(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第5064483号
(P5064483)
(45)発行日　平成24年10月31日(2012.10.31)　(24)登録日　平成24年8月17日(2012.8.17)

(51)Int.Cl.　F I
G06F 12/00 (2006.01)　G06F 12/00 513A

請求項の数 10 (全 61 頁)

(21)出願番号	特願2009-502890 (P2009-502890)
(86)(22)出願日	平成19年3月22日 (2007.3.22)
(65)公表番号	特表2009-544064 (P2009-544064A)
(43)公表日	平成21年12月10日 (2009.12.10)
(86)国際出願番号	PCT/US2007/007261
(87)国際公開番号	WO2007/112009
(87)国際公開日	平成19年10月4日 (2007.10.4)
審査請求日	平成22年3月23日 (2010.3.23)
(31)優先権主張番号	60/785,672
(32)優先日	平成18年3月23日 (2006.3.23)
(33)優先権主張国	米国(US)
(31)優先権主張番号	11/725,206
(32)優先日	平成19年3月16日 (2007.3.16)
(33)優先権主張国	米国(US)

(73)特許権者　500046438
マイクロソフト　コーポレーション
アメリカ合衆国　ワシントン州　９８０５２－６３９９　レッドモンド　ワン　マイクロソフト　ウェイ
(74)代理人　100077481
弁理士　谷　義一
(74)代理人　100088915
弁理士　阿部　和夫
(72)発明者　アトゥール　アドヤ
アメリカ合衆国　９８０５２　ワシントン州　レッドモンド　ワン　マイクロソフト　ウェイ　マイクロソフト　コーポレーション　インターナショナル　パテンツ内

最終頁に続く

(54)【発明の名称】増分ビューの保守を用いたマッピングアーキテクチャ

(57)【特許請求の範囲】
【請求項１】
データサービスをアプリケーションに提供する方法であって、コンピュータが、
明確に定義された意味を有する宣言的言語を使用して、エンティティデータモデルを用いるエンティティスキーマと、データベースに関連付けられたリレーショナルデータベーススキーマとの間の双方向マッピングを提供するステップであって、前記エンティティデータモデルは、前記アプリケーションによって使用されるデータを概念レベルのエンティティとして定義し、前記マッピングは、前記エンティティスキーマと前記リレーショナルデータベーススキーマとの相互関係を記述する、ステップと、
前記エンティティスキーマ、前記リレーショナルデータベーススキーマ、および前記マッピングをコンパイルして、クエリビューおよび更新ビューを生成するステップであって、クエリビューは、前記データベースのテーブルとの関係で前記エンティティをクエリとして表し、更新ビューは、前記クエリビューのエンティティとの関係で前記データベースのテーブルを表す、ステップと、
データへのアクセスを要求する前記アプリケーションからのクエリ要求であって、前記エンティティスキーマを対象とするクエリ要求に応答して、前記クエリビューをアンフォールディングして前記データベースを対象とするクエリに変換し、前記データベースにクエリを行うことによって、要求側のアプリケーションの代わりに前記クエリ要求を処理するステップと、
データの更新を要求する前記アプリケーションからの更新要求に応答して、前記更新ビ

ューにビュー保守アルゴリズムを適用して前記データベースを更新することによって、要求側のアプリケーションの代わりに前記更新要求を処理するステップと
を含むことを特徴とする方法。
【請求項２】
前記要求側のアプリケーションから、前記データベースを更新する際に使用するためのデータを備えたプログラミング言語のオブジェクトを受信することをさらに含むことを特徴とする請求項１に記載の方法。
【請求項３】
前記要求側のアプリケーションから、前記データベースを更新する際に使用するためのデータを備えた作成命令、挿入命令、更新命令、または削除命令を受信することをさらに含むことを特徴とする請求項１に記載の方法。
【請求項４】
前記要求側のアプリケーションから、前記データベースを更新する際に使用するためのデータを備えたデータ操作言語（ＤＭＬ）の式を受信することをさらに含むことを特徴とする請求項１に記載の方法。
【請求項５】
前記更新ビューにビュー保守アルゴリズムを適用することは、前記更新ビューに関するデルタ式を作成することを含み、
前記デルタ式をクエリビューと組み合わせるためにビューアンフォールディングを使用することをさらに含むことを特徴とする請求項１に記載の方法。
【請求項６】
前記エンティティスキーマは、クラス、リレーションシップ、継承、集約、および複合型をサポートすることを特徴とする請求項１に記載の方法。
【請求項７】
データサービスをアプリケーションに提供するデータアクセスシステムを実装するためにコンピュータによって実行可能なプログラムであって、前記コンピュータに、
明確に定義された意味を有する宣言的言語を使用して、エンティティデータモデルを用いるエンティティスキーマと、データベースに関連付けられたリレーショナルデータベーススキーマとの間の双方向マッピングを提供する動作であって、前記エンティティデータモデルは、前記アプリケーションによって使用されるデータを概念レベルのエンティティとして定義し、前記マッピングは、前記エンティティスキーマと前記リレーショナルデータベーススキーマとの相互関係を記述する、動作と
前記エンティティスキーマ、前記リレーショナルデータベーススキーマ、および前記マッピングをコンパイルして、クエリビューおよび更新ビューを生成する動作であって、クエリビューは、前記データベースのテーブルとの関係で前記エンティティをクエリとして表し、更新ビューは、前記クエリビューのエンティティとの関係で前記データベースのテーブルを表す、動作と、
データへのアクセスを要求する前記アプリケーションからのクエリ要求であって、前記エンティティスキーマを対象とするクエリ要求を、前記クエリビューをアンフォールディングして前記データベースを対象とするクエリに変換し、前記データベースにクエリを行うことによって、要求側のアプリケーションの代わりに処理する動作と、
データの更新を要求する前記アプリケーションからの更新要求を、前記更新ビューにビュー保守アルゴリズムを適用して前記データベースを更新することによって、要求側のアプリケーションの代わりに処理する動作と
を実行させることを特徴とするプログラム。
【請求項８】
前記コンピュータに、前記要求側のアプリケーションから、前記データベースを更新する際に使用するためのデータを備えたプログラミング言語のオブジェクトを受信する動作をさらに実行させることを特徴とする請求項７に記載のプログラム。
【請求項９】

前記コンピュータに、前記要求側のアプリケーションから、前記データベースを更新する際に使用するためのデータを備えた作成命令、挿入命令、更新命令、または削除命令を受信する動作をさらに実行させることを特徴とする請求項７に記載のプログラム。
【請求項１０】
前記コンピュータに、前記要求側のアプリケーションから、前記データベースを更新する際に使用されるデータを備えたデータ操作言語（ＤＭＬ）の式を受信する動作をさらに実行させることを特徴とする請求項７に記載のプログラム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、増分ビューの保守を用いたマッピングアークテクチャに関する。
【背景技術】
【０００２】
アプリケーションとデータベースとのブリッジングは、長年にわたる問題である。１９９６年に、ＣａｒｅｙおよびＤｅＷｉｔｔは、オブジェクト指向データベースおよび永続プログラミング言語を含む多数のテクノロジが、クエリおよび更新処理、トランザクションスループット、ならびにスケーラビリティにおける制限のために、広く受け入れられなかった理由の概要を示した。彼らは、２００６年にオブジェクト／リレーショナル（Ｏ／Ｒ）データベースが中心になるであろうと推測した。実際には、ＤＢ２（登録商標）データベースシステムおよびＯｒａｃｌｅ（登録商標）データベースシステムは、従来のリレーショナルエンジンの上にハードワイヤードのＯ／Ｒマッピングを使用する組み込みオブジェクトレイヤを含む。しかし、これらのシステムによって提供されるＯ／Ｒ機能は、マルチメディアデータ型および空間データ型を除いて、企業データを格納するのにはほとんど使用されないと思われる。その理由の中には、データおよびベンダの独立性、レガシデータベースを移行するコスト、ビジネスロジックが中間層ではなくデータベース内で動作する際のスケールアウトの困難性、ならびにプログラミング言語との統合が不十分であることがある。
【０００３】
１９９０年代半ばから、クライアント側のデータマッピングレイヤが、インターネットアプリケーションの成長によって活気付けられ、人気を獲得している。そのようなレイヤのコア機能は、明示的なマッピングによって駆動されるアプリケーションのデータモデルにしっかりと合ったデータモデルを公開する、更新可能ビューを提供することである。多数の製品およびオープンソースプロジェクトが現れ、これらの機能を提供している。事実上全てのエンタープライズフレームワークが、クライアント側の永続レイヤ（例えば、Ｊ２ＥＥのＥＪＢ）を提供する。ＥＲＰアプリケーションおよびＣＲＭアプリケーションなど、ほとんどのパッケージ化されたビジネスアプリケーションは、専用のデータアクセスインターフェース（例えば、ＳＡＰ　Ｒ／３のＢＡＰＩ）を組み込んでいる。
【０００４】
広く使用されているＪａｖａ（登録商標）用のオープンソースのＯＲＭ（Ｏｂｊｅｃｔ－Ｒｅｌａｔｉｏｎａｌ　Ｍａｐｐｉｎｇ）フレームワークの１つが、Ｈｉｂｅｒｎａｔｅ（登録商標）である。Ｈｉｂｅｒｎａｔｅは、複数の継承マッピングシナリオ、オプティミスティック並行性制御、および包括的オブジェクトサービスをサポートする。Ｈｉｂｅｒｎａｔｅの最新のリリースは、ＥＪＢ　３．０標準規格に準拠し、ＥＪＢ　３．０標準規格は、Ｊａｖａ（登録商標）　Ｐｅｒｓｉｓｔｅｎｃｅ　Ｑｕｅｒｙ　Ｌａｎｇｕａｇｅを含む。商業的側面において、一般的なＯＲＭは、Ｏｒａｃｌｅ　ＴｏｐＬｉｎｋ（登録商標）およびＬＬＢＬＧｅｎ（登録商標）を含む。後者は．ＮＥＴプラットフォームで動作する。これらおよび他のＯＲＭは、そのターゲットプログラミング言語のオブジェクトモデルに密結合される。
【０００５】
ＢＥＡ（登録商標）は、最近、ＡＬＤＳＰ（ＡｑｕａＬｏｇｉｃ　Ｄａｔａ　Ｓｅｒｖ

ｉｃｅｓ　Ｐｌａｔｆｏｒｍ（登録商標））と呼ばれる新しいミドルウェア製品を導入した。これは、アプリケーションデータのモデル化にＸＭＬスキーマを使用する。ＸＭＬデータは、ＸＱｕｅｒｙを使用してデータベースおよびウェブサービスからアセンブルされる。ＡＬＤＳＰのランタイムは、複数のデータソースにまたがるクエリをサポートし、クライアント側のクエリ最適化を実行する。更新は、ＸＱｕｅｒｙビューに対するビュー更新として実行される。更新が一意の変換を有しない場合は、開発者が、命令コードを使用して更新ロジックをオーバーライドする必要がある。ＡＬＤＳＰのプログラミングサーフェスは、ＳＤＯ（ｓｅｒｖｉｃｅ　ｄａｔａ　ｏｂｊｅｃｔ）に基づいている。

【０００６】

今日のクライアント側のマッピングレイヤは、様々な程度の機能、堅固性、および総所有コスト（ＴＣＯ：ｔｏｔａｌ　ｃｏｓｔ　ｏｆ　ｏｗｎｅｒｓｈｉｐ）を提供する。典型的に、ＯＲＭによって使用されるアプリケーションとデータベースのアーチファクトとの間のマッピングは、曖昧な意味を有し、ケースバイケースで推論することになる。シナリオ主導の実装は、サポートされるマッピングの範囲を制限し、しばしば、拡張が難しい脆弱なランタイムをもたらす。ごく少数のデータアクセスソリューションが、データベースコミュニティによって開発されたデータ変形技法を活用し、しばしば、クエリ変換および更新変換に関するアドホックな解決策に依拠する。

【発明の開示】

【発明が解決しようとする課題】

【０００７】

データベースの研究は、永続レイヤの構築に活用することができる多数の強力な技法に貢献してきた。それでも、著しいギャップがある。最も重大なギャップの中に、マッピングを介した更新のサポートがある。クエリと比較して、更新は、マッピングにまたがってデータの一貫性を保つ必要があり、ビジネスルールなどをトリガする可能性があるので、扱うのがはるかに難しい。その結果、商用データベースシステムおよびデータアクセス製品は、更新可能なビューに関する非常に制限されたサポートを提供する。最近、研究者は、双方向変形などの代替なアプローチに取り組んでいる。

【０００８】

伝統的に、概念モデル化は、データベースおよびアプリケーションの設計、リバースエンジニアリング、およびスキーマ変換に制限されてきた。多数の設計ツールが、ＵＭＬを使用する。ごく最近の概念モデル化のみが、産業強度（ｉｎｄｕｓｔｒｙ－ｓｔｒｅｎｇｔｈ）のデータマッピングソリューションに進出し始めた。例えば、エンティティおよびリレーションシップの概念は、ＡＬＤＳＰとＥＪＢ　３．０との両方で姿を表している。ＡＬＤＳＰは、Ｅ－Ｒスタイルのリレーションシップを複合型ＸＭＬデータの上にオーバーレイするが、ＥＪＢ　３．０は、クラス注釈を使用してオブジェクト間のリレーションシップを指定することを可能にする。

【０００９】

スキーママッピング技法は、Ｍｉｃｒｏｓｏｆｔ（登録商標）ＢｉｚＴａｌｋ　Ｓｅｒｖｅｒ（登録商標）、ＩＢＭ（登録商標）Ｒａｔｉｏｎａｌ　Ｄａｔａ　Ａｒｃｈｉｔｅｃｔ（登録商標）、およびＥＴＬ（登録商標）ツールなど、多数のデータ統合製品で使用される。これらの製品は、開発者が、データ変形を設計し、またはマッピングからデータ変形をコンパイルして、ｅ－コマースメッセージを変換するかデータウェアハウスをロードすることを可能にする。

【課題を解決するための手段】

【００１０】

データアクセスアーキテクチャを実装および使用するためのシステム、方法、およびコンピュータ読み取り可能な媒体を提供する。該データアクセスアーキテクチャは、アプリケーションによって使用され得るデータを、データベース内で永続するデータにマッピングする、マッピングアーキテクチャを含む。一実施形態では、該マッピングアーキテクチャは、２つタイプのマッピングビュー、すなわち、クエリの変換を助けるクエリビューと

、更新の変換を助ける更新ビューを使用する。増分ビューの保守（ｉｎｃｒｅｍｅｎｔａｌ　ｖｉｅｗ　ｍａｉｎｔｅｎａｎｃｅ）を使用して、アプリケーションとデータベースとの間でデータを変換することができる。さらなる態様および実施形態を、以下で説明する。

【００１１】

本発明にかかる増分ビューの保守を用いたマッピングアーキテクチャのシステムおよび方法を、添付の図面を参照してさらに説明する。

【発明を実施するための最良の形態】

【００１２】

新規のデータアクセスアーキテクチャ

一実施形態では、本革新を、このセクションで説明する新規のデータアクセスアーキテクチャ、「Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ（エンティティフレームワーク）」において実装し、その諸態様を組み込むことができる。そのようなＥｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの例が、本件特許出願人によって開発されたＡＤＯ．ＮＥＴ　ｖＮＥＸＴ（登録商標）データアクセスアーキテクチャである。以下に、多くの実装固有の詳細とともに、ＡＤＯ．ＮＥＴ　ｖＮＥＸＴデータアクセスアーキテクチャを全般的に説明するが、この詳細は本発明の実践に必要とみなされるべきではない。

【００１３】

＜概要＞

従来のクライアント－サーバアプリケーションは、そのデータに対するクエリおよび永続性のオペレーションをデータベースシステムに委ねる。データベースシステムは、行およびテーブルの形式のデータを操作するが、アプリケーションは、より高水準のプログラミング言語構造（クラス、構造体など）に関してデータを操作する。アプリケーション層とデータベース層との間のデータ操作サービスにおけるインピーダンス不整合は、従来のシステムにおいても問題であった。サービス指向アーキテクチャ（ＳＯＡ：ｓｅｒｖｉｃｅ－ｏｒｉｅｎｔｅｄ　ａｒｃｈｉｔｅｃｔｕｒｅ）、アプリケーションサーバ、および多層（ｍｕｌｔｉ－ｔｉｅｒ）アプリケーションの出現とともに、プログラミング環境とよく統合され、任意の層において動作可能な、データアクセスサービスおよびデータ操作サービスの必要性が非常に高まっている。

【００１４】

本件特許出願人のＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、抽象化のレベルをリレーショナルレベルから概念（エンティティ）レベルまで高める、データをプログラミングするためのプラットフォームであり、これによって、アプリケーションサービスとデータ中心のサービスとのインピーダンス不整合が著しく減少される。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ、全体的なシステムアーキテクチャ、および基礎になる技術についての諸態様を以下で説明する。

【００１５】

＜序論＞

現代のアプリケーションは、全ての層でデータ管理サービスを必要とする。現代のアプリケーションは、ますます豊富な形式のデータを処理する必要があり、このデータは、構造化されたビジネスデータ（Ｃｕｓｔｏｍｅｒｓ（顧客）とＯｒｄｅｒｓ（注文）など）だけではなく、電子メール、カレンダ、ファイル、および文書などの、半構造化されたコンテンツおよび構造化されていないコンテンツも含む。これらのアプリケーションは、より機敏な意思決定プロセスを可能にするために、複数のデータソースからのデータを統合すると同時に、このデータを収集し、浄化し、変形し、格納する必要がある。これらのアプリケーションの開発者は、その生産性を高めるために、データアクセスツール、プログラミングツール、および開発ツールを必要とする。リレーショナルデータベースは、ほとんどの構造化データのデファクトストアになっているが、そのようなデータベースが公開するデータモデル（および機能）と、アプリケーションが必要とするモデル化機能との間の不整合、すなわち周知のインピーダンス不整合の問題がある傾向がある。

【００１６】
他の２つの要因も、企業システムの設計において重要な部分を果たす。第１に、アプリケーションのデータ表現は、基礎になるデータベースのデータ表現と異なって進化する傾向がある。第２に、多くのシステムが、異なる程度の機能を有する別個のデータベースバックエンドから構成される。中間層のアプリケーションロジックは、これらの差を調整してデータのより均一なビューを提示する、データ変形に関与する。これらのデータ変形は、すぐに複雑になる。それらを実装すること、特に基礎になるデータが更新可能である必要があるときの実装は、難しい問題であり、アプリケーションに複雑性を加える。アプリケーション開発のかなりの部分、一部の事例では４０％までが、これらの問題に対処するためのカスタムデータのアクセスロジックの記述に使われる。

【００１７】
同一の問題が、データ中心サービスについて存在するが、深刻さはより少ない。クエリ、更新、およびトランザクションなどの従来のサービスは、論理スキーマ（リレーショナル）レベルで実装されてきた。しかし、複製および分析など、より新しいサービスの大多数は、典型的により高水準の概念データモデルに関連付けられたアーチファクトに対して最もよく動作する。例えば、ＳＱＬ　ＳＥＲＶＥＲ（登録商標）Ｒｅｐｌｉｃａｔｉｏｎは、制限された形式のエンティティを表現する「論理レコード」と呼ばれる構造を発明した。同様に、ＳＱＬ　Ｓｅｒｖｅｒ　Ｒｅｐｏｒｔｉｎｇ　Ｓｅｒｖｉｃｅｓは、ＳＤＭＬ（ｓｅｍａｎｔｉｃ　ｄａｔａ　ｍｏｄｅｌ　ｌａｎｇｕａｇｅ）と呼ばれるエンティティ同様のデータモデルの上にレポートを作成する。これらのサービスのそれぞれは、概念エンティティを定義してこれらをリレーションテーブルにマッピングする、カスタムツールを有する。したがって、Ｃｕｓｔｏｍｅｒエンティティを、ある方法で複製用に、別の方法でレポート構築用に、さらに別の方法で他の分析サービス用になど、定義してマッピングする必要がある。アプリケーションに関して、各サービスは典型的に、最終的にはこの問題に対するカスタムソリューションを作成することになり、その結果、サービス間には、コードの重複と制限されたインターオペラビリティとが存在する。

【００１８】
ＨＩＢＥＲＮＡＴＥ（登録商標）およびＯＲＡＣＬＥ　ＴＯＰＬＩＮＫ（登録商標）などのＯＲＭ（Ｏｂｊｅｃｔ－ｔｏ－ｒｅｌａｔｉｏｎａｌ　ｍａｐｐｉｎｇ）技術は、カスタムデータアクセスロジックの良く知られた代替である。データベースとアプリケーションとの間のマッピングは、カスタム構造内でまたはスキーマ注釈を介して表現される。これらのカスタム構造は、概念モデルと同様に見えることがあるが、アプリケーションは、この概念モデルに対して直接プログラミングすることができない。マッピングは、データベースとアプリケーションとの間に、ある程度の独立性を提供するが、同一データのわずかに異なるビューを有する複数のアプリケーションを扱うという問題（例えば、Ｃｕｓｔｏｍｅｒエンティティの異なる射影を見ようとする２つのアプリケーションを考慮されたい）、または、より動的な傾向があるサービスの必要性という問題（先験的クラス（ｐｒｉｏｒｉ　ｃｌａｓｓ）生成技法は、基礎になるデータベースがより早く進歩する可能性があるので、データサービスについてはうまく働かない）は、これらの解決策によってうまく対処されていない。

【００１９】
ＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、アプリケーションおよびデータ中心サービスのインピーダンス不整合を著しく減少させる、データに対するプログラミングのプラットフォームである。ＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、少なくとも次の観点で他のシステムおよびソリューションと異なる。

【００２０】
１．Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、豊富な概念データモデル（Ｅｎｔｉｔｙ　Ｄａｔａ　Ｍｏｄｅｌ、すなわちＥＤＭ）、および、このモデルのインスタンスを操作する新しいデータ操作言語（Ｅｎｔｉｔｙ　ＳＱＬ）を定義する。ＳＱＬと同様に、ＥＤＭは値ベースである。すなわち、ＥＤＭは、エンティティの構造的な諸態様を定義するので

あって、振舞い（またはメソッド）は定義しない。

【００２１】

２．このモデルは、クエリおよび更新の強力な双方向（ＥＤＭ－リレーショナル）マッピングをサポートするミドルウェアマッピングエンジンを含む、ランタイムによって具象化される。

【００２２】

３．アプリケーションおよびサービスは、値ベースの概念レイヤに対して、または概念（エンティティ）抽象化にまたがって階層化できるプログラミング言語固有のオブジェクト抽象化に対して、直接プログラミングすることができ、ＯＲＭ同様の機能性を提供する。値ベースのＥＤＭ概念の抽象化は、アプリケーションとデータ中心サービスとの間でデータを共有するための、オブジェクトよりもより柔軟性のある基礎であると考える。

【００２３】

４．最後に、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋは、本件特許出願人の新しいＬＩＮＱ（Ｌａｎｇｕａｇｅ Ｉｎｔｅｇｒａｔｅｄ Ｑｕｅｒｙ）技術を活用する。該ＬＩＮＱ技術は、クエリ表現をネイティブに用いてプログラミング言語を拡張し、アプリケーションに関するインピーダンス不整合をさらに減少させ、一部のシナリオでは完全に除去する。

【００２４】

ＡＤＯ．ＮＥＴ Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋは、Ｍｉｃｒｏｓｏｆｔ ．ＮＥＴ Ｆｒａｍｅｗｏｒｋなどの、より大きいフレームワークに組み込むことができる。

【００２５】

データアクセスアーキテクチャに関するこの説明の残りは、ＡＤＯ．ＮＥＴ Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋの実施形態の文脈で、次のように編成される。「動機づけ」セクションは、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋの追加の動機づけを提供する。「Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋ」セクションは、Ｅｎｔｉｔｙ ＦｒａｍｅｗｏｒｋおよびＥｎｔｉｔｙ Ｄａｔａ Ｍｏｄｅｌを提示する。「プログラミングパターン」セクションは、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋのプログラミングパターンを説明する。「Ｏｂｊｅｃｔ Ｓｅｒｖｉｃｅｓ」セクションは、Ｏｂｊｅｃｔ Ｓｅｒｖｉｃｅｓモジュールの概要を示す。「マッピング」セクションは、Ｅｎｔｉｔｙ ＦｒａｍｅｗｏｒｋのＭａｐｐｉｎｇコンポーネントに焦点を当て、「クエリ処理」セクションおよび「更新処理」セクションは、クエリおよび更新がどのように処理されるかを説明する。「メタデータ」および「ツール」は、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋのメタデータサブシステムおよびツールコンポーネントを説明する。

【００２６】

＜動機づけ＞

このセクションは、より高水準のデータモデル化レイヤが、なぜアプリケーションおよびデータ中心サービスに必須になったのかを論じる。

【００２７】

（データアプリケーションにおける情報レベル）

データベース設計を作成するための今日の支配的な情報モデル化方法は、情報モデルを４つの主要なレベル、すなわち、物理、論理（リレーショナル）、概念、およびプログラミング／プレゼンテーションにファクタリングする。

【００２８】

物理モデルは、データが、メモリ、ワイヤ、またはディスクなどの物理的なリソース内でどのように表現されるかを記述する。このレイヤで検討される概念のボキャブラリは、レコードフォーマット、ファイルの区画およびグループ、ヒープ、ならびにインデックスを含む。物理モデルは、典型的に、アプリケーションに対して可視ではなく、物理モデルに対する変更は、アプリケーションロジックに影響を与えてはいけないが、アプリケーションのパフォーマンスに影響を与えることがある。

【００２９】

論理データモデルは、ターゲットドメインの完全で正確な情報モデルである。リレーショナルモデルは、ほとんどの論理データモデルに関する選択の表現である。論理レベルで検討される概念は、テーブル、行、主キー／外部キー制約、および正規化を含む。正規化は、データの一貫性、さらなる並行性、およびより良いＯＬＴＰパフォーマンスを達成するのを助けるが、アプリケーションに関する重大な課題をも導入する。論理レベルで正規化されたデータは、しばしばフラグメント化されすぎており、アプリケーションロジックは、複数のテーブルからの行を、アプリケーションドメインのアーチファクトにさらに類似しているより高水準のエンティティに、アセンブルする必要がある。

【００３０】

概念モデルは、問題ドメインからのコア情報エンティティおよびそのリレーションシップをキャプチャする。周知の概念モデルは、１９７６年にＰｅｔｅｒ　Ｃｈｅｎによって紹介されたエンティティ・リレーションシップモデル（Ｅｎｔｉｔｙ－Ｒｅｌａｔｉｏｎｓｈｉｐ　Ｍｏｄｅｌ）である。ＵＭＬは、概念モデルのより最近の例である。ほとんどのアプリケーションは、概念設計のフェーズをアプリケーション開発のライフサイクルの早期に含む。しかし、残念ながら、概念データモデル図は、「壁にピンで止められた」状態が続き、時とともにアプリケーション実装の現実からますます離れつつある。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの重要な目標は、概念データモデル（次のセクションで説明するＥｎｔｉｔｙ　Ｄａｔａ　Ｍｏｄｅｌによって具現化される）を、データプラットフォームの具象的なプログラム可能な抽象化にすることである。

【００３１】

プログラミング／プレゼンテーションモデルは、概念モデルのエンティティおよびリレーションシップを、手元のタスクに基づいて異なる形式でどのようにマニフェスト（提示）する必要があるかを記述する。エンティティの中には、アプリケーションビジネスロジックを実装するために、プログラミング言語オブジェクトに変形される必要があるもの、ウェブサービスの呼出しのためにＸＭＬストリームに変形される必要があるもの、ユーザインターフェースのデータバインディングのために、リストまたはディクショナリなどのメモリ内の構造に変形される必要があるものがある。当然、普遍的なプログラミングモデルまたはプレゼンテーションの形式は存在せず、したがって、アプリケーションは、エンティティを様々なプレゼンテーションの形式に変形する柔軟なメカニズムを必要とする。

【００３２】

ほとんどのアプリケーションおよびデータ中心サービスは、Ｏｒｄｅｒをリレーショナルデータベーススキーマにおいて正規化できる複数のテーブルについてではなく、Ｏｒｄｅｒなどの高水準の概念に関して推論する傾向がある。注文は、プレゼンテーション／プログラミングレベルにおいて、注文に関連付けられた状態およびロジックをカプセル化するＶｉｓｕａｌ　ＢａｓｉｃまたはＣ＃内のクラスインスタンスとして、またはウェブサービスと通信するＸＭＬストリームとして現れることがある。正しいプレゼンテーションモデルは存在しないが、具象的な概念モデルを提供すること、および、そのモデルを、様々なプレゼンテーションモデルおよび他の高水準データサービスとの間での柔軟性のあるマッピングの基礎として使用できることには、価値がある。

【００３３】

（アプリケーションおよびサービスの進化）

１０～２０年前の、データに基づくアプリケーションは、典型的に、データモノリス、すなわち、論理スキーマレベルでデータベースシステムと対話する動詞オブジェクト関数（例えば、ｃｒｅａｔｅ－ｏｒｄｅｒ、ｕｐｄａｔｅ－ｃｕｓｔｏｍｅｒ）によってファクタリングされたロジックを有する閉じたシステムとして構造化されていた。いくつかの顕著な傾向が、今日、近代のデータに基づくアプリケーションをファクタリングし、展開する方法を形成している。これらの中の主要なものは、オブジェクト指向ファクタリング、サービスレベルアプリケーション構造、およびより高水準のデータ中心サービスである。概念エンティティは、今日のアプリケーションの重要な部分である。これらのエンティティは、様々な表現にマッピングされ、様々なサービスにバインドされなければならない

。正しい表現またはサービスバインディングは存在しない。すなわち、ＸＭＬ表現、リレーショナル表現、およびオブジェクト表現の全てが重要であるが、これらの１つが、全てのアプリケーションにとって十分であることはない。したがって、より高水準のデータモデル化レイヤをサポートし、複数のプレゼンテーションレイヤをプラグインすることをも可能にするフレームワークの必要性が存在する。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、これらの要件を満たすことを目的とする。

【００３４】

データ中心サービスも、同様の方法で進化してきている。２０年前に「データプラットフォーム」によって提供されたサービスは、最小限であり、ＲＤＢＭＳの論理スキーマの周囲に焦点を合わせていた。これらのサービスには、クエリおよび更新、アトミックトランザクション、ならびに、バックアップおよびロード／抽出などのバルクオペレーションが含まれていた。

【００３５】

ＳＱＬ　Ｓｅｒｖｅｒ自体は、従来的なＲＤＢＭＳから、概念スキーマレベルで実現されるエンティティにまたがる価値の高い複数のデータ中心サービスを提供する、完全なデータプラットフォームへと進化しつつある。ＳＱＬ　Ｓｅｒｖｅｒ製品内の複数のより高水準のデータ中心サービス（２つだけを例に挙げると、ＲｅｐｌｉｃａｔｉｏｎおよびＲｅｐｏｒｔ　Ｂｕｉｌｄｅｒ）は、そのサービスを概念スキーマレベルで、ますます提供している。現在、これらのサービスのそれぞれは、別々のツールを有し、概念エンティティを記述し、これらを基礎になる論理スキーマレベルにマッピングする。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの目標は、これらのサービスの全てが共有できる、共通のより高水準の概念抽象化を提供することである。

【００３６】

＜Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ＞

本明細書で説明するＥｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの前に存在した本件特許出願人のＡＤＯ．ＮＥＴフレームワークは、データアクセス技術であり、該技術は、アプリケーションがデータストアに接続し、該データストア内に含まれるデータを様々な方法で操作することを可能にした。これは、Ｍｉｃｒｏｓｏｆｔ　．ＮＥＴ　Ｆｒａｍｅｗｏｒｋの一部であり、．ＮＥＴ　Ｆｒａｍｅｗｏｒｋクラスライブラリの残りに高度に統合された。以前のＡＤＯ．ＮＥＴフレームワークは、２つの主要な部分、すなわち、プロバイダとサービスとを有していた。ＡＤＯ．ＮＥＴプロバイダは、特定のデータストアとの話し方を知っているコンポーネントである。プロバイダは、機能の３つのコア部分から構成される。すなわち、コネクションが、基礎になるデータソースに対するアクセスを管理し、コマンドが、データソースに対して実行されるコマンド（クエリ、プロシージャ呼出しなど）を表し、データリーダーが、コマンド実行の結果を表す。ＡＤＯ．ＮＥＴサービスは、オフラインのデータプログラミングシナリオを可能にするＤａｔａＳｅｔなど、プロバイダ中立コンポーネントを含む（ＤａｔａＳｅｔは、データソースに関わらず一貫したリレーショナルプログラミングモデルを提供する、データのメモリ常駐表現である）。

【００３７】

（Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの概要）

ＡＤＯ　．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、予め存在する既存のＡＤＯ．ＮＥＴプロバイダモデルを基に、次の機能性を追加する。

１．概念スキーマのモデル化を助ける新しい概念データモデルである、Ｅｎｔｉｔｙ　Ｄａｔａ　Ｍｏｄｅｌ（ＥＤＭ）。

２．ＥＤＭのインスタンスおよびクエリのプログラム的表現（カノニカルコマンドツリー（ｃａｎｏｎｉｃａｌ　ｃｏｍｍａｎｄ　ｔｒｅｅ））を操作して、異なるプロバイダと通信するための新しいデータ操作言語（ＤＭＬ：ｄａｔａ　ｍａｎｉｐｕｌａｔｉｏｎ　ｌａｎｇｕａｇｅ）である、Ｅｎｔｉｔｙ　ＳＱＬ。

３．概念スキーマと論理スキーマとの間のマッピングを定義する能力。

４．概念スキーマに対するＡＤＯ．ＮＥＴプロバイダのプログラミングモデル。

５．ＯＲＭ同様の機能を提供するオブジェクトサービスレイヤ。
６．データを．ＮＥＴ言語からのオブジェクトとして、プログラミングすることを容易にするＬＩＮＱ技術との統合。

【００３８】

（Ｅｎｔｉｔｙ　Ｄａｔａ　Ｍｏｄｅｌ）

エンティティデータモデル（ＥＤＭ：Ｅｎｔｉｔｙ　Ｄａｔａ　Ｍｏｄｅｌ）は、豊富なデータ中心アプリケーションの開発を可能にする。ＥＤＭは、Ｅ－Ｒドメインからの概念を用いて典型的なリレーショナルモデルを拡張する。本明細書で提供する例示的な実施形態では、ＥＤＭの組織的概念は、エンティティとリレーションシップとを含む。エンティティは、トップレベルのアイテムを識別（ｉｄｅｎｔｉｔｙ）で表し、リレーションシップは、２つまたはそれ以上のエンティティを関係付ける（または、２つまたはそれ以上のエンティティの間のリレーションシップを記述する）のに使用される。

【００３９】

一実施形態では、ＥＤＭは、Ｃ＃（ＣＬＲ）のようにオブジェクト／参照ベースではなく、リレーショナルモデル（およびＳＱＬ）のように値ベースである。いくつかのオブジェクトプログラミングモデルを、ＥＤＭの上に容易に階層化することができる。同様に、ＥＤＭは、永続性に関して１つまたは複数のＤＢＭＳ実装にマッピングすることができる。

【００４０】

ＥＤＭおよびＥｎｔｉｔｙ　ＳＱＬは、データプラットフォーム用のより豊富なデータモデルおよびデータ操作言語を表し、ＣＲＭおよびＥＲＰなどのアプリケーション、レポート作成、ビジネスインテリジェンス、複製、および同期化などのデータ集中型サービス、ならびにデータ集中型アプリケーションが、その必要性により近い構造および意味のレベルでデータをモデル化して操作することを可能にすることが意図されている。次に、ＥＤＭに関係する様々な概念を論じる。

【００４１】

・ＥＤＭの型

ＥｎｔｉｔｙＴｙｐｅは、エンティティの構造を記述する。エンティティは、そのエンティティの構造を記述するゼロまたはそれ以上のプロパティ（属性、フィールド）を有することができる。さらに、エンティティ型は、キー、すなわち、エンティティのコレクション内でエンティティインスタンスを一意に識別する値を有するプロパティのセットを、定義しなければならない。ＥｎｔｉｔｙＴｙｐｅは、別のエンティティ型から派生する（またはサブタイプ化する）ことができ、ＥＤＭは、単一の継承モデルをサポートする。あるエンティティのプロパティは、単純型または複合型とすることができる。ＳｉｍｐｌｅＴｙｐｅは、スカラ（またはアトミック）型（例えば、整数、ストリング）を表し、ＣｏｍｐｌｅｘＴｙｐｅは、構造化されたプロパティ（例えば、Ａｄｄｒｅｓｓ）を表す。ＣｏｍｐｌｅｘＴｙｐｅは、ゼロまたはそれ以上のプロパティから構成され、これらのプロパティ自体は、スカラ型または複合型のプロパティとすることができる。ＲｅｌａｔｉｏｎｓｈｉｐＴｙｐｅは、２つ（またはそれ以上）のエンティティ型の間のリレーションシップを記述する。ＥＤＭ　Ｓｃｈｅｍａは、型のグループ化メカニズムを提供し、型は、スキーマで定義されなければならない。型名と組み合わされたスキーマの名前空間は、特定の型を一意に識別する。

【００４２】

・ＥＤＭインスタンスモデル

エンティティインスタンス（または単にエンティティ）は、論理的にＥｎｔｉｔｙＳｅｔ内に含まれる。ＥｎｔｉｔｙＳｅｔは、エンティティの同種コレクションである。すなわち、あるＥｎｔｉｔｙＳｅｔ内の全てのエンティティは、同一の（または派生した）ＥｎｔｉｔｙＴｙｐｅを有しなければならない。ＥｎｔｉｔｙＳｅｔは、概念的にはデータベースのテーブルに類似し、エンティティは、テーブルの行に類似する。エンティティインスタンスは、正確に１つのエンティティセットに属さなければならない。同様に、リレ

ーションシップのインスタンスは、論理的にＲｅｌａｔｉｏｎｓｈｉｐＳｅｔ内に含まれる。ＲｅｌａｔｉｏｎｓｈｉｐＳｅｔの定義が、リレーションシップのスコープを決定する。すなわち、これは、リレーションシップに参加するエンティティ型のインスタンスを保持するＥｎｔｉｔｙＳｅｔを識別する。ＲｅｌａｔｉｏｎｓｈｉｐＳｅｔは、概念的にはデータベース内のリンクテーブルに類似する。ＳｉｍｐｌｅＴｙｐｅおよびＣｏｍｐｌｅｘＴｙｐｅを、単に、ＥｎｔｉｔｙＴｙｐｅのプロパティとしてインスタンス化することができる。ＥｎｔｉｔｙＣｏｎｔａｉｎｅｒは、ＥｎｔｉｔｙＳｅｔおよびＲｅｌａｔｉｏｎｓｈｉｐＳｅｔの論理グループ化であり、ＳｃｈｅｍａがＥＤＭ型に関してどのようなグループ化メカニズムであるかに類似する。

【００４３】

・例示的なＥＤＭ　Ｓｃｈｅｍａ

サンプルのＥＤＭスキーマを以下に示す。

【００４４】

【表１】

```
<?xml version="1.0" encoding="utf-8"?>
<Schema Namespace="AdventureWorks" Alias="Self" …>
  <EntityContainer Name="AdventureWorksContainer">
    <EntitySet Name="ESalesOrders"
               EntityType="Self.ESalesOrder" />
    <EntitySet Name="ESalesPersons"
               EntityType="Self.ESalesPerson" />
    <AssociationSet Name="ESalesPersonOrders"
          Association="Self.ESalesPersonOrder">
      <End Role="ESalesPerson"
           EntitySet="ESalesPersons" />
      <End Role="EOrder" EntitySet="ESalesOrders" />
    </AssociationSet>
  </EntityContainer>

  <!-- Sales Order Type Hierarchy-->
  <EntityType Name="ESalesOrder" Key="Id">
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="AccountNum" Type="String"
              MaxLength="15" />
  </EntityType>
  <EntityType Name="EStoreSalesOrder"
              BaseType="Self.ESalesOrder">
    <Property Name="Tax" Type="Decimal"
              Precision="28" Scale="4" />
  </EntityType>

  <!-- Person EntityType -->
  <EntityType Name="ESalesPerson" Key="Id">
    <!-- Properties from SSalesPersons table-->
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="Bonus" Type="Decimal"
              Precision="28" Scale="4" />
    <!-- Properties from SEmployees table-->
    <Property Name="Title" Type="String"
              MaxLength="50" />
    <Property Name="HireDate" Type="DateTime" />
    <!-- Properties from the SContacts table-->
    <Property Name="Name" Type="String"
              MaxLength="50" />
    <Property Name="Contact" Type="Self.ContactInfo"
              Nullable="false" />
  </EntityType>
```

【００４５】

【表２】

```
    <ComplexType Name="ContactInfo">
      <Property Name="Email" Type="String"
                MaxLength="50" />
      <Property Name="Phone" Type="String"
                MaxLength="25" />
    </ComplexType>
    <Association Name="ESalesPersonOrder">
      <End Role="EOrder" Type="Self.ESalesOrder"
           Multiplicity="*" />
      <End Role="ESalesPerson" Multiplicity="1"
           Type="Self.ESalesPerson" />
    </Association>
  </Schema>
```

【００４６】

（高水準のアーキテクチャ）

このセクションは、ＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋのアーキテクチャの概要を示す。その主な機能コンポーネントは、図１に示されており、以下のものを含む。

【００４７】

データソース固有のプロバイダ。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ　１００は、ＡＤＯ．ＮＥＴデータプロバイダモデルを基礎とする。ＳＱＬ　Ｓｅｒｖｅｒ　１５１、１５２、リレーショナルソース１５３、非リレーショナル１５４、およびＷｅｂサービス１５５ソースなど、複数のデータソース用の特定のプロバイダ１２２～１２５がある。プロバイダ１２２～１２５を、ストア固有のＡＤＯ．ＮＥＴ　Ｐｒｏｖｉｄｅｒ　ＡＰＩ　１２１から呼び出すことができる。

【００４８】

ＥｎｔｉｔｙＣｌｉｅｎｔプロバイダ。ＥｎｔｉｔｙＣｌｉｅｎｔプロバイダ１１０は、具象的な概念プログラミングレイヤを表す。これは、新しい値ベースのデータプロバイダであり、ここでは、データは、ＥＤＭエンティティおよびリレーションシップに関してアクセスされ、エンティティベースのＳＱＬ言語（Ｅｎｔｉｔｙ　ＳＱＬ）を使用してクエリ／更新される。ＥｎｔｉｔｙＣｌｉｅｎｔプロバイダ１１１は、Ｅｎｔｉｔｙ　Ｄａｔａ　Ｓｅｒｖｉｃｅｓ　１１０パッケージの一部を形成し、Ｅｎｔｉｔｙ　Ｄａｔａ　Ｓｅｒｖｉｃｅｓ　１１０パッケージは、メタデータサービス１１２と、クエリおよび更新パイプライン１１３と、トランザクションサポート１１５と、ビューマネージャランタイム１１６と、複数のフラットなリレーショナルテーブルにまたがる更新可能ＥＤＭビューをサポートするビューマッピングサブシステム１１４とを含むこともできる。テーブルとエンティティとの間のマッピングは、マッピング仕様言語を介して宣言的に指定される。

【００４９】

Ｏｂｊｅｃｔ　Ｓｅｒｖｉｃｅｓおよび他のプログラミングレイヤ。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ　１００のＯｂｊｅｃｔ　Ｓｅｒｖｉｃｅｓコンポーネント１３１は、複数のエンティティにまたがる豊富なオブジェクト抽象化、これらのオブジェクトにまたがるサービスの豊富なセットを提供し、アプリケーションが、よく知られたプログラミング言語構造を使用して命令コーディング経験１６１内でプログラミングすることを可能にする。このコンポーネントは、オブジェクトの状態管理サービス（変更追跡、識別解決（ｉｄｅｎｔｉｔｙ　ｒｅｓｏｌｕｔｉｏｎ）を含む）を提供し、オブジェクトおよびリレーションシップをナビゲートし、ロードするサービスをサポートし、Ｘｌｉｎｑ　１３２などのコンポーネントを使用してＬＩＮＱおよびＥｎｔｉｔｙ　ＳＱＬを介するクエリを

サポートして、オブジェクトの更新および永続化を可能にする。

【００５０】

Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋは、１３０と類似の複数のプログラミングレイヤが、ＥｎｔｉｔｙＣｌｉｅｎｔプロバイダ１１１によって公開される値ベースのｅｎｔｉｔｙ ｄａｔａ ｓｅｒｖｉｃｅｓレイヤ１１０にプラグオンされることを可能にする。Ｏｂｊｅｃｔ Ｓｅｒｖｉｃｅｓ １３１コンポーネントは、ＣＬＲオブジェクトを表面化するそのようなプログラミングレイヤの１つであり、ＯＲＭ同様の機能性を提供する。

【００５１】

メタデータサービス１１２コンポーネントは、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋ １００の設計時間およびランタイムの必要性と、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋ上のアプリケーションとに関するメタデータを管理する。ＥＤＭ概念（エンティティ、リレーションシップ、ＥｎｔｉｔｙＳｅｔ、ＲｅｌａｔｉｏｎｓｈｉｐＳｅｔ）、ストア概念（テーブル、列、制約）、およびマッピング概念に関連する全てのメタデータが、メタデータインターフェースを介して公開される。メタデータコンポーネント１１２は、モデル主導型のアプリケーション設計をサポートするドメインモデル化ツールの間のリンクとしても機能する。

【００５２】

設計ツールおよびメタデータツール。Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋ １００は、ドメインデザイナ１７０と統合して、モデル主導型のアプリケーション開発を可能にする。このツールは、ＥＤＭ設計ツール、モデル化ツール１７１、マッピング設計ツール１７２、ブラウジング設計ツール１７３、バインディング設計ツール１７４、コード生成ツール１７５、およびクエリモデラーを含む。

【００５３】

サービス。レポート作成１４１、同期化１４２、ウェブサービス１４３、およびビジネス分析などの豊富なデータ中心サービスを、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋ １００を使用して構築することができる。

【００５４】

＜プログラミングパターン＞

ＡＤＯ．ＮＥＴ Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋは、ＬＩＮＱと一緒に、アプリケーションコードとデータとの間のインピーダンス不整合を大幅に減少させることによって、アプリケーション開発者の生産性を高める。このセクションでは、論理レイヤ、概念レイヤ、およびオブジェクト抽象化レイヤでのデータアクセスプログラミングパターンの進化を説明する。

【００５５】

例示のＡｄｖｅｎｔｕｒｅＷｏｒｋｓデータベースに基づく次のリレーショナルスキーマのフラグメントを検討されたい。このデータベースは、図２に図示されるような、リレーショナルスキーマに従う、ＳＣｏｎｔａｃｔｓ ２０１テーブル、ＳＥｍｐｌｏｙｅｅｓ ２０２テーブル、ＳＳａｌｅｓＰｅｒｓｏｎｓ ２０３テーブル、およびＳＳａｌｅｓＯｒｄｅｒｓ ２０４テーブルから構成される。

【００５６】

【表３】

```
SContacts (ContactId, Name, Email, Phone)
SEmployees (EmployeeId, Title, HireDate)
SSalesPersons (SalesPersonId, Bonus)
SSalesOrders (SalesOrderId, SalesPersonId)
```

【００５７】

ある日付の前に雇われた販売員の名前および雇われた日付を取得する、アプリケーションコードのフラグメントを考慮されたい（下記に示す）。このコードフラグメントには、回答を必要とするビジネス質問にほとんど関係しない４つの主な短所がある。第１に、こ

のクエリは、英語では非常に簡潔に述べることができるが、ＳＱＬステートメントは、非常に冗長的であり、開発者が、正規化されたリレーショナルスキーマを意識して、ＳＣｏｎｔａｃｔｓテーブル、ＳＥｍｐｌｏｙｅｅｓテーブル、およびＳＳａｌｅｓＰｅｒｓｏｎテーブルから適切な列を集めるのに必要なマルチテーブル結合を定式化する必要がある。さらに、基礎になるデータベーススキーマに対する全ての変更が、下記のコードフラグメントの対応する変更を必要とする。第２に、ユーザは、データソースへの明示的な関連付け（ｃｏｎｎｅｃｔｉｏｎ）を定義しなければならない。第３に、返される結果は、強く型付けされていないので、非既存の列の名前に対する全ての参照は、クエリが実行された後にのみキャッチされることになる。第４に、このＳＱＬステートメントは、Ｃｏｍｍａｎｄ　ＡＰＩに対するストリングプロパティであり、その定式化（ｆｏｒｍｕｌａｔｉｏｎ）における全てのエラーは、実行時にのみキャッチされることになる。このコードは、ＡＤＯ．ＮＥＴ　２．０を使用して記述されているが、コードパターンおよびその短所は、ＯＤＢＣ、ＪＤＢＣ、またはＯＬＥ－ＤＢなどの、全ての他のリレーショナルデータアクセスＡＰＩにあてはまる。

【００５８】

【表４】

```
void EmpsByDate(DateTime date) {
using( SqlConnection con =
  new SqlConnection (CONN_STRING) ) {
    con.Open();
    SqlCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
    SELECT SalesPersonID, FirstName, HireDate
    FROM SSalesPersons sp
      INNER JOIN SEmployees e
      ON sp.SalesPersonID = e.EmployeeID
      INNER JOIN SContacts c
      ON e.EmployeeID = c.ContactID
    WHERE e.HireDate < @date";
    cmd.Parameters.AddWithValue("@date",date);

    DbDataReader r = cmd.ExecuteReader();
    while(r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
        r["HireDate"], r["FirstName"]);
} } }
```

【００５９】

この例示のリレーショナルスキーマを、図３に図示されるようなＥＤＭスキーマを介して概念レベルでキャプチャすることができる。これは、ＳＣｏｎｔａｃｔｓ　２０１テーブル、ＳＥｍｐｌｏｙｅｅｓ　２０２テーブル、およびＳＳａｌｅｓＰｅｒｓｏｎｓ　２０３テーブルを抽象化する、エンティティ型ＥＳａｌｅｓＰｅｒｓｏｎ　３０２を定義する。これは、エンティティ型、ＥＳｔｏｒｅＯｒｄｅｒ　３０１とＥＳａｌｅｓＯｒｄｅｒ　３０３との間の継承関係もキャプチャする。

【００６０】

概念レイヤでの同等のプログラムは、次のように記述される。

【００６１】

【表５】

```
void EmpsByDate (DateTime date) {
using( EntityConnection con =
  new EntityConnection (CONN_STRING) ) {
    con.Open();
    EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
      SELECT VALUE sp
      FROM ESalesPersons sp
      WHERE sp.HireDate < @date";
    cmd.Parameters.AddWithValue ("date",
        date);
    DbDataReader r = cmd.ExecuteReader(
              CommandBehavior.SequentialAccess);
    while (r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
        r["HireDate"]], r["FirstName"])
} } }
```

【００６２】

このＳＱＬステートメントは、かなり単純化されており、ユーザは、もはや正確なデータベースレイアウトについて知る必要がない。さらに、アプリケーションロジックを、基礎になるデータベーススキーマに対する変更から分離することができる。しかし、このフラグメントは、それでもなお、ストリングベースであり、プログラミング言語型でチェックすることの利点が得られず、弱く型付けされた結果を返す。

【００６３】

エンティティの回りに薄いオブジェクトラッパーを追加し、Ｃ＃のＬＩＮＱ拡張を使用することによって、次のように、インピーダンス不整合のない同等の関数を再記述することができる。

【００６４】

【表６】

```
void EmpsByDate(DateTime date) {
  using (AdventureWorksDB aw =
    new AdventureWorksDB()) {
    var people = from p in aw.SalesPersons
             where p.HireDate < date
             select p;
    foreach (SalesPerson p in people) {
      Console.WriteLine("{0:d}\t{1}",
          p.HireDate, p.FirstName);
} } }
```

【００６５】

このクエリは単純である。すなわち、アプリケーションが、基礎になるデータベーススキーマに対する変更から（十分に）分離され、クエリが、Ｃ＃のコンパイラによって完全に型チェックされる。クエリに加えて、オブジェクトと対話し、オブジェクトに対して通常のＣｒｅａｔｅ、Ｒｅａｄ、Ｕｐｄａｔｅ、およびＤｅｌｅｔｅ（ＣＲＵＤ）オペレーションを実行することができる。これらの例を、更新処理のセクションで説明する。

【００６６】

＜Ｏｂｊｅｃｔ　Ｓｅｒｖｉｃｅｓ＞

Ｏｂｊｅｃｔ Ｓｅｒｖｉｃｅｓコンポーネントは、概念（エンティティ）レイヤ上のプログラミング／プレゼンテーションレイヤである。これは、プログラミング言語と値ベースの概念レイヤエンティティとの間の対話を容易にする複数のコンポーネントを備えている。プログラミング言語ランタイム（例えば．ＮＥＴ、Ｊａｖａ（登録商標））ごとに１つのオブジェクトサービスが存在することが予期される。．ＮＥＴ ＣＬＲをサポートするように設計される場合は、全ての．ＮＥＴ言語のプログラムが、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋと対話することができる。Ｏｂｊｅｃｔ Ｓｅｒｖｉｃｅｓは、次の主要コンポーネントから構成される。

【００６７】

ＯｂｊｅｃｔＣｏｎｔｅｘｔクラスは、データベース接続、メタデータワークスペース、オブジェクト状態マネージャ、およびオブジェクトマテリアライザを備えている。このクラスは、Ｅｎｔｉｔｙ ＳＱＬ構文またはＬＩＮＱ構文のいずれかでのクエリの定式化を可能にするために、オブジェクトクエリインターフェース、ＯｂｊｅｃｔＱｕｅｒｙ＜Ｔ＞を含み、強く型付けされたオブジェクトの結果をＯｂｊｅｃｔＲｅａｄｅｒ＜Ｔ＞として返す。ＯｂｊｅｃｔＣｏｎｔｅｘｔは、プログラミング言語レイヤと概念レイヤとの間のクエリおよび更新（すなわち、ＳａｖｅＣｈａｎｇｅｓ）オブジェクトレベルインターフェースも公開する。オブジェクト状態マネージャは、３つの主要な機能を有する。すなわち、（ａ）クエリ結果をキャッシュし、識別解決を提供し、オーバーラップするクエリ結果からオブジェクトをマージするようにポリシを管理し、（ｂ）メモリ内の変更を追跡し、（ｃ）更新処理インフラストラクチャに入力される変更リストを構築する、という３つの主要な機能を有する。オブジェクト状態マネージャは、キャッシュ内の各エンティティの状態、すなわち、デタッチ済み（キャッシュから）、追加済み、未変更、変更済み、および削除済み、という状態を管理し、その状態遷移を追跡する。オブジェクトマテリアライザ（Ｏｂｊｅｃｔ ｍａｔｅｒｉａｌｉｚｅｒ）は、概念レイヤからのエンティティ値と、対応するＣＬＲオブジェクトとの間で、クエリおよび更新中に変形を実行する。

【００６８】

＜マッピング＞

一実施形態では、ＡＤＯ．ＮＥＴ Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋなどの汎用データアクセスレイヤのバックボーンは、アプリケーションデータとデータベースに格納されたデータとの間のリレーションシップを確立するマッピング（ｍａｐｐｉｎｇ）とすることができる。アプリケーションは、オブジェクトレベルまたは概念レベルでオブジェクトをクエリして、更新し、これらのオペレーションは、マッピングを介してストアに変換される。マッピングソリューションによって対処する必要のある、複数の技術的課題がある。特に宣言的なデータ操作が不要である場合に、１対１のマッピングを使用してリレーショナルテーブルの各行を１つのオブジェクトとして公開するＯＲＭを構築することは、比較的容易である。しかし、より複雑なマッピング、セットベースのオペレーション、パフォーマンス、マルチＤＢＭＳベンダのサポート、および他の要件が加わるにつれて、アドホックソリューションは、すぐに手におえなくなる。

【００６９】

（問題：マッピングを介する更新）

マッピングを介してデータにアクセスすることの問題を、「ビュー」に関してモデル化することができる。すなわち、クライアントレイヤ内のオブジェクト／エンティティは、テーブルの行にまたがる豊富なビューと考えることができる。しかし、ビューのうち限られたクラスだけが更新可能であることは周知であり、例えば、商用データベースシステムは、結合または統合を含むビュー内の複数のテーブルに対する更新を許容しない。非常に単純なビューに対するものであっても、一意の更新変換を見つけることは、ビューによる更新振舞いについての内在する指定が不十分なために、ほとんど不可能である。研究により、ビューから更新セマンティクスを引き出すことは困難であり、かなりのユーザの専門知識を必要とする可能性があることが示されている。しかし、マッピング主導型のデータアクセスについては、ビューに対する全ての更新について明確に定義された変換が存在す

ることが有利である。

【００７０】

さらに、マッピング主導型のシナリオでは、更新の可能性の要件は、単一のビューを超えている。例えば、Ｃｕｓｔｏｍｅｒエンティティおよびｏｒｄｅｒエンティティを操作するビジネスアプリケーションは、効果的に２つのビューに対するオペレーションを実行する。場合によって、一貫したアプリケーションの状態は、複数のビューを同時に更新することによってのみ達成可能である。そのような更新についてのケースバイケースの変換は、更新ロジックの組み合わせ的爆発（ｃｏｍｂｉｎａｔｏｒｉａｌ　ｅｘｐｌｏｓｉｏｎ）を生じる可能性がある。その実装をアプリケーション開発者に委ねることは、アプリケーション開発者がデータアクセスの最も複雑な部分の１つに手作業で取り組むことを必要とするので、十分ではない。

【００７１】

（ＡＤＯ．ＮＥＴのマッピングアプローチ）

ＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、上記の課題に対処することを目指す革新的なマッピングアーキテクチャをサポートする。ＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、次のアイデアを活用する。

１．指定：マッピングは、明確に定義された意味を有し、広範囲のマッピングシナリオを専門家でないユーザの理解できる範囲内に置く、宣言的言語（ｄｅｃｌａｒａｔｉｖｅ　ｌａｎｇｕａｇｅ）を使用して指定される。

２．コンパイル：マッピングは、クエリビューおよび更新ビューと呼ばれる、ランタイムエンジン内でクエリ処理および更新処理を駆動する、双方向ビューにコンパイルされる。

３．実行：更新変換は、具体化されたビューの保守、すなわち堅固なデータベース技術を活用する一般的なメカニズムを使用して行われる。クエリ変換は、ビューアンフォールディング（ｖｉｅｗ　ｕｎｆｏｌｄｉｎｇ）を使用する。

【００７２】

この新しいマッピングアーキテクチャは、原理づけられた古くならない方法でマッピング主導型の技術の強力なスタックを構築することを可能にする。さらに、この新しいマッピングアーキテクチャは、すぐに実用に関連する興味深い研究の方向性を広げる。次のサブセクションは、マッピングの指定およびコンパイルを説明する。実行は、下記のクエリ処理および更新処理のセクションで検討する。本明細書で提供する例示的なマッピングアーキテクチャのさらなる態様および実施形態は、下記の「さらなる態様および実施形態」と題するセクションにおいても説明する。

【００７３】

（マッピングの指定）

マッピングは、マッピングフラグメントのセットを使用して指定される。各マッピングフラグメントは、$Q_{Entities}=Q_{Tables}$の形式の制約であり、ここで、$Q_{Entities}$は、エンティティスキーマ（アプリケーション側）に対するクエリであり、Q_{Tables}は、データベーススキーマ（ストア側）に対するクエリである。マッピングフラグメントは、エンティティデータのある部分がリレーショナルデータのある部分にどのように対応するかを記述する。すなわち、マッピングフラグメントは、他のフラグメントと独立に理解することができる、指定の基礎的な単位である。

【００７４】

説明のために、図４のサンプルのマッピングシナリオを考慮されたい。図４は、エンティティスキーマ（左）とデータベーススキーマ（右）との間のマッピングを図示する。ＸＭＬファイルまたはグラフィカルツールを使用して、このマッピングを定義することができる。エンティティスキーマは、本明細書のＥｎｔｉｔｙ　Ｄａｔａ　Ｍｏｄｅｌセクションのスキーマに対応する。ストア側には、４つのテーブル、すなわち、ＳＳａｌｅｓＯｒｄｅｒｓ、ＳＳａｌｅｓＰｅｒｓｏｎｓ、ＳＥｍｐｌｏｙｅｅｓ、およびＳＣｏｎｔａｃｔｓがある。エンティティスキーマ側には、２つのエンティティセット、すなわち、Ｅ

ＳａｌｅｓＯｒｄｅｒおよびＥＳａｌｅｓＰｅｒｓｏｎｓと、１つのアソシエーションセットＥＳａｌｅｓＰｅｒｓｏｎＯｒｄｅｒｓがある。

【００７５】

このマッピングは、図５に示されるエンティティスキーマおよびリレーショナルスキーマに対するクエリに関して表される。

【００７６】

図５では、フラグメント１は、ＥＳａｌｅｓＯｒｄｅｒｓ内の正確な型ＥＳａｌｅｓＯｒｄｅｒの全てのエンティティに関する（Ｉｄ，ＡｃｃｏｕｎｔＮｕｍ）値のセットが、ＩｓＯｎｌｉｎｅがｔｒｕｅであるＳＳａｌｅｓＯｒｄｅｒｓテーブルから取り出された（ＳａｌｅｓＯｒｄｅｒＩｄ，ＡｃｃｏｕｎｔＮｕｍ）値のセットと同一であることを示している。フラグメント２も同様である。フラグメント３は、アソシエーションセットのＥＳａｌｅｓＰｅｒｓｏｎＯｒｄｅｒｓを、ＳＳａｌｅｓＯｒｄｅｒｓテーブルにマッピングし、各アソシエーションエントリが、このテーブル内の各行の主キーと外部キーのペアに対応することを示している。フラグメント４、５、および６は、ＥＳａｌｅｓＰｅｒｓｏｎｓエンティティセット内のエンティティが、３つのテーブルＳＳａｌｅｓＰｅｒｓｏｎｓ、ＳＣｏｎｔａｃｔｓ、およびＳＥｍｐｌｏｙｅｅｓにまたがって分割されることを示している。

【００７７】

（双方向ビュー）

このマッピングは、ランタイムを駆動する双方向のＥｎｔｉｔｙ　ＳＱＬビューにコンパイルされる。クエリビューは、テーブルに関してエンティティを表し、更新ビューは、エンティティに関してテーブルを表す。

【００７８】

更新ビューは、仮想構造に関して永続データを指定するため、いくらか反直観的なものである可能性があるが、後で示すように、的確な方法で更新をサポートするために更新ビューを活用することができる。生成されるビューは、明確に定義された意味でマッピングを「尊重し（ｒｅｓｐｅｃｔ）」、下記のプロパティを有する（この提示は、わずかに単純化されており、特に永続状態が仮想状態によって完全には決定されないことに留意されたい）。

```
Entities = QueryViews(Tables)
Tables = UpdateViews(Entities)
Entities = QueryViews(UpdateViews(Entities))
```

【００７９】

最後の条件は、ラウンドトリップ基準（ｒｏｕｄｔｒｉｐｐｉｎｇ　ｃｒｉｔｅｒｉｏｎ）であり、これは、確実に、全てのエンティティデータを永続させ、ロスのない方法でデータベースから再アセンブルできるようにする。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋに含まれるマッピングコンパイラは、生成されたビューがラウンドトリップ基準を満たすことを保証する。このマッピングコンパイラは、そのようなビューを入力マッピングから作ることができない場合に、エラーを発生する。

【００８０】

図６は、図５のマッピングのマッピングコンパイラによって生成される双方向ビュー、すなわち、クエリビューと更新ビューを示す。一般に、これらのビューは、必要なデータ変形を明示的に指定するので、入力マッピングより著しく複雑である可能性がある。例えば、QV_1では、ＥＳａｌｅｓＯｒｄｅｒｓエンティティセットは、ＳＳａｌｅｓＯｒｄｅｒｓテーブルから構築され、その結果、ＥＳａｌｅｓＯｒｄｅｒまたはＥＳｔｏｒｅＳａｌｅｓＯｒｄｅｒのいずれかが、ＩｓＯｎｌｉｎｅフラグがｔｒｕｅであるか否かに応じてインスタンス化されることになる。ＥＳａｌｅｓＰｅｒｓｏｎｓエンティティセットをリレーショナルテーブルから再アセンブルするためには、ＳＳａｌｅｓＰｅｒｓｏｎｓテーブル、ＳＥｍｐｌｏｙｅｅｓテーブル、およびＳＣｏｎｔａｃｔｓテーブルの間で結合を実行する必要がある（QV_3）。

【００８１】
ラウンドトリップ基準を満たすクエリビューおよび更新ビューを手で書くことは、厄介なことであり、かなりのデータベース専門知識を必要とする。したがって、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋの本実施形態は、組み込みマッピングコンパイラによって作成されたビューだけを受け入れるが、他のコンパイラまたは手によって作成されたビューを受け入れることは、代替的な実施形態では確かに妥当と思われる。
【００８２】
（マッピングコンパイラ）
Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋは、ＥＤＭスキーマ、ストアスキーマ、およびマッピングからクエリビューおよび更新ビューを生成する、マッピングコンパイラを含む（メタデータのアーチファクトは、本明細書のメタデータのセクションで論じる）。これらのビューを、クエリパイプラインおよび更新パイプラインによって取り込む（ｃｏｎｓｕｍｅ）。最初のクエリがＥＤＭスキーマに対して実行されると、このコンパイラを、設計時またはランタイムのいずれかにおいて呼び出すことができる。このコンパイラで使用されるビュー生成アルゴリズムは、正確な再記述のためのａｎｓｗｅｒｉｎｇ－ｑｕｅｒｉｅｓ－ｕｓｉｎｇ－ｖｉｅｗｓ技法に基づく。
【００８３】
＜クエリ処理＞
（クエリ言語）
一実施形態では、Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋを、複数のクエリ言語を処理するように設計することができる。Ｅｎｔｉｔｙ ＳＱＬおよびＬＩＮＱの実施形態を、本明細書においてより詳細に説明するが、同一または類似の原理を他の実施形態に拡張できることは理解されよう。
【００８４】
・Ｅｎｔｉｔｙ ＳＱＬ
Ｅｎｔｉｔｙ ＳＱＬは、ＥＤＭインスタンスにクエリして操作するように設計された、ＳＱＬの派生物である。Ｅｎｔｉｔｙ ＳＱＬは、以下の方法で標準のＳＱＬを拡張する。
１．ＥＤＭ構造（エンティティ、リレーションシップ、複合型など）のネイティブサポート：コンストラクタ、メンバアクセッサ、型問合せ、リレーションシップナビゲーション、ネスト／アンネストなど。
２．名前空間。Ｅｎｔｉｔｙ ＳＱＬは、名前空間を、型および関数のグループ化構造として使用する（ＸＱｕｅｒｙおよび他のプログラミング言語と同様）。
３．拡張可能な関数。Ｅｎｔｉｔｙ ＳＱＬは、非組み込み関数をサポートする。全ての関数（ｍｉｎ、ｍａｘ、ｓｕｂｓｔｒｉｎｇなど）は、名前空間内で明示的に定義され、通常は基礎になるストアから、クエリにインポートされる。
４．ＳＱＬと比較される、サブクエリおよび他の構造についてのより直交の扱い（ｏｒｔｈｏｇｏｎａｌ ｔｒｅａｔｍｅｎｔ）。
【００８５】
Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋは、例えば、ＥｎｔｉｔｙＣｌｉｅｎｔプロバイダレイヤにおいてＯｂｊｅｃｔ Ｓｅｒｖｉｃｅｓコンポーネント内でクエリ言語として、Ｅｎｔｉｔｙ ＳＱＬをサポートすることができる。サンプルのＥｎｔｉｔｙ ＳＱＬクエリは、本明細書のプログラミングパターンのセクションに示されている。
【００８６】
・Ｌａｎｇｕａｇｅ Ｉｎｔｅｇｒａｔｅｄ Ｑｕｅｒｙ（ＬＩＮＱ）
ＬＩＮＱは、クエリ関連構造をＣ＃およびＶｉｓｕａｌ Ｂａｓｉｃなどのメインストリームプログラミング言語に導入する．ＮＥＴプログラミング言語における革新である。クエリ式（ｑｕｅｒｙ ｅｘｐｒｅｓｓｉｏｎｓ）は、外部ツールまたは言語プリプロセッサによって処理されるのではなく、言語自体のファーストクラス表現（ｆｉｒｓｔ－ｃｌａｓｓ ｅｘｐｒｅｓｓｉｏｎｓ）である。ＬＩＮＱは、クエリ式が、豊富なメタデー

タ、コンパイル時の構文チェック、静的な型付け、および以前には命令コードのみに使用可能であったＩｎｔｅｌｌｉＳｅｎｓｅから利益を得ることを可能にする。ＬＩＮＱは、トラバーサル、フィルタ、結合、射影、ソート、およびグループ化のオペレーションを、直接かつ宣言的な方法で全ての．ＮＥＴベースのプログラミング言語で表すことを可能にする、汎用の標準クエリ演算子のセットを定義する。Ｖｉｓｕａｌ　ＢａｓｉｃおよびＣ＃などの．ＮＥＴ言語も、クエリ理解、すなわち標準クエリ演算子を活用する言語構文拡張をサポートする。Ｃ＃でのＬＩＮＱを使用するクエリの例は、本明細書のプログラミングパターンのセクションに示されている。

【００８７】

（カノニカルコマンドツリー）

一実施形態において、カノニカルコマンドツリー（より単純にはコマンドツリー）を、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ内の全てのクエリのプログラム的（ツリー）表現とすることができる。Ｅｎｔｉｔｙ　ＳＱＬまたはＬＩＮＱを介して表されたクエリを、まず解析し、コマンドツリーに変換することができ、全ての後続処理を、このコマンドツリーに対して実行することができる。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、コマンドツリー構築／編集ＡＰＩを介してクエリを動的に構築する（または編集する）ことを可能にすることもできる。コマンドツリーは、クエリ、挿入、更新、削除、およびプロシージャ呼出しを表すことができる。コマンドツリーは、１つまたは複数の式（Ｅｘｐｒｅｓｓｉｏｎ）から構成される。式は、単に、ある計算を表し、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、定数、パラメータ、算術演算、リレーショナル演算（射影、フィルタ、結合など）、関数呼出しなどを含む、様々な式を提供することができる。最後に、コマンドツリーは、ＥｎｔｉｔｙＣｌｉｅｎｔプロバイダと基礎になるストア固有プロバイダとの間のクエリのための通信の手段として使用することができる。

【００８８】

（クエリパイプライン）

クエリ実行を、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの一実施形態では、データストアに委ねることができる。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋのクエリ処理のインフラストラクチャは、Ｅｎｔｉｔｙ　ＳＱＬまたはＬＩＮＱのクエリを、追加のアセンブリ情報とともに、基礎になるストアによって評価可能な１つまたは複数の基本的なリレーショナルのみのクエリに分解することに関与し、この追加のアセンブリ情報を使用して、より単純なクエリのフラットな結果を、より豊富なＥＤＭ構造に再形成する。

【００８９】

Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、例えば、ストアがＳＱＬ　Ｓｅｒｖｅｒ　２００と同様の機能をサポートしなければならないと仮定することができる。クエリを、このプロファイルに適合する、より単純なフラットなリレーショナルクエリに分解することができる。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋの他の実施形態は、ストアがクエリ処理のより大きな部分を担うことを可能にすることができるであろう。

【００９０】

典型的なクエリを、次のように処理することができる。

【００９１】

構文およびセマンティクスの分析。Ｅｎｔｉｔｙ　ＳＱＬクエリは、まずメタデータサービスコンポーネントからの情報を使用して解析され、意味的に分析される。ＬＩＮＱクエリは、適切な言語コンパイラの一部として解析され、分析される。

【００９２】

カノニカルコマンドツリーへの変換。クエリは、ここで、それが元々どのように表現されていたかに関わりなくコマンドツリーに変換され、検証される。

【００９３】

マッピングビューアンフォールディング。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋ内のクエリは、概念（ＥＤＭ）スキーマをターゲットとする。これらのクエリは、その代わりに基礎になるデータベーステーブルおよびビューを参照するように変換されなければならない

。このプロセスは、マッピングビューアンフォールディングと称されるが、データベースシステムのビューアンフォールディングメカニズムに類似する。ＥＤＭスキーマとデータベーススキーマとの間のマッピングは、クエリビューおよび更新ビューにコンパイルされる。クエリビューは、次いで、ユーザクエリにアンフォールドされ、このクエリは、ここで、データベーステーブルおよびビューをターゲットとする。

【００９４】

構造型の除去（Ｓｔｒｕｃｔｕｒｅｄ　Ｔｙｐｅ　Ｅｌｉｍｉｎａｔｉｏｎ）。構造型への全ての参照は、ここで、クエリから除去され、再アセンブリ情報に追加される（結果アセンブリをガイドするために）。これは、型コンストラクタ、メンバアクセッサ、型問合せの式に対する参照を含む。

【００９５】

射影のプルーニング。クエリを分析し、クエリ内の未参照式を除去する。

【００９６】

ネストのプルアップ。クエリ内の全てのネストオペレーション（ネストされたコレクションの構築）は、フラットリレーショナル演算子のみを含むサブツリー上のクエリツリーのルートにプッシュアップされる。通常、ネストオペレーションは、左外部結合（または、外部適用（ｏｕｔｅｒ　ａｐｐｌｙ））に変換され、続くクエリからのフラット結果が、適切な結果に再アセンブルされる（下記の結果アセンブリを参照されたい）。

【００９７】

変形。ヒューリスティック変形のセットを適用して、クエリを単純化する。これは、フィルタプッシュダウン、変換を結合するための適用、ｃａｓｅ式フォールディングなどを含む。余分な結合（自己結合、主キー、外部キー結合）は、このステージで除去される。ここでのクエリ処理インフラストラクチャは、いかなるコストベースの最適化も実行しないことに留意されたい。

【００９８】

プロバイダ固有コマンドへの変換。クエリ（すなわち、コマンドツリー）を、ここでプロバイダに渡し、おそらくはプロバイダのネイティブＳＱＬダイアレクトの、プロバイダ固有コマンドを作成する。このステップをＳＱＬＧｅｎと呼ぶ。

【００９９】

実行。プロバイダコマンドが実行される。

【０１００】

結果アセンブリ。プロバイダからの結果（ＤａｔａＲｅａｄｅｒ）が、以前に集められたアセンブリ情報を使用して適切な形式に再形成され、単一のＤａｔａＲｅａｄｅｒが、呼出し側に返される。

【０１０１】

マテリアライゼーション。Ｏｂｊｅｃｔ　Ｓｅｒｖｉｃｅｓコンポーネントを介して発行されたクエリについて、結果は、適切なプログラミング言語オブジェクトにマテリアライズされる。

【０１０２】

（ＳＱＬＧｅｎ）

前のセクションで論じたように、クエリ実行は、基礎になるストアに委ねられる。そのような実施形態において、クエリは、まず、ストアにとって適切な形式に変換されなければならない。しかし、異なるストアは、ＳＱＬの異なるダイアレクトをサポートし、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋがそれらの全てをネイティブにサポートすることは、現実的ではない。代わりに、クエリパイプラインは、クエリをコマンドツリーの形式でストアプロバイダに引き渡すことができる。ストアプロバイダは、次いで、そのコマンドツリーをネイティブコマンドに変換することができる。これは、コマンドツリーをプロバイダのネイティブＳＱＬダイアレクトに変換することによって達成することができ、したがって、このフェーズはＳＱＬＧｅｎである。次いで、結果のコマンドを実行して、関連する結果を作成することができる。

【０１０３】
＜更新処理＞
このセクションは、更新処理を例示的なＡＤＯ．ＮＥＴ　Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋでどのように実行できるかを説明する。一実施形態では、更新処理の２つのフェーズ、すなわち、コンパイル時とランタイムがある。本明細書で提供される双方向ビューのセクションでは、マッピング仕様をビュー表現のコレクションにコンパイルするプロセスを説明した。本セクションは、どのようにビュー表現をランタイムで活用して、オブジェクトレイヤで実行されるオブジェクト変更（または、ＥＤＭレイヤでのＥｎｔｉｔｙ　ＳＱＬ　ＤＭＬ更新）をリレーショナルレイヤにおける同等のＳＱＬ更新に変換するかについて説明する。
【０１０４】
（ビュー保守を介する更新）
例示的なＡＤＯ．ＮＥＴマッピングアーキテクチャで活用される洞察の１つは、マテリアライズビュー保守のアルゴリズムを活用して、双方向ビューを介して更新を伝搬することができることである。このプロセスを図７に示す。
【０１０５】
図７の右側に図示されたデータベースの内部のＴａｂｌｅｓは、永続データを保持する。図７の左側に図示されたＥｎｔｉｔｙＣｏｎｔａｉｎｅｒは、この永続データの仮想状態を表すが、これは、典型的にはＥｎｔｉｔｙＳｅｔｓ内のエンティティのわずかな部分のみが、クライアント上でマテリアライズされるからである。目標は、Ｅｎｔｉｔｉｅｓの状態の更新ΔＥｎｔｉｔｉｅｓを、Ｔａｂｌｅｓの永続状態の更新ΔＴａｂｌｅｓに変換することである。このプロセスを増分ビューの保守と称する。なぜなら、更新は、エンティティの変更された諸態様を表す更新ΔＥｎｔｉｔｉｅｓに基づいて実行されるからである。
【０１０６】
これは、次の２つのステップを使用して行うことができる。
１．ビュー保守：
ΔTables = ΔUpdateViews (Entities, ΔEntities)
２．ビューアンフォールディング：
ΔTables = ΔUpdateViews (QueryViews (Tables), ΔEntities)
【０１０７】
ステップ１では、ビュー保守アルゴリズムが更新ビューに適用される。これは、デルタ式のセットΔＵｐｄａｔｅＶｉｅｗｓを作成し、これにより、ΔＴａｂｌｅｓをΔＥｎｔｉｔｉｅｓ、およびＥｎｔｉｔｉｅｓのスナップショットからどのようにして取得するか知ることができる。後者は、クライアント側では完全にはマテリアライズされないので、ステップ２で、ビューアンフォールディングを使用して、デルタ式をクエリビューと組み合わせる。一緒に、これらのステップは、入力として初期データベース状態およびエンティティに対する更新を受け取ってデータベースに対する更新を計算する式を、生成する。
【０１０８】
このアプローチは、一度に１オブジェクトであると同時にセットベースである更新（すなわち、データ操作ステートメントを使用して表される更新）に対して機能する、クリーンで均一なアルゴリズムをもたらし、堅固なデータベース技術を活用する。実際には、ステップ１は、更新変換に十分であることが多い。なぜなら、多くの更新は、現在のデータベース状態に直接には依存しないからである。この情況では、ΔＴａｂｌｅｓ＝ΔＵｐｄａｔｅＶｉｅｗｓ（ΔＥｎｔｉｔｉｅｓ）である。ΔＥｎｔｉｔｉｅｓが、キャッシュ・エンティティに対する一度に１オブジェクトの変更のセットとして与えられる場合は、ΔＵｐｄａｔｅＶｉｅｗｓ式を計算するのではなく、変更されたエンティティに対してビュー保守アルゴリズムを直接実行することによって、ステップ１をさらに最適化することができる。
【０１０９】

（オブジェクトに対する更新の変換）

上記で概説されたアプローチを説明するために、少なくとも５年間勤務した有資格販売員にボーナスおよび昇格を与える次の例を検討されたい。

【０１１０】

【表７】

```
using(AdventureWorksDB aw =
      new AdventureWorksDB(…)) {
  // People hired at least 5 years ago
  Datetime d = DateTime.Today.AddYears(-5);
  var people = from p in aw.SalesPeople
           where p.HireDate < d
           select p;

  foreach(SalesPerson p in people) {
    if(HRWebService.ReadyForPromotion(p)) {
      p.Bonus += 10;
      p.Title = "Senior Sales Representative";
    }
  }
  aw.SaveChanges(); // push changes to DB
}
```

【０１１１】

ＡｄｖｅｎｔｕｒｅＷｏｒｋｓＤＢは、ツール生成されたクラスであり、該クラスは、データベース接続、メタデータワークスペース、およびオブジェクトキャッシュデータ構造を備えＳａｖｅＣｈａｎｇｅｓメソッドを公開する、ＯｂｊｅｃｔＣｏｎｔｅｘｔと呼ばれる汎用オブジェクトサービスクラスから派生する。Ｏｂｊｅｃｔ　Ｓｅｒｖｉｃｅｓセクションで説明したように、オブジェクトキャッシュは、エンティティのリストを管理し、このリストのそれぞれは、デタッチ済み（キャッシュから）、追加済み、未変更、変更済み、および削除済みという状態のうちの１つである。上記のコードフラグメントは、ＥＳａｌｅｓＰｅｒｓｏｎオブジェクトの肩書プロパティおよびボーナスプロパティを変更する更新を記述し、この肩書プロパティおよびボーナスプロパティは、ＳＥｍｐｌｏｙｅｅｓテーブルおよびＳＳａｌｅｓＰｅｒｓｏｎｓテーブルにそれぞれ格納される。ＳａｖｅＣｈａｎｇｅｓメソッドの呼出しによってトリガされる、オブジェクト更新を対応するテーブル更新に変形するプロセスは、次の４つのステップを含むことができる。

【０１１２】

変更リスト生成。エンティティセットごとの変更のリストが、オブジェクトキャッシュから作成される。更新は、削除および挿入される要素のリストとして表される。追加されるオブジェクトは、挿入になる。削除されるオブジェクトは、削除になる。

【０１１３】

値式の伝搬。このステップは、変更ビューおよび更新ビューのリスト（メタデータワークスペース内に保持される）をとり、増分マテリアライズビューの保守式、ΔＵｐｄａｔｅＶｉｅｗｓを使用して、オブジェクト変更のリストを、基礎になる作用テーブル（ａｆｆｅｃｔｅｄ　ｔａｂｌｅ）に対する代数ベーステーブルの挿入式および削除式のシーケンスに変形する。この例について、関連する更新ビューは、図６に示されたUV_2およびUV_3である。これらのビューは、単純な射影－選択クエリであり、したがって、ビュー保守ルールの適用は、容易である。挿入（Δ^+）および削除（Δ^-）について同一である、次のΔＵｐｄａｔｅＶｉｅｗｓ式を求める。

【０１１４】

【表８】

```
ΔSSalesPersons =    SELECT p.Id, p.Bonus
           FROM ΔESalesPersons AS p
ΔSEmployees =  SELECT p.Id, p.Title
           FROM ΔESalesPersons AS p
ΔSContacts =   SELECT p.Id, p.Name, p.Contact.Email,
           p.Contact.Phone FROM ΔESalesPersons AS p
```

【０１１５】

上記に示されるループが、エンティティE_{old}＝ＥＳａｌｅｓＰｅｒｓｏｎｓ（１，２０，“”，“Ａｌｉｃｅ”，Ｃｏｎｔａｃｔ（“ａ@ｓａｌｅｓ”，ＮＵＬＬ））を、E_{new}＝ＥＳａｌｅｓＰｅｒｓｏｎｓ（１，３０，“Ｓｅｎｉｏｒ．．．”，“Ａｌｉｃｅ”，Ｃｏｎｔａｃｔ（“ａ@ｓａｌｅｓ”，ＮＵＬＬ））に更新したと仮定する。次に、初期デルタは、挿入について、Δ^{+}ＥＳａｌｅｓＯｒｄｅｒｓ＝｛E_{new}｝であり、削除について、Δ^{-}ＥＳａｌｅｓＯｒｄｅｒｓ＝｛E_{old}｝である。Δ^{+}ＳＳａｌｅｓＰｅｒｓｏｎｓ＝｛（１，３０）｝、Δ^{-}ＳＳａｌｅｓＰｅｒｓｏｎｓ＝｛（１，２０）｝が得られる。次に、ＳＳａｌｅｓＰｅｒｓｏｎｓテーブルに対する計算された挿入および削除が、単一の更新に組み合わされ、この更新は、Ｂｏｎｕｓ値に３０をセットする。ＳＥｍｐｌｏｙｅｅｓに対するデルタが、同様に計算される。ＳＣｏｎｔａｃｔｓについて、Δ^{+}ＳＣｏｎｔａｃｔｓ＝Δ^{-}ＳＣｏｎｔａｃｔｓが得られ、したがって、更新は不要である。

【０１１６】

作用ベーステーブルに対するデルタを計算することに加え、このフェーズは、（ａ）参照整合性の制約を考慮に入れた、テーブル更新を実行すべき正しい順序、（ｂ）最終的な更新をデータベースに提示する前に必要な、ストアによって生成されたキーの取り出し、および（ｃ）オプティミスティック並行性制御に関する情報を収集することに関与する。

【０１１７】

ＳＱＬ　ＤＭＬまたはストアドプロシージャ呼出しの生成。このステップは、挿入されるデルタおよび削除されるデルタのリストに加えて、並行処理に関係する追加の注釈を、ＳＱＬ　ＤＭＬステートメントまたはストアドプロシージャ呼出しのシーケンスに変形する。この例では、影響される販売員について生成される更新ステートメントは、次の通りである。

【０１１８】

【表９】

```
BEGIN TRANSACTION
UPDATE [dbo].[SSalesPersons] SET [Bonus]=30
WHERE [SalesPersonID]=1
UPDATE [dbo].[SEmployees]
SET [Title]= N'Senior Sales Representative'
WHERE [EmployeeID]=1
END TRANSACTION
```

【０１１９】

キャッシュの同期化。更新が実行されると、キャッシュの状態は、データベースの新しい状態に同期化される。したがって、必要な場合に、ミニクエリ処理のステップを実行して、新しい変更されたリレーショナル状態を、それに対応するエンティティおよびオブジェクトの状態に変形する。

【０１２０】

＜メタデータ＞

メタデータサブシステムは、データベースカタログに類似し、Ｅｎｔｉｔｙ　Ｆｒａｍ

ｅｗｏｒｋの設計時およびランタイムのメタデータの必要を満たすように設計される。
【０１２１】
（メタデータアーチファクト）
メタデータアーチファクトは、例えば、以下のものを含むことができる。
【０１２２】
概念スキーマ（ＣＳＤＬファイル）：概念スキーマは、概念スキーマ定義言語（ＣＳＤＬ：Ｃｏｎｃｅｐｔｕａｌ　Ｓｃｈｅｍｅ　Ｄｅｆｉｎｉｔｉｏｎ　Ｌａｎｇｕａｇｅ）ファイル内で定義することができ、ＥＤＭ型（エンティティ型、リレーションシップ）と、データについてアプリケーションの概念ビューを記述するエンティティセットとを含む。
【０１２３】
ストアスキーマ（ＳＳＤＬファイル）：ストアスキーマ情報（テーブル、列、キーなど）は、ＣＳＤＬボキャブラリ項目を使用して表されることがある。例えば、ＥｎｔｉｔｙＳｅｔｓは、テーブルを示し、プロパティは、列を示す。これらを、ストアスキーマ定義言語（ＳＳＤＬ：Ｓｔｏｒｅ　Ｓｃｈｅｍａ　Ｄｅｆｉｎｉｔｉｏｎ　Ｌａｎｇｕａｇｅ）ファイル内で定義することができる。
【０１２４】
Ｃ－Ｓマッピング仕様（ＭＳＬファイル）：概念スキーマとストアスキーマとの間のマッピングは、典型的にはマッピング仕様言語（ＭＳＬ：Ｍａｐｐｉｎｇ　Ｓｐｅｃｉｆｉｃａｔｉｏｎ　Ｌａｎｇｕａｇｅ）ファイル内のマッピング仕様にキャプチャされる。この仕様を、マッピングコンパイラによって使用して、クエリビューおよび更新ビューを作成する。
【０１２５】
プロバイダマニフェスト：プロバイダマニフェストは、各プロバイダによってサポートされる機能性の記述を提供することがあり、次の例示的な情報を含むことができる。
１．プロバイダによってサポートされるプリミティブ型（ｖａｒｃｈａｒ、ｉｎｔなど）、およびそれらが対応するＥＤＭ型（ｓｔｒｉｎｇ、ｉｎｔ３２など）。
２．プロバイダに関する組み込み関数（および、そのシグネチャ）。
【０１２６】
この情報を、Ｅｎｔｉｔｙ　ＳＱＬパーサによってクエリ分析の一部として使用することができる。これらのアーチファクトに加えて、メタデータサブシステムは、生成されたオブジェクトクラス、および、該オブジェクトクラスとこれに対応する概念エンティティ型との間のマッピングを追跡し続けることもできる。
【０１２７】
（メタデータサービスのアーキテクチャ）
Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋによって消費されるメタデータは、異なるソースから異なるフォーマットで来る可能性がある。メタデータサブシステムを、統一された低水準のメタデータインターフェースのセットに上に作成することができ、該インターフェースのセットにより、メタデータランタイムが、異なるメタデータの永続フォーマット／ソースの詳細と独立に機能することが可能になる。
【０１２８】
例示的なメタデータサービスは、次を含むことができる。
異なるタイプのメタデータの列挙。
キーによるメタデータ検索。
メタデータのブラウジング／ナビゲーション。
一時メタデータの作成（例えば、クエリ処理用）。
セッション独立のメタデータのキャッシュと再使用。
【０１２９】
メタデータサブシステムは、次のコンポーネントを含む。メタデータキャッシュは、異なるソースから取り出されたメタデータをキャッシュし、メタデータを取り出して操作す

る共通ＡＰＩを顧客に提供する。メタデータは、異なる形式で表され、異なる位置に格納されることがあるので、メタデータサブシステムは、ローダーインターフェースを有利にサポートすることができる。メタデータローダーは、ローダーインターフェースを実装し、適切なソース（ＣＳＤＬ／ＳＳＤＬファイルなど）からのメタデータのロードに関与する。メタデータワークスペースは、いくつかのメタデータを集約して、メタデータの完全なセットをアプリケーションに提供する。メタデータワークスペースは、通常、概念モデル、ストアスキーマ、オブジェクトクラス、およびこれらの構造の間のマッピングに関する情報を含む。

【０１３０】

＜ツール＞

一実施形態において、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、開発の生産性を高めるための設計時のツールのコレクションを含むこともできる。例示的なツールは、次の通りである。

【０１３１】

モデルデザイナ：アプリケーションの開発における早期のステップの１つは、概念モデルの定義である。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、アプリケーション設計者および分析者が、エンティティおよびリレーションシップに関してアプリケーションの主な概念を記述することを可能にする。モデルデザイナは、この概念モデル化のタスクを対話的に実行することを可能にするツールである。設計のアーチファクトは、データベース内でその状態を永続することができるＭｅｔａｄａｔａコンポーネントに直接にキャプチャされる。モデルデザイナは、モデル記述（ＣＳＤＬを介して指定される）を生成して、消費することもでき、ＥＤＭモデルをリレーショナルメタデータから合成することができる。

【０１３２】

マッピングデザイナ：ＥＤＭモデルが設計されると、開発者は、概念モデルをリレーショナルデータベースにどのようにマッピングするかを指定することができる。このタスクは、マッピングデザイナによって容易にされ、マッピングデザイナは、図８に図示されるユーザインターフェースを提示することができる。マッピングデザイナは、ユーザインターフェースの左側に提示されるエンティティスキーマ内のエンティティおよびリレーションシップが、図８のユーザインターフェースの右側に提示されるデータベーススキーマに反映されるようにデータベース内のテーブルおよび列にどのようにマッピングされるかを、開発者が記述するのを助ける。図８の中央セクションに提示されたグラフ内のリンクは、Ｅｎｔｉｔｙ　ＳＱＬクエリと同等のものとして宣言的に指定されたマッピング式を、視覚化する。これらの式は、クエリビューおよび更新ビューを生成する双方向のマッピングコンパイルコンポーネントへの入力となる。

【０１３３】

コード生成：ＥＤＭ概念モデルは、ＡＤＯ．ＮＥＴコードパターン（コマンド、接続、データリーダー）に基づいてよく知られた対話モデルを提供するので、多くのアプリケーションに十分である。しかし、多くのアプリケーションは、強く型付けされたオブジェクトとしてデータと対話することを好む。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、ＥＤＭモデルを入力として取り、エンティティ型の強く型付けされたＣＬＲクラスを作成する、コード生成ツールのセットを含む。これらのコード生成ツールは、強く型付けされたオブジェクトコンテキスト（例えば、ＡｄｖｅｎｔｕｒｅＷｏｒｋｓＤＢ）を生成することもでき、この強く型付けされたオブジェクトコンテキストは、モデルによって定義された全てのエンティティおよびリレーションセットの強く型付けされたコレクション（例えば、ＯｂｊｅｃｔＱｕｅｒｙ＜ＳａｌｅｓＰｅｒｓｏｎ＞）を公開する。

【０１３４】

さらなる態様および実施形態

＜マッピングサービス＞

一実施形態において、図１の１１４などのマッピングコンポーネントは、マッピングの全ての態様を管理し、Ｅｎｔｉｔｙ　Ｃｌｉｅｎｔプロバイダ１１１によって内部的に使

用される。マッピングは、２つの潜在的に異なる型空間内の構造の間の変形を論理的に指定する。例えば、エンティティ（概念空間内で、その用語が上記で使用されるとき）を、図８に図式的に示されるように、ストレージ空間内のデータベーステーブルに関して指定することができる。

【０１３５】

規定のマッピング（ｐｒｅｓｃｒｉｂｅｄ　ｍａｐｐｉｎｇ）は、システムが構造の適切なマッピングを自動的に決定するマッピングである。非規定のマッピング（ｎｏｎ－ｐｒｅｓｃｒｉｂｅｄ　ｍａｐｐｉｎｇ）は、アプリケーション設計者がマッピングの様々なファセットを制御することを可能にする。マッピングは、複数のファセットを有することができる。マッピングのエンドポイント（エンティティ、テーブルなど）、マッピングされるプロパティのセット、更新の振舞い、遅延ローディングなどのランタイムの影響、更新時の衝突解決の振舞いなどが、そのようなファセットのごく部分的なリストである。

【０１３６】

一実施形態において、マッピングコンポーネント１１４は、マッピングビューを作成することができる。ストレージ空間とスキーマ空間との間のマッピングを検討されたい。エンティティは、１つまたは複数のテーブルからの行で構成される。クエリビューは、スキーマ空間内のエンティティを、ストレージ空間内のテーブルに関するクエリとして表す。エンティティは、クエリビューを評価することによってマテリアライズすることができる。

【０１３７】

エンティティのセットに対する変更を、対応するストアテーブルに戻して反映させる必要があるときは、その変更を、逆の方法でクエリビューを介して伝搬することができる。これは、データベースにおけるビュー／更新の問題に類似し、更新伝搬プロセスは、論理的に、反対のクエリビューに対して更新を実行する。このために、更新ビューという概念を導入し、これらのビューは、エンティティに関してストアテーブルを記述し、反対のクエリビューと考えることができる。

【０１３８】

しかし、多くの場合に、本当に関心を持っているのは、増分変更である。更新デルタビュー（Ｕｐｄａｔｅ　Ｄｅｌｔａ　Ｖｉｅｗｓ）は、対応するエンティティコレクションに対する変更に関して、テーブルに対する変更を記述するビュー（クエリ）である。したがって、エンティティコレクション（または、アプリケーションオブジェクト）に関する更新処理は、更新デルタビューを評価することによってテーブルに対する適切な変更を計算すること、および、その後これらの変更をテーブルに適用することを含む。

【０１３９】

同様の方法で、クエリデルタビューは、基礎になるテーブルに対する変更に関して、エンティティコレクションに対する変更を記述する。無効化、より一般的には通知が、クエリデルタビューの使用を必要とする可能性があるシナリオである。

【０１４０】

データベース内のビューのように、クエリとして表されたマッピングビューを、ユーザクエリを用いて構成することができ、このことは、マッピングのより一般化された取り扱いにつながる。同様に、クエリとして表されたマッピングデルタビューは、更新を扱うためのより一般的で的確なアプローチを可能にする。

【０１４１】

一実施形態において、マッピングビューの能力を制限することができる。マッピングビューで使用されるクエリ構造は、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋによってサポートされる全てのクエリ構造のサブセットのみとすることができる。これにより、特にデルタ式の場合において、より単純かつ効率的なマッピング式が可能になる。

【０１４２】

デルタビューを、代数変更計算スキーマを使用して、マッピングコンポーネント１１４内で計算して、更新（およびクエリ）ビューから更新（およびクエリ）デルタビューを作

ることができる。代数変更計算スキーマのさらなる態様については後述する。

【０１４３】

更新デルタビューは、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋが、計算アプリケーションによって行われるエンティティ変更を、データベース内のストアレベルの更新に自動的に変換することによって、更新をサポートすることを可能にする。しかし、多くの場合に、マッピングは、パフォーマンスおよび／またはデータの整合性のために追加の情報を用いて増補されなければならない。

【０１４４】

一部の事例では、基礎になるストアテーブルの一部または全てに対して、エンティティにおける更新を直接マッピングすることは、望ましくないことがある。そのような事例では、更新を、ストアドプロシージャを介してファンネルし、データ検証を可能にすると同時に信頼境界を維持しなければならない。マッピングは、エンティティにおける更新およびクエリを処理するためのストアドプロシージャの指定を可能にする。

【０１４５】

マッピングは、Ｏｂｊｅｃｔ　Ｓｅｒｖｉｃｅｓ　１３１内のオプティミスティック並行性制御のサポートも提供することができる。特に、あるエンティティのプロパティを、タイムスタンプフィールドまたはバージョンフィールドなどの並行性制御フィールドとしてマークすることができ、ストアにおける並行性制御フィールドの値がエンティティ内の並行性制御フィールドと同一である場合にのみ、これらのオブジェクトに対する変更が成功する。両方のオプティミスティック並行性制御フィールドは、ストア固有レイヤ１２０ではなく、アプリケーションオブジェクトレイヤでのみ関連することに留意されたい。

【０１４６】

一実施形態において、アプリケーション設計者は、ＭＳＬを使用して、マッピングの様々な態様を記述することができる。典型的なマッピング仕様は、次のセクションの１つまたは複数を含む。

１．Ｄａｔａ領域は、クラス、テーブル、および／またはＥＤＭ型の記述を含むことができる。これらの記述は、既存のクラス／テーブル／型を記述することができ、または、そのようなインスタンスを生成するのに使用することができる。サーバ生成値、制約、主キーなどは、このセクションの一部として指定される。

２．Ｍａｐｐｉｎｇセクションは、型空間の間の実際のマッピングを記述する。例えば、ＥＤＭエンティティの各プロパティは、テーブル（またはテーブルのセット）からの１つまたは複数の列に関して指定される。

３．Ｒｕｎｔｉｍｅ領域は、実行を制御する様々なノブ、例えば、オプティミスティック並行性制御パラメータおよびフェッチ戦略を指定することができる。

【０１４７】

（マッピングコンパイラ）

一実施形態において、ドメインモデル化ツールのマッピングコンポーネント１７２は、マッピング仕様をクエリビュー、更新ビュー、および対応するデルタビューにコンパイルするマッピングコンパイラを含むことができる。図９は、クエリビューおよび更新ビューを生成するためのＭＳＬのコンパイルを図示する。

【０１４８】

コンパイルパイプラインは、次のステップを実行する。

１．ＡＰＩ　９００から呼び出されるビュージェネレータ９０２が、オブジェクト←→エンティティ間のマッピング情報９０１（ＭＳＬを介して指定される）を変換し、Ｏ←→Ｅ（オブジェクト－エンティティ）空間内のクエリビュー、更新ビュー、および対応する（クエリおよび更新）デルタ式９０４を作成する。この情報は、メタデータストア９０８に置くことができる。

２．ビュージェネレータ９０６が、エンティティ←→ストア間のマッピング情報９０３（ＭＳＬを介して指定される）を変換し、Ｅ←→Ｓ（エンティティ－ストア）空間内のクエリビュー、更新ビュー、および対応する（クエリおよび更新）デルタ式９０７を作成す

る。この情報は、メタデータストア９０８に置くことができる。
３．依存性分析９０９コンポーネントは、ビュージェネレータ９０６によって作成されたビューを検査し、参照整合性および他のそのような制約に反しない更新について一貫した依存性順序（ｄｅｐｅｎｄｅｎｃｙ　ｏｒｄｅｒ）９１０を決定する。この情報は、メタデータストア９０８に置くことができる。
４．次に、ビュー、デルタ式、および依存性順序９０８が、メタデータサービスコンポーネント（図１の１１２）に渡される。

【０１４９】

（更新処理）
このセクションは、更新処理パイプラインを説明する。一実施形態において、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、２種類の更新をサポートすることができる。単一オブジェクト変更とは、オブジェクトグラフをナビゲートしている間に個々のオブジェクトに対して行われる変更である。単一オブジェクト変更について、システムは、現在のトランザクションにおいて作成、更新、および削除されたオブジェクトを追跡する。これは、オブジェクトレイヤにおいてのみ使用可能である。クエリベースの変更とは、例えばテーブルを更新するためにリレーショナルデータベースで行われるように、オブジェクトクエリに基づいて更新／削除ステートメントを発行することによって実行される変更である。図１の１３１などのＯｂｊｅｃｔ　Ｐｒｏｖｉｄｅｒは、単一オブジェクト変更をサポートするように構成されることがあるが、クエリベースの変更をサポートするように構成されないことがある。一方、Ｅｎｔｉｔｙ　Ｃｌｉｅｎｔプロバイダ１１１は、クエリベースの変更をサポートすることができるが、単一オブジェクト変更をサポートすることはできない。

【０１５０】

図１０は、一例示的実施形態における更新処理の図を提供する。図１０では、アプリケーションレイヤ１０００のアプリケーションのユーザ１００１は、そのようなアプリケーションによって操作されるデータに対する変更を保存する１００２ことができる。オブジェクトプロバイダレイヤ１０１０では、変更リストがコンパイルされる１０１１。変更グループ化１０１２は、変更リストに対して実行される。制約処理１０１３は、メタデータストア１０１７に保存される制約情報および依存性モデル１０２２を作ることができる。拡張オペレーションが実行される１０１４。並行性制御式を生成し１０１５、並行性モデル１０２３をメタデータストア１０１７に保存することができる。オブジェクト－エンティティコンバータ１０１６は、オブジェクト－エンティティデルタ式１０２４をメタデータストア１０１７に保存することができる。

【０１５１】

エンティティ式ツリー１０１８は、ＥＤＭ　Ｐｒｏｖｉｄｅｒレイヤ１０３０に渡される。選択的更新スプリッタ１０３１は、必要に応じてある種の更新を選択し、分割することができる。ＥＤＭストアコンバータ１０３２は、エンティティ－ストアデルタ式１０３３をメタデータストア１０３６に保存することができる。クエリビューアンフォールディングコンポーネント１０３５は、クエリマッピングビュー１０３５をメタデータストア１０３６に保存することができる。エンティティ－ストア補正（ｃｏｍｐｅｎｓａｔｉｏｎ）１０３７を実行し、ストア式ツリー１０３８をストアプロバイダレイヤ１０４０に渡す。

【０１５２】

ストアプロバイダレイヤ１０４０では、単純化（ｓｉｍｐｌｉｆｉｅｒ）コンポーネント１０４１が最初に動作し、続いてＳＱＬ生成コンポーネント１０４２が、データベース１０４４に対して実行されるＳＱＬ更新１０４３を生成することができる。全ての更新結果を、サーバ生成値の処理のためにＥＤＭプロバイダレイヤ１０３０内のコンポーネント１０３９に渡すことができる。コンポーネント１０３９は、結果をオブジェクトプロバイダレイヤ内の同様のコンポーネント１０２１に渡すことができる。最後に、全ての結果または更新確認１００３が、アプリケーションレイヤ１０００に返される。

【０１５３】
上述したように、更新デルタビューは、マッピングコンパイルの一部として生成される。これらのビューを更新プロセスで使用して、ストアでのテーブルに対する変更を識別する。
【０１５４】
ストアでの関係テーブルのセットについて、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、ある順序で更新を有利に適用することができる。例えば、外部キー制約の存在は、変更が特定のシーケンスで適用されることを必要とする場合がある。依存性分析フェーズ（マッピングコンパイルの一部）は、コンパイル時に計算可能な全ての依存性順序付け要件を識別する。
【０１５５】
一部の事例では、静的な依存性分析の技法は、例えば循環参照整合性制約（または自己参照整合性制約）に関して不十分であることがある。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、オプティミスティックアプローチを採用し、そのような更新の進行を可能にする。ランタイムで、循環が検出されると、例外を発生する。
【０１５６】
図１０に図示されるように、アプリケーションレイヤ１０００におけるインスタンスベースの更新に関する更新処理パイプラインは、次のステップを有する。
【０１５７】
変更グループ化１０１２：変更トラッカからの異なるオブジェクトコレクションに従って変更をグループ化する。例えば、コレクションＰｅｒｓｏｎに関する全ての変更を、そのコレクションの挿入セット、削除セット、および更新セットにグループ化する。
【０１５８】
制約処理１０１３：このステップは、ビジネスロジックが値レイヤで実行されないという事実を補正する全てのオペレーションを実行し、本質的に、オブジェクトレイヤが変更セットを拡張することを可能にする。カスケード削除補正および依存性順序付け（それぞれのＥＤＭ制約に対する）が、ここで実行される。
【０１５９】
拡張オペレーションの実行１０１４：追加の（例えば削除）オペレーションが実行され、その結果、対応するビジネスロジックを実行できるようになる。
【０１６０】
並行性制御式ジェネレータ１０１５：変更されたオブジェクトが不整合（ｓｔａｌｅ）かどうかを検出するために、マッピングメタデータで指定されるタイムスタンプの列または列のセットをチェックする式を生成することができる。
【０１６１】
オブジェクト－ＥＤＭの変換１０１６：挿入、削除、および更新のオブジェクトセットに関して指定された変更リストは、ここで、メタデータストア１０１７に格納されたマッピングデルタ式を使用して変換され、これらのマッピングデルタ式は、図９を参照して説明したマッピングコンパイルの後に格納される。このステップの後に、変更は、ＥＤＭエンティティに関してのみ表される式ツリー１０１８として使用可能である。
【０１６２】
ステップ１０１８からの式ツリーは、次に、ＥＤＭ－Ｐｒｏｖｉｄｅｒレイヤ１０３０内のＥＤＭプロバイダに渡される。ＥＤＭプロバイダでは、式ツリーが処理され、変更がストアに送られる。この式ツリー１０１８を別の方法で作成することもでき、アプリケーションがＥＤＭプロバイダに対して直接にプログラミングすると、そのアプリケーションは、ＥＤＭプロバイダに対してＤＭＬステートメントを実行することができることに留意されたい。そのようなＤＭＬステートメントは、まず、ＥＤＭプロバイダによってＥＤＭ式ツリー１０１８に変換される。ＤＭＬステートメントまたはアプリケーションレイヤ１０００から取得される式ツリーは、次の方法で処理される。
【０１６３】

選択的更新スプリッタ１０３１：このステップでは、更新の一部が、挿入および削除に分離される。一般に、更新を、そのままでより下位のレイヤに伝搬する。しかし、ある事例では、デルタ式のルールがその事例について開発されていないこと、または正しい変換が実際にベーステーブルに対する挿入および／または削除をもたらすことのいずれかの理由で、そのような更新を実行することができない場合がある。

【０１６４】

ＥＤＭ－ストア変換１０３２：ＥＤＭレベル式ツリー１０１８、適切なマッピングからデルタ式を使用してストア空間に変換される。

【０１６５】

クエリマッピングビューアンフォールディング１０３４：式ツリー１０１８は、何らかのＥＤＭレベル概念を含むことがある。これらを除去するために、クエリマッピングビュー１０３５を使用して式ツリーをアンフォールドし、ストアレベルの概念のみに関するツリー１０３８を取得する。ツリー１０３８は、オプションとして、Ｅ－Ｓ補正コンポーネント１０３７によって処理される。

【０１６６】

現在ストア空間の項目である式ツリー１０３８は、ここで、以下のステップを実行するストアプロバイダレイヤ１０４０内のストアプロバイダに与えられる。

単純化１０４１：式ツリーは、論理式変換ルールを使用することによって単純化される。

ＳＱＬの生成１０４２：式ツリーが与えられると、ストアプロバイダは、式ツリー１０３８から実際のＳＱＬ　１０４３を生成する。

ＳＱＬの実行１０４４：実際の変更が、データベースに対して実行される。

サーバ生成値：サーバによって生成された値は、ＥＤＰレイヤ１０３０に返される。プロバイダ１０４４は、サーバ生成値をレイヤ１０３０内のコンポーネント１０３９に渡し、コンポーネント１０３９は、マッピングを使用してこれらをＥＤＭ概念に変換する。アプリケーションレイヤ１０００は、これらの更新１００３をピックアップし、そのレイヤ内で利用される様々なアプリケーションおよびオブジェクトにインストールされるオブジェクトレベル概念に伝搬する。

【０１６７】

多くの場合に、ストアテーブルは、例えばデータベースアドミニストレータ（ＤＢＡ：Ｄａｔａｂａｓｅ　Ａｄｍｉｎｉｓｔｒａｔｏｒ）ポリシのために、直接更新可能ではない可能性がある。テーブルに対する更新は、ある妥当性チェックを実行できるようにするために、ストアドプロシージャを介してのみ可能であることがある。そのような情況では、マッピングコンポーネントは、「生の（ｒａｗ）」挿入、削除、および更新のＳＱＬステートメントを実行するのではなく、オブジェクト変更をこれらのストアドプロシージャへの呼出しに変換しなければならない。他の場合では、「ストアド」プロシージャを、ＥＤＰ　１０１０またはアプリケーションレイヤ１０００で指定することができ、その場合、マッピングコンポーネントは、変更されたオブジェクトをＥＤＭ空間に変換し、その後、適切なプロシージャを呼び出さなければならない。

【０１６８】

これらのシナリオを可能にするために、ＭＳＬは、ストアドプロシージャをマッピングの一部として指定することを可能にし、さらに、ＭＳＬは、様々なデータベースの列をストアドプロシージャのパラメータにどのようにマッピングするかを指定するメカニズムもサポートする。

【０１６９】

ＥＤＰレイヤ１０１０は、オプティミスティック並行性制御をサポートする。ＣＤＰが変更のセットをストアに送信するとき、変更される行が、別のトランザクションによって既に変更されている場合がある。ＣＤＰは、ユーザがそのような衝突を検出でき、その後、そのような衝突を解決できるような方法をサポートしなければならない。

【０１７０】

ＭＳＬは、衝突検出の単純なメカニズム、すなわち、タイムスタンプ、バージョン番号、変更済みの列をサポートする。衝突が検出されると、例外が発生し、衝突しているオブジェクト（またはＥＤＭエンティティ）は、アプリケーションによる衝突解決に使用可能である。

【０１７１】

＜例示的なマッピング要件＞

マッピングインフラストラクチャは、有利には、様々なオペレーションをアプリケーション空間からリレーショナル空間に変換する能力を提供することができ、例えば、開発者によって記述されたオブジェクトクエリが、リレーショナル（ストレージ）空間に変換される。これらの変換は、データの過剰なコピーを伴わない効率的なものでなければならない。マッパー（ｍａｐｐｅｒ）は、次の例示的なオペレーションの変換を提供することができる。

【０１７２】

１．クエリ：オブジェクトクエリは、バックエンドリレーショナルドメインに変換される必要があり、データベースから取得されるタプル（ｔｕｐｌｅｓ）は、アプリケーションオブジェクトに変換される必要がある。これらのクエリを、セットベースのクエリ（例えば、ＣＳＱＬまたはＣ＃　Ｓｅｑｕｅｎｃｅ）またはナビゲーションベース（例えば、単純に参照に従うこと）とすることができることに留意されたい。

【０１７３】

２．更新：アプリケーションによってそのオブジェクトに対して行われる変更は、元のデータベースに伝搬される必要がある。やはり、オブジェクトに対して行われる変更は、セットベースとするか、または個々のオブジェクトに対する変更とすることができる。考慮すべきもう１つの側面は、変更されているオブジェクトが、完全にメモリにロードされるか、または部分的にロードされるかである（例えば、オブジェクトにぶら下がっているコレクションが、メモリ内に存在しない場合がある）。部分的にロードされたオブジェクトに対する更新について、これらのオブジェクトがメモリに完全にロードされることを必要としない設計が、好ましいことがある。

【０１７４】

３．無効化および通知：ミドルティアまたはクライアントティアで実行中のアプリケーションは、一部のオブジェクトがバックエンドで変更する際に、通知されることを望む場合がある。したがって、ＯＲマッピングコンポーネントは、登録をオブジェクトレベルでリレーショナル空間に変換しなければならず、同様に、変更されるタプルに関するメッセージがクライアントによって受信されるとき、ＯＲマッパーは、これらの通知をオブジェクト変更に変換しなければならない。ＷｉｎＦＳが、そのような「通知」を、そのＷａｔｃｈｅｒメカニズムを介してサポートすることに留意されたい。しかし、この場合は、マッピングは規定されており、Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋは、非規定のマッピングに対するＷａｔｃｈｅｒをサポートしなければならない。

【０１７５】

４．通知と同様のメカニズムも、ミドルティアまたはクライアントティアで実行中のＥｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋプロセスから不整合のオブジェクト（ｓｔａｌｅ　ｏｂｊｅｃｔ）を無効化するために必要とされる。Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋが、衝突する読み取り／書き込みを処理するためにオプティミスティック並行性制御のサポートを提供する場合、アプリケーションは、（トランザクションが、オブジェクトの読み取り／書き込みのためにアボートされないように）Ｅｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋでキャッシュされるデータが適度に新しいことを保証し、できそうでない場合は、アプリケーションは、古いデータに関して決定をし、および／またはそのトランザクションを後にアボートすることができる。したがって、通知と同様に、ＯＲマッパーは、データベースサーバからの「無効化」メッセージを、オブジェクト無効化に変換しなければならないことがある。

【０１７６】

５．バックアップ／リストア／同期：エンティティのバックアップおよびミラーリングは、一部の実施形態に組み込むことができる２つの特徴である。これらの特徴の要件は単に、ＯＲマッパーの観点から、エンティティに対する特殊化されたクエリに変換することができ、あるいは、そのようなオペレーションの特別なサポートを提供することができる。同様に、同期は、オブジェクト変更、衝突などをストアに変換するため、およびその逆に変換するために、ＯＲマッピングエンジンからのサポートを必要とする。

【０１７７】

６．並行性制御への関与：ＯＲマッパーは、有利には、アプリケーションがオプティミスティック並行性制御を使用することができる異なる方法、例えば、タイムスタンプ値、フィールドの何らかの特定のセットを使用することなどをサポートすることができる。ＯＲマッパーは、タイムスタンププロパティなどの並行性制御情報を、オブジェクト空間へ／オブジェクト空間から、リレーショナル空間から／リレーショナル空間へ変換しなければならない。ＯＲマッパーは、ペシミスティック並行性制御（ｐｅｓｓｉｍｉｓｔｉｃ ｃｏｎｃｕｒｒｅｎｃｙ ｃｏｎｔｒｏｌ）（例えば、Ｈｉｂｅｒｎａｔｅのような）のサポートさえ提供することができる。

【０１７８】

７．ランタイムエラー報告。本明細書で説明される例示的な実施形態では、ランタイムエラーは、通常、ストレージレベルで発生する。これらのエラーを、アプリケーションレベルに変換することができる。ＯＲマッパーを使用して、これらのエラー変換を容易にすることができる。

【０１７９】

＜マッピングシナリオ＞

Ｅｎｔｉｔｙ Ｆｒａｍｅｗｏｒｋがサポートできる例示的な開発シナリオを論じる前に、ＯＲマッパーの様々な論理的な部分を説明する。一実施形態では、図１１に図示されるように、ＯＲマッピングに５つの部分がある。

【０１８０】

１．オブジェクト／クラス／ＸＭＬ（別名、アプリケーション空間）１１０１：開発者は、好みの言語でクラスおよびオブジェクトを指定し、最終的に、これらのクラスは、ＣＬＲアセンブルにコンパイルされ、リフレクションＡＰＩを介してアクセス可能である。これらのクラスは、永続メンバおよび非永続メンバも含み、また、言語固有の詳細をこの部分に含めることができる。

【０１８１】

２．Ｅｎｔｉｔｙ Ｄａｔａ Ｍｏｄｅｌ Ｓｃｈｅｍａ（別名、概念空間）１１０２：ＥＤＭ空間は、開発者によって、データをモデル化するのに使用される。上述のように、データモデルの指定は、ＥＤＭ型、アソシエーションを介するエンティティ間のリレーション、継承などに関して行われる。

【０１８２】

３．データベーススキーマ（別名、ストレージ空間）１１０３：この空間では、開発者は、テーブルがどのようにレイアウトされるかを指定し、外部キーおよび主キーの制約などの制約もここで指定される。この空間における指定は、ベンダ固有の特徴、例えばネストされたテーブル、ＵＤＴなどを利用することができる。

【０１８３】

４．オブジェクト－ＥＤＭマッピング１１０４：このマッピングは、様々なオブジェクトおよびＥＤＭエンティティが互いにどのように関係するかを指定し、例えば、配列を１対多のアソシエーションにマッピングすることができる。このマッピングが自明（ｔｒｉｖｉａｌ）／恒等（ｉｄｅｎｔｉｔｙ）であることは必須ではなく、例えば、複数のクラスを所与のＥＤＭ型にマッピングすることができ、逆も可能であることに留意されたい。これらのマッピングにおいて冗長性／非正規化を有しても有しなくてもよいことに留意されたい（当然、非正規化を用いると、オブジェクト／エンティティの一貫性を保つという問題に突き当たる可能性がある）。

【０１８４】
５．ＥＤＭ－ストアマッピング１１０５：このマッピングは、ＥＤＭエンティティおよびＥＤＭ型がデータベース内の異なるテーブルにどのように関係するかを指定し、例えば、異なる継承マッピング戦略をここで使用することができる。
【０１８５】
開発者は、空間１１０１、１１０２、または１１０３の１つまたは複数、およびそれらの間の１つまたは複数のマッピングの間の対応するマッピングを指定することができる。いずれかのデータ空間が欠けている場合、開発者は、その空間をどのように生成すべきかに関するヒントを与えるか、対応する規定のマッピングを用いてＥＤＰがそれらの空間を自動的に生成すると期待することができる。例えば、開発者が、既存のクラス、テーブル、およびそれらの間のマッピングを指定する場合、ＥＤＰは、内部ＥＤＭスキーマ、ならびに対応するオブジェクト－ＥＤＭマッピングおよびＥＤＭ－ストアマッピングを生成する。当然、最も一般的な事例では、開発者は、完全な制御を有し、２つのマッピングとともにこれらの３つの空間内でデータモデルを指定することができる。下記の表は、ＥＤＰでサポートされる異なるシナリオを示す。これは、開発者がオブジェクト、ＥＤＭエンティティ、テーブルを指定できる事例、または指定できない事例の網羅的リストである。
【０１８６】
【表１０】

シナリオ	オブジェクトを指定できるか?	ＣＤＭを指定できるか?	テーブルを指定できるか?	マッピングを指定できるか?
(A)	Y			
(B)		Y		
(C)			Y	
(D)	Y	Y		OE
(E)	Y		Y	OS
(F)		Y	Y	ES
(G)	Y	Y	Y	OE、ES

【０１８７】
上記のＥＤＰがサポートを望むシナリオに応じて、（規定の手法で、またはヒントが提供される場合はそのヒントに基づいて）指定されていないデータ空間およびマッピングを作成するツールを提供しなければならない。内部ＯＲマッピングエンジンは、マッピングの５つの部分の全て（オブジェクト、ＥＤＭ仕様、テーブル、ＯＥマッピング、ＥＳマッピング）が使用可能であると仮定する。したがって、マッピング設計は、最も一般的な事例、すなわち上記の表の（Ｇ）をサポートすべきである。
【０１８８】
＜マッピング仕様言語＞
開発者の観点からのＯＲマッパーの「可視」部分の１つは、マッピング仕様言語、すなわちＭＳＬであり、開発者は、この言語を使用して、マッピングの様々な部分を互いにどのように結び付けられるかを指定する。ランタイムコントロール（例えば、遅延フェッチ、オプティミスティック並行性制御の問題）も、ＭＳＬを使用して指定される。
【０１８９】
マッピングを３つの異なる概念に分割する。各概念は、マッピングプロセスの異なる懸念事項に対処する。これらの仕様を単一ファイルに格納するのか、複数ファイルに格納するのか、あるいは外部リポジトリ（例えば、データ仕様のための）を介して指定するのかについて言及しないことに留意されたい。
【０１９０】

１．データ仕様：この領域では、開発者は、クラス記述、テーブル記述、およびＥＤＭ記述を指定することができる。これらの記述を、生成目的の仕様として提供することがあり、あるいは既に存在するテーブル／オブジェクトの仕様とすることができるであろう。

【０１９１】

オブジェクトおよびテーブルの仕様を、我々のフォーマットで記述することができ、あるいは、何らかのインポートツールを使用して外部メタデータリポジトリからインポートすることができる。

【０１９２】

サーバ生成値、制約、主キーなどの指定が、このセクションで行われる（すなわち、ＥＤＭ仕様では、制約は型指定の一部として指定される）ことに留意されたい。

【０１９３】

２．マッピング仕様：開発者は、様々なオブジェクト、ＥＤＭ型、およびテーブルのマッピングを指定する。開発者は、オブジェクト－ＥＤＭマッピング、ＥＤＭ－ストアマッピング、およびオブジェクト－ストアマッピングを指定することが可能である。このセクションは、データ仕様との冗長性を最小限にすることを試みる。

【０１９４】

３つのマッピングケース（ＯＳ、ＥＳ、およびＯＥ）の全てにおいて、各クラスのマッピングを、トップレベルで「直接に」または別のクラスの内部で「間接に」のいずれかで指定する。各マッピングでは、フィールド／プロパティが、別のフィールド、フィールドのスカラ関数、コンポーネント、またはセットにマッピングされる。更新を可能にするために、これらのマッピングは双方向である必要がある。すなわち、オブジェクトからストア空間へおよびその逆に進むことによって、どの情報も失われてはならない。オブジェクトが読み取り専用になるように、非双方向マッピングも許容することができる。

オブジェクト－ＥＤＭマッピング：一実施形態において、全てのオブジェクトのマッピングをＥＤＭ型に関して指定する。

ＥＤＭ－ストアマッピング：一実施形態において、全てのエンティティのマッピングをテーブルに関して指定する。

オブジェクト－ストアマッピング：一実施形態において、全てのオブジェクトのマッピングをテーブルに関して指定する。

【０１９５】

３．ランタイム仕様：一実施形態において、開発者は、実行を制御する様々なノブ、例えば、オプティミスティック並行性制御パラメータ、およびフェッチ戦略を指定することが可能である。

【０１９６】

ここに、ＯＰｅｒｓｏｎオブジェクトがアドレスのセットを含む事例のマッピングファイルの例がある。このオブジェクトは、ＥＤＭ　Ｅｎｔｉｔｙ型にマッピングされ、セットは、インラインセット型にマッピングされる。データは、２つのテーブル、すなわち、一方は人（ｐｅｒｓｏｎ）用のテーブル、および他方はアドレス用のテーブルに格納される。前述したように、開発者が全てのオブジェクト、ＥＤＭ型、およびテーブルを指定することは必須ではなく、上記の表のケース（Ｇ）を示しているに過ぎない。仕様は、特定の構文を記述することは想定されておらず、この仕様は、本明細書で開示される概念を中心とするシステムの設計を説明し、可能にすることを意図されたものである。

【０１９７】

・オブジェクト仕様

【０１９８】

【表１１】

ObjectSpec OPerson {
string name;
Set<Address> addrs;
}
	ObjectSpec OAddress {
	string state;
	}

【０１９９】

・ＥＤＭ仕様

各ＣＰｅｒｓｏｎがＣＡｄｄｒｅｓｓアイテムのコレクションを有するように、１つのエンティティ型ＣＰｅｒｓｏｎ、およびインライン型ＣＡｄｄｒｅｓｓを指定する。

【０２００】

【表１２】

EDMSpec Entity CPerson {
string name;
int pid;
Set<CAddress> addrs;
Key: {pid}
}
	EDMSpec InlineType CAddress {
	string state;
	int aid;
	}

【０２０１】

・ストア仕様

２つのテーブル型、ＳＰｅｒｓｏｎおよびＳＡｄｄｒｅｓｓを、それらのキー（ｔｐｉｄおよびｔａｉｄ）とともに指定する。

【０２０２】

【表１３】

TableSpec SPerson {
int pid;
nvarchar(10) name;
Key: {pid}
}
	TableSpec SAddress {
	int aid;
	string state;
	Key: {aid}
	}

【０２０３】

・オブジェクト－ＣＤＭマッピング

ＯＰｅｒｓｏｎの後のマッピングは、オブジェクト型ＯＰｅｒｓｏｎがＥｎｔｉｔｙＣＰｅｒｓｏｎにマッピングされることを述べるものである。その後のリストは、ＯＰｅｒｓｏｎの各フィールドがどのようにマッピングされるかを指定し、ｎａｍｅは名前にマッピングされ、ａｄｄｒｓコレクションはアドレスコレクションにマッピングされる。

【０２０４】

【表１４】

Object-CDM OPerson {
EntitySpec = CPerson
name ↔ name
addrs ↔ addrs
}
}
	Object-CDM OAddress {
	InlineTypeSpec = CAddress
	state ↔ state
	}

【０２０５】

・ＥＤＭ－ストアマッピング

ＥＤＭエンティティ型、ＣＰｅｒｓｏｎは、テーブル型ＳＰｅｒｓｏｎに、そのキー属性および名前ｃｎａｍｅ属性でマッピングされる。ＩｎｌｉｎｅＴｙｐｅ　ＣＡｄｄｒｅｓｓは、単純な手法でＳＡｄｄｒｅｓｓにマッピングされる。テーブルＳＡｄｄｒｅｓｓは、外部キーをＳＰｅｒｓｏｎに格納する場合があることに留意されたい。この制約は、マッピングではなく、テーブルのデータモデル仕様において指定されている可能性がある

。

【０２０６】

【表１５】

EDM-Store CPerson {
TableSpec =
SPerson
name ↔ name
pid ↔ pid
}
	EDM-Store CAddress {
	TableSpec = SAddress
	aid ↔ aid
	state ↔ state
	}
		EDM-Store CPerson_Address {
		TableSpec = SAddress
		aid ↔ aid
		pid ↔ pid
		}

【０２０７】

・ランタイム仕様

開発者は、ＯＰｅｒｓｏｎに対するオプティミスティック並行性制御を、ｐｉｄフィールドおよびｎａｍｅフィールドに対して行うことを指定することを望む場合がある。ＯＡｄｄｒｅｓｓについて、開発者は、ｓｔａｔｅフィールドに対する並行性制御を指定することができる。

【０２０８】

【表１６】

RuntimeSpec OPerson {
Concurrency fields: {pid, name}
}
	RuntimeSpec OAddress
	Concurrency fields: {state}
	}

【０２０９】

＜マッピング設計の概要＞

ＨｉｂｅｒｎａｔｅおよびＯｂｊｅｃｔＳｐａｃｅｓなどの、ほとんどのＯＲマッピング技術は、重要な短所を有する。すなわち、これらの技術は、比較的アドホックな手法で更新を処理する。オブジェクト変更をサーバにプッシュバックする必要があるとき、これらのシステムによって使用されるメカニズムは、ケースバイケースの原則で更新を扱い、これによりステムの拡張性が制限される。より多くのマッピングケースがサポートされるにつれて、更新パイプラインは、より複雑になり、更新のためにマッピングを構成することが難しくなる。システムが進化するにつれて、システムのこの部分は、正しいことを保証しながら変更することが非常に面倒になる。

【０２１０】

そのような問題を避けるために、２つタイプの「マッピングビュー」を使用してマッピングプロセスを実行する、新規の手法を使用する。このマッピングビューの一方は、クエリを変換するのに役立ち、他方は更新を変換するのに役立つ。図１２に示されるように、ＭＳＬ仕様１２０１がＥＤＰによって処理されると、ＥＤＰは、コアマッピングエンジンの実行のために、内部的に２つのビュー１２０２および１２０３を生成する。後でわかるように、これらのビューに関してマッピングをモデル化することによって、リレーショナルデータベースにおけるマテリアライズビュー技術に関する既存の知識を活用することができる。特に、正しく的確で拡張可能な手法で更新をモデル化するために、増分ビューの保守技法を利用する。次に、マッピングビューのこれら２つのタイプについて論じる。

【０２１１】

クエリマッピングビュー（Ｑｕｅｒｙ Ｍａｐｐｉｎｇ Ｖｉｅｗｓ）、すなわちＱＭＶｉｅｗという考えを使用して、テーブルデータをオブジェクトにマッピングし、更新マッピングビュー（Ｕｐｄａｔｅ Ｍａｐｐｉｎｇ Ｖｉｅｗｓ）、すなわちＵＭＶｉｅｗという考えを使用して、オブジェクト変更をテーブル更新にマッピングする。これらのビューは、これらが構築される（主な）理由ゆえに命名される。クエリビューは、オブジェクトクエリをリレーショナルクエリに変換し、入ってくるリレーショナルタプルをオブジェクトに変換する。したがって、ＥＤＭ－ストアマッピングについて、各ＱＶｉｅｗは、

ＥＤＭ型が様々なテーブルからどのように構築されるかを示す。例えば、Ｐｅｒｓｏｎエンティティが、２つのテーブルＴ＿ＰおよびＴ＿Ａの結合から構築される場合、Ｐｅｒｓｏｎをこの２つのテーブルの間の結合に関して指定する。クエリが、Ｐｅｒｓｏｎコレクションにおいて要求されると、ＰｅｒｓｏｎのＱＭＶｉｅｗは、ＰｅｒｓｏｎをＴ＿ＰおよびＴ＿Ａに関する式に置換する。次いで、この式が適切なＳＱＬを生成する。次いで、このクエリがデータベースで実行され、応答をサーバから受信すると、ＱＭＶｉｅｗが、返されたタプルからオブジェクトをマテリアライズする。

【０２１２】

オブジェクト更新を処理するために、ＱＭＶｉｅｗを介して変更をプッシュし、リレーショナルデータベース用に開発された「ビュー更新」技術を活用することを想像することができる。しかし、更新可能なビューは、それらに対する複数の制限があり、例えば、ＳＱＬ　Ｓｅｒｖｅｒは、１つのビュー更新を通じて複数のベーステーブルを変更することを許容しない。したがって、ＥＤＰで許容されるマッピングのタイプを制限する代わりに、本発明の実施形態は、制約がはるかに少ないマテリアライズビュー技術の別の態様、すなわちビュー保守を活用する。

【０２１３】

システム内の各テーブルをＥＤＭ型に関して表すために、更新マッピングビュー、ＵＭＶｉｅｗを指定する。すなわち、ある意味で、ＵＭＶｉｅｗはＱＭＶｉｅｗの逆である。ＥＤＭ－ストアの境界上のテーブル型のＵＭＶｉｅｗは、異なるＥＤＭ型を使用してそのテーブル型の列を構築する方法を表す。したがって、Ｐｅｒｓｏｎオブジェクト型をテーブル型Ｔ＿ＰおよびＴ＿Ａにマッピングすることを指定した場合、Ｔ＿ＰおよびＴ＿Ａに関してＰｅｒｓｏｎ型のＱＭＶｉｅｗを生成するだけではなく、Ｐｅｒｓｏｎオブジェクト型を与えられるとＴ＿Ｐの行をどのように構築できるかを指定する、ＵＭＶｉｅｗも生成する（Ｔ＿Ａについても同様である）。あるトランザクションが、いくつかのＰｅｒｓｏｎオブジェクトを作成、削除、または更新する場合、更新ビューを使用して、そのような変更を、オブジェクトから、Ｔ＿ＰおよびＴ＿Ａに対するＳＱＬの挿入ステートメント、更新ステートメント、および削除ステートメントに変換することができる。ＵＭＶｉｅｗにより、リレーショナルタプルがどのようにオブジェクトから（ＣＤＭ型を介して）取得されるかがわかるため、ＵＭＶｉｅｗは、これらの更新を実行する際に役立つ。図１３および１４は、高水準で、ＱＭＶｉｅｗおよびＵＭＶｉｅｗがクエリ変換および更新変換でどのように使用されるかを示す。

【０２１４】

オブジェクトに対するビューとしてテーブルをモデル化する、このアプローチが与えられると、オブジェクトに対する更新を元のテーブルに伝搬するプロセスは、オブジェクトが「ベースリレーション」でありテーブルが「ビュー」である、ビュー保守の問題に類似する。ビュー保守の問題に対処する膨大な量のデータベース文献があり、これを我々の目的のために活用することができる。例えば、ベースリレーションに対する増分変更を、ビューに対する増分変更にどのように変換できるかについて示す、重要な多数の研究がある。代数的アプローチを使用して、ビューに対する増分更新を実行するのに必要な式を判定する。この式をデルタ式と呼ぶ。増分ビューの保守に、手続き的アプローチではなく、代数的アプローチを使用することが適切であるが、これは、このアプローチが最適化および更新の単純化に対してより修正可能なものであるからである。

【０２１５】

一般に、ＥＤＰのコアエンジンでマッピングビューを使用することの利点には、次のものが含まれる。

１．ビューは、オブジェクトとリレーションとの間のマップを表すための相当な能力（ｐｏｗｅｒ）および柔軟性を提供する。ＯＲマッピングエンジンのコア部分内の制限されたビュー式の言語から始めることができる。時間およびリソースが許す限り、ビューの能力を使用して、システムを体裁よく進歩させることができる。

２．ビューは、クエリ、更新、およびビュー自体で非常に的確に構成されることが知ら

れている。構成可能性、特に更新に関する更新可能性は、以前に試みられたＯＲマッピングアプローチの一部に関し、問題のある争点であった。ビューベースのテクノロジを採用することによって、そのような懸念事項を回避することができる。

【０２１６】

ビューの考えを使用することによって、データベース文献の重要な多数の研究を活用できるようになる。

【０２１７】

＜更新に関するアーキテクチャ的階層化＞

本発明の諸態様の実装において考慮すべき重要な問題は、クエリマッピングビューおよび更新マッピングビューを表すマッピングビュー言語（ＭＶＬ）の能力が何であるかである。これは、ＥＤＭとストアとの間のマッピングとともにオブジェクトとＥＤＭとの間の非規範的マッピング（ｎｏｎ－ｐｒｅｓｃｒｉｐｔｉｖｅ　ｍａｐｐｉｎｇ）の全てをキャプチャするのに、ほぼ十分に強力である。しかし、全ての非リレーショナルＣＬＲおよびＥＤＭの概念をネイティブにサポートするＭＶＬについて、全てのそのような構造についてデルタ式または増分ビューの更新ルールを設計する必要がある。特に、一例示的実施形態は、次の非リレーショナル代数の演算子／概念に関する更新ルールを必要とすることがある。

複合型－オブジェクト、タプルコンストラクタ、フラット化、複合定数などの部分にアクセスする。

コレクション－ネスト化およびアンネスト化、セット構築／フラット化、相互適用（ｃｒｏｓｓ　ａｐｐｌｙ）など。

配列／リスト－要素の順序付けは、リレーショナル構造ではなく、明らかに、順序付けリストの代数は、非常に複雑である。

【０２１８】

モデル化される必要がある、ＣＬＲ／Ｃ＃の他のＥＤＭ構造およびオブジェクト構造。

【０２１９】

これらの構造に関する増分更新のためのデルタ式を開発することできる。ＭＶＬでネイティブに構造の大きなセットをサポートすることに関連する主な問題は、これがコアエンジンをかなり複雑にする可能性があることである。一実施形態において、より望ましいアプローチは、「コアマッピングエンジン」が単純なＭＶＬを処理するようにシステムを階層化し、このコアの上に非リレーショナル構造を階層化することである可能性がある。そのような設計について次に論じる。

【０２２０】

ＯＲマッピングに関するアプローチは、「階層化」によって上記の問題に対処する。コンパイル時に、まず、オブジェクト空間、ＥＤＭ空間、およびデータベース空間内の各非リレーショナル構造（ＷｉｎＦＳは、ネスティング、ＵＤＴなどをサポートする）を、対応するリレーショナル構造に所定の手法で変換し、次いで、リレーショナル構造の間で要求される非規定の変換を実行する。このアプローチを、階層化ビューマッピングアプローチと称する。例えば、クラスＣＰｅｒｓｏｎがアドレスのコレクションを有する場合、まず、このコレクションを１対多のアソシエーションとしてリレーショナル構造に変換し、次に、要求される非規定の変換をテーブルに対してリレーショナル空間で実行する。

【０２２１】

（ＭＶＬ分解）

ＭＶＬは、２つのレイヤ、すなわち、リレーショナル項（ｔｅｒｍ）内での実際の非規範的マッピングを扱うレイヤと、リレーショナル項への非リレーショナル構造の規範的変換を扱うレイヤに分解される。前者の言語を、Ｒ－ＭＶＬ（リレーショナル－ＭＶＬ）と称し、対応するマッピングをＲ－ＭＶＬマッピングと称する。同様に、後者の（より強力な）言語を、Ｎ－ＭＶＬ（非リレーショナル－ＭＶＬ）と称し、マッピングを、Ｎ－ＭＶＬマッピングと称する。

【０２２２】

一実施形態において、マッピングは、全ての非リレーショナル構造が、クエリパイプラインおよび更新パイプラインの終りにプッシュされるように、設計を構造化することによって提供される。例えば、オブジェクトマテリアライゼーションは、オブジェクト、配列、ポインタなどを構築することを含む場合があり、そのような「演算子」は、キューパイプラインの上にプッシュされる。同様に、更新がオブジェクトに対して行われるとき、パイプラインの始めで非リレーショナルオブジェクト（例えば、ネストされたコレクション、配列）に対する変更を変換し、その後、これらの変更を、更新パイプラインを通じて伝搬する。ＷｉｎＦＳなどのシステムでは、更新パイプラインの終りでＵＤＴに変換する必要がある。

【０２２３】

非規定のマッピングをＲ－ＭＶＬに制限することによって、増分ビューの保守ルールに関するリレーショナル構造の小さいセットが存在する。そのようなルールは、リレーショナルデータベース用に既に開発されている。Ｒ－ＭＶＬで許容される単純化された構造／スキーマを、Ｒｅｌａｔｉｏｎａｌｌｙ－Ｅｘｐｒｅｓｓｅｄ　Ｓｃｈｅｍａ、すなわちＲＥＳと称する。したがって、一部の非リレーショナル構造が、（例えば）オブジェクトドメイン内でサポートされる必要があるとき、対応するＲＥＳ構造、ならびにオブジェクトとＲＥＳ構造との間の規定の変換を見つけ出し、例えば、ＲＥＳ空間内でオブジェクトコレクションを１対多のアソシエーションに変換する。さらに、非リレーショナル構造Ｎに対する更新を伝搬するために、Ｎからの挿入、削除、および更新をＮの対応するＲＥＳ構造に変換するデルタ式を見つけ出す。これらのデルタ式は、設計時に規定され、生成されており、例えば、コレクションに対する変更を１対多のアソシエーションにどのようにプッシュすべきか分かることに留意されたい。実際の非規定のマッピングに関するデルタ式は、リレーショナルデータベースに関する増分ビューの保守ルールを使用して自動的に生成される。この階層化方法論は、多量の非リレーショナル構造に関する一般化された増分ビューの保守ルールを見つけ出すという要件を取り除くだけではなく、内部更新パイプラインをも単純化する。

【０２２４】

階層化マッピングアプローチが、通知パイプラインにおける利点と同様の利点も有することに留意されたい。タプルに対する変更をサーバから受信すると、それらをオブジェクトに対する増分変更に変換する必要がある。これは、これらの変更を伝搬するためにクエリマッピングビューを使用する必要があること、すなわち、ＱＭＶｉｅｗのデルタ式を生成すること、を除いて更新パイプラインと同一の要件である。

【０２２５】

更新および通知パイプラインを単純化することとは別に、ＭＶＬの階層化は、重要な利点を有する。すなわち、「上位言語」（オブジェクト、ＥＤＭ、データベース）が、コアマッピングエンジンに重大な影響を与えることなく発展することを可能にする。例えば、新しい概念がＥＤＭに追加される場合、行う必要があることは、それをその構造の対応するＲＥＳに変換する規定の方法を見つけ出すことだけである。同様に、非リレーショナル概念がＳＱＬ　Ｓｅｒｖｅｒ内に存在する（例えば、ＵＤＴ、ネスティング）場合、これらの構造を規定の手法でＭＶＬに変換して、ＭＶＬおよびコアエンジンに対する影響を最小限にすることができる。ＲＥＳ－ストアとストアテーブルとの間の変換は、必ずしも恒等変換（ｉｄｅｎｔｉｔｙ　ｔｒａｎｓｌａｔｉｏｎ）ではないことに留意されたい。例えば、ＵＤＴ、ネスティングなどをサポートするバックエンドシステム（ＷｉｎＦＳバックエンドなど）において、この変換は、規定のオブジェクトリレーションに類似する。

【０２２６】

図１５は、マッピングビューのコンパイル時およびランタイムの処理を図示する。１５０１、１５０２、および１５０３によって示されるように、ＭＳＬにおいてデータモデルおよびマッピング仕様が与えられると、まず、非リレーショナル構造１５１１、１５１２、および１５１３に対応するＲＥＳ　１５２１、１５２２、および１５２３と、これらの構造とＲＥＳとの間の規定の変換、すなわちＮ－ＭＶＬマッピングとを生成する。次いで

、開発者によって要求される非規定のマッピングについて、クエリマッピングビューおよび更新マッピングビューと、Ｒ－ＭＶＬのオブジェクト－ＥＤＭと、Ｒ－ＭＶＬのＥＤＭ－ストアとを生成する。これらのマッピングビューは、Ｒ－ＭＶＬ言語を使用してＲＥＳで動作することに留意されたい。このポイントで、クエリマッピングビューおよび更新マッピングビューのデルタ式（ビュー保守式）を生成し、そのようなルールは、リレーショナル構造用に開発済みである。ＱＭＶｉｅｗのデルタ式が、通知のために必要であることに留意されたい。Ｎ－ＭＶＬマッピングについて、デルタ式は設計時に決定される。これは、これらのマッピングが規定されており、例えば、Ａｄｄｒｅｓｓコレクションを１対多のアソシエーションにマッピングするときに、対応するビュー保守式も設計するからである。

【０２２７】

上記のビューおよび変換（Ｎ－ＭＶＬおよびＲ－ＭＶＬ）が与えられると、これらを構成して、ストア１５３３内のテーブルに関してオブジェクト１５３１を表すことができるクエリマッピングビューと、オブジェクト１５３１に関してストア１５３３を表すことができる更新マッピングビューとを得ることができる。図に示されているように、１５３２内のＥＤＭエンティティがランタイム用のマッピングから完全に除去されないようにマッピングビューを保つことを選択することができ、これらのビューを保持するためには、ＥＤＭ制約を利用する或る種のクエリ最適化を可能にすることが必要である。もちろん、このことは、ランタイムに実際にＥＤＭエンティティを格納することを意味するのではない。

【０２２８】

図１６は、様々なコンポーネントが上述のビューコンパイル処理を達成する方法を示す。アプリケーションが、ＡＰＩ　１６００を呼び出す。ビュージェネレータ１６０１、１６０３は、３つの機能に関与する。すなわち、非リレーショナル構造をＲＥＳ構造に変換すること、クエリ／更新ビューを生成すること、更新および通知を伝搬するためのデルタ式を生成すること、という３つの機能に関与する。これらは、これらの機能を実行する際にメタデータ１６０２を使用することができる。ＯＥビューコンポーザ１６０５は、オブジェクトおよびＥＤＭ情報を取り込み、ＥＤＭ型に関するオブジェクトの代数式となるようにそれを構成する。同様に、ＥＳビューコンポーザ１６０６は、テーブルに関するＥＤＭ型の代数式を作る。これらのビューを、さらにＯＳビューコンポーザ１６０７において構成して、メタデータストア１６０８内のビューの単一のセットを得る。上述したように、クエリ最適化の機会のために、ビューの２つのセットを保持することができる。最後に、依存性分析コンポーネント１６０４も、ＥＳビュージェネレータの出力に対して動作して、依存性順序をメタデータストア１６０８に提供することができる。

【０２２９】

（マップコンパイルの要約）

要約すると、クラス、ＥＤＭ型、またはテーブルの仕様Ｍごとに、対応するＲＥＳ、および、Ｍと対応するＲＥＳとの間の規定の変換を生成する。したがって、図１５に図示されるように、次を生成する。

１．Ｍに対応するＲＥＳ　ＲＥＳ－ＣＤＭ（Ｍ）、ＲＥＳ－Ｏｂｊｅｃｔ（Ｍ）、またはＲＥＳ－Ｓｔｏｒｅ（Ｍ）と表される。

２．各仕様ＭをＲＥＳリレーションに関して表すための規定の変換。

３．上記ＲＥＳリレーションをＭに関して表すための規定の変換。

４．クエリマッピングビュー：ＥＤＭ型に関してオブジェクトを表すＯＥ　ＱＭＶｉｅｗ、ストア（テーブル）に関してＥＤＭ型を表すＥＳ　ＱＭＶｉｅｗの２つのビューがある。

５．更新マッピングビュー：オブジェクトに関してＥＤＭ型を表すＯＥ　ＵＭＶｉｅｗ、ＥＤＭ型に関してストアテーブルを表すＥＳ　ＵＭＶｉｅｗの２つのビューがある。

６．更新の増分保守のために、ＵＭＶｉｅｗに対するデルタ式も生成する。

【０２３０】

これらのビューを構成した後に、４つのマップで終わる。これらのマップは、メタデータストア１６０８に格納され、集合的にコンパイル済みマッピングビューと称される。

クエリマップ：ＣＤＭ／テーブルに関してオブジェクト／ＣＤＭを表す。

更新マップ：ＣＤＭ／オブジェクトに関してテーブル／ＣＤＭを表す。

更新デルタ式：ＣＤＭ／オブジェクトのデルタに関してテーブル／ＣＤＭのデルタを表す。

通知デルタ式：ＣＤＭ／テーブルのデルタに関してオブジェクト／ＣＤＭのデルタを表す。

依存性順序：様々な挿入、削除、更新のオペレーションを異なるリレーションに対して実行しなければならない順序。この順序は、更新プロセス中にデータベース制約に違反しないことを保証する。

【０２３１】

（コレクションの例）

ここで、検討してきたＰｅｒｓｏｎの例に関する規定の変換および非規定の変換のマッピングについて、簡潔に示す。クエリマッピングビューと更新マッピングビューとの両方を提示し、対応するビュー保守式については、下記でさらに検討する。

【０２３２】

（ＲＥＳ）

ＯＰｅｒｓｏｎを、ＲＥＳ構造のＲ＿ＯＰｅｒｓｏｎに変換し、Ｒ＿ＯＰｅｒｓｏｎは、ｎａｍｅおよびｐｉｄを単純に反映する。同様に、ＯＡｄｄｒｅｓｓを、Ｒ＿ＯＡｄｄｒｅｓｓに変換する。アドレスのコレクションを変換するために、１対多のアソシエーション、Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓを使用する。ＥＤＭ構造についても同様である。テーブル（Ｒ＿ＳＰｅｒｓｏｎ、Ｒ＿ＳＡｄｄｒｅｓｓ）のＲＥＳは、ＳＰｅｒｓｏｎおよびＳＡｄｄｒｅｓｓへの恒等マッピングである。これらのＲＥＳは、次の通りである。

【０２３３】

【表１７】

R_OPerson (pid, name) R_OAddress (aid, state) R_OPerson_Address (pid, aid)	R_CPerson (pid, name) R_CAddress (aid, state) R_CPerson_Address (pid, aid)	R_SPerson(pid, name) R_SAddress(pid, aid, state)

【０２３４】

（クエリマッピングビュー）

オブジェクトーストアマッピング（オブジェクトーＥＤＭマッピングおよびＥＤＭーストアマッピングにまたがって構成される）を示す。

【０２３５】

・ＲＥＳ空間における非規定ビュー

オブジェクトとＥＤＭ空間との間のマッピングは、本質的に恒等である。３つのビュー、Ｒ＿ＣＰｅｒｓｏｎ、Ｒ＿ＣＡｄｄｒｅｓｓ、およびＲ＿ＣＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓの全てが、Ｒ＿ＳＰｅｒｓｏｎおよびＲ＿ＳＡｄｄｒｅｓｓに対する単純な射影である。

【０２３６】

【表１８】

CREATE VIEW R_OPerson (pid, name) AS SELECT pid, name FROM R_CPerson	CREATE VIEW R_OPerson_Address (pid, aid) AS SELECT pid, aid FROM R_CPerson_Address	CREATE VIEW R_OAddress (aid, state) AS SELECT aid, state FROM R_CAddress

【０２３７】

【表１９】

CREATE VIEW R_CPerson (pid, name) AS SELECT pid, name FROM R_SPerson	CREATE VIEW R_CPerson_Address (pid, aid) AS SELECT pid, aid FROM R_SAddress	CREATE VIEW R_CAddress (aid, state) AS SELECT aid, state FROM R_SAddress

【０２３８】

・規定変換（ＲＥＳ－オブジェクトに関するオブジェクト）

ＯＰｅｒｓｏｎオブジェクトは、Ｒ＿ＯＡｄｄｒｅｓｓとＲ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓの結合を行い、結果をネストすることによって、Ｒ＿ＯＰｅｒｓｏｎ、Ｒ＿ＯＡｄｄｒｅｓｓ、およびＲ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓを使用して表される。

【０２３９】

【表２０】

```
CREATE PRESCRIBED VIEW OPerson (pid, name, addrs) AS
SELECT pid, name, NEST(SELECT Address(a.aid, a.state)
                       FROM R_OAddress a, R_OPerson_Address pa
                       WHERE pa.pid = p.pid AND a.aid = pa.aid)
R_OPerson p
```

【０２４０】

・ＣＰｅｒｓｏｎの構成されたビュー

単純化の後の構成された式は、以下とすることができる（この例に関し、テーブルとそのＲＥＳ構造との間に恒等変換があることを想起されたい）。

【０２４１】

【表２１】

```
CREATE VIEW OPerson (pid, name, addrs) AS
SELECT pid, name, NEST(SELECT Address(a.aid, a.state)
                       FROM SAddress a
                       WHERE a.pid = p.pid)
SPerson p
```

【０２４２】

最終的なビューは、「直接」マッピングアプローチを使用することによってうられることが予想されるものを示す。ＲＥＳアプローチの１つの利点は、更新パイプラインに関するデルタ式をみるときや、クエリマッピングビューのデルタ式が必要な通知パイプラインにおいても現れる。

【０２４３】

（更新マッピングビュー）

・ＲＥＳ空間内の非規定のビュー

Ｒ＿ＳＰｅｒｓｏｎに関するＵＭＶｉｅｗは、単にＲ＿ＣＰｅｒｓｏｎに対する射影であり、Ｒ＿ＳＡｄｄｒｅｓｓは、Ｒ＿ＣＡｄｄｒｅｓｓを１対多のアソシエーションテーブル、Ｒ＿ＣＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓと結合することによって構築される。ＣＤＭとオブジェクト空間との間のマッピングは、恒等である。

【０２４４】

【表２２】

CREATE VIEW R_CPerson (pid, name) AS SELECT pid, name FROM R_OPerson	CREATE VIEW R_CPerson_Address (pid, aid) AS SELECT pid, aid FROM R_OPerson_Address	CREATE VIEW R_CAddress (aid, state) AS SELECT aid, state FROM R_OAddress

【０２４５】

【表２３】

CREATE VIEW R_SPerson (pid, name) AS SELECT pid, name, FROM R_CPerson	CREATE VIEW R_SAddress(aid, pid, state) AS SELECT aid, pid, state FROM R_CPerson_Address, R_CAddress WHERE R_CPerson_Address.aid = R_CAddress.aid

【０２４６】

・規定の変換（オブジェクトに関するＲＥＳ－オブジェクト）

オブジェクトをＲＥＳに変換し、その結果、更新をオブジェクト空間からＲＥＳ空間にプッシュできるようにする必要がある。Ｒ＿ＯＰｅｒｓｏｎの規定の変換は、単純な射影であるが、Ｒ＿ＯＡｄｄｒｅｓｓおよびＲ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓの変換は、人とそのアドレスとの間で結合を実行することによって達成される。これは、「ポインタ結合」または「ナビゲーション結合」である。

【０２４７】

【表２４】

CREATE PRESCRIBED VIEW R_OPerson (name, pid) AS SELECT name, pid FROM OPerson	CREATE PRESCRIBED VIEW R_OAddress(state, aid) AS SELECT a.state, a.aid FROM OPerson p, p.addrs a	CREATE PRESCRIBED VIEW R_OPerson_Address(pid, aid) AS SELECT pid, aid FROM OPerson p, p.addrs a

【０２４８】

・構成された更新マッピングビュー

上記のビューを構成して（およびいくつかの単純化を伴って）、次の構成された更新マッピングビューを得る。

【０２４９】

【表２５】

CREATE VIEW SPerson (pid, name) AS SELECT pid, name, FROM OPerson	CREATE VIEW SAddress(aid, pid, state) AS SELECT a.aid, p.pid, a.state FROM OPerson p, p.addrs a

【０２５０】

したがって、テーブルＳＰｅｒｓｏｎを、ＯＰｅｒｓｏｎに対する単純な射影として表すことができ、ＳＡｄｄｒｅｓｓは、ＯＰｅｒｓｏｎをそのアドレスと結合することによって取得される。

【０２５１】

（ビューの検証）

生成されたビューが満たす必要がある重要なプロパティは、それらのビューが「ラウンドトリップ」しなければならないことである。すなわち、情報の消失を全て防ぐために、エンティティ／オブジェクトが保存され、その後取り出される際に情報の消失がないことを保証しなければならない。言い換えると、全てのエンティティ／オブジェクトＤについて、次を保証することを望む。

D = QMView(UMView(D))

【０２５２】

ビュー生成アルゴリズムは、このプロパティを保証する。このプロパティが真である場合は、「クエリビューおよび更新ビューがラウンドトリップする」、または双方向であるとも言う。ここで、人－アドレスの例についてこのプロパティを実証する。単純にするために、ＲＥＳ空間におけるラウンドトリップに焦点を当てる。

【０２５３】

・Ｒ＿ＯＰｅｒｓｏｎに関する検証

ＯＰｅｒｓｏｎに関するクエリビューにおいてＳＰｅｒｓｏｎを置換することによって、次が得られる。

【０２５４】

【表２６】

```
R_OPerson(pid, name, age) =
SELECT pid, name, age FROM (SELECT pid, name, age FROM R_SPerson)
```

【０２５５】

簡約化して次を得る。

【０２５６】

【表２７】

```
R_OPerson(pid, name, age) = SELECT pid, name, age FROM R_SPerson
```

【０２５７】

これは、ＳＥＬＥＣＴ＊ＦＲＯＭ　Ｐｅｒｓｏｎと同等である。

【０２５８】

・ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓに関する検証

Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓについて、わずかに複雑である。次を有する。

【０２５９】

【表２８】

```
R_OPerson_Address (pid, aid) = SELECT pid, aid FROM R_SAddress
```

【０２６０】

Ｒ＿ＳＡｄｄｒｅｓｓについて置換することによって、次が得られる。

【０２６１】

【表２９】

```
R_OPerson_Address (pid, aid) =
SELECT pid, aid
FROM (SELECT aid, pid, state
        FROM R_OPerson_Address pa, R_OAddress a
        WHERE pa.aid = a.aid)
```

【０２６２】

これは、次のように簡約化される。

【０２６３】

【表３０】

```
R_OPerson_Address (pid, aid) =
SELECT pid, aid FROM R_OPerson_Address pa, R_OAddress a WHERE pa.aid =
a.aid
```

【０２６４】

上記が実際はＳＥＬＥＣＴ＊ＦＲＯＭ　Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓであることを示すために、外部キーの依存性、Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓ．ａｉｄ→Ｒ＿ＯＡｄｄｒｅｓｓ．ａｉｄを有することが必要である。この依存性が成り立たない場合は、ラウンドトリップすることができない。しかし、セット値付きプロパティａｄｄｒｓの範囲がＲ＿ＯＡｄｄｒｅｓｓなので、これは成り立つ。この外部キー制約は、次の２つの方法で表すことができる。

【０２６５】

【表３１】

1. $R\_OPerson\_Address.aid \subseteq R\_OAddress.aid$
2. $\pi_{aid,pid}(R\_OPerson\_Address \bowtie_{aid} R\_OAddress) = R\_OPerson\_Address$

【０２６６】
この制約を上の式において置換することによって、次が得られる。
【０２６７】
【表３２】

R_OPerson_Address (pid, aid) = SELECT pid, aid FROM R_OPerson_Address

【０２６８】
・アドレスに関する検証
Ｒ＿ＯＡｄｄｒｅｓｓは、次のように与えられる。
【０２６９】
【表３３】

R_OAddress (aid, state) = SELECT aid, state FROM R_SAddress

【０２７０】
Ｒ＿ＳＡｄｄｒｅｓｓについて置換することによって、次が得られる。
【０２７１】
【表３４】

```
R_OAddress(aid, state) =
SELECT aid, state
FROM (SELECT aid, pid, state
        FROM R_OPerson_Address pa, R_OAddress a
        WHERE pa.aid = a.aid)
```

【０２７２】
これを、次のように言い換えることができる。
【０２７３】
【表３５】

```
R_OAddress(aid, state) = SELECT aid, state FROM R_OPerson_Address pa,
R_OAddress a
                        WHERE pa.aid = a.aid
```

【０２７４】
ここで、Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓとの結合は、外部キー依存性、Ｒ＿ＯＡｄｄｒｅｓｓ．ａｉｄ→Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓ．ａｉｄが成り立つ場合は不必要である。この依存性は、Ｒ＿ＯＡｄｄｒｅｓｓが、実存的にＲ＿ＯＰｅｒｓｏｎに依存する（すなわち、アドレスが合成物である）場合にのみ成り立つ。そうでない場合には、ビューはラウンドトリップしない。したがって、次の制約がある。
【０２７５】
【表３６】

$\pi_{aid,\ state}(R\_OAddress \bowtie_{aid} R\_OPerson\_Address) = R\_OAddress$

【０２７６】
したがって、次の式が得られる。

【０２７７】
【表３７】
R_OAddress(aid, state) = SELECT aid, state FROM R_OAddress

【０２７８】
＜クエリ変換＞
（クエリトランスレータ）
ＥＤＰクエリトランスレータ（ＥＱＴ）は、マッピングメタデータを利用することによって、クエリをオブジェクト／ＥＤＭ空間からプロバイダ空間に変換することに関与する。ユーザクエリを、様々な構文、例えば、ｅＳＱＬ、Ｃ＃　Ｓｅｑｕｅｎｃｅ、ＶＢ　ＳＱＬなどで表すことができる。ＥＱＴアーキテクチャを図１７に示す。ここで、ＥＱＴの様々なコンポーネントを説明する。
【０２７９】
パーサ１７１１は、ｅＳＱＬ、ＬＩＮＱ（Ｌａｎｇｕａｇｅ　Ｉｎｔｅｇｒａｔｅｄ　Ｑｕｅｒｙ）、Ｃ＃　Ｓｅｑｕｅｎｃｅ、およびＶＢ　Ｓｑｌを含む複数の形式の１つで表されたユーザクエリを解析することによって、構文分析を実行する。全ての構文エラーが、この時に検出され、フラグを立てられる。
【０２８０】
ＬＩＮＱについて、構文分析（および意味分析）は、言語（Ｃ＃、ＶＢなど）自体の構文分析フェーズと統合される。ｅＳＱＬについて、構文分析フェーズは、クエリプロセッサの一部である。典型的に、言語ごとに１つの構文アナライザがある。
【０２８１】
構文分析フェーズの結果が、解析ツリーである。このツリーは、次いで意味分析フェーズ１７１２に供給される。
【０２８２】
パラメータバインダおよび意味分析コンポーネント１７１２は、ユーザクエリ内のパラメータを管理する。このモジュールは、クエリ内のパラメータのデータ型および値を追跡する。
【０２８３】
意味分析フェーズは、構文分析フェーズ１７１１によって作成された解析ツリーを意味的に検証する。クエリ内の全てのパラメータは、この時点で既にバインドされていなければならない。すなわち、そのデータ型が知られていなければならない。全ての意味エラーが、ここで検出され、フラグを立てられる。成功の場合、このフェーズの結果が意味ツリーである。
【０２８４】
ＬＩＮＱについて、前述したように、意味分析フェーズは、言語自体の意味分析フェーズと統合される。典型的には、言語ごとに１つの構文ツリーがあるので、言語ごとに１つのセマンティックアナライザがある。
【０２８５】
意味分析フェーズは、論理的には次のものから構成される。
１．名前解決：クエリ内の全ての名前が、この時点で解決される。これは、エクステント、型、型のプロパティ、型のメソッドなどへの参照を含む。副作用として、そのような式のデータ型も推論される。このサブフェーズは、メタデータコンポーネントと対話する。
２．型のチェックおよび推論：クエリ内の式が型をチェックされ、結果の型が推論される。
３．検証：他の種類の検証が、ここで行われる。例えば、ＳＱＬプロセッサ内で、クエリブロックがグループ化の節（ｇｒｏｕｐ－ｂｙ　ｃｌａｕｓｅ）を含む場合、このフェーズを使用して、選択リストがグループ化のキー（ｇｒｏｕｐ－ｂｙ　ｋｅｙｓ）または集約関数だけを参照できるという制約を実施することができる。

【０２８６】
意味分析フェーズの結果は、意味ツリーである。この時、クエリは有効とみなされ、さらなる意味エラーは、後のクエリコンパイル中のどの時にも発生してはならない。
【０２８７】
代数化フェーズ１７１３は、意味分析フェーズ１７１２の結果を取り込み、これを代数変形にとってより適した形式に変換する。このフェーズの結果は、論理拡張されたリレーショナル演算子ツリー、別名、代数ツリーである。
【０２８８】
代数ツリーは、コアリレーショナル代数演算子、すなわち、選択、射影、結合、合併に基づき、これをネスト／アンネスト、ピボット／アンピボットなどの追加演算で拡張する。
【０２８９】
クエリトランスレータのビューアンフォールディングフェーズ１７１４は、おそらくは再帰的に、ユーザクエリ内で参照される全てのオブジェクトに関するＱＭＶｉｅｗ式を置換する。ビュー変換プロセスの終りに、ストア項でクエリを記述するツリーを得る。
【０２９０】
オブジェクトレイヤの事例では、ビューアンフォールディングが、ストアスペースまで行われている可能性があり（メタデータリポジトリ内に格納された、最適化されたＯＳマッピングがある場合）、あるいは、クエリツリーが、ＥＤＭレイヤまで変形されている可能性がある。後者の事例では、このツリーを取り込み、ＥＤＭ概念をストア概念に変換するという要件を用いて、このツリーをビューアンフォールディングコンポーネントに再供給する必要がある。
【０２９１】
変形／単純化コンポーネント１７１５は、プロバイダ１７３０固有とすることができ、あるいは、代替的な実施形態では、様々なプロバイダによって活用できるＥＤＰ汎用コンポーネントとすることができる。クエリツリーに対する変形を実行する少数の理由がある。
【０２９２】
１．ストアにプッシュする演算子：ＥＱＴは、複合演算子（例えば、結合、フィルタ、集約）をストアにプッシュする。あるいは、そのような演算は、ＥＤＰで実装される必要がある。ＥＤＰの値マテリアライゼーションレイヤは、ネスティングなどの「非リレーショナル補償」演算だけを実行する。演算子Ｘをキューツリーの値マテリアライゼーションノードより下にプッシュすることができず、値マテリアライゼーションレイヤがオペレーションＸを実行できない場合は、そのクエリは不正であると宣言する。例えば、クエリが、プロバイダにプッシュできない集約演算である場合、値マテリアライゼーションレイヤはいかなる集約も実行することができないので、そのクエリを不正と宣言する。
【０２９３】
改善されたパフォーマンス：クエリの複雑さの軽減は重要であり、巨大なクエリをバックエンドストアに送信することを回避したい。例えば、ＷｉｎＦＳでの現在のクエリの一部は、非常に複雑であり、実行に長い時間を要する（対応する手書きクエリは、１桁以上高速である）。
【０２９４】
改善されたデバッグ可能性：より単純なクエリにより、開発者がシステムをデバッグし、何がおきているかを理解することがより容易になる。
【０２９５】
変形／単純化モジュール１７１５は、クエリを表す代数ツリーの一部または全てを同等のサブツリーに変形することができる。これらのヒューリスティックベースの変形が、論理的であること、すなわち、コストモデルを使用して行われるのではないことに留意されたい。論理的変形の種類には、次の例示的なプロバイダ固有サービスを含めることができる。

サブクエリフラット化（ビューおよびネストされたサブクエリ）
結合除去
述部の除去および統合
述部のプッシュダウン
共通のサブ式の除去
射影のプルーニング
外部結合→内部結合の変形
左相関の除去

【０２９６】

このＳＱＬ生成モジュール１７３１は、生成されるＳＱＬがプロバイダに固有なので、プロバイダコンポーネント１７３０の一部である。単純化の後に、代数ツリーが、プロバイダに渡され、このプロバイダは、適切なＳＱＬを生成する前に、プロバイダ固有の変形または単純化をさらに実行することができる。

【０２９７】

クエリがサーバで実行された後に、結果は、ＥＤＰクライアントにストリーミングされる。プロバイダ１７３０は、アプリケーションによって使用可能なＤａｔａＲｅａｄｅｒを公開して、結果をＥＤＭエンティティとして取得する。値マテリアライゼーションサービス１７４１は、これらのリーダーを取り込み、これらを関連するＥＤＭエンティティに変換することができる（新しいＤａｔａＲｅａｄｅｒとして）。これらのエンティティをアプリケーションによって消費することがあり、あるいは、新しいＤａｔａＲｅａｄｅｒを上のオブジェクトマテリアライゼーションサービスに渡すことができる。

【０２９８】

ＥＱＴ　１７００は、マテリアライゼーションをキューツリー内の演算子として表す。これは、通常のクエリ変換パイプラインが、ＥＤＭ空間内でオブジェクトを作ることを可能にし、このオブジェクトは、その後、実際のマテリアライゼーションを実行するために特殊な帯域外のオペレーションを必要とするのではなく、ユーザに直接に供給されることができる。これは、部分的オブジェクトフェッチ、イーガーローディング（ｅａｇｅｒ　ｌｏａｄｉｎｇ）などのような、様々な最適化をユーザクエリに対して実行することも可能にする。

【０２９９】

（クエリの例）

我々が展開していた、人－アドレスの例を検討されたい。ユーザが、次のクエリ（ＷＡ内の全ての人を見つける）を実行することを望むと仮定する。このクエリを擬似ＣＳＱＬで次のように記述することができる。

【０３００】

【表３８】

```
SELECT x.name FROM OPerson x, x.addrs y WHERE y.state = “WA”
```

【０３０１】

この時点でＰｅｒｓｏｎのクエリビューを使用してビューアンフォールディングを行うと、次が得られる。

【０３０２】

【表３９】

```
SELECT x.name
FROM (SELECT pid, name,
        NEST(SELECT OAddress(a.aid, a.state) FROM SAddress a where a.pid =
p.pid)
        FROM SPerson p) as x, x.addrs y
WHERE y.state = “WA”
```

【０３０３】
このクエリは、バックエンドサーバに送信する前に単純化することができる。
【０３０４】
【表４０】

```
SELECT p.name
FROM SPerson p, SAddress a
WHERE p.pid = a.pid
```

【０３０５】
（メタデータ）
ＥＱＴは、クエリのコンパイル中および実行中に様々なメタデータを必要とする。このメタデータは、次を含む。
【０３０６】
アプリケーション－空間メタデータ：ユーザクエリを検証するために意味分析中に必要な、エクステント／コレクション、型、型プロパティ、型メソッドに関する情報。
【０３０７】
スキーマ－空間メタデータ：ビューコンパイル中に必要なエンティティコレクション、ＣＤＭ型、およびプロパティに関する情報。変形に関するエンティティ間のリレーションシップおよびエンティティに対する制約についての情報。
ストレージ－空間メタデータ：上で説明したもの。
アプリケーション－＞スキーママッピング：ビュー拡張に必要なビュー定義を表す論理演算子ツリー。
スキーマ－＞ストレージマッピング：上で説明したもの。
【０３０８】
（エラー報告パイプライン）
クエリ処理の様々な段階でのエラーは、ユーザが理解できる用語で報告されるべきである。様々なコンパイル時および実行時のエラーが、クエリ処理中に発生することがある。構文分析中および意味分析中のエラーは、ほとんどがアプリケーション空間内であり、ほんのわずかな特殊処理を必要とする。変形中のエラーは、ほとんどがリソースエラー（メモリ不足など）であり、多少の特殊処理を必要とする。コード生成中および後続のクエリ実行中のエラーは、適切に処理されることが必要な可能性がある。エラー報告における主要な課題は、より低いレベルの抽象化で発生するランタイムエラーを、アプリケーションレベルの意味のあるエラーにマッピングすることである。これは、より低いレベルのエラーを、ストレージマッピング、概念マッピング、およびアプリケーションマッピングを介して処理することが必要であることを意味する。
【０３０９】
（クエリの例）
サンプルのＯＯクエリは、ワシントン州にアドレスを有する全ての人の名前をフェッチする。
【０３１０】
【表４１】

```
SELECT p.name
FROM OPerson p, p.addrs as a
WHERE a.state = 'WA'
```

【０３１１】
・ステップ１：リレーショナル項への変換
このクエリを、項またはＲ＿ＯＰｅｒｓｏｎ、Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓ、およびＲ＿ＯＡｄｄｒｅｓｓで表された次の純リレーショナルクエリに変換することができる。本質的に、必要な場合に様々なナビゲーションプロパティ（ドット「．」式）を結

合式に拡張しようとしている。

【０３１２】

【表４２】

```
SELECT p.name
FROM R_OPerson p, R_OPerson_Address pa, R_OAddress a
WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = 'WA'
```

【０３１３】

このクエリは、まだオブジェクトドメイン内であり、オブジェクトエクステントに関することに留意されたい。

【０３１４】

・ステップ２：ビューアンフォールディング：ストア空間への変換

ここで、クエリをＳＱＬに変換するためにビューアンフォールディングを行う。

【０３１５】

【表４３】

```
SELECT p.name
FROM (SELECT pid , name, age FROM SPerson) p,
       (SELECT pid , aid FROM SAddress) pa,
       (SELECT aid, state FROM SAddress) a
WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = 'WA'
```

【０３１６】

・ステップ３：クエリの単純化

ここで、一連の論理変形を適用して、このクエリを単純化する。

【０３１７】

【表４４】

```
SELECT p.name
FROM SPerson p, SAddress pa, SAddress a
WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = 'WA'
```

【０３１８】

次に、ＳＡｄｄｒｅｓｓの主キー（ａｉｄ）における余分な自己結合を除去し、次を得る。

【０３１９】

【表４５】

```
SELECT p.name
FROM SPerson p, SAddress a
WHERE p.pid = a.pid AND a.state = 'WA'
```

【０３２０】

上記全てが、かなり単純である。ここで、ＳＱＬ　Ｓｅｒｖｅｒに送信することができるクエリとなる。

【０３２１】

＜更新に関するコンパイル時の処理＞

ＥＤＰは、アプリケーションが新しいオブジェクトを作成し、更新し、削除し、これらの変更を永続的に格納することを可能にする。ＯＲマッピングコンポーネントは、これらの変更がバックエンドストア変更に正しく変換されることを保証する必要がある。前述したように、オブジェクトに関してテーブルを宣言する更新マッピングビューを使用する。そのようなビューを使用することによって、本質的に、更新の伝搬の問題が、ベースリレーションに対する変更がビューに伝搬される必要があるマテリアライズビュー保守の問題となった。ここでＵＭＶｉｅｗｓの場合は、「ベースリレーション」はオブジェクトであり、「ビュー」はテーブルである。この問題をこの手法でモデル化することによって、リレーショナルデータベースの世界で開発されてきたビュー保守技術の知識を活用することができる。

【０３２２】
（更新マッピングビューの生成）
クエリの場合と同様に、更新に関する多数のマッピング作業が、コンパイル時に実行される。クラス、ＥＤＭ型、およびテーブルのＲｅｌａｔｉｏｎａｌｌｙ Ｅｘｐｒｅｓｓｅｄ Ｓｃｈｅｍａとともに、これらの型と対応するＲＥＳ構造との間の規定の変換を生成する。また、クラスのＲＥＳ構造とＥＤＭ型との間、およびＥＤＭ型のＲＥＳ構造とストアテーブルとの間で更新マッピングビューも生成する。
【０３２３】
我々が展開していた、人－アドレスの例の助けを得て、これらのＵＭＶｉｅｗｓを理解しよう。構築されたオブジェクト（Ｒ＿ＯＰｅｒｓｏｎ、Ｒ＿ＯＡｄｄｒｅｓｓ、Ｒ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓ）に関するＲＥＳ制約を想起されたい。
【０３２４】
・更新マッピングビュー（オブジェクトのＲＥＳに関するテーブルのＲＥＳ）
Ｒ＿ＯＰｅｒｓｏｎのＵＭＶｉｅｗは、単にＲ＿ＳＰｅｒｓｏｎに対する射影であり、Ｒ＿ＳＡｄｄｒｅｓｓは、Ｒ＿ＯＡｄｄｒｅｓｓを、１対多のアソシエーションテーブル、すなわちＲ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓと結合することによって構築される。
【０３２５】
【表４６】

CREATE VIEW R_SPerson (pid, name) AS SELECT pid, name, FROM R_OPerson	CREATE VIEW R_SAddress(aid, pid, state) AS SELECT aid, pid, state FROM R_OPerson_Address pa, R_OAddress a WHERE pa.aid = a.aid

【０３２６】
・規定の変換（オブジェクトに関するＲＥＳ）
オブジェクトをＲＥＳに変換し、その結果、更新をオブジェクト空間からＲＥＳ空間にプッシュできるようにする必要がある。「ｏ２ｒ」関数を使用して、オブジェクトの仮想メモリアドレスをｐｉｄキーおよびａｉｄキーに変換する。この実装において、単純にオブジェクトのシャドウ状態からキーを得ることができる。Ｒ＿ＯＰｅｒｓｏｎに関する規定の変換は、単純な射影であるが、Ｒ＿ＯＡｄｄｒｅｓｓおよびＲ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓに関する変換は、人とそのアドレスとの間の結合を実行することによって達成される。
【０３２７】
【表４７】

CREATE PRESCRIBED VIEW R_OPerson (name, pid) AS SELECT name, pid FROM OPerson	CREATE PRESCRIBED VIEW R_OAddress(state, aid) AS SELECT a.state, a.aid FROM OPerson p, p.addrs a
CREATE PRESCRIBED VIEW R_OPerson_Address(pid, aid) AS SELECT p.pid, a.aid FROM OPerson p, p.addrs a	

【０３２８】
・構成された更新マッピングビュー
上記のビューを構成して（およびいくつかの単純化を伴って）、次の構成された更新マッピングビューを得る。
【０３２９】

【表４８】

CREATE VIEW SPerson (pid, name) AS SELECT pid, name FROM OPerson	CREATE VIEW SAddress(aid, pid, state) AS SELECT a.aid, p.pid, a.state FROM OPerson p, p.addrs a

【０３３０】

したがって、テーブルＳＰｅｒｓｏｎを、ＯＰｅｒｓｏｎに対する単純な射影として表すことができ、ＳＡｄｄｒｅｓｓは、ＯＰｅｒｓｏｎをそのアドレスと結合することによって取得される。

【０３３１】

（デルタ式の生成）

アプリケーションが、そのオブジェクト変更をバックエンドに保存することを求めるとき、諸実施形態は、これらの変更をバックエンドストアに変換することができる。すなわち、オブジェクト（ベースリレーション）のデルタ式に関するテーブル（ビュー）のデルタ式を生成することができる。これは、ビュー生成／コンパイルプロセスのＲＥＳ構造への分解が実際に役立つ領域である。非規定のマッピングに関するデルタ式は、比較的容易に生成することができる。これは、非規定のマッピングがリレーショナル空間にあり（ＲＥＳは純粋にリレーショナルである）、リレーショナルデータベースにおける大量の研究がこの目標を達成するために行われてきたからである。例えば、データベース文献では、デルタ式ルールが、選択、射影、内部結合、外部結合、準結合、共用体、交差、および差などのリレーショナル演算子に関して表されるビューについて開発されてきた。非リレーショナル構造について、必要なものは、非リレーショナル構造をＲＥＳ空間へ／から変換する規定のデルタ式を設計することだけである。

【０３３２】

Ｐｅｒｓｏｎの例を用いてデルタ式を理解しよう。ＲＥＳ構造（例えば、Ｒ＿ＳＡｄｄｒｅｓｓ）が２つのオブジェクトコレクション（Ｒ＿ＯＡｄｄｒｅｓｓとＲ＿ＯＰｅｒｓｏｎ＿Ａｄｄｒｅｓｓ）の結合として表される事例を検討されたい。そのようなビューに関するデルタ式は、次のルールを使用することによって得ることができる（結合ビューがＶ＝Ｒ　ＪＯＩＮ　Ｓであると想定されたい）。

【０３３３】

【表４９】

$$i(V) = [i(R) \text{ JOIN } Snew] \text{ UNION } [i(S) \text{ JOIN } Rnew]$$
$$d(V) = [d(R) \text{ JOIN } S] \text{ UNION } [d(S) \text{ JOIN } R]$$

【０３３４】

この式では、ｉ（Ｘ）およびｄ（Ｘ）は、リレーションまたはビューＸに関する挿入または削除されたタプルを表し、R^{new}は、全ての更新が適用された後のベースリレーションＲの新しい値を表す。

【０３３５】

したがって、ランタイムの更新を容易にするために、一例示的実施形態は、まず、コンパイル時に次のデルタ式を生成することができる。

１．更新されたオブジェクトコレクション１８０１のグループのデルタ変更式に関する、ＲＥＳリレーション１８１１の規定のデルタ変更式１８０３。例えば、ｉ（ＯＰｅｒｓｏｎ）に関するｉ（Ｒ＿ＯＰｅｒｓｏｎ）。

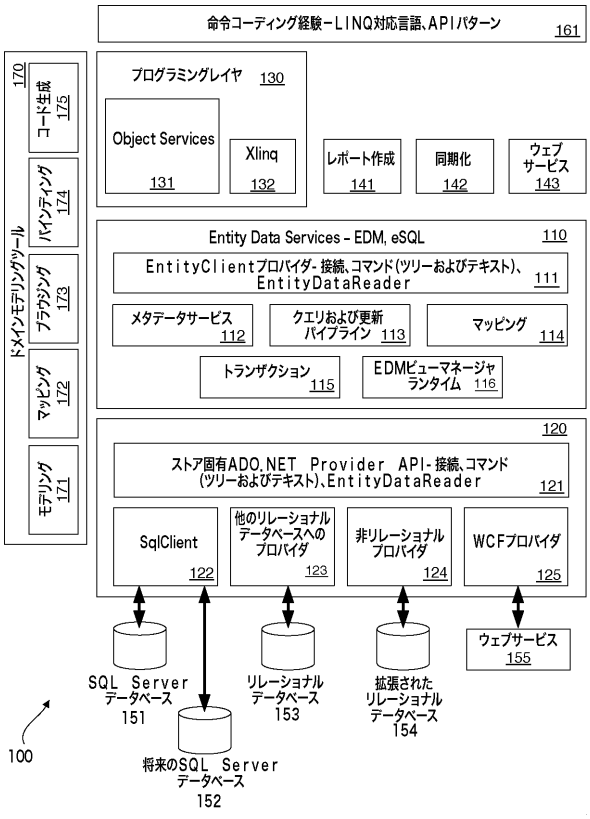
２．ＲＥＳリレーション１８１２のデルタ変更式に関するテーブル１８０２の規定のデルタ変更式１８０４。例えば、ｉ（Ｒ＿ＳＰｅｒｓｏｎ）に関するｉ（ＳＰｅｒｓｏｎ）。

３．オブジェクトのＲＥＳリレーションのデルタ式に関して表された、テーブルのＲＥＳリレーションのデルタ式１８１３。例えば、ｉ（Ｒ＿ＯＰｅｒｓｏｎ）に関するｉ（Ｒ＿ＳＰｅｒｓｏｎ）。

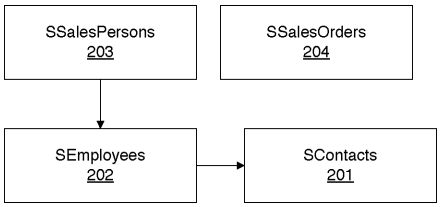
【０３３６】
上記の（１）、（２）、および（３）を構成して、オブジェクト１８２１（例えば、ＯＰｅｒｓｏｎ）のデルタ式に関するテーブル１８２２（例えば、ＳＰｅｒｓｏｎ）のデルタ式１８２０を取得することができる。この構成を図１８に図示する。したがって、クエリの場合と同様に、コンパイル時に、オブジェクトからテーブルへの直接変換を有する。更新の場合は、実際にＲＥＳ分解を活用してデルタ式を生成している（ＱＭＶｉｅｗについて、この利点は、通知に適用可能である）。
【０３３７】
更新に関するデルタ式は必要ないことに留意されたい。後でわかるように、モデル更新は、モデル更新を挿入および削除のセットに置くことによってモデル化されることが可能であり、後の処理ステップが、その後、変更を実際にデータベースに適用する前にそれらを更新に再変換する。このアプローチの理由の１つは、増分ビューの保守に関する既存の研究は、典型的には、更新に関するデルタ式がないことである。代替的に、そのような式が開発されるより複雑な実施形態が、実現可能である。
【０３３８】
ビュー構成を実行した後、テーブルに関するデルタ式は、純粋に、オブジェクトコレクションとオブジェクトの挿入および削除のセットとに関するものとすることができ、例えば、ｉ（ＳＰｅｒｓｏｎ）は、ＯＰｅｒｓｏｎ、ｉ（ＯＰｅｒｓｏｎ）、およびｄ（ＯＰｅｒｓｏｎ）の項目である。これらのデルタ式の一部は、オブジェクトコレクションが計算されることを必要とし、例えば、ｉ（ＯＰｅｒｓｏｎ）は、その計算にＥＰｅｒｓｏｎを必要とする。しかし、コレクション全体が、ＥＤＰクライアントでキャッシュされない場合がある（あるいは、コレクションの最も一貫した最新の値に対してこのオペレーションを実行したいことがある）。この問題に対処するために、対応するクエリマッピングビューを使用してオブジェクトコレクションをアンフォールドする。例えば、ＯＰｅｒｓｏｎにＱＭＶｉｅｗを使用し、ＳＰｅｒｓｏｎおよび必要な場合に他のリレーションに関してこれを表す。したがって、一実施形態において、コンパイルプロセスの終りに、ＳＰｅｒｓｏｎに関する全てのデルタ式が、ｉ（ＯＰｅｒｓｏｎ）、ｄ（ＯＰｅｒｓｏｎ）、およびリレーションＳＰｅｒｓｏｎ自体に関して表され、ランタイムに、ＯＰｅｒｓｏｎの挿入および削除のセットが与えられると、サーバで実行可能な関連するＳＱＬステートメントを生成することができる。
【０３３９】
要約すると、テーブルのＲＥＳ構造とオブジェクトとの間のＱＭＶｉｅｗおよびＵＭＶｉｅｗ、ならびにこれらの構造とテーブル／オブジェクトとの間の規定の変換が与えられると、次の例示的ステップを実行することができる。
１．上記のステップ１、２、および３で言及したデルタ式を生成する。
２．オブジェクト（ＯＰｅｒｓｏｎ）のデルタ式およびオブジェクトコレクション自体に関するテーブル（ＳＰｅｒｓｏｎ）のデルタ式を有するように、これらの式を構成する。
３．オブジェクトコレクションをそのＱＭＶｉｅｗを使用してアンフォールドして、オブジェクトのデルタ式およびテーブル自体に関するテーブル（ＳＰｅｒｓｏｎ）のデルタ式を取得する。すなわち、オブジェクトコレクションを除去する。諸実施形態において、このアンフォールディングを回避するか、コレクション全体がクライアントでキャッシュされることを知ることを可能にする、特別な事例が存在する可能性がある。
４．ランタイム作業を減らすように、式を単純化／最適化する。
【０３４０】
本明細書で明示的に説明された特定の実施形態に加えて、他の態様および実施態様が、本明細書に開示された仕様を考慮することから当業者に明らかなるであろう。添付の特許請求の範囲の真の範囲および趣旨とともに、この仕様および図示された実装は単なる例示としてみなされるべきと意図されている。
【図面の簡単な説明】

【０３４１】
【図１】本明細書で検討される例示的なＥｎｔｉｔｙ　Ｆｒａｍｅｗｏｒｋのアーキテクチャを示す図である。
【図２】例示的なリレーショナルスキーマを示す図である。
【図３】例示的なエンティティデータモデル（ＥＤＭ）スキーマを示す図である。
【図４】エンティティスキーマ（左）とデータベーススキーマ（右）との間のマッピングを示す図である。
【図５】エンティティスキーマおよびリレーショナルスキーマに対するクエリに関して表されたマッピングを示す図である。
【図６】図５のマッピングに関してマッピングコンパイラによって生成される双方向ビュー、すなわち、クエリビューと更新ビューを示す図である。
【図７】双方向ビューを介して更新を伝搬するマテリアライズビュー保守アルゴリズムを活用するプロセスを示す図である。
【図８】マッピングデザイナユーザインターフェースを示す図である。
【図９】クエリビューおよび更新ビューを生成するためにマッピング仕様言語（ＭＳＬ）で指定されるマッピングのコンパイルを示す図である。
【図１０】更新処理を示す図である。
【図１１】オブジェクトリレーショナル（ＯＲ）マッパーの例示的な論理部分を示す図である。
【図１２】ＭＳＬ仕様で指定されたマッピングを処理するときのエンティティデータプラットフォーム（ＥＤＰ）によるクエリビューおよび更新ビューの生成を示す図である。
【図１３】クエリ変換におけるＱＭＶｉｅｗの使用を示す図である。
【図１４】更新変換におけるＵＭＶｉｅｗの使用を示す図である。
【図１５】マッピングビューのコンパイル時処理およびランタイム処理を示す図である。
【図１６】ビューコンパイル処理における様々なコンポーネントの対話を示す図である。
【図１７】マッピングメタデータを利用して、オブジェクト／ＥＤＭ空間からデータベース空間へクエリを変換する、ＥＤＰクエリトランスレータ（ＥＱＴ）のアーキテクチャを示す図である。
【図１８】オブジェクトのデルタ式に関してテーブルのデルタ式を取得するための様々なデルタ式の構成を示す図である。

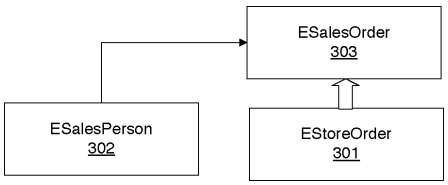
【図１】


命令コーディング経験－LINQ対応言語、APIパターン 161
ドメインモデリングツール 170
コード生成 175
バインディング 174
ブラウジング 173
マッピング 172
モデリング 171
プログラミングレイヤ 130
Object Services 131
Xlinq 132
レポート作成 141
同期化 142
ウェブサービス 143
Entity Data Services – EDM, eSQL 110
EntityClientプロバイダ-接続、コマンド(ツリーおよびテキスト)、EntityDataReader 111
メタデータサービス 112
クエリおよび更新パイプライン 113
マッピング 114
トランザクション 115
EDMビューマネージャランタイム 116
120
ストア固有ADO.NET Provider API-接続、コマンド(ツリーおよびテキスト)、EntityDataReader 121
SqlClient 122
他のリレーショナルデータベースへのプロバイダ 123
非リレーショナルプロバイダ 124
WCFプロバイダ 125
ウェブサービス 155
SQL Server データベース 151
将来のSQL Server データベース 152
リレーショナルデータベース 153
拡張されたリレーショナルデータベース 154
100

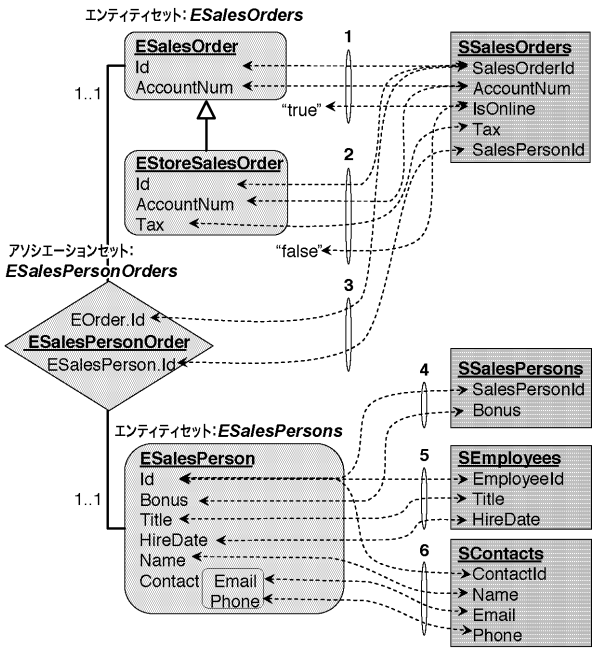
【図２】


SSalesPersons 203
SSalesOrders 204
SEmployees 202
SContacts 201

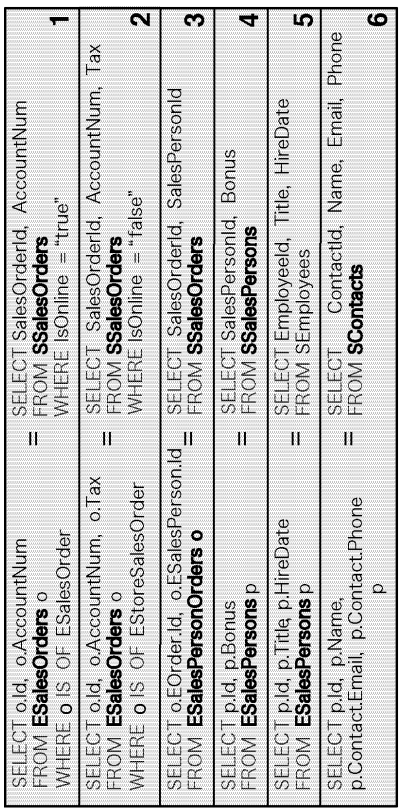
【図３】


ESalesOrder 303
ESalesPerson 302
EStoreOrder 301

【図４】


エンティティセット: ESalesOrders
ESalesOrder
Id
AccountNum
1..1
"true"
SSalesOrders
SalesOrderId
AccountNum
IsOnline
Tax
SalesPersonId
EStoreSalesOrder
Id
AccountNum
Tax
"false"
アソシエーションセット:
ESalesPersonOrders
EOrder.Id
ESalesPersonOrder
ESalesPerson.Id
SSalesPersons
SalesPersonId
Bonus
エンティティセット:ESalesPersons
ESalesPerson
Id
Bonus
Title
HireDate
Name
Contact
Email
Phone
SEmployees
EmployeeId
Title
HireDate
SContacts
ContactId
Name
Email
Phone

【図５】


1
SELECT o.Id, o.AccountNum
FROM ESalesOrders o
WHERE o IS OF ESalesOrder
=
SELECT SalesOrderId, AccountNum
FROM SSalesOrders
WHERE IsOnline = "true"
2
SELECT o.Id, o.AccountNum, o.Tax
FROM ESalesOrders o
WHERE o IS OF EStoreSalesOrder
=
SELECT SalesOrderId, AccountNum, Tax
FROM SSalesOrders
WHERE IsOnline = "false"
3
SELECT o.EOrder.Id, o.ESalesPerson.Id
FROM ESalesPersonOrders o
=
SELECT SalesOrderId, SalesPersonId
FROM SSalesOrders
4
SELECT p.Id, p.Bonus
FROM ESalesPersons p
=
SELECT SalesPersonId, Bonus
FROM SSalesPersons
5
SELECT p.Id, p.Title, p.HireDate
FROM ESalesPersons p
=
SELECT EmployeeId, Title, HireDate
FROM SEmployees
6
SELECT p.Id, p.Name,
p.Contact.Email, p.Contact.Phone
p
=
SELECT ContactId, Name, Email, Phone
FROM SContacts

【図６】

クエリビュー

```
ESalesOrders =                                                    QV1
SELECT
  CASE WHEN T.IsOnline = True
       THENESalesOrder(T.SalesOrderId, T.AccountNum)
       ELSE EStoreSalesOrder(T.SalesOrderId,
                             T.AccountNum, T.Tax)
  END
FROM SSalesOrders AS

ESalesPersonOrders =                                              QV2
SELECT ESalesPersonOrder(
         CreateRef(ESalesOrders, T.SalesOrderId),
         CreateRef(ESalesPersons, T.SalesPersonId))
FROM SSalesOrders AS T
```

更新ビュー

```
SSalesOrders =                                                    UV1
SELEC T o.Id, po.Id, o.AccountNum,
       TREAT(o AS EStoreSalesOrder).Tax AS Tax,
       CASE WHEN o IS OF ESalesOrder THEN TRUE ELSE FALSE END
       AS IsOnline
FROM ESalesOrders AS o
INNER JOIN ESalesPersonOrders AS po
ON o.SalesOrderId = Key(po.EOrder).Id
```

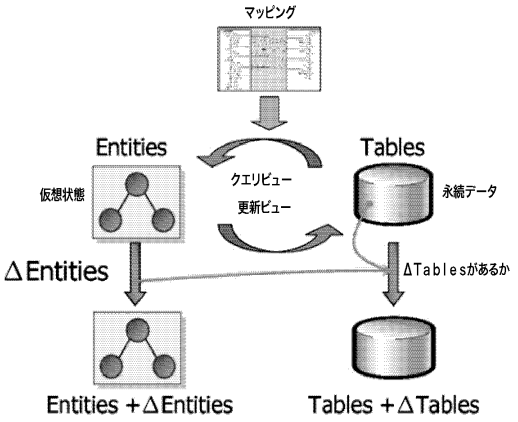
クエリビュー

```
ESalesPersons =                                                   QV3
SELECT ESalesPerson(p.SalesPersonId , p.Bonus,
                    e.Title , e.HireDate,
                    c.Name , Contact(c.Email , c.Phone))
FROM SSalesPersons AS p, SEmployees AS e, SContacts AS c
WHERE p.SalesPersonId = e.EmployeeId
AND e.EmployeeId = c.ContactId
```

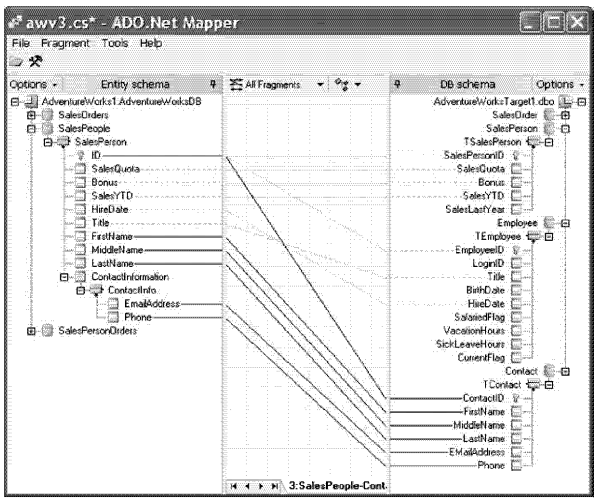
更新ビュー

```
SSalesPersons =                                                   UV2
SELECT p.Id, p.Bonus FROM ESalesPersons AS p

SEmployees =                                                      UV3
SELECT p.Id, p.Title, p.HireDate FROM ESalesPersons AS p

SContacts =                                                       UV4
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone
FROM ESalesPersons AS p
```

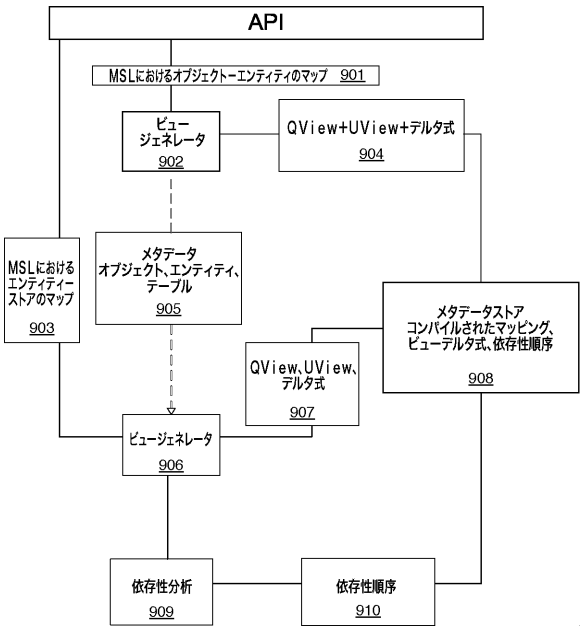
【図７】



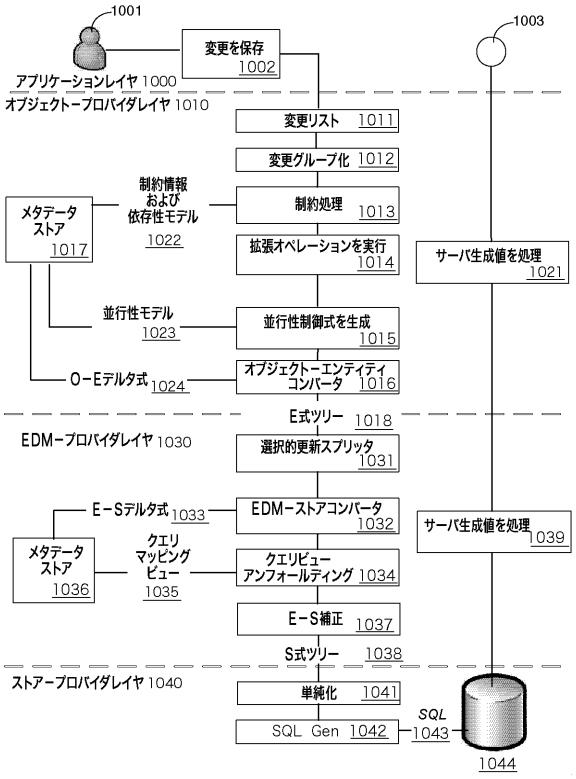
【図８】



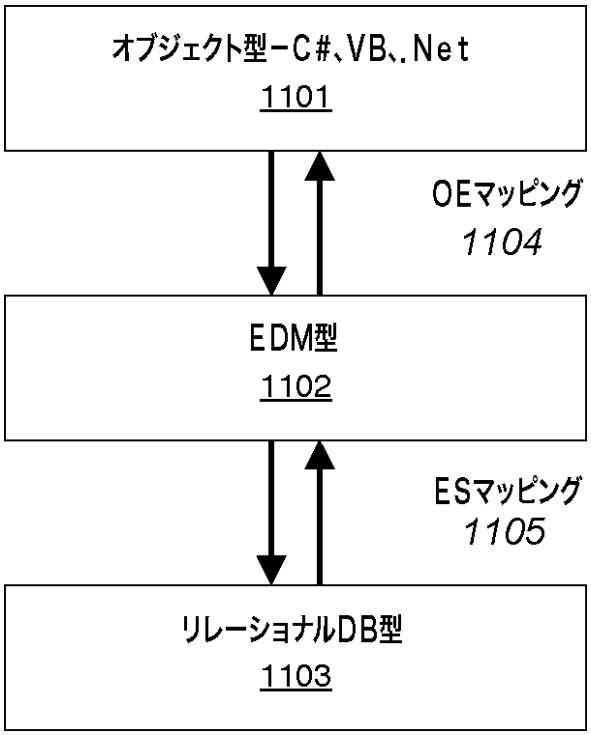
【図９】



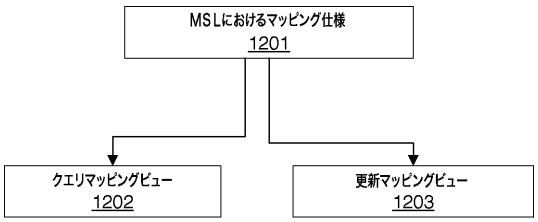
【図１０】



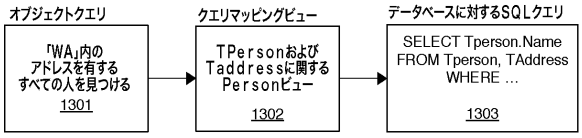
【図１１】



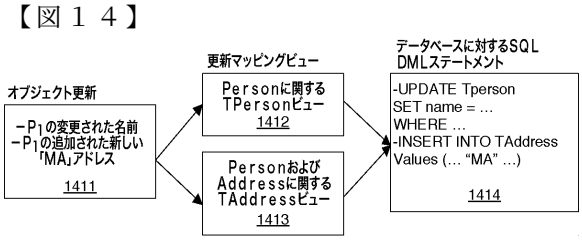
【図１２】



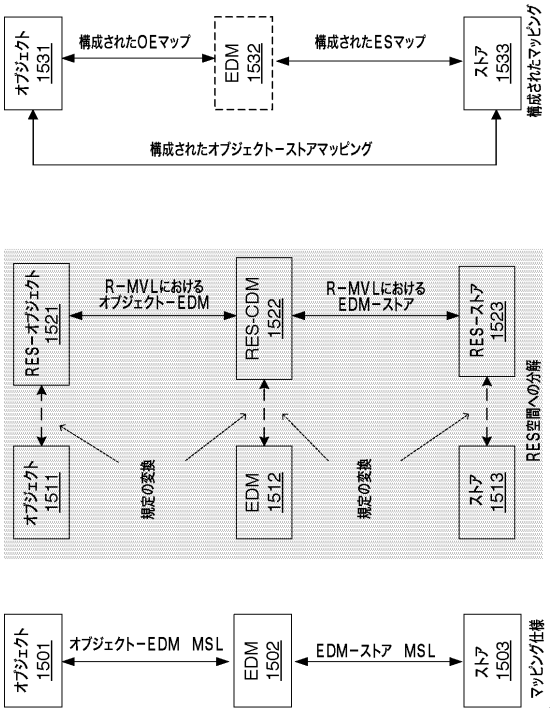
【図１３】



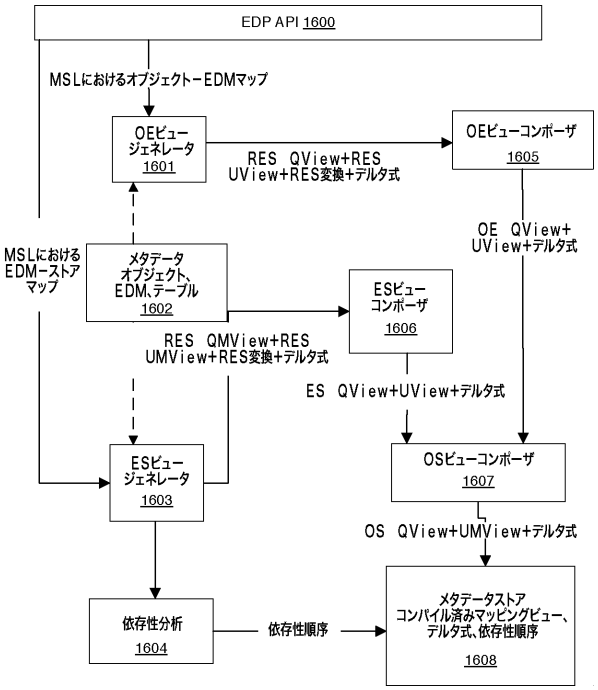
【図１４】



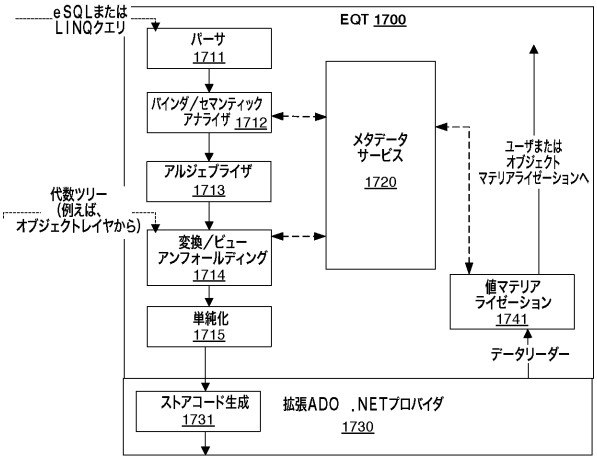
【図１５】



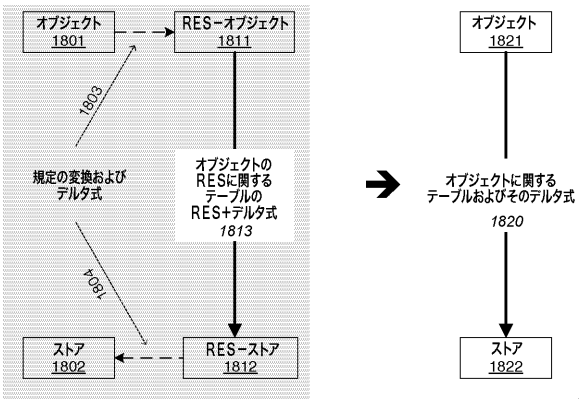
【図１６】


EDP API 1600
MSLにおけるオブジェクト－EDMマップ
OEビュージェネレータ 1601
OEビューコンポーザ 1605
RES QView+RES UView+RES変換+デルタ式
OE QView+UView+デルタ式
MSLにおけるEDM－ストアマップ
メタデータオブジェクト、EDM、テーブル 1602
ESビューコンポーザ 1606
RES QMView+RES UMView+RES変換+デルタ式
ES QView+UView+デルタ式
ESビュージェネレータ 1603
OSビューコンポーザ 1607
OS QView+UMView+デルタ式
依存性分析 1604
依存性順序
メタデータストアコンパイル済みマッピングビュー、デルタ式、依存性順序 1608

【図１７】


eSQLまたはLINQクエリ
EQT 1700
パーサ 1711
バインダ／セマンティックアナライザ 1712
アルジェブライザ 1713
変換／ビューアンフォールディング 1714
単純化 1715
メタデータサービス 1720
代数ツリー（例えば、オブジェクトレイヤから）
ユーザまたはオブジェクトマテリアライゼーションへ
値マテリアライゼーション 1741
データリーダー
ストアコード生成 1731
拡張ADO .NETプロバイダ 1730

【図１８】


オブジェクト 1801
RES－オブジェクト 1811
1803
規定の変換およびデルタ式
オブジェクトのRESに関するテーブルのRES+デルタ式 1813
1804
ストア 1802
RES－ストア 1812
オブジェクト 1821
オブジェクトに関するテーブルおよびそのデルタ式 1820
ストア 1822

フロントページの続き

(72)発明者 ジョセ エー. ブラークレイ
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション インターナショナル パテンツ内
(72)発明者 パーーエーク ラーソン
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション インターナショナル パテンツ内
(72)発明者 セルゲイ メルニク
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション インターナショナル パテンツ内

審査官 桜井 茂行

(56)参考文献 米国特許第０６０５８３９１（ＵＳ，Ａ）
米国特許第０６８６５５６９（ＵＳ，Ｂ１）
米国特許第０６９１５３０５（ＵＳ，Ｂ２）
米国特許第０６６１８７３３（ＵＳ，Ｂ１）
小島 功，アクティブデータベースの動向と応用へのインパクトについて，情報処理学会研究報告，日本，社団法人情報処理学会，１９９４年１１月１８日，第９４巻第９９号，P. 73-78

(58)調査した分野(Int.Cl.，ＤＢ名)
G06F 12/00
G06F 17/30