Informatica® Identity Resolution
10.2 HotFix 1

Design Guide

Informatica Identity Resolution Design Guide
10.2 HotFix 1
September 2021

Publication Date: 2022-01-26

Table of Contents

Preface

The *IIR Designer Guide* provides information about the steps needed to design, define and load an Identity Resolution "System".

Learning About Informatica Identity Resolution

This section provides details of documentation available with the Informatica Identity Resolution product.

Introduction Guide

Introduces Identity Resolution and it's related terminology. It may be read by anyone with no prior knowledge of the product who requires a general overview of Identity Resolution.

Installation Guide

This manual is intended to be the first technical material a new user reads before installing the Identity Resolution software, regardless of the platform or environment.

Design Guide

This is a guide that describes the steps needed to design, define and load an Identity Resolution "System".

Developer Guide

This manual describes how to develop a custom search client application using the Identity Resolution API.

Operations Guide

This manual describes the operation of the run-time components of Identity Resolution, such as servers, search clients and other utilties.

Populations and Controls Guide

This manual describes SSA-Name3 populations and the controls they support. The latter are added to the Controls statement used within an IDX-Definition or Search-Definition section of the SDF.

Release Notes

The Release Notes contain information about what's new in this version of Identity Resolution. It is also summarizes any documentation updates as they are published.

What Do I Read If. . .

I am. . .

. . . a business manager

The INTRODUCTION to Identity Resolution will address questions such as "Why have we got Identity Resolution?", "What does Identity Resolution do"?

I am. . .

. . . installing the product?

Before attempting to install IIR you should read the INSTALLATION GUIDE to learn about the prerequisites and to help you plan the installation and implementation of the Identity Resolution.

I am. . .

...an Analyst or Application Programmer?

A high-level overview is provided specifically for Application Programmers in the INTRODUCTION to Identity Resolution.

When designing and developing the application programs, refer to the DEVELOPER GUIDE which describes a typical application process flow and API parameters. Working example programs that illustrate the calls to IIR in various languages are available under the `<IIR_client_installation>/samples` directory.

I am. . .

...designing and administering Systems?

The process of designing, defining and creating Systems is described in the DESIGN GUIDE. Administering the servers and utilities is described in the OPERATIONS manual.

Informatica Resources

Informatica provides you with a range of product resources through the Informatica Network and other online portals. Use the resources to get the most from your Informatica products and solutions and to learn from other Informatica users and subject matter experts.

Informatica Network

The Informatica Network is the gateway to many resources, including the Informatica Knowledge Base and Informatica Global Customer Support. To enter the Informatica Network, visit https://network.informatica.com.

As an Informatica Network member, you have the following options:

- Search the Knowledge Base for product resources.
- View product availability information.
- Create and review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to find product resources such as how-to articles, best practices, video tutorials, and answers to frequently asked questions.

To search the Knowledge Base, visit https://search.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

Use the Informatica Documentation Portal to explore an extensive library of documentation for current and recent product releases. To explore the Documentation Portal, visit https://docs.informatica.com.

If you have questions, comments, or ideas about the product documentation, contact the Informatica Documentation team at infa_documentation@informatica.com.

Informatica Product Availability Matrices

Product Availability Matrices (PAMs) indicate the versions of the operating systems, databases, and types of data sources and targets that a product release supports. You can browse the Informatica PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services and based on real-world experiences from hundreds of data management projects. Informatica Velocity represents the collective knowledge of Informatica consultants who work with organizations around the world to plan, develop, deploy, and maintain successful data management solutions.

You can find Informatica Velocity resources at http://velocity.informatica.com. If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that extend and enhance your Informatica implementations. Leverage any of the hundreds of solutions from Informatica developers and partners on the Marketplace to improve your productivity and speed up time to implementation on your projects. You can find the Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through the Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link:
https://www.informatica.com/services-and-training/customer-success-services/contact-us.html.

To find online support resources on the Informatica Network, visit https://network.informatica.com and select the eSupport option.

CHAPTER 1

Introduction

This chapter includes the following topics:

Overview

The process of setting up an IIR System is summarized below. It is assumed that the Installation process has already been performed so that an initialized IIR Rulebase already exists in the user's database.

The detail associated with these steps can be found in the following chapters.

Requirements Analysis

The first step is to define the search and matching requirements for the system. To begin, list the types of searches that will be required.

One IIR system can include multiple search types ("search definitions"). For example, a Customer Search system may require,

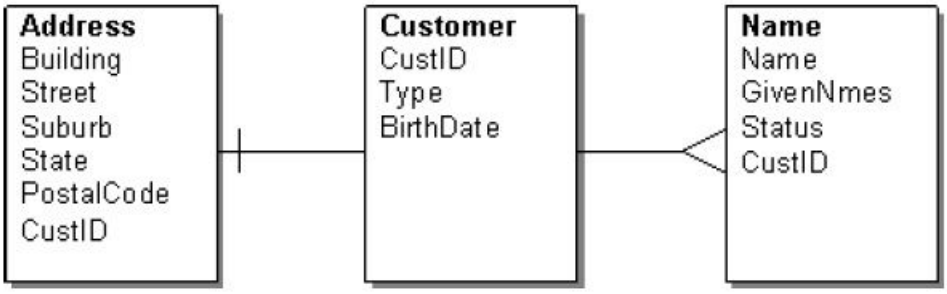
- Front-office Customer lookup
- Customer take-on duplicate checking
- An address search

Data Analysis

For each search type, decide what user data is needed for searching, matching and display and map the database tables and relationships that hold that data.

For example, the Customer Search system needs to search on customer name or address, requires name, address, date of birth and sex for matching, and in addition to the matching fields, requires Customer- ID, Customer-Type (individual or organization) and Name-Status (current, former or alias) for display.

This data is stored in User Source Tables as follows:



Create the IIR System Definition

Create the System Definition File (SDF) using either

- the GUI SDF Wizard, which is a quick and intuitive method for building simple SDFs, or
- the GUI System Editor, which is a full-featured, but low-level editor. You may start by cloning an existing system (a sample System is loaded during the Installation process), or
- a simple text editor, like Wordpad or vi.

The SDF describes the fields used for searching, matching and display and where that data is acquired from. It also nominates the SSA-NAME3 Population to be used for the search and matching rules.

For example, the SDF `User-Source-Table` section for the Name search requirements in the Customer Search example might look like this,

```
Section: User-Source-Tables
create_idt
                    name_idt
sourced_from
                    user.CUSTOMER.CUSTID,
                    user.CUSTOMER.TYPE,
                    user.CUSTOMER.BIRTHDATE,
                    user.NAME.NAME                                          $surname,
                    user.NAME.GIVENNMES                   $given_names,
                    user.NAME.STATUS,
                    user.ADDRESS.BUILDING $building,
                    user.ADDRESS.STREET$street,
                    user.ADDRESS.SUBURB,
                    user.ADDRESS.STATE,
                    user.ADDRESS.POSTALCODE
transform
                    concat ($given_names, $surname) NAME C(100) order 1
     concat ($building, $street) ADDRESS C(100) order 2
```

```
join_by
                user.CUSTOMER.CUSTID = user.NAME.CUSTID,
                user.CUSTOMER.CUSTID = user.ADDRESS.CUSTID
```

Select the SSA-NAME3 Population rules

Install a SSA-NAME3 Standard Population for use by IIR and select an appropriate population to provide the rules that define how the Key Building, Search Strategies and Matching Purposes operate on a particular population of data.

There is one Standard Population set for each country that Informatica Corporation supports. For more information on selecting the right population for your data, please see the SSA-NAME3 online documentation, or talk to Informatica's technical support.

Start the IIR Search and Rulebase Servers

From the IIR Console, start the Search Server and use the Online or Batch Search Clients to test the System's Search Definitions.

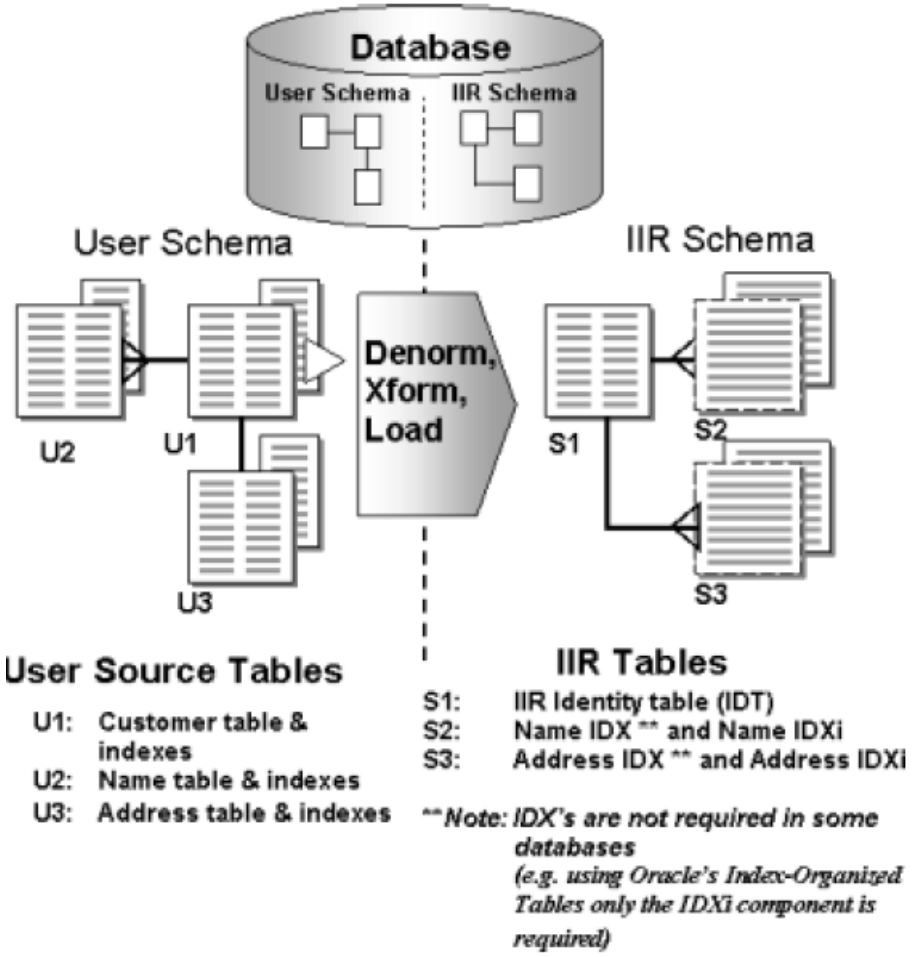
Create the IIR System

Use the IIR Console to create the new System. This will parse and load the SDF to the IIR Rulebase.

Load the IIR Identity Table and Index(es)

Use the IIR Console to load the IIR Identity Table "IDT" and Index(es) "IDX" for the System. This will extract data from the User Source Tables, denormalize, transform, compress and key it into the IIR Tables.

The following diagram illustrates on how to load the IIR Identity table:



Tune Searches

Adjust the Search Definitions as necessary to achieve optimal results. This can be done with the Console's System Editor.

For more information on tuning searches, see the Tuning a Search section of this guide.

CHAPTER 2

Defining a System

This chapter includes the following topics:

Overview

A System Definition File is created using a GUI tool or text editor as described in **Editing a System**. After the initial load, the definitions may be maintained with the System Editor.

Alternatively, you may clone an existing System using the Console and then tailor it to your requirements using the System Editor. This chapter details the System Definition from the context of coding an SDF file, but the same keywords and parameters are equally applicable to the System Editor.

The System Definition File (SDF) contains four sections:

- System
- User Source Tables
- Files
- Views

Each section is identified by a `Section:` keyword. Some sections may be empty if they are not required. For example, the `Files` section is not used when input data is read from database tables, known as User Source Tables (USTs).

The `System` section is used to define the System, ID-Tables, Loader-Definitions, Loader Jobs, Logical Files and Search-Definitions. The `User-Source-Table` section is used to define how to extract data from User tables to create the IIR Identity Table.

The `Files` and `Views` sections are used to define the layout of flat-input files, which are only used when data is not being read from the User's database tables. The `Views` section may also contain input and output views used by Relate and other utilities.

Syntax

This section describes the common syntax that can be used. Any variations will be detailed in the appropriate section:

- Each line is terminated by a newline;
- Each line has a maximum length of 255 bytes;
- Lines starting with an asterisk are treated as comments;
- All characters following two asterisks (**) on a line are treated as comments.

Quoted strings can be continued onto the next line by coding two adjacent string tokens. For example, the `COMMENT=` keyword is another way of defining a comment inside the definition files. If a `COMMENT` field is very long, it could be specified like this:

```
COMMENT="This is a very long comment that"
" continues on the next line. "
```

Some definitions will require the specification of a character such as a filler character. It will be denoted as `<char>` in the following text. The Definition Language permits a `<char>` to be specified in a number of ways:

c a printable character

"c" the character embedded in double quotes (")

numeric(dd) the decimal value `dd` of the character in the collating sequence

numeric(x'hh') the hexadecimal value (`hh`) of the character in the collating sequence

Numeric data in the `System` section be suffixed by an optional modifier to help specify large numbers.

k units of 1000 (one thousand)

m units of 1000000 (one million)

g units of 1000000000 (one billion)

k2 units of 1024 (base 2 numbers)

m2 units of 1048576

g2 units of 1073741824

The System Definition File contains multiple definitions, each identified by a heading. The statements that follow each definition heading take the form, `<field> = <value>`

Restrictions

This section provides information on the restrictions of Name Length and Embedded Spaces.

Name Length

System-Name is restricted to 31 bytes. All other Rulebase object names are limited to 32 bytes. The maximum length of Database object names such as tables and indexes are constrained by the host DBMS.

Embedded Spaces

Embedded spaces are not permitted in Rulebase object names or file names.

Files and Environment Variables

The Console Server interprets all files names specified in the SDF. Therefore they must be valid file names which are accessible from the machine where the Console Server is running.

IIR does not allow spaces within file names or paths.

Similarly, any environment variables used in the SDF are interpreted in the environment defined for the Console Server. The environment variables that are private to each client are defined when starting a Console Client and are passed to the Server where they override the variables in the Console's startup environment.

System Section

The major components of the `System` section are as follows:

- System-Definition. Defines global parameters and is specified only once.
- IDT-Definition. Defines how to build an Identity Table (IDT), including the columns selected from the source table.
- IDX-Definition. Defines rules for constructing an Identity Index (IDX). An IDX is used to create a fuzzy (or exact) index on column(s) in the IDT. An IDT may be indexed by many IDXs.
- Loader-Definition. Controls the IIR Table Loader.
- Job-Definition. Provides additional details required by the Table Loader.
- Logical-File-Definition. Provides additional details required by the Table Loader.
- Search-Definition. Defines a search strategy, including the selection of candidates (using an IDX) ,and the matching parameters.
- Multi-Search-Definition. Combines the results of several searches into one search.

System Definition

This object begins with the `SYSTEM-DEFINITION` keyword. It supports the following parameters:

NAME=

A character string that defines the name of the System and is a mandatory parameter. The System name is limited to a maximum of 31 bytes. Embedded spaces in the name are not permitted.

ID=

This is one or two digit alpha-numeric to be assigned to the System. Every System in a Rulebase must have a unique ID. This is a mandatory parameter.

COMMENT=

This is a text field that is used to describe the System's purpose.

DEFAULT-PATH=

IIR will create various files while processing some jobs. The PATH parameters can be used to specify where those files will be placed. IIR will use the path specified in the `DEFAULT-PATH` parameter. If `DEFAULT-PATH` has not been specified, the current directory will be used. It is valid to specify a value of "+" for the path.

This represents the value associated with the environment variable SSAWORKDIR. This is especially recommended when running an API program and/or system *remotely* , that is from a directory other than the SSAWORKDIR directory.

OPTIONS=

This is used to specify System wide defaults.

COMPRESS-METHOD(n) the compression method to use when storing Compressed-Key-Data. Method 0 (the default) will store records longer than 255 bytes as multiple adjacent IDX records. This improves performance as it takes advantage of the locality of reference in the DBMS' cache. Method 1 will truncate the IDX records if their size exceeds 255 bytes and fetch them from the IDT instead. This causes additional I/O when truncated IDX records are referenced.

VPD-SECURE marks the System as secure. IIR will insist that all search users provide VPD context information prior to starting a search. Refer to the *Virtual Private Databases* section in this guide for details.

DATABASE-OPTIONS=

This parameter is used to customize the behavior of IIR when loading tables and indexes.

These options give fine control over table size, placement and sorting and are particularly important when loading large amounts of data. The syntax is:

```
Type (Name, Database_Type, Text1 [, Text2]), ...
```

where

Type is one of the following: IDT, IDX, IDTBAD, IDTCP, IDTERR, IDTFD, IDTLOAD , IDTTMP, IDXTMP, IDXSORT, SEQUENCE, or IDTPART. A detailed description for each one appears below.

Name is the name of the IDT or IDX. A value of "default" may be specified to define defaults for all IDTs and IDXs. A specific name will override the default for a particular Type.

Database_Type] is either ORA for Oracle, UDB (for IBM UDB/DB2), MSQ for Microsoft SQL Server, or MVS (for DB2 running on z/OS). This allows the specification of options for all target database types, permitting the same SDF to be used for multiple target databases.

Text1 and Text2 a list of values whose format and content is Type dependent. All types require at least Text1 to be provided, and some types require Text2.

Type IDT / IDX

IIR creates database tables and indexes using CREATE TABLE and CREATE INDEX SQL statements.

This Type is used to specify where those tables and indexes will be placed and/or what attributes they will have.

This gives the local Database Administrator control over the extent sizes and location (Tablespace) of IIR tables and indexes.

Text1 is used to specify options for Table creation, while Text2 is used for Index creation options. They must start with table= and index= respectively.

The strings that follow these tokens are DBMS specific text which is appended to the CREATE TABLE and CREATE INDEX SQL statements. Text is appended immediately after the specification of the columns in the table (for CREATE TABLE) and immediately after the list of columns in the index (for CREATE INDEX). The text must be syntactically correct for the target DBMS, otherwise a run-time error will result when the table or index is created.

Oracle: The IDX is implemented as an Index Organized Table (IOT). The storage for an IOT is defined using the table= parameter. The index= parameter is ignored in this case.

Example 1:

```
DATABASE-OPTIONS=
IDT (default, ora, "table=storage(initial 10M next 1M)",
"index=storage(initial 10M next 1M)"),
IDX (default, ora, "table=storage(initial 10M next 1M)",
"index=storage(initial 10M next 1M)"
```

This example defines default extent sizes for both IDTs and IDXs stored on an Oracle database.

UDB: In this example, tables and indexes created for the IDT called `idt-39` are placed in `TABLESPACE USERSPACE1` and indexes on those tables have a `PCTFREE` value of zero. All IDX indexes have a `PCTFREE` of zero.

```
DATABASE-OPTIONS=
IDT (idt-39, udb, "table=IN USERSPACE1 INDEX IN USERSPACE1", "index=PCTFREE 0",
IDX (default, udb, "", "index=PCTFREE 0")
```

Type IDTBAD

This Type is used to specify the path for the file created by the DBMS load utility that contains rejected records.

Oracle: The default path is the current work directory.

UDB: The default is `/tmp` on the server machine. When using a `UDB LOAD` utility, this path is relative to the machine where the DBMS server is running.

```
DATABASE-OPTIONS=
           IDTBAD (IDT55, udb, "/tmp/rejects/"),
```

Type IDTCP

This Type is used to specify the Code Page of the data to be loaded. It is only used by MS SQL. Oracle and UDB ignore it. The parameter is passed to the bcp mass load utility with the `-C` switch. The Code Page effects the translation of `CHAR` and `VARCHAR` columns to the Server's code page and should only be specified if the SQL Server's code page is not the same as the client's code page.

The "client" in this situation is the machine where the IIR Table Loader runs on (which is normally the machine where the Console Server is running).

```
DATABASE-OPTIONS=
           IDTCP (default, msq, "437")
```

Type IDTERR

This `Type` is used to set the number of data errors that will be tolerated while loading the IDT. `Text1` specifies the maximum number of data errors that are allowed. The default is zero. The Table Loader will terminate abnormally if more than `Text1` data errors are encountered. Rejected records are written to a file with an extension of `.err` in the `IDTBAD` directory.

```
DATABASE-OPTIONS=
         IDTERR (IDT96, ora, "100"),
```

This allows up to 100 data errors when loading IDT96. If more than 100 errors occur, the Table Load will terminate with an error.

Type IDTFD

This `Type` is used to set the field delimiter character(s). This character is used to delimit fields in the IDT loader file. The default value is 0x01. The specified value must be in the range 1 to 255 and no field may contain the delimiter character.

```
DATABASE-OPTIONS=
           IDTFD (IDT96, ora, "255")
```

If it is not possible to select a field delimiter character that is not contained in any field value, use a fixed-format IDT load file (Loader-Definition, `OPTIONS=Fixed`).

Oracle: Two delimiter characters may be defined by specifying a value in the range 1 to 65535. The first delimiter character is defined as `value%256` and the second delimiter is `value/256`. A delimiter character value of 0x00 is not permitted. For example,`DATABASE-OPTIONS= IDTFD (IDT96, ora, "258")`

defines the first and second delimiter as 0x02 and 0x01 respectively.

Type IDTLOAD

This Type is used to specify the name and path of the database mass load utility. If this parameter has not been defined, IIR uses the utility specified by the `SSASQLLDR` environment variable. For example:

```
DATABASE-OPTIONS=
IDTLOAD (default, ora, "/opt/ora8i/product/8.1.6/bin/sqlldr"),
IDTLOAD (default, udb, "s:\sqllib\bin\db2")
```

Type IDTTMP / IDXTMP

This `Type` is used to control the placement of temporary loader files. By default, files are created in the `DEFAULT-PATH`. When large amounts of data are to be loaded, it is advantageous to place the files on different disks to avoid I/O contention problems.

`Text1` specifies the name of the directory where the file will be placed. For example:

```
DATABASE-OPTIONS=
        IDTTMP (IDT96, ora, "c:/tmp"),
        IDXTMP (nameidx, ora, "d:/tmp")
```

This places the data file for `IDT96` in `c:/tmp` and the data file for the IDX named `nameidx` in `d:/tmp`.

Type IDXSORT

This `Type` is used to control the sorting of the IDX data file. `Text1` has the following format:`MEM=nnn[M|k|G],THREADS=t,PATH1=path1,PATH2=path2`

where

`nnn` is the size of the sort buffer in megabytes (or kilobytes/gigabytes if k/G is specified). The default size is 64MB. Do not make the memory buffer so large as to cause swapping, as this will negate the benefit of a fast memory based sort.

`t` is the number of threads to use for the sort process. It defaults to the number of CPUs on the machine where the Table Loader runs.

`path1` is the directory where any temporary sort work-files will be created. Very large loads may require two sort work-files. These should be placed on different disks if possible.

`path2` is the directory where the second sort work-file will be created (if necessary).

The sort path used is a hierarchy, depending on which parameters have been specified. The parameters in order of highest to lowest precedence are `IDXSORT`, `SORT-WORK-PATH` and `DEFAULT-PATH`.

For example:

```
DATABASE-OPTIONS=
         IDXSORT (default, ora, "mem=256M,threads=2"),
         IDXSORT (nameidx, ora, "mem=512M,path1=/dev1,path2=/dev2")
```

Type IDTPART

This `Type` is used to specify storage parameters for partitioned UDB databases. This is necessary because each unique index of a table in a partitioned tablespace must contain all distribution columns of the table.

We can specify a partitioning key for the IDT with an `IDT` database option:

```
DATABASE-OPTIONS=
          IDT(IDT-00, udb, "table=in TESTSPACE partitioning key(EmpNum) using
hashing"),
```

Now each unique index will require the partitioning key as an additional key. This can be implemented with an `IDTPART`, which provides a list of the partitioning keys.

For example:

```
DATABASE-OPTIONS= IDTPART(IDT-00, udb, "EmpNum")
```

Type SEQUENCE

Use the `SEQUENCE` type to generate record IDs in a database sequence for a single identity table or all identity tables.

Use the following format:

```
DATABASE-OPTIONS=SEQUENCE(default|<IDT name>, <Database type>,
["start=<N>","increment=<N>"])
```

The format uses the following attributes:

Default or IDT name

Specifies whether to enable database sequence in all identity tables or in the specified identity table. The value `default` enables database sequence in all identity tables.

Database type

Indicates the type of database you use. Use one of the following values:

- `ora` for Oracle.
- `udb` for IBM Db2.
- `msq` for Microsoft SQL Server.

N

Optional. Indicates the starting number and the incremental value for the database sequence. Default is 1.

The following example sets the database sequence for a specific identity table in Oracle database:

```
DATABASE-OPTIONS= SEQUENCE(IDT38, ora, " ", "increment=3")
```

Identity Table Definition

This begins with the `IDT-DEFINITION` keyword and defines an IIR Identity Table. The fields are as follows:

Field	Descriptions
NAME=	A character string that specifies the name of the IDT. This is a mandatory parameter
DENORMALIZED-VIEW=	A character string that specifies the name of the `View-Definition` that defines the layout of the denormalized data for Flattening. Refer to *Flattening IDTs* section in this guide for details. This is an optional parameter.

Field	Descriptions
FLATTEN-KEYS=	A character string that defines the IDT columns used to group denormalized rows during the Flattening process. A maximum of 36 columns may be specified. Refer to Flattening IDTs section in this guide for details. This is an optional parameter.
OPTIONS=	This is used to specify various options to control the data stored in the IDT: `FLAT-KEEP-BLANKS` a flattening option that maintains positionality of columns in the flattened record. Refer to the *Flattening IDTs* section for details. `FLAT-REMOVE-IDENTICAL` a flattening option that removes identical values from a flattened row. Refer to the *Flattening IDTs* section for details.

Identity Index Definition

This begins with the `IDX-DEFINITION` keyword and defines an IIR Identity Index. The fields are as follows:

Field	Description
NAME=	A character string that specifies the name of the IDX. The maximum length is 32 bytes. This is a mandatory parameter.
COMMENT=	A free-form description of this IDX
ID=	A two-letter character string used to generate the names of the actual database table that represents the IDX. Each IDX must have a unique table name (generated from the target database's userid (schema), System Qualifier, and ID). Refer to the *OPERATIONS guide, Database Object Names* section for details. This is a mandatory parameter.
IDT-NAME=	The name of the IDT Table that this IDX belongs to (as defined in the `File-Definition`or `User-Source-Table` sections). This is a mandatory parameter. If you intend to use the AnswerSet feature with this IDT, then this name may need to be kept short so that the name of the AnswerSet table does not exceed maximum length supported by the host DBMS. Refer to the *DEVELOPER GUIDE, AnswerSet* section for details.
AUTO-ID-FIELD=	This field is not required if loading data from a User Source Table. The name of a field defined in the Files section that contains a unique record label referred to as the Record Source If no such field exists in the IDT, IIR can generate one (see `AUTO-ID` and `AUTO-ID-NAME` parameters). If IIR is being asked to generate an Id, the user can choose the name of the `AUTO-ID-FIELD`, however that name must be defined as a field in the Files section (if using a transform clause, this happens automatically).
KEY-LOGIC=	This parameter describes the key-logic to be used to `generate keys` for the IDX. It may differ from the Search-Logic used to search the IDX. Refer to the Key Logic section for details. This is a mandatory parameter.

Field	Description
PARTITION=(field [, length , offset [, null-partition-value]]),. . .	This option instructs Identify Resolution to build a concatenated key from the Key-Field, which is defined by `KEY-LOGIC=` and up to five fields or sub-fields taken from the record. For large files, the key might not be selective enough because it retrieves too many records. The field, length, and offset represent the field name, number of bytes to concatenate to the key and offset into the field (starting from 0) respectively. The length and offset are optional. If omitted, the entire field is used. Use `null-partition-value`, which defaults to spaces, to specify a value for an empty field. If you specify the `NO-NULL-PARTITION` option, it can ignore the records that contain a null partition value. For the `NO-NULL-PARTITION` option to work, all the values of the specified fields must be null. **Note:** Partition values are case sensitive. Data might be mono-cased using a transform when loading the data. Be sure to use the correct case when entering search data. For non-displayable data, the value might be specified in hexadecimal notation in the form: `hex(hexadecimal value).`
OPTIONS=	Specifies options to control the keys and data stored in the IDX. The supported options are as follows: - `ALT`. Stores alternate keys in the IDX. This option is on by default. When disabled (`--Alt`) only the first key from the key-stack is stored in the IDX. - `FULL-KEY-DATA`. Stores all IDT fields in the IDX. This is the default. The data is stored in uncompressed form unless you specify the `Compress-Key-Data` option. - `FULL-KEY-DATA`. Stores all IDT fields in the IDX. This is the default. The data is stored in uncompressed form unless you specify the `Compress-Key-Data` option. - `COMPRESS-KEY-DATA(n)`. Stores all IDT fields in compressed form using a fixed record length of n bytes. n can not exceed (m - PartitionLength - KeyLength - 4) where m=255 for Oracle and m=250 for UDB. Selecting the appropriate value for n is discussed in the Compressed Key Data section. - `COMPRESS-METHOD(n)`. Overrides the default set by the System Options that specify the compression method to use when storing Compressed-Key-Data. Method 0 (the default) will store records longer than 255 bytes as multiple adjacent IDX records. This improves performance as it takes advantage of the locality of reference in the DBMS' cache. - `Method 1`. Truncates the IDX records if their size exceeds 255 bytes and fetch them from the IDT instead. This causes additional I/O when truncated IDX records are referenced. - `LITE`. Creates a Lite Index. See the Key Logic section for more information. - `NO-NULL-FIELD`. Does not store IDX entries for rows that contain a Null-Field. A Null-Field is defined in the Key-Logic section. - `NO-NULL-KEY` . Does not store Null-Keys in the IDX. A Null-Key is defined in the Key-Logic section. - `NO-NULL-PARTITION`. Does not store keys in the IDX that contains `null-partition-value`, as defined by the `PARTITION=`keyword.

Loader Definition

This begins with the LOADER-DEFINITION keyword. The fields are as follows:

Field	Description
NAME=	A character string that identifies a Loader-Definition. It is used when scheduling a Table Loader job. This is a mandatory parameter. The name must not match any Search-Definitions nor Multi-Search- Definition names in the same System.
COMMENT=	Free-form description of this Loader-Definition.
JOB-LIST=	A list of Job-Definition names that are to be executed when this Loader-Definition is scheduled to run. Up to 32 jobs may be listed. This is a mandatory parameter.
DEFAULT-PATH=	IIR will create various files while processing some jobs. The PATH parameters can be used to specify where those files will be placed. This parameter overrides the value in the System Definition. IIR will use the path specified in the DEFAULT-PATH parameter. If DEFAULT-PATH has not been specified, the current directory will be used. It is valid to specify a value of "+" for the path. This represents the value associated with the environment variable *SSAWORKDIR* . This is especially recommended when running an API program and/or system remotely, that is from a directory other than the SSAWORKDIR directory.
SORT-WORK1-PATH=	IIR will create sort work files during the Load. This parameter controls the placement of these files, and overrides the DEFAULT-PATH parameter in the Loader-Definition.

Field	Description
SORT-WORK2-PATH=	IIR will create sort work files during the load. This parameter controls the placement of these files and overrides the DEFAULT-PATH parameter in the Loader-Definition.
OPTIONS=	This is used to nominate various options for the Loader. APPEND append records to the IDT and IDXs. If omitted, the loader will assume this is an initial load and create the IDT and IDXs. APPEND must not be used with synchronized source input, as IDTs created from synchronized data sources must be loaded with a single execution of the Table Loader. KEEP-TEMP keep temporary files. If omitted, temporary files are deleted when no longer necessary. KEEP-LOG keep Loader log files. If omitted, log files are deleted when the Loader completes successfully. Loader log files are copied to the Table Loader's log file before being deleted. FIXED Create the IDT load file using fixed-length records instead of variable length delimited records. This option is necessary if the data contains any columns with binary values that are not permitted in a delimited file, that is, CR/LF, Ctrl-Z and/or the field delimiter character. Unicode data should always be loaded using fixed-length records. **Note:** The MS SQL Server interface always generates fixed length records, so this parameter does not need to be specified. CONVENTIONAL-PATH Oracle: instructs the Loader to invoke Oracle's SQL*Loader without the DIRECT-PATH option. This degrades performance but enables SQL*Loader to load tables over a network when the version of SQL*Loader does not match the version of the database instance. **Note:** DIRECT-PATH loads will specify the UNRECOVERABLE option. This means that all IDTs and IDXs should be backed up after being loaded as they can not be restored during media recovery. UDB/DB2: Conventional-Path performs an Import operation. If not specified, the Loader runs the UDB Load utility. MS-SQL Server: has no effect. GENERIC-LOAD instructs the Loader to load records using the DBMS' native insert and commit statements. This is provided as a "catch all" solution for those DBMSs that do not have a high-speed massload utility.

Job Definition

This begins with the JOB-DEFINITION keyword. The fields are as follows:

Field	Description
NAME=	This is a character string that specifies the name of the job. It is a mandatory parameter.
COMMENT=	This is a text field that is used to describe the Job's purpose.
IDX=	This is a character string that specifies the name of an IDX to be loaded by this job. The IDX name indirectly implies the name of the IDT to be loaded (as an IDX can only belong to one IDT). If the Load-All-Indexes option has been specified, all IDXs defined for the IDT will be loaded. Either this parameter or IDT= must be specified, but not both.

Field	Description
IDX-LIST=	This is a comma-separated list of IDX names used in conjunction with the `Load-All-Indexes` option to limit the number of IDXs to be loaded. Normally `Load-All-Indexes` means that all IDXs that have been defined are to be loaded. The list must be enclosed in double quotes. For example, `IDX-LIST= "zip,addr"`
IDT=	This is a character string that specifies the name of the IDT to be loaded by this job. This parameter automatically enables the `IDT-Only` option. It is used for the situation when an IDT has no IDXs defined, or when you wish to load the IDT only. This parameter is mutually exclusive with `IDX=`.
FILE=	A mandatory parameter used to define the name of the Logical-File entity that describes the source of input data for the loader job.
INPUT-SELECT=n INPUT-SELECT=Count(n)|Skip(n)|Sample(n)	This parameter is used to define input processing options. When specified in the first form above, the number `n` is treated as the number of records to be read from the input. An equivalent method of specifying this is `Count(n)`. The value n must be a positive non-zero number. You may skip some records before processing begins by specifying `Skip(n)`. You may also process every n^{th} record by specifying `Sample(n)`.
INPUT-HEADER=	Describes the number of bytes to ignore at the start of an input file. This is useful for some types of files that contain a fixed length header before the actual data records. This is only relevant when reading input from a flat file.
OPTIONS=	This is used to nominate various options for the Job. `AUTO-ID` generate a unique `Record Source Id` in the `AUTO-ID-FIELD`. `LOAD-ALL-INDEXES` instructs the loader to create all IDXs defined for the IDT in one execution of the Loader. This is the most efficient way to build the IDXs. `RE-INDEX` is used to create a new IDX for an existing IDT (created by a previous Loader job). Re-Index instructs the loader to read records from the IDT instead of from the normal input source such as User Source Tables or flat files. **Note:** The Update Synchronizer must not update the IDT while the Loader builds the new index. `IDT-ONLY` load IDT only (no IDXs). This is useful for loading "intermediate IDTs" that do not have any IDXs defined. `NO-IDT` load IDXs only (no IDT). This is useful when the IDT has already been loaded with the `IDT-ONLY` option.

Logical File Definition

This begins with the `LOGICAL-FILE-DEFINITION` keyword. The fields are as follows:

Field	Description
NAME=	This is a character string that identifies the Logical-File. A Job object refers to a logical-file with its `FILE=` parameter. This is a mandatory parameter.
COMMENT=	This is a text field that is used to describe the Logical File's purpose.
PHYSICAL-FILE=	A character string that specifies the file name containing the input data. When reading input from a flat file, this will specify the full filename including the path. When reading input from User Source Tables it specifies the name of the IDT defined in the `CREATE_IDT` or `DEFINE_SOURCE` clause of the `User-Source-Table` section. The name should be enclosed in double-quotes. This is a mandatory parameter. See the *Reading Flat File Input from a Named Pipe* section for information on reading file input from a Named Pipe.

FORMAT={SQL|Text|Binary|XML|Delimited}	Describes the format of the input file. When reading from a User Source Table, specify a format of SQL. Otherwise, when reading from a flat-file, the following options may be used:
	`Text` files contain records terminated by a newline.
	`Binary` files do not contain line terminators. Records must be fixed in length and match the size (and layout) of the input View.
	`XML` files contain XML messages. This format is used when loading a flat-file created by the Siebel Connector.
	`Delimited` files contain variable length records and fields which are delimited. By default, records are separated by a newline, fields are separated by a comma (,) and the field delimiter is a double-quote ("). You may change the defaults by defining
	`FORMAT= Delimited, Record-Separator(<char>), Field-Separator(<char>), Field-Delimiter(<char>)`
	All three values are always used when the Delimited format is processed. There is no way of specifying that a particular delimiter should not be used. However, you may specify a value that does not appear in your data such as a non-ASCII value. For example, if a field delimiter is not used, the following could be specified:
	`Field-Delimiter(Numeric(255)) Record-Separator=<char> Field-Separator=<char> Field-Delimiter=<char>`
	These parameters are usually specified as sub-fields on the `FORMAT` definition. However, for convenience, they may be defined as separate fields.

Field	Description
VIEW=View	The name of the Database View Definition to be used when reading the flat input file. It must not be specified if data will be read from User Source Tables. The View is used to translate the contents of the input file into the layout to be stored in the IDT (specified by the `IDT-NAME=` parameter of the IDX-Definition).
XSLT=	In the case of an XML format input file, an XSLT stylesheet may be specified. The stylesheet will be used to transform an XML format input file into one that matches the IDT.
AUTO-ID-NAME=	This parameter is used when IIR has been requested to generate a `Record Source ID` field (see the `AUTO-ID-FIELD=` and `Auto-Id` parameters). The generated ID is composed of a text string concatenated to a base-36 sequence number. The value for the text string portion is specified by using the `AUTO-ID-NAME=` parameter. It is limited to 32 bytes. The sequence number is generated by IIR. The resulting ID field is stored on the IDT in the field defined by the `AUTO-ID-FIELD` parameter. This field must be defined in the Files section. We recommend defining an ID field with attributes F,10. This leaves ample room for an `Auto-Id-Name` and several characters for the sequence number. Since the latter is a base 36 number, it allows for 1.6 million records in 4 characters, 60 million using 5 characters, or up to 3.6 Gig with a 6 character sequence number. The length attribute of the ID field is not limited to 10 bytes. It may be increased when you have a large number of records and/or a long `Auto-Id-Name` prefix.

Search Definition

A `SEARCH-DEFINITION` is used to define parameters for a search executed by the Identity Resolution. It begins with the `SEARCH-DEFINITION` keyword.

Field	Description
NAME=	This is a character string that identifies the Search-Definition. This is a mandatory parameter. The name must not match any Loader-Definition nor Multi-Search Definition names in the same System.
IDX=	This is a character string that identifies the IDT to be searched. This is a mandatory parameter.
COMMENT=	This is a text field that is used to describe the Search's purpose.
SORT-WORK1-PATH=	IIR may create sort work files when sorting a large result set. This parameter controls the placement of these files and overrides the value in the System-Definition.
SORT-WORK2-PATH=	IIR may create sort work files when sorting a large result set. This parameter controls the placement of these files and overrides the value in the System-Definition.

Field	Description
FILTER=	An optional string containing an SQL expression used to remove some candidates from the Candidate Set. Refer to the *SQL Filters* section in this guide for information about when and how to use filters.
SEARCH-LOGIC=	This parameter describes the Search-Logic to be used to generate search ranges to find candidate records from the IDT. It may differ from the Key-Logic used to generate keys for the IDT (as defined in the IDXDefinition). Refer to the *Key Logic* section for details. This is a mandatory parameter.
SCORE-LOGIC=	This parameter describes the normal matching logic used to refine the set of candidate records found by the Search-Logic. Refer to the *Score-Logic* section for details. This is a mandatory parameter.
PRE-SCORE-LOGIC=	This optional parameter describes the lightweight matching logic used to refine the set of candidate records found by the Search-Logic. Refer to the *Score-Logic* section for details.
SORT=	A comma-separated list of keywords used to control the sort order of records returned by the search. Multiple sort keys are permitted. The keys will be used in the order of definition. If not specified, the records will be sorted using a primary descending sort on Score, and a secondary ascending sort using the whole record. `Memory(recs)` The maximum number of records to sort in memory. If the set contains more than recs records, the sort will use temporary work files on disk. The default is 1000 records. `Field(fieldname)` The name of the field to be used as the sort key. In addition to IDT field names, you may specify the following values: `sort_on_score` will sort on the Score `sort_on_record` will sort on the whole record. `Type(type)` The type of the sort for the previously defined key. Valid types are: 1. A Ascending 2. D Descending `Format(format)` The format of the sort key. Valid formats are: `FORMAT_TEXT` text data `FORMAT_SHORT` native signed short `FORMAT_USHORT` native unsigned short `FORMAT_LONG` native long `FORMAT_ULONG` native unsigned long `FORMAT_FLOAT` native float `FORMAT_DOUBLE` native long float `FORMAT_NN_SHORT` non-native short `FORMAT_NN_USHORT` non-native unsigned short `FORMAT_NN_LONG`non-native long `FORMAT_NN_ULONG`non-native unsigned long

Field	Description
CANDIDATE-SET-SIZE-LIMIT=n	Informs the Identity Resolution to optimize the matching process by eliminating any duplicate candidates that have already been processed. n specifies the maximum number of unique entries in the list. The default limit is 10000 records. A value of 0 disables this processing. If there are more than n unique candidates, only the first n duplicates are removed. Any candidates that do not fit into the list may be processed several times, and if accepted by matching, added to the final set several times. The TRUNCATE-SET option will terminate the search for candidates once the list becomes full. It is used to prevent very wide searches. However, if a search is terminated prematurely there is no guarantee that any of the candidates will be accepted or the best candidates have been found.
OPTIONS=	A comma-separated list of keywords used to control various search options: UNIQUE-KEYS specifies that no duplicate sort keys will be returned. Sort keys are defined with the SORT= keyword. SEARCH-NULL-PARTITIONany search for a record containing a Null-Partition value will search all other partitions. Any search for a record with a non-null partition value will search the null-partition as well. **Note:** The entire partition value must be null for this to work SEQUENCES specifies that detailed matching information is to be saved for each record in the search set. This information can be retrieved using **ids_search_get_complete**. TRUNCATE-SET modifies the behavior of CANDIDATE-SET-SIZE-LIMIT. Searches normally continue until all candidates have been considered. Truncate-Set will terminate the selection of candidates once the candidate set is full, thereby limiting the number of candidates that will be considered. Response code 1 is returned to the caller of **ids_search_start** or **ids_search_dedupe_start**when the set is truncated. HIDDEN prevents this search from being listed by the Search Clients if the search is not designed to be used independently (that is it should only be used as part of a Multi-Search) IGNORE-NOTCH-OVERRIDE Ignore any adjustments made to the match levels on an API call which requests a particular Match-Tolerance. The tolerance is honored but the adjustments are ignored. FIRST stop on first accepted match. Used for specialized applications where any acceptable match should terminate the search process. FILTER-APPEND Append dynamic filter to static filter defined in this Search-Definition. Refer to the SQL Filters section of this guide for details. FILTER-REPLACE Allow dynamic filters to replace the static filter defined in this Search-Definition. Refer to the SQL Filters section in this guide for details. UNMATCHED-STATS Deprecated. Exists only for backward compatibility.

Simple Search is a type of search where multiple entities of varying types can be searched using a single consolidated index. The index and search definitions must use a population that supports the Generic_Field. Simple search uses all columns in the index as possible search and match fields. Search definition of a Simple Search requires that the columns be combined in the SEARCH/SCORE logic using COMBINE=<field name>:DELIM-NONE. See the Populations and Controls Guide for further details on COMBINE option.

Cluster Definition

A `CLUSTER-DEFINITION` is used to define parameters for a clustering process. Refer to the Static Clustering section in this guide for details.

Field	Description
NAME=	This is a character string that identifies the Cluster-Definition. This is a mandatory parameter.
COMMENT=	This is a text field that is used to describe the Clustering's purpose.
SEARCH=	This is a character string that identifies the "host" Search that this Cluster-Definition belongs to. The clustering process will use the Search-Logic defined in the referred Search-Definition. Each Cluster- Definition can belong to a single Search and each Search can contain only one Cluster-Definition.

Multi-Search Definition

Use the `MULTI-SEARCH-DEFINITION` section to define parameters for a cascade of searches that Identity Resolution runs.

The following excerpt is a sample multi-search definition:

```
multi-search-definition
*======================
NAME=                          multi_svoc_person
IDT-NAME=                      idt_svoc3
SEARCH-LIST=                   search_svoc_person,
                               search_svoc_ssn
OPTIONS=                       Full-Search
```

Begin the definition with the `MULTI-SEARCH-DEFINITION` keyword, and use the following parameters:

NAME=

Required. Unique identifier for the multi-search definition. The name must not match any loader definition or search definition names in a system.

COMMENT=

Brief description about the multi-search.

SEARCH-LIST=

List of searches to run. Separate the searches by commas, and enclose the whole string in double quotes. Run all the searches against the same identity table. You can specify a maximum of 16 searches.

When you use the statistical output view field `IDS-MS-SEARCH-NUM`, the returned result refers to the search names in the list. For example, the value 1 refers to the first search in the list and value 2 refers to the second search in the list.

IDT-NAME=

Required. Identifier for the identity table against which you want to perform the multi-search.

IDL-NAME=

Optional. Identifier for the link table that stores the search results.

SOURCE-DEDUP-MAX=

Optional. Number of records to cache. A multi-search caches the records that a search processes and uses the results when another search in the search list retrieves them as candidates. This option helps only if all the searches use the same `Score-Logic`. The default value of 0 disables this option.

DEDUP-PROGRESS=

Optional. Maximum number of identity table records to process to find duplicate records before returning the control to a client process. The DupFinder process treats each identity record as a search record instead of reading search records provided by a client. The search process does not return control to the client until it finds a duplicate record. The process might take a long time if you have few duplicate records in the identity table. The client uses the time interval to write progress messages. The default value of 0 disables this option.

OPTIONS=

A comma-separated list of keywords that control the multi-search. Use the following options:

- FULL-SEARCH. Processes all the searches in the list.
- LINK-PK. Specifies that the link table contains primary key columns in addition to the normal data.
- LINK-HARD-LIMIT. Specifies that the link table does not contain links for the rejected candidates.
- LINK-SELF. Specifies that the link table contains links for the rows that found themselves.
- CONVENTIONAL-PATH. Specifies that the link table uses the Conventional-Path option to load.

User-Job-Definition

A User-Job-Definition is the SDF equivalent of a Console Job. Prior to version 2.6, Jobs could only be defined using the Console's **Jobs** > **Edit** > **New Job** dialog. It is envisioned that customers will continue to use the GUI to define Jobs. However, in order to facilitate the transfer of pre-defined jobs between Dev, Test, QA and Prod environments a new mechanism was needed to export the definition from the Rulebase into an SDF, and the SDF was enhanced by adding syntax for defining User-Jobs.

Jobs are exported from the Rulebase into SDF format using **System** > **Export** > **SDF**. The parameters associated with a Job Step mirror the parameter names in the equivalent GUI dialog used to create it. The easiest way to get started is to define a Job using the Console and export it to an SDF.

A `User-Job-Definition` contains two parameters:

Field	Description
NAME=	Defines the name of the User-Job. All subordinate User-Step-Definitions quote this name to associate themselves with the User-Job-Definition.
COMMENT=	This is an optional text field that is used to describe User-Job's purpose.

User-Step-Definition

Each User-Job-Definition is associated with one or more User-Step-Definitions. They are equivalent to Job Steps added with the `Add Step` button in the Console and contain the following parameters:

Field	Description
COMMENT=	This is a text field that is used to describe step's purpose.
JOB=	This is the name of the User-Job-Definition that this step belongs to.
NUMBER=	This field is used to order steps within a User-Job-Definition. `NUMBER` is a printable numeric value starting from 0. That is, the first step has `NUMBER = 0`. There must be no gaps in the numbering of steps.
NAME=	This is the type of Job step. A list of valid types is visible in the Console dialog when you click `New Step`. Names containing spaces should be enclosed in quotes (").
TYPE=	This is the type of step. A list of valid job types and their associated parameters can be generated by running, `%SSABIN%\pdf -diir reportFileName`or `$SSABIN/pdf -diir reportFileName`

PARAMETERS =	This is a list of parameters and values required by the step. A list of valid job types and their associated parameters can be generated by running, `%SSABIN%\pdf -diir reportFileName` or `$SSABIN/pdf -diir reportFileName`
	For example:
	`User-Job-Definition`
	`*==================`
	`COMMENT= "Load data and run a relate"`
	`NAME= Job01`
	`*`
	`User-Step-Definition`
	`*==================`
	`COMMENT= "Load IDT/IDX"`
	`JOB= Job01`
	`NUMBER= 0`
	`NAME= "Load ID Table"`
	`TYPE= "Load ID Table"`
	`PARAMETERS= ("Loader Definition","Table-1")`
	`*`
	`User-Step-Definition`
	`*==================`
	`COMMENT= "Run Relate (SEARCH-1)"`
	`JOB= Job01`
	`NUMBER= 1`
	`NAME= Relate`
	`TYPE= Relate`
	`PARAMETERS= ("Input File","data/relx03.in"),`
	`("Output File","relx03.out"),`
	`("Search Definition","SEARCH-1"),`
	`("Output Format",4),`
	`("Input View",dATAIN21),`
	`("Append New Line",true),`
	`("Trim Trailing Blanks",true),`
	`("Binary Input",false),`
	`("Output View","(none)")`

Key-Logic / Search-Logic

`KEY-LOGIC` is defined in the IDX-Definition to describe how to build keys to be stored in the IDX. Search-Logic is defined in the Search-Definition to specify how to build search ranges to read the IDX.

An IDX normally contains multiple fuzzy keys for each IDT record. Each IDX contains compressed display and matching data. This is known as a heavy index. IIR can also create a Lite Index . This is a native DBMS index containing exact values from a single field in the IDT record. A Lite index does not contain any redundant display or matching data. Searches utilizing a Lite Index will acquire display and matching data by reading the IDT record.

Syntax

Key-logic facilities are provided by a SSA-NAME3 Standard or Custom Population. The syntax is as follows:

```
KEY|SEARCH-LOGIC = SSA,
                          System (system),
                          Population (population),
                          Controls (controls),
                          Field(field_list),
                          Null-Field(null-field-value),
                          Null-Key(null-key-value)
```

where

`system` is the System Name of the Population Rules DLL. The system is a sub-directory of the population rules directory. It has a maximum length of 32 bytes. The location of the population rules directory is defined by the environment variable *SSAPR* . The System Name is case sensitive and it should be same as the sub-directory name.

`population` defines the name of the population DLL in the System directory (maximum length of 32 bytes).

Note: Population Name is case sensitive and it should be exactly similar to the one defined in the licence file.

`controls` defines the Controls to be used. Refer to the *STANDARD POPULATIONS* guide for a detailed description of these fields.

`field_list` is a comma-separated list of IDT fields to be used to generate keys or ranges. If more than one field is provided, the field_list must be enclosed in quotes (").

`null_field_value` An optional parameter that defines the null value for the Field. Records with a null value in their key field can be ignored by using the NO-NULL-FIELD option to prevent them from being indexed in the IDX. The default value is spaces (blanks).

`null_key_value` An optional parameter that defines the value of a null key. Records containing fields that generate a null key can be ignored by using the `NO-NULL-KEY` option to prevent the key from being stored in the IDX. The default value is `K$$$$$$$`.

Controls

A Key-Logic in an IDX-Definition controls the generation of keys. Therefore the Controls should specify the `KEY_LEVEL`. For example,

```
KEY-LOGIC=SSA,
          System(default), Population(test),
          Controls("FIELD=Person_Name KEY_LEVEL=Standard"),
          Field(LastName)
```

A Search-Logic in a Search-Definition controls the generation of search ranges. Therefore the Controls should specify a `SEARCH_LEVEL`. For example,

```
SEARCH-LOGIC=SSA,
                 System(default), Population(test),
                 Controls("FIELD=Person_Name SEARCH_LEVEL=Typical"),
                 Field(LastName)
```

Repeating Key Fields

The Field parameter of the Key- or Search-Logic can be used to generate keys or ranges for multiple fields (of the same type). This is accomplished by specifying a list of fields. For example:

```
KEY-LOGIC=SSA,
             System(default), Population(test),
             Controls("FIELD=Person_Name KEY_LEVEL=Standard"),
             Field("InsuredName,ClaimantName,PayeeName")
```

will generate keys for all three name fields and store them in the IDX.

When indexing a Group of repeating fields (or flattened fields), you must list each individual field name. For example, the following source definition, `test.person.name Name [5] C(20)`

generates a Group in the File-Definition of this IDT, which is subsequently expanded to the following list of field names:

```
FIELD=Name, C, 20
FIELD=Name_2, C, 20
FIELD=Name_3, C, 20
FIELD=Name_4, C, 20
FIELD=Name_5, C, 20
```

A Key-Logic that indexes all occurrences must list each individual field:

```
KEY-LOGIC=SSA,
          System(default), Population(test),
          Controls("FIELD=Person_Name KEY_LEVEL=Standard"),
          Field("Name,Name_2,Name_3,Name_4,Name_5")
```

A shorthand method of specifying repeating fields exists and takes the form of:`field [ * | {x | y-z }, ... ]`

Some examples of this notation are (assuming a maximum of 5 occurrences):

```
Name[1-5] = Name,Name_2,Name_3,Name_4,Name_5
Name[2,3] = Name[2-3] = Name_2,Name_3
Name[1,4-5]= Name,Name_4,Name_5
Name[5] = Name_5
Name[*] = Name,Name_2,Name_3,Name_4,Name_5
```

Therefore some of the ways the above Key-Logic example could also be written are:

```
KEY-LOGIC=SSA,
            System(default), Population(test),
             Controls("FIELD=Person_Name KEY_LEVEL=Standard"),
             Field("Name[*]")
KEY-LOGIC=SSA,
             System(default), Population(test),
             Controls("FIELD=Person_Name KEY_LEVEL=Standard"),
             Field("Name[1-5]")
KEY-LOGIC=SSA,
             System(default), Population(test),
             Controls("FIELD=Person_Name KEY_LEVEL=Standard"),
             Field("Name[1,2,3,4,5]")
```

Specifying Null Values

You can specify the optional `null-field-value` and `null-key-value` parameters in one of the following formats:

Simple text string. For example, `abc` or `"abc"`.

A hexadecimal value. For example, `Hex(8000000000)`.

A filled string. For example, `Fill(" ", 5)` for five blank spaces.

A single character. For example, `Numeric(0)` for 0 byte.

A sequence of values in parentheses. For example, `("ab", numeric(10), cd, Fill (numeric(0), 15))`.

Lite Index

Lite Indexes have lower storage costs than heavy indexes, but they require additional I/O when you use them in a search.

Use the following syntax to create keys and ranges for a Lite Index:

`KEY-LOGIC=User, Field(field)`, where `field` is the name of the column in the IDT that you want to index.

The `Key-Logic` parameter for the IDX-Definition must specify `OPTIONS=Lite`.

The `Key-Logic=User` parameter does not support the `NO-NULL-FIELD` and `NO-NULL-KEY` options because the database that you use controls the user-type index.

Note: A Lite Index does not support the `KEY-SCORE-LOGIC` and `KEY-PRE-SCORE-LOGIC` keywords.

SSA-NAME3 v1

Parameters supporting SSA-NAME3 v1 are no longer documented in this manual, as SSA-NAME3 v2 (or later) is the recommended version to use with IIR. Refer to a previous version of this guide if you require information about deprecated parameters.

Score-Logic

Score Logic is defined in the Search-Definition. It specifies how the Search Server will select records to be returned from the set of candidates. Score-Logic facilities are provided by a SSA-Name3 Standard or Custom Population.

`PRE-SCORE-LOGIC` is used to define an optional "light weight" scoring scheme that is processed before the normal `SCORE-LOGIC`. Its purpose is to make a fast, inexpensive decision to accept or reject the current record to avoid passing it to the more costly `SCORE-LOGIC`. Refer to the *Reducing Scoring Costs* section for details.

Syntax

```
[PRE-]SCORE-LOGIC=SSA,
                        System (system),
                        Population (population),
                        Controls (controls),
                        Matching-Fields (field-list)
```

where

`system` is the System Name of the Population Rules DLL.

`population` defines the population name from the DLL.

`controls` defines the controls to be used for this Score-Logic. Refer to the *STANDARD POPULATIONS* guide for a detailed description of these fields. Controls should specify the desired `PURPOSE` and `MATCH_LEVEL`.

`field-list` is a comma-separated list of the form`field_name:data_type` where `field_name` is the name of a field in the IDT and the `data_type` is the data type that this field represents, as defined in the Population Rules.

Repeating Fields

The `field_list` defines which IDT fields will be used for matching and the type of data they represent (`data_type`). Matching will use repeat logic when two or more fields of the same `data_type` are specified. A

run-time error will occur if a data_type defined as mandatory for the PURPOSE has not been specified in the field_list. Optional data_types may be omitted.

For example,

```
SCORE-LOGIC=SSA,
                System(default), Population(test),
                Controls("PURPOSE=Person_Name MATCH_LEVEL=Loose"),
                Matching-Fields("LastName:Person_Name,Alias:Person_Name")
```

When using a Group of repeating fields (or flattened fields) for matching, you must list each individual field name. For example, the following source definition,

```
test.person.nameName [5] C(20)
```

generates a Group in the File-Definition of this IDT, which is subsequently expanded to the following list of field names:

```
FIELD=Name, C, 20
FIELD=Name_2, C, 20
FIELD=Name_3, C, 20
FIELD=Name_4, C, 20
FIELD=Name_5, C, 20
```

The Matching-Fields parameter must list the fields Name, Name_2, Name_3, Name_4 and Name_5.

It is also possible to use the shorthand method of describing repeating fields. For details refer to the Key Logic section. For Example:

```
SCORE-LOGIC=SSA,
                System(default), Population(test),
                Controls("PURPOSE=Person_Name MATCH_LEVEL=Loose"),
                Matching-Fields("Name[1-3]:Person_Name")
```

is equivalent to:

```
SCORE-LOGIC=SSA,
                System(default),
                Population(test),
                Controls("PURPOSE=Person_Name MATCH_LEVEL=Loose"),
                Matching-Fields("Name:Person_Name,"
                "Name_2:Person_Name,"
                "Name_3:Person_Name")
```

Decision Processing

The Population module returns a Score and a Decision. The Score is a number between 0 and 100, with 100 representing a perfect match. The score is used to sort records prior to returning them to the user. The Decision specifies whether the match was:

- good (Accepted), or
- bad (Rejected), or
- somewhere in between (Undecided).

It is set by the Population DLL depending on the MATCH_LEVEL requested in the Controls (or specified as an override in the Search API).

The Search Server will return both Accepted and Undecided records in response to a search request.

SSA-NAME3 v1

Parameters supporting SSA-NAME3 v1 are no longer documented in this guide, as SSA-NAME3 v2 (or later) is the recommended version to use with IIR. Please refer to a previous version of this guide if you require information on deprecated parameters.

External Scoring Routines

IIR does not support external scoring routines at this time.

User-Source-Table Section

The `User-Source-Table` (UST) section specifies how to build IDTs from User Source Tables or flat files. It is also used to define an SQL source as input data for relate.

An IDT can be built from:

- a single UST
- multiple USTs with a join operation
- multiple USTs with a merge (union) operation
- a (sequential) flat file

The IIR Table Loader can transform fields while extracting them from the data source. It can concatenate fields, insert text and change the case of the source fields, as described below.

The UST parser will automatically generate File and View definitions. There is no need to add anything to the `Files` or `Views` sections.

The User Source Tables must be accessible to the userid (referred to as the SSA userid) used by the IIR Table Loader. You must `GRANT SELECT` privileges to the `SSA` userid on the USTs.

For example, to grant select authority to the user `SSA` on the table `EMP` in SCOTT's schema on the database named `server8.17`, you must

```
CONNECT scott/tiger@server8.17;
GRANT SELECT ON emp TO SSA;
```

General Syntax

The UST section uses an SQL-like syntax. The following general points should be noted:

- lines beginning with two dashes "--" are treated as comments.
- tokens can be substituted with the value of an environment variable by specifying the environment variable name surrounded by #s. eg the Source Schema could be specified as `#myschema#`. When parsed it would be substituted with the value of the environment variable `myschema`.
- all definitions include either the `SYNC` or `NOSYNC` clause. `SYNC` specifies that the IDT must be synchronized with any changes to the UST(s). Specifying `SYNC` means that triggers will be created on the source table to enable the Update Synchronizer to reapply updates to the ID-Tables. Triggers are not created if the IDT is created from a flat file and/or the Source database does not support triggers.
- `NOSYNC` means that the IDT will not be synchronized with any changes made to the Source Tables.
- each table definition is terminated by a semicolon.
- multiple table definitions are permitted in this section.

Primary Keys

Primary Keys (PK) are used to establish a relationship between records in the USTs and IDT. When synchronized USTs are updated, the PK value(s) are passed to the Update Synchronizer to tell it which rows were changed. The PK values are provided by either

- transactions created by triggers defined on the USTs, or

- user created transactions stored in a "flat file".

Columns selected as Primary Keys must not contain any binary zero (NUL) characters in their value. Therefore binary columns, date / timestamp and/or character columns containing binary should not be used (e.g. W columns containing UTF-16 / UCS-2 data).

Furthermore, columns that have been converted to C(64), as listed in the following *Source Data Types* section, can not be used as primary keys. These include the REAL, FLOAT, DOUBLE and NUMBER (with scale > 0) data types. In general, PK fields should be selected from CHAR, VARCHAR and/or NUMBER (scale = 0) columns.

Columns selected as Primary Keys must not contain any zero length data that are not NULL. Only DB2/UDB supports columns with this type of semantics.

Unique Primary Keys

Synchronization works most efficiently when PKs are unique. They are defined to be unique by specifying the (default) synchronization level of `reject_duplicate_pk`.

The nominated PK fields serve two purposes:

- to define a unique PK for the IDT, and
- to define a unique PK for the primary UST (defined below).

This is so that a delete from the primary UST can tell the Synchronizer which IDT record to delete. If the is not unique, the Synchronizer will delete multiple IDT records. If a UST record is added with a non-unique PK, the Synchronizer will report a " constraint violation".

Non-Unique Primary Keys

PKs do not need to be unique and are defined as such using the synchronization level of `allow_duplicate_pk`. However non-unique PKs are more expensive to synchronize, as more records will be processed.

Non-Synchronized USTs

IDTs created from UST that are not synchronized must still define a PK field, even though there is no link maintained between the USTs and IDT.

Rules

In the following paragraphs the term "must/may" is used and should be read as "must" when using unique PKs, and "may" when using non-unique PKs.

The `primary` UST is the table that is used to source the first primary key column for the IDT. The primary UST must/may contain a unique primary key. It may be a compound primary key.

Each IDT record must/may contain a unique primary key. The `sourced_from` clause is used to nominate the source column(s) which form the PK. The first `m` keys fields (`PK1..PKm`) specify the primary keys for the primary UST.

If the IDT is created by joining the primary to a secondary table and there is a one to many (1:M) relationship between them, there will be multiple rows in the IDT for each unique value. In this case, you should define additional columns (`PKm+1..PKn`) which when concatenated to `PK1..m` will form a unique key for each IDT record.

When merging tables, the primary key must be unique within a given User Source Table but does not need to be unique within the IDT because IIR automatically adds an additional qualifier to each IDT record.

In general, IIR allows the PK to be composed of up to 36 columns. However, the number of columns permitted in a composite index may be further limited by the host DBMS. For example on UDB the limit is 10 columns.

All IDT's will have a generated two byte field call `CL_ID`. This field is used in Cluster governance. For normal IDT records this field will be left unpopulated.

Source Data Types

The following tables list the supported source data types and what they are converted to for storage in the IDT and IDX.

The first column shows the native data types that can be read from User Source Tables.

The second column shows the equivalent IIR data type documented in the *Files & Views* chapter, *Format & Data Types* section of this manual). Data read from USTs are converted to a common data type to enable combining source data from heterogeneous source database types. For example, an IDT stored on Oracle can be merged from source data read from UDB and MS-SQL Server.

When loading data from flat files instead of USTs, the File and View definitions are specified using these IIR data types. When loading from a UST, files and views representing the UST and IDT are generated automatically.

The third column shows the mapping from IIR data types back to native host DBMS data types. The IDT is stored as a native table on the target DBMS so that it may be queried using SQL. The IDX holds compressed keys and data (IIR data types) and is not user accessible.

Oracle

UST Data Type	Data Type (IDX)	IDT Data Type
	F	VARCHAR2
CHAR VARCHAR2 CLOB NUMBER (scale > 0) DATE [2] TIMESTAMP	C	VARCHAR2 /VARCHAR2 CLOB [1]
	V	VARCHAR2
RAW BLOB	B	RAW / BLOB[3]
	I	NUMBER(10) / RAW [4]
	G	NUMBER / RAW [5]
NUMBER (scale = 0) INT SMALLINT	N	NUMBER
	R	NUMBER
NCHAR NVARCHAR2 NCLOB	W	NVARCHAR2 /NCLOB [6]

	X	CHAR
	Z	CHAR

[1]VARCHAR2 for column lengths <= 4000, otherwise CLOB.

[2] DATEs and TIMESTAMPs are converted to C(64) fields by default. The length may be overridden. The default installation date mask determines the date format. This is specified either explicitly with the initialization parameter `NLS_DATE_FORMAT` or implicitly with the initialization parameter `NLS_TERRITORY`. It can also be set for a session with the ALTER SESSION command via a logon trigger defined for the SSA user.

[3] RAW for column lengths <= 2000, otherwise BLOB.

[4] NUMBER(10) if the length is 2 or 4, otherwise RAW.

[5] NUMBER if the length is 2 or 4, otherwise RAW.

[6] NVARCHAR2 for column lengths <= 4000, otherwise NCLOB.

UDB

UST Data Type	Data Type (IDX)	IDT Data Type
	F	VARCHAR
CHAR VARCHAR DATE[8] TIMESTAMP [8] NUMBER (scale > 0)	C	VARCHAR / CLOB[7]
	V	VARCHAR
	B	VARCHAR FOR BIT DATA / BLOB [9]
	I	INTEGER
	G	INTEGER
NUMBER (scale=0) DECIMAL INTEGER SMALLINT BIGINT [10]	N	NUM
	R	NUM
	W	VARCHAR / DBCLOB [11]
	X	CHAR
	Z	CHAR

[7]VARCHAR for column lengths <= 32000, otherwise CLOB.

[8]DATEs and TIMESTAMPs are converted to C(64) fields by default.

[9]VARCHAR FOR BIT DATA for column lengths <= 32000, otherwise BLOB.

[10]Negative BIGINTs are not supported for use as a PK and/or join column.

[11]VARCHAR for column lengths <= 32000, otherwise DBCLOB.

Microsoft SQL Server

UST Data Type	Data Type (IDX)	IDT Data Type
SQL_CHAR	F	VARCHAR / TEXT [12]
SQL_VARCHAR SQL_DATE SQL_TIME SQL_TIMESTAMP SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP SQL_NUMERIC [13] SQL_DECIMAL[13] SQL_FLOAT SQL_REAL SQL_DOUBLE SQL_GUID	C	VARCHAR / TEXT [12]
SQL_BINARY SQL_VARBINARY SQL_BIT	B	VARBINARY / IMAGE[14]
SQL_NUMERIC [15] SQL_DECIMAL [15] SQL_INTEGER SQL_SMALLINT SQL_TINYINT SQL_BIGINT	N	DECIMAL
SQL_WCHAR	W	NVARCHAR / NTEXT [16]
SQL_WVARCHAR		

[12]VARCHAR when column length <= 8000, otherwise TEXT.

[13]scale > 0 (floating point numbers).

[14]VARBINARY when column length <= 8000, otherwise IMAGE.

[15]scale = 0 (whole numbers).

[16]NVARCHAR when column length <= 4000, otherwise NTEXT.

Create_IDT

The simplest form of IDT is created from column values extracted from a single User Source Table. The general form of the syntax is:

```
CREATE_IDT IDT-Name
SOURCED_FROM [connection_string] source_clause
[TRANSFORM transform_clause]
[SELECT_BY select_clause]
sync_clause
;
```

where

IDT-Name is the name of the IDT Table to be created. This must be the same as defined by the `IDT-NAME` parameter in the `IDX-Definition` section and the `PHYSICAL-FILE-NAME` in the `Logical-File-Definition`.

Connection String

The `connection_string` is used to specify connection information for access to the UST on the source database. When using flat file input `connection_string` must be set to `flat_file`.

The format of the `connection_string` is documented in the *OPERATIONS guide, Rulebase/Database Names* section.

When connecting to a source database, the `connection_string's SystemQualifier` has no meaning, and should be set to an unused number such as 99 to highlight that fact. For example, when connecting to an Oracle source database the `connection_string` is as follows:

```
odb:99:scott/tiger@oracle920
```

If a *connection_string* is not supplied, it defaults to the connection details of the database used when initializing the Rulebase, i.e. the value of `SSA_DBNAME` environment variable.

If the source tables *are to be synchronized*, you *must* specify the userid associated with the UST database. This is the same userid that was created and used to install the Update Synchronizer components. Further more, the (default) SSA userid on the UST database must have SELECT privileges on the source tables to be referenced. See the *Installation Guide* for details.

If the source tables *do not require synchronization* , any valid userid with SELECT privileges on the source tables may be used.

Source Clause

This section is used to specify the source of the data.

The `source_clause` is used to nominate the UST columns that are to be extracted when creating the IDT. Source fields can be either added directly into the IDT or used in a transformation process, the result of which is added to the IDT.

The `source_clause` syntax is:

```
[src_schema.]table.column [(PKn)] [tgt_col [r] [fmt(len)]] [,...]
```

where

src_schema is the name of the schema that the `table.column` belongs to. The default value is your userid.

table the source table name which contains the `column`.

Oracle: Synonyms (private, public, recursive) may be used but are converted to the real `schema.table` name during parsing.

column the name of the source column to be extracted.

(PKn) nominates `column` as the nth component of the primary key.

tgt_col the name of the column when stored in the IDT. If omitted, it defaults to the value of `column`. If *tgt_col* is prefixed with '$', it is treated as a *virtual field* . Virtual fields are not stored in the IDT. They can only be referenced in a *transform_clause* . If *tgt_col* is prefixed with '$$', it is treated as a special type of *virtual field* that is read from the source database during Table Loading. *Tgt_col* names prefixed by an underscore ('_') are reserved for internal use.

[r] the number of times that *tgt_col* will repeat in the IDT. This parameter is used to declare a repeating field for the flattening process (see the Flattening IDTs). The default value is 1. Column names are generated using the same semantics as the GROUP= clause (described in the *Files and Views* section). Note that the braces are required.

fmt (len) the format and length of the `tgt_col` in the IDT. When omitted, they default to the format and length of the source column. Valid formats are defined in the File Definition section of this guide.

Note that the default length may be insufficient when the source and target databases use different character sets. Refer to the Globalization section of the *OPERATIONS* manual for details.
When the source is a flat file `src_schema.table.column` must be omitted (as there are no USTs), and *tgt_col* , *fmt* and *len* must be provided to describe the layout of the IDT.

Transform Clause

A `transform_clause` is really an optional part of a `source_clause` or `merge_clause`. It is used to specify how virtual fields are to be combined/transformed and characteristics of the resulting field stored in the IDT.

The `transform_clause` syntax is:

```
transform_rule tgt_col [r] fmt(len) [order n] [,...]
```

where

transform_rule nominates the transformation process which will convert virtual fields into a value to be stored in `tgt_col`. Details appear in the next section.

tgt_col is the name of the column to be stored in the IDT.

fmt (len) is the format and length of the `tgt_col`.

[r] an optional number of times that tgt_col will repeat in the IDT. This parameter is used to declare a repeating field for the flattening process. See the Flattening IDTs section for details.

The default value is 1. Column names are generated using the same semantics as the `GROUP=` clause (described in the **Files & Views** section). Note that the braces are required.

order n is used to override the default order of fields in the IDT. Normally fields are placed in the IDT in the order of definition in the `source_clause`and `transform_clause`. You may override this order by nominating an integer value for n, starting from 1.

Transform Rules

Transform rules fall into three categories:

- rules that use no virtual fields `(Source-Id, Insert-Constant)`
- rules that operate on a single virtual field `(Upper, Lower, Convert-Field)`
- rules that operate on many fields `(Append, Concat, Insert-Field)`

Each virtual field can only be referenced once in the `transform_clause`. In the unlikely event that the same source column is to be used in more than one transform clause, add an extra `source_clause` for that column but give it a different virtual field name.

The transform rules use the following format:

```
Source-Id
Insert-Constant ("text")
Upper (vf)Lower (vf)
Insert-Field (vf, offset, ...)
Convert-Field(<Source Field>, <Date Format>) <Target Field> C(<Length in Bytes>)
Convert-Field(<Source Field>, <Encoding Format>) <Target Field> C(<Length in Bytes>)
Append ( vf | lower (vf) | upper (vf) | "text" , ... )
Concat ( vf | lower (vf) | upper (vf) | "text" , ... )
```

You can use one of the following transform rules:

`Source_Id`

Generates a unique ID into `tgt_col`. It should only be used in conjunction with the `AUTO-ID` parameter. The name of the `tgt_col` must match that defined in `AUTO-ID-FIELD` in the `IDX-Definition` section.

`Insert-Constant`

Injects a string of `text` into `tgt_col`.

`Upper`

Converts the virtual field `vf` defined in the `source_clause` to uppercase.

`Lower`

Converts the virtual field `vf` defined in the `source_clause` to lowercase.

`Insert-Field`

Combines a number of virtual fields into `tgt_col`. Each field is stored at the specified `offset` in `tgt_col`. Offsets start from 0.

`Convert-Field` **for Date**

Converts the date format of the source field to the specified date format for the target field. The rule can also truncate the date based on the length that you specify in bytes. The date format of the source field must be `YYYY-MM-DD`.

The Convert-Field rule uses the following parameters:

- `Source Field`. Specifies the field whose date format you want to use.
- `Date Format`. Specifies the date format for the target field.
  Use one of the following values:
  - 0. Retains the same date format as that of the source field.
  - 1. Converts the date format of the source field to the `DD-MM-YYYY` format.
  - 2. Converts the date format of the source field to the `DD-MON-YYYY` format.
  - 3. Converts the date format of the source field to the `DD-MON-YY` format.
  - 4. Converts the date format of the source field to the `MM-DD-YYYY` format.

  Note: `MON` indicates the complete name of a month.
- `Target Field`. Field that uses the converted date format.
- `Length in Bytes`. Length of the converted date to use. When you use the date format 2, ensure that the length is at least 11 bytes.

The following Convert-Field rule converts the date format of the `$date` field for the `DOB` field:

```
convert-field($date, 2) DOB C(10)
```

`Convert-Field` **for Encoding Format**

Converts the encoding format for fields from UTF-8 to UTF-16 and from UTF-16 to UTF-8.

The Convert-Field rule uses the following parameters for the encoding format conversion:

- `Source Field`. Specifies the field that you want to use for conversion.
- `Encoding Format`. Specifies the encoding format for the target field.
  Use one of the following formats:
  - `convert-field(8)`. Converts from UTF-16 to UTF-8.
  - `convert-field(6)`. Converts from UTF-8 to UTF-16. Converting to UTF-16 might double the size. The transforms truncate the output if it is not large enough for the converted text.
- `Target Field`. Field that uses the converted encoding format.
- `Length in Bytes`. Length of the converted encoding format to use.

Consider the following sample user table:

```
create_idt IDT547
sourced_from #SSA_DBNAME#
#SSA_SCHEMA#.table.PolicyHolderName $field1,
#SSA_SCHEMA#.table.Gender $field2,
#SSA_SCHEMA#.table.Address $field3,
#SSA_SCHEMA#.table.PolicyID (PK1),
#SSA_SCHEMA#.table.PINCODE $field4
```

Use the following format to convert from UTF-16 to UTF-8:

```
TRANSFORM
convert-field($field1, 8) PolicyHolderName C(20),
convert-field($field2, 8) Gender C(6),
convert-field($field3, 8) Address C(50),
convert-field($field4, 8) PINCODE C(10)
```

`Append`

Combines the virtual fields and text by stripping trailing spaces from all fields and joining them together into `tgt_col`.

`Concat`

Functions same as `Append` but adds a single space between the fields when joining them.

Select Clause

Is any valid SQL expression (for a WHERE clause) that can be used to select a `subset of records` from the UST. That is, the `select_by` clause acts a final filter to remove records after selecting / joining rows from the UST. IIR does not parse this expression. It is simply appended to the WHERE clause generated by IIR using an AND logical condition.

```
SELECT ... WHERE <IIR_expression> AND (<select_clause>)
```

It is important to ensure that the `select_clause` is syntactically correct. Failure to do so will result in run-time SQL errors in the Table Loader and/or Update Synchronizer.

Note: All columns referenced in the `select_clause` must also appear in the `sourced_from` list (for SYNC systems only). This ensures that they are replicated in the IDT and that correct synchronization will occur.

Do not try to limit the number of rows loaded from the UST using a physical limit. For example,

```
select_by ROWNUM <= 1000
```

This approach instructs the SQL Optimizer to return the first 1000 rows but has the disadvantage that the "first 1000 rows" may not be the same ones when the system is loaded a second time. It will produce inconsistent results.

Use a logical limit when selecting a subset of records from the UST. For example,

```
select_by EmpId >= 50000 AND EmpId <= 51000
```

This ensures a repeatable set of records.

Sync Clause

The sync clause determines whether or not this IDT will be synchronized with updates to the UST. If it is synchronized an optional Synchronization Level can be specified. The syntax is:

```
NOSYNC | SYNC [sync_level [txn_source_clause]]
```

where `sync_level` is either:

REJECT_DUPLICATE_PK Rejects all duplicates (the default).

REPLACE_DUPLICATE_PK Replaces duplicates when synchronizing from an NSA Txn-Source (see Txn-Source Clause below).

WARN_DUPLICATE_PK Produces a warning when a duplicate is added.

ALLOW_DUPLICATE_PK Allows non-unique PKs.

Refer to the *OPERATIONS guide, Update Synchronizer* chapter for more details about the Synchronization Level.

Txn-Source Clause

Use `txn_source_clause` to specify the method to provide synchronizer transactions. If you do not specify, an identity table created from the user source tables uses trigger-generated transactions and an identity table created from a flat-file uses flat-file transactions. Use the following syntax for `txn_source_clause`:

```
TXN-SOURCE { TRIGGER | MANUAL | FLAT_FILE | NSA }
```

where

TRIGGER indicates that the triggers are automatically created on the user source tables to generate transactions in the Synchronizer's Transaction Table.

MANUAL indicates that the triggers are not created on the user source tables. You have to manually insert transactions in the Synchronizer's Transaction Table when a user source table is updated.

FLAT-FILE indicates that the synchronizer transactions are provided in a flat-file. Use this option only when you source data from a flat-file.

NSA indicates that the synchronizer transactions are provided in a NSA Transaction Table. You can use this option when you source data from a flat-file.

For example:

```
CREATE_IDT test-idt
SOURCED_FROM odb:99:scott/tiger@oracle8.17
           SCOTT.EMP.EMPNO (PK)    EmpNo,
           SCOTT.EMP.ENAME         EmployeeName
SELECT_BY
          (scott.emp.empno > 7800)
SYNC
;
```

The example creates the `test-idt` identity table. The data is extracted from an Oracle service named `oracle8.17` by using the userid `scott` whose password is `tiger`. The identity table contains two fields, `EmpNo` and `EmployeeName`. The fields have formats and lengths similar to their corresponding source fields, `EMPNO` and `ENAME`, in the SCOTT.EMP table. Only employees with employee numbers greater than 7800 are extracted and loaded into the IDT.

Any updates made to `SCOTT.EMP` are applied to `test-idt` by the Update Synchronizer by using the default synchronization level of `REJECT_DUPLICATE_PK`. Synchronizer transactions are generated by using triggers on the source table.

Flat_File Input

The syntax for `flat_file` source is similar to the database source, but differs in the following aspects:

- DB source column names are omitted
- the layout of the file is specified using the target fields
- each field must have an explicit format and length definition
- `PKn` definitions appear before field names
- virtual fields (used for transforms) must provide format and length information

For example:

```
CREATE_IDT
        IDT264
SOURCED_FROM flat_file
          (pk)     RowId               W(16),
                     $FirstName          W(50),
                     $MiddleName       W(50),
                     $LastName           W(50)
TRANSFORM
    concat (upper ($LastName), $MiddleName, $FirstName)     Name W(150),
SYNC REPLACE_DUPLICATE_PK
TXN-SOURCE NSA
;
```

Join_By

You can create an IDT by joining two or more USTs from a single source database. This is normally done when the source tables are normalized into multiple tables and the IDT needs to support search and matching strategies by using data from all source tables.

The syntax is identical to the syntax of a single UST with the addition of `join_expression`.

```
CREATE_IDT IDT-Name
SOURCED_FROM connection_string source_clause
[TRANSFORM   transform_clause]
JOIN_BY      join_expression
[SELECT_BY   select_expression]
SYNC | NOSYNC
;
```

Note: You cannot join columns that are of binary data types.

Source Clause

For more information about the source clause, see the Create_IDT section.

Transform Clause

For more information about the transform clause, see the Create_IDT section.

Join Expression

The primary table is a UST that you mention in the `sourced_from` clause. A secondary table is another UST that you join to the primary table by using a foreign key stored in the primary.

Perform an outer join on the primary table by adding all the rows in the primary table to the IDT even if any row fails to join to any secondary table. In this case, the columns extracted from the secondary table are set to NULL.

Use *join_expression* to specify how to join the primary table to one of the secondary tables or more. Generally, it specifies how to join a parent table to a child. Specify *join_expression* for each pair of tables that

you plan to join. The expression defines the relationship between the tables, where the parent contains the foreign key of the child.

Use the following format for *join_expression*:

```
parent_table_column = child_table_column [AND p_col = c_col], . . .
```

where,

parent_table_column is a fully qualified column name in the `parent table`.

child_table_column is a fully qualified column name in the `child table`.

p_col is an unqualified column name in the `parent table` (for compound keys).

c_col is an unqualified column name in the `child table`(for compound keys).

All *parent_table_columns* specified in a *join_expression* must be included in the IDT as non-virtual fields. For performance reasons, it is recommended that *parent_table_columns* are indexed. If `SYNC` is also specified, all join fields must be indexed.

Example

The IDT `idt_xform` is to be created. Columns are extracted from two tables, `TESTX50A` and `TEXTX50B`. These tables belong to schema `#SSA_UID#` which is evaluated at parse time using the environment variable `SSA_UID`.

The tables are joined using the `EMPNO` column in `TESTX50A` and the `SSN` column in table `TESTX50B`.

Various transformations are used. Columns `given` and `family` are concatenated to form a field called `NAME`. The `order` parameter is used to change the default ordering of fields in the IDT to: `myid, NAME, EMPNO, SSN, TITLE, Addr`, and `Phone`.

```
CREATE_IDT
          idt_xform
SOURCED_FROM odb:99:scott/tiger@server8.17
#SSA_UID#.TESTX50A.EMPNO     (PK)             EmpNo N(25),
     #SSA_UID#.TESTX50A.given                    $given,
     #SSA_UID#.TESTX50A.family                  $family,
     #SSA_UID#.TESTX50A.ADDR                          $addr,
     #SSA_UID#.TESTX50B.SSN,
     #SSA_UID#.TESTX50B.PHONE                    $phone
TRANSFORM
     source-id                           myid     f(10) order 1,
      insert-constant("hello there")        title    c(15),
     upper($addr)                        Addr     c(30),
     lower($phone)                       Phone    c(12),
     concat($given,$family)              NAME     c(50) order 2
JOIN_BY
     #SSA_UID#.TESTX50A.EMPNO = #SSA_UID#.TESTX50B.SSN
NOSYNC
;
```

Merged_From

An IDT can be created by merging the contents of two or more User Source Tables. Multiple heterogeneous source databases are permitted. The columns extracted from the tables are mapped into a common format using multiple `merge_clause` and `transform_clause` pairs (one pair per UST).

```
CREATE_IDT IDT-Name
MERGED_FROM [connection_string] merge_clause
[TRANSFORM transform_clause] ...
SYNC | NOSYNC
;
```

Primary Key

The `(PK)` column must contain unique values within a given User Source Table. However it does not need to be unique within the merged IDT because IIR automatically qualifies the `(PK)` column so that it can identify which source table the record came from. To do this, IIR automatically adds a column called `IDS_PARSE_BLOCK_NO` to the IDT and populates it with the `merge_clause` occurrence number (starting from 1) used to create each IDT record. This column name is reserved and can not be specified as a `tgt_col` name.

Source Clause

Refer to the Source Clause description in the Create_IDT section.

Transform Clause

Refer to the Transform Clause description in the Create_IDT section.

Merge Clause

The `merge_clause` is identical to the `source_clause` in syntax but its semantics differ:

- The first `merge_clause/transform_clause` pair is used to define the IDT column names, formats, lengths and order. It also nominates the primary-key field `(PK)`.
- The second and subsequent pairs define the mapping from source columns in other USTs to the `tgt_cols` defined in the first pair. They cannot specify format, length, `PK` nor order. `tgt_col` names must match those defined by the first pair.
- All columns in a `merged_from` clause must come from the same source table.

Example

The IDT `idt_xform_merge` is to be created from data extracted from tables `TEXTX51A` and `TESTX51B`. These tables are found on different databases (`mars` and `jupiter` respectively).

The common layout of the IDT is defined by the first `merge_clause/transform_clause` pair as:

MyIdC(10)

NAMEC(50)

AddrC(50)

SSNformat/length same as `TESTX51A.SSN`

Columns `ADDR_L1, ADDR_L2, ZIP` from table `TESTX51A` will be concatenated to form a value for `Addr`.

Columns `GIVEN` and `FAMILY` from table `TESTX51B` will be concatenated to form a value for `NAME`. The field `EMPNO` in `TESTX51B` maps to `SSN`.

```
CREATE_IDT
       idt_xform_merge
MERGED_FROM odb:99:scott/tiger@mars
       #SSA_UID#.TESTX51A.FULL_NAME           NAME C(50),
       #SSA_UID#.TESTX51A.SSN        (PK)     SSN,
       #SSA_UID#.TESTX51A.ADDR_L1             $a1,
       #SSA_UID#.TESTX51A.ADDR_L2             $a2,
       #SSA_UID#.TESTX51A.ZIP                 $zipcode
TRANSFORM
       source-id                              MyId c(10) order 1,
       concat($a1,$a2,$zipcode)               ADDR c(50) order 3

MERGED_FROM odb:99:buzz/lightyear@jupiter
       #SSA_UID#.TESTX51B.EMPNO    SSN,
       #SSA_UID#.TESTX51B.GIVEN    $given,
       #SSA_UID#.TESTX51B.FAMILY   $family,
       #SSA_UID#.TESTX51B.ADDR     ADDR
TRANSFORM
```

```
       concat($given,$family) NAME
SYNC
;
```

Define_Source (relate input)

The UST Section is also used to define database source tables to be used to supply input records to a `relate` batch search process. The syntax is similar to the `CREATE_IDT` clause, with a few minor differences:

- `DEFINE_SOURCE` replaces `CREATE_IDT`,
- `PKn` is not permitted,
- `SYNC` is not permitted,
- `NOSYNC` is optional (and implied).

The syntax is:

```
DEFINE_SOURCE SrcName
SOURCED_FROM [connection_string] source_clause
[TRANSFORM   transform_clause]
[JOIN_BY     join_expression]
[SELECT_BY   select_clause]
[NOSYNC]
;
```

where

SrcName is the name of the data source.

When the System is loaded, a File and View named `SrcName` is created.

The View is used as an input view for relate `(-iSrcName)` to describe the input records. The target field names in the `source_clause` (and therefore View) must match the field names in the IDT in order for these fields to be mapped to IDT fields.

Example

```
define_source SRC05
sourced_from odb:99:uid/pwd@svc
        #SSA_UID#.TESTX05A.RECNUM       RecNum C(5),
        #SSA_UID#.TESTX05A.LASTNAME     $last,
        #SSA_UID#.TESTX05A.FIRSTNAME    $first,
        #SSA_UID#.TESTX05A.MIDDLENAME   $middle,
        #SSA_UID#.TESTX05A.ADDR1        $a1,
        #SSA_UID#.TESTX05A.ADDR2        $a2
transform
        concat ($last,$first,$middle)   Name C(50),
        concat ($a1, $a2)               Addr C(40)
;
```

The following View-Definition is generated automatically to define the record layout for relate:

```
VIEW-DEFINITION
NAME=src05
FIELD=RecNum,C,5
FIELD=Name,C,50
FIELD=Addr,C,40
```

Sourcing from Microsoft Excel

A Microsoft Excel spreadsheet may be used as a data source using an appropriate ODBC driver. It is treated as an unsynchronized source database. We present a few tips to make the process easier:

The DSN should be configured appropriately, listing all necessary parameters including the name of the file containing the spreadsheet.

The source table name is usually the sheet name followed by a dollar sign. For example, Sheet1$. However, as this is a non-standard table name, ODBC requires it to be surrounded by back quotes: ‘Sheet1$‘. Unfortunately, back quotes are not permitted by the SDF syntax. This may be overcome by creating a Named Range in Excel.

Select the entire sheet, or the portion to be made visible to IIR, followed by the menu options

Insert > **Name** > **Define...** and specify a name for the range using alphanumeric characters only (e.g. `test_named_range`). This name is then specified as the table name in the SDF definition.

```
create_IDT xltest
sourced_from odb:99:ssa/ssa@myExcel
        test_named_range.id (PK),
        test_named_range.name NAME C(50),
        test_named_range.address ADDRESS C(50),
        test_named_range.city CITY C(50),
        test_named_range.state STATE C(2),
        test_named_range.zip ZIP C(10)
NOSYNC
;
```

Files and Views Sections

These sections are used to define files and views. Files describe the layout of IIR Tables created from flat-file input. Views are used to:

- describe the layout of flat-file input data
- transform flat-file input data
- format the output from the Search Server

Data Source

When loading IIR Tables from User Source Tables, the Files Definition and View Definition files are created automatically from the User Source Table Definition.

The name of the generated Files definition is the same as the IDT name in the `CREATE_IDT` clause. The View name(s) are created by concatenating the IDT name with a sequence number starting from 1.

When loading IIR Tables from flat (external) files, the File and View definitions must be hand-coded. The File definition describes the layout of the IDT, while the View Definition describes the layout of the input file.

File Definition

A File Definition begins with the `FILE-DEFINITION` keyword. It is used to describe the layout of the IIR Tables. Although it is possible to specify more than one File Definition, only one definition can be in effect for each IDT.

A File Definition contains one parameter which is unique to File Definitions. This is the `ID=` parameter. It is mandatory and is used to allocate a unique number to the IDT (for internal use). Specify a positive, non-zero number that is less than 2000. File numbers greater than 1999 are reserved for internal use. File numbers must be unique within the scope of the target database.

View Definition

A View Definition begins with the `VIEW-DEFINITION` keyword. As many views as necessary may be defined. A view normally consists of a subset of fields taken from a File Definition. It may contain additional fields that are added by user-defined Transformations (see below).

Syntax

The definition starts with a name as described above. It is followed by:

```
NAME=
```

This character string names the file or view. It may be up to 32 bytes long.

```
ID=
```

This parameter is only specified for File Definitions; see the File Definition section above.

```
FIELD=name,format,length[,PKn][,Xform]
```

The parameter is used to define the fields that comprise the File or View. The maximum number of fields is platform specific. The order of `FIELD` definition statements determines the physical ordering of fields on the database record and/or view.

Name A character string that names a field. It may be up to 32 bytes in length.

Format The format of the field. This may be defined using a pre-defined format, or as a customized format. See the section below.

Length The length of the field in bytes. The maximum length is platform specific.

PKn defines this field as a Primary Key. This is useful when creating an IDT from a flat-file, which will subsequently be used to create a Link Table with the `Link-PK` option.

Xform A transform definition. Refer to Transformations section for details.

```
GROUP=name, number of members
```

The parameter is used to define a group of repeating fields within a file or view. A GROUP= specifies that the fields defined within the group definition should be defined as multiple adjacent fields of equal length. The group definition must be terminated by `END_GROUP` The field names defined by the group will be of the format `FIELD=fieldname_x`where `x = 2..n` and `n` is the number of members in the group.

Name A character string that names the group. It may be up to 32 bytes in length.

number of members The number of members in this repeating group.

File & View Data Types

The format of a field may be specified in one of two ways:

- a pre-defined format
- a customized format definition

A pre-defined format is a shorthand way of selecting a pre-defined group of field attributes. It is selected by specifying the pre-defined name for the `<format>` value.

A customized format definition is used when no pre-defined format exists for the combination of attributes that you desire. You may combine various field attributes to your liking (as long as they do not conflict).

Pre-defined Formats

The following table lists the pre-defined formats and their attributes:

Format	Compression	Data	Base	Conv	Just	Filler
F	Fixed	Text	N/A	No	Right	' '
C	Variable	Text	N/A	No	Left	' '
V [17]	V-type	Text	N/A	No	Left	' '
B [18]	Variable	Binary	N/A	No	Left	0x00
I [19]	Variable	Integer	0	Yes	Right	0x00
G	Fixed	Integer	0	Yes	Right	0x00
N [20]	Variable	Numeric	10	Yes	Right	'0'
X [20]	Variable	Numeric	16	Yes	Right	'0'
Z [20]	Variable	Numeric	36	Yes	Right	'0'
R [20,21]	Variable	Numeric	10	Yes	Right	' '
W [22]	Variable	Binary	N/A	No	Left	0x0020

[17] V-type will compress multiple embedded blanks from the input view into one blank in the target field. Therefore it should only be defined in a File-Definition. It has no effect in a View-Definition.

[18]The Binary data type can hold any unstructured data. It is not necessarily a base 2 number.

[19]Valid integer lengths are 1, 2, 3 or 4 bytes.

[20]The Numeric data type is the printed representation of an unsigned (positive) number.

[21]R format is equivalent to N, but with leading spaces instead of zeroes.

[22] W format represents a "wide" UNICODE character encoded as UTF-16. Note that the byte order for filler value is platform specific. The length of the field in the File-Definition / View-Definition is defined in bytes. A field definition of W(n) bytes creates an associated database column of type NVARCHAR(n/2) characters.

Compression

The compression attribute determines how a field is compressed/decompressed when writing to/reading from the database. Fixed compression is effectively no compression at all; the field is stored/retrieved without any compression or decompression. Variable compression means that the data will be compressed when written to the database and decompressed (to match the view definition) when read from the database. The filler character and justification determine which character is stripped /added from the record and from which end.

V-type compression will compress multiple adjacent blanks into a single blank. This can occur anywhere within the field. Note that this process is irreversible. Decompression will only add blanks at one end of the field.

Filler Character/Justification

The decompression will add the filler character to the field to pad it to the necessary length

The compression logic will remove the filler character from either the left or right end of the field (depending on justification), until no more filler characters are left or the field is empty. The decompression will add the filler character to the field to pad it to the necessary length (as specified in the view). If the field is left justified, the truncation/padding occurs on the right and vice-versa.

Data Type

The Data type of the data. The following data types are supported:

- *Text* character data
- *Binary* binary (unstructured) data
- *Integer* 1, 2, 3 or 4 byte unsigned integers
- *Numeric* fields containing printable numeric digits '0' to '9' and 'a' to 'z' (when using base 36).

Base / Base Conversion

The Integer and Numeric data types support base conversion, that is the view may request a base which differs from the base of the stored data. Base conversion is only possible for bases 2 through 36.

Customized Format Definition Syntax

A customized format definition consists of a pre-defined format plus overrides for some of its default attributes. The syntax is as follows:

```
predefined_format ( format_ specifier, ... )
```

where `format_specifier` is one of the following:

- *Fixed* fixed compression
- *Text* text data
- *Ljust* left justification
- *Rjust* right justification
- *Filler(<char>)* filler character
- *Base(nn)* base nn

Examples

This is an example of a pre-defined format definition that creates a twenty-four byte character field named EmployeeName.

```
FIELD=EmployeeName,C,24
```

This is an example of a field definition with a customized format.

```
FIELD=MyBitFlags,X (Base(2),Filler(0)),16
```

It represents a numeric field of 16 bytes of base 2 (binary base). Each position will be either 'zero' or 'one' and leading zeroes (the filler) will extend the value on the left if it is shorter than 16 characters. If we want the value as four hexadecimal digits instead then we could use this format:

```
FIELD=MyBitFlags,X (Base(16),Filler(0)), 4
```

It happens to be equivalent to the predefined format X, so it could also be written as:

```
FIELD=MyBitFlags, X, 4
```

This is an example of a field definition with a customized format that is based upon the pre-defined format C but overrides the justification attribute:

```
FIELD=Address, C(Rjust), 50
```

Transformations

Fields within a View Definition may specify an optional transformation. A transformation definition follows the field's length attribute and has the following format:

```
Xform(transform [, parameter-list])
```

where *transform* is one of the following:

insert-constant inserts a character string into the current source field. The `parameter-list` contains the character string.

uppercase converts a source character field to upper case.

lowercase converts a source character field to lower case.

insert-field inserts the current source field into the target field at an offset specified in the `parameter list`.

concat concatenates the current source field to the target field, leaving a space between the two fields.

append appends the current source field to the last non blank character in the target field.

append-string appends a character string to the last non blank character in the source field. The `parameter-list` contains the character string.

filler a no operation transformation

fill fill the source file with the string specified in the `parameter-list`.

The example below shows a generalized transform as part of an element in a View Definition.

Transform either manipulates the `src` field of the view (see example below) or a target field specified in the `parameter-list`(and corresponding to a field in the File Definition). `insert-constant, uppercase, lowercase, append-string` and `fill` work on the `src` field while `insert-field, concatenate` and `append` put `src` data into the specified target field.

```
FIELD=src, format, size, XFORM(transform [, parameter-list])
```

Multiple transformations can be defined on a single target field. They are processed in the order specified in the `parameter-list` for `concatenate` and `append`, while multiple `insert-string` transforms will occurs in the order they are specified. Multiple transforms should not specify the same order as in:

```
FIELD=Name1, C, 15, XFORM(concatenate, "target name,1")
FIELD=Name2, C, 5, XFORM(concatenate, "target name,1")
```

The behavior in such cases is undefined.

Append

The `append` transform has the following syntax:

```
Xform (append, "target field-name, order")
```

field-name specifies the field to which the current field will be appended. *order* specifies the order in which to append multiple fields to *field-name*. Specify a number, starting from 1.

Concatenate

The `concatenate` transform has the following syntax:

```
Xform (concatenate, "target field-name, order")
or
Xform (concat, "target field-name, order")
```

field-name specifies the field to which the current field will be concatenated. order specifies the order in which to concatenate multiple fields to field-name. Specify a number, starting from 1.

Insert-Field

The insert-field transform has the following syntax:

```
Xform (insert-field, "target field-name [,offset]")
```

`field-name` specifies the field to which the current field will be appended at offset `offset`. The default value of offset is 0.

Insert-Constant

The `insert-constant` transform has the following syntax:

```
Xform (insert-constant , "Constant String")
```

This transform inserts a constant string into the source field.

A special expression `"Numeric(NUM)"` can be used instead of a literal string. This means that the first posit "Numeric(NUM)"ion of the target field will contain the character value of the decimal number NUM using the computer's native character set. If the target fields is longer than one byte, the remaining bytes are filled with spaces.

UpperCase

The `uppercase` transform has the following syntax:

```
Xform (uppercase)
```

It will upper case the contents of the source field.

LowerCase

The lowercase transform has the following syntax:

```
Xform (lowercase)
```

It will lower case the contents of the source field.

AppendString

The append-string transform has the following syntax:

```
Xform (append-string, "Constant String")
```

The `Constant String` is appended after the last non blank space in the source field. If the `Constant String` is larger than the space left in the source field it will over write characters in the source field. For example in the transform below if the field inserted into `LABEL` from the input record is 3 characters (no trailing spaces) the @ symbol will overwrite the last character in `LABEL`.

```
FIELD=LABEL, C, 3, Xform(append-string, @),
```

Fill

The fill transform has the following syntax:

```
Xform (fill, "Fill string")
```

A special expression `"Numeric(NUM)"` can be used instead of a literal string. This means that the target field will be filled with the character value of the decimal number `NUM` using the computer's native character set.

Example 1

```
VIEW-DEFINITION
*==============
NAME=DATAIN
* Insert "hello" into Title
```

```
FIELD=Title, C, 10, Xform(insert-constant, "hello")
* Upper case the contents of Surname
FIELD=Surname, C, 15, Xform(uppercase)
* Lower case the contents of First
FIELD=First, C, 45, Xform(lowercase)
* Insert Addr1 in ADDR at offset 0
FIELD=Addr1, C, 45, Xform(insert-field, "target ADDR")
* Insert Addr1 in ADDR at offset 45
FIELD=Addr2, C, 45, Xform(insert-field, "target ADDR,45")
* Concatenate Name1 to the contents of NAME in sub field 1
FIELD=Name1, C, 45, Xform(concatenate, "target NAME,1")
* Concatenate Name2 to the contents of NAME in sub field 2
FIELD=Name2, C, 45, Xform(concatenate, "target NAME,2")
```

Example 2

The view definition in this example expects a 5 field input record where the fields are Name1 (15 chars), Name2 (5 chars), Category (4 chars), Status (2 chars) and Reference (4 chars). All of these fields are stored in a single output field name (50 chars). The order of the input fields is swapped around in the output and a number of labels are inserted.

```
VIEW-DEFINITION
*==============
NAME=DATAIN
* Read all the input fields and store them in
* the order needed
FIELD=Name1, C, 15, Xform(concatenate, "target name,8")
FIELD=Name2, C, 5, Xform(concatenate, "target name,6")
FIELD=CAT, C, 4, Xform(concatenate, "target name,4")
FIELD=STAT, C, 2, Xform (append, "target name,2")
FIELD=REF, C, 4, Xform (append, "target name,10")
* Set up the first label
FIELD=LABEL1, C, 3, Xform (append-string, LB1),
        Xform(append, "target name,1"),
        Xform(concatenate, "target name,9")
* Set up the second label
FIELD=LABEL2, C, 3, XFORM(insert-constant, LB2),
        Xform(concatenate, "target name,3"),
        Xform(concatenate, "target name,7")
* Fill Filler with the symbol |
FIELD=Filler, C ,2, Xform(fill, "|"),
        Xform(append, "target name,5")
```

For this view an input record of:

```
La Perouse JamesEXPLDD4778
```

Would produce an output name field of:

```
LB1DD LB2 EXPL|| James LB2 La Perouse LB14778
```

Where the transforms occurred in the following order:

1. append LABEL1 (LB1),
2. append STAT (DD),
3. concatenate LABEL2 (LB2),
4. concatenate CAT (EXPL),
5. concatenate Filler (||),
6. concatenate Name2 (James),
7. concatenate LABEL2 (LB2),
8. concatenate Name1 (La Perouse),
9. concatenate LABEL1 (LB1) and

10. append REF (4778).

Output Views

Output views are used to format search results returned by the Search Server. Rows are returned in IDT format when no output view has been specified or in a different format when an output view has been defined.

Output Views are also used to inject search statistics into the output.

Statistical Fields

When the following field names are specified in the output view, the Search Server will provide statistics gathered during the search process.

Note that some data in the output of a search is specific to each row. For example, the score is a property that changes for each row. Other statistics are relevant to the search set as a whole. For example, the number of candidates selected is the same for each row returned because it is a property of the set.

Also note that it is not possible to retrieve search statistics if the search does not return any rows. In this case, a dummy IDT row (filled with asterisks) can be returned, solely as a vehicle to return the statistics. In this situation, the count from an `ids_search_start` will return 0 but `ids_search_get` can be called to return the dummy row.

Note: If the Search-Definition specifies a Score-Logic without a Key-Score-Logic and an IDX with full key data is available, IIR will upgrade the Score-Logic to Key-Score-Logic as the latter is more efficient. In this case one should request KSL-* statistics instead of SL-* statistics.

Field Name	Description
IDS-SCORE	Score from matching
IDS-IDX-IO	Number of IDX rows retrieved
IDS-IDT-IO	Number of IDT rows retrieved
IDS-MS-SEARCH-NUM	Ordinal number of the successful search within a Multi-Search starting from 1. 0 is returned for unsuccessful searches. See the `SEARCH-LIST` Multi- Search parameter for more details.
IDS-XXX-ACCEPTED-COUNT	Number of records accepted by scoring
IDS-XXX-UNDECIDED-COUNT	Number of undecided records
IDS-XXX-REJECTED-COUNT	Number of rejected records
IDS-XXX-TOTAL-COUNT	Total number of records scored

where

XXX is `KPSL` (Key-Pre-Score-Logic), `KSL` (Key-Score-Logic), `PSL` (Pre-Score-Logic) or `SL` (Score-Logic).

CHAPTER 3

Flattening IDTs

This chapter includes the following topics:

Concepts

Flattening is the process of packing denormalized data created by joining tables in a "one to many" (1:M) relationship, into repeating groups in the IDT. It can significantly increase search performance.

The problem

A typical database design consists of multiple tables that have been normalized to some degree. While full normalization is logically elegant and efficient to update, it performs poorly for read access when the tables need to be joined to satisfy a query.

IIR provides very fast search performance by storing all search, matching and display data together in the same table, thereby avoiding the need for costly joins at search time. The IIR Table Loader denormalizes data while creating the IDT.

A disadvantage of denormalization is the explosion in the number of rows that occurs when joining tables that have a 1:M relationship, since the SQL engine produces one row for every possible combination. As a result, storage, retrieval and matching costs increase.

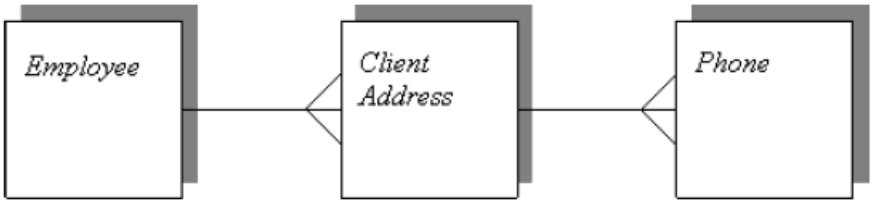
To overcome this problem, IIR can collapse (flatten) related denormalized rows into one IDT row that contains repeating groups. This means:

- significantly faster search performance, and
- reduced database storage costs for the IDT and IDXs

The most significant performance benefits occur when the Key-Field for the IDX is sourced from the 1 table in the 1:M relationship. This is due to the fact that the use of repeating groups for the M data reduces the number of rows containing the same Key-Field value, which in turn produces less keys to index (at load time) and less candidates to retrieve and match at search time.

Logical Design

Consider the following logical design where the tables have been fully normalized. Each employee of a contracting business can work for many clients. Each client's business premises may have many phone lines.



Suppose there is only one employee currently:

Emp_Id	Name
E001	Joe Programmer

Joe contracts his services to three companies located around the state:

Company_Id	Address	FK_Emp_Id
C001	3 Lonsdale St	E001
C002	19 Torrens St	E001
C003	4 Rudd St	E001

Each company has the following phone numbers:

Company_Id	Phone
C001	62575400
C001	62575401
C002	98940000
C003	52985500
C003	52985501
C003	52985502

Denormalized Data

A simple SQL query to denormalize this information will generate six rows of data because Joe Programmer works at three offices, each having multiple phone numbers.

Emp_Id	Name	Company_Id	Address	Phone
E001	Joe Programmer	C001	3 Lonsdale St	62575400
E001	Joe Programmer	C001	3 Lonsdale St	62575401
E001	Joe Programmer	C002	19 Torrens St	98940000
E001	Joe Programmer	C003	4 Rudd St	52985500
E001	Joe Programmer	C003	4 Rudd St	52985501
E001	Joe Programmer	C003	4 Rudd St	52985502

If the search application needs to search on Name, IIR will create fuzzy keys on the Name column.

As there are many rows with the same value, duplicate keys will be created and maintained. At search time, multiple rows will be retrieved and matched.

Flattened Data

To reduce the number of rows, IIR can flatten the six rows into one.

This is achieved by declaring some columns in the IDT as repeating fields. If data sourced from the M tables (`Company_Id`, `Address` and `Phone`) were defined to repeat up to six times, all of the data could be flattened into one row.

Note: The table below has been turned on its side due to space limitations. It represents a single row of data with column names on the left and data values on the right. Note how the data from the "1" Table no longer repeats.

Column	Value
Emp_Id	E001
Name	Joe Programmer
Company_Id [1]	C001
Company_Id [2]	C002
Company_Id [3]	C003
Address [1]	3 Lonsdale St
Address [2]	3 Lonsdale St
Address [3]	19 Torrens St
Address [4]	4 Rudd St

Column	Value
Address [5]	4 Rudd St
Address [6]	4 Rudd St
Phone [1]	62575400
Phone [2]	62575401
Phone [3]	98940000
Phone [4]	52985500
Phone [5]	52985501
Phone [6]	52985502

An IDT with this structure would have only one row and the IDX would contain six times fewer `Name` keys. The number of candidates selected during a search on the `Name` IDX would also decrease by a factor of six.

Note: This structure does not improve the performance of an IDX created on the `Address` fields, as they contain duplicate values. However, Flattening-Options (discussed later) can be defined to remove duplicate entries from the repeating fields in order to provide some benefit as well.

Syntax

Flattening is enabled by defining:

- the layout of the denormalized (joined) data,
- the layout of the flattened IDT, including the maximum number of occurrences for repeating fields,
- the columns used to determine how to group denormalized data (Flattening-Keys), and
- Flattening-Options.

IDT-Definition

The `Denormalized-View` keyword is used to define the name of the `View-Definition` that provides the layout of the denormalized data. The denormalized data can be read from either a flat-file or UST. If read from a flat-file, you must provide a View-Definition. The view's name must be the IDT name, suffixed with `"-DENORM"`. If the data is sourced from USTs, a View-Definition is generated automatically from the `User-Source-Tables` section.

Note: A denormalized view is only generated when the UST Definition contains repeating fields defined with the [ ] notation.

The `Flatten-Keys` keyword is used to define the IDT columns used to group (sort) denormalized rows. If more than one column is specified, use a comma-separated list surrounded by quotes ("). The denormalized data are sorted by the `Flatten-Keys` and all rows with the same key value are packed into the same flattened row. Refer to the *Flattening Process* section for details.

The Options keyword in the IDT-Definition is used to specify various flattening options that affect how data is packed into the repeating groups. Refer to the *Flattening Options* section for details.

For example,

```
IDT-Definition
*==================
NAME= IDT112
DENORMALIZED-VIEW= IDT112-DENORM
FLATTEN-KEYS= "EmpId,EmpName"
OPTIONS= Flat-Remove-Identical
```

IDT Layout

The IDT layout can be provided as a `File-Definition` (when sourcing from a flat-file) or from the `User-Source-Table` section when sourcing from USTs. The IDT layout must be identical to the Denormalized-View (same column names, types, and order) with the exception that some columns are defined to repeat.

Repeating fields are defined by immediately following the target-field name with [n] where n represents the maximum number of occurrences for a repeating field.

Note: All columns extracted from non-primary (M) tables should use [n] notation.

For example,

```
Section: User-Source-Tables

create_idt IDT112
sourced_from   odb:99:ssa/ssa@ora817
        Employee.EmpId (PK1)       EmpId,
        Employee.Name              EmpName,
        Address.CompanyId          CompanyId [6],
        Address.Address            Address   [6],
        Phone.Num                  Phone     [6]
join_by
        Employee.EmpId = Address.EmpFK,
        Address.CompanyId = Phone.CompanyId
sync
```

Flattening Process

Denormalized rows are read using the Denormalized-View and then sorted by the flattening keys.

The sorted data is used to build IDT rows. Data is moved from denormalized rows into an IDT row. An IDT row is written out either

- at the break (change) of a flattening key value, or
- when the IDT row is full

The latter will occur when a repeating field is full. If more data arrives with the same key value, additional IDT rows are built with the same key value.

The construction phase also verifies that non-repeating fields have the same value in all denormalized rows because it is not possible to store more than one value. An incorrect design or selection of the flattening keys can result in this situation. If this occurs the load will issue warnings similar to this:

```
Warning: Illegal break in denormalized record n, field f: 'xxx'
```

Flattening process uses the default values for threads and memory that are built into sorting routine. User can set the environment variables with appropriate values, in order to get better load/sort performance.

For example, to set 16 threads and a 1 GB sort buffer

```
SSASORTOPTS=:sortnthreads+16
SSASORTMEM=1g
```

Environment Variables

SSASORTOPTS

The above variable is to define the sorting options that can be used.

SSASORTMEM

This is used to define the maximum amount of memory to be used by IIR-SORT. It accepts the sizes in bytes/KB/MB/GB.

```
SSASORTMEM=n[k|m|g]
```

n represents the number of allocation units

k Kilo-bytes (KB)

m Mega-bytes (MB)

g Giga-bytes (GB)

Note that the actual size of each extent is actually one less that the size nominated by this parameter.

Flattening Options

Options for controlling how data is packed into repeating fields can be specified with the `IDT-Definition's OPTION=` parameter:

Flat-Keep-Blanks

By default, blank fields are not moved into a repeating group. This option keeps blank fields, thereby maintaining positionality between the groups, allowing one to infer that, for example, the `nth Company_ Id` is related to the `nth Address`. This option requires the same number of repeats to be defined for all repeating groups.

Flat-Remove-Identical

By default, identical values in a repeating field are kept, maintaining positionality. With this option enabled, duplicate values are removed, so that each repeating field can be sized according to the number of unique values it is expected to hold. This option does not maintain positionality.

Removal of identical values only applies to the current IDT row under construction. If a repeating field overflows causing another IDT row to be created with the same key values, the values stored in the previous row are not considered when searching for duplicates.

Tuning / Load Statistics

The Table Loader produces statistics that can be used to select appropriate values for repeat counts. For example:

```
Flatten:.............................  Denormalized-In  4196
Flatten:                               Unique-Keys      2094
Flatten:                               IDT Out          2894
Flatten:                               alt_ent_num      [ 2]
Flatten:.............................                   [ 0]   1585  75.69%
Flatten:                                                      [ 1]   363
93.03%
Flatten:                                                      [ 2]       71
96.42%
Flatten:                                                      [ 3]      24
97.56%
Flatten:.............................                 [ 4]      14     98.23%
Flatten:                                                    [ 5]     9
98.66%
Flatten:                                                    [ 6]     9
99.09%
Flatten:                                                    [ 7]     3
99.24%
Flatten:.............................                 [ 8]     2   99.33%
Flatten:                                                    [ 9]     7
99.67%
Flatten:                                                      [ 10]    3
99.81%
Flatten:                                                    [ 11]    0
99.81%
Flatten:.............................                 [ 12]    0    99.81%
Flatten:                                                  [ 13]      0
99.81%
Flatten:                                                    [ 14]      0
99.81%
Flatten:                                                  [ 15]    2   99.90%
Flatten:.............................                 [ 16]    0     99.90%
Flatten:                                                    [ 17]    0
99.90%
Flatten:                                                    [ 18]      1
99.95%
Flatten:                                                      [ 19]      0
99.95%
Flatten:.............................                 [ 20]    0    99.95%
Flatten:                                                    [ 21]      0
99.95%
Flatten:                                                    [ 22]      0
99.95%
Flatten:                                                     [ 23]      0
99.95%
Flatten:.............................                 [ 24]    0     99.95%
Flatten:                                                    [ 25]    0
99.95%
Flatten:                                               [ 26]    0     99.95%
Flatten:                                               [ 27]    1    100.00%
```

This report shows that 4196 rows were created as a result of the denormalization (SQL join) process, while 2094 of those rows contained unique keys (`FLATTEN_KEYS=`). After flattening, 2894 rows were loaded to the IDT. This means that 800 rows overflowed to secondary rows because the number of repeats was insufficient to hold all occurrences.

The next part of the report shows a histogram for each repeating field. The field `alt_ent_num` was defined to have two repeats ([2]) for this load test. The histogram tabulates the actual number of repeats in each normalized row prior to flattening and a cumulative percentage of rows. For example, the line

```
Flatten: [ 1] 363 93.03%
```

states that 363 rows have only one occurrence, and this accounts for 93% of the total rows. The table also tells us that the maximum number of occurrences was 27 and that 1585 rows had no value for this field.

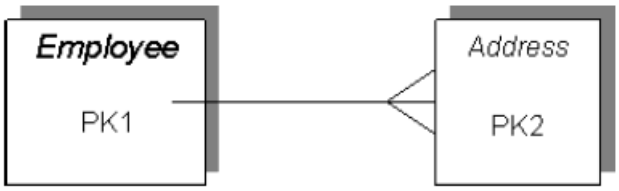
The cumulative percentage can be used to select an appropriate value for the number of repeats. For example, if `alt_ent_num` were defined to have six repeats, over 99% of values would fit into one IDT row (no overflows).

Design Considerations

Synchronization

Flattening effects the selection of PK field(s). An unflattened IDT created from a 1:M relationship normally has a compound primary key constructed from a key field from the 1 table concatenated with a key field from the M table to ensure a unique PK. For example,

Assume the following rows were created by denormalization:



Emp1 Addr1

Emp1 Addr2

Emp1 Addr3

If these rows are flattened into a single row, Emp could serve as a unique primary key:

Emp1 Addr1 Addr2 Addr3

If the number of address field repeats in the flattened table was set to [2], the flattening would produce two rows (due to the overflow), rendering Emp unsuitable to be a unique PK.

Emp1 Addr1 Addr2

Emp1 Addr3

Unless you can guarantee that the number of repeating fields are adequate to hold all occurrences of the repeating data, you should define the PK with a Sync Level of `Allow_Duplicate_PK`. Flattening will improve synchronization performance (even when duplicates are allowed) by reducing the number of rows processed for a particular key value.

Matching on Multiple Repeats

Matching on multiple repeating fields can be problematic if there are overflow records. For example, suppose we have one Identity (ID= I1). This identity has three alias names (N1, N2 and N3) and three addresses (A1, A2 and A3).

Full denormalization (without flattening) will produce 9 rows:

```
I1 N1 A1
I1 N1 A2
I1 N1 A3
I1 N2 A1
I1 N2 A2
```

```
I1 N2 A3
I1 N3 A1
I1 N3 A2
I1 N3 A3
```

If you define the IDT layout as *Name [3]* and *Addr [3]* , you will get one flattened row:

```
I1 N1 N2 N3 A1 A2 A3
```

If you define the IDT as *Name [2]* and *Addr [2]* you will get two rows due to the overflow:

```
I1 N1 N2 A1 A2
I1 N3 A3
```

The above layout is a problem if you search on *N3* and match on *N3 + A1* because those values are in different rows. The solution is an IDT layout with *ID* , *Name* , *Addr [3]* which provides:

```
I1 N1 A1 A2 A3
I1 N2 A1 A2 A3
I1 N3 A1 A2 A3
```

This is still an improvement over 9 rows. When only one matching field repeats, it is also valid to use overflows: *ID* , *Name* , *Addr [2]* which provides:

```
I1 N1 A1 A2
I1 N1 A3
I1 N2 A1 A2
I1 N2 A3
I1 N3 A1 A2
I1 N3 A3
```

So the bottom line is that you have to design your IDT carefully. Do not attempt to match two repeating fields unless you can guarantee that all occurrences will be in one flattened row.

CHAPTER 4

Link Tables

This section describes about Link Tables.

Concepts

An Identity Link Table (IDL) is an SQL accessible table that contains information about the relationship (links) between rows of an IDT. A link has directionality. That is, a parent row is linked to a child row. The strength of the link is defined by a score (0 to 100).

Defining

Links are established by a search and matching process. A `Multi-Search-Definition` is used to define the

- name of the Link Table (`IDL-NAME=`),
- search(es) that are used to establish the relationships,
- options to control the content of the Link Table.

For example,

```
Multi-Search-Definition
*=======================
NAME=                      search-link
IDL-NAME=                  IDL05
IDT-NAME=                  IDT05
SEARCH-LIST=              search-1,search-addr
OPTIONS=                  Full-Search, Link-PK
*
```

Creation Options

Link-PK adds the PK field values from the parent and child records to the Link Table. Refer to section *Layout* for details.

Link-Hard-Limit prevents adding rejected candidates to the Link Table. The default is to add all candidates to the table.

Link-Self adds links for rows that can find themselves. By default, links are not stored for these rows.

Layout

A standard Link Table contains the following columns:

Column Name	Meaning
RECID	A unique record ID for this Link Record
IDS_PARENT	The RECID of the parent IDT row

Column Name	Meaning
IDS_CHILD	The RECID of the child IDT row
IDS_SCORE	Result of scoring parent with child (0-100)

The default table only stores the RECIDs of IDT rows because they are guaranteed to be unique.

The Multi-Search Option `Link-PK` instructs IIR to inject the PK columns of the parent and child rows into the IDL as well. The parent and child column names are prefixed by `IDS_P_`and `IDS_C_` respectively.

Creating

A Link Table is created from the Console with **Tools** > **Link Table**, or by running `relate` with the `-xl` switch.

Multiple-threads are supported, so you can create the IDL with `-n3` for example. The Synchronizer does not maintain the relationships in Link Tables.

CHAPTER 5

Loading a System

This chapter includes the following topics:

Overview

This section describes the steps necessary to create and maintain a new System. These steps are carried out through the IIR Console.

To use the Console Client, the Console Server must be running, see the *OPERATIONS guide, Starting the Console Server section*.

System States

A system has three main states:

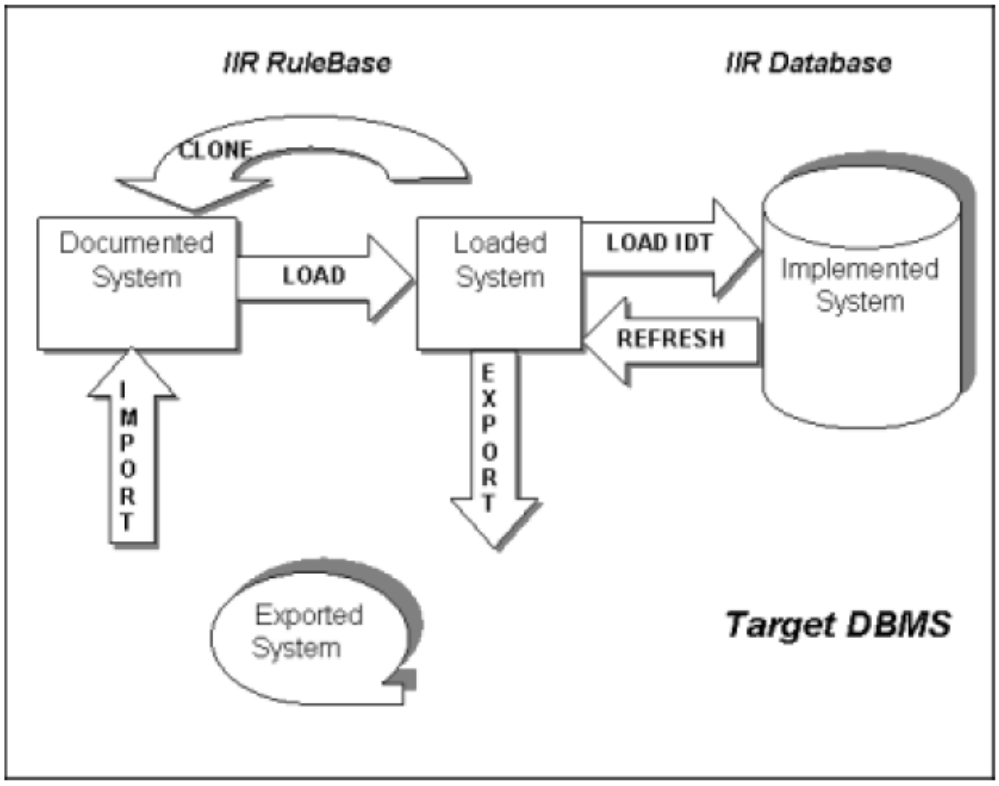
- Documented
- Loaded
- Implemented

A **Documented System** is stored in the Rulebase in unparsed form. It holds a copy of the definitions read from an SDF file or cloned from an existing system. It has not been parsed yet, so the System rules can be incomplete or may contain syntax errors.

A Documented System is converted into a **Loaded System** by parsing it (**System** > **Load** button). The parsing process will check the syntax and semantics of the rules and convert them to an executable form.

The executable rules are stored in the Rulebase ready for use by other IIR processes such as the Table Loader, Search Server and Update Synchronizer.

An **Implemented System** is one that has been physically implemented on the target database. This means that the rules stored in the Rulebase have been executed by the Table Loader to create IDTs, IDXs and IDLs. To achieve this, click **System Load IDT** button. An Implemented System is available for searching and synchronization.



Creating a System

You can create a system from an SDF or clone a system that you currently loaded. You can also launch the SDF Wizard and create a system in the SDF Wizard.

To create a system, in the Identity Resolution Console, click **System** > **New**, and select one of the following options:

Create a System from an SDF

This option reads an SDF and loads it as a document. This option parses the SDF to create executable rules. Provide the following values:

- Name and path of the SDF. Store the SDF on a file system that is accessible from the computer where the Console Server runs. If you do not specify the path, the default directory for the SDF is `SSAWORKDIR`.
- System Name. The value must be the same as the name defined in the SDF.
- Identity Resolution Database connection information. The information defines the physical database that implements the system.

Clone the current System

You can clone a system that you currently loaded and create a system. This option copies the parsed rules and changes the system name to a user-specified value. This option does not make any modifications to the rules. Use the System Editor to customize the system before parsing and loading it.

Create SDF

This option launches the SDF Wizard. Use the SDF Wizard to create a system from scratch. For more information about the SDF Wizard, see the SDF Wizard section.

Editing a System

IIR provides several mechanisms to edit your System.

SDF Wizard

The SDF Wizard provides a simple user interface designed to help new users create a System Definition File (SDF) with a minimum of effort and knowledge. The aim of the utility is to make the definition of commonly used features as simple as possible, by hiding complexity. Thus, some advanced features of an SDF cannot be defined using the wizard.

A System may be created using a data-driven or search-driven approach. A data-driven approach starts by defining the underlying table that will be searched, followed by search-strategies. Conversely, a search-driven approach defines the search requirements first, followed by the tables and indexes containing the search data. Naturally, all necessary search parameters cannot be fully defined without reference to the underlying data, so a search-driven approach requires the Search object to be revisited once the data objects have been defined. We recommend using a data-driven approach due to its elegance and simplicity.

To assist new users that are unfamiliar with Systems, the wizard contains cheat-sheets that provide step-by-step instructions for creating the necessary objects to support your search requirements. Cheatsheets are accessible through the **Help** menu.

Note: An SDF cannot be edited with the SDF Wizard if it has been modified by anything other than the SDF Wizard. A checksum in the generated SDF is used to detect changes made by an external editor.

GUI System Editor

The System Edit or provides an easy to use interface to change the rules that define a System. It is sensitive to the state of objects and prevents editing an object's definition when it has already been implemented. For example the layout of an IDT cannot be changed if it has been loaded. Similarly, you cannot change key building rules for an IDX if it is already loaded, nor can you change the Search- Definition(s) used to load a Link Table.

Note: The System Editor can only be used to edit systems created using the Console. A System loaded using low-level batch utilities cannot be edited.

System Template

The GUI System Editor uses a system template to add new components (IDTs, IDXs, Searches, etc). A system template is provided in the Server Installation in `ids/systems/system_template.sys`. A new template is shipped with every Fix CD. The console performs verification of the template when the console is started or when a new Rulebase is created. If the current template is not up to date then the template will be imported from the new template file. The template must not be edited.

Starting

The System Editor is invoked from the Console Client's **System** > **Edit** button. The Editor works on a **copy of the rules** for a given System. If you have ended a previous editing session without Loading the rules, or the Load failed, the Editor will prompt you to either:

- Keep previous changes and continue editing, or
- Discard previous changes and restart editing

Editing

A tree structure is used to navigate to the object you wish to edit. Objects that belong in a logical hierarchy will appear in the tree structure in two places:

- as an individual node attached to the root of the tree, and
- as part of a hierarchy.

For example, IDXs are attached to their parent IDT and also to the root of the tree. The individual nodes attached to the root may be edited, cloned, etc. The nodes in the hierarchical tree structure are present nly to highlight the relationship between objects and can not be modified.

If you make any changes to an object, you must either "Save Changes" or "Discard Changes" before moving to another object. Saving and discarding only effects the copy of the rules.

The copy of the rules is not parsed until you click the **Load** button. A successful Load process will replace the existing rules with the copy and deletes the copy of the rules. If the Load process fails, the original system remains in effect and the copy of the rules you have been editing is available for correction by restarting the Editor.

The **Close** button will exit the editor without saving the most recent changes you have made.

Cloning and Adding

An existing object can be cloned to create a new object of the same type. You will be prompted to name the new object after which you can edit it.

If you wish to add a new object that does not already exist, you will need to use the **Add** button. Click the **System** name in the tree structure and then click the **Add** button. Select the type of object you wish to add and enter a name for it.

Help

Help is available as tool tips. Hover the mouse pointer over the item that you want help for and a tool tip appears.

Advanced Options

By default, the Editor display only the most common options that the average user will want to see. If you wish to use some of the more advanced options, click the **Show Advanced** button will enable access to all options for a given object.

Rulebase Editor

The Rulebase Editor provides a low-level interface for the maintenance of rules stored in the Rulebase.

To start from the Console Client, click **Rulebase** > **Edit**.

Note: The Rulebase Editor is intended only for use by an Informatica Corporation Consultant. **Incorrect use may damage your Rulebase**.

Implementing a System

This step implements the parsed rules by running the IIR Table Loader to:

- extract and denormalize data from the User Source Tables
- create triggers on the User Source Tables (for synchronized systems)
- apply transform rules
- generate keys
- mass load the IDT and its associated IDXs.

▶ To implement a System, select **System** > **Load IDT**. You will be prompted for the name of the Table Loader Definition to execute.
Refer to the *OPERATIONS guide, IIR Table Loader chapter* for detailed information about the Table Loader.

To un-implement a System

A System can be *unimplemented* if required. This process deletes the database objects that were created when the system was implemented. It will delete the following:

- IDTs
- IDXs
- IDLs
- triggers
- transactions stored in the Synchronizer's Transaction Table

▶ To *un-implement* a System, select **System** > **Refresh**.

System Status

Setting a System's `Status` attribute controls access rights to its `rules`. This mechanism can be used to prevent accidental changes to the rules. A successful Load of the ID-Tables will automatically change the System status to `Locked` to prevent accidental changes to an implemented System.

To change the status, using the IIR Console, select **System** > **Status** button.

The status values in order of increasing restrictiveness are:

- Build (least restrictive)
- Test
- Production
- Locked
- Prototype (most restrictive)

The following table describes the restrictions imposed by these values in terms of the ability to read, update, delete or run a System with a given status.

Status	Description	Read	Upd	Del	Run
Build	Under construction - unusable	yes	yes	yes	no
Test	Uncontrolled - any use	yes	yes	yes	yes
Production	Relaxed Production	yes	yes	no	yes
Locked	Frozen production	yes	no	no	yes
Prototype	Secured prototype - read-only use	yes	no	no	no

The System Loader cannot remove a System from the IIR Rulebase unless the System has a status that allows "deletion" (that is `build` or `test`).

If a status does not have "run" privilege the Search Server will not get access to the Rulebase and therefore searches cannot be performed.

Without the "update" privilege (that is `prototype` or `locked`), utilities such as the Table Loader cannot be run.

Prototype systems cannot be changed in any way apart from being copied. Informatica Corporation may ship sample systems with this status. Customers should not set their systems to prototype status, as they can not be altered in any way, including changing the status.

The status of a System only applies protection to the System's rules stored in the IIR Rulebase. It does not apply protection to the IIR Identity Table and Indexes, which can still be affected by IIR utilities such as **dbinit**.

System Backup / Transfer

A System's rules can be written to an operating system file. This file can be used as a backup, or as a mechanism to transfer the System to another Rulebase.

Exporting a System will write the *parsed rules* to a file. Therefore only a Loaded System should be exported. Exporting a Documented System will result in an incomplete set of rules.

Export

- To export the system definition, select **System** > **Export**.

This will prompt for the name of a file to contain the exported system. The exported system contains all of the System rules.

Import

1. To import a System, select **System** > **New**. This will prompt for the name of a file that contains the system to be imported.

2. Select the **Import System** radio button.

   This will prompt for the name of a file that contains the system to be imported.

   The file must have been created using the **System** > **Export** function using a Rulebase Server of the same version that will be used to import it.

   Before the import system option is used, it must be Loaded. Use either **System** > **Load**

   or, **System** > **Edit** and then **Save & Load**.

   Once this step is completed, it is possible to Load the ID-Tables.

CHAPTER 6

Static Clustering

This chapter includes the following topics:

Overview

Once data have been loaded into an IDT, it may be *clustered* . This is the process of grouping like rows from a *static copy* of the IDT. For example, you may cluster by name in order to identify duplicates, or you may wish to cluster by name and address to identify "households". The data are extracted from the IDT at the start of the clustering process, grouped, and a report is produced. The clusters are then discarded.

The columns to cluster on, and the search and match strategies used, are all user-definable. However, only columns containing character data (*C* , *F* ,*V* , *N* , *R* and *W*) can be used for searching and matching. All columns in the IDT are available to be displayed in the final cluster report.

Process

This section describes about the process that are supported.

Define a Search

Use the Console's GUI System Editor to define a Search-Definition. The search and match strategy used in the clustering process is defined by selecting the appropriate search options.

Load IDT

The IDT cannot be clustered unless data has been loaded into it. To start the Table Loader utility, select the **System** > **Load IDT** button.

Run Clustering

Click the **Tools** > **Run Clustering** button. Select the search to use and specify names for the output report(s) to be generated. An "all clusters" report contains single-member clusters (*Singles*), as well as multi-member clusters (*Plurals*). You may want to generate reports containing only Singles or only Plurals. Specifying a report file name for a particular option enables report creation for that option. A report is not created if the file name is left blank.

Maximum DB Size Select the maximum database size for this static clustering. The default is 32 GB. Note that the created database will not allocate the chosen amount of disk space but the size will grow as required.

Maximum Sort Memory Select the maximum amount of memory to be used by SSA-SORT. The default is 1 GB.

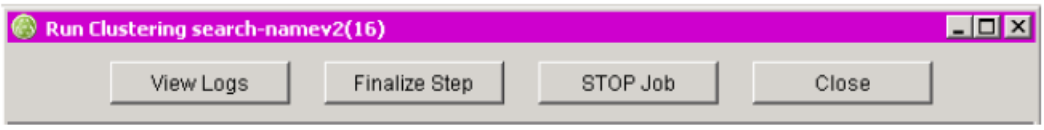
Maximum Cache Size Select size of the memory cache established by SSA-DB. The default size is 1 GB.

Finalize Clustering Step

The clustering process can only be stopped in between the steps or during the cluster phase itself, not during other steps such as loading keys, sorting, etc.

This can be useful if you want to stop the clustering process before it finishes. Hence, you can view the intermediate results and assess its performance.

The **Finalize Step** button will be enabled once the clustering process starts. Click the **Finalize Step** button causes clustering process to finalize and execute next step which is post job process, so that you may view the intermediate results of clustering.



View Reports

Once the clustering has completed, you may view the cluster report using the **Tools** > **Report Viewer**, or right-click the report file name in the list of Job output files in the log viewer.

CHAPTER 7

Simple Search

This chapter includes the following topics:

Simple Search Overview

Simple search refers to a type of search that you can perform across multiple field types, such as people, organizations, or addresses. You can search on one or more fields at a time.

You may not know the type of search information or want to search for information across multiple field types in some systems. In such situations, applications can generate a generic index for all Person Names, Organization Names, and Address that make up the searchable data domain. You can use the index to process search across entities.

Simple Search Requirements

Data that forms the domain of the search can contain multiple types of fields. Simple search uses SSA-NAME3 for operations. SSA-NAME3 engine indexes data from each field type and stores for subsequent use by search processes. Using Standard Populations, you can set up an application to index and search on the following three field types:

- Person Names
- Organization Names
- Addresses

For Simple Search, a population that supports `Generic_Field` for keys and search strategies, and `Generic` match purpose is required.

Generic_Field

The `Generic_Field` is used to index Person Names, Organization Names, Addresses, or combination of these, while the `Generic` match purpose can score records match.

The SSA-NAME3 algorithm that builds the keys and the search ranges for Generic Data is invoked by calling SSA-NAME3 and by passing `FIELD=Generic_Field` in the Controls parameter of `get_keys`.

The `Generic_Field` algorithm is designed to overcome some of the errors and variation that are common across entities such as Person Names, Organization Names, Addresses, and Addresses.

Generic Match Purpose

Use `Generic` Match Purpose to match candidates to the search information. `Generic` Match Purpose matches general non-specific data. The required field to match candidates to the search information is `Generic_Field`.

Simple Search Definition

You can define a simple search with a regular search definition in a system definition file.

Use the SDF Wizard to create a simple search definition. However, you can also create a definition outside the SDF Wizard subject to the rules for each of the components in the system definition file.

Index Definition

You can use any index that uses the indexing algorithm `Generic_Field` as Generic index. Normally, generic indexes include all columns that need to be searchable using simple search. A column in a Generic index must share a common IDT datatype.

Search Definition

Any search based on a generic index can potentially behave as a simple search. The limit on number of characters that make up a search name is reduced by three for simple searches.

Search Logic

Search fields that make up the simple search must contain all fields used by the associated generic index. Only the `Generic_Field` algorithm may be used in search logic of a simple search. The CONTROLS must specify a combine option for all search fields with no delimiters between them.

For example, `Controls ("FIELD=Generic_Field SEARCH_LEVEL=Typical COMBINE=Generic_Field:DELIM-NONE")`

Score Logic

Score fields that make up the simple search must contain all fields that the associated generic index uses. You can use only the `Generic` purpose in score logic of a simple search. The CONTROLS must specify a combine option for all search fields with no delimiters between them.

For example, Controls `("FIELD=Generic_Field SEARCH_LEVEL=Typical COMBINE=Generic_Field:DELIM-NONE")`

The match fields must be of type `Generic_Field`.

For example, `Matching-Fields ("FNAME:Generic_Field,LNAME:Generic_Field,ORG:Generic_Field,ADDR:Generic_Field")`

Processing View

Simple search requires a special view definition that is used for search input processing. The processing view must have a name that contains the search name and the suffix `-GP`.

For example, if the search name is `my_search1`, the view name must be `my_searcg1-GP`

The processing view must contain all fields required to assemble the simple search.

Display View

Simple search requires a special view definition that is used for search fields display. The display view must have a name with the characters `-GD` as a suffix to search name.

For example, if the search name is `my_search1`, the view name must be `my_searcg1-GD`.

The display view must contain a single field of size that is the total size of fields used in the processing view. The name of the field must not be an existing IDT column. The field name chosen for display view appears as a field when viewed through search clients. When using API `search_search_layout`, the name of field appears as a search column in the search layout.

Note: To construct search input, it may be necessary to get attributes such as lengths and offsets of search columns. Such details should be obtained using `search_search_layout`. In case of simple search, the search fields will not be a part of the IDT and therefore IDT layout should not be used to determine lengths and offsets.

Simple Search Scenario

You can create a search definition based on the requirements.

For example, you have the CUSTOMER database table with the following columns:

Name	Datatype	Length
CUSTOMER_ID	NUMBER	10
CUSTOMER _NAME	VARCHAR	32
CUSTOMER _ADDRESS	VARCHAR	32
CUSTOMER _ORG	VARCHAR	32

To provide search capability on Customer Name, Customer Address and Customer Company, you can create a search definition with the `Generic_Field` in KEY LOGIC and with the `Generic` purpose in the SCORE LOGIC.

You can create the following search-definition:

```
 search-definition
 NAME= simple_search
 IDX= my_idx
 SEARCH-LOGIC= SSA,
               System (my_system),
               Population (usa),
               Controls ("FIELD=Generic_Field SEARCH_LEVEL=Typical
COMBINE=Generic_Field:DELIM-NONE"),
               Field ("CUST_NAME, CUST_ADDRESS,CUST_ORG")
 SCORE-LOGIC=  SSA,
```

```
               System (my_system),
               Population (usa),
               Controls ("PURPOSE=Generic MATCH_LEVEL=Typical
COMBINE=Generic_Field:DELIM-NONE"),
               Matching-Fields ("CUST_NAME:Generic_Field, CUST_ADDRESS:Generic_Field,
CUST_ORG:Generic_Field")
*
```

CHAPTER 8

Search Performance

A search client calls the Identity Resolution with a search record. The Server retrieves a set of similar records from the database (known as the candidate-set).

Each candidate is scored against the search record to determine the degree of similarity. Records that score above the score-threshold defined in the Search-Definition are returned to the search client.

The process may be optimized by:

- reducing the size of the candidate-set, thereby reducing the amount of scoring required, and/or
- reducing the cost of scoring two records, and/or
- reducing the size of the IDX to improve database cache efficiency

The following sections discuss ways in which to achieve these goals.

Reducing Candidate Set Size

Flattening

This feature is used to reduce the size of the candidate set and the size of the IDX. Refer to the Flattening IDTs section in this guide.

Partitions

This feature is used to reduce the size of the candidate set.

For very large files, the key generated from the `KEY-FIELD` may have a high selectivity due to the sheer volume of data present on the file. Therefore searching for candidates using the key will create very large candidate sets.

If the nature of the data is well understood, it may be possible to qualify the key with additional data from the record so that the "qualified key" becomes more selective.

The `PARTITION` option instructs IIR to build a concatenated key from the Key-Logic `FIELD` and up to five fields/sub-fields taken from the IDT record. The partition information forms a high-order qualifier for the key (it is prefixed to the key).

For example, an application may wish to search all names in a telephone directory. If we are willing to only examine candidates from a particular region, we could partition the data using a post-code or some other information that can group the candidates into regions. Performance is improved by reducing the size of candidate sets. The disadvantage is that candidates will only be selected from within regions; not outside. If this makes sense from the perspective of the "business problem" being solved then partitioning can be used.

SQL Filters

SQL Filters may be used to reduce the size of the candidate set. They are evaluated by the DBMS as candidates are selected. By removing candidates within the DBMS engine, we reduce network overheads and matching costs within the Search Server.

An SQL Filter is a DBMS-specific SQL expression that is appended to the query that selects candidate rows from the IDX/IDT. It can only be used to remove candidates.

For example, a particular Search-Definition may perform a fuzzy search using a Name. Without filters, all candidates are returned to the Search Server to be matched against the search record. If a filter was specified requesting `STATE = 'NY'`, only candidates living in New York would be returned for matching. Although the same effect can be achieved using a partition, SQL filters provide more flexibility.

The SQL may be provided statically using the `FILTER=` parameter in the `Search-Definition` or at run-time with the `ids_search_filter` API. The Search-Definition options `Filter-Append`and `Filter-Replace` are used to control whether, and how dynamic filters will operate.

The DBMS needs access to native columns in the IDT in order to evaluate the expression in the filter. Therefore the columns mentioned in a filter must only refer to the columns available in the IDT. Note that source column names are prefixed with `"IDS_"` in the IDT.

A search that uses a filter will access both the IDX and IDT, performing significantly more database I/O than a standard search that only accesses the IDX. Therefore a filter-based search may be less efficient (slower), especially if it selects a large percentage of the IDX (which may result in full-index and/or full-table-scans of the IDX and IDT respectively). There is a balance point, beyond which the extra I/O is more expensive than performing a standard search and scoring all candidates. Customers should evaluate where this point lies based on their particular data and search requirements.

Note: Lite Index cannot support a search filter.

Performance Tips

If an IDX will always be used in conjunction with a filter, and will never be used without one, specify `--Full-Key-Data` when creating the IDX. This will reduce its size and improve database cache efficiency. If full-index scans and the IDX are performed (due to wide searches), consider creating a dedicated IDX specifying `--Full-Key-Data` to reduce its size.

If the filter clause is of the form "column = value", consider creating a partition instead of using an SQL filter.

Consider creating a database index on the column(s) specified in the filter clause. Remember to collect optimizer statistics for the new index, and consider creating a frequency histogram (on Oracle), if the index values are not evenly distributed.

Static Filters

A static filter is defined in a Search-Definition with the `FILTER=` parameter. It can be overridden at run-time only if the Search Option `Filter-Replace` has been specified.

Dynamic Filters

Dynamic filters are provided with the `ids_search_filter` API. If the `Search-Definition` specifies the `Filter-Replace` option, the dynamic filter will replace the filter specified in the Search- Definition (if one was defined).

If the `Search-Definition` specifies the `Filter-Append` option, the dynamic filter will be appended to the filter specified in the Search-Definition (if one was defined).

Skeletal Filters

Skeletal filters are a combination of static and dynamic filters. If the SQL defined in the `FILTER=` parameter contains any substitution variables, a dynamic filter is used to provide values to be substituted into the SQL at run-time. Substitution variables are of the form `$n`, where `n`= 1, 2,. . .

For example: `FILTER= "IDS_STATE = '$1'"`

defines a skeletal filter containing one substitution variable (`$1`).

The API function `ids_search_filter` must be called before the search is started in order to provide values to be substituted into the skeletal SQL. Values remain in effect until the caller provides new values or switches to a new search.

The API call must provide a delimited string that contains enough values to satisfy the skeletal SQL's requirements. The first character in the string defines the delimiter character used to separate values in the string.

For example: `|NY|`

defines `|` as the delimiter and `NY` as the first value. Note that the string must be terminated with the delimiter character. The effective filter (after substitution) is changed to, `IDS_STATE = 'NY'`

Multi-Search

Dynamic and skeletal filters defined with `ids_search_filter` have an order of precedence when used in a Multi-Search.

IIR will use a filter defined for a sub-search (if present) before using a filter defined for the Multi-Search.

Using PL/SQL and User-Defined Functions

As noted earlier, a search that uses an SQL Filter will read both the IDX and IDT. The latter is necessary to access values of columns that appear in the WHERE predicate. Normally, the redundant data held in the IDX are stored in a proprietary compressed format and are inaccessible to the DBMS engine. However, if the redundant data were uncompressed, improved performance would result since IDX rows alone are sufficient to evaluate the predicate, removing the need to read IDT rows.

An alternative SQL Filter mechanism is available for just such a case. It requires that:

- redundant data stored in the IDX are uncompressed. This is achieved by specifying the `--Full-Key-Data`option to remove the redundant (compressed) IDT data from the IDX row, and by specifying the `Key-Data=`parameter to store up to 5 uncompressed IDT columns in the IDX row (immediately following the 8-byte fuzzy key).
- the creation of a PL/SQL or User-Defined Function that, given an IDX row, can examine the Key-Data to decide whether the particular row should be returned as a candidate for further matching,
- an SQL Filter clause that calls the function (passing the IDX row) and interprets the result appropriately.

For example:

The following IDX Definition disables Full-Key-Data (which stores all IDT columns in the IDX) and specifies a list of IDT columns to be appended after the 8-byte fuzzy key (Name and State). Up to 5 columns (or part thereof) may be specified with the Key-Data parameter, as long as the total key length does not exceed the maximum key size permitted by the particular DBMS (~ 250 bytes).

```
idx-definition
*==================
NAME=                namev2
ID=                  u5
IDT-NAME=          IDT305
KEY-LOGIC=          SSA, System(default), Population(test),
```

```
                    Controls ("FIELD=Person_Name KEY_LEVEL=Standard"),
                    Field("name")
KEY-DATA=          Name,State
OPTIONS=          --Full-Key-Data
```

The full syntax for specifying Key-Data is: `KEY-DATA = field[,length,offset], ...`

where `field` is the field name, `length` is an optional number of bytes to include (the entire field is included by default), and `offset` is an optional starting offset (base 0).

The Search-Definition includes a filter clause that calls a user-written function. The function call must include a field named `SSAKEY` to provide the function with access to the Key-Data. The response code returned by the function determines whether the row is kept or discarded. For example:

```
search-definition
*====================
NAME=                  search-filter-ny
IDX=                     namev2
KEY-LOGIC=             SSA, System(default), Population(test),
                         Controls ("FIELD=Person_Name SEARCH_LEVEL=typical"),
                         Field("name")
KEY-SCORE-LOGIC=     SSA, System(default), Population(test),
                         Controls("PURPOSE=Person_Name MATCH_LEVEL=Typical"),
                         Matching-Fields("Name:person_name")
FILTER=                  "#ids_sql_filter.demo(SSAKEY)=1"
OPTIONS=               Filter-Replace
```

Note that the filter statement:

- begins with '#'. This character informs the search engine to use the new semantics, otherwisethe original filter mechanism is used.
- passes the entire IDX record (`SSAKEY`) to the function named `ids_sql_filter.demo`. The function examines the Key-Data and returns a decision to either keep or discard the row. The response-code is arbitrary. In this example, a value of 1 will keep the candidate row.

The corresponding function, written in PL/SQL, is as follows.

```
CREATE OR REPLACE PACKAGE BODY ids_sql_filter AS
         FUNCTION demo (
                     ssakey     IN RAW)
                     RETURN     NUMBER
         IS
                     rec              VARCHAR (255);
                     state          VARCHAR (2);
         BEGIN
                     rec := utl_raw.cast_to_varchar2 (ssakey);
                     state := SUBSTR(rec,33,2);
                     IF (state = 'NY') THEN
                               RETURN 1;
                     END IF;
                     RETURN 0;
        END demo;
END ids_sql_filter;
```

The sample function performs the following actions:

- converts the input parameter `ssakey` from raw to char, which allows the Key-Data to be extracted using a substring function.
- extracts the value of State from the Key-Data. In this case, the Key-Data contains an 8-byte fuzzy-key, followed by a Name (for a length of 24 bytes), followed by State (2 bytes). Thus, the State field begins at offset 32 (base 0). The Oracle substring function assumes offsets start from 1. Therefore the substring command refers to position 33 for a length of 2 bytes to exact the value of State.

- The extracted State value is compared to a value of 'NY'. When the candidate value is identical, the function returns 1, meaning that the candidate row should be kept. Otherwise, it returns 0, which effectively discards the candidate row.

Note that the value of `NY` does not need to be hard-coded. It could have been passed in as the second parameter to the function by hardcoding it in the filter statement, or even set dynamically by using a substitution variable. For example,

```
FILTER= "#ids_sql_filter.demo(SSAKEY,$1)=1"
```

Note: When using this alternative SQL Filter method, only the Key-Data fields will be available for matching, display, and return as part of the search results.

Reducing Scoring Costs

This section describes about Pre-Scoring.

Pre-Scoring

This feature is used to reduce the cost of scoring two records.

The most expensive part of the matching process is the scoring. Once a set of candidate records is selected each candidate is scored against the search record. In practice, the scoring is often complex in nature and involves several methods scoring several fields on the search and file records.

If it is possible to quickly reject a candidate record by scoring it with a "light weight" (inexpensive) scoring scheme we can avoid the need to call a more complex scheme.

The Search-Definition's `PRE-SCORE-LOGIC` is used to define an optional "light weight" scoring scheme which is processed before the normal `SCORE-LOGIC`. If it returns a "reject" condition, the more expensive `SCORE-LOGIC` phase is not called.

Scoring Phases

Scoring happens in two distinct phases known as Pre Scoring and Scoring. There are three possible outcomes as a result of the score in each phase. A record may be:

- rejected, or
- accepted, or
- passed to the next scoring phase.

Rejection occurs when the score is less than the `Reject` limit for that phase (limits are built into your SSA-NAME3 population but can be adjusted using the `CONTROLS` parameter in the System definition). A rejected record is not passed to any other phases. It is simply discarded.

A record is accepted when its score is greater than or equal to the `Accept` limit for that phase. It does not participate in any further scoring phases; it is simply added to the search's result set.

A record that has a score which is greater than or equal to the `Reject` limit and less than the `Accept` limit is deemed to be "undecided" and is passed to the next scoring phase. If no more phases exist the record is returned as part of the search result set.

Adjusting the Accept and Reject limits

Accept and Reject limits are defined by SSA-NAME3 V2 Population Rules. The standard Populations usually define an Accept limit to be less than or equal to 100. If a record reaches a score that is greater than or equal to the Accept limit for that phase, the record is accepted and the record does not take part any further

scoring phases as explained above. However, if the reason for using multiple scoring phases is only to reject early and never to accept until the final scoring phase, then the default Accept limit is not useful. In order to never accept records in a scoring phase, one should specify a scoring phase with the Accept limit of 101, which can be never reached. This can be done using the following syntax in the CONTROLS parameter in the Scoring phase definition:

```
Controls("Purpose=PurposeName MATCH_LEVEL=MatchLevel±AdjA±AdjR")
```

Where AdjA is the Accept level adjustment and AdjR is the Reject level adjustment. To force the Accept limit to become 101 the exact value of +101 must be used as in the following example:

```
Controls("Purpose=Address MATCH_LEVEL=Typical+101+0")
```

where the special value +101 is forcing the Accept level to become exactly 101. The Reject adjustment should also be specified. Omitting the Reject limit adjustment means that the Reject level is adjusted by the same amount as the Accept level and the value 101 would cause all records to be rejected, which is clearly not useful. By specifying the value +0 the Reject limit is not changed from the original value specified in the Population Rules.

See the *SSA-NAME3 POPULATIONS and CONTROLS* manual for further details.

Reducing Database I/O

This section describes about the IDX size and Compressed Key Data.

IDX Size

The physical size of the IDX will determine how efficiently the database cache will operate. Reducing the size of the IDXs will improve performance. This is achieved by selecting the most appropriate Compressed-Key-Data value (as described in the Compressed Key Data section) and using flattening (as described in the Flattening IDTs section) to reduce the number of rows.

Compressed Key Data

The IDX stores fuzzy keys and identity data for matching. The identity data is compressed and stored using an algorithm selected with the Identity-Table-Definition's `Compress-Method` parameter. All methods will compress the Identity data and store it in the IDX together with its fuzzy key.

If the length of the IDX record exceeds the DBMS's limit for an indexable column IIR can either,

- `Method 0` split the IDX record into multiple adjacent segments (which are all shorter than the length limit).
- `Method 1` truncate the IDX record at the length limit and only store one segment. This forces additional I/O at run time if the IDX record is selected for matching, as the matching data must be read from the IDT record.

IDX segments are fixed in length and have the following layout:

Partitions	Fuzzy SSA-NAME3 Key	Compressed Identity Data

The segment length is the sum of

- partition length (optional, user defined)
- SSA-NAME3 key length (5 or 8 bytes)
- Compress-Key-Data(n) parameter

- 4 bytes of additional overhead

The maximum segment length depends on the host DBMS:

- Oracle 255 bytes
- UDB 250 bytes

Since the segment length is fixed, choosing an appropriate value for `n` is important because it affects the total amount of space used and the I/O performance of the index. Determining an optimal value for n requires knowledge of the characteristics of the source data and how well it can be compressed.

If `n` is set too high, all segments will use more space than necessary. If `n` is too low, records will be split into multiple segments, incurring extra overhead for the duplication of the Partition and Fuzzy Key in each segment.

Measuring Compression

The IIR Table Loader can be used sample the source data and produce a histogram of the Compressed Identity Data lengths.

A sample table appears below:

```
loadit>   Histogram:.........................    KeyLen      Count Percent
loadit>   Histogram-KG:                            17          2   0.18%
loadit>   Histogram-KG:                            19         12   1.23%
loadit>   Histogram-KG:                            20         72   7.54%
loadit>   Histogram-KG:.......................     21        133  19.21%
loadit>   Histogram-KG:                            22        201  36.84%
loadit>   Histogram-KG:                            23        195  53.95%
loadit>   Histogram-KG:                            24        155  67.54%
loadit>   Histogram-KG:.......................     25        115  77.63%
loadit>   Histogram-KG:                            26         78  84.47%
loadit>   Histogram-KG:                            27         62  89.91%
loadit>   Histogram-KG:                            28         24  92.02%
loadit>   Histogram-KG:.......................     29         29  94.56%
loadit>   Histogram-KG:                            30         11  95.53%
loadit>   Histogram-KG:                            31          9  96.32%
loadit>   Histogram-KG:                            32         11  97.28%
loadit>   Histogram-KG:.......................     33          9  98.07%
loadit>   Histogram-KG:                            36          5  98.51%
loadit>   Histogram-KG:                            37          6  99.04%
loadit>   Histogram-KG:                            38          7  99.65%
loadit>   Histogram-KG:.......................     41          4 100.00%
```

`KeyLen` is the length of the Identity Data after compression

`Count` is the number of records with that length

`Percent` is the cumulative percentage of the number of records having lengths less than or equal to the current `KeyLen`

Segment Lengths

The histogram can be converted into more useful data by running

```
%SSABIN%\histkg ReportFile
```

where `ReportFile` is the name of the log file containing the IIR Table Loader output. It will produce a report similar to the one below:

```
Index Name            name
KeyData                110
CompLen                 30

   key len      count       %      bytes     comp-1     comp-2      segs
        17          2    0.18         34         78         88         2
        19         12    1.23        228        468        528        12
        20         72    7.54       1440       2808       3168        72
        21        133   19.21       2793       5187       5852       133
        22        201   36.84       4422       7839       8844       201
        23        195   53.95       4485       7605       8580       195
        24        155   67.54       3720       6045       6820       155
        25        115   77.63       2875       4485       5060       115
        26         78   84.47       2028       3042       3432        78
        27         62   89.91       1674       2418       2728        62
        28         24   92.02        672        936       1056        24
        29         29   94.56        841       1131       1276        29
        30         11   95.53        330        429        484        11
        31          9   96.32        279        351        792        18
        32         11   97.28        352        429        968        22
        33          9   98.07        297        351        792        18
        36          5   98.51        180        195        440        10
        37          6   99.04        222        234        528        12
        38          7   99.65        266        273        616        14
        41          4     100        164        156        352         8
     Total       1140              27302      44460      52404      1191

Keydata Offset          5
Key Overhead            4
Block Size           8192
Block Overhead        400

   compLen       bytes        segs segs/key     DB-bytes   DB-blocks
         1      409530       27302   23.949       545367          67
         2      223024       13939   12.227       293091          36
         3      161126        9478    8.314       209256          26
         4      130806        7267    6.375       168081          21
         5      112974        5946    5.216       143778          18
         6       99920        4996    4.382       126059          16
         7       93534        4454    3.907       117066          15
         8       84150        3825    3.355       104555          13
         9       81650        3550    3.114       100770          13
        10       81336        3389    2.973        99763          13
        11       75550        3022    2.651        92137          12
        12       69342        2667    2.339        84117          11
        13       66447        2461    2.159        80207          10
        14       66388        2371    2.080        79767          10
        15       67599        2331    2.045        80872          10
        16       69330        2311    2.027        82608          11
        17       71300        2300    2.018        84632          11
        18       73440        2295    2.013        86861          11
        19       74910        2270    1.991        88302          11
        20       74732        2198    1.928        87812          11
```

```
21      72135      2061    1.808      84505        11
22      66960      1860    1.632      78219        10
23      61605      1665    1.461      71769         9
24      57380      1510    1.325      66676         9
25      54405      1395    1.224      63064         8
26      52680      1317    1.155      60923         8
27      51455      1255    1.101      59374         8
28      51702      1231    1.080      59533         8
29      51686      1202    1.054      59394         8
30      52404      1191    1.045      60103         8
31      53190      1182    1.037      60891         8
32      53866      1171    1.027      61556         8
33      54614      1162    1.019      62304         8
34      55776      1162    1.019      63526         8
35      56938      1162    1.019      64747         8
36      57850      1157    1.015      65685         9
37      58701      1151    1.010      66555         9
38      59488      1144    1.004      67353         9
39      60632      1144    1.004      68555         9
40      61776      1144    1.004      69758         9
41      62700      1140    1.000      70713         9
```

The first section of the report summarizes the histogram.

Value	Description
IndexName	the name of the IDX
KeyData	the sum of the length of the IDT columns (the Identity Data, uncompressed)
CompLen	Compress-Key-Data(n) value
Key Len	the length of the Identity Data after compression
Count	the number of records with this `KeyLen`
Bytes	`KeyLen * Count`
Comp-1	total bytes to store `Count` records with `KeyLen` using Method 1 (1 segment only)
Comp-2	total bytes to store `Count` records with `KeyLen` using Method 0 (multiple segments)
Segs	the number of segments required to store `Count` records of `KeyLen`

The second part of the report gives space estimates for various values of Compress-Key-Data(n)

Value	Description
KeyDataOffset	length of the Fuzzy Key including any partition
KeyOverhead	the overhead associated with storing the segment on the host DBMS (assumed)
Blocksize	DBMS block size (assumed)

Value	Description
BlockOverhead	DBMS overhead when storing records within a block including control structures and padding (assumed)
compLen	`n` from Compress-Key-Data(n)
Bytes	the number of bytes required to store segments of this size
Segs	the number of segments used
Segs/Key	the average number of segments per IDX record.
DB-Bytes	the number of bytes for segment of this size (scaled up by `KeyOverhead`)
DB-Blocks	the number of blocks for segments of this size (based on the `Blocksize` and `BlockOverhead`)

To optimize performance, select the largest `compLen` value that minimizes `DB-blocks` and set `Compress-Key-Data` to this value (35 in the example above).

Note: You may need to customize the histkg script `(%SSABIN%\histkg.awk)` if the block size and block overhead values are not correct for your DBMS.

Search Statistics and Tracing

This section contains tips on diagnosing on tuning your search strategies.

Tracing a Search

The Search Server can create a detailed trace file showing:

- the search record (in IDT format)
- all candidate records selected using the defined Search-Logic
- the results of matching the search record with each candidate record

API

This can be enabled from an API program by specifying LOGTEST= in the options parameter of the `ids_system_open call`:

```
LOGTEST="<full_path_of_log_file>"
```

relate

The batch search utility relate will create a trace file when the `--3` switch in used to nominate the name of the trace file. For example,

```
%SSABIN%\relate --3%SSAWORKDIR%\mysearch.trc
```

Java Search Client

The Java Search Client can create a trace file. This is enabled / disabled with the menu item **Options** > **Create Trace File** which toggles tracing on and off.

The default trace filename is `%SSAWORKDIR%\SystemName.trc`

A different filename may be specified with **Options** > **Set Trace Filename** .

Note: The trace file is written with buffered I/O. It is not completely flushed to disk until the user closes the current search. The Search Client will do this automatically when

- tracing is turned off, or
- you switch to a different Search, or
- the application is closed.

Sample Output

An annotated trace output file is provided to demonstrate how to interpret the information found in the trace. The file is located in the Server's bin directory and is named `%SSABIN%\srchtrc.txt`

Search Statistics

If no records are found, either no candidates were found or scoring rejected all candidates. To determine which, stop the servers, delete all log files, and restart. Run a relate with -vs (statistics) and an input file with just one record. Stop the servers to flush the logs. Check the Server log file for the following:

```
apic>                  Hstogram:                              FS FindSet - unique     recs
countnt
apic>                  Histogram-FS:
1               2
apic>                  Histogram-FS:
2                183
apic>                  Histogram-FS:.............3                 57
apic>                  Histogram-FS:
4                42
apic>                  Histogram-FS:
5                16
apic>                  Histogram-FS:
6                13
apic>                  Histogram-FS:.............8                 17
apic>                  Histogram-FS:
11               24
```

This histogram shows the number of searches that found n candidates (n is the left hand column, number of searches on the right).

If you see the log as follows:

```
apic>          Histogram-FS:                0           1
```

then it means there are no candidates found. The possible causes are:

- Key-Logic for IDX does not match key-logic for search (different fields/algorithms perhaps?)

If it actually found some candidates, check for:

```
apic>                          Matches-Total:.....................KSL         1265
apic>                           Matches-Accepted:
KSL       915
apic>                          Matches-Rejected:
KSL        28
apic>                           Matches-Undecided:
KSL              32
```

This provides information on what happened during scoring. If all candidates were rejected, some possible causes are:

- score threshold too high

- bad method options

Try setting the score limit very low so that everything is accepted. Then use relate's -u and -l switches to set upper (or lower) score limits until you figure out what the real limit should be.

Expensive Searches

Relate can be used to identify expensive searches. When started with the -s switch, it will produce two histograms showing:

- search transaction duration (in 50 millisecond groups)
- result set size (groups of 20 records)

For example,

```
relate> Search Duration Histogram
relate> 0.050 sec 999 99.90%
relate> 0.100 sec 0 99.90%
relate> 0.150 sec 1 100.00%
relate> Total elapsed time: 5.547 sec
relate> Result Set Histogram
relate> 20 recs 1000 100.00%
relate> Reads 1000
relate> Writes 1386
```

This histogram shows that 999 searches were executed in less than 50 milliseconds and one search completed in less than 150ms. All searches returned less than 20 records.

The first search of the session usually incurs extra "set up" overhead, including connecting to the database, and is usually slower when compared to the other searches. This can be verified by running relate with the -ss switch which produces individual transaction timings:

```
relate> Txn                      1 0.109 sec
relate> Txn                      2 0.016 sec
relate> Txn                      3 0.000 sec
relate> Txn                      4 0.000 sec
relate> Txn                      5 0.000 sec
relate> Txn                      6 0.015 sec
relate> Txn                      7 0.000 sec
relate> Txn                      8 0.016 sec
relate> Txn                      9 0.000 sec
relate> Txn                     10 0.015 sec
```

The Search Transaction Histogram can be used to get a general feel for the cost of the search strategy.

In general, the cost is proportional to the number of candidates selected by the Search-Logic.

Individual transaction timings are useful for locating expensive searches. Once the expensive search record is found, use the methods documented under "Tracing a Search" to locate the cause.

The Result Set Histogram can be used to determine the optimal size of the sort memory (SORT= parameter of the Search-Definition).

Output View Statistics

An output view may be defined to include various statistical fields, including the number of candidates selected and matched. Refer to the Output Views section in this guide for details.

relperf

The `relperf` utility is used to generate comparative performance statistics for a specified Search using a range of search strategies (search widths and match tolerances). By comparing the number of candidates selected with the number of accepted matches, `relperf` helps to determine the most appropriate strategy for a particular search problem.

Given a representative set of search data, `relperf` runs multiple search processes using all available search widths and match tolerances and then collates and summarizes the results.

The user specifies the search statistics to be reported by defining an output view that includes special statistical fields generated by the Search Server. Refer to the Output Views section in this guide for a list of statistical fields that are available.

Although statistical fields may be written in any numeric format to an output view, the report file will only summarize the statistics for fields that have a field type of 'R'. Therefore the statistical fields to be summarized in the report must have a format of R. Specify a length for the field that is large enough to handle the number of rows processed. `R,10` is adequate for most situations.

By default the report produced is tab delimited, which is suitable for importing into a spreadsheet. Use the `-t` switch to generate a report file that uses spaces instead of tabs.

Starting from the Command Line

`relperf` can be started from the command line as follows:

For Win32, type the following:

```
%SSABIN%\relperf Search Infile Outfile OutputView-rRulebase-pSystem -hHost:Port -
wWorkDir[Optional Switches]
```

For Unix, type the following:

```
$SSABIN/relperf Search Infile Outfile OutputView-rRulebase -pSystem -hHost:Port[Optional
Switches]
```

The values passed in the command line are described in the following table:

Parameter	Description	Mandatory
Search	Nominates the Search Definition to use. If multiple searches are to be run, separate them with a comma. For example,`searchname1,searchname2,searchname3`	Yes
Infile	Name of the file containing input records	Yes
Outfile	Name of the report file to generate	Yes
OutputView	Name of the output view to use.	
-rRulebase	Name of the Rulebase	Yes
-pSystem	Name of the System	Yes
-hHost:Port	Name of the host and port number (may be Search or Connection server).	Yes
-iInputViewName	Nominates the view that describes the input records. If not specified, the IDT layout is assumed.	

Parameter	Description	Mandatory
-nx[:y[:z]]	Use x search threads with an input queue. of y records and an output queue of z records per thread.	
-wWorkDir	Work Directory	Yes
-t	Change report format to not use tabs.	
-bTempfile	To specify a temporary file for relperf to use. By default relperf will use `'relperf.out'` in the Work Directory.	
-s	Create a second report for each search ordered by match tolerances. An example of which can be seen in the Example reports section below.	
-a	Create an alternate style report with a histogram of accepted count. An example of which can be seen in the Example reports section below.	

-c	Creates a default statistical view for use during the relperf run. This view will contain the following fields:
	
	`ksl-total-count`
	`ksl-accepted-count`
	`ksl-rejected-count`
	`ksl-undecided-count`
	`idx-io`
	`idt-io`
	
	An Output view does not need to be specified when using this option. However if an output view is specified a view will be created for the run that will consist of all the fields in the specified output view plus any of the default statistical fields not already present.	

-dDatabase	Name of the Database. Must be specified when using the -c option.	
-eRulebaseHost	Name of the Rulebase Host. Must be specified when using the -c option.	

Example reports

Here is an example of a `relperf` report created using a simple output view and a Search-Definition named `search-namev2`. It shows that as the search width increases from Narrow to Typical to Exhaustive, the number of candidates selected also increases. For a given set of candidates (with the same search width), the number of accepted matches increases as the match tolerance becomes looser.

```
Search                 Match
Candidates                        Accepted
Widths                 Tolerances             ---------------------
---------------------------
                                                                  Average     Std
Dev           Average         Std Dev         % of Cand
search-namev2

Narrow                 Conservative         1.05            0.94
0.82            0.61                  78.10
Narrow                 Typical                    1.05            0.94
0.82            0.61                  78.10
Narrow                 Loose                      1.05            0.94
1.01            0.89                  96.19
Typical           Conservative          2.47            3.92
1.22            0.54                  49.39
```

```
Typical             Typical                 2.47            3.92
1.26            0.58                51.01
Typical             Loose                   2.47         3.92
2.25            3.63               91.09
Exhaustive Conservative         4.59            6.38                    1.22
0.54                26.58
Exhaustive Typical                  4.59            6.38
1.37            0.72                29.85
Exhaustive Loose                    4.59            6.38
2.89            3.69                62.96
```

This is the output view definition used for this report:

```
VIEW-DEFINITION
*==============
NAME=relx98stat
FIELD=Name,             C, 8
FIELD=ksl-total-count,R, 4
FIELD=ksl-accepted-count,R, 4
```

This is an example of a report created using the "-s" switch:

```
Search                 Match                                  Candidates
Accepted
Widths                 Tolerances            ---------------
------------------------------
                         Average Std Dev  Average  Std Dev    % of Cand

search-namev2

Narrow     Conservative   1.05    0.94     0.82     0.61         78.10
Narrow     Typical        1.05    0.94     0.82     0.61         78.10
Narrow     Loose          1.05    0.94     1.01     0.89         96.19
Typical    Conservative   2.47    3.92     1.22     0.54         49.39
Typical    Typical        2.47    3.92     1.26     0.58         51.01
Typical    Loose          2.47    3.92     2.25     3.63         91.09
Exhaustive Conservative   4.59    6.38     1.22     0.54         26.58
Exhaustive Typical        4.59    6.38     1.37     0.72         29.85
Exhaustive Loose          4.59    6.38     2.89     3.69         62.96
Extreme    Conservative   4.59    6.38     1.22     0.54         26.58
Extreme    Typical        4.59    6.38     1.37     0.72         29.85
Extreme    Loose          4.59    6.38     2.89     3.69         62.96
Narrow     Conservative   1.05    0.94     0.82     0.61         78.10
Typical    Conservative   2.47    3.92     1.22     0.54         49.39
Exhaustive Conservative   4.59    6.38     1.22     0.54         26.58
Extreme    Conservative   4.59    6.38     1.22     0.54         26.58
Narrow     Typical        1.05    0.94     0.82     0.61         78.10
Typical    Typical        2.47    3.92     1.26     0.58         51.01
Exhaustive Typical        4.59    6.38     1.37     0.72         29.85
Extreme    Typical        4.59    6.38     1.37     0.72         29.85
Narrow     Loose          1.05    0.94     1.01     0.89         96.19
Typical    Loose          2.47    3.92     2.25     3.63         91.09
Exhaustive Loose          4.59    6.38     2.89     3.69         62.96
Extreme    Loose          4.59    6.38     2.89     3.69         62.96
```

This is an example showing the additional columns in a report created using the "-a" switch:

Note: A report generated with the "-a" switch does not output columns representing standard deviations.

```
Number of Accepted within Range
-------------------------------------------------------------------------
0                1            2 - 10            11 - 100             101-1000
1001-10000                > 10000
26             69           5                        0
0                            0                                  0
26             69                 5
0                             0
```

```
0                                   0
26              58                  16                          0
0                               0                                       0
0                   84                  16
0                                   0
0                                       0
0                   81                  19
0                                   0
0                                       0
0                   65                  33
2                                   0                           0
0
0                   84                  16
0                                   0
0                                       0
0                   74                  26
0                                   0
0                                       0
0                   45                  53
2                           0
0                                       0
```

Performance Tips

- The search performance is directly proportional to the number of candidates selected.
- Use the results to assess how many additional matches will be found by using a wider search and/or looser match strategy. Is it worth processing twice as many candidates when only a few more records were accepted? The answer depends on the search problem, but at least you can assess the cost and the benefit.
- Use the average number of accepted matches (plus n times the standard deviation) to set the SORT=Memory() parameter to ensure that all sorts are performed in-core. Setting `n=3` ensures that 99% of sets will fit into memory.
- Use the average number of candidates (plus `n` times the standard deviation) to ensure that the `Candidate-Set-Size-Limit` is large enough to record candidates that have already been processed.

CHAPTER 9

Miscellaneous Issues

This chapter includes the following topics:

Backup and Restore

IIR relies on the standard facilities of the host DBMS to maintain integrity of its tables and indexes.

The Rulebase tables and indexes must be stored on the same database as the IIR Tables so that the rules are kept in synch with the data in case the database is restored and/or repaired using a DBMS utility.

System rules within a Rulebase may be backed up and restored using the IIR Console Export and Import functions.

These functions transfer the rules to/from an operating system file. The file can then be saved (for off-site backup purposes) or used to transfer the rules to another Rulebase.

Note: System rules may only be imported with the same version of the software used to export them.

User Exits

User Exits (UE) provide a way to extend the capabilities of IIR at specific points in its processing logic. IIR supports two types of User Exits:

- User Source Exit
- Transform Exit

User Exits must conform to a specific protocol in order to communicate with IIR. They are written in C and compiled and linked as a DLL/shared library. IIR will load the UE and call a predefined entry point when a "service" is required from the exit.

The syntax, parameters, details of operations and response codes for User Exits are only available on request.

User Source Exit

A User Source Exit (USE) is used to provide access to User Source Tables. It may use any valid SQL supported by the host DBMS.

USEs should only be used when IIR does not support the type of SQL access required. For example, if you wish to join and merge several tables in one operation.

USEs can also be used to access heterogeneous database architectures. For example, to load an IDT on an Oracle database using source data read from UDB.

USEs can only be defined for NoSync systems.

USEs must be coded in a multi-threaded manner because they will be called from a thread in the Table Loader. For example, a USE that connects to Oracle must allocate and use a thread context.

Note: By coding a USE, the user accepts responsibility for providing all services for User Source Table access that IIR requires. If the protocol is not strictly adhered to, a loss of IIR data integrity will result. An erroneous user exit may cause the Table Loader or Update Synchronizer to terminate abnormally.

Transform Exit

A Transform Exit (TE) is used to alter a record read from the source database prior to insertion into the IDT.

The TE exit is called after reading a record from the source database and before any field level transforms have been performed. The exit is a "record level" exit, meaning that it has access to the entire record.

Upon return from the exit, IIR will perform field-level transforms while converting the view record into IDT layout.

The normal use for a TE is to insert data into a new field. A new field can be created with the Filler transform by specifying a name, format and length. The field can then be referenced in the TE.

The user-exit is enabled by adding a user-exit transform definition. For example,

```
transform
                    filler                                          credit C(10),
                    user-exit     ("myexit.dll myep")
```

defines a field called `"credit"` which can be populated by the user-exit.

Virtual Private Databases (VPD)

Virtual Private Databases (VPD) is an Oracle specific feature available as a standard part of Enterprise Edition. VPD provides fine-grained access control that is data-driven, context-dependent and rowbased.

A security administrator defines database `contexts` for individual users or groups of users. These contexts are used to limit the data that a user can see when they execute SQL statements.

Contexts are usually set (enabled) using a login trigger so that a user has no control over the context they use. Once enabled, the context name is used by Oracle to find predefined rules that are used to dynamically modify SQL statements executed by the user.

The security administrator defines contexts and their associated rules. For example, user U1 logs on, a login trigger fires to set a context C1. When the user executes the following SQL:

```
SELECT * FROM EMP
```

Oracle adds `WHERE DEPT = 10` to the query so that user U1 can only see rows from his/her own department. Another user U2 using context C2 might have the predicate `WHERE DEPT = 20` added to their otherwise identical SQL. In this way, each user only sees data that is relevant to them.

IIR Implementation

The SSA userid is used to extract source data from the UST and create an IDT and IDXs. In order to see all rows in the UST, the SSA userid must be exempted from VPD based restrictions. This is done with the following SQL:

```
GRANT EXEMPT ACCESS POLICY TO SSA;
```

Once all the data is loaded into the IDT and IDXs, ordinary database access controls secure this data, since it is private to the SSA user. A dictionary-alias is normally used to hide the SSA userid and password from search clients.

IIR provides access to the IDT/IDX data through the Search Server. To guard against unauthorized access the System administrator defines the System-Definition option`VPD-Secure` and/or defines the environment variable *SSASECUREENV* .

When either is defined, IIR insists that each user provide a context using **ids_set_vpd_user** API prior to starting a search. In response to a search request, the Search Server will build a result set and for each record in that set, issue SQL queries using the user's context to screen out any record that the user is not permitted to see.

Environment Variables

A VPD System requires an additional environment variable to be defined. *SSASECURECONNECT* specifies a database connection string for a "proxy user" that will be used to screen records from the result set.

The connection string is a normal IIR connection string such as

```
odb:99:ssaproxy/ssaproxy@oracle920
```

or may be the special keyword SSAUSER, in which case IIR uses the current Rulebase connection string to create a proxy user connection to the UST database.

Proxy User Context

IIR establishes a context for the proxy user by calling a PL/SQL package provided by the security administrator.

Each search user must call the API function **ids_set_vpd_user** to nominate the name of the package and the parameter to be passed to it.

Restrictions

IDTs may not be created by merging (`merged_from` clause)

IDTs may not be flattened.

`LOGTEST` tracing is disabled in a VPD environment.

The only IIR Search Client that supports VPD is `relate`. Use the `-V` switch to specify the name of the context setting package and its parameter.

Large File Support

IIR contains support for working with "flat files" larger than 2GB.

Some operating systems limit the maximum file size to 2GB. We will refer to these as "small systems". Other systems have native support for files larger than 2GB. We will refer to these as "large systems".

To overcome the size limitation, IIR can combine the free space on a number of file-systems into one large logical file. A logical file larger than 2GB is composed of a number of small files known as "extents". Each extent must be less than 2GB in size.

Although this feature is not necessary on "large systems", it can be used to distribute the data over multiple file systems thereby making use of fragmented space.

Large File Support is designed for binary files. Restrictions apply to its use for text files, as described later.

Directory File

The management of extents can default to internal rules or can be user-supplied. A file known as the "directory file" can be defined to specify the number of extents as well as their names and sizes.

Each large file's directory file is named using operating-system specific rules. Currently the file, if present, is the name of the large file with `.dir` appended. For example, the directory file for `db.dat` would be called `db.dat.dir`.

Directory files contain multiple lines of text. Each line contains two blank separated fields used to define an extent. The size of the extent (in bytes) is followed by the name of the extent. The maximum extent size is limited to 2GB - 1. An asterisk (*) may be used as a shorthand notation for the maximum extent size.

For example, these definitions define two extents. The first is limited to 1Mb and the second extent defaults to 2GB - 1.

```
1048576                 db.dat.ext1
*                         db.dat.ext2
```

If all extents are to be of equal size, you can define a template for the base name of the extents. For example,

```
1048575                 db.dat.ext
*
```

will allocate extents of size 1048575 and name them using the rules documented in section Default Extent Names below. Note that the second line containing the asterisk enables this type of processing.

Also, this mode of processing requires the extent size (1048575 in this example) to be a power of 2 (1048576 in this example) minus 1. To allow all extents to have the maximum size of 2GB-1 use:

```
*                                       db.dat.ext
*
```

To allow all large files to have maximum size extents, create a file called `extents.dir` with the following text:

```
*       %f*
```

Note: It is possible to set the maximum extent size using the environment variable *SSAEXTENTSIZE* . Example, *SSAEXTENTSIZE =256k*will limit the size of all extent to 256K (minus 1).

Default Extent Names

If a directory file does not exist when a large file is opened, default rules are used to name the extents.

Each extent size defaults to 2GB - 1.

The first extent has the same name as the large file. Second and subsequent extents are created by appending two characters to the file name. The extensions are named aa, ab, ac,... az, ba,. . . zz. The means that (1+26*26) extents are possible giving a maximum logical file size of 1.3Tb

Using the example above, the extents would be named:

```
db.dat
db.dat.extaa
db.dat.extab...
```

Small System Rules

Small systems support large files using the rules above. Extents are defined using a directory file. If a directory file does not exist default sizes and names are used.

Large System Rules

Large systems have native support for files larger than 2GB. Operating systems such as Windows NT 4.0, HPUX and Digital/Unix are in this category.

Large files do not use extents on these systems unless a directory file is defined. In the latter case, extents are still limited to 2GB - 1.

Restrictions

Large file support was designed for binary files. In general, `text` files are not supported by the extent mechanism.

Text files do not default to use extents when a directory file is not present. Even if a directory file is defined and extents are used, correct results can not be guaranteed on every platform.

If you wish to use large text files, you should use an operating system that supports them natively.

Flat File Input from a Named Pipe

The IIR Table Loader can read input data from a named pipe. Relate cannot read Win32 pipes at this stage.

UNIX Platforms

On UNIX platforms the IIR input processor can read input from a Named Pipe. This means that it is possible to read data from another database without the need to create large intermediate files.

The concept is identical on all UNIX platforms, although the command used to create a named pipe may vary between implementations. The following example in applicable to LINUX.

```
mkfifo $SSAWORKDIR/inpipe
```

To use the pipe, specify its name as the Physical-File parameter in the Logical-File-Definition of the input file:

```
logical-file-definition
*======================
NAME=                                              lf-input
PHYSICAL-FILE=                                   "+/inpipe"
COMMENT=                                           "named pipe for the loader"
VIEW=                                              DATAIN
INPUT-FORMAT=                                      TEXT
AUTO-ID-NAME=                                      Job1
```

Windows Platforms

To use a named pipe on Windows environments, you need to specify the name of the pipe in the Microsoft format:

```
\\server\pipe\pipe_name
```

where

- `server` is the server name or dot (.) for the current machine
- `pipe` the word "pipe"
- `pipe_name` the pathname of the named pipe file

To use the pipe, specify its name as the Physical-File parameter in the Logical-File-Definition of the input file:

```
logical-file-definition
*======================
NAME=                                          lf-input
COMMENT=                                     "input file"
PHYSICAL-FILE=                             "\\.\pipe\pipe_name"
VIEW=                                          DATAIN
FORMAT=                                          TEXT
AUTO-ID-NAME=                                  JobN


logical-file-definition
*======================
NAME=                                          lf-input
COMMENT=                                     "input file"
PHYSICAL-FILE=                             "\\.\pipe\pipe_name"
VIEW=                                          DATAIN
FORMAT=                                          TEXT
AUTO-ID-NAME=                                  JobN
```

CHAPTER 10

Limitations

IIR design and host DBMS impose some limits:

Database

- Only one field per file with a format B(inary) can exceed 255 bytes in length.
- C(haracter) fields can not exceed 2000 bytes.
- There is a maximum of 255 fields per file.
- File-name length is limited to 23 bytes.
- Index-name length is limited to 22 bytes.
- IDX Key is limited to 255 bytes.
- A maximum of 4G - 1 rows per IDT.
- LOB columns are limited to 16 MB.

Object Names

Object names (tables, indexes, columns, functions) follow the rules of the host DBMS, with the following additional restrictions:

- identifiers must not exceed 32 bytes in length
- MS-SQL: Only regular identifiers are supported (up to 32 bytes in length)

IDENTITY functions (MS-SQL Server)

MS-SQL Server provides a function named @@IDENTITY that returns the identity value generated by the last performed `INSERT`, `SELECT INTO` or bulk copy statement.

For a synchronized system, if any source data tables contain identity columns, the @@IDENTITY function is not safe to use for the purpose of returning the last updated row from any of these source data tables. This is due to IIR using an identity column in the `IDS_UPD_SYNC_TXN` table.

Whenever a source table row is updated, a trigger will fire and insert a row into `IDS_UPD_SYNC_TXN`. Therefore the @@IDENTITY function will return the identity of the last row inserted into `IDS_UPD_SYNC_TXN`, not the identity of the last updated source table row.

MS SQL 2000 provides two new functions for handling this:

- `SCOPE_IDENTITY()` will return the last `IDENTITY` value inserted into an `IDENTITY` column in the same scope. Two statements are in the same scope if they are in the same stored procedure, function, or batch. Since it returns values inserted only within the current scope, it will not report on the `IDS_UPD_SYNC_TXN` table.
- `IDENT_CURRENT`('Table name') will return the last `IDENTITY` value generated for the specified table in any session and any scope. Since it returns values inserted only in the specified table, it will not report on the `IDS_UPD_SYNC_TXN` table.

Note: We recommend that one of these functions be used instead of the @@IDENTITY function.

Synchronizer

- Maximum of 36 PK fields per IDT.
- Each PK field value must be less than 1000 bytes in length
- The combined length of PK field names and PK field values can not exceed 2000 bytes.

Servers

- Maximum of 1024 TCP/IP connections per Server
- Maximum of 4096 database connections per Database Module (`ssadbm.dll`).

CHAPTER 11

Error Messages

This section describes about the error messages that you need to troubleshoot.

Oracle

When the IIR Table Loader spawns SQL*Loader to load files, a performance option (Direct-Path) is used which bypasses the database nucleus and directly writes data blocks to the database files. When running a Direct-Path load across a network to an Oracle server having a different architecture (byte order, etc), an error message as follows will be returned:

```
ORA-02352: Direct path connection must be homogeneous
```

The solution is to either

1. run the load on the same machine, or
2. run a Conventional-Path load by specifying the Loader-Definition `Options=Conventional-Path`. This is slow compared to the first option.

INDEX

T

U

V