1

ナレッジ・モジュールの概要

ナレッジ・モジュールとは

ナレッジ・モジュール（KM）はコード・テンプレートです。それぞれの KM は、データ統合プロセス全体の中の、独立した 1 つのタスクに対応しています。KM のコードの表示は、実行される形式とほとんど同じですが、Oracle Data Integrator（ODI）の代替メソッドが含まれている点で異なるため、一般的に多様な統合ジョブで使用できます。生成および実行されるコードは、ODI のデザイナ・モジュールで定義された宣言規則およびメタデータから導出されます。

- KM は、複数のインタフェースまたはモデルの間で再使用されます。ハンドコードされたスクリプトおよびプロシージャを使用する数百ものジョブの動作を変更するには、開発者がそれぞれのスクリプトまたはプロシージャを変更する必要があります。これに対して、ナレッジ・モジュールの利点は、1 回の変更が即座に数百の変換に伝播されることです。KM は、実行される論理タスクに基づきます。KM には、物理オブジェクト（データストア、列、物理パスなど）への参照は含まれません。
- KM は、影響分析のために分析可能です。
- スタンドアロンでは実行できません。インタフェース、データストアおよびモデルからのメタデータが必要です。

KM は、6 つの異なるカテゴリに分類されます。その要約を次の表に示します。

ナレッジ・モジュール	説明	使用場所
リバースエンジニアリング KM	Oracle Data Integrator の作業リポジトリに格納するメタデータを取得します。	カスタマイズされたリバースエンジニアリングを実行するモデル
チェック KM	制約と照合してデータの一貫性をチェックします。	モデル、サブモデルおよびデータストア（データ整合性監査の場合） インタフェース（フロー制御または静的制御の場合）
ロード KM	異機種間データをステージング領域にロードします。	異機種間ソースを使用するインタフェース
統合 KM	ステージング領域からターゲットにデータを統合します。	インタフェース
ジャーナル化 KM	ソース・ステージング領域に、チェンジ・データ・キャプチャ・フレームワーク・オブジェクトを作成します。	モデル、サブモデルおよびデータストア（ジャーナルの作成、開始、停止、およびサブスクライバの登録のため）
サービス KM	データ操作 Web サービスを生成します。	モデルおよびデータストア

Oracle Data Integrator には、すぐに使用できる 100 以上のナレッジ・モジュールが含まれています。

次の各項では、それぞれの種類のナレッジ・モジュールについて説明します。

リバースエンジニアリング・ナレッジ・モジュール（RKM）

RKM の主な役割は、モデルのカスタマイズ済リバースエンジニアリングを実行することです。RKM は、アプリケーションまたはメタデータ・プロバイダに接続し、作成されたメタデータを変換して Oracle Data Integrator のリポジトリに書き込みます。メタデータは、一時的に SNP_REV_xx 表に書き込まれます。その後、RKM によって Oracle Data Integrator API がコールされ、これらの表からデータが読み取られ、Oracle Data Integrator の作業リポジトリのメタデータ表に、増分更新モードで書き込まれます。図に示すと次のようになります。

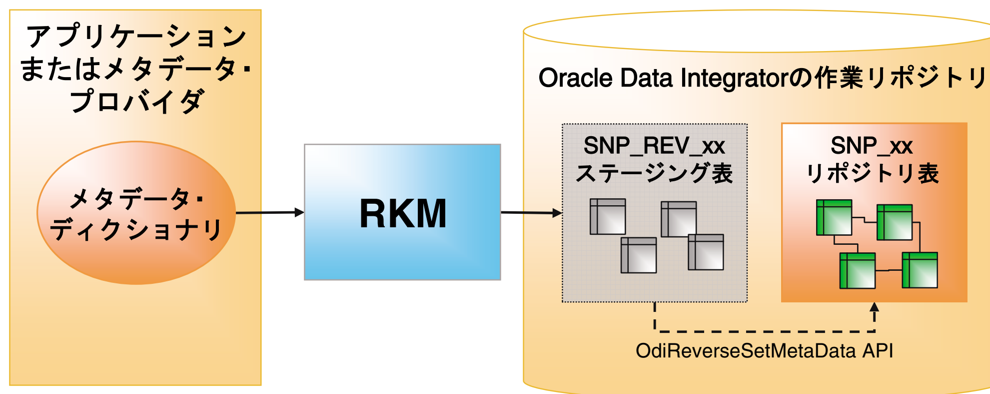


図 1-1 リバースエンジニアリング・ナレッジ・モジュール

通常の RKM は、次の手順に従います。

1. 前の実行で取得された SNP_REV_xx 表を、OdiReverseResetTable コマンドを使用してクリーンアップします。
2. メタデータ・プロバイダからサブモデル、データストア、列、一意キー、外部キー、条件を取得し、SNP_REV_SUB_MODEL、SNP_REV_TABLE、SNP_REV_COL、SNP_REV_KEY、SNP_REV_KEY_COL、SNP_REV_JOIN、SNP_REV_JOIN_COL、SNP_REV_COND の各表に書き込みます。
3. OdiReverseSetMetaData API をコールして、作業リポジトリ内のモデルを更新します。

チェック・ナレッジ・モジュール（CKM）

CKM は、データセットのレコードが定義済の制約と一致することをチェックします。CKM は、データの整合性を保持するために使用され、全体的なデータ品質の決定に関与します。CKM は、次の 2 つの方法で使用できます。

- 既存のデータの一貫性をチェックする。これは、STATIC_CONTROL オプションを Yes に設定して、任意のデータストアまたはインタフェース内で実行できます。1 つ目のケースでは、チェックされるデータは現在データストアにあるデータです。2 つ目のケースでは、ターゲット・データストアのデータがロード後にチェックされます。

- ターゲット・データストアにレコードをロードする前に、受け取ったデータの一貫性をチェックする。これは、FLOW_CONTROL オプションを使用して実行します。このケースでは、作成されたフローのターゲット・データストアの制約が、ターゲットに書き込まれる前に、CKM によってシミュレートされます。

要約 : CKM では、既存の表もしくは IKM によって作成された I$ 一時表をチェックできます。

CKM は、チェックする一連の制約および表の名前を受け入れます。また、拒否されたすべてのレコードの書込み先となる E$ エラー表を作成します。さらに、チェック結果セットから誤ったレコードを削除します。

次の図は、STATIC_CONTROL モードおよび FLOW_CONTROL モードの両方で、CKM がどのように動作するかを示しています。

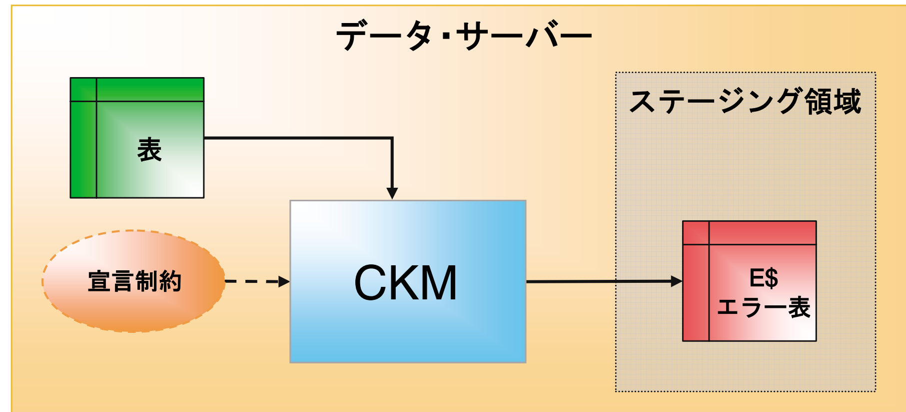


図 1-2 チェック・ナレッジ・モジュール（STATIC_CONTROL）

STATIC_CONTROL モードの場合、CKM は表の制約を読取り、表のデータと照合してチェックします。制約と一致しないレコードは、ステージング領域の E$ エラー表に書き込まれます。

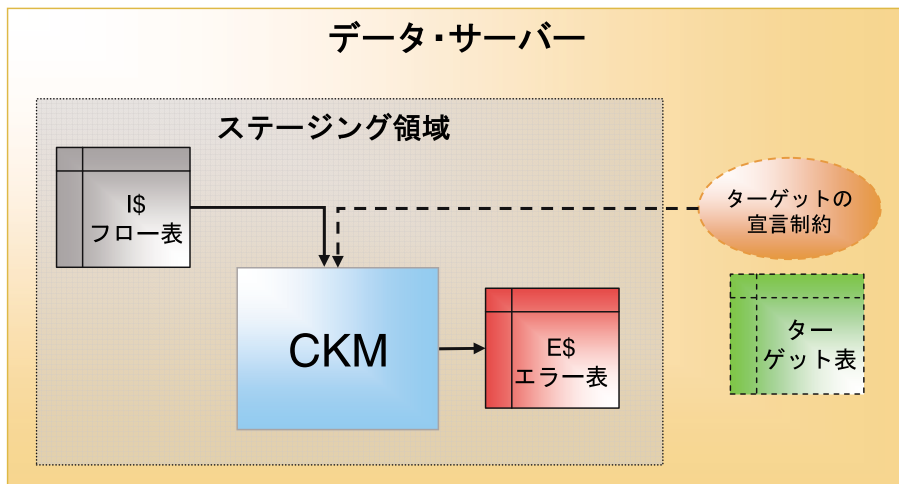


図 1-3 チェック・ナレッジ・モジュール（FLOW_CONTROL）

FLOW_CONTROL モードの場合、CKM は、インタフェースのターゲット表の制約を読み取ります。これらの制約を、ステージング領域の I$ フロー表に含まれるデータと照合してチェックします。これらの制約に違反するレコードは、ステージング領域の E$ 表に書き込まれます。

どちらの場合も、通常、CKM は次のタスクを実行します。

1. ステージング領域に E$ エラー表を作成します。エラー表には、データストアと同じ列に加えて、エラー・メッセージ、チェックの実行元、チェックの日付などのトレースに使用する追加の列が必要です。
2. チェックする必要がある主キー、代替キー、外部キー、条件、必須列ごとに、誤ったレコードを E$ 表に隔離します。
3. 必要に応じて、チェック済の表から誤ったレコードを削除します。

ロード・ナレッジ・モジュール（LKM）

LKM は、ソース・データをリモート・サーバーからステージング領域へロードします。一部のソース・データストアがステージング領域と同じデータ・サーバー上にない場合に、インタフェースによって使用されます。次の図に示すように、LKM は、ソース・サーバーで実行する必要がある宣言規則を実装し、単一の結果セットを取得して、ステージング領域の C$ 表に格納します。

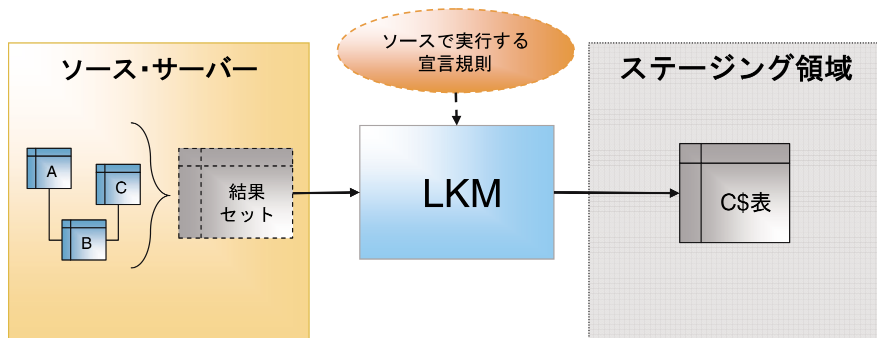


図 1-4 ロード・ナレッジ・モジュール

1. LKM は、ステージング領域に C$ 一時表を作成します。この表では、ソース・サーバーからロードされたレコードが保持されます。
2. LKM は、ソース上で適切な変換を実行することで、ソース・サーバーから一連の事前変換済レコードを取得します。ソース・サーバーが RDBMS の場合は通常、単一の SQL SELECT 問合せを使用して実行されます。ソースに SQL の機能がない場合（フラット・ファイルまたはアプリケーションなど）、LKM は、単純に適切な方法（ファイルの読取りまたは API の実行）でソース・データを読み取ります。
3. LKM は、ステージング領域の C$ 表にレコードをロードします。

インタフェースでは、異なるソースからデータストアを使用する際にいくつかの LKM が必要になる場合があります。すべてのソース・データストアがステージング領域と同じデータ・サーバー上にある場合は、LKM は不要です。

統合ナレッジ・モジュール（IKM）

IKM は、変換された最終データをターゲット表に書き込みます。それぞれのインタフェースでは、単一の IKM が使用されます。IKM の起動時には、リモート・サーバーの全ロード・フェーズのタスクは完了済であるとみなされます。つまり、すべてのリモート・ソース・データセットが LKM によってステージング領域の C$ 一時表にロードされている、もしくはソース・データストアがステージング領域と同じデータ・サーバー上にあるとみなされます。そのため、IKM によって実行されるのは、C$ 表およびステージング領域と同じデータ・サーバー上にある表に対するステージングおよびターゲットの変換、結合およびフィルタ処理のみです。通常、作成されるセットは IKM によって処理され、ターゲットにロードされる前に I$ 一時表に書き込まれます。これらの変換済最終レコードは、インタフェースで選択された IKM に応じて複数の方法で書き込むことができます。ターゲットに単純に追加するか、増分更新または緩やかに変化するディメンションのために比較することが可能です。IKM には 2 種類あります。ステージング領域がターゲット・データストアと同じサーバー上にあることを前提条件とする IKM、およびこの条件に当てはまらない場合に使用できる IKM です。これらを図に示します。

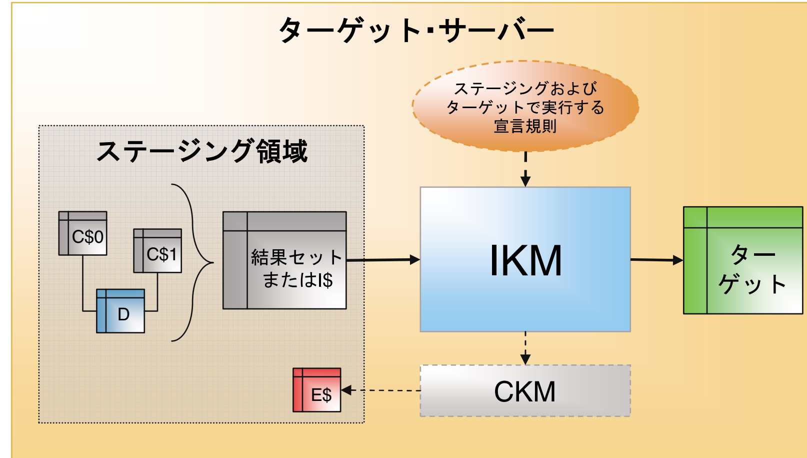


図 1-5 統合ナレッジ・モジュール（ステージング領域がターゲット上にある場合）

ステージング領域がターゲット・サーバー上にある場合は、通常、IKM によって次の手順が行われます。

1. セット指向の単一の SELECT 文を実行し、すべての C$ 表および（図の D のような）ローカル表のステージング領域およびターゲットの宣言規則を実行します。これにより、結果セットが生成されます。
2. 単純な追加 IKM では、この結果セットがターゲット表に直接書き込まれます。より複雑な IKM では、この結果セットを格納する I$ 表が作成されます。
3. ターゲットの制約と照合してデータ・フローをチェックする必要がある場合は、CKM をコールして誤ったレコードを隔離し、I$ 表をクレンジングします。

4. 定義された戦略（増分更新、緩やかに変化するディメンションなど）に従って、I$ 表からターゲットにレコードを書き込みます。
5. I$ 一時表を削除します。
6. 必要に応じて、CKM を再びコールしてターゲット・データストアの一貫性をチェックします。

この種類の KM は、ターゲット・サーバー外部のデータを操作しません。大量のジョブを実行する場合は、最大限の効率を得るために、データ処理はセット指向で行われます。

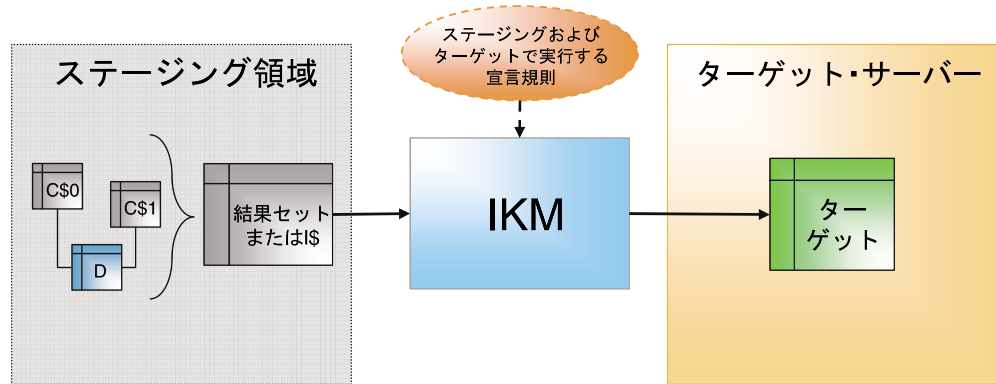


図 1-6 統合ナレッジ・モジュール（ステージング領域がターゲットと異なる場合）

図 1-6「統合ナレッジ・モジュール（ステージング領域がターゲットと異なる場合）」のように、ステージング領域がターゲット・サーバーと異なる場合は、通常、IKM によって次の手順が行われます。

1. セット指向の単一の SELECT 文を実行し、すべての C$ 表および（図の D のような）ステージング領域の表に対して宣言規則を実行します。これにより、結果セットが生成されます。
2. 定義された戦略（追加または増分更新）に従って、この結果セットをターゲット・データストアにロードします。

このアーキテクチャには、次のような一定の制限があります。

- CKM を使用して、処理対象のデータにデータ整合性監査を実行することはできません。
- データは、ターゲットにロードする前にステージング領域から抽出する必要があります。これによってパフォーマンスの問題が発生する場合があります。

ジャーナル化ナレッジ・モジュール（JKM）

JKM は、モデル、サブモデルまたはデータストア上に、チェンジ・データ・キャプチャのインフラストラクチャを作成します。JKM は、CDC インフラストラクチャの初期化方法を定義するために、インタフェースではなくモデル内で使用されます。次の図に示すように、このインフラストラクチャは、サブスクライバ表、変更の表、この表のビューおよび 1 つ以上のトリガーまたはログ取得プログラムで構成されます。

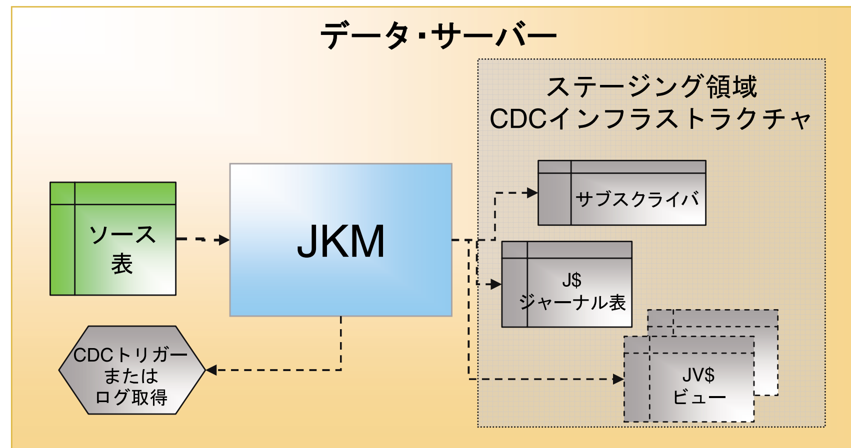


図 1-7　ジャーナル化ナレッジ・モジュール

サービス・ナレッジ・モジュール（SKM）

SKM は、データ操作 Web サービスを作成して、サービス指向アーキテクチャ（SOA）インフラストラクチャにデプロイします。SKM はモデル上で設定されます。SKM は、各データストアの Web サービスに対して生成される、様々な操作を定義します。他の KM とは異なり、実行可能コードを生成しません。かわりに Web サービス・デプロイメントのアーカイブ・ファイルを生成します。SKM は、Oracle Data Integrator の Web サービス用フレームワークを使用して、Java コードを生成するように設計されています。生成されたコードは後でコンパイルされ、最終的にアプリケーション・サーバーのコンテナにデプロイされます。

2

Oracle Data Integrator の代替 API

KM は、Oracle Data Integrator の代替 API を使用して、テンプレートとして記述されます。この API の詳細は、オンライン・ドキュメントに含まれています。API メソッドは、文字列値を戻す Java メソッドです。すべての API メソッドは、odiRef という単一のオブジェクト・インスタンスに属します。同じメソッドを使用しても、メソッドを起動する KM の種類によって、異なる値が戻される可能性があります。そのため、API は KM の種類によって分類されます。

> **注意：** 下位互換性を考慮して、**odiRef** API を **snpRef** API と呼ぶことがあります。snpRef および odiRef の各オブジェクト・インスタンスはシノニムです。この項に含まれる一部の例では、新しい odiRef の表記のかわりに、古い snpRef の表記がまだ使用されています。

この API の動作方法を理解できるように、次の例では、KM での CREATE TABLE 文の記述方法、および関連するデータストアに応じて生成されるコードについて説明します。

<table>
<tr><td>KM 内部のコード</td><td><pre>Create table <%=odiRef.getTable("L", "INT_NAME", "A")%>
(
<%=odiRef.getColList("", "\t[COL_NAME] [DEST_CRE_DT]", ",\n", "", "")%>
)</pre></td></tr>
<tr><td>PRODUCT データストアに対して生成されたコード</td><td><pre>Create table db_staging.I$_PRODUCT
(
    PRODUCT_ID numeric(10),
    PRODUCT_NAME varchar(250),
    FAMILY_ID numeric(4),
    SKU varchar(13),
    LAST_DATE timestamp
)</pre></td></tr>
<tr><td>CUSTOMER データストアに対して生成されたコード</td><td><pre>Create table db_staging.I$_CUSTOMER
(
    CUST_ID numeric(10),
    CUST_NAME varchar(250),
    ADDRESS varchar(250),
    CITY varchar(50),
    ZIP_CODE varchar(12),
    COUNTRY_ID varchar(3)
)</pre></td></tr>
</table>

適切なメタデータを使用して KM を実行した結果、商品および顧客の表に対して異なるコードが生成されたことがわかります。

次の各トピックでは、主要な代替 API の一部について説明し、KM 内での使用方法を説明します。読みやすくするため、例ではタグ <% と %>、および odiRef オブジェクト参照が省略されています。

データストアおよびオブジェクト名の使用方法

デザイナで作業する場合、実行コンテキストによって変化するデータベース名やスキーマ名などの物理情報を指定することはまずありません。正しい物理情報は、実行時に Oracle Data Integrator によって提供されます。

代替 API には、実行時のコンテキストを考慮してオブジェクトまたはデータストアの完全修飾名を算出するメソッドが備わっています。これらのメソッドを次の表に示します。

完全修飾名取得の対象	使用するメソッド	適用可能な場所
MY_OBJECT という名前のすべてのオブジェクト	`getObjectName("L", "MY_OBJECT", "D")`	すべての KM およびプロシージャ
ターゲット・データストア	`getTable("L", "TARG_NAME", "A")`	LKM、CKM、IKM、JKM
I$ データベース	`getTable("L", "INT_NAME", "A")`	LKM、IKM
C$ データストア	`getTable("L", "COLL_NAME", "A")`	LKM
E$ データストア	`getTable("L", "ERR_NAME", "A")`	LKM、CKM、IKM
チェック済データストア	`getTable("L", "CT_NAME", "A")`	CKM
外部キーによって参照されるデータストア	`getTable("L", "FK_PK_TABLE_NAME", "A")`	CKM

表、列および式のリストの使用方法

多くの場合、項目のリストからコードを生成するには、while ループまたは for ループが必要になります。Oracle Data Integrator は、リストに基づいてコードを生成できる強力なメソッドを提供することでこの問題に対処しています。これらのメソッドはイテレータとして機能します。代替マスクまたはパターンおよびセパレータを提供すると、メソッドによって、すべての解決済パターンがセパレータで区切られた単一の文字列が戻されます。

すべてのメソッドは 1 つの文字列を戻し、少なくとも次の 4 つのパラメータを受け入れます。

- 開始 : 作成される文字列を開始するための文字列。
- パターン : リストの各項目の値にバインドされる属性を使用した代替マスク。
- セパレータ : 置き換えられた各パターンと後続のパターンを区切るための文字列。
- 終了 : 作成される文字列の最後に追加される文字列。

一部のメソッドは、リストの一部の項目のみを取得するためのフィルタとして機能する、追加のパラメータ（セレクタ）を受け入れます。

これらのメソッドの一部について、次の表に要約を示します。

メソッド	説明	適用可能な場所
`getColList()`	Oracle Data Integrator で最も頻繁に使用されるメソッドです。使用されたコンテキスト内で実行する必要がある列および式のリストを戻します。たとえば、次のようなリストの生成に使用できます。 ■ CREATE TABLE 文内の列 ■ 更新キーの列 ■ LKM、CKM または IKM 内の SELECT 文の式 ■ ロード・スクリプトのフィールド定義 このメソッドでは、5 番目のパラメータとしてセレクタが受け入れられるため、必要に応じて項目をフィルタ処理できます。	LKM、CKM、IKM、JKM、SKM
`getTargetColList()`	ターゲット・データストアの列のリストを戻します。 このメソッドでは、5 番目のパラメータとしてセレクタが受け入れられるため、必要に応じて項目をフィルタ処理できます。	LKM、CKM、IKM、JKM、SKM
`getAKColList()`	代替キー用に定義された列のリストを戻します。	CKM、SKM
`getPKColList()`	主キーの列のリストを戻します。かわりに、セレクタ・パラメータを PK に設定して `getColList` を使用することもできます。	CKM、SKM
`getFKColList()`	現在の外部キーの、参照する列および参照される列を戻します。	CKM、SKM
`getSrcTablesList()`	インタフェースのソース表のリストを戻します。できるかぎり、`getFrom` メソッドをかわりに使用してください。`getFrom` メソッドについては後述します。	LKM、IKM
`getFilterList()`	インタフェースのフィルタ式のリストを戻します。通常は `getFilter` メソッドの方が適しています。	LKM、IKM
`getJoinList()`	インタフェースの結合式のリストを戻します。通常は `getJoin` メソッドの方が適しています。	LKM、IKM
`getGrpByList()`	インタフェースのマッピングで集計関数が検出されると、GROUP BY 句に表示される式のリストを戻します。通常は `getGrpBy` メソッドの方が適しています。	LKM、IKM
`getHavingList()`	インタフェースのフィルタで集計関数が検出されると、HAVING 句に表示される式のリストを戻します。通常は `getHaving` メソッドの方が適しています。	LKM、IKM
`getSubscriberList()`	サブスクライバのリストを戻します。	JKM

次の各例では、これらのメソッドを使用したコードの生成方法を説明します。

getTargetColList を使用した表の作成

KM 内のコード	`Create table MYTABLE` `<%=odiRef.getTargetColList("(\n", "\t[COL_NAME] [DEST_WRI_DT]", ",\n", "\n)")%>`

生成されるコード	`Create table MYTABLE`
	`(`
	`    CUST_ID numeric(10),`
	`    CUST_NAME varchar(250),`
	`    ADDRESS varchar(250),`
	`    CITY varchar(50),`
	`    ZIP_CODE varchar(12),`
	`    COUNTRY_ID varchar(3)`
	`)`

– 開始は `"(\n"` に設定されています。生成されるコードはカッコで始まり、その後に改行（\n）が続きます。

– パターンは `"\t[COL_NAME] [DEST_WRI_DT]"` に設定されています。生成されるコードは、各ターゲット列に対してループ処理を行い、タブ文字（\t）、列名（[COL_NAME]）、空白および変換先の書込み可能データ型（[DEST_WRI_DT]）の順に生成します。

– セパレータは `",\n"` に設定されています。生成される各パターンはカンマ（,）および改行（\n）によって後続のパターンと区切られます。

– 終了は `"\n)"` に設定されています。生成されるコードは、改行（\n）およびその後に続くカッコで終わります。

getColList を使用した INSERT VALUES 文

KM 内のコード	`insert into MYTABLE`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]", ", ", "\n", "INS AND NOT TARG")%>`
	`<%=odiRef.getColList(",", "[COL_NAME]", ", ", "", "INS AND TARG")%>`
	`)`
	`Values`
	`(`
	`<%=odiRef.getColList("", ":[COL_NAME]", ", ", "\n", "INS AND NOT TARG")%>`
	`<%=odiRef.getColList(", ", "[EXPRESSION]", ", ", "", "INS AND TARG")%>`
	`)`

生成されるコード	`insert into MYTABLE`
	`(`
	`CUST_ID, CUST_NAME, ADDRESS, CITY, COUNTRY_ID`
	`, ZIP_CODE, LAST_UPDATE`
	`)`
	`Values`
	`(`
	`:CUST_ID, :CUST_NAME, :ADDRESS, :CITY, :COUNTRY_ID`
	`, 'ZZ2345', current_timestamp`
	`)`

この例では、MYTABLE に挿入する必要がある値は、ターゲット列と同じ名前のバインド変数または定数式（ターゲット上で実行される場合）のどちらかです。これら 2 つの異なる項目セットを取得するために、セレクタ・パラメータを使用してリストが分割されています。

- "INS AND NOT TARG": まず、文の "Values" 部分にあるバインド変数（:[COL_NAME]）にマップされる、カンマ区切りの列のリスト（[COL_NAME]）を生成します。INSERT 文に含めるフラグが設定されており、**ターゲット上で実行されない**列のみを取得するために、フィルタ処理します。
- "INS AND TARG": 次に、NSERT 文に含めるフラグが設定されており、**ターゲット上で実行される**式（[EXPRESSION]）に対応する、カンマ区切りの列のリスト（[COL_NAME]）を生成します。項目が 1 つでも検出された場合、リストの開始文字はカンマになります。

getSrcTableList の使用方法

KM 内のコード	`insert into MYLOGTABLE`
	`(`
	`INTERFACE_NAME,`
	`DATE_LOADED,`
	`SOURCE_TABLES`
	`)`
	`values`
	`(`
	`'<%=odiRef.getPop("POP_NAME")%>',`
	`current_date,`
	`'' <%=odiRef.getSrcTablesList("|| ", "'[RES_NAME]'", " || ',' || ", "")%>`
	`)`

生成されるコード	`insert into MYLOGTABLE`
	`(`
	`INTERFACE_NAME,`
	`DATE_LOADED,`
	`SOURCE_TABLES`
	`)`
	`values`
	`(`
	`'Int.CUSTOMER',`
	`current_date,`
	`'' || 'SRC_CUST' || ',' || 'AGE_RANGE_FILE' || ',' || 'C$0_CUSTOMER'`
	`)`

この例では、getSrcTableList によって、インタフェースでソースとして使用されるリソース名のリストを含むメッセージが生成されます。このリストは MYLOGTABLE に追加されます。使用されるセパレータは、連結演算子（||)、引用符で囲まれたカンマ（',')、再び同じ連結演算子の順序で構成されます。表のリストが空の場合は、MYLOGTABLE の SOURCE_TABLES 列が空の文字列（''）にマップされます。

ソースの SELECT 文の生成

LKM および IKM は、両方ともソースの結果セットを操作します。LKM の結果セットは、ソース上で実行する必要があるマッピング、フィルタ処理および結合に従って事前変換されたレコードを表します。一方、IKM の結果セットは、ステージング領域で実行されるマッピング、フィルタ処理および結合と一致する変換済レコードを表します。

これらの結果セットを作成するには、通常、KM で SELECT 文を使用します。Oracle Data Integrator には、このコードを生成するための高度な代替メソッド（`getColList` を含む）が用意されています。

メソッド	説明	適用可能な場所
`getFrom()`	適切なソース表、左側外部結合、右側外部結合および完全外部結合を含む、SELECT 文の FROM 句を戻します。このメソッドでは、トポロジからの情報を使用して、ソースまたはターゲットのテクノロジの SQL 機能を判別します。テクノロジでサポートされている場合、FROM 句は、適切なキーワード（INNER、LEFT など）およびカッコを使用して適宜作成されます。 ■ LKM で使用した場合、ソース・サーバーによって実行される必要がある FROM 句が戻されます。 ■ IKM で使用した場合、ステージング領域サーバーによって実行される必要がある FROM 句が戻されます。	LKM、IKM
`getFilter()`	AND 演算子で区切られたフィルタ式を戻します。 ■ LKM で使用した場合、ソース・サーバーによって実行される必要がある FILTER 句が戻されます。 ■ IKM で使用した場合、ステージング領域サーバーによって実行される必要がある FILTER 句が戻されます。	LKM、IKM
`getJrnFilter()`	ジャーナル化されたソース・データストア用の特別なジャーナル・フィルタ式を戻します。このメソッドは、CDC フレームワークとともに使用する必要があります。	LKM、IKM
`getGrpBy()`	マッピングで集計関数が検出されると、GROUP BY 句を戻します。 GROUP BY 句には、集計関数を含まない列を参照するすべての対応付けの式が含まれます。集計関数のリストは、トポロジに含まれるテクノロジの言語によって定義されます。	LKM、IKM
`getHaving()`	フィルタで集計関数が検出されると、HAVING 句を戻します。 HAVING 句には、集計関数を含むすべてのフィルタ式が含まれます。集計関数のリストは、トポロジに含まれるテクノロジの言語によって定義されます。	LKM、IKM

任意の SQL RDBMS ソース・サーバーから結果セットを取得するには、次の SELECT 文を LKM で使用します。

```
select  <%=odiRef.getPop("DISTINCT_ROWS")%>
        <%=odiRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP] [CX_COL_NAME]", ",\n\t", "",
"")%>
from    <%=odiRef.getFrom()%>
where   (1=1)
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getJoin()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

最終フロー・データを作成するために、任意の SQL RDBMS ステージング領域サーバーから結果セットを取得するには、次の SELECT 文を IKM で使用します。ターゲット上で実行されず、

書込み可能な列に対応付けされている式のみを取得するために、getColList はフィルタ処理されています。

```
select <%=odiRef.getPop("DISTINCT_ROWS")%>
       <%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "(not TRG) and REW")%>
from   <%=odiRef.getFrom()%>
where  (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

すべてのフィルタおよび結合は AND で始まり、SELECT 文の WHERE 句は常に true（1=1）の条件で始まります。

API によるその他の情報の取得

次に示すメソッドは、有用な追加情報を提供します。

メソッド	説明	適用可能な場所
getPop()	現在のインタフェースに関する情報を戻します。	LKM、IKM
getInfo()	ソース・サーバーまたはターゲット・サーバーに関する情報を戻します。	任意のプロシージャまたは KM
getSession()	現在実行中のセッションに関する情報を戻します。	任意のプロシージャまたは KM
getOption()	特定のオプションの値を戻します。	任意のプロシージャまたは KM
getFlexFieldValue()	フレックス・フィールド値に関する情報を戻します。フレックス・フィールドの値は、リスト・メソッドを使用してパターン・パラメータの一部に指定できます。	任意のプロシージャまたは KM
getJrnInfo()	CDC フレームワークに関する情報を戻します。	JKM、LKM、IKM
getTargetTable()	インタフェースのターゲット表に関する情報を戻します。	LKM、IKM、CKM
getModel()	リバースエンジニアリングの処理中に現在のモデルに関する情報を戻します。	RKM

コード生成の高度な技術

条件分岐および高度なプログラミング技術を使用してコードを生成できます。Oracle Data Integrator でのコード生成は、<% および %> のタグで囲んだすべての Java コードを解釈できます。Java 言語の完全な参照は、http://java.sun.com を参照してください。

次に示す例は、これらの高度な技術の使用方法を説明しています。

KM またはプロシージャ内のコード	生成されるコード
<% String myTableName; myTableName = "ABCDEF"; %> drop table <%=odiRef.getObjectName(myTableName.toLowerCase())%>	drop table SCOTT.abcdef

<%
String myOptionValue=odiRef.getOption("Test");
 	
if **(myOption.equals("TRUE"))**)
{
out.print("/* Option Test is set to TRUE */");
}
else
{
%>
/* The Test option is not properly set */
<%
}
%>
...
	オプション Test が TRUE に設定されている場合：
	/* Option Test is set to TRUE */
	...
	
	それ以外の場合：
	/* The Test option is not properly set */
	...

Create table <%=odiRef.getObjectName("XYZ")%>
(
<%
String s;
s = "ABCDEF";
for (int i=0; i < s.length(); i++)
{
%>
<%=s.charAt(i)%> char(1),
<%
}
%>
G char(1)
)
	Create table ADAMS.XYZ
	(
	A char(1),
	B char(1),
	C char(1),
	D char(1),
	E char(1),
	F char(1),
	G char(1)
	)

3 リバースエンジニアリング・ナレッジ・モジュール（RKM）

RKM プロセス

RKM を使用したリバースエンジニアリング戦略のカスタマイズは、一般的に単純です。通常、手順はどの RKM でも同じです。

1. OdiReverseResetTable コマンドをコールして前の実行の SNP_REV_xx 表をリセットします。
2. メタデータ・プロバイダからサブモデル、データストア、列、一意キー、外部キー、条件を取得し、SNP_REV_SUB_MODEL、SNP_REV_TABLE、SNP_REV_COL、SNP_REV_KEY、SNP_REV_KEY_COL、SNP_REV_JOIN、SNP_REV_JOIN_COL、SNP_REV_COND の各表に格納します。SNP_REVxx 表の詳細は、「SNP_REV_xx 表参照」項を参照してください。
3. OdiReverseSetMetaData コマンドをコールして、現在の Oracle Data Integrator モデルに変更を適用します。

例として次に示すステップは、Oracle 用 RKM からの抜粋です。この RKM の追加情報は、『Oralce Data Integrator Knowledge Modules リファレンス・ガイド』を参照してください。

手順	コード例
SNP_REV 表のリセット	OdiReverseResetTable -MODEL=<%=odiRef.getModel("ID")%>
表およびビューの取得	/*=======================*/ /* Command on the source */ /*=======================*/ Select t.TABLE_NAME TABLE_NAME, t.TABLE_NAME RES_NAME, replace(t.TABLE_NAME,'<%=snpRef.getModel("REV_ALIAS_LTRIM")%>','') TABLE_ALIAS, substr(tc.COMMENTS,1,250) TABLE_DESC, t.NUM_ROWS R_COUNT From ALL_TABLES t, ALL_TAB_COMMENTS tc Where t.OWNER = '<%=snpRef.getModel("SCHEMA_NAME")%>' and t.TABLE_NAMElike '<%=snpRef.getModel("REV_OBJ_PATT")%>' and tc.OWNER(+)= t.OWNER and tc.TABLE_NAME(+) = t.TABLE_NAME /*=======================*/ /* Command on the target */ /*=======================*/ insert into SNP_REV_TABLE (I_MOD,TABLE_NAME,RES_NAME,TABLE_ALIAS,TABLE_TYPE,TABLE_DESC,IND_SHOW,R_COUNT) values (<%=odiRef.getModel("ID")%>,:TABLE_NAME,:RES_NAME,:TABLE_ALIAS,'T',:TABLE_DESC,'1',:R_COUNT)

<table>
<tr><th>手順</th><th>コード例</th></tr>
<tr><td>表の列の取得</td><td><pre>
/*=======================*/
/* Command on the source */
/*=======================*/
select
c.TABLE_NAME   TABLE_NAME,
c.COLUMN_NAME   COL_NAME,
c.DATA_TYPE   DT_DRIVER,
substr(cc.COMMENTS,1,250)COL_DESC,
c.COLUMN_ID   POS,
decode(C.DATA_TYPE, 'NUMBER', c.DATA_PRECISION,
nvl(c.DATA_PRECISION,c.DATA_LENGTH)) LONGC,
c.DATA_SCALE   SCALEC,
decode(c.NULLABLE,'Y','0','1')   COL_MANDATORY
from   ALL_TAB_COLUMNS   c,
       ALL_COL_COMMENTS   cc,
       ALL_OBJECTSo
Where o.OWNER = '<%=snpRef.getModel("SCHEMA_NAME")%>'
and   (o.OBJECT_TYPE   = 'TABLE' or o.OBJECT_TYPE= 'VIEW')
and o.OBJECT_NAME   like '<%=snpRef.getModel("REV_OBJ_PATT")%>'
and   cc.OWNER(+)   = c.OWNER
and   cc.TABLE_NAME(+)   = c.TABLE_NAME
and   cc.COLUMN_NAME(+)   = c.COLUMN_NAME
and   o.OWNER   = c.OWNER
and   o.OBJECT_NAME = c.TABLE_NAME
/*=======================*/
/* Command on the target */
/*=======================*/
insert into SNP_REV_COL
(I_MOD,TABLE_NAME,COL_NAME,DT_DRIVER,COL_DESC,POS,LONGC,SCALEC,
COL_MANDATORY,CHECK_STAT,CHECK_FLOW)
values
(<%=odiRef.getModel("ID")%>,:TABLE_NAME,:COL_NAME,:DT_DRIVER,
:COL_DESC,:POS,:LONGC,:SCALEC,:COL_MANDATORY,'1','1')
</pre></td></tr>
<tr><td>その他</td><td></td></tr>
<tr><td>メタデータの設定</td><td><pre>OdiReverseSetMetaData -MODEL=<%=odiRef.getModel("ID")%></pre></td></tr>
</table>

詳細は、次の各 RKM を参照してください。

RKM	説明
RKM Oracle	Oracle 用リバースエンジニアリング・ナレッジ・モジュール。
RKM Teradata	Teradata 用リバースエンジニアリング・ナレッジ・モジュール。
RKM DB2 400	DB2/400 用リバースエンジニアリング・ナレッジ・モジュール。表の詳細名ではなく短縮名を取得します。
RKM File（FROM EXCEL）	ファイル用リバースエンジニアリング・ナレッジ・モジュール。Microsoft Excel スプレッドシート形式のファイルの説明に基づきます。
RKM Informix SE	Informix Standard Edition 用リバースエンジニアリング・ナレッジ・モジュール。
RKM Informix	Informix 用リバースエンジニアリング・ナレッジ・モジュール。
RKM SQL（JYTHON）	JDBC 準拠データベース用リバースエンジニアリング・ナレッジ・モジュール。Jython を使用して JDBC API をコールします。

SNP_REV_xx 表参照

SNP_REV_SUB_MODEL

説明 : サブモデル用のリバースエンジニアリング一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
SMOD_CODE	varchar(35)	○	サブモデルのコード
SMOD_NAME	varchar(100)		サブモデルの名前
SMOD_PARENT_CODE	varchar(35)		親サブモデルのコード
IND_INTEGRATION	varchar(1)		OdiReverserSetMetadata API により内部的に使用
TABLE_NAME_PATTERN	varchar(35)		このサブモデル内でデータストアを分散するための自動割当てマスク
REV_APPY_PATTERN	varchar(1)		データストア分散規則 : 0: 分散なし 1: サブモデルに存在しない全データストアの自動分散 2: 全データストアの自動分散

SNP_REV_TABLE

説明 : データストアのリバースエンジニアリング用の一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
TABLE_NAME	varchar(100)	○	データストアの名前
RES_NAME	varchar(250)		データストアの物理名
TABLE_ALIAS	varchar(35)		このデータストアのデフォルトの別名
TABLE_TYPE	varchar(2)		データストアの種類 : T: 表またはファイル V: ビュー Q: キューまたはトピック ST: システム表 AT: 表の別名 SY: シノニム
TABLE_DESC	varchar(250)		データストアの説明
IND_SHOW	varchar(1)		このデータストアの表示 / 非表示の指定 : 0: 非表示 1: 表示
R_COUNT	numeric(10)		推定される行数
FILE_FORMAT	varchar(1)		レコードの形式（ファイルおよび JMS メッセージのみに適用） F: 固定長ファイル D: デリミタ付きファイル
FILE_SEP_FIELD	varchar(8)		フィールド・セパレータ（ファイルおよび JMS メッセージのみに適用）
FILE_ENC_FIELD	varchar(2)		テキスト・デリミタ（ファイルおよび JMS メッセージのみに適用）
FILE_SEP_ROW	varchar(8)		行セパレータ（ファイルおよび JMS メッセージのみに適用）
FILE_FIRST_ROW	numeric(10)		ファイルでスキップする数値またはレコード（ファイルおよび JMS メッセージのみに適用）
FILE_DEC_SEP	varchar(1)		ファイルの数値フィールドに使用するドデフォルトの小数セパレータ（ファイルおよび JMS メッセージのみに適用）
SMOD_CODE	varchar(35)		この表が配置されるサブモデルのコード。値がない場合、表はメイン・モデルに配置されます。

SNP_REV_COL

説明 : 列用のリバースエンジニアリング一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
TABLE_NAME	varchar(100)	○	表の名前
COL_NAME	varchar(100)	○	列の名前
COL_HEADING	varchar(35)		列の概略
COL_DESC	varchar(250)		列の詳細
DT_DRIVER	varchar(35)		列のデータ型。このデータ型は、このテクノロジの Oracle Data Integrator トポロジで定義されたデータ型コードと一致する必要があります。
POS	numeric(10)		表内の列の序数位置
LONGC	numeric(10)		列の文字長または数値精度基数
SCALEC	numeric(10)		列の 10 進数字
FILE_POS	numeric(10)		固定長ファイルでの列の開始バイト位置（ファイルおよび JMS メッセージのみに適用）
BYTES	numeric(10)		列のバイトの数値（ファイルおよび JMS メッセージのみに適用）
IND_WRITE	varchar(1)		列が書込み可能かどうか : 0 は不可、1 は可能
COL_MANDATORY	varchar(1)		列が必須かどうか : 0 は必須でない、1 は必須
CHECK_FLOW	varchar(1)		必須の制約をフロー制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。
CHECK_STAT	varchar(1)		必須の制約を静的制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。
COL_FORMAT	varchar(35)		列の書式。通常、このフィールドはファイルおよび JMS メッセージに適用され、日付書式の説明に使用されます。
COL_DEC_SEP	varchar(1)		列の小数セパレータ（ファイルおよび JMS メッセージのみに適用）
REC_CODE_LIST	varchar(250)		複数のレコード・ファイルをフィルタ処理するためのレコード・コード（ファイルおよび JMS メッセージのみに適用）
COL_NULL_IF_ERR	varchar(1)		エラー発生時にこの列を NULL に設定するかどうか（ファイルおよび JMS メッセージのみに適用）

SNP_REV_KEY

説明 : 主キー、代替キーおよび索引のリバースエンジニアリング用の一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
TABLE_NAME	varchar(100)	○	表の名前
KEY_NAME	varchar(100)	○	キーまたは索引の名前
CONS_TYPE	varchar(2)	○	キーの種類 : PK: 主キー AK: 代替キー I: 索引
IND_ACTIVE	varchar(1)		この制約がアクティブかどうか : 0 は非アクティブ、1 はアクティブ
CHECK_FLOW	varchar(1)		この制約をフロー制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。
CHECK_STAT	varchar(1)		制約を静的制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。

SNP_REV_KEY_COL

説明 : 主キー、代替キーまたは索引に含まれる列のリバースエンジニアリング用の一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
TABLE_NAME	varchar(100)	○	表の名前
KEY_NAME	varchar(100)	○	キーまたは索引の名前
COL_NAME	varchar(100)	○	キーに属する列の名前
POS	numeric(10)		キー内の列の序数位置

SNP_REV_JOIN

説明 : 参照（外部キー）のリバースエンジニアリング用の一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
FK_NAME	varchar(100)	○	参照または外部キーの名前
TABLE_NAME	varchar(100)	○	参照する表の名前
FK_TYPE	varchar(1)		外部キーの種類 : D: データベースの外部キー U: ユーザー定義の外部キー C: ユーザー定義の複合外部キー
PK_CATALOG	varchar(35)		参照される表のカタログ
PK_SCHEMA	varchar(35)		参照される表のスキーマ
PK_TABLE_NAME	varchar(100)		参照される表の名前
CHECK_STAT	varchar(1)		制約を静的制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。
CHECK_FLOW	varchar(1)		制約をフロー制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。

列	型	必須	説明
IND_ACTIVE	varchar(1)		この制約がアクティブかどうか : 0 は非アクティブ、1 はアクティブ
DEFER	varchar(1)		将来の使用のために予約済
UPD_RULE	varchar(1)		将来の使用のために予約済
DEL_RULE	varchar(1)		将来の使用のために予約済

SNP_REV_JOIN_COL

説明 : 参照（または外部キー）に含まれる列のリバースエンジニアリング用の一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
FK_NAME	varchar(100)	○	参照または外部キーの名前
FK_COL_NAME	varchar(100)	○	参照する表の列名
FK_TABLE_NAME	varchar(100)		参照する表の名前
PK_COL_NAME	varchar(100)	○	参照される表の列名
PK_TABLE_NAME	varchar(100)		参照される表の名前
POS	numeric(10)		外部キー内の列の序数位置

SNP_REV_COND

説明 : 条件およびフィルタ（チェック制約）のリバースエンジニアリング用の一時表。

列	型	必須	説明
I_MOD	numeric(10)	○	モデルの内部 ID
TABLE_NAME	varchar(100)	○	表の名前
COND_NAME	varchar(35)	○	条件またはチェック制約の名前
COND_TYPE	varchar(1)	○	条件の種類 : C: Oracle Data Integrator の条件 D: データベースの条件 F: 永続フィルタ
COND_SQL	varchar(250)		この条件またはフィルタを適用するための SQL 式
COND_MESS	varchar(250)		この条件のエラー・メッセージ
IND_ACTIVE	varchar(1)		この制約がアクティブかどうか : 0 は非アクティブ、1 はアクティブ
CHECK_STAT	varchar(1)		制約を静的制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。
CHECK_FLOW	varchar(1)		制約をフロー制御にデフォルトで含めるかどうか : 0 は含めない、1 は含める。

4 データ整合性の戦略（CKM）

標準のチェック・ナレッジ・モジュール

CKM は、事前定義済の一連の制約に従って、データストアのデータ品質をチェックします。CKM は、静的制御で既存のデータのチェックに使用するか、IKM から起動されたフロー制御でフロー・データのチェックに使用できます。また、指定すると、チェックした表から誤ったレコードを削除することもできます。

標準の CKM では、次の 2 種類の表が管理されます。

– 各データ・サーバーの単一のサマリー表。名前は SNP_CHECK_TAB で、データ・サーバーのデフォルト物理スキーマの作業スキーマで作成されます。この表には、それぞれの表および制約のエラーのサマリーが含まれます。たとえば、データ・ウェアハウスの全体的なデータ品質の分析に使用できます。

– チェックされた各データストアのエラー表。名前は E$_<DatastoreName> です。エラー表には、データ品質プロセスによって拒否された実際のレコードが含まれます。

これらの表の推奨列は次のとおりです。

表	列	説明
SNP_CHECK_TAB	CATALOG_NAME	チェックされた表のカタログ名（適用される場合）
	SCHEMA_NAME	チェックされた表のスキーマ名（適用される場合）
	RESOURCE_NAME	チェックされた表のリソース名
	FULL_RES_NAME	チェックされた表の完全修飾名。たとえば <catalog>.<schema>.<table>
	ERR_TYPE	エラーの種類 : ■ 'F': フロー制御中にデータストアがチェックされた場合 ■ 'S': 静的制御を使用してデータストアがチェックされた場合
	ERR_MESS	エラー・メッセージ
	CHECK_DATE	データストアがチェックされた日時
	ORIGIN	チェック操作の実行元。この列は、チェックの実行方法によって、データストア名、もしくはインタフェース名および ID のどちらかに設定されます。
	CONS_NAME	違反があった制約の名前
	CONS_TYPE	制約の種類 : ■ 'PK': 主キー ■ 'AK': 代替キー ■ 'FK': 外部キー ■ 'CK': チェック条件 ■ 'NN': 必須列
	ERR_COUNT	チェック処理中にこの制約によって拒否されたレコードの合計数
E$ エラー表	[チェックされた表の列]	チェックされたデータストアのすべての列を含むエラー表
	ERR_TYPE	エラーの種類 : ■ 'F': フロー制御中にデータストアがチェックされた場合 ■ 'S': 静的制御を使用してデータストアがチェックされた場合
	ERR_MESS	違反があった制約に関連するエラー・メッセージ
	CHECK_DATE	データストアがチェックされた日時
	ORIGIN	チェック操作の実行元。この列は、チェックの実行方法によって、データストア名、もしくはインタフェース名および ID のどちらかに設定されます。
	CONS_NAME	違反があった制約の名前
	CONS_TYPE	制約の種類 : ■ 'PK': 主キー ■ 'AK': 代替キー ■ 'FK': 外部キー ■ 'CK': チェック条件 ■ 'NN': 必須列

標準の CKM は、次の手順で構成されます。

- サマリー表を削除して作成します。DROP 文は、サマリー表のリセットに関して設計者が要求する場合のみ実行されます。CREATE 文は常に実行されますが、表がすでに存在する場合は、エラーが許可されます。
- 前の実行のサマリー・レコードをサマリー表から削除します。
- エラー表を削除して作成します。DROP 文は、エラー表の再作成に関して設計者が要求する場合のみ実行されます。CREATE 文は常に実行されますが、表がすでに存在する場合は、エラーが許可されます。
- 前の実行で拒否されたレコードをエラー表から削除します。
- 主キー制約に違反するレコードを拒否します。
- 代替キー制約に違反するレコードを拒否します。
- 外部キー制約に違反するレコードを拒否します。
- チェック条件制約に違反するレコードを拒否します。
- 必須列制約に違反するレコードを拒否します。
- 必要に応じて、チェックした表から拒否されたレコードを削除します。
- サマリー表に、検出されたエラーのサマリーを挿入します。

CKM コマンドにタグを付けて、コードの生成方法を指定する必要があります。使用できるタグは次のとおりです。

- 主キー : コマンドによって、主キー制約のチェックに必要なコードが定義されます。
- 代替キー : コマンドによって、代替キー制約のチェックに必要なコードが定義されます。Oracle Data Integrator では、コードの生成時にそれぞれの代替キーに対してこのコマンドが使用されます。
- 結合 : コマンドによって、外部キー制約のチェックに必要なコードが定義されます。Oracle Data Integrator では、コードの生成時にそれぞれの外部キーに対してこのコマンドが使用されます。
- 条件 : コマンドによって、条件制約のチェックに必要なコードが定義されます。Oracle Data Integrator では、コードの生成時にそれぞれのチェック条件に対してこのコマンドが使用されます。
- 必須 : コマンドによって、必須列制約のチェックに必要なコードが定義されます。Oracle Data Integrator では、コードの生成時に必須列に対してこのコマンドが使用されます。
- エラー削除 : コマンドによって、チェックした表から拒否されたレコードを削除するのに必要なコードが定義されます。

CKM Oracle からの抜粋を次に示します。

<table>
<tr><th>手順</th><th>コード例</th><th>実行する条件</th></tr>
<tr><td>チェック表の削除</td><td><pre>drop table
<%=snpRef.getTable("L","CHECK_NAME","W")%></pre></td><td>常時。エラー許可。</td></tr>
<tr><td>チェック表の作成</td><td><pre>create table
<%=snpRef.getTable("L","CHECK_NAME","W")%>
(
       CATALOG_NAME
       <%=snpRef.getDataType("DEST_VARCHAR",
"100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
,
       SCHEMA_NAME
       <%=snpRef.getDataType("DEST_VARCHAR",
"100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
,
       RESOURCE_NAME
       <%=snpRef.getDataType("DEST_VARCHAR",
"100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
,
       FULL_RES_NAME
       <%=snpRef.getDataType("DEST_VARCHAR",
"100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
,
       ERR_TYPE
       <%=snpRef.getDataType("DEST_VARCHAR", "1",
"")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,
       ERR_MESS
       <%=snpRef.getDataType("DEST_VARCHAR",
"250", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
,
       CHECK_DATE
       <%=snpRef.getDataType("DEST_DATE", "",
"")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,
       ORIGIN
       <%=snpRef.getDataType("DEST_VARCHAR",
"100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
,
       CONS_NAME
       <%=snpRef.getDataType("DEST_VARCHAR", "35",
"")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,
       CONS_TYPE
       <%=snpRef.getDataType("DEST_VARCHAR", "2",
"")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,
       ERR_COUNT
       <%=snpRef.getDataType("DEST_NUMERIC", "10",
"")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>
)</pre></td><td>常時。エラー許可。</td></tr>
</table>

手順	コード例	実行する条件
エラー表の作成	create table <%=snpRef.getTable("L","ERR_NAME", "W")%> (ROW_ID ROWID, ERR_TYPE <%=snpRef.getDataType("DEST_VARCHAR", "1", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, ERR_MESS <%=snpRef.getDataType("DEST_VARCHAR", "250", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, CHECK_DATE <%=snpRef.getDataType("DEST_DATE", "", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, <%=snpRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + snpRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>, ORIGIN <%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, CONS_NAME <%=snpRef.getDataType("DEST_VARCHAR", "35", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, CONS_TYPE <%=snpRef.getDataType("DEST_VARCHAR", "2", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>)	常時。エラー許可。

手順	コード例	実行する条件
PK エラーの隔離	insert into <%=snpRef.getTable("L","ERR_NAME", "W")%> (ROW_ID, ERR_TYPE, ERR_MESS, ORIGIN, CHECK_DATE, CONS_NAME, CONS_TYPE, <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>) select rowid, '<%=snpRef.getInfo("CT_ERR_TYPE")%>', '<%=snpRef.getPK("MESS")%>', '<%=snpRef.getInfo("CT_ORIGIN")%>', <%=snpRef.getInfo("DEST_DATE_FCT")%>, '<%=snpRef.getPK("KEY_NAME")%>', 'PK', <%=snpRef.getColList("", snpRef.getTargetTable("TABLE_ALIAS")+".[COL_NAME]", ",\n\t", "", "MAP")%> from <%=snpRef.getTable("L", "CT_NAME", "A")%> <%=snpRef.getTargetTable("TABLE_ALIAS")%> where (<%=snpRef.getColList("",snpRef.getTargetTable("TABLE_ALIAS")+".[COL_NAME]", ",\n\t\t", "", "PK")%>) in (select <%=snpRef.getColList("", "[COL_NAME]", ",\n\t\t\t", "", "PK")%> from <%=snpRef.getTable("L","CT_NAME","A")%> group by <%=snpRef.getColList("", "[COL_NAME]", ",\n\t\t\t", "", "PK")%> having count(1) > 1) <%=snpRef.getFilter()%>	主キー。
チェックした表からのエラーの削除	delete from <%=snpRef.getTable("L", "CT_NAME", "A")%> T where T.rowid in(select ROW_ID from <%=snpRef.getTable("L","ERR_NAME", "W")%>)	エラー削除。

注意 : CKM を使用してインタフェースからフロー制御を実行する場合は、許可するエラーの最大数を定義できます。この数は、ログ・カウンタがエラーに設定されている CKM で各コマンドによって戻されたレコードの合計数と比較されます。

ケース・スタディ : CKM をカスタマイズして存在しない参照を動的に作成する

データ・ウェアハウスをロードする場合など、受け取ったレコードが他の表のデータを参照する必要があるにもかかわらず、参照されるレコードがまだ存在していないことがあります。

たとえば、1 日の販売取引レコードを受け取り、これらのレコードが商品 SKU を参照するとします。商品表に商品が存在しない場合は、標準 CKM のデフォルトの動作では、販売取引レコードが拒否され、データ・ウェアハウスにロードされずにエラー表に格納されます。ただし、プロジェクトの要件を満たすためには、この販売レコードをデータ・ウェアハウスにロードし、その場で空の商品を作成してデータの一貫性を確保する必要があります。その後、データ分析者は、単純にエラー表を分析して商品表に自動的に追加された商品の欠落情報を作成します。

この例を図示すると次のようになります。

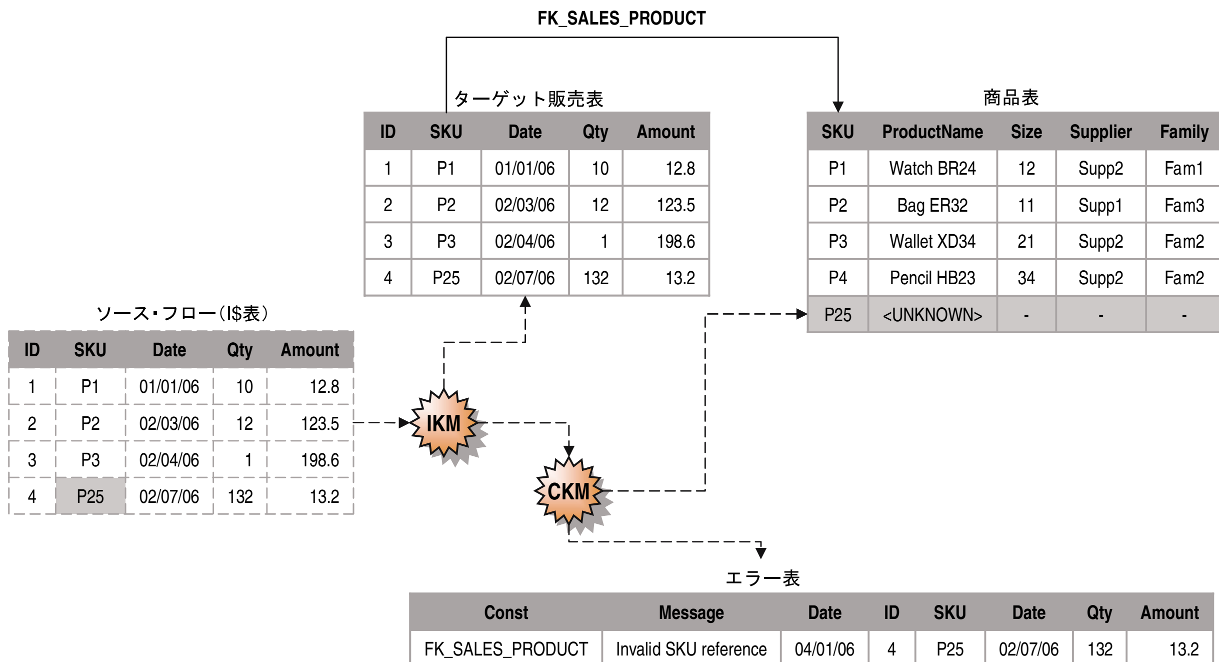


図 4-1 その場での参照の作成

– ソース・フロー・データは、IKM によって I$ 表にステージングされます。IKM によって CKM がコールされ、データ品質がチェックされます。

– CKM によって、ターゲットの販売表および商品表の間で定義された FK_SALES_PRODUCT 外部キーを含む、それぞれの制約がチェックされます。商品表に商品 P25 が存在しないため、レコード ID 4 が拒否されてエラー表に格納されます。

– CKM によって欠落している P25 の参照が商品表に挿入され、商品名に <UNKNOWN> という値が割り当てられます。その他のすべての列は空またはデフォルトの値に設定されます。

– ソース・フロー I$ 表の一貫性が確保されたため、拒否されたレコードは CKM によってソース・フロー I$ 表から削除されません。

– IKM によってフロー・データがターゲットに書き込まれます。

このような CKM を実装する場合、Oracle Data Integrator のデフォルト・メタデータで一部の情報が欠落していることに気付きます。たとえば、各外部キーに対して、参照される表の <UNKNOWN> 値（この場合は ProductName）を保持する列の名前を定義しておくと便利です。また、すべての外部キーが同様に動作するわけではないため、この制約で欠落している参照を自動作成する必要があるかどうかを説明するインジケータを、それぞれの外部キーに使用すると便利です。この追加情報は、Oracle Data Integrator Security の「Reference」オブジェクト上で単純にフレックス・フィールドを追加すると取得できます。FK_SALES_PRODUCT 制約では、次の図に示すように、このメタデータの入力が許可されます。フレックス・フィールドの詳細は、Oracle Data Integrator のドキュメントを参照してください。

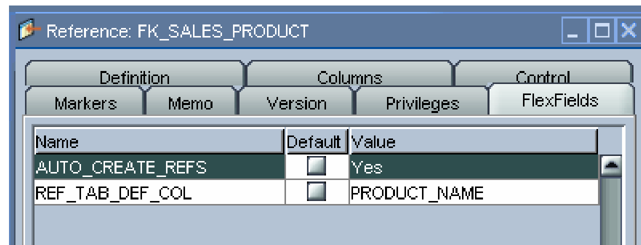


図 4-2 FK_SALES_PRODUCT 外部キーへのフレックス・フィールドの追加

これで、必要なすべてのメタデータが用意できました。要件に合せてデフォルトの CKM を拡張できます。したがって、CKM の手順は次のようになります。

– サマリー表を削除して作成します。
– 前の実行のサマリー・レコードをサマリー表から削除します。
– エラー表を削除して作成します。**追加の列をエラー表に追加して、制約の動作を格納します。**
– 前の実行で拒否されたレコードをエラー表から削除します。
– 主キー制約に違反するレコードを拒否します。
– 各代替キー制約に違反するレコードを拒否します。
– 各外部キー制約に違反するレコードを拒否します。
– **各外部キーについて、AUTO_CREATE_REFS が yes に設定されている場合は、参照される表に欠落している参照を挿入します。**
– 各チェック条件制約に違反するレコードを拒否します。
– 各必須列制約に違反するレコードを拒否します。
– 必要に応じて、チェックした表から拒否されたレコードを削除します。**制約の動作が Yes に設定されているレコードは削除しません。**
– サマリー表に、検出されたエラーのサマリーを挿入します。

このような CKM の実装の詳細を次に示します。

<table>
<tr><th>手順</th><th>Teradata 用コード例</th></tr>
<tr><td>エラー表の作成</td><td>

```
create multiset table <%=odiRef.getTable("L","ERR_NAME", "A")%>,
       no fallback,
       no before journal,
       no after journal
(
       AUTO_CREATE_REFS varchar(3),
       ERR_TYPE       varchar(1) ,
       ERR_MESS       varchar(250)  ,
       CHECK_DATE     timestamp ,
       <%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " +
odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>,
       ORIGIN         varchar(100) ,
       CONS_NAME      varchar(35) ,
       CONS_TYPE      varchar(2) ,
)
```

</td></tr>
<tr><td>FK エラーの隔離</td><td>

```
insert into <%=odiRef.getTable("L","ERR_NAME", "A")%>
(
       AUTO_CREATE_REFS,
       ERR_TYPE,
       ERR_MESS,
       CHECK_DATE,
       ORIGIN,
       CONS_NAME,
       CONS_TYPE,
       <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>
)
select '<%=odiRef.getFK("AUTO_CREATE_REFS")%>',
       '<%=odiRef.getInfo("CT_ERR_TYPE")%>',
       '<%=odiRef.getFK("MESS")%>',
       <%=odiRef.getInfo("DEST_DATE_FCT")%>,
       '<%=odiRef.getInfo("CT_ORIGIN")%>',
       '<%=odiRef.getFK("FK_NAME")%>',
       'FK',
[... etc.]
```

</td></tr>
</table>

<table>
<tr><th>手順</th><th>Teradata 用コード例</th></tr>
<tr><td>欠落している参照の挿入</td><td>

```
<% if  (odiRef.getFK("AUTO_CREATE_REFS").equals("Yes")) { %>
insert into <%=odiRef.getTable("L", "FK_PK_TABLE_NAME", "A")%>
(
<%=odiRef.getFKColList("", "[PK_COL_NAME]", ",", "")%>
, <%=odiRef.getFK("REF_TAB_DEF_COL")%>
)
select distinct
<%=odiRef.getFKColList("", "[COL_NAME]", ",", "")%>
,'<UNKNOWN>'
from  <%=odiRef.getTable("L","ERR_NAME", "A")%>
where
       CONS_NAME = '<%=odiRef.getFK("FK_NAME")%>'
And    CONS_TYPE = 'FK'
And    ORIGIN = '<%=odiRef.getInfo("CT_ORIGIN")%>'
And    AUTO_CREATE_REFS = 'Yes'
<%}%>
```

</td></tr>
<tr><td>チェックした表からの拒否されたレコードの削除</td><td>

```
delete from <%=odiRef.getTable("L", "CT_NAME", "A")%>
where  exists (
              select 'X'
              from   <%=odiRef.getTable("L","ERR_NAME", "A")%> as E
              where  <%=odiRef.getColList("","(("+odiRef.getTable("L",
"CT_NAME", "A")+".[COL_NAME]\t= E.[COL_NAME]) or
("+odiRef.getTable("L", "CT_NAME", "A")+".[COL_NAME] is null and
E.[COL_NAME] is null))", "\n\t\tand\t", "", "UK")%>
              and E.AUTO_CREATE_REFS <> 'Yes'
)
[... etc.]
```

</td></tr>
</table>

5 ロード戦略（LKM）

エージェントの使用方法

エージェントは、ソース・サーバー上で JDBC を使用して結果セットを読み取ることができ、JDBC を使用してこの結果セットをターゲット・ステージング領域サーバーの C$ 表に書き込むことができます。このメソッドを使用するには、Oracle Data Integrator のドキュメントに記載されているように、ナレッジ・モジュールに SELECT/INSERT 文が含まれている必要があります。このメソッドでは、配列フェッチ機能を使用してデータが配列の行ごとに読み取られ、バッチ更新機能を使用して行ごとに書き込まれるため、大量のデータには適していません。

この戦略を使用する一般的な LKM には、次の手順が含まれます。

手順	コード例
ステージング領域から C$ 表を削除します。この表が存在しない場合はエラーを無視します。	`drop table <%=odiRef.getTable("L", "COLL_NAME", "A")%>`

ステージング領域で C$ 表を作成します。	`create table <%=odiRef.getTable("L", "COLL_NAME", "A")%>`
	`(`
	`<%=odiRef.getColList("", "[CX_COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "","")%>`
	`)`

<table>
<tr><th>手順</th><th>コード例</th></tr>
<tr><td>SELECT/INSERT コマンドを使用してソース結果セットを C$ 表にロードします。

ソース上で SELECT およびステージング領域で INSERT がそれぞれ実行されます。エージェントは、JDBC API を使用してメモリー内でデータ型変換を実行します。</td><td>ソース・サーバーによって実行されるソース・タブ上のコード：<pre>
select <%=odiRef.getPop("DISTINCT_ROWS")%>
       <%=odiRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP]
[CX_COL_NAME]", ",\n\t", "", "")%>
from   <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getJoin()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
</pre>ステージング領域で実行されるターゲット・タブ上のコード：<pre>
insert into <%=odiRef.getTable("L", "COLL_NAME", "A")%>
(
       <%=odiRef.getColList("", "[CX_COL_NAME]", ",\n\t", "","")%>
)
values
(
       <%=odiRef.getColList("", ":[CX_COL_NAME]", ",\n\t", "","")%>
)
</pre></td></tr>
<tr><td>ターゲット内で IKM による統合が終了した後、C$ 表を削除します。この手順は、開発者がデバッグのために C$ 表を保持できるようにするためのオプションの値に依存させることができます。</td><td><pre>
drop table <%=odiRef.getTable("L", "COLL_NAME", "A")%>
</pre></td></tr>
</table>

ローダーの使用方法

フラット・ファイルでのローダーの使用方法

インタフェースにフラット・ファイルがソースとして含まれている場合、標準の LKM File to SQL ではなく、ステージング領域のテクノロジに最も効率的なロード・ユーティリティを利用できる戦略を使用できます。ほとんどの RDBMS には、フラット・ファイルを表にロードするための高速ロード・ユーティリティが含まれています。

Oracle で作業する場合は、SQL*LOADER または EXTERNAL TABLE のどちらかを使用できます。

Teradata は、3 つ異なるのユーティリティを提案しています。これらは、空の表に大きいファイルをロードするための FastLoad、大きいファイルの複雑なロード（増分ロードを含む）に使用できる MultiLoad、および小さいファイルの継続したロードに使用できる TPump です。

LKM で必要なのは、単純に C$ ステージング領域にファイルをロードすることです。すべての変換は、RDBMS で IKM によって実行されます。したがって、ロード・ユーティリティを使用する一般的な LKM では、次の手順が行われます。

- ステージング領域で C$ 表を削除し、作成します。
- C$ ステージング表にファイルをロードするために、ロード・ユーティリティによって要求されるスクリプトを生成します。
- 適切なオペレーティング・システム・コマンドを実行してロードを開始し、リターン・コードをチェックします。
- ユーティリティによってログ・ファイルが作成された場合は、このファイルをエラー処理のために分析します。
- 統合 KM が終了したら、C$ 表を削除します。

次の表は、この戦略を使用する LKM File to Oracle（EXTERNAL TABLE）からの抜粋を示しています。完全なコードは、KM を参照してください。

手順	コード例
Oracle ディレクトリの作成	`create or replace directory dat_dir AS '<%=snpRef.getSrcTablesList("", "[SCHEMA]", "", "")%>'`

外部表の作成	`create table <%=snpRef.getTable("L", "COLL_NAME", "W")%>`
	`(`
	`        <%=snpRef.getColList("", "[CX_COL_NAME]\t[DEST_WRI_DT]", ",\n\t", "","")%>`
	`)`
	`ORGANIZATION EXTERNAL`
	`(`
	`        TYPE ORACLE_LOADER`
	`        DEFAULT DIRECTORY dat_dir`
	`        ACCESS PARAMETERS`
	`<% if (snpRef.getSrcTablesList("", "[FILE_FORMAT]", "", "").equals("F")) {%>(`
	`                RECORDS DELIMITED BY NEWLINE`
	`                <%=snpRef.getUserExit("EXT_CHARACTERSET")%>`
	`                <%=snpRef.getUserExit("EXT_STRING_SIZE")%>`

手順 | **コード例**

```
              BADFILE       '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.bad'
              LOGFILE       '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.log'
              DISCARDFILE'<%=snpRef.getSrcTablesList("", "[RES_NAME]",
"", "")%>_%a.dsc'
              SKIP          <%=snpRef.getSrcTablesList("",
"[FILE_FIRST_ROW]", "", "")%>
              FIELDS
              <%=snpRef.getUserExit("EXT_MISSING_FIELD")%>
              (
                     <%=snpRef.getColList("",
"[CX_COL_NAME]\tPOSITION([FILE_POS]\\:[FILE_END_POS])", ",\n\t\t\t",
"","")%>
              )
       )
<%} else {%>(
              RECORDS DELIMITED BY NEWLINE
              <%=snpRef.getUserExit("EXT_CHARACTERSET")%>
              <%=snpRef.getUserExit("EXT_STRING_SIZE")%>
              BADFILE       '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.bad'
              LOGFILE       '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.log'
              DISCARDFILE'<%=snpRef.getSrcTablesList("", "[RES_NAME]",
"", "")%>_%a.dsc'
              SKIP          <%=snpRef.getSrcTablesList("",
"[FILE_FIRST_ROW]", "", "")%>
              FIELDS  TERMINATED BY  '<%=snpRef.getSrcTablesList("",
"[SFILE_SEP_FIELD]", "", "")%>'
              <% if(snpRef.getSrcTablesList("", "[FILE_ENC_FIELD]", "",
"").equals("")){%>
              <%} else {%>OPTIONALLY ENCLOSED BY
'<%=snpRef.getSrcTablesList("", "[FILE_ENC_FIELD]", "",
"").substring(0,1)%>' AND '<%=snpRef.getSrcTablesList("",
"[FILE_ENC_FIELD]", "", "").substring(1,2)%>' <%}%>
              <%=snpRef.getUserExit("EXT_MISSING_FIELD")%>
              (
                     <%=snpRef.getColList("", "[CX_COL_NAME]",
",\n\t\t\t", "","")%>
              )
       )
<%}%>  LOCATION (<%=snpRef.getSrcTablesList("", "'[RES_NAME]'", "",
"")%>)
)
<%=snpRef.getUserExit("EXT_PARALLEL")%>
REJECT LIMIT <%=snpRef.getUserExit("EXT_REJECT_LIMIT")%>
NOLOGGING
```

リモート・サーバーでのアンロード / ロードの使用方法

ソースの結果セットがリモート・データベース・サーバー上にある場合、エージェントを使用してデータを転送するかわりに、データをファイルにアンロードしてからステージング領域にロードする方法があります。通常、大量のデータを処理する場合には、この方法が最も効率的です。多くの場合、この戦略に従う LKM の手順は、次のようになります。

- ステージング領域で C$ 表を削除し、作成します。
- ソース・アンロード・ユーティリティ（MSSQL bcp または DB2 unload など）または OdiSqlUnload ツールを使用して、ソースから一時フラット・ファイルへデータをアンロードします。
- C$ ステージング表に一時ファイルをロードするために、ロード・ユーティリティによって要求されるスクリプトを生成します。
- 適切なオペレーティング・システム・コマンドを実行してロードを開始し、リターン・コードをチェックします。
- 必要に応じて、エラー処理のために、ユーティリティによって作成されたログ・ファイルを分析します。
- 統合 KM が終了したら、C$ 表を削除します。

LKM SQL to Teradata（TPUMP-FASTLOAD-MULTILOAD）は、これらの手順に従い、汎用の OdiSqlUnload ツールを使用してすべてのリモート RDBMS からデータをアンロードします。もちろん、ソース RDBMS に高速アンロード・ユーティリティが装備されている場合は、この KM を最適化できます。

次の表は、この LKM からのコードの抜粋を示しています。

手順	コード例
OdiSqlUnload を使用してソースからデータをアンロードします。	（以下のコードを参照）

```
OdiSqlUnload "-DRIVER=<%=snpRef.getInfo("SRC_JAVA_DRIVER")%>"
       "-URL=<%=snpRef.getInfo("SRC_JAVA_URL")%>"
       "-USER=<%=snpRef.getInfo("SRC_USER_NAME")%>"
       "-PASS=<%=snpRef.getInfo("SRC_ENCODED_PASS")%>"
       "-FILE_FORMAT=variable"
       "-FIELD_SEP=<%=snpRef.getOption("FIELD_SEP")%>"
       "-FETCH_SIZE=<%=snpRef.getInfo("SRC_FETCH_ARRAY")%>"
       "-DATE_FORMAT=<%=snpRef.getOption("UNLOAD_DATE_FMT")%>"
"-FILE=<%=snpRef.getOption("TEMP_DIR")%>/<%=snpRef.getTable("L",
"COLL_NAME", "W")%>"


select <%=snpRef.getPop("DISTINCT_ROWS")%>
<%=snpRef.getColList("", "\t[EXPRESSION]", ",\n", "", "")%>
from   <%=snpRef.getFrom()%>
where  (1=1)
<%=snpRef.getJoin()%>
<%=snpRef.getFilter()%>
<%=snpRef.getJrnFilter() %>
<%=snpRef.getGrpBy()%>
<%=snpRef.getHaving()%>
```

Oracle Data Integrator には、この戦略を使用する次のナレッジ・モジュールが同梱されています。

パイプ処理されたアンロード / ロードの使用方法

アンロード / ロード戦略を使用する場合、データを 2 度ステージングする必要があります。最初は一時ファイル、2 度目は C$ 表でのステージングです。そのため、ディスク領域が余分に使用され、能率上の問題が発生する可能性があります。より効率的な代替方法は、アンロードおよびロードのユーティリティ間でパイプラインを使用することです。あいにく、一部のオペレーティング・システムではファイルベースのパイプライン（FIFO）がサポートされていません。

エージェントが UNIX 上にインストールされている場合は、パイプ処理されたアンロード / ロード戦略を使用できます。LKM で実行する手順は次のようになります。

- ステージング領域で C$ 表を削除し、作成します。
- オペレーティング・システム上で、（たとえば UNIX では mkfifo コマンドを使用して）パイプライン・ファイルを作成します。
- C$ ステージング表に一時ファイルをロードするために、ロード・ユーティリティによって要求されるスクリプトを生成します。
- 適切なオペレーティング・システム・コマンドを実行して、デタッチ済プロセスとして（コマンドの最後に & を使用して）ロードを開始します。ロードが開始すると、すぐに FIFO 内のデータを待機します。
- ソース・アンロード・ユーティリティ（MSSQL bcp または DB2 unload など）または OdiSqlUnload ツールを使用して、ソース RDBMS のデータの FIFO へのアンロードを開始します。
- ロード・プロセスを結合して終了するまで待機します。処理中エラーをチェックします。
- 必要に応じて、追加のエラー処理のために、ユーティリティによって作成されたログ・ファイルを分析します。
- 統合 KM が終了したら、C$ 表を削除します。

Oracle Data Integrator は、この戦略を使用する LKM SQL to Teradata（パイプ処理された TPUMP-FAST-MULTILOAD）を提供しています。それぞれのデタッチ済プロセス（またはスレッド）の動作をより詳細に制御するために、この KM は Jython を使用して記述されています。OdiSqlUnload ツールも、コール可能オブジェクトとして Jython で使用できます。次の表は、この LKM からのコードの抜粋を示しています。完全なコードは、実際の KM を参照してください。

手順	コード例
OdiSqlUnload コマンドのトリガーに使用される Jython の関数	(下記コード参照)

```
import com.sunopsis.dwg.tools.OdiSqlUnload as JOdiSqlUnload
import java.util.Vector as JVector
import java.lang.String
from jarray import array
...
srcdriver     = "<%=snpRef.getInfo("SRC_JAVA_DRIVER")%>"
srcurl = "<%=snpRef.getInfo("SRC_JAVA_URL")%>"
srcuser = "<%=snpRef.getInfo("SRC_USER_NAME")%>"
srcpass = "<%=snpRef.getInfo("SRC_ENCODED_PASS")%>"
fetchsize     = "<%=snpRef.getInfo("SRC_FETCH_ARRAY")%>"
...

query  = """select<%=snpRef.getPop("DISTINCT_ROWS")%>
<%=snpRef.getColList("", "\t[EXPRESSION]", ",\n", "", "")%>
from   <%=snpRef.getFrom()%>
where  (1=1)
<%=snpRef.getJoin()%>
<%=snpRef.getFilter()%>
<%=snpRef.getJrnFilter() %>
<%=snpRef.getGrpBy()%>
<%=snpRef.getHaving()%>
"""
...
def odisqlunload():
       odiunload = JOdiSqlUnload()

       # Set the parameters
       cmdline = JVector()
       cmdline.add(array(["-DRIVER", srcdriver], java.lang.String))
       cmdline.add(array(["-URL", srcurl], java.lang.String))
       cmdline.add(array(["-USER", srcuser], java.lang.String))
       cmdline.add(array(["-PASS", srcpass], java.lang.String))
       cmdline.add(array(["-FILE_FORMAT", "variable"],
java.lang.String))
       cmdline.add(array(["-FIELD_SEP", fieldsep], java.lang.String))
       cmdline.add(array(["-FETCH_SIZE", fetchsize], java.lang.String))
       cmdline.add(array(["-FILE", pipename], java.lang.String))
       cmdline.add(array(["-DATE_FORMAT", datefmt], java.lang.String))
       cmdline.add(array(["-QUERY", query], java.lang.String))

       odiunload.setParameters(cmdline)

       # Start the unload process
       odiunload.execute()
```

手順	コード例
パイプ処理されたロードを実行する主要関数	(code below)

```
...

utility= "<%=snpRef.getOption("TERADATA UTILITY")%>"
if utility == "multiload":
       utilitycmd="mload"
else:
       utilitycmd=utility

# when using Unix pipes, it is important to get the pid
# command example : load < myfile.script > myfile.log & echo $!>
mypid.txt ; wait $!
# Note: the PID is stored in a file to be able to kill the fastload in
case of crash

loadcmd= '%s < %s > %s & echo $!> %s ; wait $!'%
(utilitycmd,scriptname, logname, outname)

...

def pipedload():
       # Create or Replace a Unix FIFO
       os.system("rm %s" % pipename)
       if os.system("mkfifo %s" % pipename) <> 0:
              raise "mkfifo error", "Unable to create FIFO %s" %
pipename

       # Start the load command in a dedicated thread
       loadthread = threading.Thread(target=os.system,
                                          args=(loadcmd,),
                                          name="snptdataload")
       loadthread.start()
       # now that the fastload thead has started, wait
       # 3 seconds to see if it is running
       time.sleep(3)
       if not loadthread.isAlive():
              os.system("rm %s" % pipename)
              raise "Load error", "(%s) load process not started" %
loadcmd

       # Start the SQLUnload process
       try:
              OdiSqlUnload()
       except:
              # if the unload process fails, we have to kill
              # the load process on the OS.
```

手順	コード例

	`# Several methods are used to get sure the process is killed`
	`# get the pid of the process`
	`f = open(outname, 'r')`
	`pid = f.readline().replace('\n', '').replace('\r', '')`
	`f.close()`
	`# close the pipe by writing something fake in it`
	`os.system("echo dummy > %s" % pipename)`
	`# attempt to kill the process`
	`os.system("kill %s" % pid)`
	`# remove the pipe`
	`os.system("rm %s" % pipename)`
	`raise`
	
	`# At this point, the unload() process has finished, so we need to wait`
	`# for the load process to finish (join the thread)`
	`loadthread.join()`

RDBMS 固有の戦略の使用方法

一部の RDBMS には、同じテクノロジのサーバー間でデータを共有するためのメカニズムが備わっています。たとえば、次のとおりです。

- Oracle には、2 つのリモート Oracle サーバー間でデータをロードするためのデータベース・リンクが装備されています。
- Microsoft SQL Server には、リンク・サーバーが装備されています。
- IBM DB2 400 には、DRDA によるファイル転送が装備されています。

6

統合戦略（IKM）

ターゲット上のステージング領域での IKM

単純な置換および追加

すべてのソース・データがすでにステージング領域に存在する場合、既存のターゲット表でデータを統合するための最も単純な戦略は、ターゲットでレコードを挿入することです。したがって、最も単純な IKM は、次の 2 つの手順で構成されます。

– ターゲット表からすべてのレコードを削除します。この手順は、インタフェースの設計者によって設定されるオプションに依存させることができます。

– すべてのソース・セットのソース・レコードを変換して挿入します。リモート・ソース・データを処理する場合、LKM には、事前変換済の結果セットを使用した C$ 表がすでに用意されています。ターゲット（およびステージング領域）と同じサーバーにあるソース・データセットをインタフェースが使用する場合、データセットは他の C$ 表に結合されます。そのため、統合操作は、ターゲット Teradata ボックスの変換能力をすべて利用する、単純な INSERT/SELECT 文になります。

これらの手順の詳細を次の例で示します。

手順	コード例
ターゲット表からデータを削除します。この手順は、「Delete all rows?」チェック・ボックス・オプションに依存させることができます。	`delete from <%=odiRef.getTable("L","INT_NAME","A")%>`

手順	コード例

フロー・レコードをターゲットに追加します。	`insert into <%=odiRef.getTable("L","INT_NAME","A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`)`
	`select <%=odiRef.getPop("DISTINCT_ROWS")%>`
	`<%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`from <%=odiRef.getFrom()%>`
	`where (1=1)`
	`<%=odiRef.getJoin()%>`
	`<%=odiRef.getFilter()%>`
	`<%=odiRef.getJrnFilter()%>`
	`<%=odiRef.getGrpBy()%>`
	`<%=odiRef.getHaving()%>`

この非常に単純な戦略は、デフォルトの Oracle Data Integrator KM ではこの状態で提供されていません。フロー・データを制御しないことを選択した場合に、制御追加 IKM の特殊なケースとして取得できます。

次項では、これらの KM の詳細を説明します。

データ品質チェックでの追加

前の例では、フロー・データは単純にターゲット表に挿入され、データ品質チェックは行われませんでした。誤ったレコードをエラー表（E$）に隔離するための CKM をコールする前に、統合表（I$）という一時表にフロー・データを格納する手順を追加することで、この方法を改良できます。このような IKM の手順は、次のようになります。

- ステージング領域でフロー表を削除し、作成します。データ品質チェックのために CKM に渡せるように、ターゲット表と同じ列を使用して I$ 表が作成されます。
- フロー・データを I$ 表に挿入します。すべてのソース・セットのソース・レコードが変換され、単一の INSERT/SELECT 文内で I$ 表に挿入されます。
- データ品質チェックのために CKM をコールします。ターゲット表に対して定義されたそれぞれの制約が、CKM によってフロー・データ上でシミュレートされます。これにより、エラー表が作成され、誤ったレコードが挿入されます。また、制御した表から誤ったレコードがすべて削除されます。したがって、CKM が完了すると、I$ 表には有効なレコードのみが含まれます。そのため、ターゲット表へのレコードの挿入を安全に実行できます。
- ターゲット表からすべてのレコードを削除します。この手順は、インタフェースの設計者によって設定されるオプションに依存させることができます。
- 単一の INSERT/SELECT 文内で、I$ 表のレコードをターゲット表に追加します。
- I$ 一時表を削除します。

また、場合によっては、フローに追加して再びターゲットに適用されるように、前のエラーをリサイクルすると便利です。この方法は、存在しない可能性がある商品 ID を参照する、1 日の販売取引を受け取る場合などに役立ちます。たとえば、参照される商品 ID が商品表に存在しないため、販売レコードが拒否され、エラー表に隔離されたとします。これは、インタフェースの最初に実行時に発生します。まもなく、データ管理者によって欠落している商品 ID が作成されます。これにより、拒否されたレコードが有効になるため、インタフェースの次の実行時にターゲットに適用しなおす必要が出てきます。

このメカニズムは、IKM できわめて簡単に実装できます。データ品質チェックのために CKM をコールする前に、前の実行で拒否されたレコードをすべてフロー表（I$）に挿入する追加の手順を追加するだけです。

また、この IKM を拡張して、単純な置換追加戦略をサポートできます。フロー制御の手順はオプションになります。設計者がデータ品質をチェックすることを選択した場合、データは I$ 表からターゲットに適用されます。それ以外の場合は C$ ステージング表などのソース・セットから適用されます。

このような IKM の手順の一部を次に示します。

手順	コード例	実行条件

ステージング領域でフロー表を作成します。	`create table <%=odiRef.getTable("L", "INT_NAME", "A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "INS")%>`
	`)`
		FLOW_CONTROL が YES に設定されている。

フロー・データを I$ 表に挿入します。	`insert into <%=odiRef.getTable("L","INT_NAME","A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`)`
	`select <%=odiRef.getPop("DISTINCT_ROWS")%>`
	`<%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`from   <%=odiRef.getFrom()%>`
	`where  (1=1)`
	`<%=odiRef.getJoin()%>`
	`<%=odiRef.getFilter()%>`
	`<%=odiRef.getJrnFilter()%>`
	`<%=odiRef.getGrpBy()%>`
	`<%=odiRef.getHaving()%>`
		FLOW_CONTROL が YES に設定されている。

手順	コード例	実行条件

拒否された前のレコードをリサイクルします。	insert into
	<%=odiRef.getTable("L","INT_NAME","A")%>
	(
	        <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "INS and REW")%>
	)
	select <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "INS and REW")%>
	from <%=odiRef.getTable("L","ERR_NAME", "A")%> <%=odiRef.getInfo("DEST_TAB_ALIAS_WORD")%> E
	where not exists(
	              select 'X'
	              from
	<%=odiRef.getTable("L","INT_NAME","A")%> <%=odiRef.getInfo("DEST_TAB_ALIAS_WORD")%> T
	              where <%=odiRef.getColList("", "T.[COL_NAME]\t= E.[COL_NAME]", "\n\t\tand\t", "", "UK")%>
	              )
	and E.ORIGIN= '<%=odiRef.getInfo("CT_ORIGIN")%>'
	and E.ERR_TYPE= '<%=odiRef.getInfo("CT_ERR_TYPE")%>'
		RECYCLE_ERRORS が Yes に設定されている。

データ品質チェックを実行するために CKM をコールします。	<%@ INCLUDE CKM_FLOW DELETE_ERRORS%>	FLOW_CONTROL が YES に設定されている。
ターゲット表からすべてのレコードを削除します。	delete from <%=odiRef.getTable("L","TARG_NAME","A")%>	DELETE_ALL が Yes に設定されている。

<table>
<tr><th>手順</th><th>コード例</th><th>実行条件</th></tr>
<tr><td>レコードを挿入します。フロー制御が Yes に設定されている場合、データは I$ 表から挿入されます。それ以外の場合は、ソース・セットから挿入されます。</td><td>

```
<%if (
odiRef.getOption("FLOW_CONTROL").equals("1") ) {
%>
insert into
<%=odiRef.getTable("L","TARG_NAME","A")%>
(
	<%=odiRef.getColList("", "[COL_NAME]",
",\n\t", "", "((INS and !TRG) and REW)")%>
	<%=odiRef.getColList(",", "[COL_NAME]",
",\n\t", "", "((INS and TRG) and REW)")%>
)
select <%=odiRef.getColList("", "[COL_NAME]",
",\n\t", "", "((INS and !TRG) and REW)")%>
	<%=odiRef.getColList(",", "[EXPRESSION]",
",\n\t", "", "((INS and TRG) and REW)")%>
from	<%=odiRef.getTable("L","INT_NAME","A")%>
<%} else {%>
insert into
<%=odiRef.getTable("L","TARG_NAME","A")%>
(
	<%=odiRef.getColList("", "[COL_NAME]",
",\n\t", "", "(INS and REW)")%>
)
select <%=odiRef.getPop("DISTINCT_ROWS")%>
	<%=odiRef.getColList("", "[EXPRESSION]",
",\n\t", "", "(INS and REW)")%>
from	<%=odiRef.getFrom()%>
where	<% if
(odiRef.getPop("HAS_JRN").equals("0")) { %>
	(1=1)
	<%} else {%>
	JRN_FLAG <> 'D'
	<% } %>
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
<% } %>
```

</td><td>INSERT が Yes に設定されている。</td></tr>
</table>

増分更新

増分更新戦略は、更新キーと呼ばれる一連の列に従って、フローのレコードをターゲットの既存のレコードと比較して、ターゲット表にデータを統合する場合に使用します。同じ更新キーを持つレコードは、関連付けられているデータが同じでない場合に更新されます。ターゲットに存在しないレコードは挿入されます。多くの場合、この戦略は、変更されたレコードを追跡する必要がない場合に、ディメンション表に使用されます。

このような IKM の課題は、パフォーマンスの問題につながることが多い行単位の処理方法を使用するかわりに、セット指向の SQL に基づくプログラミングを使用してすべての操作を実行することです。このような戦略を構築するための最も一般的な方法は、通常、変換済ソース・セットを格納する統合一時表（I$）に依存します。この方法の詳細は次のとおりです。

- ステージング領域でフロー表を削除し、作成します。データ品質チェックのために CKM に渡せるように、ターゲット表と同じ列を使用して I$ 表が作成されます。この表には、挿入対象のレコード（I）および更新対象のレコード（U）のフラグ付けに使用される IND_UPDATE 列も含まれます。
- フロー・データを I$ 表に挿入します。すべてのソース・セットのソース・レコードが変換され、単一の INSERT/SELECT 文内で I$ 表に挿入されます。IND_UPDATE 列は、デフォルトで I に設定されます。
- 設計者がエラーのリサイクルを選択した場合は、前の実行で拒否されたレコードを追加します。
- データ品質チェックのために CKM をコールします。ターゲット表に対して定義されたそれぞれの制約が、CKM によってフロー・データ上でシミュレートされます。エラー表が作成され、誤ったレコードがすべて挿入されます。また、チェックした表から誤ったレコードがすべて削除されます。したがって、CKM が完了すると、I$ 表には有効なレコードのみが含まれます。
- I$ 表を更新して、ターゲットと同じ更新キー値を持つすべてのレコードに対して IND_UPDATE 列を U に設定します。したがって、すでにターゲットに存在するレコードには U フラグが付きます。この手順には通常、UPDATE/SELECT 文を使用します。
- I$ 表を再び更新し、すでに U フラグが付いており、列の値がターゲットと完全に同じであるすべてのレコードに対して IND_UPDATE 列を N に設定します。これらのフロー・レコードは、ターゲット・レコードと完全に一致するため、ターゲット・データの更新に使用する必要はありません。この手順が終了すると、I$ 表にフラグ付きのレコードが含まれるため、ターゲットに変更を適用するための I$ 表の準備が完了します。
  - I: ターゲットに挿入するレコードです。
  - U: ターゲットの更新に使用するレコードです。
  - N: すでにターゲットに存在するため無視するレコードです。
- I$ 表の U フラグ付きレコードを使用してターゲットを更新します。操作するデータの量を最小限にするために、UPDATE 文は INSERT 文の前に実行してください。
- I$ 表の I フラグ付きレコードをターゲットに挿入します。
- I$ 一時表を削除します。

もちろん、この方法は、基礎になるデータベースに応じて最適化できます。たとえば、Teradata では、フロー・データとターゲット表の間に左側外部結合を使用して、IND_UPDATE 列がすでに適切に設定されている I$ 表を移入する方が効率的な場合があります。

注意： 更新キーは、常に一意であることが必要です。ほとんどの場合、主キーが更新キーとして使用されます。ただし、ID 列、ランク関数または順序のように、増分を使用して自動的に算出される場合は、主キーを使用できません。この場合は、ソースにある、列に基づく更新キーを使用する必要があります。

このような IKM の手順の一部を次に示します。

<table>
<tr><th>手順</th><th>コード例</th><th>実行条件</th></tr>
<tr><td>ステージング領域でフロー表を作成します。</td><td><pre>create <%=odiRef.getOption("FLOW_TABLE_TYPE")%>
table <%=odiRef.getTable("L", "INT_NAME", "A")%>,

 (

       <%=odiRef.getColList("",
"[COL_NAME]\t[DEST_WRI_DT] " +
odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "",
"")%>,

       IND_UPDATE char(1)

)</pre></td><td></td></tr>
<tr><td>（更新キーを使用して）更新の対象を指定します。</td><td><pre>update <%=odiRef.getTable("L", "INT_NAME", "A")%>

from   <%=odiRef.getTable("L", "TARG_NAME", "A")%>
T

set   IND_UPDATE = 'U'

where <%=odiRef.getColList("",
odiRef.getTable("L", "INT_NAME", "A") +
".[COL_NAME]\t= T.[COL_NAME]", "\nand\t", "",
"UK")%></pre></td><td>INSERT または UPDATE が Yes に設定されている。</td></tr>
<tr><td>データの比較によって更新の対象外を指定します。</td><td><pre>update <%=odiRef.getTable("L", "INT_NAME", "A")%>

from   <%=odiRef.getTable("L", "TARG_NAME", "A")%>
T

set   IND_UPDATE = 'N'

where <%=odiRef.getColList("",
odiRef.getTable("L", "INT_NAME", "A") +
".[COL_NAME]\t= T.[COL_NAME]", "\nand\t", "",
"UK")%>

and   <%=odiRef.getColList("","((" +
odiRef.getTable("L", "INT_NAME", "A") +
".[COL_NAME] = T.[COL_NAME]) or (" +
odiRef.getTable("L", "INT_NAME", "A") +
".[COL_NAME] IS NULL and T.[COL_NAME] IS NULL))",
" \nand\t", "", "((UPD and !TRG) and !UK) ")%></pre></td><td>UPDATE が Yes に設定されている。</td></tr>
<tr><td>既存のレコードでターゲットを更新します。</td><td><pre>update <%=odiRef.getTable("L", "TARG_NAME", "A")%>

from   <%=odiRef.getTable("L", "INT_NAME", "A")%>
S

set   <%=odiRef.getColList("", "[COL_NAME]\t=
S.[COL_NAME]", ",\n\t", "", "(((UPD and !UK) and
!TRG) and REW)")%>

       <%=odiRef.getColList(",",
"[COL_NAME]=[EXPRESSION]", ",\n\t", "", "(((UPD
and !UK) and TRG) and REW)")%>

where <%=odiRef.getColList("",
odiRef.getTable("L", "TARG_NAME", "A") +
".[COL_NAME]\t= S.[COL_NAME]", "\nand\t", "",
"(UK)")%>

and   S.IND_UPDATE= 'U'</pre></td><td>UPDATE が Yes に設定されている。</td></tr>
</table>

手順	コード例	実行条件
新しいレコードを挿入します。	insert into <%=odiRef.getTable("L","TARG_NAME","A")%> (<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%> <%=odiRef.getColList(",", "[COL_NAME]", ",\n\t", "", "((INS and TRG) and REW)")%>) select <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%> <%=odiRef.getColList(",", "[EXPRESSION]", ",\n\t", "", "((INS and TRG) and REW)")%> from <%=odiRef.getTable("L","INT_NAME","A")%> where IND_UPDATE = 'I'	INSERT が Yes に設定されている。

更新の対象外を指定するためにデータの値を比較する場合、I$ 表とターゲット表の間の結合は、列ごとに次のように表されます。

```
Target.ColumnN = I$.ColumnN or (Target.ColumnN is null and I$.ColumnN is null)
```

これを行うことで、NULL 値を別の NULL 値と一致させるための NULL 値の比較が許可されます。より簡潔に記述するには、結合関数を使用します。したがって、WHERE 条件は次のように記述できます。

```
<%=odiRef.getColList("","coalesce(" + odiRef.getTable("L", "INT_NAME", "A") +
".[COL_NAME], 0) = coalesce(T.[COL_NAME], 0)", " \nand\t", "", "((UPD and !TRG) and
!UK) ")%>
```

注意 : UPDATE 文によって更新された列は、INSERT 文で使用されるものと同じではありません。UPDATE 文では、セレクタ UPD and not UK を使用して、インタフェース内の Update のマークが付いたマッピングのうち、更新キーに属さないマッピングのみがフィルタ処理されます。INSERT 文では、セレクタ INS を使用して、インタフェース内の insert のマークが付いたマッピングのみが取得されます。

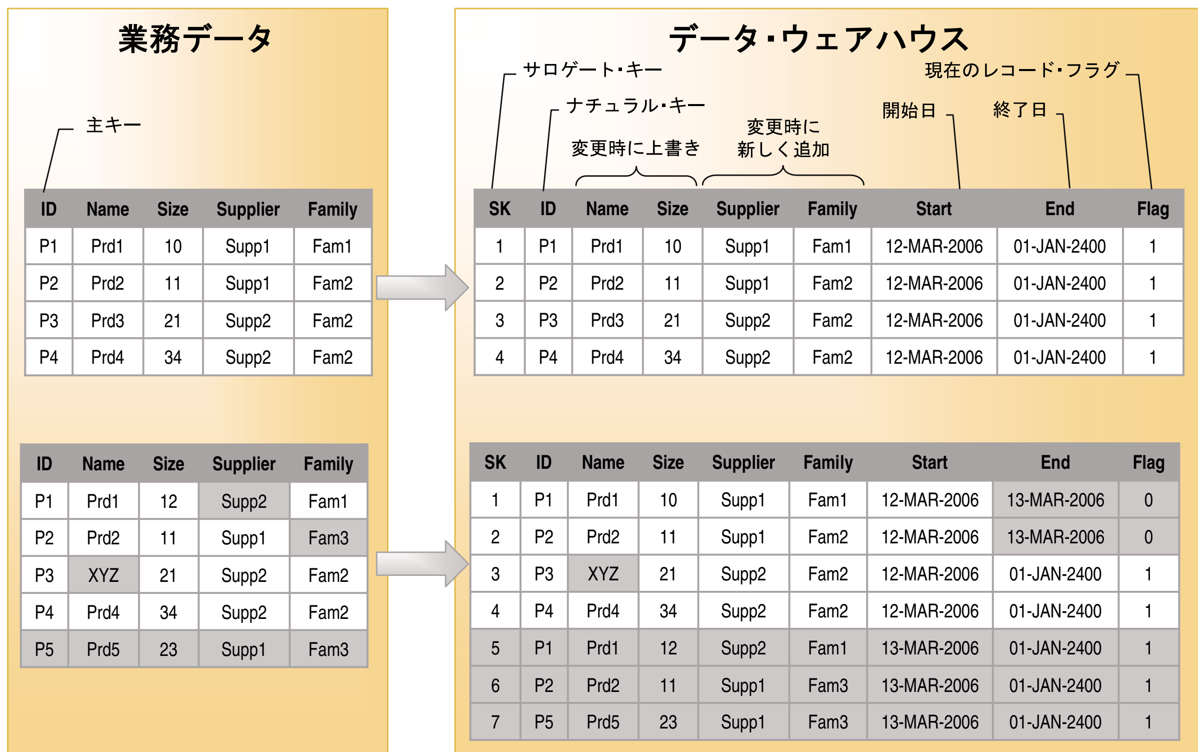
ターゲットの UPDATE 文および INSERT 文が、同じトランザクション（トランザクション 1）に属することが重要です。1 つでも条件を満たさない場合は、ターゲットへのデータの挿入または更新は一切行われません。

緩やかに変化するディメンション

タイプ 2 の緩やかに変化するディメンションは、最も一般的なデータ・ウェアハウス・ロード戦略の 1 つです。多くの場合、一部の列で発生した変更を追跡するために、ディメンション表のロードに使用されます。緩やかに変化するディメンションの一般的な表には、次の列が含まれます。

- 自動的に算出されるサロゲート・キー。通常は数値列で、ID 列、ランク関数または順序などの自動番号が含まれます。
- ナチュラル・キー。業務システムの実際の主キーを表す列のリストです。
- 変更時に上書きされる可能性がある列。
- 変更時に新しいレコードの作成が必要な列。
- データ・ウェアハウスでレコードが作成された日付を示す開始日列。
- レコードが不要になった日付（終了日）を示す終了日列。
- レコードが現在のレコード（1）か古いレコード（0）かを示す、現在のレコード・フラグ。

次の図は、ディメンションが緩やかに変化する商品の動作の例です。業務システムでは、商品は主キーの役割をする ID によって定義されます。各商品には、名前、サイズ、仕入先、および ファミリがあります。業務システムで仕入先またはファミリが更新されるたびに、この商品の新しいバージョンをデータ・ウェアハウスに格納する必要があります。



ID	Name	Size	Supplier	Family
P1	Prd1	10	Supp1	Fam1
P2	Prd2	11	Supp1	Fam2
P3	Prd3	21	Supp2	Fam2
P4	Prd4	34	Supp2	Fam2

SK	ID	Name	Size	Supplier	Family	Start	End	Flag
1	P1	Prd1	10	Supp1	Fam1	12-MAR-2006	01-JAN-2400	1
2	P2	Prd2	11	Supp1	Fam2	12-MAR-2006	01-JAN-2400	1
3	P3	Prd3	21	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1
4	P4	Prd4	34	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1

ID	Name	Size	Supplier	Family
P1	Prd1	12	Supp2	Fam1
P2	Prd2	11	Supp1	Fam3
P3	XYZ	21	Supp2	Fam2
P4	Prd4	34	Supp2	Fam2
P5	Prd5	23	Supp1	Fam3

SK	ID	Name	Size	Supplier	Family	Start	End	Flag
1	P1	Prd1	10	Supp1	Fam1	12-MAR-2006	13-MAR-2006	0
2	P2	Prd2	11	Supp1	Fam2	12-MAR-2006	13-MAR-2006	0
3	P3	XYZ	21	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1
4	P4	Prd4	34	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1
5	P1	Prd1	12	Supp2	Fam1	13-MAR-2006	01-JAN-2400	1
6	P2	Prd2	11	Supp1	Fam3	13-MAR-2006	01-JAN-2400	1
7	P5	Prd5	23	Supp1	Fam3	13-MAR-2006	01-JAN-2400	1

図 6-1 緩やかに変化するディメンションの例

この例では、2006 年 3 月 12 日にデータ・ウェアハウスで商品のディメンションが最初に初期化されています。すべてのレコードが挿入され、算出されたサロゲート・キーおよび 2400 年 1 月 1 日に設定された偽の終了日が割り当てられています。これらのレコードは業務システムの現在の状態を表すため、現在のレコード・フラグは 1 に設定されています。最初のロードの後、業務システムで次の変更が発生します。

1. 商品 P1 の仕入先が更新されます。
2. 商品 P2 のファミリが更新されます。
3. 商品 P3 の名前が更新されます。
4. 商品 P5 が追加されます。

これらの更新によるデータ・ウェアハウス・ディメンションへの影響は、次のとおりです。

1. P1 の仕入先の更新の結果、新しい現在のレコード（サロゲート・キー 5）が作成され、前のレコード（サロゲート・キー 1）が終了します。
2. P2 のファミリの更新の結果、新しい現在のレコード（サロゲート・キー 6）が作成され、前のレコード（サロゲート・キー 2）が終了します。
3. P3 の名前の更新では、単純にターゲット・レコードがサロゲート・キー 3 を使用して更新されます。
4. 新しい商品 P5 により、新しい現在のレコード（サロゲート・キー 7）が作成されます。

この動作を実装するナレッジ・モジュールを作成するには、サロゲート・キー、ナチュラル・キー、開始日などの役割をする列を把握しておく必要があります。Oracle Data Integrator では、次の図に示すように、ターゲットの緩やかに変化するディメンションのデータストアの各列について、追加のメタデータ・フィールドにこの情報を設定できます。

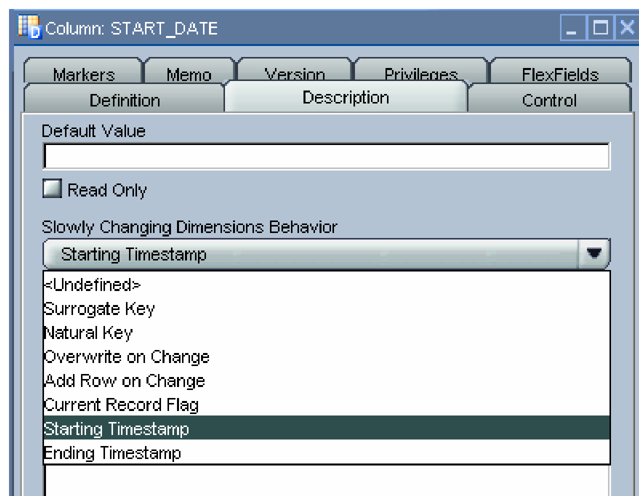


図 6-2 緩やかに変化するディメンション列の動作

インタフェースでこのようなデータストアを移入する場合、IKM は、getColList() 代替メソッドで SCD_xx セレクタを使用して、このメタデータにアクセスします。

Oracle Data Integrator では、タイプ 2 の緩やかに変化するディメンションの実装は次のように行われます。

– 異なるソース・セットからのフロー・データを保持するために、I$ フロー表を削除して作成します。
– ナチュラル・キーの列、変更時に上書きする列および変更時に行を追加する列に適用するマッピングのみを使用して、I$ 表にフロー・データを挿入します。開始日を現在の日付に設定し、終了日を定数に設定します。
– 拒否された前のレコードをリサイクルします。
– フローに対するデータ品質チェックを実行するために、CKM をコールします。

- ターゲットの現在のレコードと比較して、ナチュラル・キーの列および変更時に行を追加する列が変更されていない場合は、I$ 表のレコードに U フラグを付けます。
- U フラグでフィルタ処理された I$ フローを使用して、変更時に上書きされる列でターゲットを更新します。
- （I$ 表にナチュラル・キーが存在する）古いレコードを終了し、それらのレコードの現在のレコード・フラグを 0、終了日を現在の日付にそれぞれ設定します。
- 現在のレコード・フラグが 1 に設定されている、新しい変更レコードを挿入します。
- I$ 一時表を削除します。

前述のように、この方法はプロジェクト固有のニーズに合せることができます。場合によっては、作成された SQL を、さらに調整および最適化する必要があります。

Teradata の緩やかに変化するディメンション IKM の手順の一部を次に示します。

手順	コード例
MINUS 文を使用してフロー・データを I$ 表に挿入します。	(code below)

```
insert /*+ APPEND */ into <%=snpRef.getTable("L","INT_NAME","W")%>
(
       <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "(((INS OR
UPD) AND NOT TRG) AND REW)")%>,
       IND_UPDATE
)
select <%=snpRef.getUserExit("OPTIMIZER_HINT")%>
<%=snpRef.getPop("DISTINCT_ROWS")%>
       <%=snpRef.getColList("", "[EXPRESSION]", ",\n\t", "", "(((INS
OR UPD) AND NOT TRG) AND REW)")%>,
<%if (snpRef.getPop("HAS_JRN").equals("0")) {%>
       'I' IND_UPDATE
<%} else {%>
       JRN_FLAG
<%}%>
from   <%=snpRef.getFrom()%>
where  (1=1)
<%=snpRef.getJoin()%>
<%=snpRef.getFilter()%>
<%=snpRef.getJrnFilter() %>
<%=snpRef.getGrpBy()%>
<%=snpRef.getHaving()%>
minus
select
       <%=snpRef.getColList("","[COL_NAME]", ",\n\t", "", "(((INS OR
UPD) AND NOT TRG) AND REW)")%>,
       'I'    IND_UPDATE
from   <%=snpRef.getTable("L", "TARG_NAME", "A")%>
```

手順 / **コード例**

ターゲットで更新が必要なレコードにフラグを付けます。

```
update <%=snpRef.getTable("L", "INT_NAME", "W")%>
set    IND_UPDATE = 'U'
where  (<%=snpRef.getColList("","[COL_NAME]", ", ", "", "UK")%>)
       in     (
              select <%=snpRef.getColList("","[COL_NAME]",
",\n\t\t\t", "", "UK")%>
              from   <%=snpRef.getTable("L", "TARG_NAME", "A")%>
              )
```

ターゲットの更新可能な列を更新します。

```
update <%=snpRef.getTable("L", "TARG_NAME", "A")%> T
set    (
       <%=snpRef.getColList("", "T.[COL_NAME]", ",\n\t", "", "((UPD
AND (NOT UK) AND (NOT TRG)) AND REW)")%>
       <%=snpRef.getColList(",", "T.[COL_NAME]", ",\n\t", "", "((UPD
AND (NOT UK) AND TRG) AND REW)")%>
       ) =
              (
              select <%=snpRef.getColList("", "S.[COL_NAME]",
",\n\t\t\t", "", "((UPD AND (NOT UK) AND (NOT TRG)) AND REW)")%>
                     <%=snpRef.getColList(",", "[EXPRESSION]",
",\n\t\t\t", "", "((UPD AND (NOT UK) AND TRG) AND REW)")%>
              from   <%=snpRef.getTable("L", "INT_NAME", "W")%> S
              where  <%=snpRef.getColList("", "T.[COL_NAME]
=S.[COL_NAME]", "\n\t\tand\t", "", "UK")%>
               )
where  (<%=snpRef.getColList("", "[COL_NAME]", ", ", "", "UK")%>)
       in     (
              select <%=snpRef.getColList("", "[COL_NAME]",
",\n\t\t\t", "", "UK")%>
              from   <%=snpRef.getTable("L", "INT_NAME", "W")%>
              where  IND_UPDATE = 'U'
              )
```

新しいレコードを挿入します。

```
insert into<%=snpRef.getTable("L","TARG_NAME","A")%>
       (
       <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS AND
(NOT TRG)) AND REW)")%>
       <%=snpRef.getColList(",", "[COL_NAME]", ",\n\t", "", "((INS AND
TRG) AND REW)")%>
       )
select <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS AND
(NOT TRG)) AND REW)")%>
       <%=snpRef.getColList(",", "[EXPRESSION]", ",\n\t", "", "((INS
AND TRG) AND REW)")%>
from   <%=snpRef.getTable("L","INT_NAME","W")%>
where  IND_UPDATE = 'I'
```

ケース・スタディ：ロード前にターゲット表をバックアップする

プロジェクトの要件の 1 つが、現在のデータをロードする前に各データ・ウェアハウス表をバックアップすることであるとします。この要件は、たとえば大きな問題が発生したときに、データ・ウェアハウスを前の状態にリストアする場合に役立ちます。

この要件を解決する最初の方法として、各ターゲット・データストアのデータを対応するバックアップ・データストアに複製するためのインタフェースの開発があげられます。これらのインタフェースは、データ・ウェアハウスを移入するインタフェースより先にトリガーされます。あいにく、この解決方法では、ターゲット・データストアごとに追加のインタフェースを作成する必要があるため、大規模な開発およびメンテナンスが発生します。開発およびメンテナンスするインタフェースの数は、少なくとも 2 倍になります。

より簡潔な解決方法として、ターゲット・データストアの移入に使用する IKM で、この動作を実装することがあげられます。これは、ターゲットへの書込みの手順の直前にバックアップ表に書き込む単一の INSERT/SELECT 文を使用して行います。したがって、データのバックアップが自動化され、インタフェース開発者の介入が不要になります。

この例では、IKM 増分更新でこの動作を実装する方法を説明します。

- ステージング領域で I$ フロー表を削除し、作成します。
- フロー・データを I$ 表に挿入します。
- 拒否された前のレコードをリサイクルします。
- データ品質チェックのために CKM をコールします。
- I$ 表を更新して IND_UPDATE 列を U に設定します。
- I$ 表を再び更新して IND_UPDATE 列を N に設定します。
- **ロード前にターゲット表をバックアップします。**
- I$ 表の U フラグ付きレコードを使用してターゲットを更新します。
- I$ 表の I フラグ付きレコードをターゲットに挿入します。
- I$ 一時表を削除します。

バックアップ表の名前がターゲット表の名前の後に _BCK を追加したものと同じと仮定すると、バックアップ手順のコードは次のように表されます。

手順	コード例
バックアップ表を削除します。	`Drop table <%=odiRef.getTable("L","TARG_NAME","A")%>_BCK`

バックアップ表を作成および移入します。	`Create table <%=odiRef.getTable("L","TARG_NAME","A")%>_BCK as`
	`select <%=odiRef.getTargetColList("", "[COL_NAME]", ",", "")%>`
	`from <%=odiRef.getTable("L","TARG_NAME","A")%>`

ケース・スタディ：法的コンプライアンスのためにレコードを追跡する

一部のデータ・ウェアハウス・プロジェクトでは、法的コンプライアンスのために、ターゲット表に対する挿入または更新の各操作を追跡する必要があります。このような追跡により、業務分析者は、特定の期間にデータに何が発生したかを把握することができます。

この動作は、緩やかに変化するディメンション・ナレッジ・モジュールを使用して実現できる場合でも、フロー・データをターゲット表に適用する前に、単純にそのコピーを作成して実現することが可能です。

各ターゲット表に対応する表があり、同じ列および次のような追加の法的コンプライアンスの列が含まれるとします。

- ジョブ ID
- ジョブ名
- 操作の日時
- 操作の種類（挿入または更新）

また、ターゲットに挿入および更新を適用した後、IKM の終了直前に I$ 表から直接この表を移入するとします。たとえば、増分更新 IKM の場合の手順は次のようになります。

- ステージング領域で I$ フロー表を削除し、作成します。
- フロー・データを I$ 表に挿入します。
- 拒否された前のレコードをリサイクルします。
- データ品質チェックのために CKM をコールします。
- I$ 表を更新して IND_UPDATE 列を U または N に設定します。
- I$ 表の U フラグ付きレコードを使用してターゲットを更新します。
- I$ 表の I フラグ付きレコードをターゲットに挿入します。
- **法的コンプライアンスのために I$ 表をバックアップします。**
- I$ 一時表を削除します。

法的コンプライアンス表の名前がターゲット表の名前の後に _RGC を追加したものと同じと仮定すると、この手順のコードは次のように表されます。

手順	コード例

法的コンプライアンスのために I$ 表をバックアップします。	`insert  into <%=odiRef.getTable("L","TARG_NAME","A")%>_RGC`
	`(`
	`JOBID,`
	`JOBNAME,`
	`OPERATIONDATE,`
	`OPERATIONTYPE,`
	`<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "")%>`
	`)`
	`select <%=odiRef.getSession("SESS_NO")%> /* JOBID */,`
	`<%=odiRef.getSession("SESS_NAME")%> /* JOBNAME */,`
	`Current_timestamp /* OPERATIONDATE */,`
	`Case when IND_UPDATE = 'I' then 'Insert' else 'Update' end`
	`<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "")%>`
	`from   <%=odiRef.getTable("L","INT_NAME","A")%>`
	`where  IND_UPDATE <> 'N'`

この例は、低いコストの実装で既存のナレッジ・モジュールを柔軟に適応させ、複雑な要件でも簡単に満たせることを示しています。

ターゲットと異なるステージング領域での IKM

ファイルからサーバーへの追加

ソースが単一のファイルで構成されており、そのファイルを最も効率的な方法でターゲット表に直接ロードする場合があります。Oracle Data Integrator のデフォルト設定では、ステージング領域をターゲット・サーバー上に置き、このようなジョブを LKM（ファイルを C$ 表にステージングするため）および IKM（C$ 表のソース・データをターゲット表に適用するため）を使用して実行することが推奨されます。当然、ソース・データが変換されていない場合は、ターゲットにロードする前に C$ ステージング表にファイルをロードする必要はありません。

この問題の対処法の 1 つは、ターゲットに直接ファイルデータをロードできる IKM を使用することです。そのためには、ステージング領域をソース・ファイルの論理スキーマに設定する必要があります。この設定を行うと、Oracle Data Integrator によって自動的に複数接続 IKM の使用が推奨されます。この IKM を使用すると、リモート・ステージング領域とターゲット間でデータを移動できます。

ローダーを使用した、ファイルからターゲット表への IKM には、次のような手順が含まれます。

- 適切なロード・ユーティリティ・スクリプトを生成します。
- ロード・ユーティリティを実行します。

サーバーからサーバーへの追加

ターゲットと異なるステージング領域を使用し、このステージング領域を RDBMS に設定する場合は、ステージング領域からリモート・ターゲットに変換済データを移動する IKM を使用できます。このような IKM は LKM に非常に似ており、ほぼ同じ規則に従います。

一部の IKM では、エージェントを使用し、配列を使用してステージング領域からデータを取得し、バッチ更新を使用してターゲットに書き込みます。その他の IKM では、ステージング領域からファイルまたは FIFO にアンロードし、バルク・ロード・ユーティリティを使用してターゲットをロードします。

エージェントを使用する場合の手順は、一般的に単純です。

- オプションの値に依存するターゲット・データを削除します。
- ステージング領域からターゲットへデータを挿入します。この手順には、ステージング領域で実行される「Command on Source」タブの SELECT 文が含まれます。「Command on Target」タブでバインド変数を使用して INSERT 文が記述され、ターゲット表でバッチごとに実行されます。

アンロード / ロード戦略を使用する場合の手順は、通常、選択する IKM の種類に依存します。ただし、ほとんどの IKM で次の一般手順が行われます。

- OdiSqlUnload を使用して、ステージング領域からファイルまたは FIFO パイプラインへデータをアンロードします。
- ロード・ユーティリティ・スクリプトを生成します。
- ロード・ユーティリティをコールします。

サーバーからファイルまたは JMS への追加

ターゲット・データストアがファイルまたは JMS キューまたはトピックの場合、ステージング領域をターゲット以外の場所に設定する必要があります。したがって、ファイルをターゲットにする場合やデータストアをキューする場合は、複数接続 IKM を使用する必要があります。この IKM を使用すると、ステージング領域からこのターゲットへ変換済データがエクスポートされます。ファイルまたはキューへのデータのエクスポート方法は、IKM によって異なります。たとえば、エージェントを使用してステージング領域のレコードを選択させ、Oracle Data Integrator の標準機能を使用してファイルまたはキューに書き込むこともできます。また、ターゲットが JMS に基づいていない場合は、Teradata FastExport などの特定のアンロード・ユーティリティを使用することも可能です。

このような IKM の一般的な手順は、次のようになります。

- オプションに依存するターゲット・ファイルまたはキューをリセットします。
- ステージング領域からファイルまたはキューへデータをアンロードします。

7 独自のナレッジ・モジュールを開発するためのガイドライン

独自の KM を開発する場合の主要なガイドラインの 1 つは、決してゼロから始めないことです。Oracle Data Integrator には、すぐに使用できる 100 以上の KM が同梱されています。そのため、独自のテクノロジ用に記述されていなくても、まずはこれらの既存の KM を確認することをお薦めします。サンプルが多いほど、独自のコードの開発時間は短縮されます。たとえば、既存の KM を複製してテクノロジを変更したり、コードの行を別のコードからコピーすることから拡張を開始できます。

独自の KM を開発する際には、その KM が統合プロセスの特定の段階を対象とすることに留意してください。その他の留意事項は次のとおりです。

- LKM は、リモート・ソース・データセットをステージング領域（C$ 表）にロードするように設計されています。
- IKM は、ステージング領域のソース・フローをターゲットに適用します。まず、C$ 表を変換および結合して単一の I$ 表を作成し、CKM をコールしてこの I$ 表のデータ品質チェックを実行して、最後にフロー・データをターゲットに書き込みます。
- CKM は制約として表されるデータ品質規則と照合して、データストアまたはフロー表（I$）でデータ品質をチェックします。拒否されたレコードはエラー表（E$）に格納されます。
- RKM は、SNP_REV_xx 一時表を使用して、メタデータ・プロバイダから Oracle Data Integrator のリポジトリへメタデータを抽出します。
- JKM は、チェンジ・データ・キャプチャ・インフラストラクチャを作成します。

次に示す一般的な過ちに注意してください。

- KM の多用 : 通常のプロジェクトで必要な KM の数は 5 未満です。
- KM でのハードコードされた値（カタログまたはスキーマ名を含む）の使用 : かわりに getTable()、getTargetTable()、getObjectName() などの適切な代替メソッドを使用してください。
- KM での変数の使用 : かわりに、オプションまたはフレックス・フィールドを使用して設計者から情報を収集してください。
- 完全に Jython または Java で記述された KM: それ以外の方法がない場合に使用してください。一般的には SQL の方が読取りおよびメンテナンスを簡単に行えます。
- チェック・ボックス・オプションのかわりに <%if%> 文を使用したコード生成の条件付け。

KM に適用されるその他の一般的なコード記述の推奨事項 :

- コードは正しくインデントしてください。
- 生成されるコードも、読み取れるようにインデントしてください。
- select や insert などの SQL キーワードは、読み取りやすくするために小文字で記述してください。