. With this additional support, most people would say these two records represent the same person. So the categories outside the core categories for a rule might either support or refute the claim of a match made by the core categories. These are called the *secondary categories*. Because each secondary category might support or refute, you must define a threshold score that marks the boundary between supporting and refuting.

Some categories, such as date of birth, when they appear as a secondary category, have a very strong impact on the judgment of a match or non-match. Other categories, such as phone number, have a very weak impact on the judgment of a match or non-match. So a supporting and refuting strength must be defined for each secondary category.

Finally when looking at different rules, you might see that a match on the core categories with certain rules might give far more confidence in the match compared to a match on the core categories with other rules. The core categories of a rule have a *match strength*.

To summarize:

- You must define a querylet for each category that returns a similarity score for the category.
- A set of match rules are needed. For each rule you must define:

A core set of categories that must match.

A match threshold for each core category.

An overall match strength for the core categories.

A supporting strength for each secondary category.

A refuting strength for each secondary category.

A match threshold for each secondary category that defines the boundary between supporting and refuting scores for the category.

A Match Case Score Combiner represents **one** match rule. The full set of rules is implemented by creating a match case combiner for each rule. The records match if any one of the rules are satisfied, so an OR score combiner is used to combine the output of the match case combiners.

The following is an implementation of the two match rules used in the *Match Case Score Combiner Rule* examples:

```
// Querylets for each category (implementation not shown.)
NetricsQuery name_cat ;           // Name category querylet.
NetricsQuery street_cat;          // Street category querylet.
NetricsQuery location_cat;        // Location category querylet.
NetricsQuery phone_cat;           // Phone category querylet.
NetricsQuery dob_cat;             // Date of Birth category querylet.
NetricsQuery []cat_qlets = new NetricsQuery[] {
    name_cat, street_cat, location_cat, phone_cat, dob_cat
};
// Match Case querylet for rule 1.
NetricsQuery rule_1 = new NetricsQuery.MatchCase(
    cat_qlets,    // All of our category querylets.
    0.8,          // This is the match strength,
                  // this is a moderately strong match case.
    // This defines the thresholds for both core categories and
    // secondary categories. A negative value indicates it is
    // a core category. The threshold is the absolute value.
    new double [] { -0.70, -0.80, -0.75, 0.85, 0.60 },
    // This serves double duty, for the core categories it is a
    // weighting factor, similar to the weight on an AND combiner.
    // for a secondary category it is the supporting strength.
    // Phone number is a weak supporter, DOB is a very strong
    supporter.
    new double [] { 1.0, 0.80, 0.80, 0.15, 0.40 },
    // This is the refuting strength for a secondary category,
    // entries for core categories are ignored.
    // phone number is a very weak refuter, DOB is a very strong
    refuter.
    new double [] { 0.0, 0.0, 0.0, 0.05, 0.50 }
);
// Match Case querylet for rule 2.
NetricsQuery rule_2 = new NetricsQuery.MatchCase(
    cat_qlets,    // All of our category querylets.
    0.85,          // This is the match strength,
                  // this is a slightly stronger match case.
    // Notice we set the threshold for categories slightly lower
    when
    // using them as a core categories. Remember it can still be
    // rejected if the combination of scores is below our overall
    // cut off score. So a little leeway helps pick up cases that
```

```

        // will be strengthened by strong matches in other categories.
        new double [] { -0.70, 0.85, 0.80, -0.80, -0.50 },
        // street and location are fairly strong supporters.
        new double [] { 1.0, 0.35, 0.35, 0.60, 0.90 },
        // but street and location are very weak refuters.
        new double [] { 0.0, 0.10, 0.10, 0.0 0.0 }
    );
    // The full query
    NetricsQuery full_query =
        NetricsQuery.Or(null, new NetricsQuery[] { rule_1, rule_2 } ) ;

```

This example is only for illustrative purposes. The proper threshold and strength scores to use depends on the nature of your data and business needs. It is a good practice to use trial queries against real data to tune these values. In addition, querylet references are usually used to improve performance. See the next section on [Querylet References](#) to see this example updated to use referenced querylets.

The supporting strength and refuting strength scores are used to define how much the core score is raised or lowered. This represents a percentage of the difference between the raw score from the core categories and a perfect match score of 1.0 or a perfect non-match match score of 0.0.

The following table shows some examples:

Score Examples

Raw Core Score	Secondary Category Score	Secondary Category Threshold Score	Supporting Strength Score	Refuting Strength Score	Output Score
any score	1.0	any score	1.0	N/A	1.0
any score	0.0	any score	N/A	1.0	0.0
0.8	1.0	any score	0.5	N/A	0.9
0.8	0.9	0.8	0.5	N/A	0.85
0.8	1.0	any score	0.3	N/A	0.86

Raw Core Score	Secondary Category Score	Secondary Category Threshold Score	Supporting Strength Score	Refuting Strength Score	Output Score
0.8	1.0	any score	0.1	N/A	0.82
0.5	1.0	any score	0.5	N/A	0.75
0.8	0.0	any score	N/A	0.5	0.4
0.5	0.0	any score	N/A	0.5	0.25

Because the strength scores represent a percentage increase, a supporting or refuting strength of 1.0 always pushes the output score to either 1.0 for a supporting weight or 0.0 for a refuting weight if the secondary category is a perfect match or perfect non-match respectively. In the fourth example, the reward is one half of the reward for a perfect match because the secondary category score of 0.9 is one half of the way between the threshold value of 0.8 and a perfect match of 1.0.

This shows how one secondary category affects the score. Each secondary category supplies a positive (supporting) or negative (refuting) increment to the raw score from the core categories. These are summed to determine the final overall score. A cap of 1.0 and a floor of 0.0 is imposed on the final score.

The important thing to note is that large supporting or refuting weights have a very large effect on the final score. In general, supporting and refuting weights should be small. This is especially true of supporting weights. A supporting weight of 0.5 is a very large supporting weight. Generally, a supporting weight should never be higher than the overall record cut off score. A supporting weight higher than the cutoff score would boost even a 0.0 score from the core categories. It is a good practice to the cutoff. This implies that the associated secondary category by itself would indicate a record match. If a single category is strong enough to indicate a match, it should appear as a core category in its own match case.

The situation for refuting weights is a little different. If there is a category that, when present, by itself indicates a match or non-match (for example, a trusted customer ID value), you might want it to have a very high refuting score in the other match cases. Essentially, this indicates that these cases apply only when the highly trusted category is not available.

Querylet References

In some query situations, it might be necessary to repeat the same querylet multiple times. This most often comes up in record matching situations where multiple cases are available that indicates that a record matches. See the examples of this in the [Match Case Score Combiner](#) section. Because it can be time consuming to evaluate a querylet, unnecessary evaluations should be avoided. Referencing querylets provides a means of doing this when there are repeated querylets.

Points to note about querylet references:

- **References are by name.** To create a reference to a querylet the querylet that will be referenced must have a name.
- **The referenced querylet must exist in the query tree.** The reference is essentially a pointer to another querylet on the tree. If the referenced querylet is not on the query tree, the TIBCO Patterns server returns an error.
- **Any type of querylet can be referenced, except for other querylet references.** References can be to entire sub-trees within the query tree, not just to the score generators.
- **Circular references are not allowed.** The query will be rejected by the TIBCO Patterns server if it contains a querylet reference that references one of its ancestors.
- **There are no order dependencies.** A reference can be to another querylet anywhere on the query tree as long as it does not create a circular reference.

The following is the example from the Match Case Score Combiner example but modified to use querylet references:

```
// Querylets for each category (implementation not shown.)
NetricsQuery name_cat;           // Name category querylet.
NetricsQuery street_cat;         // Street category querylet.
NetricsQuery location_cat;       // Location category querylet.
NetricsQuery phone_cat;          // Phone category querylet.
NetricsQuery dob_cat;            // Date of Birth category querylet.
// We create references on the category querylets. To do that
// they must be named.
name_cat.setName("NameCategory");
street_cat.setName("StreetCategory");
location_cat.setName("LocationCategory");
phone_cat.setName("PhoneCategory");
dob_cat.setName("DateOfBirthCategory");
```

```

NetricsQuery []cat_qlets = new NetricsQuery[] {
    name_cat, street_cat, location_cat, phone_cat, dob_cat
};
// Now create a category array using references.
NetricsQuery []cat_ref_qlets = new NetricsQuery[] {
    name_cat.getReference(),
    street_cat.getReference(),
    location_cat.getReference(),
    phone_cat.getReference(),
    dob_cat.getReference()
};
// Match Case querylet for rule 1.
NetricsQuery rule_1 = new NetricsQuery.MatchCase(
    cat_qlets,    // One of these must have the original
                 // querylets so they can be referenced.
    0.8,          // This is the match strength,
                 // this is a moderately strong match case.
    // This defines the thresholds for both core categories and
    // supporting categories. A negative value indicates it is
    // a core category. The threshold is the absolute value.
    new double [] { -0.70, -0.80, -0.75, 0.85, 0.60 },
    // This serves double duty, for the core categories it is a
    // weighting factor, similar to the weight on an AND combiner.
    // for a supporting category it is the supporting strength.
    // Phone number is a weak supporter, DOB is a very strong
    supporter.
    new double [] { 1.0, 0.80, 0.80, 0.15, 0.40 },
    // This is the refuting strength for a supporting category,
    // entries for core categories are ignored.
    // phone number is a very weak refuter, DOB is a very strong
    refuter.
    new double [] { 0.0, 0.0, 0.0, 0.05, 0.50 }
);
// Match Case querylet for rule 2.
NetricsQuery rule_2 = new NetricsQuery.MatchCase(
cat_ref_qlets, // All following rules should use the references.
    0.85,          // This is the match strength,
                 // this is a slightly stronger match case.
    // Notice we set the threshold for categories slightly lower
    when
    // using them as a core categories. Remember it can still be
    // rejected if the combination of scores is below our overall
    // cut off score. So a little leeway helps pick up cases that
    // will be strengthened by strong matches in other categories.
    new double [] { -0.70, 0.85, 0.80, -0.80, -0.50 },
    // street and location are fairly strong supporters.
    new double [] { 1.0, 0.35, 0.35, 0.60, 0.90 },
    // but street and location are very weak refuters.

```

```

        new double [] { 0.0, 0.10, 0.10, 0.0 0.0 }
    );
    // The full query
    NetricsQuery full_query =
        NetricsQuery.Or(null, new NetricsQuery[] { rule_1, rule_2 }) ;

```

In this example, there are only two rules. When more than two match rules using the same set of querylets, it is a good practice to use querylet references for all but the first rule. As the original querylets are named, and names must be unique, the original querylets must be used only once.

Consider using a reference querylet whenever an identical querylet is repeated two or more times in a query tree. Querylet references have the biggest impact when:

- The repeated querylet is a complex query with several Simple, Cognate, or Variable Attribute querylets.
- The repeated querylet has a large query string for a Simple, Cognate, or Variable Attribute query.
- It is a joined search using single-parent mode.
- The return set is large, or the prefilter output is large.

Weighting Factors

There are several weights or weighting factors that can be applied. All of these weights belong to one of the following types:

- Penalizing Weights: Weights that lower the final score. That is, the score is penalized by the given factor when the weight is applied.
- Structural Weights: Weights that adjust the relative importance of different portions of a query, but do not impose a penalty.

The following example illustrates the difference between these two types of weights.

Run a query against the following record:

First	Last
Bob	Taf

Then, use both a simple query (penalizing weights) and a cognate query (structural weights).

The results for two different queries with combinations of weights is shown in the following table.

Query	Query Type	First Name Weight	Last Name Weight	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
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

Note that in spite of an exact match, a penalty weight, as used by a simple query, lowers the score, but with a structural weight, as used by the cognate query, an exact match always has a 1.0 score.

In the second example, there is a complete mismatch between 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. With the penalty weights of the simple query, lowering the weight of the first name field does not affect the

score, as it is already zero and zero times anything is still zero. But with the structural weights of the cognate query, 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 represents two thirds of the query, so the score is now two thirds of the perfect score of 1.0.

If you reverse the weighting, the perfect match on the last name is penalized by a factor of 0.5 in the simple query 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 represents only one third of the query so the score is reduced to just one third of a perfect match.

Different Types of Weights

With a cognate query, you expect every field to match. The field set represents parts of a whole. By using weights, you adjust the relative importance of the parts, but do not penalize the total score for matching a particular 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 whole in itself. In this case, you might prefer matches in one field over another. Thus, you can penalize the less preferred fields. The same argument is valid for the AND vs. OR score combiners. The AND is part of a whole, so the weights are structural weights. The OR is an alternative, so penalty weights apply. Thesauri and weighted dictionaries are covered in [Thesaurus and Term Weighting](#).

The following table provides the different weights and whether they are structural or penalizing.

Weight	Weight Type
Simple Query Field Weights	Penalizing
Cognate Query Field Weights	Structural
Cognate Query Non-Cognate Weight	Penalizing
Attributes Query Attribute Weights	Structural
AND Querylet Weight	Structural

Weight	Weight Type
OR Querylet Weight	Penalizing
Classic Thesaurus Weight	Penalizing
Weighted Term Semantic Weight	Structural
Combined Thesaurus Weight	Penalizing
Combined Thesaurus Semantic Term Weight	Structural

Scoring Modes

A match score is a measure of textual similarity that always has a value between 0.0 and 1.0, with 1.0 being the best possible score. If the query (or querylets) and the set of fields in the record are completely identical, this qualifies as a perfect match.

For scenarios such as:

- The query matches perfectly to only a part or “phrase” of the field or record.
- The record or field contains a perfect match to only a part or "phrase" of the query.

TIBCO Patterns offers the following selection of scoring modes.

- The normal scoring mode (default) measures the degree to which the content of the query is found in the record. A perfect score means the query is perfectly contained in the record; the presence of extra unmatched information in the record does not lower the score. Use normal scoring for substring or keyword searches, or interactive queries where you type as little text as possible in order to locate the desired records.
- The reverse scoring mode measures the degree to which the content of the record is found in the query. A perfect score means the record is perfectly contained in the query; the presence of extra unmatched information in the query does not lower the score. (The reverse scoring mode, in other words, “reverses” the sense of the normal score.) Use reverse scoring for retrieving lists of standard keywords, locations or names that are embedded within a body of text used as the query.

- The symmetric scoring mode measures the degree to which the query and the record are identical (or “contained in each other”). The score is lowered if either the query contains information not present in the record, or the record contains information not present in the query. Use this score for record-to-record or field-to-field comparisons, when the query represents the entirety of the text expected to be found in the record, and vice versa.
- The minimum scoring mode is the minimum of the normal and reverse scores. In some situations of record-to-record comparison, this might be a better indicator of match quality than the symmetric score. It puts a higher penalty on extra information in either the record or the query than symmetric scoring would.
- The maximum scoring mode is the maximum of the normal and reverse scores. Use this scoring mode in the relatively rare situation where either the query or the record might contain “extraneous” information, and you do not wish the score penalized in either event.

The type of the scoring mode is based on whether the generated score is the same when the value of the query and the value in the record are swapped:

- **Symmetrical type:** the score for the swapped values is identical. These are the symmetric, minimum, and maximum scoring modes. Only scoring modes of the symmetrical type are used in the Machine Learning Platform to train the Learn models. These modes are also used in most cases in deduplication applications that use the Deduplication Framework.
- **Asymmetrical type:** the score for the swapped values is different. These are the normal and reverse scoring modes. They must not be used with Learn models, and should not be used in deduplication applications.

Note that Normal and Reverse scoring is not a simple “contained in” comparison. The score considers factors such as tokenization (splitting text into words) and relative position. For example, a query for "cat" against a record value of "category" does not yield a perfect score even though "cat" is entirely contained in the record value because the query has "cat" as a separate token, the record does not. On the other hand, a record value of "dog and cat fight" would get a perfect score.

The scoring mode is selectable independently for each component of a complex query. In general, symmetric scoring is appropriate for cognate queries, whereas any of the scoring modes might be appropriate for simple queries, depending on the scenario.

For other details concerning the interpretation of match scores, such as score thresholds, score cutoffs, and tie breaking, see [Interpreting and Handling Patterns Output](#).

The Score Records Command

TIBCO Patterns provides a *score records* command. Using this command, you can perform a query against a given set of records instead of a table. This command is similar to a search command with a few key differences:

- Instead of passing the name of a table or set of tables to be searched a set of records is passed in.
- **Joined searches are not supported by the score records command.**
As tables are not involved, the concept of joined tables does not apply.
- **All records passed in are always scored and returned.**
This command is explicitly a request to calculate and return the match score for a set of records, and not a "search" request. Therefore, all records are always returned, even if the record is assigned the "reject" score.
- **Filtering predicates do not apply.**
All records given are always scored and returned and therefore, a filtering predicate is ignored.
- Prefilters are never used, and therefore, all prefilter search options are ignored.
- **Cutoff options do not apply.**
Again, all records are always scored and returned, and therefore, cutoff options are ignored.
- A query that has no query data is not considered an error when used with the score records command. In a search command, a query that has no query data is rejected as an error. With no data, the TIBCO Patterns server can make no reasonable selection among the records in the table. The score records command is not making a selection, it only provides scores, and therefore this restriction does not apply to the score records command.

The score records command ignores options that do not apply, it does not reject them with an error. This allows the same query and search options used with a search command to be used with a score records command. The output of the score records command uses the same format as the output of the search command, including ordering the records from the highest score to the lowest. The special reject score is considered the lowest possible score.

Some of the possible use cases for the score records commands are:

- Scoring record pairs when training a Learn model.

- Applications where a stream of records must be tested for a match to a particular query. Instead of loading the records into a temporary table, they can be tested directly.
- Applications that need a two-phase search.
The first phase is a rough search to see if there are any candidates. The second phase applies a more precise, but more expensive, search to refine or obtain additional score information on the records. The score records command can be used in the second phase to avoid searching the entire table twice.

Thesaurus and Term Weighting

This section covers the use of thesaurus tables, plus a couple of variants of the thesaurus table used in semantic term weighting.

- [Classic Thesaurus Tables](#)
- [Weighted Dictionary](#)
- [Combined Thesaurus Tables](#)
- [Comparing Thesaurus Table Variants](#)
- [Additional Considerations Related to Thesaurus Tables](#)

Classic Thesaurus Tables

A “classic” thesaurus table specifies sets of terms (words or phrases) that the matching algorithm detects and treats as equivalent. This is useful in cases where you match words or phrases with each other despite having dissimilar spellings. There is a classic thesaurus table in operation in [Introduction](#), where the nickname “Peggy” was enabled to match the name “Margaret” with a strong contribution to the match score. Terms in a classic thesaurus table (or the variant tables that are discussed later) can consist of single words or multiple-word phrases, as in the following equivalence classes:

Equivalence class terms	
laptop	notebook
high blood pressure	hypertension

Note the three-word phrase that occurs as the first term in the second row.

The following table represents another set of equivalence classes for color designations:

Equivalence class terms				
yellow	lemon	sunflower		
yellow	canary	cream	ivory	maize
yellow	goldenrod			
green	cyan	aqua	teal	turquoise
blue	cyan	aqua	teal	turquoise

These classes illustrate that the same term might occur in more than one class. For example, the term “yellow” is a member of each of the first three equivalence classes. The presence of a common term does not cause the three classes to merge; they remain distinct. Thus, “goldenrod” is not regarded as equivalent to “lemon” or “canary”, although both are regarded as equivalent to “yellow”. The use of a term common to several classes lets you equate a less-precise term (“yellow”) with several sets of more-precise terms (shades of “yellow” arranged in several groups), without equating more-precise terms with each other.

The fourth and fifth classes illustrate a somewhat different use of terms common to multiple classes. There are many intermediate shades between “green” and “blue”; you might want to equate all of these with both “green” and “blue”, without equating the extremes of “green” and “blue” with each other.

When matching using a thesaurus, a thesaurus weight can be given. This *thesaurus weight* specifies a penalty to be applied whenever equivalences are detected and utilized in matching. Similar to other kinds of weight values, the thesaurus weight is a value between 0.0 and 1.0, with 1.0 signifying that no penalty is applied for matches based on the equivalence class. If you desire matches that depend upon thesaurus equivalence to match strongly, but not to outrank high-quality matches that do not depend on thesaurus equivalence, set the thesaurus weight to a fairly high value of less than 1.0, such as 0.9 or 0.95.

Weighted Dictionary

Consider an example with matching company names where the query consists of the text string “ABC Corporation”. Using a simple query against the Company Name field, and normal scoring, the search returns the following results list:

0.71	Busy Corporation
0.70	XYZ Corporation
0.54	ABC Corp.
0.41	ABC Co.
0.30	ABC
0.25	Busy Corp.

The reason for this unsatisfactory result is that the query contains the term “Corporation”, whose importance in this query is much less than the substantial length of the word suggests. But the matching algorithm, being language-independent, matches “Corporation” as eagerly as it matches “ABC”. Company Name fields contain a small number of such *lightweight terms* that possess slight, if any, significance for a match.

TIBCO Patterns allows you to attach semantic weight values to specific terms like “Corporation” and “Incorporated” by including such terms in a variant of a thesaurus table called a *weighted dictionary*. A weighted dictionary identifies certain terms (words or phrases) as possessing either less importance in a match, or more importance in a match, than the typical word or phrase. The weights assigned to terms in a weighted dictionary are real values greater than or equal to 0.0. If the term weight is less than 1.0, the term is a *lightweight term* whose importance (if detected in a query or record) is to be considered less than that of the typical word or phrase. If the term weight is greater than 1.0, the term is a *heavyweight term* whose importance (if detected in a query or record) is to be considered greater than that of the typical word or phrase. (Consider every term in the query or a record as having a default semantic weight of 1.0.)

Consider the following weighted dictionary to improve the results of the previous search:

Term Weight	Equivalence class terms	
0.1	Company	Co
0.1	Corporation	Corp
0.1	Incorporated	Inc

Note that in addition to the class terms each equivalence class now has a term weight associated with it. The equivalence class allows the equating of a term like “Incorporated” with abbreviations like “Inc.” The weight value is not a thesaurus weight but a semantic term weight. Here the weight value is used to designate the terms in all three classes as lightweight terms with a semantic term weight of 0.1. This means that these terms have roughly one-tenth the importance they would otherwise have in the context of a match.

Perform the “ABC Corporation” query again. Here is the new results list:

1.00	ABC Corp.
0.92	ABC Co.
0.90	ABC
0.19	Busy Corporation
0.19	XYZ Corporation
0.16	Busy Corp.

Results show "ABC Corp" emerging as a top match followed by "ABC Co" and "ABC". The scores for these three records have increased, while the scores for the other three records have dropped. This is the effect of the term weight decreasing the significance of “Corporation” and “Corp”: both the matching of “ABC” and the failure to match “ABC” count for much more than they did. Moreover, with the thesaurus-like equivalence of “Corporation” and “Corp”, the top record receives a perfect score of 1.0.

Note that the record "ABC" has gone from a score of 0.30 to 0.90. This is because even though it does not appear in the record, the "Corporation" string in the query is matched by the "Corporation" term in the weighted dictionary and its effective length is reduced to

one-tenth its original length. Therefore, instead of most of the query being unmatched, most of it is now matched.

i Note: Unlike the classic thesaurus, the term weights are applied even if there is no match between query and record for the term. Term weights are applied whenever the term is found in either the query, the record or both.

Combined Thesaurus Tables

A combined thesaurus table combines the features of a classic thesaurus table and a weighted dictionary: each class has both a thesaurus weight (used for penalizing thesaurus substitutions) and a semantic term weight. With a classic thesaurus, there is a single penalizing thesaurus weight that is applied when equivalent but non-identical terms are matched between query and record. With a weighted dictionary, each equivalence class has a separate term weight that is applied to the terms whenever they occur in the query or a record. This increases or decreases the significance of matching or failing to match these terms.

To continue the Company Names example, in addition to designating terms like “Incorporated” as lightweight terms, you can add equivalences for common names and abbreviations such as “American Broadcasting Company” and “ABC”. You can enter these equivalences in the combined thesaurus table with a term weight of 1.0 (neither lightweight nor heavyweight), plus whatever thesaurus weight (substitution penalty) is required. You might require different substitution penalties for common company nicknames such as “IBM” and “Big Blue”. The combined thesaurus provides this flexibility in a single table.

Comparing Thesaurus Table Variants

Here is a summary of the similarities and differences across the three types of thesaurus tables:

- The sole purpose of a classic thesaurus table is to permit “substitutions”: the matching of lexically dissimilar terms between query and record.

- Unlike the other two forms of thesauri, the classic thesaurus allows only a single penalty that applies to all classes within the thesaurus rather than a penalty or weight per class.
- The primary purpose of a weighted dictionary is to specify the relative importance of certain terms. The term weight is a structural weight, not a penalty. That is, it does not necessarily lower the score of a record but simply adjusts the influence of a particular term on the score, whether the term is matched or unmatched. (Unmatched material in the query influences the value of the normal score, unmatched material in the record influences the reverse score, and unmatched material in either the query or the record influences the symmetric score.)
- A weighted dictionary can define equivalences as a secondary feature (For example, equating abbreviated and non-abbreviated forms of a lightweight term). The weighted dictionary, however, provides no method for applying a penalty to these substitutions.
- Since the primary purpose of the weighted dictionary is the semantic weighting of individual terms rather than substitution of equivalent terms, it is permissible for a weighted dictionary to contain equivalence classes containing only a single term (and the term weight).
- The combined thesaurus table provides the ability to specify any combination of both types of relationships (substitutions with penalties, plus semantic weighting) in a single table structure. If an individual class in a combined thesaurus specifies a semantic term weight of 1.0, then it functions only to specify possible substitutions, as specified in the classic thesaurus table. If an individual class in a combined thesaurus table specifies a semantic term weight of other than 1.0, this weight adjusts the influence of the term whenever it occurs (matched or unmatched). If the same class contains more than one element, substitutions might occur, which are penalized according to the thesaurus weight for the class. Unlike the classic thesaurus, the combined thesaurus allows a different substitution penalty to be specified for each equivalence class.

Additional Considerations Related to Thesaurus Tables

Here are some additional considerations related to thesaurus tables (all variants):

- TIBCO Patterns lets you associate a specific thesaurus table with each simple, cognate, or Variable Attributes query. Thus, advanced queries that combine the results of several simple or cognate queries can involve several different thesaurus tables.
- By default, TIBCO Patterns thesaurus tables incorporate a limited degree of error-tolerance in the detection of thesaurus terms. A simple misspelled term in either the query or the record (or even in both) generally does not interfere with its detection as a term occurring in a thesaurus table. Exact-only detection of thesaurus terms is also provided as an option.
- In either the weighted dictionary or the combined thesaurus table, the semantic term weight for a class might be given the special value -1.0, indicating a stop token. A term identified as a stop token is ignored entirely – it is not matched, nor is the fact of its not matched influence the score in any way. Note that this is different than a term weight of 0.0. With a zero weight, a term is still allowed to participate in matching and thus might influence the score. With a term weight of -1.0, the term is effectively removed from the query and record before matching is performed.
- A particular word or phrase in a query or record might have multiple substitution possibilities. Although the rules for resolving such ambiguities are complex, in general, they are resolved in such a way as to maximize the overall score of the query and record combination.
- Thesaurus data (equivalence classes and associated weight values) can be constructed programmatically at the API level, or read from the files in CSV format. The data can then be loaded into the TIBCO Patterns server as a resident in-memory thesaurus available for all searches.
- Alternatively, a thesaurus might be defined at query time and supplied (along with its content) as one of the parameters of the search. A thesaurus table so defined exists only for the duration of the query, and is therefore known as an *ephemeral thesaurus*. An ephemeral thesaurus is appropriate in cases where possible synonyms or term weights have to be generated dynamically based on the content of the query, and cannot be predefined for all possible queries. Although there is no size limit for an ephemeral thesaurus, it is strongly recommended to limit them to a few classes in size. Creating an ephemeral thesaurus by reading from a CSV file is likewise possible, but not recommended, for obvious performance reasons.

Internationalization and Character Maps

This section covers internationalization, and the character maps that TIBCO Patterns uses to equate certain classes of characters.

- [Internationalization](#)
- [Character Maps](#)
- [Custom Character Maps](#)
- [Additional Considerations Related to Character Maps](#)

Internationalization

The mathematical matching algorithms of TIBCO Patterns are, by their very nature, independent of particular languages. As a practical matter, TIBCO Patterns expects textual data in in-memory tables to be UTF-8 encoded. Once decoded internally, however, the core matching algorithms perform intelligent inexact matching upon this text as strings of abstract symbols, regardless of the particular alphabet or writing system from which the symbols are drawn.

Various writing systems have features that differentiate characters in ways that are irrelevant to the kind of matching you want to do. The most obvious example is the letter case of alphabets such as the Roman alphabet. By default, the matching Roman letters is case-insensitive. Similarly, it is not sensitive to the accented and unaccented versions of a character, differences in punctuation, and so on. To equate different versions of a character into one essential version is one function of the character maps in TIBCO Patterns. Another function is to strip out characters or classes of characters that are irrelevant in matching.

Character Maps

Associated with every text field of an in-memory table is a character map. Each text field can be independently associated with either one of the built-in character maps or a custom character map that you define. A character map defines how every possible character occurring in a field must be mapped before inexact matching is performed. This mapping

of characters can be used to accomplish letter case folding, removal of punctuation, translating the various types of whitespace to a common value, reducing accented versions of a character to the unaccented version, and other special character mappings that your application requires.

The character maps associated with the fields of an in-memory table are established at the time of table creation and cannot be changed.

Each thesaurus created also has a character map associated with it.

If no character map is explicitly defined for a field or thesaurus a standard default character map is used. The default character map performs the following character mappings:

- Letter case folding—according to the rules defined by the Unicode Consortium for folding all alphabetic characters to a common letter case.
- Diacritics folding—according to rules defined by the Unicode Consortium for stripping letters of their diacritic marks and other character “normalizations”.
- Character class mappings—all characters belonging to the “whitespace” class and all characters belonging to the “punctuation” class except for the ampersand (&) are mapped to the blank character (Unicode code point: 0x20). Whitespace is as defined by the Unicode standard. For characters outside the standard ASCII range punctuation is as defined by the Unicode standard, for characters within the ASCII range anything that is not a letter, digit, or white space is considered punctuation.

Besides the default character map, a second predefined character map is available that does not map "punctuation" characters (that is, all punctuation characters remain unchanged).

Custom Character Maps

If the predefined character maps are not satisfactory, you can create your custom character maps. A custom character map is created by defining a set of rules, each of which specifies a mapping a set of one or more characters or for an entire class of characters. Shorthand is provided for invoking Unicode Consortium rules for the folding of cases and diacritics. You can also conveniently reference the “punctuation” and “whitespace” classes as defined in [Character Maps](#). You can also create rules that define mappings explicitly, character by character. Thus, there are four types of mapping rules that are available to define custom character maps:

1. Letter case folding (Unicode rules)
2. Diacritic folding and character normalization (Unicode rules)
3. “Whitespace” and “punctuation” class mapping (as defined in [Character Maps](#))
4. Explicitly defined mappings

These four types of mappings are ordered from lowest to highest precedence. The explicitly defined mappings have the highest precedence. This allows you to override any predefined character mapping rules.

For example, to create a character map that folds all the letters to a common case except for “A” and maps all punctuation to blank except for “&”, add to the standard mappings an explicit mapping of “A” to “A” and “&” to “&”.

For more details about defining the custom character maps, see TIBCO® Patterns Programmer’s Guide.

Additional Considerations Related to Character Maps

Here are a few additional considerations related to character maps:

- A custom character map must be created and assigned a name before it can be used. Custom character maps are stored in the Patterns server as named items analogous to in-memory tables. Once created, however, the custom character maps cannot be deleted or updated – they exist for the life of the TIBCO Patterns server process.
- Thesaurus tables also have character maps. If you do not specify a character map for a thesaurus table, the default character map is used.
- By default, all fields used within a simple or cognate query must have the same character map. It is possible to override this when using the "C" interface and create simple or cognate searches across fields with different character maps. However, this can lead to invalid results and should never be done without consulting your TIBCO representative.
- The character map for a thesaurus should be the same as the character map for the fields to which the thesaurus is applied. Thesaurus matching does not work properly if the thesaurus and field do not use the same character map.

- Although a character map cannot be deleted, the addition of a character map can be rolled back by aborting the transaction that added it.
- A character map that is added by an open transaction cannot be read or used by any transaction other than the transaction that added the character map. The only exception to this is the character map list operation, which shows the character map in the listing. This prevents a character map that could be rolled back from being assigned to a thesaurus or table that is created by another transaction.

Interpreting and Handling Patterns Output

This section provides a basic understanding of how to interpret the results returned by TIBCO Patterns, and how your application can make the most effective use of them.

- [The Meaning of a Match Score](#)
- [The Number of Matches Requested](#)
- [Dynamic Score Cutoffs](#)
- [Tie-breaking Rules](#)
- [Match Visualization](#)

The Meaning of a Match Score

The match score returned by TIBCO Patterns, always a value between 0.0 and 1.0, is a confidence measure based on the overall similarity between a simple or complex query and one or more fields of a given record. The score value is influenced by several factors such as patterns of similar characters, the similarity of token structure, thesaurus equivalences, specially weighted terms, and so on. TIBCO Patterns makes effective use of most of this score range: a record scoring 0.3, or even 0.2, might have a significant likelihood of being a record of interest, depending on the context of your application.

This point reiterates the point that was made in the [Introduction](#). In the world of inexact matching, the answer to the question “Does it match?” is not always a clear “yes” or “no”, but often a “maybe.” Put another way, the value of inexact matching resides precisely in that range of scores that are neither “poor” nor “perfect.”

Of course, the match score cannot and does not take everything into account, like the nature of your application, your business context, or your business rules. This is why TIBCO Patterns always returns a requested number of matches, rather than arbitrarily cutting off the list at a particular score threshold. You are the best judge of how the confidence measure represented by the match score relates to your business and application context.

The Number of Matches Requested

TIBCO Patterns allows you to specify a definite “number of matches requested”. If you don’t specify this number explicitly, a default value of 25 matches is used. For the best performance, this number must be large enough to capture the highest number of results expected to be “of interest,” but no larger. Arbitrarily raising this number to a large value (in the hundreds or the thousands) can have a heavy impact on performance. Most applications do not require more than 50 to 100 matches to be returned. For many applications, a lesser value is more than sufficient.

The Returned Scores

For each record, a set of scores is returned. For most applications, the "Match Score," also called the "Sort Score," is the only score that is needed. This is the overall match score for the record; it is the score that is used to sort the records when selecting the top-scoring records. As stated in the section [Scoring Modes](#), there are several different scoring modes available. One of these is selected as the scoring mode for the top level querylet of the query tree. This becomes the "Match Score" for the query as a whole.

Some applications might need to access other scores as part of a post processing step. By default, the scores for all of the different scoring modes are returned. In addition, if the top level querylet is a score combiner, which is usually the case for most applications, the scores for each of the querylets of this top level combiner are returned. As with the overall scores, the scores for all of the different scoring modes are returned. Applications that have complex business rules based on the match strength of various components of the record can use these scores as input to these rules in post processing the returned records.

In some cases, the query might be a complex query with more than two levels. In this case, the querylet of interest might be several levels down in the query tree. By default, its score is not returned. Consider the example presented in the topic of [Complex Queries](#). Suppose you have a business rule that states that records that are matched on the fax phone number should be treated differently than those that are matched on the home or cell phone number. By default, you do not have access to the match scores for the individual phone numbers as they are two levels down in the query tree. However, the match score for the phone number querylets can be obtained by assigning a name to the querylets. The match score for each named querylet is returned, allowing the application to determine which phone number was matched. Unlike the top levels, for named querylets, only the selected match score (also known as the sort score) is returned. See the individual API

reference documents and the topic "Named Querylets" in the *TIBCO Patterns Programmer's Guide* for details on how to assign names to querylets and retrieve their scores.


Named querylets can provide a more convenient means of retrieving component scores even when they are at the top level. If query structures are built dynamically, the ordering of the different querylets might not be fixed. It can be more convenient to assign a name to the querylets of interest and retrieve the scores by name rather than by position.

Dynamic Score Cutoffs

Some applications present search results directly to an end user while other applications pass results on to another application for automatic processing. Depending on the nature of the application, you can minimize the presentation or processing of results that have a small likelihood of being authentic matches ("false positives"), while avoiding the loss of records of interest ("false negatives").

This means applying some cutoff method for the results list. TIBCO Patterns provides several dynamic cutoff methods that can be configured within your application:

- **Exact-plus-N** — Returns exact matches (with a score of 1.0), plus the highest scoring N inexact matches, where N is a fixed number that you specify.
- **Percent-of-top** — Returns records with scores greater than or equal to a specified percentage of the top score returned (that is, the score of the first record in the results list).
- **Percent-gap** — Returns records until a gap is encountered between consecutive scores that is greater than a specified percentage of the top score returned. This is the best method for dynamically eliminating the most false positives.
- **Absolute** — Returns records with scores greater than a specified score value.

 **Note:** When using the Absolute cutoff method, select the cutoff score based on careful evaluation of runs with the real data.

When a cutoff method is selected, the "number of matches requested" continues to function as a ceiling on the number of matches returned. Dynamic score cutoffs only reduce the size of the results list. For example, regardless of the cutoff method selected, if the number of matches requested is 25, no more than 25 records are returned. Hence, if

you select a cutoff method, for example Exact-plus-N, with more than 25 exact matches to the query in the table, the results list will contain only a portion (25) of the exact matches.

If your application requires a dynamic cutoff method, pursue it in a systematically as it depends on the nature of the data and your particular application's tolerance level of false positives.

Studying the score profiles of a representative set of queries against the actual data is usually the best way to determine an appropriate method for limiting search results. You look for a score region such that scores greater than the region almost always represent authentic matches, and scores less than the region almost always represent non matches. In case there is no such region, that is, if scores of definite authentic matches frequently overlap with scores of definite inauthentic matches, you must tune the query structure or the weighting of the parts of a complex query, to create such a “region of separation” between definite authentic and definite inauthentic matches.



Tip: The preceding section explains the definite authentic and definite inauthentic matches for a reason. There is always an intermediate region of results which you regard as possible authentic matches. The required “region of separation” inevitably includes such “maybe” results. This is precisely the region you will “manage” when you select the cutoff method.

The choice of cutoff method depends on the following factors:

- Whether the region of separation corresponds reliably to an absolute range of score values. If it does, select an absolute score cutoff somewhere in the region. Otherwise, select one of the cutoff methods that is relative to the top scoring item in the list, or the Exact-plus-N method.
- Your policy regarding tolerance of false positives. This depends on the application. With interactive searches, the appearance of some false positives at the end of the results ensures that no true-though-very-inexact matches are missed. On the other hand, in an application that triggers actions, such as the merging of records based on the results of a search, you might want to minimize false positives at the cost of a few more false negatives.

Cutoffs When Using a Learn Model

When using a Learn Model to score records, the scores returned might change each time the model is retrained. This might alter the appropriate cutoff score (the Learn UI

application in TIBCO Patterns has a tool for selecting an appropriate absolute cutoff score for a model). To avoid the need to alter application parameters to update the cutoff score each time a Learn model is retrained, TIBCO Patterns allows a cutoff score to be embedded in the Learn Model itself. As an option, a query can use this embedded cutoff score as an absolute cutoff score for the query. This ensures the cutoff used for the model is always the correct cutoff for that model.

Use of Business Criteria to Re-sort the Results List

Consider an intelligent selection of a cutoff method as a prerequisite in any setting where an application re-sorts the final results list according to criteria other than the match score, that is, following business rules or other requirements. In such cases, a stricter approach defining the dynamic cutoff method and its parameters might be required, so that a small number of high quality matches and fewer false positives are returned. The reason is that a re-sorting of the results list according to business criteria can easily position less-similar matches ahead of more-similar or even exact matches. It might confuse an end user, especially if clear non-matches are positioned ahead of clear matches. Re-sorting proves to be a pitfall for a requesting application also if the application assumes that high-quality matches necessarily precede lower-quality matches.

Tie-breaking Rules

Besides dynamic cutoff methods, TIBCO Patterns also provides optional tie-breaking rules that allow you to control the ordering of the results list if multiple records receive identical scores.

You specify tie-breaking rules in an ordered list. When a group of records receives the same score, the first rule in the list is applied. If the application of this rule also results in a tie between two or more records, the second rule in the list is applied, and so on. If two or more records remain tied when all tie-breaking rules are exhausted, the order of these records is undefined.

The available tie-breaking rules include the following:

- **Scoring Mode** – Ties are broken based on a particular type of scoring mode, which should be other than the primary scoring mode(s) specified for the search. For instance, when the primary scoring mode is normal scores, a common practice is to break normal-score ties using the symmetric scores for the records involved. The effect is that, when the query partially matches a set of records with identical

normal scores, records with less unmatched text are ordered ahead of records with more unmatched text.

- **Field value** – Ties are broken based on comparing the value of a specified field in the tied records. If a text field is specified for tie-breaking, the resultant ordering is “alphabetical” sorted from lowest to highest based on Unicode code point values, local lexical ordering is not used. If the specified field is numeric, the resultant ordering is by decreasing values (that is, greatest first).
- **Match alignment** – Ties are broken in favor of records where most of the matching text occurs closer to the beginning of the field. (This rule is meaningful only for the simple and cognate query types, not for date comparisons or predicate queries.)
- **Record key** - Ties are broken according to the ascending sort order of the record key. As all keys must be unique, this is guaranteed to break a tie. Use this tie-breaking rule as the final rule in a list of tie-breaking rules whenever you want the search results to be presented in a deterministic order.

If you do not specify a set of tie-breaking rules, a default set is employed consisting of: tie-breaking by symmetric score, followed by tie-breaking by record key.

Match Visualization

Many search technologies allow end users to see which portions of a returned record matched the query. The matching algorithm used in TIBCO Patterns permits users to see, not only which portions of returned records matched the query, but the degree to which each matched portion of a record contributed to the overall measure of similarity between the record and the query. For every character in a returned record, TIBCO Patterns returns an “intensity” value that can be used to differentially highlight portions of a record according to their match “intensity”. This differential highlighting is called *match visualization*. Typically, different font colors or type sizes are used to emphasize characters according to their associated intensity values.

If requested the TIBCO Patterns server returns four values, known as V, P, D, and N, for each character in the searchable text fields of every record returned.

The V value provides all the visualization information needed for the vast majority of applications for which match visualization is required. It is a “composite” or “summary” intensity value reflecting the various dimensions of the matching algorithms. The P, D, and N values represent these dimensions as separate values. The P value indicates the length of the segment of matching text to which this character position belongs (zero for unmatched

characters). The D value measures how far away this matching segment is from the corresponding segment in the query, given the optimal alignment of the query over the searchable record text computed by the matching algorithm. The N value for a matched character is set to one if that matched character was deemed “noise” by the matching algorithm, and hence contributed much less to the overall similarity score.

The most typical color visualization scheme associates six shades of a particular hue with different ranges of the V value, with brighter shades corresponding to higher V values (greater match intensities). The V values are split evenly into 7 score ranges. The lowest range is typically not highlighted (these matching characters are generally “noise”), and the successively higher ranges are assigned the six shades of increasing brightness.

To make this kind of color visualization easier, TIBCO Patterns provides an interface for specifying a base color (usually selected to suit the color palette of a web page), and a matching color (representing the strongest match intensity). TIBCO Patterns then computes the color gradient and returns result lists already formatted with the appropriate HTML tags for color visualization.

For more details and other visualization options, see the TIBCO® Patterns Programmer’s Guide.

Predicates

This section provides an understanding of the uses of predicate expressions and how to make the most effective use of filtering predicates.

- [Filtering with Predicates](#)
- [Constructing Predicate Expressions](#)
- [Error Conditions with Predicates](#)
- [Predicates and Performance](#)
- [Predicate Queries](#)

Filtering with Predicates

A *predicate* (or predicate expression) is a logical expression similar to an SQL WHERE clause. By far the most common use of predicates in TIBCO Patterns is to filter the results of the search. Just as a WHERE clause can be used to limit the output of a SELECT command in SQL, a predicate expression can be used to limit the output of a search in TIBCO Patterns. Predicate expressions used in this way are called *filtering predicates*.

An example of a filtering predicate is given in [Introduction](#), where the filtered results are based on the content of a STATE field, using the predicate expression:

\$“STATE” = “CA”

Filtering predicates can only modify a search, not constitute it. That is, you cannot execute a search consisting only of a filtering predicate like \$“STATE” = “CA” (to find all records in a table with “CA” as the value of the STATE field). Filtering predicates do not stand alone; they merely modify a search involving the usual simple or cognate query types.

Filtering predicates must be constructed so that the result of their evaluation is either true or false. (This is naturally accomplished through the use of the Boolean operators provided for predicate expressions.) Only records for which the predicate expression evaluates to true are returned by the search.

Filtering predicates are used when there is some specific exact match condition that can be used to eliminate a record from consideration. For example, if you are searching for vehicle

registrations and looking for a commercial vehicle, you can use a filtering predicate that selects only commercial registrations. If, however, the criteria cannot be guaranteed to eliminate a record from consideration, you cannot use a filtering predicate. In the example from the [Introduction](#), if there was a significant probability that the STATE field could have an incorrect value, then filtering based on the STATE field would not be valid as you could have some matching records with incorrect STATE values that would be eliminated by the filtering predicate.

Constructing Predicate Expressions

The simple predicate in the example [Filtering with Predicates](#), `$"STATE" = "CA"`, exhibits all of the main features of predicate expressions:

- Predicate expressions generally contain references to one or more fields of the table. In the example, `$"STATE"` is a reference to the content of the STATE field of the record being evaluated.
- Predicate expressions usually also contain constant “data” of their own. In the example, `"CA"` is a constant value private to the predicate expression.
- Predicate expressions usually use operators to test the truth of a condition involving given data and the content of one or more fields. In the example, the `"="` operator tests for (case-sensitive) string equality between the constant value `"CA"` and the value of the STATE field.

Constants, Field References, and Operators

Field references in predicate expressions can refer to fields by name or by numeric column position. Field names in predicate expressions are the same as used elsewhere in the TIBCO Patterns API. They can be qualified with table names (for joined records) and attribute names. In a joined search, column position refers to the position in the joined record, not the table. Constant values used in predicate expressions include boolean, integer, double-precision float, string, byte block, and byte-block array. The following table illustrates how field references and constants are represented in predicate strings:

Examples	Description
?TRUE?, ?FALSE?	Boolean constants (letter-case insensitive)
123, 0777, 0x8FFF	Integer constants (decimal, octal, hexadecimal)
123.45, 0.17e-10	Floating point values
"string value"	String constants
: "byte block"	Byte block (note the preceding colon)
#2	Table field specified by the column position (zero-based)
\$"first name"	Table field specified by the field name
\$"varattrs:Color"	Variable attribute specified by name. The text in quotes contains a colon separating the field name for the container field from the Variable Attribute name.
[:"block 1", : "block 2"], [:]	Block array (note the special representation for empty array)

Integer and floating point values must be within the defined range for "C" integers and doubles, respectively. Both string constants and byte block constants support the basic XML/HTML entity encoding scheme for representing the double quote character itself (that is, "). 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.)

i Note: Do not insert an encoded NULL character into a string constant, since the string is converted to a standard "C" NULL terminated string. Inserting a NULL therefore effectively terminates the string value at that position. Byte blocks on the other hand might contain NULL characters as they are not NULL terminated strings but use an explicit length value, which is computed automatically and need not be provided by the user.

Operators in predicate expressions are either unary (like absolute value, logical inverse, or type conversion operators) or binary (like string equality or addition operators). In general, operators accept as operands constant values, field references, or other (nested) predicate expressions. Not all combinations of operators and constant or field-value types make sense. In such cases, the predicate expression is rejected by the search with an error.

In addition, there is a special type of unary operator called a predicate function. These operators accept a single value of the special type "argument list". An argument list is a special type that is constructed using special operators, but it is not necessary to know the details of an argument list construction unless you are using the "C" interface. The Java and .NET API's provide method calls to create each of the defined predicate functions. In predicate string expressions the argument lists are contained within braces.

```
geo_distance { 40.0, 75.0, $"latitude", $"longitude", "miles" }
```

calls the geo-distance function to compute the distance in miles between latitude 40.0, longitude 75.0, and the location defined by the latitude and longitude fields of the record.

TIBCO Patterns provides a rich set of operators with which to construct predicate expressions. Here is the complete list of unary operators (note that some operators have one or more synonyms):

Operator	Description
int	Type conversion to integer
dbl, double, float	Type conversion to double
date	Type conversion to date
date_time, datet	Type conversion to date-time
eudate, dateeu	Type conversion to date (expects European day/month/year format)
eudate_time,	Type conversion to date-time (expects European

Operator	Description
dateeu_time, eudatet, dateeut	day/month/year format)
blk, block	Type conversion to byte block
-	Unary minus
+	Unary plus (does nothing)
not	Boolean logical inverse
split, tokenize	Split string or block into words
abs	Absolute value of integer or double
geo_distance, geodistance, geod	Predicate function to compute the distance between two points on the Earth's surface.
if	Predicate function that implements a conditional expression.
to_score, toscore	Predicate function to normalize a floating point value into a 0.0 - 1.0 score range.

Here is the complete list of binary operators (note that several operators have synonyms):

Operator	Description
+	Addition of integers and doubles; concatenation of strings and byte blocks
-	Subtraction of integers and doubles
*	Multiplication of integers and doubles

Operator	Description
/	Division of integers and doubles
**	Exponentiation of integers and doubles
and	Logical AND of Booleans
or	Logical OR of Booleans
=, ==	Equality of integers, doubles, strings, blocks, dates, date-times
~, ~=	Letter case insensitive equality of strings, blocks
<	Comparison of integers, doubles, strings, blocks, dates, date-times
~<	Letter case insensitive comparison of strings, blocks
<=	Comparison of integers, doubles, strings, blocks, dates, date-times
~<=	Letter case insensitive comparison of strings, blocks
>	Comparison of integers, doubles, strings, blocks, dates, date-times
~>	Letter case insensitive comparison of strings, blocks
>=	Comparison of integers, doubles, strings, blocks, dates, date-times
~>=	Letter case insensitive comparison of strings, blocks
i_in	Letter case insensitive substring match of strings, blocks.
in	Substring match of strings, blocks
superset	Set comparison of block arrays
subset	Set comparison of block arrays
split, tokenize	Split string or block into words using a specified separator

i Note: All lexical comparisons are based on Unicode code points and DON'T take into consideration locale specific sorting conventions for characters.

Operator precedence

All unary operators have higher precedence than binary operators. That is,

```
block abs $"count1" - $"count2"
```

is equivalent to

```
( block ( abs $"count1" ) ) - $"count2"
```

which is probably not what was intended. Parentheses can be used to alter the default precedence relations:

```
block abs ( $"count1" - $"count2" )
```

Since predicate functions are unary operators, they bind to their argument list more tightly than any binary operator. For example,

```
geod { 40.0, 75.0, $"lat", $"long", "kilometers" } ** 2
```

is equivalent to:

```
( geod { 40.0, 75.0, $"lat", $"long", "kilometers" } ) ** 2
```

and not equivalent to:

```
geod ( { 40.0, 75.0, $"lat", $"long", "kilometers" } ** 2 )
```

Binary operators are left associative, for example,

$1 + 2 + 3$

is implemented as

$(1 + 2) + 3$

The binary operators listed by precedence from highest to lowest are:

1. ******

2. *
3. /
4. +
5. -
6. tokenize, split
7. =, ~=, ==, ===, <, ~<, <=, ~<=, >, ~>, >=, ~>=, in, i_in, superset, subset
8. and
9. or

This listing reflects “standard” precedence relations, but with the following caveats:

- + (addition) has a slightly higher precedence than does - (subtraction), thus $a - b + c$ is equivalent to $a - (b + c)$, not to $(a - b) + c$
- * (multiplication) similarly has slightly higher precedence than does / (division)
- Note that all comparison operators have the same precedence.

More Examples of Predicate Expressions

Include in the search results only records of males born on or after January 1st, 1980:

```
( $"sex" ~= "m" OR $"sex" ~= "male" ) AND $"birth date" >= date
"1/1/1980"
```

i Note: There is no explicit Date constant, so this is how a constant date value is expressed. Note the use of case-insensitive string comparison, and the type conversion of string to date.

Include in the search results only records of the category "tool" at an average price of less than ten dollars:

```
 $"category" ~= "tool" AND ( $"max price" + $"min price" ) / 2.0 < 10.0
```

Include in the search results only those persons whose previous and current weight differs by less than five pounds:

```
abs ( $"cur weight" - $"prev weight" ) < 5.0
```

Include in the search results only the records within 25 miles of a given point.

```
geod { 40.0, 75.0, $"lat", $"long", "miles" } <= 25.0
```

Error Conditions with Predicates

To be used as a filtering predicate, a predicate expression must evaluate to a Boolean value (true or false). TIBCO Patterns responds with an error if a filtering predicate expression is unable to produce a Boolean value. For example, the predicate expression:

```
DATE $"Creation Date"
```

is not valid as a filtering predicate, since it never produces a Boolean result.

On the other hand, even well-formed Boolean predicates might fail to evaluate true or false for some records. Consider the following predicate expression where Clicks is an integer field:

```
$"Clicks" > 50
```

Since it is a Boolean expression, it is accepted as a filtering predicate. But the success of the evaluation depends on the integer value in the field Clicks. If a record is loaded with the value "123" in the Clicks field, the predicate evaluates to true. If a record was loaded with the value "21", it evaluates to false.

The predicate evaluation fails if:

- The value stored in the table is the special "invalid" value. For example, if the value passed in for the "Clicks" field of the record was "orange", this is mapped to the special invalid value and the predicate evaluation fails on this record.
- If the value stored in the table is the special "empty" value. For example, in the example, `$"Clicks" > 50`, if the "Clicks" field for the record was empty.

By default, if a filtering 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 that evaluate to true, and excludes records that fail to evaluate as well as the records that evaluate to false.

In some cases, however, it might be required to retain records in the search that have “empty” or “invalid” values in the field referenced by a filtering predicate. Using search options, you can control the behavior of filtering predicates in these cases.

The rules for dealing with "empty" and "invalid" values are:

- By default, if a referenced field contains an empty or invalid value, the record is rejected.
- The user can specify that empty values should not be rejected, instead the predicate expression is evaluated with the empty value using the rule that "{empty}" < "{any valid non-empty value}".
- The user can specify that invalid values should not be rejected, instead the expression must be treated as if it returned true.

See TIBCO® Patterns Programmer's Guide for further details.

Predicates and Performance

Filtering predicates are very useful as a complement to inexact matching. However, certain performance issues might arise with filtering predicates that can seem counterintuitive. Because of the interaction between the inexact and exact selection criteria, a query with a highly selective predicate can take a lot longer to process than one with no predicate or with a predicate that filters out only a small percentage of records.

Fortunately, predicate partition indexes provide a solution for the performance issues connected with very selective predicates. When used correctly, partition indexes can greatly improve query throughput in such cases. Predicate indexes work only with the GIP prefilter. See [Prefilters and Scaling](#) for a description of prefilters.

Partition Indexes

A *partition index* is a special data structure associated with a field of an in-memory table that speeds processing when using filtering predicates that make comparisons with the value of that field. Though a given field can have multiple partition indexes, a partition index is always associated with exactly one field, which can be of any field type. Partition indexes are defined for a table when the table is created, and can neither be added nor removed subsequently. Any number of indexes can be defined for a table.

A partition index is of one of two types: primary or secondary. 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. A secondary index provides much less improvement than a primary index, but at low memory cost.

Both types of indexes work by partitioning records into groups based on the value of the field. This allows the search to be constrained from the start to those groups (partitions) for which the predicate expression might evaluate to true, skipping those partitions that cannot evaluate to true. You define a partition index (of either type) by defining these groups, essentially by specifying an upper limit value for each group.

For example, to index a Date of Birth field you can define December 31st for each year from 1900 to 2100 as the partitions, separating all the dates of birth into separate years (dates in the future are set to avoid redefining the table each year). So the partition values would be: "12/31/1900", "12/31/1901", "12/31/1902", ... "12/31/2099", "12/31/2100". If your filtering predicate was:

```
DATE "6/7/1985" <= $"dob" AND $"dob" <= DATE "6/7/1987"
```

then your partition index would allow you to skip all records except those in the partitions for "12/31/1985", "12/31/1986" and "12/31/1987". 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. Creating an index might not be required if the predicate expression filters out only a few percent of records.

A few additional points related to partition indexes are:

- A larger number of partitions allows the index to be more selective and thus gain greater performance improvements. However, there is a limit of 1021 partitions. In general, it is best to use more partitions to improve performance. However, increasing the number of partitions increases memory usage, especially with a primary index.
- Having the first primary index on a table involves a moderate amount of overhead. Each additional primary index on a table after the first incurs a very large memory overhead penalty. (Although a child table can have one or more primary indexes, the first primary partitioned index incurs a very large memory overhead penalty.) If you need multiple partition indexes for a table, the most selective or most used field should be made the primary index, the others should be secondary. A table should have only one primary index unless there is a very compelling reason to have additional primary indexes, and you are willing to pay a very high price in memory usage. A parent table can have at the most one primary index.

- The IN, SUPERSET, SUBSET, and predicate function operators cannot use partitioned indexes to speed their evaluation.
- Only predicate expressions comparing native (unconverted) values of the field can use the index. For example, if a birth date value is stored in the table as a text field rather than a date field, a predicate expression of the form:
`DATE $"birthday" > DATE "12/7/1986"`
 cannot use the index as the field value is converted to DATE before being compared. (If dates are to be used in predicate comparisons, store them as fields of the date field type.)
- An OR operator in which any sub-expression cannot use a predicate index (for example, uses the IN operator) also cannot use the partition index.
- Certain complex expressions, especially those that combine the NOT operator with the OR operator, might not be able to use a partition index. If your application requires a complex filtering predicate, consult TIBCO Support to select the indexes and predicate expressions best suited to your needs.
- A joined search cannot use a primary index on a child table. However a non-joined search can use a primary index on a child table.

Predicate Queries

There is a second use of predicate expressions besides filtering predicates, which was briefly explained in [Designing Queries for TIBCO Patterns](#). Sometimes, you might want to use a predicate expression as a query type alongside simple, cognate, and date comparisons within a complex query structure. Predicates used in this way are called predicate queries.

In general, predicate queries are useful when one or more of the following conditions are true:

- You do not need for inexact matching on a given field.
- You do not need for the logic operations available with predicate expressions.
- You must perform comparisons on numeric fields based on numeric rather than text values.
- You need to perform date range tests on a date field.

Not all fields of tables require or benefit from inexact matching. Coded fields such as State (of the U.S.), Gender, Marital Status, or Product Type typically have content limited to a set of fixed values. When these values are relevant in a search, they are often good candidates for predicate queries. Using exact-matching predicates on such fields also saves memory and improves performance.

Comparison of numeric and date values often involves special logic different from fuzzy string matching, for instance, matching based on numeric or date range. The integer, float, and date field types of Patterns in-memory tables are provided primarily for predicate support – they are more efficient in predicate expressions and use less memory than the equivalent values as text strings.

In predicate queries, the predicate expression need not be constructed in such a way that it evaluates either true or false; it is required only that it evaluate to a value that can be converted to a double value in the usual score interval of 0.0 to 1.0 or the special reject score of -1.0. This includes Boolean values, which map to 1.0 or 0.0 for true and false respectively, and the integers 0 and 1. This “predicate score” can then be combined with other scores in the complex query. For example, you can use a predicate expression to compute a score based on the geographical distance between two locations, normalized onto the interval [0.0, 1.0].

The following expression is a predicate query where records more than 25 miles away are assigned a zero score:

```
toscore{ geod{ 40.0, 75.0, $"lat", $"long", "miles" }, 25.0, 0.0 }
```

The following is an example that shows how the conditional function can be used to assign a score to coded field values with eye color:

```
if { $"eyes" ~= "blue", 0.1, $"eyes" ~= "hazel", 0.5, $"eyes" ~=  
"brown", 1.0, 0.1 }
```

Another use for the conditional function is to create tiered scores:

```
if { $"age" >= 50, 0.25, $"age" >= 35, 0.5, $"age" >= 25, 1.0, $"age" >= 21, 0.75, 0.0 }
```

Here, a score is assigned based on an age bracket, with anyone under 21 getting a score of 0.0.

Rules to Handle Predicate Errors

Use the following rules to handle predicate query errors that can occur due to invalid or empty data values:

- any expression that encounters an "empty" field value, is assigned the empty score.
- any expression that encounters an "invalid" field value, is assigned the invalid score.
- By default, the empty and invalid scores are 0.0, except for sub-queries of an RLINK query or Match Case query, where they default to -1.0.
- You can explicitly reset these scores to any valid score value or the special -1.0 score, to reject the record.
- An empty or invalid constant in a predicate query will always cause the query to be rejected with an error.



Note: This differs from [Filtering with Predicates](#) where it is possible to have predicate expression evaluation continue with "empty" values.

Transactions

This section covers transactions in detail. It covers key differences between transactions in TIBCO Patterns and transactions in a typical DBMS.

- [Transactions](#)
- [TIBCO Patterns Transactions versus RDBMS Transactions](#)
- [Explicit versus Implicit Transactions](#)
- [Marking versus Locking](#)
- [Dirty Reads and Object Status](#)
- [Transaction Size, Memory, and Performance](#)

Transactions

TIBCO Patterns supports a concept of a transaction. A *transaction* is a group of operations that modify one or more data objects on the server that can be committed or rolled back as a unit. *Data Objects* include: tables, thesauri, Learn models, and Character Maps. A checkpoint of a table is also considered a data object and checkpoint and restore operations can be associated with a transaction. The checkpoint or restore operation is committed or rolled back when the transaction is committed or aborted. Records within a table are also considered data objects that can be tracked on an individual basis.

The use of transactions is optional. This section is irrelevant if you do not need transactions or plan to use joined tables or a clustered TIBCO Patterns server environment. Users who plan to use joined tables or the TIBCO Patterns [Clustering](#) feature should review this section even if not planning on using transactions. Transactions are used in implementing joined tables and clustering, so an understanding of transactions is important when using these features.

A transaction in TIBCO Patterns is similar to a transaction in an RDBMS, but the intent and usage are more limited. The intent of transactions within TIBCO Patterns is limited to providing a means of backing out a set of separate operations as a unit. It is not intended to provide the full range of locking, concurrency control, and data consistency options that are normally provided by an RDBMS.

The primary motivation for the introduction of transactions is the introduction of joined tables. In many cases, a parent record and its child records must be updated as a unit. If updating any of these records fail, any associated record that has already been added should be backed out. The parent and child records are kept in different tables, thus it requires multiple operations to perform the complete update. To implement the backing out of the already performed updates without transactions is tedious. The application needs to remember which records were updated and their previous state. Using a transaction to associate these operations, users can back out all of the changes with a single transaction abort command. Transactions are useful wherever a set of updates needs to be performed in its entirety or not at all.

TIBCO Patterns Transactions versus RDBMS Transactions

Although transactions in TIBCO Patterns are similar to transactions in an RDBMS, the following are key differences:

- **All reads are dirty reads:** with most RDBMS, users can specify an isolation level when working with records and tables. That is, whether they want to see only the committed state or the “dirty” version of an object. TIBCO Patterns always shows the latest version of an object, even if that version has not yet been committed.
- **Locks are not held between operations:** instead of leaving items locked between operations within a transaction, TIBCO Patterns simply marks them as belonging to the transaction that updated them. If another operation, not associated with the same transaction, attempts to update the same item, instead of blocking that item, an error is returned.

These differences have many implications that are described in the following sections.

Explicit versus Implicit Transactions

All operations that modify a data object are associated with a transaction. However, the user is not required to use transactions and can issue a command that modifies a data object without using a transaction. In this case, the TIBCO Patterns server creates an *implicit transaction* that exists for the life of the command, and is automatically committed

or aborted when the command completes. The user is unaware of this and commands behave as expected.

The user can create an *explicit transaction* and associate commands with this transaction. In this case, the changes are not committed or aborted when the command completes. Even if the command terminates with an error, the completed changes are not rolled back. However, any partial updates are rolled back to a point that ensures that the objects involved are in a usable state. For example, if a command is issued to add a large set of records, and the `do max work` flag is not set, normally an error on one record causes all records added to be rolled back. However, under an explicit transaction, the records that were added are left in the table, only any partial updates of the record that had the error are rolled back. So when associated with an explicit transaction, a command always leaves all objects in a usable state, but the rolling back or committing of any completed operations is not done until the transaction is committed or aborted.

After creating an explicit transaction and performing all of the desired operations for that transaction, the user should either commit or abort the transaction. However, if any of the commands associated with the transaction failed, an attempt to commit the transaction is rejected. The user can override this restriction and forcibly commit the transaction, but it is strongly discouraged. The state of the data after an error is indeterminate. Although the data might be *usable* from the standpoint that it does not crash the TIBCO Patterns server, it is probably not valid from the user's point of view. The user should always abort transactions where commands failed with an error.

i Note: For transactions a command is considered to have failed if, on executing in an implicit transaction instead of an explicit transaction, it rolls back all the changes. Some commands might return an error code but still commit their changes. An example is, a partial checkpoint or partial restore error code from a checkpoint or restore command. For the purposes of transaction processing, these commands are considered to have succeeded even though they returned an error message.

Marking versus Locking

Most RDBMS control access to shared data objects by using *locks*. One operation at a time can hold the lock, all other operations requesting access are blocked, and sit idle until the lock is released. At that point, one of the waiting operations acquires the lock and the

others continue to wait. Most RDBMS hold locks until a transaction is completed. A *mark* never causes an operation to block, instead, if the mark is already held, the requesting operation receives an immediate error.

The TIBCO Patterns server uses a combination of locks and marks to control access to data objects. Within a specific command, all data objects are locked by blocking locks for the duration of the command. This is regardless of whether it is associated with an explicit transaction or an implicit transaction. Marks are also placed on the data objects. The locks are released when the command completes and the marks are released when the transaction is committed or aborted. As implicit transactions are committed or aborted automatically before the locks are released, the marks are cleared before the locks are released and thus are not normally seen by the user. So the discussion of marks, with just a few exceptions, pertains only to explicit transactions and how data objects are treated between commands within an explicit transaction. The exceptions are when using joined tables or when accessing a cluster through a gateway.

Some implications:

- A command that attempts to access an object being updated by another command blocks until the first command completes. But it blocks only until the individual command completes, not until the transaction is closed.
- Deadlocking on data object locks does not happen when using TIBCO Patterns. By isolating locks within a single command, TIBCO Patterns can ensure that these locks are acquired in a deadlock safe manner (a hierarchical allocation scheme is used). This means that the mechanisms provided by RDBMS to prevent deadlocks, such as timeouts on locks and deadlock detection algorithms, are not needed.

It might seem that this method of using marks instead of locks puts a burden on the application. Although it might appear the application must implement a check and retry mechanism to handle transaction conflict errors, in nearly all cases it is not necessary. A properly functioning application should never get a transaction conflict error. A conflict error is generated only when two or more transactions attempt to make conflicting changes to the same object at the same time. The final state of the object becomes unpredictable as it depends on which transaction happened to get in first. Leaving data in an unpredictable state is not desirable. Thus, receiving a transaction conflict error from TIBCO Patterns indicates a problem in the application that should be corrected. If applications simply ignore all transaction conflict errors, the effective results are the same as if blocking locks had been used. (The one exception is if using a clustered arrangement where both commands might fail instead of one or the other. If you use blocking locks, this situation is likely to result in a deadlock situation, which the application would need to handle, so there is still no net gain in simplicity for the application.)

Marking and Access Rules

Marks on an object are not necessarily mutually exclusive. In some situations, it is safe to allow two or more transactions to perform certain operations on the same data object at the same time. The main example is updating records in a table. Two or more transactions might update records in a table at the same time, as long as they are different records. The general rule is that commands that modify an object as a whole are mutually exclusive, whereas commands that modify only some portion of an object are not. Currently, all operations on Thesauri, Character Maps, Learn Models and Records operate on the associated object as a whole, and are thus exclusive. Commands that operate on the table as a whole and are thus mutually exclusive are: add (that is, create or load), delete, rename (that is, move), checkpoint, restore and bulk (that is, fast) loads of records. (The bulk (or fast) load of large batches of records is considered to change the table as a whole for performance reasons.) The other record operations are exclusive at the record level, but not at the table level. However, they are exclusive at the table level with operations that modify the table as a whole. Thus, two different transactions can update records on the same table at the same time (as long as they are different records), but one transaction is not allowed to delete a table when another transaction is updating records in that table.

The rules of marking mentioned before pertain to two or more different transactions. The same transaction can act on the same object multiple times. Thus, the same transaction might add a new table, load it with records and then rename the table. These marks do not restrict actions performed by the same transaction.

Marks are retained until the transaction that created them is closed. This is true even for objects that are deleted. When a table is deleted, it is no longer visible, even before the transaction is closed. So in the following sequence:

1. Transaction 1 deletes table A.
2. Transaction 2 adds table A.
3. Transaction 1 is committed.

Transaction 2 receives a transaction conflict error at step 2. This is necessary to maintain the integrity of the data objects. If the application aborts the first transaction, it must be able to restore the deleted table. If another transaction were allowed to add a new table with the same name, the restore would either fail or be forced to delete the new table.

When dealing with joined tables, there is one other kind of mark that is placed on a parent table. When a child table is added to the parent table, it is marked as *held*. Although no change is being made to it, the parent table is necessary for the existence of the new child table. So actions on that table are blocked until the transaction is closed.

Unlike the marks discussed previously, marks indicating a data object are held block actions by all transactions, even the transaction that created the mark. Adding a new child table and then deleting its parent is invalid even when performed in the same transaction. Held marks are not mutually exclusive, however, two or more transactions might hold the same parent table at the same time. The same transaction might also place multiple held marks on the same parent table.

Dirty Reads and Object Status

As mentioned, all reads are *dirty reads*. That is, they show the object state as of the last completed command, regardless of whether the transaction is closed or still open. Read operations include all list commands, the inexact search (that is, query) commands, and the record get commands. For example, a user opens a transaction and deletes a table under that transaction, and then runs a command to list all tables, the deleted table does not appear in the list. If the application aborts the transaction, the table is restored and a table list operation shows the restored table.

For all of the list commands, information on any marks on the object is returned along with the other statistics on the object. The user can use this to determine if an object is in a “dirty” state, and thus potentially invalid. Query commands also return a dirty flag for each record returned. An application can use this flag to disregard “dirty” records if required.

Deleted objects are not listed or returned even though the transaction that deleted them is still open. As mentioned previously, marks are retained for the deleted objects. You need to watch for the following scenario:

1. An application opens a transaction and deletes a table.
2. Another application lists tables and detects that the table is missing so tries to restore it and receives a conflict error.
3. The first application commits the transaction.

When using a gateway in a clustered configuration, the mentioned scenario can occur even if explicit transactions are not used. Deleted objects are not visible, so you cannot see that they are “dirty” until you attempt to modify them.

Transaction Size, Memory, and Performance

There is no fixed limit on the number of commands that can be issued under a single transaction. However, there are memory size and performance constraints on the size of a transaction. The memory used by a deleted object is not freed until the transaction is committed. Thus, an application that deletes a large number of tables or a very large table and then recreates them under the same transaction might find that the TIBCO Patterns server runs out of memory because it is holding two copies of all of these tables. The same is true if a very large number of record operations are performed under the same transaction.

A performance impact is associated with very large transactions. Operations on thesauri, character maps or Learn models are generally not a concern as it is very rare that hundreds, much fewer thousands of these operations are performed in a single transaction. However it is not unusual for hundreds of thousands of records to be updated at once. Performing hundreds of thousands of individual record operations in a single transaction significantly impacts performance and memory usage. If a very large number of records are to be added, use a bulk (fast) load. This avoids the creation of tracking information on individual records, which saves both time and memory. (But this locks the entire table, excluding other transactions from updating records.) Use explicit transactions only when necessary and keep them as small as possible. As a rule, limit non-bulk record operations to about 64K – 100K records in a single transaction. The TIBCO Patterns server handles far larger numbers, but there might be significant performance impacts.

There is no fixed limit on how long an explicit transaction can be held open. However, open transactions can delay administrative operations and the server's internal bookkeeping. Explicit transactions should be committed or rolled-back as quickly as possible.

There is no fixed limit on the number of explicit transactions. However, keeping a large number of explicit transactions impacts both memory usage and server speed.

Joins

This section describes the joins feature and its implementation with examples.

- [Overview](#)
- [Tables](#)
- [Records](#)
- [Compound Records](#)
- [Joined Searches](#)

Overview

TIBCO Patterns supports the concept of *joins*. Using *joins* you can search data spread across multiple tables in a way that the fields of each table appear to be merged into a single table.

Before the joins feature was introduced, there were two basic methods for performing a join query. The first is to merge the records of the tables before loading them into a TIBCO Patterns server using a *denormalization* process. The denormalization process involves creating a separate merged record for each valid combination of records from the individual tables. Denormalization is practical when only two tables are involved. For more than two tables, denormalization can result in creating very large tables. An update operation on a record in one of the separate tables requires updating many records in the merged table, making it difficult to maintain the denormalized table.

The second method is to query each table separately and then merge the query results. With this approach, you might lose accuracy. The result might not include the best merged record. The best merged record might contain one or more individual records that match very poorly. If each table is queried individually, these poorly matching records are not returned. Hence, the best merged record is lost.

Using the joined tables feature of TIBCO Patterns to do a joined search eliminates the issues with denormalization and merging of separate query results.

Joins Example

A joined search involves querying or retrieving data from fields across multiple tables. Let us consider three tables:

Tables	Fields
Persons	first_name, last_name, dob, SSN
Addresses	street, city, state
Phones	number, type

A joined search across the three tables is used to perform the following searches:

- Retrieving all records with name "John", "Smith", who lives at "123 Main St.", "Clarksburg", and with phone number "609-883-1010."
- Retrieving addresses and phone numbers of records with the name "John", "Smith", born on "June 22, 1963."

The second search queries only one table but retrieves data from three separate tables. Therefore, a joined search is needed.

To perform these searches, there must be a connection that ties the three tables together. The most common approach in SQL are to provide a foreign key field in each child table. The contents of the foreign key field is the unique key value for the parent record. For our example, let us assume the unique key for the "Persons" record is contained in a field named "id" in the Persons table. We assume the "Addresses" and "Phones" table each have foreign key fields named "person_id". We also assume the "Addresses" and "Phones" table have unique key fields of their own named "addr_id" and "phone_id" respectively.

An SQL statement to perform the joined search is:

```
SELECT * FROM Persons, Addresses, Phones
      WHERE Persons.id = Addresses.person_id AND Persons.id
      = Phones.person_id
      AND Persons.first_name = "John" AND Persons.last_name
      = "Smith"
      AND Addresses.street = "123 Main St."
      AND Addresses.city = "Clarksburg"
      AND Phones.number = "609-833-1010"
```

The FROM clause gives a list of tables involved.

The WHERE clause gives the linking between the records.

In TIBCO Patterns this linking is done when tables and records are created, unlike in SQL, where it is done at query time.

Tables

In TIBCO Patterns, three types of tables are available: Standard tables, Parent tables, and Child tables. The type of table is set at the time the table is created and does not change throughout the life of the table.

Standard Table

A standard table does not share a parent or child relationship with any other table. Before the joins feature was introduced, all tables were standard tables. Joins operations cannot be performed on standard tables. Standard is the default table type.

Parent Table

A table designated as a parent table when it is created can act as the root, or parent in a join relationship. A parent table can have multiple child tables. The child table is linked to its parent table when it is created. So, it is necessary to create the parent table before creating the child table.

In [Joins Example](#), "Persons" is the parent table for the joined set of tables: "Persons", "Addresses", and "Phones."

A parent table is also a standard table. Parent tables behave as standard tables when regular searches are performed on them. The process for adding, deleting, or updating records of a parent table is the same as that of a standard table. However, there are a few limitations when working with parent tables.

Restrictions

- **A parent table cannot have more than one primary predicate index associated with it.**

Although it is allowed to have more than one primary index on a standard table, it is not recommended. So, this is not a major restriction. For more information about partitioned indexes, see [Predicate Indexes and Joined Searches](#).

- **A parent table cannot be deleted if one or more child tables are linked to it.**
You cannot have a child table without a parent table. The TIBCO Patterns server handles this by rejecting the deletion of any parent table that has one or more child tables linked to it. This also includes delete operations performed using the table rename function. When reloading a parent table, the standard method of loading the new version of the table to a temporary table name, and then renaming it to the permanent name, does not work. The rename operation fails because the existing parent table cannot be deleted unless all the child tables are deleted.
- A parent table cannot be a child table. Therefore, TIBCO Patterns does not support cascaded or multiple level join structures.

Child Table

A child table is the leaf of a joined set of tables used in a joined search. In [Joins Example](#), "Addresses" and "Phones" are child tables because the link is from the records of the "Addresses" and "Phones" tables to the records of the "Persons" table. In TIBCO Patterns, a table must be designated as a child table when it is created. The parent table of the child table is specified at the time it is created and is fixed for the life of the table and cannot be modified.

- The relationship between child and parent tables is between physical tables, not table names. Renaming either the child table or the parent table doesn't alter the relationship between them.
- In TIBCO Patterns a child table cannot be a parent table.
- You cannot have a child table without a parent table.

- Regular searches can be performed on a child table. A child table is also a standard table and in most respects behaves like a standard table.

Special considerations are involved when using a predicate index on a child table. For more information about predicate indexes and memory usage, see [Predicate Indexes and Joined Searches](#).

Considerations

The roles of tables are fixed at the time they are created. Therefore, a table might have only one role. There are restrictions on the types of join relations that are supported. Only a star schema join relationship is supported. That is one parent table with any number of child tables. Cascading and many-to-many schemas are not supported.

Records

Records in a child table, whether standard records or child records, can be in an orphan state, that is, they have no parent. Records in the orphan state are called *orphan records*.

Standard Records

Before the introduction of the joins feature all records were standard records. A standard record cannot be joined to any other record. It contains a unique record key and a set of one or more data fields. It can be added to any of the table types. When added to a parent table, a standard record becomes a parent record. However, if added to a child table, a standard record remains a standard record. It is never linked to a parent record.

Parent Records

A parent record is a standard record with the additional feature that allows child records to be linked to it. Adding a record to a parent table makes it a parent record. Therefore, all records in a parent table are parent records. Standard and child tables cannot hold parent records. Add, update, and delete operations are performed in the same way as standard records. Parent records can have multiple child records.

Child Records

A child record is a standard record with the addition of a parent record key. A standard record has a single key value that forms the unique key for the record. A child record has two key values: the record key and the parent record key. The combination of the two forms the unique key value for a child record. As a child record has only one parent key value, it has at most one parent record. In the case of the [Joins Example](#), the “person_id” fields of the “Addresses” and “Phones” tables are the parent record key of a child record. In TIBCO Patterns, the parent record key of the child record is the foreign key value for the record. It defines the link to the parent record.

Child records can only be added to a child table. The link between the child record and a parent record is established at the time a child record is added, or the associated parent is added or deleted. The link is fixed for the life of the child record and parent record. The parent key value in the child record is used to determine the parent record it is linked to. The parent record is always in the parent table of the child table the child record is added to. Similar to the link between the child table and the parent table, the link between the child record and the parent record is a link between physical records, not key values. The link exists until either the parent record or child record is deleted. Links survive updates of either record.

- If a parent record with the parent key value does not exist at the time the child record is added, it becomes an orphan record.
- If the parent record of a child record is deleted, the child record becomes an orphan record.
- When a parent record is added, any child records with that parent key are linked.

Parent/Child links are established regardless of the order the records are added, but adding a parent record after its children carries a significant performance penalty. When adding a large number of records, it is much faster to add the parent records first.

Orphan Records

An orphan record is any record in a child table that is not linked to a parent record. All standard records in a child table are always orphan records as they cannot be linked to a parent record. A child record can also be an orphan. This happens when the parent table contains no record with the parent key contained in the child record.

A child record that is an orphan is still a child record. You must provide both portions of the key, record-key, and parent-record-key, even though the record is not linked to a parent record.

Considerations

Because a child record has at the most one parent record, and a child record can appear only in a child table, all linkages in TIBCO Patterns is one-to-many linkages. Many-to-many relationships between records are not supported.

Compound Records

A *compound* record is a parent record and all of the child records are linked to the parent record. For each child table of a parent table, a compound record can have zero or more child records. TIBCO Patterns provides various methods for operating on compound records. Compound records can be retrieved based on the parent key or a cursor on the parent table. Record delete operations can be performed on compound records. All child records for a particular parent key can be retrieved or deleted. In all operations on compound records, the parent table and the set of child tables (to be included in the compound record) can be specified. The set of child tables specified can be any subset of the set of child tables linked to the parent table. The operation is applied only to those child tables. For most commands, the default is to apply the operation to all child tables of the parent table.

Higher level APIs, such as the Java and .NET APIs, provide direct support for compound records. There is an object type that represents a compound record, and methods that return records of this type or explicitly perform operations on compound records. Operations on compound records must perform independent operations on each of the tables involved. The server does not directly support compound operations. The Java and .NET APIs use transactions to ensure any updates are committed or rolled back on all tables as a whole. This is handled within the methods and is not visible to the user. However, be aware that separate operations are being performed. If compound record read and update operations are performed in parallel, the read operation can return a partially updated compound record. This is because reads are "dirty" and child tables are updated in separate commands.

The low level "C" API does not have any direct support for compound records. It provides methods for retrieving all child keys linked to a particular parent key or set of parent keys. Given this information, higher level operations can be implemented.

Joined Searches

The equivalent Java API query in TIBCO Patterns for the example mentioned in [Joins Example](#), is:

```
NetricsJoin join_def = new NetricsJoin("Persons",
NetricsJoin.JOIN_FULL_AND_PARTIALS,
                                false,
                                null,
                                String [] { "Addresses", "Phones" }
                                );
NetricsQuery q_first_name = NetricsQuery.Simple(
                                "John",
                                new String [] { "Persons.first_
name"}},
                                null
                                );
NetricsQuery q_last_name = NetricsQuery.Simple(
                                "Smith",
                                new String [] {"Persons.last_name"},
                                null
                                );
NetricsQuery q_street = NetricsQuery.Simple(
                                "123 Main St.",
                                new String [] {"Addresses.street"},
                                null
                                );
NetricsQuery q_city = NetricsQuery.Simple(
                                "Clarksburg",
                                new String [] {"Addresses.city"},
                                null
                                );
NetricsQuery q_phone_num = NetricsQuery.Simple(
                                "609-884-1010",
                                new String [] {"Phones.number"},
                                null
                                );
NetricsQuery query = NetricsQuery.And(null,
                                new NetricsQuery [] {
                                    q_first_name,
                                    q_last_name,
                                    q_street,
                                    q_city,
                                    q_phone_num
                                }
                                );
```

```
NetricsSearchCfg search_def = new NetricsSearchCfg(join_def);
search_def.setNetricsQuery(query);
```

There is no equivalent to the WHERE clause expression of the SQL statement. A joined search must always have a parent table. It must also have at least one child table, and the child table must be a child of the given parent. However, it is not required to list all the child tables of the parent. Any subset of the child tables of the parent can be included in the joined search.

In a joined fuzzy search, the relationship between records is firmly fixed, it is only the matching of values in the records that is fuzzy. The relationship between records is fixed when the records are created.

- A fuzzy joined search is always equivalent to a standard fuzzy search on a single table that is a fully denormalized version of the joined tables.

A fully denormalized table has all the fields from each of the individual tables. For example, consider the three tables to have the following records:

Persons Table

id	first_name	last_name	dob	SSN
1	john	smith	6/23/1963	012345678
2	john	smythe	5/23/1960	021345678
3	jim	myth	8/14/1984	918273645

Addresses Table

addr_id	person_id	street	city	state
1	1	123 Main	Clarksburg	NJ

addr_id	person_id	street	city	state
2	1	St	Robbinsville	NJ
		32 Route 33		
1	2	213 Main St	Clarksburg	NJ
3	1	312 State Rd	Princeton	NJ
1	3	231 Ferry Rd	Ewing	NJ

Phones Table

phone_id	person_id	number	type
1	1	609-833-1001	home
2	1	732-981-1100	cell
1	3	609-442-0101	cell
1		609-833-0011	none

The fully denormalized table:

Denormalized Table

id	first_name	last_name	addr_id	street	phone_id	number	type
1	john	smith	1	123 Main St	1	609-833-1001	home
1	john	smith	1	123 Main St	2	732-981-1100	cell
1	john	smith	2	32 Route 33	1	609-833-1001	home
1	john	smith	2	32 Route 33	2	732-981-1100	cell
1	john	smith	3	312 State Rd	1	609-833-1001	home
1	john	smith	3	312 State Rd	2	732-981-100	cell
2	john	smythe	1	213 Main St			
					1	609-883-011	none
3	jim	myth	1	231 Ferry Rd	1	609-442-0101	cell

There is a separate record for each possible combination of parent and child records. As there were three child records for “John Smith” in the Addresses table and two in the Phones table, you can see 6 separate records for “John Smith” in the de-normalized table. There are two rows that are not fully populated. One is a parent record that has no child in the “Phones” table, and the other is an orphan record in the “Phones” table.

Specifying Field Names

TIBCO Patterns closely follows the SQL conventions for handling field names in joined searches.

- The table name precedes the field name and is separated by a period. Note that, as TIBCO Patterns allows blanks in both table names and field names, no blank separators are allowed on either side of the period.
- A table name qualifier is only required when the field name is not unique across the set of tables in the joined search.
- As in SQL, an alias can be assigned for a child table. However, unlike SQL, an alias name cannot be assigned to the parent table in a joined search. When an alias name is assigned, it must be used instead of the actual table name in all the field names for that query.

Modifying the example to include aliases for the child tables the SQL query is:

```
SELECT * FROM Persons, Addresses AS A, Phones AS P
WHERE Persons.id = A.person_id AND Persons.id = P.person_id
...
AND A.street = "123 Main St."
```

TIBCO Patterns Java API:

```
NetricsJoin join_def = new NetricsJoin("Persons",
NetricsJoin.JOIN_FULL_AND_PARTIALS,
false,
String [] { "A",
"P" },
String [] {
"Addresses", "Phones" }
);
...
```

```

NetricsQuery q_street = NetricsQuery.Simple(
    "123 Main St.",
    new String []
{"A.street"},
    null
);

```

There are some restrictions on alias names.

- All alias names must be unique.

Invalid Example

```

NetricsJoin join_def = new NetricsJoin("Persons",
    NetricsJoin.JOIN_
FULL_AND_PARTIALS,
    false,
    String [] { "A",
    "A" }, // invalid!
    String [] {
    "Addresses", "Phones" }
);

```

- An alias name cannot be the same as a table in a different position.

Invalid Example

```

NetricsJoin join_def = new NetricsJoin("Persons",
    NetricsJoin.JOIN_
FULL_AND_PARTIALS,
    false,
    String [] { "A",
    "Addresses" }, // invalid!
    String [] {
    "Addresses", "Phones" }
);

```

Invalid Example

```
NetricsJoin join_def = new NetricsJoin("Persons",
                                       NetricsJoin.JOIN_
FULL_AND_PARTIALS,
                                       false,
                                       String [] {
"Phones", "P" }, // invalid!
                                       String [] {
"Phones", "Phones" }
                                       );
```

Valid Example

```
NetricsJoin join_def = new NetricsJoin("Persons",
                                       NetricsJoin.JOIN_
FULL_AND_PARTIALS,
                                       false,
                                       String [] {
"Addresses", "P" }, // OK.
                                       String [] {
"Addresses", "Phones" }
                                       );
```

- Alias names must be used when the same child table appears multiple times in the child tables list. Each child table must have a unique identifier. If the same table is used multiple times it must be assigned different alias names.

The special parent key field name

A special field name: “^parent”, that is, the special character caret (Unicode code point 5E) followed by the word “parent”, is used to refer to the parent record key value of a child record. This special field name is used only for joined searches. As this special field exists for each child record it must always be qualified with a table (or alias) name when more than one child table is specified in the search. This special field is treated as having a field

type of non-searchable text. Therefore, it can't be used in simple or cognate searches. It can be used in predicate expressions, in both filtering predicates or query predicates.

Joined Records

The result of a joined search is a joined record. In SQL, the fields to be returned are specified in the SELECT clause. But in TIBCO Patterns, all the fields of a record are always returned. There is no "SELECT" mechanism for selecting fields to be returned. This is true for standard searches as well as joined searches. In a joined search, all the fields from all tables are always included for every record that is returned. It behaves as if they were records from the fully denormalized table, including the special parent record key fields and the record keys from each child table.

In the example mentioned, the TIBCO Patterns joined records would have the following fields:

Joined Records

Sample Values				
field_name	Record 1	Record 6	Record 7	Record 8
key	1<tab>1<tab>1	1<tab>3<tab>2	2<tab>1<tab>	<tab><tab>1
parent-record-key	1	1	2	
Persons.first	john	john	john	
Persons.last	smith	smith	smythe	
Persons.dob	6/23/1963	6/23/1963	5/23/1960	
Persons.SSN	012345678	012345678	021345678	
Addresses.	1	3	1	
Addresses.^parent	1	1	2	
Addresses.street	123 Main St.	312 State Rd	213 Main St	

Sample Values				
field_name	Record 1	Record 6	Record 7	Record 8
Addresses.city	Clarksburg	Princeton	Clarksburg	
Addresses.state	NJ	NJ	NJ	
Phones.	1	2		1
Phones.^parent	1	1		
Phones.number	609-883-1001	732-981-1100		609-883-0011
Phones.type	home	cell		none

Description of Terms

Term	Description
key	<p>This is the unique key value for the joined record. It is not a data field. Hence, not included in the list of field names. A joined record has the same structure as any other record. This includes having a unique key value. For a joined record, the unique key value is the key value of each of the component records separated by the tab character (represented in the samples by “<tab>”). It is an ordered set with the parent table first, followed by each child table, in the order specified in the join definition for the search. If a record for a table does not exist, there is no key value, but the “<tab>” separators are still included. This can be seen in the Record 7 and Record 8 sample values.</p>
parent-record-key	<p>It is the special parent record key that is included with all child records. Like the key, it is not a data field in the record and does not have a field name. A joined record is returned as a child record if it contains any child records. The parent record key value is the parent record key value of the child record. Note that all child records in a joined record will always have the same parent record key value.</p> <p>If a joined record does not contain any child records, it is returned as a standard record. This standard record does not contain a parent</p>

Term	Description
	record key value. Note that Record 8 does not have a parent record key value. This contains a single orphan record from a child table, but the record is a standard record without a parent record key. So the returned joined record is a standard record without a parent record key. If the orphan record was a child record with a parent key, the returned joined record would be a child record and contain a parent record key value also.
Persons.first, Persons.last, Persons.dob, Persons.SSN	Data fields from the Persons table
Addresses.	This is a special data field that contains the key value of the Addresses record. The key value of each child record is included as a data field in the returned record with a field name that consists of the table name (or table alias) followed by a period. It has no field name parameter. The individual record key values can be retrieved using these special fields if required. Although this appears in the returned joined record as a data field, it cannot be used as a field in a search query.
Addresses.^parent	This is the special parent record key field discussed in the section The special parent key field name . It contains the parent record key value from the child record. As mentioned earlier, this field is accessible as a data field in a query. But as it is considered a text-only field and not a searchable text field, it can only be used in predicate expressions.
Addresses.street, Addresses.city, Addresses.state	Data fields from the Addresses child table
Phones.	<p>This is the special data field containing the key value of the Phones record. Note that Record 7 has no value for this field.</p> <p>For a child table, if no record is included in the joined record, this field is empty. As mentioned earlier, all fields are always included in the returned record, even if no record is included for that table. You can</p>

Term	Description
	find the difference between a field that is empty because the record had no data for that particular field and a field that is empty because no record for that table was included by checking the associated key field value. If it is empty, no record was included (a record must have a non-empty key). If it is populated, the record is included, but had no data for that field.
Phones.^parent	<p>This is the special parent record key field for the “Phones” table. Note that for Record 8 this field has no value.</p> <p>Depending on the join type, standard records can be returned in a search as orphan records. Standard records have no parent record key value and always appear as orphan records.</p>
Phones.number, Phones.type	Data fields for the “Phones” table

Classification of Joined Records

A joined record can be classified into 3 types:

1. Full Record

A full record is one that contains a record from each of the tables in the joined search. In the previous example Records 1 and 6 are full records.

2. Partial Record

A partial record is a record that has no records for one or more of the child tables in the joined search.

Record 7 mentioned in the [Joined Records](#) table is a partial record.

3. Orphan Record

An orphan record contains no parent record. As records are linked through the parent record, you cannot link two or more child records without a parent record. Therefore, an orphan joined record always contains exactly one child record. Record 8 in the previous example is an orphan record.

Join Search Modes and Types

Two options are available for controlling the joined record selection process. They define:

- How to treat multiple versions of records with the same parent
- What types of joined records should be returned

The settings used depend on the question to be answered by the query.

Search Mode: Single Parent versus Multi-Parent

Consider a search for "John Smith" on the data presented in [Persons Table](#), [Addresses Table](#), and [Phones Table](#). Looking at the denormalized data in [Denormalized Table](#), you see that there are six possible combinations for the first parent record with the name "John Smith". If the goal is to find the correct "John Smith," there is no need to return all six combinations for the "John Smith" joined record. Only the best matching combination is needed. Returning additional combinations is harmful as it increases the chance that a valid match on a different parent record is lost because the returned set of records is filled with the unneeded combinations of the first parent record. When the objective is to find the best matching entity, it is best to use a single parent mode search. A *single parent mode search* returns only the single, best matching, joined record for a particular parent record.

Consider a different scenario, where you need to find all addresses and phone numbers for a particular person. Here, returning the one best combination for each parent would be incorrect as you are specifically interested in all the possible combinations of child records. When the focus is on the data in the child records, it is best to use a multi-parent mode search. A *multi-parent mode search* returns all combinations of child records for any parent record where the entire combination scores high enough to be included in the returned set of records.

The choice of a single parent or a multi-parent mode is based on the needs of the applications, but as a general rule:

- Use a single parent mode search where the desired data is in the parent record,
- Use a multi-parent mode search where the desired data is in the child records.

Consider the Java API example mentioned earlier:

```
NetricsJoin join_def = new NetricsJoin("Persons",
                                      NetricsJoin.JOIN_
FULL_AND_PARTIALS,
false,
                                      String [] {
"Addresses", "P" },
                                      String [] {
"Addresses", "Phones" }
                                      );
```

The selection of single parent versus multi-parent mode is controlled by the third argument. This is a Boolean flag. If set to `true`, the multi-parent mode is used. If set to `false`, the single parent mode is used.

Some important performance issues need to be considered when choosing the search mode. For more information, see [Performance Considerations](#).

Search Type: Selecting Record Types

The three types of joined records are full records, partial records, and orphan records. The type of record to be returned depends on the context of the search and the application. With TIBCO Patterns you can choose whether to return partial records or orphan records or both. Full records are always returned.

The four record type selections are:

- **Full Records Only**
This ignores partial and orphan records returning only fully populated joined records.
- **Full and Partial Records**
This returns full records and partial records but ignores all orphan records.
- **Full and Orphan Records**
This returns full records and orphan records but ignores partial records.
- **All Record Types**
This returns all joined records: full records, partial records, and orphan records.

In the example mentioned earlier, Full and Partial Records for a join type is used.

```
NetricsJoin join_def = new NetricsJoin("Persons",
    NetricsJoin.JOIN_FULL_AND_PARTIALS,
    false,
    String [] {
        "Addresses", "P" },
    String [] {
        "Addresses", "Phones" }
    );
```

This is based on the assumption that the address and phone number are supplemental information used to help identify the desired person, and it is not of primary importance. Assuming this, if a record matches well, despite missing some of the supplemental information, it should still be returned. So, partial records should be included in the return set. A **Full and Partial Records** join is the most suitable join type to use when:

- The primary focus of the query is on the information in the parent record.
- The child records represent optional supplemental information that helps identify a particular parent record.

Consider a scenario where the primary purpose of the search is to find the person living at "123 Main St" with phone number "609-833-1010" and it is assumed that the name is "John Smith". The query is the same, but the intent is different. In this case, returning partial records is not the right approach as you could say nothing about whether the address and phone number are the desired ones.

- The **Full Records Only** join type is the most suitable when the child record represents critical information.

Consider a scenario where the primary purpose of the search is to find who, if anyone, is living at "123 Main St" in Clarksburg. The focus is on the Addresses child record. The query is the same, but this case specifically requires orphan records. Records with no person associated with the address also need to be included. Partial records with no address information are useless in this case.

- The **Full and Orphan Records** join type is the most suitable when the primary intent is to determine information about a child record.

Consider a scenario where you need to check whether John Smith lives at "123 Main St" and has phone number "609-883-1001". If the address or number is unassigned, you have to check that too. You also have to know whether "John Smith" has no address at all.

- The **All Records** join type is the most suitable when the intent is to discover the relationship between the records.

Matching Compound Records and Querylet Grouping

A common matching problem for joined tables is the matching of *compound records* (see [Compound Records](#)). Here the input record might have zero, one or more child records for the same child table. Typically, when matching two compound records, the match is based on the best possible match of any one of the input child records for a particular child table against any one of the child records for the same table of the compound record being matched. That is the best match out of all possible combinations of child record comparisons is taken as the match score for the compound record. Consider the following input compound record for our Persons and Addresses tables (the Phones table is left out for brevity):

Input Compound Record Example

Table	Field Values
Persons	John,Smith,1968/12/23,111-22-3333
Addresses	123 Main St.,Gothem,NY
Addresses	456 2ND St.,Smallton,VT

With these two Addresses records, the match query for this might look like this:

```
NetricsQuery name_qry = NetricsQuery.Cognate(
    new String [] { "John", "Smith" },
    new String [] { "first_name", "last_name"
},
    null,
    0.9) ;
NetricsQuery dob_qry = NetricsQuery.Custom(
    "1968/12/23",
    new String [] { "dob" },
    null,
    NetricsQuery.CS_DATE) ;
NetricsQuery ssn_qry = NetricsQuery.Simple(
    "111-22-3333",
```

```

        new String [] { "SSN" },
        null) ;
NetricsQuery street1_qry = NetricsQuery.Simple(
    "123 Main St.",
    new String [] { "Addresses.street" },
    null) ;
NetricsQuery city1_qry = NetricsQuery.Simple(
    "Gothem",
    new String [] { "Addresses.city" },
    null) ;
NetricsQuery addr1_qry = NetricsQuery.And(null, new NetricsQuery [] {
    street1_qry,
    city1_qry
    }
    ) ;
NetricsQuery street2_qry = NetricsQuery.Simple(
    "456 2ND St.",
    new String [] { "Addresses.street" },
    null) ;
NetricsQuery city2_qry = NetricsQuery.Simple(
    "Smallton",
    new String [] { "Addresses.city" },
    null) ;
NetricsQuery addr2_qry = NetricsQuery.And(null, new NetricsQuery [] {
    street2_qry,
    city2_qry
    }
    ) ;
NetricsQuery addr_qry = NetricsQuery.Or(null, new NetricsQuery [] {
    addr1_qry, addr2_
qry
    }
    ) ;
NetricsQuery full_query = NetricsQuery.And(null, new NetricsQuery [] {
    name_qry,
    dob_qry,
    ssn_qry,
    addr_qry
    }
    ) ;

```

In this example, there is a separate query to match each child record, and then the "OR" combiner is used to select the one that matches best. This query will find the combination of Persons and Addresses that best matches the given Persons record and any one of the given child records.

However, there is a problem with this query that could cause it to perform poorly. That problem is related to the GIP prefilter that performs the join operation and makes the initial record selection (For more information about the GIP prefilter, see [Prefilters and Scaling](#)). The GIP prefilter does not know which querylets are associated with which child record; therefore, it assumes all querylets are associated with the same child record and attempts to select a child record that best matches *all* of the querylets. This can lead to very poor selections by the GIP prefilter. Because the GIP prefilter only makes a crude selection of a large candidate pool, it might work well enough for smaller tables. For larger tables, this can result in poor overall match results because the desired records might get pushed out of the candidate pool by many records that match all of the querylets, as good as or better than records matching the individual querylets. The *querylet grouping feature* allows you to group querylets by which child record they are associated with. By knowing which querylets came from the same child record and which came from different child records, the GIP prefilter can make better child record selections.

The querylet grouping feature allows you to associate a group name with one or more querylets on the query tree. These querylets can be of any type and can be at any position in the tree. When the GIP prefilter matches child records, it matches querylets with different group names separately. Typically a group is associated with a particular child record in the input compound record. A standard practice is to use the combination of the child table name and child record key as the group name. The *Input Compound Record Example* with group names added looks like this:

```
... // Parent record queries as above.
NetricsQuery street1_qry = NetricsQuery.Simple(
    "123 Main St.",
    new String [] { "Addresses.street" },
    null) ;
NetricsQuery city1_qry = NetricsQuery.Simple(
    "Gothem",
    new String [] { "Addresses.city" },
    null) ;
NetricsQuery addr1_qry = NetricsQuery.And(null, new NetricsQuery [] {
    street1_qry,
    city1_qry
    }
    ) ;
addr1_qry.setGroup("Addresses.key1") ;
NetricsQuery street2_qry = NetricsQuery.Simple(
    "456 2ND St.",
    new String [] { "Addresses.street" },
    null) ;
NetricsQuery city2_qry = NetricsQuery.Simple(
    "Smallton",
```

```

        new String [] { "Addresses.city" },
        null) ;
NetricsQuery addr2_qry = NetricsQuery.And(null, new NetricsQuery [] {
        street1_qry,
        city1_qry
    }) ;
addr2_qry.setGroup("Addresses.key2") ;
NetricsQuery addr_qry = NetricsQuery.Or(null, new NetricsQuery [] {
        addr1_qry, addr2_
    }) ;
NetricsQuery full_query = NetricsQuery.And(null, new NetricsQuery [] {
        name_qry,
        dob_qry,
        ssn_qry,
        addr_qry
    }) ;

```

Notice that for the address query, the group name is set on the AND query. Group names propagate down the query tree, thus setting the group on **addr1_qry** is the same as setting it on **street1_qry** and **city1_qry**.

Some things to note about querylet groups:

- Querylet groups apply only to queries on child records of a joined query. They are quietly ignored if used on querylets against parent records or in non-joined queries.
- A querylet can belong to only one group. An error occurs if two or more different groups are assigned to the same querylet. This includes assignments that were propagated down from a higher level in the query tree.
- A querylet group name can be any valid string. There is a length limit of 999 bytes when encoded as a UTF-8 string value.
- All querylets not assigned a group name belong to the same unnamed group.
- Querylets on different parts of the query tree can be assigned to the same group.
- Grouping only affects how child records are selected in the prefilter. It has no effect on the scoring of the record.
- Because a querylet can be assigned to only one group, a querylet that uses data from two or more different input child records cannot be assigned to a group.

- The primary use case for querylet grouping is in compound record matching. It is a good practice to use the querylet grouping feature to ensure accuracy when doing matching of this type. However, there is a performance penalty when querylet grouping is used. The amount of the penalty is very hard to predict, but if the load is high and performance is critical, it might be necessary to consider removing the use of the querylet grouping feature.

Predicate Indexes and Joined Searches

Many special restrictions and considerations exist when using predicate indexes on a joined table. A parent table can have at the most one primary index. As it is generally not advisable to have more than one primary index on a table, this is not a significant restriction. Apart from this restriction, predicate indexes on a parent table can be used as on any other table. A child table, however, has a number of significant restrictions on the use of predicate indexes.

Internally the join relationship between a child table and its records, and the parent table and its records, is represented as a primary index on the child table. This is similar to a primary predicate index on the child table. Essentially, a child table starts with one primary index. So, all the caveats associated with a second primary index apply to the first primary predicate index on the child table:

- The first primary predicate index on a child table incurs a large memory penalty, almost doubling the memory requirements for the table.
- As only one primary index at a time can be used, and as the join search requires the join index be used, a primary predicate index on a child table cannot be used in a joined search. It is applied as a secondary index.

A primary predicate index should be placed on a child table only if:

- The extra memory costs can be handled.
- There are many standard (non-joined) queries against the child table that can gain a major performance boost by using a primary predicate index.

Otherwise, a primary predicate index should not be put on a child table.

To obtain the performance boost available from a primary predicate index in a joined search, the partitioned field must be in the parent table. This should be considered while designing the tables. If you want to partition by state code, the state code should be in the parent record for the joined set of tables if possible.

A partitioned index works by allowing the search to skip over the partitions that do not apply. In a joined search, the partitioning applies to the parent table and to all its child tables. Using a partitioned index on a parent table you can skip over blocks of the parent table, as well as all of the associated blocks in the child records. Therefore, in a joined search, the performance boost from using a primary index on the parent table can be even larger than one would expect. As joined queries can be more expensive than a standard query, it is even more important to look for the possibility of using a filtering predicate and a primary index when performance is critical.

Performance Considerations

In addition to the standard factors of table size and query complexity, you need to consider a number of other factors for query performance in a joined search.

Fanout and Query Performance

Note that a joined query is typically a query across the fully denormalized representation of the parent and child tables involved in the query. Query performance is most closely tied to the size of the (virtual) fully de-normalized table.

- If there is a single child table, the typical size of the fully denormalized table is the size of the child table (if there are no orphans).
- When there is more than one child table it becomes more difficult to determine the size of the fully denormalized table. The size depends on the number of possible child record combinations for each parent record. If parent records tend to have many child records in a number of child tables, the denormalized table might be too large to be searchable.

In practice, the size of the denormalized table is associated with the typical or average fanout of the parent table records. The *fanout* is the number of child record combinations for the same parent record, where each child record is taken from a different parent table.

In the example mentioned earlier, the fanout for each parent record is:

Fanout

Person ID	Addresses # of children	Phones # of children	Fanout
1	3	2	6 (3x2)
2	1	0	1
3	1	1	1 (1x1)
Average	$5 / 3 = 1.67$	$3 / 3 = 1$	$8 / 3 = 2.67$

- If you know the average fanout, the query performance is proportional to the average fanout multiplied by the size of the parent table.
- If you do not know the average fanout, a crude estimate can be given by multiplying the average number of child records for the same parent record in the individual child tables. The average number of child records that a parent record has in a child table is calculated by dividing the number of child records in that child table by the number of parent records.

In the example, the average fanout estimated in this way is $1.67 * 1 = 1.67$. This is a lower bound. It assumes that the fanout for all parent records is identical. In the example, the fanouts for individual records is highly uneven. Therefore, the actual fanout is much larger. This must be considered while estimating the size of the denormalized table.

All of these methods give only a rough estimate of query performance. A query that finds a parent record with an unusually large fanout can be much slower than a query that does not find such record. The worst case performance of a query can be estimated by executing a query that matches and returns the parent record that has the maximum fanout.



Warning: Adding child tables to a query greatly impacts query performance, especially if these tables are large compared to the parent table. You should be very careful of high fanout factors when there is more than one child table.

Single Parent Search

The following performance considerations are applicable when using a single parent search.

Consider a single-parent mode query with an output size of 20 records. This is asking for the best combination of child records for each of the 20 returned parent records. Many combinations are processed in all phases of query processing till the best combination is determined. This can require analyzing many additional combinations in all earlier phases of query processing. Thus, a large fanout factor can greatly increase the amount of time and memory required to process the query.



Note: Be careful of single parent searches on multiple child tables with high fanout which can greatly impact query performance.

Example: Consider a query with four child tables. Consider a parent record that has 1,000 child records from each of these four child tables. That means there are $1,000 \times 1,000 \times 1,000 \times 1,000 = 1$ Trillion child record combinations for this parent record. To analyze all these combinations, the TIBCO Patterns server would consume all of the available memory. (Usually it filters most of these combinations out before actually generating them, but this is not always the case.)

When dealing with records with extremely large fanouts, the TIBCO Patterns server sets an upper bound on the number of combinations of child records. The limits are hard limits that cause the query to fail.

All combinations of child records for each parent record are generated, and two hard limits are applied. The first limit is on the fanout of a single parent record (which is set at 1,048,576 by default). If any parent record encountered in a search exceeds this maximum fan out limit, the entire query fails.

This is called the *single-fanout-limit*. It applies to both single parent mode and multi-parent mode searches.

The second limit is on the total number of combinations returned from an early query processing stage (which is set at 100,000 by default). The query fails if the total number of combinations exceeds this limit. This is called the *total-fanout-limit*. It applies only to single-parent mode searches.

The single-fanout-limit is higher than the total-fanout-limit because the single-fanout-limit applies at a much earlier phase in the query processing. The cost of processing a record in

the earlier phase is much lower. Most records are filtered out before the total-fanout-limit is applied.

The following values are set by using the `-J` Command-line option:

- The single-fanout-limit, using the *single-parent-rec* parameter of the `-J` command line option.
- The total-record-limit, using the *all-parent-recs* parameter of the `-J` command line option. This limit must always be greater than the GIP output size limit. The default GIP output size limit is the greater of 2,000 or 37 multiplied by the number of matches requested.

For more information, see *TIBCO Patterns Installation*.

i Note: You can also override the default values for a particular query. Consult a TIBCO representative before modifying these values.

Some additional considerations:

If a joined set of tables has one or more records with an extraordinarily large fanout:

- Queries might have poor performance.
- Queries might fail due to exceeding the single-fanout-limit. If this happens, examine the TIBCO Patterns console log file. A warning message giving the parent record key of the offending record is added in to the log whenever the single record limit is exceeded.

If a joined set of tables has a very large average fanout factor:

- Queries might have poor performance.
- Single-parent mode queries might fail due to exceeding the total-fanout-limit. To get around this, you can increase the total-fanout-limit. However, increasing the total-fanout-limit can affect the performance.

Matching Compound Records

When matching compound input records, performance is impacted if the input record has a large number of child records. The standard approach to matching compound records adds a separate querylet for each input child record. When the number of input child records is very large, this can lead to a query with hundreds or thousands of querylets.

Such queries can become very expensive to process. In extreme cases, the TIBCO Patterns server might become unresponsive or it might run out of memory and terminate.

To prevent such issues, a limit is enforced on the total number of leaf querylets in a query (1,000 by default). Only score generator querylets count towards this limit. The limit can be set only at server startup, by using the *max-querylets* parameter of the -J command line option. For more information, see the TIBCO® Patterns *Installation* guide.

Special Considerations

Checkpoint and Restore Operations and Joined Tables

You need to consider some issues when performing checkpoint or restore operations on parent tables or child tables. For a general description of Checkpoint and Restore operations, see [Checkpoints of In-Memory Data](#).

There are links between parent and child tables, and between parent and child records. To maintain the consistency of these links, any checkpoint or restore operation must checkpoint or restore all of the linked tables as a unit in a single operation.

If they are not checkpointed or restored as a unit, it might cause inconsistencies such as:

- Incorrect linking of a restored child table to an unintended parent table
- Incorrect linking of a child record to a parent record
- Failure in linking intended records
- Failure in restoring a child table because the parent table does not exist

In some cases, error messages are displayed and corrections can be made. However, in some cases, incorrect linkages can be made with no errors being reported. Such quiet errors can result in propagating erroneous results. Such errors can be difficult to correct, once found. To avoid such scenarios, TIBCO Patterns server enforces some restrictions on checkpoint and restore operations on parent and child tables.

To help you understand the restrictions related to the checkpoint and restore operations, the term *joined set of tables* is used. A *joined set of tables* is a set consisting of a parent table and all the child tables linked to the parent table.

i Note: The entire joined set of tables must be checkpointed or restored as a single unit using a single command.

This restriction works differently for checkpoint operation and restores operation.

For checkpoint operations, if a table from a joined set of tables is included in the list of tables to the checkpoint, the entire joined set of tables is checkpointed.

For restore operations, it is required that all of the tables in the joined set of tables be explicitly listed. The server rejects any request that does not do so. Requiring all tables be explicitly listed in the restore request, instead of automatically completing a join set of tables, prevents inadvertently deleting existing in-memory tables.

For example, consider four tables:

- Table A: a standard table
- Table B: a parent table
- Table C: a child table linked to table B
- Table D: a child table linked to table B

If you request a checkpoint operation on Table A, only Table A is checkpointed. This is the standard operation. If you request a checkpoint operation on Table C, three tables, namely, Table B, Table C, and Table D, are checkpointed. This shows that if a request to checkpoint a single table from a joined set of tables is made, the TIBCO Patterns server checkpoints all tables in the joined set.

Consider the scenario where all the tables mentioned before are checkpointed.

1. Shut down and restart the TIBCO Patterns server without auto-restore.
2. Create a standard Table C.
3. Issue a request to restore Table A and Table B.
4. Assume the TIBCO Patterns server automatically restores all tables in a joined set, as it does for checkpoint operations. It then reloads Table B, Table C and Table D, destroying the Table C just created.

To avoid this possibility, the server restores only explicitly requested tables. It rejects requests that include an incomplete joined set of tables. In the prior example, the restore request is rejected in its entirety because the request for restoring Table B without Table C and Table D represent an incomplete joined set of tables. Unlike table not found errors, an

incomplete joined set error causes the entire request to fail and none of the tables are restored.

This requirement for complete joined sets applies to both in-memory tables and the tables checkpointed to the disk. These are not necessarily the same and in some situations it can make it impossible to checkpoint or restore tables. Consider the following situation.

5. Create four tables as in the previous example.
6. Shut down and restart the TIBCO Patterns server without auto-restore.
7. Create three new tables:
 - Table C: a parent table
 - Table B: a child table linked to Table C
 - Table E: a child table linked to Table C

8. Try to checkpoint Table C, Table B, and Table E.

You can observe the following:

- The checkpoint request fails as the checkpoint files on the disk contain a joined set of tables: Table C, Table D. The in-memory tables contain a joined set of tables, Table C, Table B, and Table E. The joined sets overlap but are not equal.
- If Table C, Table B, and Table E are checkpointed, the joined set of tables on the disk is broken and the original tables, Table B and Table C are lost.
- An attempt to restore Table B, Table C, and Table D results in destroying the in-memory Table C leaving Table E without a parent. So this is not allowed.

This situation can only occur when you start the server without selecting the auto-restore option (see the *TIBCO Patterns Installation Guide* for a description of the auto-restore option) and the checkpoint-restore directory contains existing checkpoint files.

Working with Mixed Record Types

i Note: Working with mixed sets of child and standard records is only appropriate when dealing with child tables as only child tables can include both types of records.

You can work with a mixed set of child records and standard records in a single command. Some points to be considered are:

- A parent record is a standard record that has been added to a parent table. It is created and used in exactly the same way as a standard record.
- A child record is distinct from a standard record. APIs have special functions, methods, and constructors for creating child records. In the Java and C# APIs, there is no separate class for child records. Both use the `NetricsRecord` class.
- When referencing a child record, both portions of the key must always be given. So there are distinct methods for deleting or fetching child records to supply the two components of a key. The standard methods can only be used to delete or fetch standard records or parent records.
- The record add and record update operations accept either standard records or child records, and perform the appropriate action based on their record type.
- While loading records from a CSV file, specify the type of records in the file, standard records or child records. This is done by identifying one of the fields in the CSV file as the parent key value, in the same way as a field in the CSV file is identified as containing the record key.

The delete and fetch method for child records can operate on standard records. If the parent key value is not provided, it operates on a standard record. The record add and record update operations can accept lists containing a combination of child and standard records. While loading from a CSV file, if the field designated as the parent-record-key field is empty, a standard record is created instead of a child record.

Prefilters and Scaling

In this section, the ways to optimize performance of your TIBCO Patterns installation are discussed. A key aspect of this is the configuration of the prefilter used by the server to accelerate the selection of candidates for inexact matching. It also discusses strategies for scaling your application for large data sets and workloads.

- [Prefilters](#)
- [Scaling for Large Data Sets and Workloads](#)


Prefilters

A prefilter accelerates matching by quickly eliminating irrelevant records from consideration before performing the full, inexact matching calculation.

The TIBCO Patterns server provides multiple prefilter technologies: GIP, SORT, and PSI. As each prefilter has certain advantages, the best suited prefilter varies by application.

References to table sizes mentioned in the Prefilter Technologies table roughly correspond to the following sizes:

- **Small table:** Less than 10 million records
- **Mid-sized table:** 10 to 20 million records
- **Large table:** Greater than 20 million records
- **Very large table:** Greater than 50 million records

 **Note:** These sizes are approximations and vary widely depending on the content of the records and the nature of the application.

Prefilter Technologies

Criteria	GIP	SORT	PSI
Use Case	The default prefilter. Most applications use the GIP prefilter. Excels at applications with a wide variety of queries.	Deduplication of mid-sized tables.	Deduplication of very large tables. Applications with a single fixed query against a large table where performance is critical.
Ease of Use	Very easy to use. No setup needed, all searchable fields are indexed. No additional query information is needed.	Default indexes might work, but manual definition of indexes might be needed. Search lookup fields might need to be specified along with the query.	
Query Time Performance	Good for small-sized to mid-sized tables. Query times increase significantly on large tables (greater than 20 million records).	SORT has the best query time performance. It scales well with table size.	PSI query performance similar to GIP for small-sized to mid-sized tables, but scales better for large and very large tables.
Record Load and Update Performance	GIP has excellent performance for all record operations.	Table load and record add and update performance is slower than GIP.	Table load and record add and update performance is the same or slightly slower than SORT.
Performance Tuning Options	Predicate filters and GPU accelerators can be used. With a suitable	Change the set of indexes to improve the query time performance. Reducing the number of indexes improves performance, but might harm accuracy. There is a tradeoff between accuracy and speed for SORT and PSI.	

Criteria	GIP	SORT	PSI
	<p>graphics card, GPU accelerators can increase performance by up to 4.5 times.</p> <p>In certain applications, where most records can be eliminated from consideration with a simple predicate test, using a predicate filter (See Predicates and Performance) can increase performance more than 10 times.</p>		
Query Accuracy	Excellent accuracy with a wide variety of queries and table sizes. For very large tables and queries with little information, special tuning of internal queue sizes might be needed.	For accurate results SORT must have data from many different fields to work with. SORT does not work well on large tables.	PSI works best when there is data from many different fields. It is more accurate than SORT. It has similar accuracy to GIP for many common query situations, but might not work well for queries on a single field or a small number of fields.
Memory Usage	The sizing rule of thumb for GIP is five	SORT requires the least amount of memory.	PSI can use more memory than GIP,

Criteria	GIP	SORT	PSI
	times the data size.		depending on the set of indexes defined.
Features Supported	Supports the following features: joins, variable attributes, predicate indexes, GPU acceleration, and thesaurus matching.	Does not support the following features: joins, variable attributes, predicate indexes, GPU acceleration, and thesaurus matching.	

Scaling for Large Data Sets and Workloads

TIBCO Patterns is designed to accommodate a wide variety of customer workloads with respect to data set size and query throughput while maintaining the highest matching accuracy. This section describes the scaling capabilities of TIBCO Patterns: multithreading, horizontal scaling and failover replication, and clustering.

Multithreading

TIBCO Patterns can make use of available computing resources by spawning worker threads to simultaneously process different queries to maximize throughput, thus providing scalability with very low overhead.

TIBCO Patterns algorithms use memory locality to optimize performance through judicious use of the processor cache. Almost always, the best performance in your production environment can be attained through the use of a dedicated machine running a single instance of TIBCO Patterns, with multithreading configured to use all available processor cores. For instance, if the dedicated machine has two quad-core processors with hyper threading, and thus a total of 16 hyper threading cores, you would start the TIBCO Patterns server with the maximum number of worker threads set to 16.

In production environments, to avoid contention for CPU resources and processor cache, it is suggested that the client application run on a separate machine from the TIBCO Patterns server.

Horizontal Scaling and Failover Replication

If a single dedicated server does not meet the requirements for the total throughput, an easy way to boost throughput is to distribute queries across several identical servers using industry-standard load balancers. Load balancing can also be used to provide an alternate server, or set of servers, in the event that the primary set of servers becomes unavailable. Since communications between TIBCO Patterns client applications and TIBCO Patterns servers are through standard TCP socket requests, standard load balancers or failover devices can fit seamlessly in front of a set of identical TIBCO Patterns servers.

Note that load balancing can only be used for read only operations such as queries and fetching records. For table updates, all instances of the TIBCO Patterns servers must be updated and therefore these operations cannot be performed through a load balancer, which chooses just a single instance to receive the request.

Clustering

Some very large tables might be too large to fit into memory on a single machine. Query latency times also tend to increase with table size, and might not meet requirements on a very large table when a single machine is used. To solve these problems, TIBCO Patterns allows tables to be split across a cluster of machines. A TIBCO Patterns server, called a *gateway* server, can be configured to act as a manager for this cluster. Applications perform all actions through the gateway server just like they perform them with a standard server, allowing applications to migrate from a single machine solution to a cluster solution with no change. From your client program's point of view, the tables appear to exist on the gateway as on a single instance. The splitting of tables and distribution of commands across the machines of the cluster is transparent to the application.

The strategies of horizontal scaling and clustering can be used in combination. The same way a load balancer distributes queries across multiple servers, it can distribute queries across multiple gateways, each managing its own independent cluster.

Clustering can decrease query latency time and increase total throughput to some degree. Horizontal scaling does not affect query latency, but increases total throughput to any

preferred degree. Combining these two allows handling very large tables with stringent throughput and latency requirements.

You can find the details on configuring a gateway and managing clusters in the *TIBCO Patterns Installation Guide*.

For the details on working with clusters, see the "Java API Reference" and ".NET API Reference" topics in *TIBCO Patterns Online References*.

TIBCO Patterns Machine Learning Platform

This section discusses the components of the TIBCO Patterns Machine Learning Platform: the Learn API library and the Learn UI application that is used to train and evaluate Learn Models. It also describes the ways to use trained Learn Models in TIBCO Patterns queries.

- [TIBCO Patterns Machine Learning Platform](#)
- [TIBCO Patterns Machine Learning Platform: Basic Concepts](#)
- [How to Use TIBCO Patterns Learn Models](#)

TIBCO Patterns Machine Learning Platform

TIBCO Patterns Machine Learning Platform provides machine learning models and associated supervised learning algorithms that are used for classification.

A set of training examples is provided where each example is marked as belonging to one or the other of two categories (True and False). The training algorithm builds a TIBCO Patterns Learn Model that assigns any new example to one of these categories, making it a binary classifier.

The TIBCO Patterns Machine Learning Platform is commonly used in matching and deduplication of records in a table to predict whether any two given records in a table match (represent the same entity) despite the differences in their field values. The file containing the trained Learn model is loaded to a TIBCO Patterns server and is used to provide more accurate record matches using the same query that was used to train the model. A trained model is associated with the query used to train the model and cannot be used with a different query.

TIBCO Patterns Machine Learning Platform incorporates a machine learning model trained to evaluate instances of problems defined by a particular set of features. In the context of TIBCO Patterns Machine Learning Platform, a *feature* is any characteristic of a problem that can be expressed as a real value between 0.0 and 1.0:

- The score 0.0 represents the "most false" condition for the feature.
- The score 1.0 represents the "most true" condition for the feature.

- A score between 0.0 and 1.0 represents proportional degrees of "true" and "false." A larger score is always associated with a more positive human judgment for the feature, or at least an unchanged judgment, but cannot be associated with a more negative judgment. That is, a decrease in the score of one feature cannot lead to an increase in the likelihood of the example being considered a true example.

For example, any score output by a query in TIBCO Patterns can serve as an input feature for TIBCO Patterns Learn model. However, the features for a Learn model are not limited to such scores. They can be real values obtained from other sources.

A machine learning model is a data component separate from the TIBCO Patterns server. The model can be created using the TIBCO Patterns Learn UI, programmatically using the TIBCO Patterns Learn API library, or TIBCO can create and tune the model for you as a contract service. For more information about creating a model, see *TIBCO Patterns Learn UI User's Guide* and the Learn API documentation.

Multiple machine learning models can be used in a single instance of TIBCO Patterns server.

Each model represents the domain intelligence needed to make positive or negative evaluation of feature vectors. A *feature vector* represents an instance of the classification problem and has one feature value for each feature.

TIBCO Patterns Machine Learning Platform: Basic Concepts

This section discusses the basic concepts related to the TIBCO Patterns Machine Learning Platform.

Sample Problem: Record Equivalence

The Learn Model in TIBCO Patterns performs a very simple task. The model is used to evaluate any given feature vector and obtain a single *model score* for that feature vector. When applied to matching problems, the Learn Model functions as a *score combiner* like the AND and OR combiners as described in [Designing Queries for TIBCO Patterns](#), but more mathematically sophisticated.

To understand the need for a more sophisticated score combiner, consider judging whether two records containing personal information for two individuals are equivalent, that is, whether the two records represent the same person or not. A human might consider many aspects of the two records that can be relevant to this judgment. Some of these are simple enough to be represented by a standard TIBCO Patterns query without a Learn Model. Some of the judgments are based on the combination of fields that appear similar enough, while other fields are allowed to be very different. This is the type of human judgment that a Learn model is designed to learn. Other types of judgment are based on external information known only to the human and not explicitly encoded in the match scores of the given field values. These judgments cannot be learned by the model unless appropriate information is added to the records being compared. It is important to understand the types of human judgment that the model can learn, and then provide only examples of such judgment for training the model.

The following are examples of features that do not require a Learn model. While each individual feature can be represented by a straightforward TIBCO Patterns query, the combination of scores from several such features can be used to train a Learn model:

- The name values are similar, despite fields being transposed (for example, the last name in one record is entered as the middle name in another record). A single cognate query can be used to take field transposition into account.
- The date of birth field values differ by only a single digit in the month or day of birth. A single date query can properly score such differences.
- The social security number matches exactly, indicating that it is the same person. This can be determined by a single simple query that uses the Social Security number field.

The following examples demonstrate human judgment that a Learn model is applicable for:

- The values of the name fields, such as first, middle, and last names, are very similar, or very different. The importance of the similarity of each field can be learned by the model.
- The first name, date of birth, and address are similar, while the last name is different, and the gender is female in both records. Depending on the country, this could mean that the woman has married and changed her last name.
- The Social Security number is very similar and the name fields match well, but all other fields match poorly. The match on the Social Security number probably outweighs the poor match on all other fields.
- The name fields are very similar, the state and ZIP code matches, but the street address is different, and the Social Security number is missing. The absence of an

important field, such as the Social Security number, might suggest putting more emphasis on the matching information in other field values that are present.

- The name, gender, and date of birth fields are very similar, but the address, state and ZIP code fields do not match. A strong match in only a few important fields might be enough to identify the equivalence of the two records. A model is able to learn the relative importance of matches in each field.
- The last name, street address, city, state and ZIP code fields match well, but the first name and apartment number fields are not very similar. A large amount of similar text in the record does not necessarily mean that the records are equivalent if certain important fields are very different.

The following examples demonstrate human judgment that rely on information unavailable to the Learn model:

- The first and middle names are abbreviated differently, for example, "F. Scott" and "Francis S.". It cannot be deduced from the field values what the abbreviations mean. However, a properly constructed thesaurus file could take common abbreviations into account.
- The last name in two sparsely populated records is a very common last name. Typically records have no information about how common the last name is. However, if this information is encoded in a separate numerical field, then it might be used for training a model.
- If the records are associated with new-borns in a hospital and the patient medical record numbers of two very similar records are sequential, this might suggest that the records represent twins. Text comparison of record numbers does not detect when the numbers are sequential, only that they are textually similar.
- The perfectly matching names and addresses are accompanied by a date of birth that differs by an entire generation. If so, there can be a name suffix like Jr. or Sr. A Learn model could never pick up on this information from a standard record comparison as the date query gives no information on the time between the two dates. However, a scoring predicate could be created that returns a score based on how close to a generation apart the two dates are. This would enable the Learn model to make such judgments. In addition, no information on name suffixes is available unless they are split out into a separate field, and even then the only information available is how well they match.

Your judgment on whether records represent the same person is complex, involving a large number of implicit patterns of relevant features. If a pair of records contain one or more of these relevant patterns, you tend to judge the records to be equivalent; and vice versa.

Example 1

NAME	DOB	SEX	STREET ADDRESS	CITY	STATE	ZIP
Bly, William	01/03/42	M	321 S. Orchard Ave.	Manitowoc	WI	54220
Lee, William	10/03/62	M	846 N. Orchard Ave.	Manitowoc	WI	54220

Despite the large amount of similar text, these two records represent different people. Not all of the text has equal relevance to the decision. In this example certain key similarities are lacking, such as the Date of Birth.

Example 2

NAME	DOB	SEX	STREET ADDRESS	CITY	STATE	ZIP	SSN
Smith, Jane	04/21/64	F	456 Orchard Av	Manitowoc	WI	54220	378-42-4481
Doering, Jane	04/21/64	F	1456 Willow Rd	Green Bay	WI	54301	378-42-4418

Despite the differences between these two records, they are likely equivalent because of the closeness of the Social Security numbers and the dates of birth, and because women in the U.S. often change their last names when they marry.

Example 3

NAME	DOB	SEX	STREET ADDRESS	CITY	STATE	ZIP	SSN
------	-----	-----	----------------	------	-------	-----	-----

Doering, K		M	456 Orchard Av	Manitowoc	WI	54220	387- 12- 7780
Doering, Kate	04/21/64	F	450 Orchard St	Manitowoc	WI	54220	

In this case, it is more tentative. The equivalence of the two records can be investigated based upon the matching last name and first initial, and a street address that is close but not identical. (The mismatched gender might be a data entry error.) The missing Social Security number represents a key value that would probably influence the judgment greatly if it were present; in its absence, these other features collectively wield enough influence to sway the judgment in the positive direction.

Collectively these examples illustrate several points:

- Even large amounts of similar values do not guarantee the existence of a relevant pattern of features that is sufficient for the two records to match.
- The relevance of a feature might depend on the match strength of other relevant features. For example, if name fields match closely, a matching Date of Birth becomes important as well; whereas a matching Date of Birth all by itself is not very significant.
- A feature that is usually important might lose its importance entirely in certain contexts – for example, the last name in the case of married females in the U.S.
- When features are missing, the criteria of user judgments can shift markedly.

Supervised Learning without Explicit Rules

The conventional approach to detect implicit patterns is the development of rule sets in which every relevant pattern is represented by an explicitly coded rule. Rule-based methods suffer from the limitations imposed by: the strictness of the rules themselves, the need to estimate and tune weight values (a process that tends to devolve into guesswork), and the need for completeness in the rule set, which is often unachievable in practice given the variability and complexity of real data. Moreover, while the rules detect some genuine occurrences of the pattern that each rule represents, each rule is also a source of *false positives*. These are records that are reported as matches, but are not true matches. In general, the less rigid the rule, the greater the number of false positives it generates. Since

rules must be coded directly, the addition of any new feature can involve time-consuming adjustments to the entire rule set.

The approach of TIBCO Patterns Machine Learning Platform is completely different. Rather than a programmer coding a large abstract rule set, a domain expert (who need not be a programmer, or even a technical user) is asked to make simple judgments of “Yes” or “No” when presented with a set of concrete examples. The saved "Yes" or "No" judgment is called a *label* of the feature vector.

The TIBCO Patterns machine learning algorithm uses these human judgments, together with the corresponding feature vectors, to generate a machine learning model that represents the “wisdom” of the human domain expert, without coding a single rule.

Since the scores output by TIBCO Patterns queries can be used as feature values, the “fuzziness” of features such as textual similarity is naturally taken into account.

The Learn model contains the result of the training. This takes into account all the dimensions of complexity identified in detecting implicit patterns, including the context-dependent relevance of features, and the way that human criteria change when features are missing.

Characteristics of TIBCO Patterns Machine Learning Models

A machine learning model can be created using the TIBCO Patterns Learn UI. See *TIBCO Patterns Learn UI User's Guide* for details. You can also create your own application for creating and training a model using the TIBCO Patterns Learn API (a Java library) or have TIBCO solutions engineers create and train a model for you. Every machine learning model uses a fixed set of features representing the information that might be relevant when making a particular type of decision. (In the [Sample Problem: Record Equivalence](#), this was a set of field-to-field comparison scores using TIBCO Patterns queries). To model the human decision, the chosen set of features should include all the information used to make a "Yes" or "No" decision, though it might also include irrelevant or marginally relevant features.

Defining the features is a crucial factor in developing a successful application using the TIBCO Patterns Machine Learning Platform. It is important to define the feature set so that all the relevant information is available to the Learn model. The consistency of the feature vectors and their labels that are used for training is also very important. While the machine learning algorithm is tolerant of accidental mislabeling of a few feature vectors used in

training, the human "Yes" or "No" decisions should be based only on the information that is included in the features. To train your own model, you should read the documentation provided with the TIBCO Patterns Learn UI or the Learn API carefully. You can consult the TIBCO experts while defining the features.

For a Learn Model to perform well, it must be trained with a sufficient number of examples covering all the different types of situations that might come up when in use. For example: in the [Sample Problem: Record Equivalence](#), if an incoming record has no address information, for the Learn Model to make a good prediction of whether this record matches another, the Learn Model must be trained with examples where the address information is missing. A Learn Model is said to be *well trained* if it was trained with a sufficient number of examples for all of the different matching situations that are likely to come up in real data. In practice, manually finding enough training examples to produce a well-trained Learn Model can be difficult. The Learn UI makes this task much easier. In particular, the low confidence pair finder feature of the Learn UI automatically finds and presents for labeling training examples that, if consistently labeled, ensures the Learn Model is well trained for the data loaded in the Learn UI.

A Learn Model that is well trained for the records loaded into the Learn UI data table might not be well trained for a full production set of data. The production data might contain matching situations that do not exist in the Learn UI data table. In these situations, the model might make incorrect predictions. To help detect and handle such situations, the Learn Model can output a *confidence value* with each prediction it makes. This is an approximate measure of how well trained the Learn Model is for this particular matching situation. The confidence value is a number between 0.0 and 1.0.

- A confidence value of 0.0 indicates the Learn Model had no training at all in this match situation, or that the training was completely contradictory, and thus this prediction is of very low confidence.
- A confidence value of 1.0 indicates the Learn Model was thoroughly trained for this situation with consistent training examples and thus this prediction is of very high confidence.

Depending on the release version of TIBCO Patterns that produced your Learn Model, different confidence measures might be available. Different confidence measures have different reliability. The most reliable measure, as of the 5.5 release of TIBCO Patterns, is the feature based confidence measure. However, this measure is also expensive to compute, and might not be suitable for applications with high query loads.

A Learn model is loaded into the TIBCO Patterns server. Like an in-memory table, a Learn model is a named in-memory object. Any number of Learn Models can exist at the same

time in the same instance of a TIBCO Patterns server provided enough free memory is available.

Every model is built to process a specific set of features. Therefore, the feature vectors evaluated by a particular trained model must represent the same features as the ones used to train the model. When matching records, this means that the record structure and the query used when training the model must be identical to the record structure and the query used for evaluating feature vectors with the trained model.

How to Use TIBCO Patterns Learn Models

After you create and train a Learn model, TIBCO Patterns provides two methods for using it.

- The first method uses the Learn model to evaluate any feature vector data.
- The second method integrates the Learn model with the search queries in TIBCO Patterns.

How to Use Learn Models to Evaluate Feature Vectors

You can use the Learn model for evaluating any feature vectors computed by one or more external applications. You can use the Learn model to detect hidden patterns in some particular problem domain, not tied exclusively to text or record matching. For example, fraud detection or spam detection might depend on various metrics depending on a user's pattern of interaction with a given system. The features in the feature vectors used for evaluation must be the same as the features used to train the model.

Two types of functions are required for feature vector evaluation:

- Loading and managing in-memory Learn models.
- Evaluating feature vectors using a selected in-memory Learn model. This is done using a function that obtains the score computed using the Learn model for the given feature vector.

How to Use Learn Models for Record Matching

This method makes it easy to use a Learn model to evaluate feature vectors consisting of query scores. This approach is suitable for advanced record matching applications, including record equivalence and deduplication applications.

Each feature score is computed by a TIBCO Patterns query. Each query that produces a feature score can be any of the query types described in [Designing Queries for TIBCO Patterns](#), including complex queries. An RLINK score combiner is used to combine the scores of all these queries and obtain the evaluation score using the Learn model. This allows a Learn model to be applied directly within the query to generate the match score for a record.

The RLINK score combiner is invoked like any other score combiner, for example, AND or OR. To configure an RLINK score combiner, provide the name of the model, along with the query inputs for each feature. The query configuration must exactly match the configuration used to train the model. When a model is exported from the Learn UI, it also exports a Java class with a static method that returns a `NetricsQuery` object. The returned `NetricsQuery` object incorporates the RLINK score combiner and the sub-queries for all features. Using the exported Java class ensures the correct configuration of the query.

Handling Low Confidence Predictions

In nearly all cases, it is not necessary to be concerned with low confidence predictions from a Learn Model. A Learn Model that is well trained for a data table containing a representative sample of records should have very few, if any, low confidence predictions. The recommended approach is to ensure the data table used to train the model contains data that is representative of all matching situations likely to occur, and that the model is well trained for this data table.

If it is not practical to obtain a representative sample of records, or if it is critical that you detect and handle all cases of low confidence predictions, two approaches are available for handling low confidence predictions. It should be noted that in both cases it is essential to understand the reliability of the confidence measure used and the meaning of different confidence values.



Note: Contact your TIBCO technical representative before using confidence measures.

A few points to know about confidence values:

- Every score generator produces a confidence value of 1.0.
- The RLINK combiner returns the confidence value generated by the Learn Model.
- The First Valid score combiner (as described in [Use the First Valid Score Combiner](#)) returns the confidence value of the selected child query.
- All other score combiners return their confidence value as the minimum confidence value of all of their child queries.

From these points, you can see that every query returns a confidence value.

- If RLINK combiners are not used (if a Learn Model is not used), the confidence value is 1.0.
- If one or more RLINK combiners are used, the confidence value is the minimum of the values produced by the RLINK combiners, unless a First Valid score combiner is used.

Handle Low Confidence Predictions in Your Application

In this approach, your application must check the confidence value returned for each query. The following three confidence values are available. You can use them based on your need.

Confidence Value Name	Description	Usage
Minimum confidence value	This is the lowest confidence value for any prediction made by the Learn Model during the processing of this query. It might be the confidence of a record that is not returned in the result set.	Used to ensure that the result contains all true matches. A low minimum confidence value might indicate that a matching record was incorrectly given a low score, and thus was not returned in the result set. Such value might also mean that a non-matching record was incorrectly given a high score and thus was returned.

Confidence Value Name	Description	Usage
		The application can flag this match for review, or it can issue an alternative query. You can also reissue the query wrapped in a First Valid score combiner to return those records that had a low confidence value. For information, see Finding Low Confidence Pairs .
Result set confidence value	This is the lowest confidence value of any record in the result set. It is the minimum of the individual record confidence values.	Used as a quick check to see if a result set contains any low confidence matches.
Individual record confidence value	This is the confidence for one of the records returned.	Used to ensure that query results do not contain false matches. Records with low confidence should be flagged for special processing or ignored.

Use the First Valid Score Combiner

The First Valid score combiner is designed for working with the confidence values output by a Learn Model. It selects a query based on confidence values. Each child query of the First Valid score combiner is assigned a confidence value threshold. The confidence values returned by the child queries are examined in order; the first child query that has a confidence value greater than or equal to its assigned threshold is selected as the result of the First Valid combiner.

A typical usage of the First Valid score combiner is to place an RLINK score combiner as its first child query, with an appropriate confidence value threshold. The second child query is a standard matching query that does not use a Learn Model. If the Learn Model prediction confidence value is low, the standard query is used. Otherwise, the Learn Model query is

used. In this way, the Learn Model is applied in those cases where it is well trained, and a fall back standard matching query is applied in those cases where it is not well trained.

The First Valid score combiner allows any number of child queries. Thus it is possible to have several alternative Learn Models in a single query. It selects the first Learn Model that is well trained for the particular match situation. The use case for multiple alternative models is exceedingly rare however.

Finding Low Confidence Pairs

The following are the reasons why you might want to find low confidence pairs:

1. To find new training pairs to improve a model. This is used in applications that dynamically adapt to changing data by retraining a Learn Model with new examples as they come in.
2. To find records that might have been lost because of an inadequately trained Learn Model.

The First Valid score combiner is the best way to find low confidence pairs. The First Valid score combiner has a flag that reverses its operation; instead of selecting the first child query with a confidence value greater than or equal to its threshold, it selects the first child query with a confidence value less than or equal to its threshold. This is called the *invalid only flag*. This flag can be used to return only those records that represent a poorly trained matching situation.

To find new training pairs to improve a model, a sample data set is "deduplicated" using an existing Learn Model. The RLINK query is wrapped as the first and only query of a First Valid score combiner with the inverse flag set. If no child queries of the First Valid score combiner satisfy the confidence value criteria, the -1.0 reject score is returned, causing the record to be rejected. Thus, a First Valid score combiner with a single RLINK child query and the invalid only flag set, returns only low confidence records. As all records returned represent matching situations that are poorly trained in the existing model, they are good candidates for inclusion in the training data set for the Learn Model.

A low minimum confidence value returned in a query indicates there might be one or more records that were not returned because they were assigned an incorrectly low score by a Learn Model that was not trained for the particular example. To find those potentially lost records, the original query is reissued as the first and only child query of a First Valid combiner with the invalid only flag turned on. The query returns only those records that had low confidence; the high confidence records are filtered out. In this way, the

potentially missed records can be found and processed as they are no longer pushed out of the search results by high confidence, high scoring records.

How to Use Learn Models on a Cluster

The clustering feature supports Learn models.

The Learn model can be loaded using a single command, and the gateway streams the model to each of the nodes. The Learn model file can be on either the client's file system or the gateway's file system. For more information about loading a Learn model through a gateway, see the *TIBCO Patterns Programmer's Guide* and the *Online API References*.

If the Learn model file is on the client's file system, the gateway stores the model temporarily before streaming it to the nodes. As Learn model files can be large (several Gigabytes), this requires either the temporary allocation of a large amount of gateway memory or the use of a temporary file. For details on controlling the use of memory versus temporary files, see the *TIBCO Patterns Online References*, the *TIBCO Patterns Programmer's Guide*, and the -G command-line option in the "Running the TIBCO Patterns Server" section of the *TIBCO Patterns Installation Guide*.

Query Builder and Query Builder Platform

The TIBCO Patterns Query Builder Platform is a Java based tool that eases the process of designing, testing, and administering TIBCO Patterns record matching queries. This tool eliminates or greatly reduces the need to write Java code to define and run record matching queries. The tool replaces Java code, which must be compiled and linked into an application, with an external configuration file that can be loaded at run time.

This section introduces the concepts related to TIBCO Patterns query builders and the query builder platform.

- [An Introduction to Query Builders](#)
- [An Introduction to the Query Builder Platform](#)
- [Integration with Other TIBCO Patterns Tools](#)

An Introduction to Query Builders

A TIBCO Patterns query performs a search for records matching a particular set of values. To search for "John Smith", you build a query and perform the search. To search for "Jack Jones", you must build a new query. Every new search requires a new query be built. A query builder takes in a set of values to be matched and outputs a TIBCO Patterns query to match those values.

The TIBCO Patterns Java API provides two abstract classes that define a standard interface for query builders.

- `ANetricsQueryBuilder` builds queries to match a flat record.
- `ANetricsCompoundQueryBuilder` builds queries to match a compound record. For information about compound records, see [Compound Records](#).

By implementing your query builder class as an extension of one of these classes, you can make your query builder usable in any application that is written to use one of these abstract classes. It is recommended that all applications doing record matching use one of these abstract classes to generate record matching queries.

The two abstract classes provided by the Java API provide methods that enable an implementation of a query to be adapted to new applications without code changes. Input and output data element names and table names can be remapped. The classes accept a number of input formats, taking care of the bookkeeping needed to deal with the different formats. You only need to define how to build the query, the rest of the functionality is provided by the abstract base class. This simplifies the writing of a new query builder.

An Introduction to the Query Builder Platform

The `ANetricsQueryBuilder` and `ANetricsCompoundQueryBuilder` classes provide a standard interface for query builders, making them portable across applications. They also simplify the process of writing a query builder by taking care of routine bookkeeping operations. However, you must still write the code, compile it, and link it into your application. The Query Builder Platform eliminates the need to write code. Instead, a configuration file is used to define the query. From the configuration file, the platform can either generate the code for you, or build queries directly.

Details about the format of the configuration file used to define a query can be found in the `QueryDef.xsd` schema file located in the `query_builder/schema` directory. The `customer1.xml` sample configuration file is provided in the `query_builder/samples` directory.

The Query Builder Platform provides an extension of each of the query builder abstract classes in the Java API; `GeneralQueryBuilder` extends `ANetricsQueryBuilder`, and `GeneralCompoundQueryBuilder` extends `ANetricsCompoundQueryBuilder`. `GeneralQueryBuilder` and `GeneralCompoundQueryBuilder` read a configuration file that completely defines how to build the query. Because `GeneralQueryBuilder` and `GeneralCompoundQueryBuilder` are extensions of the abstract classes, they can be plugged into any application designed to use one of the abstract classes as its query builder. The query used by the application is then defined from a configuration file read in at run time, rather than from compiled code that must be linked into the application.

The platform provides additional functionality. A tool is provided that allows query configurations to be verified and tested. A set of interfaces and abstract classes provide hooks that allow `GeneralCompoundQueryBuilder` and `GeneralQueryBuilder` to be integrated into almost any application.

Using the Query Builder Platform to Test a Query

The Query Builder Platform provides a command-line interface for testing a query definition. The two steps used to test a query definition are verifying that the configuration file is correct and verify that the query retrieves the desired records.

Verifying the configuration

This step verifies that the configuration file defining the query is valid and generates a valid TIBCO Patterns query. The command-line tool scans the configuration producing a log of all errors and questionable items found in the configuration. On request, it creates a copy of the configuration file with error messages added as annotations to the associated items in the configuration. This is often the best way to locate where the errors are in a large and complex configuration.

Verifying the query

Verifying that a configuration is valid does not verify that the query it defines returns the desired results. To do that you must run the query and examine the query results. The command-line interface allows you to run a query against a TIBCO Patterns server using data from CSV files or a TIBCO Patterns server.

A configuration file defines how the queries are run. This is a separate configuration file from the one that defines the query. In this file, you can specify the source data for generating the queries and a target TIBCO Patterns server, on which the queries are run.

The query source can be a set of CSV files, one for each table defined for the query. The rows in the files must be sorted by the PARENT key value. The records in the CSV files are merged into compound records. These compound records are used as the input to the defined query builder.

Alternatively, the query source can be a TIBCO Patterns server. Compound records are read from the server based on the table structure in the query definition.

By using the query run configuration file, you can remap the table and field names for the query source and the target server. You can also specify a cutoff score applied to all queries, and a maximum number of queries to run.

Details about the format of the configuration file used to define a query run can be found in the RunQuery.xsd schema file located in the query_builder/schema directory. The run_

customer1.xml sample query run configuration file is provided in the query_builder/samples directory.

Using the Query Builder Platform in an Application

Because of its flexibility in allowing queries to be changed quickly and easily, most applications using the Java API should use the query builder platform to implement queries. The production experience, or changes in requirements, often require that you modify the query. It is much easier to update a configuration file than to install new code.

The `GeneralCompoundQueryBuilder` and `GeneralQueryBuilder` classes are extensions of `ANetricsCompoundQueryBuilder` and `ANetricsQueryBuilder`. Therefore, they can be used in any Java application that expects an implementation of one of these classes. The `GeneralCompoundQueryBuilder` or `GeneralQueryBuilder` class must be initialized with the desired configuration, either in the constructor when created, or by calling one of the `setConfiguration` methods before being used.



Note: To use these classes, include the `TIB_tps_qbp.jar` file in the class path of the Java application.

Some applications must handle both compound record matching and single table record matching. In such cases, the application can use `GeneralCompoundQueryBuilder`. Standard and single table record matching is provided as one of the compound record matching types available in the query configuration.

To allow the `GeneralCompoundQueryBuilder` and `GeneralQueryBuilder` classes to fit seamlessly into almost any application, the following interfaces are defined by the query builder platform: `AQBPLoaderMgr`, `IQBPFileMgr`, and `AQBPLoaderMgr`.

Default implementations of these interfaces are available, which are automatically applied if the application does not supply its own. Therefore, it is not necessary to provide an implementation of any of these classes. An implementation is needed only if the default behavior is not suitable.

Interfaces of Query Builder Platform

Interface	Description
<code>AQBPLoaderMgr</code>	Gives the application control over the reporting of errors.

Interface	Description
	<p>The application can provide its own implementation of this abstract class to integrate the logging and error reporting of the query builder platform classes with the application's logging or the application can use the provided <code>QBPFiLeLogger</code> class and set options to control the reporting of errors. When using <code>QBPFiLeLogger</code> class, you can set options to perform the following actions:</p> <ul style="list-style-type: none"> • Throw exceptions for errors rather than log them. • Add or omit error annotations to the configuration file. • Control the types of log entries reported.
<code>IQBPFiLeMgr</code>	<p>Gives the application control over the reading of configuration files.</p> <p>This can include enforcing restrictions on which files can be read. It can also include allowing configurations to be read from non-standard locations, such as a DBMS or a web service.</p> <p>A basic implementation of this interface, <code>DfLtFiLeMgr</code>, is provided. It reads configuration files from a standard file system. By default, it imposes no restrictions, but it can be configured to restrict configuration files to a specific directory (and all subdirectories).</p>
<code>IQBPLOaderMgr</code>	<p>Gives the application control over the dynamic loading of classes.</p> <p>If the configuration file contains custom classes, these classes must be dynamically loaded. For security reasons, by default, classes are only loaded from the JVM class path. Applications might want to allow classes to be loaded from other locations (such as a directory for custom class files). Applications can define their own rules for the dynamic loading of classes by providing an appropriate implementation of this interface. As with the loading of configurations, this can also be used to allow custom classes to be loaded from non-standard sources, such as a DBMS or web-service.</p> <p>A basic implementation of this interface, <code>DfLtLOaderMgr</code>, is provided. By default the <code>DfLtLOaderMgr</code> loads classes only from the JVM class path. It can be set to load classes from a specific directory in addition to the JVM class path.</p>

Using the Query Builder Platform to Generate Code

You can also use the query builder platform to generate Java source that implements the query builder defined by the configuration file. The generated Java source extends and uses classes contained in the query builder platform JAR file (TIB_tps_qbp.jar). Therefore, you must include the TIB_tps_qbp.jar file in the class path of any application that uses the generated query builder classes. However, the full configuration file is not needed at run time.

- For generated single table record matching queries, a configuration file is not required at run time. The exception is when a custom query builder class requires a configuration file.
- For compound record matching queries a run time configuration file is required that defines the options for the compound query builder. The tool generates this configuration.

In most cases, no practical advantage is realized by using the generated code. It might run slightly faster, however in most cases the performance difference is not noticeable. Avoiding the need for loading configurations might be an advantage in certain applications where a file system or other data store is not available. The primary use for generating the Java source is where manual editing is necessary to provide special functionality. The general query structure can be generated from a configuration, and then customized manually.

Integration with Other TIBCO Patterns Tools

The TIBCO Patterns deduplication package provides two sample implementations of its QueryBuilder interface.

- GenericCompoundQueryBuilder wraps an ANetricsCompoundQueryBuilder object as an implementation of the deduplication framework QueryBuilder interface.
- GenericQueryBuilder wraps ANetricsQueryBuilder in a similar fashion.

The deduplication framework sample project shows how these classes can be used, in combination with the Query Builder Platform GeneralCompoundQueryBuilder class, to create a deduplication project that is driven from a query definition configuration file instead of from Java code.

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

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of Cloud Software Group, Inc.

TIBCO, the TIBCO logo, the TIBCO O logo, ActiveMatrix BusinessWorks, BusinessConnect, TIBCO Hawk, TIBCO Rendezvous, TIBCO Administrator, TIBCO BusinessEvents, TIBCO Designer, and TIBCO Runtime Agent are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (<https://www.cloud.com/legal>) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: <https://scripts.sil.org/OFL>

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at <https://www.cloud.com/legal>.

Copyright © 2010-2024. Cloud Software Group, Inc. All Rights Reserved.