(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号

特許第5113967号
(P5113967)

(45)発行日　平成25年1月9日(2013.1.9)　(24)登録日　平成24年10月19日(2012.10.19)

(51)Int.Cl.		F I		
G06F 12/00	(2006.01)	G06F	12/00	545A
G06F 17/30	(2006.01)	G06F	12/00	513D
		G06F	17/30	180D

請求項の数 14　(全 56 頁)

(21)出願番号	特願2001-516067 (P2001-516067)
(86)(22)出願日	平成12年7月26日 (2000.7.26)
(65)公表番号	特表2003-527659 (P2003-527659A)
(43)公表日	平成15年9月16日 (2003.9.16)
(86)国際出願番号	PCT/US2000/020386
(87)国際公開番号	WO2001/011486
(87)国際公開日	平成13年2月15日 (2001.2.15)
審査請求日	平成19年7月25日 (2007.7.25)
審判番号	不服2011-3240 (P2011-3240/J1)
審判請求日	平成23年2月14日 (2011.2.14)
(31)優先権主張番号	60/147,538
(32)優先日	平成11年8月5日 (1999.8.5)
(33)優先権主張国	米国 (US)
(31)優先権主張番号	09/571,036
(32)優先日	平成12年5月15日 (2000.5.15)
(33)優先権主張国	米国 (US)

(73)特許権者　502303739
オラクル・インターナショナル・コーポレイション
アメリカ合衆国、９４０６５　カリフォルニア州、レッドウッド・ショアーズ、オラクル・パークウェイ、５００
(74)代理人　100064746
弁理士　深見　久郎
(74)代理人　100085132
弁理士　森田　俊雄
(74)代理人　100083703
弁理士　仲村　義平
(74)代理人　100096781
弁理士　堀井　豊

最終頁に続く

(54)【発明の名称】インターネットファイルシステム

(57)【特許請求の範囲】

【請求項１】

データベースに記憶されたデータにアクセスする方法であって、

該方法は、プロトコルサーバが、オペレーティングシステムの１つ以上のルーチンからの、１つ以上のＩ／Ｏコマンドを受取るステップを含み、前記プロトコルサーバは、前記オペレーティングシステムで動作するよう構成されており、

前記１つ以上のルーチンは、アプリケーションから前記オペレーティングシステムへの、ファイルへアクセスするための１つ以上のコールに応答して、１つ以上のＩ／Ｏコマンドを生成しており、

該方法は、

前記プロトコルサーバが、前記１つ以上のＩ／Ｏコマンドを１つ以上のＤＢファイルシステムコマンドに変換するステップと、

ＤＢファイルサーバが、前記１つ以上のＤＢファイルシステムコマンドに応答して、１つ以上の第１のデータベースコマンドを、生成するとともに前記データベースを管理するデータベースサーバに対して発行するステップとを含み、

前記データベースサーバは、前記１つ以上の第１のデータベースコマンドを実行して前記データベースから第１のデータを検索し、前記第１のデータから生成されたファイルを第１のアプリケーションに提供し、該方法はさらに、

第２のアプリケーションに対してＤＢファイルＡＰＩを提供するステップと、

前記ＤＢファイルサーバが、前記ＤＢファイルＡＰＩを介して、前記第２のアプリケー

ションからの１つ以上の第２のＤＢファイルシステムコマンドを直接受取るステップとを含み、
前記１つ以上の第２のＤＢファイルシステムコマンドに応答して、１つ以上の第２のデータベースコマンドが、生成されるとともに前記データベースサーバに対して発行され、
前記データベースサーバは、前記１つ以上の第２のデータベースコマンドを実行して前記データベースから第２のデータを検索し、前記第２のデータから生成されたファイルを前記第２のアプリケーションに提供する、方法。
【請求項２】
前記ファイルをアプリケーションに提供する処理は、前記オペレーティングシステムにおける１つ以上のルーチンによって実行される、請求項１に記載の方法。
【請求項３】
前記ＤＢファイルサーバが、前記ＤＢファイルＡＰＩを介して、複数のファイルオペレーションを行なうためのコールを受取るステップを含み、前記複数のファイルオペレーションは、少なくとも、前記データベースに記憶された第１のファイルに対する第１のファイルオペレーションと、前記データベースに記憶された第２のファイルに対する第２のファイルオペレーションとを含み、該方法はさらに、
前記ＤＢファイルサーバが、前記複数のファイルオペレーションを、次のステップを行なうことによって、単一のトランザクションとして行なうステップを含み、該次のステップは、
もし前記複数のファイルオペレーションのうちすべてのファイルオペレーションが失敗なく完了すれば、前記複数のファイルオペレーションによってなされたすべての変化を永久のものとするステップと、
もし前記複数のファイルオペレーションのうち何らかのファイルオペレーションが失敗すれば、前記複数のファイルオペレーションのすべてによってなされたすべての変化を無効にするステップとを含む、請求項２に記載の方法。
【請求項４】
前記複数のファイルオペレーションは、前記データベースに記憶された単一のファイルに対する複数の書込オペレーションを含む、請求項３に記載の方法。
【請求項５】
前記複数のファイルオペレーションを行なうステップは、前記データベースサーバに対して１つ以上のデータベースステートメントを発行するステップを含み、前記データベースサーバは前記１つ以上のデータベースステートメントを実行して前記複数のファイルオペレーションを行なう、請求項３に記載の方法。
【請求項６】
前記複数の書込オペレーションは、前記単一のファイルを前記データベースに記憶するためにネットワーク接続にわたって転送することに相当する、請求項４に記載の方法。
【請求項７】
前記プロトコルサーバは、デバイスドライバインターフェイスとして機能する、請求項１～６のいずれか１項に記載の方法。
【請求項８】
データベースに記憶されたデータにアクセスするための命令を記憶した１つ以上のコンピュータ読取可能媒体であって、該命令は、１つ以上のプロセッサによって実行されると、以下のステップを生じさせ、
該以下のステップは、プロトコルサーバが、オペレーティングシステムの１つ以上のルーチンからの、１つ以上のＩ／Ｏコマンドを受取るステップを含み、前記プロトコルサーバは、前記オペレーティングシステムで動作するよう構成されており、
前記１つ以上のルーチンは、アプリケーションから前記オペレーティングシステムへの、ファイルへアクセスするための１つ以上のコールに応答して、１つ以上のＩ／Ｏコマンドを生成しており、
該以下のステップは、

前記プロトコルサーバが、前記１つ以上のＩ／Ｏコマンドを１つ以上のＤＢファイルシステムコマンドに変換するステップと、

ＤＢファイルサーバが、前記１つ以上のＤＢファイルシステムコマンドに応答して、１つ以上の第１のデータベースコマンドを、生成するとともに前記データベースを管理するデータベースサーバに対して発行するステップとを含み、

前記データベースサーバは、前記１つ以上の第１のデータベースコマンドを実行して前記データベースから第１のデータを検索し、前記第１のデータから生成されたファイルを第１のアプリケーションに提供し、該以下のステップはさらに、

第２のアプリケーションに対してＤＢファイルＡＰＩを提供するステップと、

前記ＤＢファイルサーバが、前記ＤＢファイルＡＰＩを介して、前記第２のアプリケーションからの１つ以上の第２のＤＢファイルシステムコマンドを直接受取るステップとを含み、

前記１つ以上の第２のＤＢファイルシステムコマンドに応答して、１つ以上の第２のデータベースコマンドが、生成されるとともに前記データベースサーバに対して発行され、

前記データベースサーバは、前記１つ以上の第２のデータベースコマンドを実行して前記データベースから第２のデータを検索し、前記第２のデータから生成されたファイルを前記第２のアプリケーションに提供する、コンピュータ読取可能媒体。

【請求項９】

前記ファイルをアプリケーションに提供する処理は、前記オペレーティングシステムにおける１つ以上のルーチンによって実行される、請求項８に記載のコンピュータ読取可能媒体。

【請求項１０】

前記ＤＢファイルサーバが、前記ＤＢファイルＡＰＩを介して、複数のファイルオペレーションを行なうためのコールを受取るステップを含み、前記複数のファイルオペレーションは、少なくとも、前記データベースに記憶された第１のファイルに対する第１のファイルオペレーションと、前記データベースに記憶された第２のファイルに対する第２のファイルオペレーションとを含み、該以下のステップはさらに、

前記ＤＢファイルサーバが、前記複数のファイルオペレーションを、次のステップを行なうことによって、単一のトランザクションとして行なうステップを含み、該次のステップは、

もし前記複数のファイルオペレーションのうちすべてのファイルオペレーションが失敗なく完了すれば、前記複数のファイルオペレーションによってなされたすべての変化を永久のものとするステップと、

もし前記複数のファイルオペレーションのうち何らかのファイルオペレーションが失敗すれば、前記複数のファイルオペレーションのすべてによってなされたすべての変化を無効にするステップとを含む、請求項９に記載のコンピュータ読取可能媒体。

【請求項１１】

前記複数のファイルオペレーションは、前記データベースに記憶された単一のファイルに対する複数の書込オペレーションを含む、請求項１０に記載のコンピュータ読取可能媒体。

【請求項１２】

前記複数のファイルオペレーションを行なうステップは、前記データベースサーバに対して１つ以上のデータベースステートメントを発行するステップを含み、前記データベースサーバは前記１つ以上のデータベースステートメントを実行して前記複数のファイルオペレーションを行なう、請求項１０に記載のコンピュータ読取可能媒体。

【請求項１３】

前記複数の書込オペレーションは、前記単一のファイルを前記データベースに記憶するためにネットワーク接続にわたって転送することに相当する、請求項１１に記載のコンピュータ読取可能媒体。

【請求項１４】

前記プロトコルサーバは、デバイスドライバインターフェイスとして機能する、請求項８～１３のいずれか１項に記載のコンピュータ読取可能媒体。

【発明の詳細な説明】

【０００１】

【優先権主張および関連出願の参照】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「インターネットファイルシステム（"Internet File System"）」と題された１９９９年８月５日出願の先行米国仮特許出願連続番号第６０／１４７，５３８号に関し、米国特許法第１１９条（ｅ）によりその国内優先権を主張する。

【０００２】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「関係システム内に階層状に構成された情報にアクセスするための階層索引付け（"Hierarchical Indexing for Accessing Hierarchically Organized Information in a Relational System"）」と題された１９９９年２月１８日出願の米国特許出願連続番号第０９／２５１，７５７号に関連する。

【０００３】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「トランザクションをサポートするファイルシステム（"File System that Supports Transactions"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，４９６号に関連する。

【０００４】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「記憶されたクエリディレクトリ（"Stored Query Directories"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，０６０号に関連する。

【０００５】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「ファイルシステムに結合されたイベント通知システム（"Event Notification System Tied to a File System"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，０３６号に関連する。

【０００６】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「ファイルがタイプ付けされたオブジェクトファイルシステム（"Object File System with Typed Files"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，４９２号に関連する。

【０００７】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「オンザフライ・フォーマット変換（"On-the-fly Format Conversion"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，５６８号に関連する。

【０００８】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric SedlarおよびMichael J. Robertsによる「インターネットファイルシステムにおけるバージョニング（"Versioning in Internet File System"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，６９６号に関連する。

【０００９】

本願は、その全文が本明細書中に完全に述べられているかのように引用により援用される、Eric Sedlarによる「データへのマルチモデルアクセス（"Multi-Model Access to Data"）」と題された２０００年５月１５日出願の米国特許出願連続番号第０９／５７１，５０８号に関連する。

【００１０】
【発明の分野】
本発明は一般に電子ファイルシステムに関し、特定的には、データベースシステムを用いてオペレーティングシステムファイルシステムを実現するシステムに関する。
【００１１】
【発明の背景】
ヒトは情報をカテゴリに分類する傾向にあり、情報が分類されるそれらカテゴリ自体は典型的に、何らかの階層状に互いに関連付けて構成される。たとえば、個々の動物は種に属し、種は属に属し、属は科に属し、科は目に属し、目は綱に属する。
【００１２】
コンピュータシステムの出現に伴ない、このような階層構成（hierarchical organization）を望むヒトの欲求を大いに反映する電子情報の記憶技術が開発されてきた。従来のオペレーティングシステムは、たとえば、階層ベースの構成原理を使用するファイルシステムを提供する。具体的には、典型的なオペレーティングシステムファイルシステム（「ＯＳファイルシステム」）においては、ディレクトリは階層に配され、文書（ドキュメント）はそれらディレクトリに記憶される。理想的には、ディレクトリ間の階層的関係は、それらディレクトリに割当てられた意味間の何らかの直観的な関係を反映する。同様に、各ドキュメントがディレクトリに記憶される場合、そのドキュメントの内容と、そのドキュメントが記憶されるディレクトリに割当てられた意味との間の何らかの直観的関係に基づいて、記憶されると理想的である。
【００１３】
図１は、（ワードプロセッサ等の）ファイルを作成して使用するソフトウェアアプリケーションがそのファイルを階層的ファイルシステム内に記憶する際に用いられる、典型的な機構を示す。図１を参照して、オペレーティングシステム１０４は、アプリケーション１０２に対してアプリケーションプログラミングインターフェイス（ＡＰＩ）を開く（expose）。そうして開かれたＡＰＩにより、アプリケーション１０２はそのオペレーティングシステムによって提供されるルーチンをコールすることができる。以後、ＯＳ　ＡＰＩの、ＯＳファイルシステムを実現するルーチンに関連する部分を、ＯＳファイルＡＰＩと称する。アプリケーション１０２は、ＯＳファイルＡＰＩを介してファイルシステムルーチンをコールして、データを検索してディスク１０８に記憶する。オペレーティングシステム１０４の方は、ディスク１０８へのアクセスを制御するデバイスドライバ１０６に対してコールを行なって、ディスク１０６からファイルを検索させたりディスク１０６にファイルを記憶させたりする。
【００１４】
ＯＳファイルシステムルーチンは、ファイルシステムの階層的構成を実現する。たとえば、ＯＳファイルシステムルーチンは、ファイル間の階層的関係に関する情報を維持し、アプリケーション１０２にファイルへのアクセスを、階層内における当該ファイルの場所に基づいて与える。
【００１５】
電子情報を階層的に構成するのに対して、関係データベース（relational database）は、情報を行列からなるテーブルに記憶する。各行は独自のRowIDによって識別される。各列は記録の属性を表わし、各行は特定の記録を表わす。データベースからのデータの検索は、データベースを管理するデータベース管理システム（ＤＢＭＳ）にクエリを提示することによって行なわれる。
【００１６】
図２は、データベースアプリケーションがデータベース内の情報にアクセスする際に用いられる典型的な機構を示す。図２を参照して、データベースアプリケーション２０２は、データベースサーバ２０４によって提供されるＡＰＩ（「データベースＡＰＩ」）を通じてデータベースサーバ２０４と対話する。このように開かれたＡＰＩにより、データベースアプリケーション２０２は、データベースサーバ２０４によってサポートされたデータ

ベース言語により構築されたクエリを用いて、データにアクセスすることができる。多くのデータベースサーバによってサポートされる言語の１つに、構造化クエリ言語（Structured Query Language，SQL）がある。データベースサーバ２０４は、データベースアプリケーション２０２に対して、すべてのデータがテーブルの行に記憶されているように見せる。しかし、データベースアプリケーション２０２にトランスペアレントなことに、データベースサーバ２０４は実際にはオペレーティングシステム１０４と対話して、データをファイルとしてＯＳファイルシステム内に記憶する。オペレーティングシステム１０４の方では、デバイスドライバ１０６に対してコールを行なって、ファイルをディスク１０８から検索させたりファイルをディスク１０８に記憶させたりする。

【００１７】

各種記憶システムにはそれぞれ、利点および限界がある。階層的に構成された記憶システムは、簡単、直観的、かつ実現が容易であって、大半のアプリケーションプログラムによって使用される標準的なモデルである。しかし、残念ながら、この階層的構成の簡易性は、複雑なデータ検索オペレーションに求められるサポートを提供することができない。たとえば、特定日に作成された特定的なファイル名を有するすべてのドキュメントを検索するのに、すべてのディレクトリの内容を検査せねばならないことがあり得る。すべてのディレクトリをサーチせねばならないので、階層的構成は検索プロセスを容易にすることはできない。

【００１８】

関係データベースシステムは、大量の情報を記憶したり、非常に柔軟にデータにアクセスするのに、好適である。階層的に構成されたシステムに対して、複雑なサーチ基準に合致するデータでさえも、関係データベースシステムからは容易にかつ効率的に検索することが可能である。しかし、クエリを公式化（formulate）してデータベースサーバに提示するプロセスは、ディレクトリの階層を単に通り抜けるのに比して直観的ではなく、多くのコンピュータユーザにとっての技術的快適度を越えるものである。

【００１９】

現時点において、アプリケーションの開発者は、それらのアプリケーションによって作成されるデータを、オペレーティングシステムによって提供される階層的ファイルシステムを通じてアクセス可能としたいか、それともデータベースシステムによって提供されるより複雑なクエリインターフェイスを通じてアクセス可能としたいか、どちらかを選択するよう求められる。一般に、データベースシステムの複雑なサーチ能力を要求しないアプリケーションについては、オペレーティングシステムによって提供されるより一般的かつより簡単な階層的ファイルシステムを使用して、それらのデータを記憶するように設計される。この場合、アプリケーションの設計およびアプリケーションの使用がどちらも簡素化されるものの、それらデータにアクセスすることのできる柔軟性およびパワーの面で制限が課されてしまう。

【００２０】

これに対し、複雑なサーチ能力が求められる場合には、アプリケーションは、データベースシステムによって提供されるクエリ機構を使用してそれらのデータにアクセスするように設計される。この場合、データにアクセスすることのできる柔軟性およびパワーは増すが、それと同時に、アプリケーションの複雑性が、設計者の観点からもユーザの観点からも増す。さらに、データベースシステムの存在も求められ、アプリケーションユーザに対して付加的な費用がかかることになる。

【００２１】

以上に鑑みて、アプリケーションが比較的簡単なＯＳファイルＡＰＩを使用してデータにアクセスすることができることが明らかに望ましい。また、より強力なデータベースＡＰＩを使用して同じデータにアクセスすることができることがさらに望ましい。

【００２２】

【発明の概要】

データベースに記憶されたデータにアクセスするための技術が提供される。一技術に従え

ば、アプリケーションはオペレーティングシステムに対して１または複数のコールを行なってファイルにアクセスする。該オペレーティングシステムは、オペレーティングシステムファイルシステムを実現するルーチンを含む。該１または複数のコールは、該オペレーティングシステムファイルシステムを実現するルーチンに対して行なわれる。該１または複数のコールに応答して、１または複数のデータベースコマンドがデータベースを管理するデータベースサーバに対して発行される。該データベースサーバはそのデータベースコマンドを実行して、データベースからデータを検索する。該データからファイルが生成されて、該アプリケーションに与えられる。

【００２３】

本発明を以下に限定のためではなく例示の目的で、添付の図面を参照して説明する。図中、同一の参照符号は同一の要素を表わす。

【００２４】

【好ましい実施例の詳細な説明】

同じデータの組に対して、データベースＡＰＩおよびＯＳファイルシステムＡＰＩを含む種々のインターフェイスを介してアクセスすることを可能にする、方法およびシステムが提供される。以下に、説明の目的で、本発明が完全に理解されるように多数の具体的な詳細が述べられるが、当業者には、本発明がそれらの具体的な詳細を伴なわずに実施され得ることは明らかであろう。他の例においては、本発明を不必要にあいまいにすることのないように、周知の構造および装置はブロック図により示される。

【００２５】

アーキテクチャ的な概観

図３は、本発明の一実施例に従って実現されるシステム３００のアーキテクチャを表わすブロック図である。図２に示すシステムと同様、システム３００は、データベースＡＰＩを提供するデータベースサーバ２０４を含み、このデータベースＡＰＩを通じて、データベースアプリケーション３１２が、データベースサーバ２０４により管理されるデータにアクセスし得る。データベースＡＰＩを通じてデータベースサーバ２０４により管理されるデータにアクセスするすべてのエンティティの観点から、データベースサーバ２０４により管理されるデータはデータベースサーバ２０４（たとえばＳＱＬ）によりサポートされるデータベース言語を用いて照会され得る関係テーブルにストアされる。これらのエンティティに対してトランスペアレントに、データベースサーバ２０４はこのデータをディスク１０８にストアする。一実施例によれば、データベースサーバ２０４は、データを直接ディスクにストア可能にすることによってオペレーティングシステム１０４のＯＳファイルシステムに伴うオーバヘッドを回避できるようにする、ディスク管理論理を実現する。したがって、データベースサーバ２０４は、（１）オペレーティングシステム１０４により提供されたＯＳファイルシステムをコールするか、または（２）データを直接ディスクにストアすることでオペレーティングシステム１０４を迂回するかのいずれかによって、データがディスクにストアされるようにし得る。

【００２６】

図２のシステムとは異なって、システム３００はトランスレーションエンジン３０８を提供し、これはオペレーティングシステム３０４ａおよび３０４ｂから受けたＩ／Ｏコマンドを、トランスレーションエンジン３０８がデータベースサーバ２０４へ発するデータベースコマンドへ変換する。Ｉ／Ｏコマンドがデータのストレージを求める場合、トランスレーションエンジン３０８はデータベースコマンドをデータベースサーバ２０４へ発し、データベースサーバ２０４により管理される関係テーブルにデータがストアされるようにする。Ｉ／Ｏコマンドがデータの検索を求める場合、トランスレーションエンジン３０８はデータベースコマンドをデータベースサーバ２０４に発し、データベースサーバにより管理される関係テーブルからデータを検索する。その後トランスレーションエンジン３０８は、こうして検索されたデータを、Ｉ／Ｏコマンドを発したオペレーティングシステムに与える。

【００２７】

オペレーティングシステム３０４ａおよび３０４ｂに対しては、トランスレーションエンジン３０８に伝達されたデータがデータベースサーバ２０４により管理される関係テーブルに最終的にストアされるという事実はトランスペアレントである。これは、オペレーティングシステム３０４ａおよび３０４ｂにトランスペアレントであるので、それらのオペレーティングシステムを含むプラットホーム上で実行されているアプリケーション３０２ａおよび３０２ｂに対してもトランスペアレントである。

【００２８】

たとえば、アプリケーション３０２ａのユーザがアプリケーション３０２ａにより与えられる「ファイルの保存」という選択肢を選択する場合を想定する。アプリケーション３０２ａはＯＳファイルＡＰＩを通じてコールを行ない、オペレーティングシステム３０４ａにファイルを保存させる。オペレーティングシステム３０４ａはトランスレーションエンジン３０８にＩ／Ｏコマンドを発し、ファイルをストアさせる。トランスレーションエンジン３０８はこれに応答して、データベースサーバ２０４に１つ以上のデータベースコマンドを発し、データベースサーバ２０４に、ファイル内に含まれるデータをデータベースサーバ２０４が保持する関係テーブルにストアさせる。データベースサーバ２０４は、このデータを直接ディスクにストアしてもよく、またはオペレーティングシステム１０４をコールしてオペレーティングシステム１０４により提供されるＯＳファイルシステムにデータをストアさせてもよい。データベースサーバ２０４がオペレーティングシステム１０４をコールすると、オペレーティングシステム１０４はこれに応答して、デバイスドライバ１０６にコマンドを送ることによりデータをディスク１０８にストアさせる。

【００２９】

別の例として、アプリケーション３０２ａのユーザがアプリケーション３０２ａにより与えられる「ファイルのロード」という選択肢を選択する場合を想定する。アプリケーション３０２ａはOS File APIを通じてコールを行ない、オペレーティングシステム３０４ａにファイルをロードさせる。オペレーティングシステム３０４ａはＩ／Ｏコマンドをトランスレーションエンジン３０８に発し、ファイルのロードを行なわせる。トランスレーションエンジン３０８は、１つ以上のデータベースコマンドをデータベースサーバ２０４に発し、データベースサーバ２０４に、検索すべきファイルを備えるデータをデータベースサーバ２０４が保持する関係テーブルから検索させる。データの検索中、データベースサーバ２０４はデータディレクトリを検索してもよく、またはオペレーティングシステム１０４をコールしてディスク１０８上のＯＳファイルからデータを検索させてもよい。一旦データが検索されると、この検索されたデータから所望のファイルが「構築される」。具体的には、この検索されたデータはファイルをリクエストしたアプリケーション３０２ａにより予測されたフォーマットにされる。こうして構築されたファイルは、トランスレーションエンジン３０８およびオペレーティングシステム３０４ａを通じて、アプリケーション３０２ａまで伝達される。

【００３０】

システム３００には数多くの新規な特徴が組入れられる。以下のセクションでは、これらの特徴をより詳細に説明する。しかしながら、当然、特定の実施例はこれらの特徴を説明するために用いられるのであり、本発明がこれらの特定の実施例に限定されることはない。

【００３１】

関係づけてストアされたデータへのＯＳファイルシステムアクセス

本発明のある局面によれば、システム３００により、アプリケーションが従来のＯＳファイルＡＰＩを通じて、データベースにストアされたデータにアクセスできるようになる。すなわち、オペレーティングシステムにより提供される標準ＯＳファイルＡＰＩをコールすることによりファイルをロードするように設計されている従来のアプリケーションが、関係テーブルにストアされたデータからオンザフライで構築されたファイルをロードできるようになる。さらに、関係テーブルからデータが発生するという事実は、アプリケーションに対しては完全にトランスペアレントである。

【００３２】
たとえば、データベースアプリケーション３１２が、データベースサーバ２０４により保持されるデータベース中のテーブルに１行のデータを挿入するというデータベースコマンドを発すると想定する。一旦その行が挿入されると、オペレーティングシステム３０４ａにより提供される比較的単純なＯＳファイルＡＰＩを用いてデータにアクセスするようにしか設計されていないアプリケーション３０２ａは、「ファイルを開く」というコマンドをオペレーティングシステム３０４ａに発する。これに応答して、オペレーティングシステム３０４ａはＩ／Ｏコマンドをトランスレーションエンジン３０８に発し、トランスレーションエンジン３０８は、１つ以上のデータベースコマンドをデータベースサーバ２０４に発することにより応答する。データベースサーバ２０４は、データベースコマンド（典型的にはデータベースクエリの形式）を実行することにより、データベースサーバ２０４に、データベースアプリケーション３１２により挿入された行を検索させる。アプリケーション３０２ａにより予測されるファイルタイプのファイルがその行に含まれるデータから構築され、こうして構築されたファイルが、トランスレーションエンジン３０８およびオペレーティングシステム３０４ａを通じて再びアプリケーション３０２ａへ戻される。

【００３３】
システム３００により、従来のＯＳファイルシステムアクセスしかサポートしていないアプリケーションが、関係づけてストアされたデータをロードできるようになるだけでなく、従来のＯＳファイルシステムアクセスしかサポートしないアプリケーションによりストアされた情報に、データベースアプリケーションが従来の照会技術を用いてアクセスできるようになる。たとえば、アプリケーション３０２ａがＯＳのコールを行ない、作成されたファイルを保存させるとする。その「ファイルの保存」コマンドはオペレーティングシステム３０４ａおよびトランスレーションエンジン３０８を通じてデータベースサーバ２０４へ伝達される。データベースサーバ２０４は「ファイルの保存」コマンドをトランスレーションエンジン３０８により発せられたデータベースコマンドの形で受け、そのファイルに含まれるデータを、データベースサーバ２０４により管理されるデータベース中に含まれる１つ以上のテーブルの１行以上の行中にストアする。データが一旦その態様でデータベース内にストアされると、データベースアプリケーション３１２はデータベースサーバ２０４にデータベースクエリを発し、データベースからデータを検索することができる。

【００３４】
データベースにおけるＯＳファイルシステム構成のエミュレート
上記で説明したように、オペレーティングシステム３０４ａおよび３０４ｂのファイルシステムルーチンに対するコールは、最終的に、トランスレーションエンジン３０８がデータベースサーバ２０４に対して発するデータベースコマンドに変換される。本発明の一実施例によれば、これらの変換を行なう処理は、オペレーティングシステム３０４ａおよび３０４ｂにより実現されたファイルシステムの特徴をデータベースサーバ２０４内でエミュレートすることにより単純化される。

【００３５】
この構成モデルに関して、ほとんどのオペレーティングシステムは、ファイル階層構造でファイルを構成するファイルシステムを実現する。したがって、アプリケーション３０２ａおよび３０２ｂが行なったこのＯＳファイルシステムのコールは、典型的には、ＯＳファイル階層構造内のその場所という観点からあるファイルを特定するだろう。このようなコールから対応するデータベースのコールへの変換を単純化するために、関係のあるデータベースシステム内の階層ファイルシステムをエミュレートするための機構が設けられる。このような機構の１つが、１９９９年2月１８日にエリック・セドラー（Eric Sedlar）により出願され「関係のあるシステムおいて階層的に構成された情報にアクセスするための階層的インデクシング（HIERARCHICAL INDEXING FOR ACCESSING HIERARCHICALLY ORGANIZED INFORMATION IN A RELATIONAL SYSREM）」と題された米国特許出願番号０９／２

５１，７５７号に詳細に記載されており、この全内容をここに引用により援用する。
【００３６】
具体的には、「HIERARCHICAL INDEXING」の出願には、階層インデックスを作成、保持、および使用して、パス名に基づいて関係のあるシステム内の情報に効率的にアクセスすることにより、階層的に構成されたシステムをエミュレートするための技術が記載される。エミュレートされた階層システムに何らかの子を有する各アイテムは、そのインデックスにインデックスエントリを有する。インデックス中のインデックスエントリは、これらのインデックスエントリに関連付けられたアイテム中の階層的な関係を反映するような方法で互いにリンクされる。具体的には、２つのインデックスエントリに関連付けられたアイテム間に親子関係が存在すれば、親アイテムに関連付けられたインデックスエントリはその子アイテムに関連付けられたインデックスエントリへの直接のリンクを有する。
【００３７】
結果的に、パス名中のファイル名のシーケンスに従って、そのパス名におけるアイテムに関連付けられたインデックスエントリ間の直接のリンクに沿って進むことにより、パス名の導出（resolution）が行なわれる。インデックスエントリがこの態様でリンクされるインデックスを用いることにより、それらのパス名に基づいてアイテムにアクセスする処理は著しく加速され、また、その処理中に行われるディスクアクセスの数は著しく減少する。
【００３８】
階層インデックス
本発明と整合性のある階層インデックスは、パス名により特定されるように、親アイテムからそれらの子へ移動するという、階層システムのパス名に基づいたアクセス法をサポートする。一実施例によれば、本発明の原理に合う階層インデックスは、次の３つのフィールドを含むインデックスエントリを採用する。RowID、FileID、およびDir__entry__list（アレイとしてストアされる）。
【００３９】
図５は、データベース内の階層ストレージシステムをエミュレートするのに用いられ得る階層インデックス５１０を示す。図６は、階層インデックス５１０がエミュレートしている特定のファイル階層構造を示す。図７は、図６に示すファイルを関係データベース内にストアするのに用いられるファイルテーブル７１０を示す。
【００４０】
階層インデックス５１０はテーブルである。RowID欄はシステムにより生成されるＩＤを含み、データベースサーバ２０４がディスク上でその行の場所を突きとめ得るようにするディスクアドレスを特定する。この関係データベースシステムによると、RowIDは、ディスクドライブにストアされたデータの場所を突きとめるためにＤＢＭＳが用いる暗示的に規定されたフィールドであり得る。インデックスエントリのFileIDフィールドは、このインデックスエントリに関連付けられたファイルのFileIDをストアする。
【００４１】
本発明の一実施例によれば、階層インデックス５１０は、子を有するアイテムに対するインデックスエントリのみストアする。したがって、エミュレートされた階層ファイルシステムという面において、階層インデックス５１０にインデックスエントリを有するアイテムは、他のディレクトリに対して親であるディレクトリおよび／または現在ドキュメントをストアしているディレクトリのみである。子を有さないそれらのアイテム（たとえば、図６のExample.doc、Access、Appl、App2、App3）は含まれないのが好ましい。所与のファイルに対するインデックスエントリのDir__entry__listフィールドは、あるアレイ中に、所与のファイルの子ファイルの各々に対する「アレイエントリ」をストアする。
【００４２】
たとえば、インデックスエントリ５１２はWindows（Ｒ）ディレクトリ６１４に対するものである。Wordディレクトリ６１６およびAccessディレクトリ６２０はWindows（Ｒ）ディレクトリ６１４の子である。よって、Windows（Ｒ）ディレクトリ６１４に対するイン

デックスエントリ５１２のDir__entry__listフィールドは、Wordディレクトリ６１６に対するアレイエントリと、Accessディレクトリ６２０に対するアレイエントリとを含む。

【００４３】

一実施例によれば、Dir__entry__listフィールドが各子に対してストアする特定の情報は、その子のファイル名およびその子のFileIDを含む。階層インデックス５１０にそれら自体のエントリを有する子に対して、Dir__entry__listフィールドは子のインデックスエントリのRowIDもストアする。たとえば、Wordディレクトリ６１６は階層インデックス５１０にそれ自体のエントリを有する（エントリ５１４）。したがって、インデックスエントリ５１２のDir__entry__listフィールドは、ディレクトリ６１６の名称（“Word”）、階層インデックス５１０におけるディレクトリ６１６に対するインデックスエントリのRowID（“Ｙ３”）、およびディレクトリ６１６のFileID（“Ｘ３”）を含む。より詳細に説明するように、Dir__entry__listフィールドに含まれる情報により、パス名に基づいた情報へのアクセスがより速くより容易になる。

【００４４】

階層インデックスのいくつかの主要な原理は以下のとおりである。

・所与のディレクトリに対するインデックスエントリのDir__entry__list情報はできるだけ少数のディスクブロックとしてまとめて保たれる。これは、最も頻繁に用いられるファイルシステムオペレーション（パス名の導出、ディレクトリのリスティング（listing））は、あるディレクトリが参照されると常にそのディレクトリ内の多数のエントリを見る必要が生じることになるからである。言換えれば、特定のディレクトリエントリが参照されると同じディレクトリ内の他のエントリもまた参照されることが多いので、ディレクトリエントリは参照に対して高い局所性を有するべきである。

【００４５】

・階層インデックスのインデックスエントリにストアされる情報は、特定のディスクブロック中のエントリの最大数に適合するように、最小に保たれなければならない。ディレクトリエントリを、それらが含まれるディレクトリを特定するキーを反復する必要のないアレイ手段にまとめてグループ分けすると、ディレクトリ内のすべてのエントリが同じキーを共有することになる。

【００４６】

・パス名の導出に要する時間は、ファイルシステム内のファイルの総数ではなく、パス内のディレクトリの数に比例すべきである。これにより、ユーザは、頻繁にアクセスされるファイルをアクセス時間の少ないファイルシステムツリーの頂上の方に保つことが可能になる。

【００４７】

これらの要素はすべて、ｉノードおよびディレクトリのUNIX(R)システムなどの典型的なファイルシステムディレクトリ構造において存在する。ここに記載のような階層インデックスを用いることにより、それらの目的と、関係のデータベースが理解しかつ照会し得る構造とが一致し、データベースサーバが、パス名の導出に用いられたものとは別の態様でファイルのアドホックサーチを行なうことが可能になる。これを行なうためには、あるインデックスのデータベース概念を用いなければならない。すなわち、ある特定の方法（この場合、階層ツリーにおけるパス名の導出）を介したアクセスを最適化するよう設計された別の態様で別個のデータ構造に配置された下位情報（この場合ファイルデータ）の部分の複製である。

【００４８】

階層インデックスの使用

ファイルのパス名に基づいてファイルにアクセスするために階層インデックス５１０がいかに用いられ得るかについて、ここで図８のフローチャートを参照して述べることにする。説明の目的で、ドキュメント６１８がそのパス名を介してアクセスされると仮定する。このファイルのパス名は/Windows（Ｒ）/Word/Example.docであり、これは以下「入力パス名」と称する。このパス名が与えられると、パス名導出処理は、この入力パス名中の第

１の名称に対するインデックスエントリの場所を階層インデックス５１０において突きとめることにより開始する。あるファイルシステムの場合、パス名における第１の名称はルートディレクトリである。したがって、エミュレートされたファイルシステム内のファイルの場所を突きとめるためのパス名導出処理は、ルートディレクトリ６１０のインデックスエントリ５０８の場所を突きとめることにより始まる（ステップ８００）。すべてのパス名導出オペレーションがルートディレクトリのインデックスエントリ５０８にアクセスすることにより始まるので、ルートディレクトリ６１０（インデックスエントリ５０８）に対するインデックスエントリの場所を示すデータは、あらゆるサーチの開始時においてルートディレクトリのインデックスエントリ５０８の場所を素早く突きとめるために、階層インデックス５１０外部の都合よい場所に保持され得る。

【００４９】

ルートディレクトリ６１０に対するインデックスエントリ５０８の場所が一旦突きとめられると、ＤＢＭＳは、入力パス名中にまだ何らかのファイル名があるかを判定する（ステップ８０２）。入力パス名中にもうファイルがなければ、制御はステップ８２０へと進み、インデックスエントリ５０８にストアされたFileIDが用いられてファイルテーブル７１０中のルートディレクトリエントリを捜す。

【００５０】

この例では、ファイル名「Windows（Ｒ）」は、入力パス名においてルートディレクトリの記号「／」の後に続く。したがって、制御はステップ８０４へと進む。ステップ８０４で、次のファイル名（たとえば「Windows（Ｒ）」が入力パス名から選択される。ステップ８０６でＤＢＭＳはインデックスエントリ５０８のDir__entry__list欄を見て、選択されたファイル名に関するアレイエントリの場所を突きとめる。

【００５１】

この例では、入力パス名においてルートディレクトリの後に続くファイル名は「Windows（Ｒ）」である。したがって、ステップ８０６は、ファイル名「Windows（Ｒ）」のアレイエントリに対するインデックスエントリ５０８のDir__entry__listをサーチすることを伴う。Dir__entry__listが選択されたファイル名のアレイエントリを含まなければ、制御はステップ８０８からステップ８１０へと進み、ここで入力パス名が無効であることを示すエラーが生成される。この例では、インデックスエントリ５０８のDir__entry__listは「Windows（Ｒ）」のアレイエントリを含んでいる。したがって、制御はステップ８０８からステップ８２２へと移る。

【００５２】

インデックスエントリ５０８のDir__entry__list中の情報は、ルートディレクトリ６１０の子の１つが実際「Windows（Ｒ）」という名称のファイルであることを示す。さらに、Dir__entry__listアレイエントリはこの子についての次の情報を含む。すなわち、これはRowIDＹ２に一致するインデックスエントリであり、このFileIDはＸ２である。

【００５３】

ステップ８２２において、入力パス名にまだ何らかのファイル名があるか否かが判定される。もうファイル名がなければ、制御はステップ８２２からステップ８２０へと移る。この例では、「Windows（Ｒ）」は最後のファイル名ではないので、制御は代わりにステップ８２４へ移る。

【００５４】

「Windows（Ｒ）」は入力パス中の最後のファイル名ではないので、Dir__entry__listに含まれるFileID情報は、このパス導出オペレーション中には用いられない。むしろ、Windows（Ｒ）ディレクトリ６１４は特定されたパスの部分にすぎず、ターゲットではないので、ファイルテーブル７１０はこの時点では調べられない。代わりに、ステップ８２４で、インデックスエントリ５０８のDir__entry__list中に見つけられる「Windows（Ｒ）」に対するRowID（Ｙ２）が用いられて、Windows（Ｒ）ディレクトリ６１４に対するインデックスエントリの場所が突きとめられる（インデックスエントリ５１２）。

【００５５】

インデックスエントリ５１２のDir__entry__listを調べて、このシステムは、入力パス名中の次のファイル名をサーチする（ステップ８０４および８０６）。この例では、ファイル名「Word」が入力パス中でファイル名「Windows（Ｒ）」の後に続く。したがって、このシステムは「Word」のアレイエントリに対するインデックスエントリ５１２のDir__entry__listをサーチする。このようなエントリはインデックスエントリ５１２のDir__entry__list中に存在し、「Windows（Ｒ）」が実際「Word」という名称の子を有していることを示す（ステップ８０８）。ステップ８２２において、入力パス中にまだファイル名があると判定され、よって制御はステップ８２４へと進む。

【００５６】

「Word」に対するアレイエントリを見つけると、このシステムは、そのアレイエントリ中の情報を読出し、Wordディレクトリ６１６に対するインデックスエントリが階層インデックス５１０中のRowIDＹ３で見つかるということと、Wordディレクトリ６１６に属する特定の情報がファイルテーブル７１０中の行Ｘ３で見つかるということとを決定する。ワードディレクトリ６１６は単に特定されたパスの部分でありターゲットではないので、ファイルテーブル７１０は調べられない。その代わり、このシステムはRowID（Ｙ３）を用いてWordディレクトリ６１６に対するインデックスエントリ５１４の場所を突きとめる（ステップ８２４）。

【００５７】

階層インデックス５１０のRowIDＹ３で、このシステムはインデックスエントリ５１４を見つける。ステップ８０４において、入力パス名から次のファイル名「Example.doc」が選択される。ステップ８０６において、インデックスエントリ５１４のDir__entry__listがサーチされて、「Example.doc」に対するアレイエントリがあることを見つけ（ステップ８０８）、これは「Example.doc」がWordディレクトリ６１６の子であることを示す。このシステムはまた、Example.docが階層インデックス５１０においてインデックス付けの情報を全く有さないことと、Example.docに関する特定の情報をFileIDＸ４を用いてファイルテーブル７１０中に見つけることができるということも見つける。Example.docはアクセスされるターゲットファイル（すなわち入力パス中の最後のファイル名）であるので、制御はステップ８２０へ移り、ここでシステムはFileIDＸ４を用いてファイルテーブル７１０中の適切な行にアクセスし、かつその行の本体欄にストアされたファイル本体（ＢＬＯＢ）を抽出する。こうして、Example.docファイルがアクセスされる。

【００５８】

このファイルのアクセスには、階層インデックス５１０のみが用いられた。テーブルのスキャンは必要なかった。ブロックのサイズおよびファイル名の長さが典型的なものであれば、少なくとも６００のディレクトリエントリが１つのディスクブロックに適合することになり、典型的なディレクトリは６００エントリ未満を有する。つまり、所与のディレクトリ中のディレクトリエントリのリストは、典型的には単一のブロックに適合することになる。言換えれば、インデックスエントリのDir__entry__listアレイ全体を含む階層インデックス５１０の各インデックスエントリは、典型的には単一のブロックに適合することになり、したがって単一のＩ／Ｏオペレーションにおいて読出され得る。

【００５９】

階層インデックス５１０中のインデックスエントリからインデックスエントリへの移動において、インデックス中のさまざまなインデックスエントリが種々の異なるディスクブロックに存在する場合、ディスクアクセスをいくらか行なう必要があるという可能性もある。しかしながら、各インデックスエントリが単一のブロックに完全に適合すれば、ディスクアクセスの数は、そのパス中のディレクトリの数以下となる。平均のインデックスエントリのサイズが単一のディスクブロックに適合しなくとも、ディレクトリごとのディスクアクセスの数は一定の項（term）となり、ファイルシステム中のファイルの総数に伴って増加することはない。

【００６０】

いくつかのファイルシステムが所有する階層的特徴をエミュレートするための技術につい

ての以上の記述は単に例示的なものである。いくつかのファイルシステムおよびプロトコルの階層的な特徴をエミュレートするために他の技術も用いられ得る。さらに、階層的特徴を所有することさえないプロトコルもあり得る。このように、本発明は、いくつかのプロトコルの階層的特徴をエミュレートするための何らかの特定の技術に限定されることはない。さらに、本発明は、本質的に階層的であるプロトコルに限定されることもない。

【００６１】

データベースにおける他のＯＳファイルシステム特徴のエミュレート

ＯＳファイルシステムの階層的な構成以外に、ほとんどのＯＳファイルシステムの別の特徴は、それらがストアするファイルについて特定のシステム情報を保持していることである。一実施例によれば、このＯＳファイルシステム特徴もまた、データベースシステム内でエミュレートされる。具体的には、トランスレーションエンジン３０８が、あるファイルの「システム」データをデータベースサーバ２０４により管理されるファイルテーブル（たとえばファイルテーブル７１０）のある行にストアさせるコマンドを発する。一実施例によれば、ファイル内容のすべてまたは大部分が、その行のある欄に大規模バイナリオブジェクト（ＢＬＯＢ）としてストアされる。このＢＬＯＢの欄に加えて、このファイルテーブルはさらに、ＯＳファイルシステムで実現されるものに対応する属性値をストアするための欄を含む。このような属性値は、たとえば、ファイルの所有者または作成者、ファイルの作成日、ファイルの最終変更データ、ファイルへのハードリンク、ファイル名、ファイルサイズ、およびファイルタイプを含む。

【００６２】

トランスレーションエンジン３０８がデータベースサーバ２０４に対して何らかのファイルオペレーションを行なわせるようにデータベースコマンドを発する場合、それらのデータベースコマンドは、そのオペレーションに伴うファイルに関連付けられた属性を適切に変更させるステートメントを含む。たとえば、新たに作成されたファイルに対するファイルテーブル中に新たな行を挿入することに応答して、トランスレーションエンジン３０８はデータベースコマンドを発し、（１）誰がそのファイルを作成しているかをユーザに示す値をその行の「所有者」欄にストアし、（２）現在の日付を示す値をその行の「作成日」欄にストアし、（３）現在の日付および時刻を示す値を「最終変更」欄にストアし、（４）ＢＬＯＢのサイズを示す値を「サイズ」欄にストアする。このファイルにおける後続のオペレーションに応答して、これらの欄中の値はこれらのオペレーションにより要求されたとおり変更される。たとえば、トランスレーションエンジン３０８が特定の行にストアされたファイルの内容を変更するデータベースコマンドを発すると、同じオペレーションの部分として、トランスレーションエンジン３０８は、その特定の行の「最終変更」値を更新するデータベースコマンドを発する。さらに、この変更がファイルサイズを変えるものであれば、トランスレーションエンジン３０８は、その特定の行の「サイズ」値を更新するデータベースコマンドも発する。

【００６３】

ほとんどのＯＳファイルシステムの別の特徴は、各ファイルごとにセキュリティを提供する能力である。たとえば、Windows（Ｒ）ＮＴ、ＶＭＳおよびＵＮＩＸ（Ｒ）のいくつかのバージョンは、各ファイルに関してさまざまなエンティティが有する権利を示すアクセス制御リストを保持する。本発明の一実施例によれば、このＯＳファイルシステム特徴は、「セキュリティテーブル」を保持することによりデータベースシステム内でエミュレートされ、このセキュリティテーブルの各行は、アクセス制御リストのあるエントリと同様の内容を含む。たとえば、このセキュリティテーブル中の行がファイルを特定する値をストアするためのある欄と、許可タイプ（たとえば読出、更新、挿入、実行、変更の許可）を表わす値をストアするための別の欄と、その許可が与えられたか否かを示すフラグをストアする別の欄と、そのファイルに対する許可の所有者を表わす値をストアする所有者欄とを含む。この所有者とは、ユーザＩＤ（userid）で特定される単一のユーザでも、グループＩＤ（groupid）で特定されるグループでもよい。グループの場合は、１つ以上の追加テーブルを用いてそのグループＩＤをそのグループのメンバーのユーザＩＤにマッピン

グする。

【００６４】

データベースサーバ２０４により管理されるファイルテーブルにストアされたあるファイルにアクセスするデータベースコマンドを発する前に、トランスレーションエンジン３０８は、アクセスを要求しているユーザが特定されたファイルに対して要求されたアクセスのタイプを実行する許可を有することを検証するデータベースコマンドを発する。このようなプリアクセスデータベースコマンドにより、セキュリティテーブルからデータが検索され、アクセスを要求しているユーザにそのアクセスの実行が許可されているか否かが判定される。このように検索されたデータが、ユーザが要求された許可を有していないと示せば、トランスレーションエンジン３０８は要求されたオペレーションを実行するコマンドを発しない。その代わりに、トランスレーションエンジン３０８は要求の発生元のオペレーティングシステムへエラーメッセージを返す。このエラーメッセージに応答して、オペレーティングシステムは、アクセスを要求したアプリケーションに、そのアプリケーションがそのオペレーティングシステムのＯＳファイルシステムに保持されるあるファイルに許可なしにアクセスしようと試みた場合に送るであろうものと同じＯＳエラーメッセージを送る。このように、エラー状況下でも、データがＯＳファイルシステムではなく関係データベース中にストアされるという事実は、アプリケーションにはトランスペアレントである。

【００６５】

異なるオペレーティングシステムにはファイルについての異なるタイプのシステム情報がストアされる。たとえば、あるオペレーティングシステムは「アーカイブ」フラグをストアするがアイコン情報はストアしない場合もあり、また別のものはアイコン情報をストアしアーカイブフラグをストアしない場合もある。ここに記載の技術を実現するデータベースシステムにより保持されるシステムデータの特定のセットは、各実現例ごとに変化し得る。たとえば、データベース２０４はオペレーティングシステム３０４ａのＯＳファイルシステムによりサポートされるシステムデータのすべてをストアし得るが、オペレーティングシステム３０４ｂのＯＳファイルシステムによりサポートされるシステムデータはいくつかしかストアし得ない。これに代えて、データベースサーバはオペレーティングシステム３０４ａおよび３０４ｂの両者によりサポートされるシステムデータの全部をストアしてもよく、またはオペレーティングシステム３０４ａおよび３０４ｂのいずれか１つによりサポートされるシステムデータの一部をストアしてもよい。

【００６６】

図３に示すように、データベースサーバ２０４は多数の別々のＯＳファイルシステムから発生したファイルをストアする。たとえば、オペレーティングシステム３０４ａはオペレーティングシステム３０４ｂとは異なっていてもよく、また、オペレーティングシステム３０４ａおよび３０４ｂの両者がオペレーティングシステム１０４とは異なっていてもよい。ＯＳファイルシステム３０４ａおよび３０４ｂは対比する特徴を有し得る。たとえば、ＯＳファイルシステム３０４ａはファイル名に文字「／」を含むことを可能にし得るのに対し、ＯＳファイルシステム３０４ｂは可能にし得ない。一実施例によれば、このような状況において、トランスレーションエンジン３０８はＯＳファイルシステム特有の規則を実現するよう構成される。このように、アプリケーション３０２ａがファイル名に文字「／」を含むファイルをストアしようと試みると、トランスレーションエンジン３０８はデータベースサーバ２０４にそのオペレーションを実行させるデータベースコマンドを発する。一方、アプリケーション３０２ｂがファイル名に文字「／」を含むファイルをストアしようと試みると、トランスレーションエンジン３０８はエラーを生じさせる。

【００６７】

これに代えて、トランスレーションエンジン３０８は、すべてのオペレーティングシステムに対する規則の単一のセットを実現するよう構成され得る。たとえば、トランスレーションエンジン３０８は、ファイル名がトランスレーションエンジン３０８によりサポートされる１つのオペレーティングシステムにおいてさえも無効な場合には、たとえそのファ

イル名がそのファイル名を特定したコマンドを発したオペレーティングシステムにおいて有効であっても、エラーを生じさせることになる、という規則を実現し得る。

【００６８】

ＯＳファイルシステムコールのデータベースクエリへの変換

ＯＳファイルシステム特徴をデータベースシステム内でエミュレートするための機構を構築することにより、ＯＳファイルシステムのコールが、ＯＳファイルシステムのコールを行なっているアプリケーションにより予期される機能性を失うことなく、トランスレーションエンジン３０８によってデータベースクエリへ変換され得る。それらのアプリケーションによりなされたこのＯＳファイルシステムコールは、それらが実行されているオペレーティングシステムにより提供されるＯＳファイルＡＰＩを通じて行なわれる。たとえば、「Ｃ」プログラミング言語で書かれたプログラムに対しては、あるオペレーティングシステムのＯＳファイルＡＰＩのインターフェイスを特定するために「stdio.h」と題されたソースコードファイルが用いられる。このstdio.hファイルはアプリケーションに含まれるので、これらのアプリケーションはＯＳファイルＡＰＩを実現するルーチンをいかに呼出すかを知ることになる。

【００６９】

ＯＳファイルＡＰＩを実現する特定のルーチンはオペレーティングシステムごとに変化し得るが、典型的には次のオペレーションを行なうためのルーチンを含む。ファイルを開く、ファイルから読出す、ファイルへ書込む、ファイル内をシークする、ファイルをロックする、およびファイルを閉じる。一般に、これらのＩ／Ｏコマンドから関係データベースコマンドへのマッピングは、

ファイルを開く＝トランザクションを開始する、パスネームを導出しファイルを含む行の場所を突きとめる

ファイルへ書込む＝更新する

ファイルから読出す＝選択する

ファイルをロックする＝ファイルに関連付けられた行をロックする

ファイル内へシークする＝カウンタを更新する

ファイルを閉じる＝トランザクションを完遂させる（Windows（Ｒ）ＯＳファイルシステムプロトコルは、ファイルデータが書込まれる直前にディレクトリエントリが完遂するよう要求する。他のプロトコルは要求しない。）

以下により詳細に説明するように、ファイルの内容を受ける前であってもファイルの名称を可視にすることを予期するファイルシステムもある。これらのファイルシステムの関連で、「ファイルを開く」Ｉ／Ｏコマンドは、名称を書込むためのトランザクションの開始、名称を書込むためのトランザクションの完遂、および内容を書込むためのトランザクションの開始に対応する。

【００７０】

一実施例によれば、カウンタを用いてファイル内の「現在場所」が追跡される。ファイルがＢＬＯＢとしてストアされる実施例において、カウンタはＢＬＯＢの始めからオフセットの形態をとり得る。「ファイルを開く」コマンドが実行されると、カウンタが作成され、問題のＢＬＯＢの実行開始アドレスを示す値に設定される。ＢＬＯＢのカウンタはこの後データがＢＬＯＢから読出されるかまたはＢＬＯＢに書込まれることに応答して増分される。シークオペレーションは、このシークオペレーションのパラメータにより指示されたＢＬＯＢ内の場所を指すようにカウンタを更新させる。一実施例によれば、これらのオペレーションは、ノリ（Nori）他により１９９７年１０月３１日に出願され「ＬＯＢロケータ（LOB LOCATORS）」と題された米国特許出願番号０８／９６２，４８２号に記載されるようなＬＯＢロケータを用いることにより容易になり、この出願の全内容をここに引用により援用する。

【００７１】

いくつかのオペレーティングシステムにおいて、ＯＳのロックはファイルを閉じても続く場合がある。この特徴をエミュレートするためには、ロックファイルコマンドが、セッシ

ョンロックのリクエストに変換される。この結果、「トランザクションの完遂」がこのファイルを閉じるコマンドに応答して実行される場合、そのファイルに関連付けられた行におけるロックは自動的に解除されない。このように確立されたロックは、ファイルのロックを解除するコマンドに応答して明示的に、またはロックが得られたデータベースセッションの終了に応答して自動的に、のいずれかで解除される。

【００７２】

進行中のＩ／Ｏオペレーション

あるファイルが作成されると、そのファイルが作成されるディレクトリはそのファイルの存在を示すように更新される。いくつかのＯＳファイルシステムにおいて、新たなファイルを示すようにディレクトリを変更することは、新たなファイルが完全に生成される前に完遂される。それらのＯＳファイルシステム用に設計されたアプリケーションには、その特徴をうまく利用するものもある。たとえば、あるアプリケーションは第１のファイルハンドルで新たなファイルを開き、そのファイル中へのデータの書込みへと進み得る。データが書込まれている間、同じアプリケーションが第２のファイルハンドルでそのファイルを開くことができる。

【００７３】

この特徴をデータベース内でエミュレートすることには特殊な問題が伴う。というのは、一般に、データベーストランザクションが完遂するまで、別のトランザクションはそのトランザクションによりなされた変更を見ることができないからである。たとえば、第１のデータベーストランザクションが第１の「開く」コマンドに応答して開始されたとする。第１のトランザクションは特定のディレクトリ中にファイルが存在することを示すようにディレクトリテーブルを更新し、その後、ファイルを含む行を挿入するようにファイルテーブルを更新する。同じアプリケーションが発した第２の「開く」コマンドに応答して第２のデータベーストランザクションが開始されると、第２のデータベーストランザクションには、ディレクトリテーブルに対する変更も、ファイルテーブル中の新たな行も、第１のトランザクションが完遂するまで見えない。

【００７４】

本発明の一実施例によれば、作成進行中のファイルのディレクトリエントリを見る能力は、そのファイルに対する行をファイルテーブル中に挿入するのに用いるトランザクションとは別のトランザクションとしてディレクトリテーブルの更新を行なわせることにより、データベースシステム内でエミュレートされる。このように、第１の開くコマンドに応答して、トランスレーションエンジン３０８はデータベースコマンドを発し、（１）第１のトランザクションを開始させ、（２）新たなファイルの存在を示すようにディレクトリテーブルを変更し、（３）第１のトランザクションを完遂させ、（４）第２のトランザクションを開始させ、（５）このファイルのある行をファイルテーブル中に挿入し、（６）第２のトランザクションを完遂させる。ディレクトリテーブルに対する変更をファイルテーブルに対する変更とは別に完遂させることにより、第２の開くコマンドに応答して開始される第３のトランザクションは、ファイルテーブル中への挿入がまだ進行している間にディレクトリテーブル内のエントリを見ることができる。第２のトランザクションに失敗すれば、このディレクトリは内容を持たずにファイルのエントリとともに残されることになる。

【００７５】

トランスレーションエンジン

本発明の一実施例によれば、トランスレーションエンジン３０８は２つの層で設計される。これらの層は図４に示される。図４を参照して、トランスレーションエンジン３０８は、プロトコルサーバ層およびＤＢファイルサーバ４０８層を含む。ＤＢファイルサーバ４０８は、アプリケーションが代替的なＡＰＩ（ここではＤＢファイルＡＰＩと称す）を通じて、データベースサーバ２０４により管理されるデータベースにストアされたデータにアクセスできるようにする。ＤＢファイルＡＰＩは、ＯＳファイルＡＰＩとデータベースＡＰＩとの両方の局面を組合せる。具体的には、ＤＢファイルＡＰＩは、従来のＯＳファ

イルＡＰＩによりサポートされたものと同様のファイルオペレーションをサポートする。
【００７６】
しかしながら、ＯＳファイルＡＰＩとは異なり、ＤＢファイルＡＰＩはトランザクションのデータベースＡＰＩの概念を組入れる。すなわち、ＤＢファイルＡＰＩにより、アプリケーションが、ファイルオペレーションのセットが原子単位で実行されることを特定できるようになる。トランザクションが行なわれたファイルシステムを有することの利点について、以下により詳細に述べる。
【００７７】
ＤＢファイルサーバ
ＤＢファイルサーバ４０８は、ＤＢファイルＡＰＩコマンドをデータベースコマンドに変換するという役割を担う。ＤＢファイルサーバ４０８が受けたＤＢファイルＡＰＩコマンドは、トランスレーションエンジン３０８のプロトコルサーバ層から来るものであってもよく、または、ＤＢファイルＡＰＩを通じてコールを発することによりファイルオペレーションを行なうよう特に設計されたアプリケーション（たとえばアプリケーション４１０）から直接のものであってもよい。
【００７８】
一実施例によれば、ＤＢファイルサーバ４０８はオブジェクト指向である。このように、ＤＢファイルサーバ４０８により供給されるルーチンがあるオブジェクトのインスタンス生成により、またそのオブジェクトに関連付けられた方法をコールすることにより呼出される。ある実現例において、ＤＢファイルサーバ４０８は次の方法を含む「トランザクション」オブジェクトクラスを規定する。挿入、保存、更新、削除、完遂およびロールバック。ＤＢファイルＡＰＩは、外部エンティティがこのトランザクションオブジェクトクラスのインスタンス生成を行ない使用できるようにする、インターフェイスを提供する。
【００７９】
具体的には、外部エンティティ（たとえばアプリケーション４１０またはプロトコルサーバ）がＤＢファイルサーバ４０８のコールを行ないトランザクションオブジェクトのインスタンスを生成すると、ＤＢファイルサーバ４０８はデータベースサーバ２０４に新たなトランザクションを始めさせるデータベースコマンドを送る。この外部エンティティはこの後トランザクションオブジェクトの方法を呼出す。ある方法を呼出すことは、結果としてＤＢファイルサーバ４０８に対するコールとなる。ＤＢファイルサーバ４０８はこのコールに応答して、データベースサーバ２０４に対応のデータベースコマンドを発する。所与のトランザクションオブジェクトの方法の呼出に応答して行なわれるデータベースオペレーションはすべて、この所与のトランザクションオブジェクトに関連付けられたデータベーストランザクションの部分として行なわれる。
【００８０】
重要なことには、ある単一のトランザクションオブジェクトに対して呼出された方法は複数のファイルオペレーションを伴う場合がある。たとえば、アプリケーション４１０は以下のようにＤＢファイルサーバ４０８と対話し得る。アプリケーション４１０は、ＤＢファイルＡＰＩを通じてコールすることによりトランザクションオブジェクトＴＸＯ１のインスタンス生成を行なう。これに応答して、ＤＢファイルサーバ４０８は、データベースサーバ２０４内でトランザクションＴＸ１を開始するデータベースコマンドを発する。アプリケーション４１０はＴＸＯ１の更新方法を呼出し、データベースサーバ２０４により管理されるデータベース中にストアされたファイルＦ１を更新する。これに応答して、ＤＢファイルサーバ４０８はデータベースサーバ２０４に、要求された更新をトランザクションＴＸ１の部分として行なわせるデータベースコマンドを発する。アプリケーション４１０はＴＸＯ１の更新方法を呼出し、データベースサーバ２０４により管理されるデータベース中にストアされた第２のファイルＦ２を更新する。これに応答して、ＤＢファイルサーバ４０８は、データベースサーバ２０４に、要求された更新をトランザクションＴＸ１の部分として行なわせるデータベースコマンドを発する。この後アプリケーション４１０は、ＴＸＯ１の完遂方法を呼出す。これに応答して、ＤＢファイルサーバ４０８は、デ

ータベースサーバ２０４にＴＸ１を完遂させるデータベースコマンドを発する。ファイルＦ２への更新に失敗した場合、ＴＸＯ１のロールバック方法が呼出され、ファイルＦ１の更新を含む、ＴＸ１によりなされたすべての変更がロールバックされる。

【００８１】

ここではトランザクションオブジェクトを用いるＤＢファイルサーバを参照して技術が述べられてきたが、他の実現例も可能である。たとえば、ＤＢファイルサーバ内で、トランザクションではなくファイルを表わすのにオブジェクトを用いることもできる。このような実現例において、ファイルオブジェクトの方法を呼出すことにより、またオペレーションが実行されようとするトランザクションを特定するデータをそれへ渡すことにより、ファイルオペレーションが行なわれ得る。したがって、本発明は、オブジェクトクラスの何らかの特定のセットを実現するＤＢファイルサーバに限定されない。

【００８２】

説明の目的で、図４に表わす実施例は、データベースＡＰＩを通じてデータベースサーバ２０４と通信する処理実行外部データベースサーバ２０４としてＤＢファイルサーバ４０８を示す。しかしながら、代替的実施例によれば、ＤＢファイルサーバ４０８の機能性はデータベースサーバ２０４に組込まれている。ＤＢファイルサーバ４０８をデータベースサーバ２０４に組込むことにより、ＤＢファイルシステムの使用中に生成される処理間通信の量が減じられる。ＤＢファイルサーバ４０８をデータベースサーバ２０４に組込むことにより作り出されるデータベースサーバは、したがって、データベースサーバ２０４により管理されるデータにアクセスするための２つの代替的なＡＰＩを、すなわちＤＢファイルＡＰＩおよびデータベースＡＰＩ（ＳＱＬ）を提供する。

【００８３】

プロトコルサーバ

トランスレーションエンジン３０８のプロトコルサーバ層は、特定のプロトコルとＤＢファイルＡＰＩコマンド間での変換を行なうという役割を担う。たとえば、プロトコルサーバ４０６ａは、オペレーティングシステム３０４ａから受けたＩ／Ｏコマンドを、それがＤＢファイルサーバ４０８に送るＤＢファイルＡＰＩコマンドに変換する。プロトコルサーバ４０６ａはまた、ＤＢファイルサーバ４０８から受けたＤＢファイルＡＰＩコマンドを、それがオペレーティングシステム３０４ａに送るＩ／Ｏコマンドに変換する。

【００８４】

実際には、プロトコルとオペレーティングシステムとは１対１対応になっていない。むしろ、オペレーティングシステムの多くは１より多い数のプロトコルをサポートし、またプロトコルの多くは、１より多い数のオペレーティングシステムによりサポートされる。たとえば、単一のオペレーティングシステムが１つ以上のネットワークファイルプロトコル（ＳＭＢ、ＦＴＰ、ＮＦＳ）、Ｅメールプロトコル（ＳＭＴＰ、ＩＭＡＰ４）、およびウェブプロトコル（ＨＴＴＰ）に対して固有のサポートをもたらす場合もある。さらに、異なるオペレーティングシステムがサポートするプロトコルのセット間にはオーバーラップがよく起こる。しかしながら、例示の目的で、オペレーティングシステム３０４ａがあるプロトコルをサポートし、オペレーティングシステム３０４ｂが別のプロトコルをサポートするという、単純化された環境が示される。

【００８５】

Ｉ／Ｏ　ＡＰＩ

上述したように、Ｉ／ＯコマンドをＤＢファイルコマンドに変換するためにプロトコルサーバが用いられる。プロトコルサーバとそれらの通信相手のＯＳファイルシステムとの間のインターフェイスは、包括的にラベル付けされたＩ／Ｏ　ＡＰＩである。しかしながら、あるプロトコルサーバにより与えられた特定のＩ／Ｏ　ＡＰＩは、（１）プロトコルサーバの通信相手のエンティティおよび（２）プロトコルサーバがいかにそのエンティティに現れるようにするか、にともに依存する。たとえば、オペレーティングシステム３０４ａはMicrosoft Windows（Ｒ）ＮＴであってもよく、プロトコルサーバ４０６ａはMicrosoft Windows（Ｒ）ＮＴに対するデバイスドライバとして出現するよう設計されてもよい。

この状況下で、プロトコルサーバ４０６ａによってオペレーティングシステム３０４ａに提示されたＩ／Ｏ　ＡＰＩは、Windows（Ｒ）ＮＴにより理解されるデバイスインターフェイスのタイプとなるであろう。Windows（Ｒ）ＮＴは、何らかのストレージ装置と通信するのと同じように、プロトコルサーバ４０６ａと通信するとされる。プロトコルサーバ４０６ａにストアされたファイルおよびそこから検索されたファイルが実際にはデータベースサーバ２０４により保持されるデータベースにストアされまたそこから検索されているという事実は、Windows（Ｒ）ＮＴには完全にトランスペアレントである。

【００８６】

トランスレーションエンジン３０８により用いられるいくつかのプロトコルサーバがそれらのそれぞれのオペレーティングシステムにデバイスドライバインターフェイスを提示し得るのに対し、他のプロトコルサーバは他のタイプのエンティティとして出現し得る。たとえば、オペレーティングシステム３０４ａはMicrosoft Windows（Ｒ）ＮＴオペレーティングシステムであってもよく、プロトコルサーバ４０６ａはそれ自身をデバイスドライバとして提示するのに対し、オペレーティングシステム３０４ｂはMicrosoft Windows（Ｒ）９５オペレーティングシステムであって、プロトコルサーバ４０６ｂがそれ自身をシステムメッセージブロック（ＳＭＢ）サーバとして提示することもある。後者の場合、プロトコルサーバ４０６ｂは、典型的にはオペレーティングシステム３０４ｂとは別のマシン上で実行していることになり、オペレーティングシステム３０４ｂとプロトコルサーバ４０６ｂとの間の通信はネットワーク接続を介して起こることになる。

【００８７】

上記の例において、プロトコルサーバにより扱われるＩ／Ｏコマンドのソースはオペレーティングシステムである。しかしながら、トランスレーションエンジン３０８はＯＳファイルシステムのコマンドとともに用いることに限定されない。むしろ、ＤＢファイルコマンドと何らかのタイプのＩ／Ｏプロトコルとの間で変換を行なうために、プロトコルサーバが設けられ得る。ＯＳファイルシステムにより用いられるＩ／Ｏプロトコル以外にも、プロトコルサーバがそれに対して設けられ得る他のプロトコルとしては、たとえば、ファイル転送プロトコル（ＦＴＰ：File Transfer Protocol）や、電子メールシステム（ＰＯＰ３またはＩＭＡＰ４）により用いられるプロトコルが挙げられる。

【００８８】

ＯＳファイルシステムとともに働くプロトコルサーバにより提供されるインターフェイスが特殊なＯＳにより指示されるのと同様、非ＯＳファイルシステムとともに働くプロトコルサーバにより提供されるインターフェイスは、Ｉ／Ｏコマンドを発するであろうエンティティに基づいて変化することになる。たとえば、ＦＴＰプロトコルに従ってＩ／Ｏコマンドを受けるように構成されたプロトコルサーバは、ＦＴＰサーバのＡＰＩを提供するであろう。同様に、ＨＴＴＰプロトコル、ＰＯＰ３プロトコルおよびＩＭＡＰ４プロトコルに従ってＩ／Ｏコマンドを受けるように構成されたプロトコルサーバは、それぞれ、ＨＴＴＰサーバ、ＰＯＰ３サーバおよびＩＭＡＰ４サーバのＡＰＩを提供するであろう。

【００８９】

ＯＳファイルシステムと同様、非ＯＳファイルプロトコルの各々は、そのファイルに対して保持される特定の属性を予測する。たとえば、ほとんどのＯＳファイルシステムが、あるファイルの最終変更日を示すデータをストアするのに対し、電子メールシステムは各Ｅメールメッセージに対して、そのＥメールメッセージが読まれたか否かを示すデータをストアする。特定のプロトコルの各々に対するプロトコルサーバは、そのプロトコルのセマンティクスが確実にデータベースファイルシステム中でエミュレートされるために要求される論理を実現する。

【００９０】

トランザクション処理されたファイルシステム

データベースシステム内で、オペレーションは一般にトランザクションの部分として行なわれる。データベースシステムは、あるトランザクションの部分であるオペレーションのすべてを単一の原子操作（atomic operation）として行なう。すなわち、オペレーション

のすべてがうまく完了するか、またはオペレーションのいずれも実行されないか、のいずれかである。トランザクションの実行中、あるオペレーションが実行され得なければ、そのトランザクションの以前実行されたオペレーションがすべて取消されるか、または「ロールバックされる」。

【００９１】

データベースシステムとは対照的に、ＯＳファイルシステムはトランザクションに基づくものではない。したがって、大規模なファイルオペレーションに失敗すれば、そのオペレーションのうちその失敗以前に実行された部分は残る。不完全なファイルオペレーションを取消すことに失敗すると、ディレクトリ構造およびファイルの破損につながるおそれがある。

【００９２】

本発明の一局面によれば、トランザクション処理されたファイルシステムが設けられる。上述したように、トランスレーションエンジン３０８はＩ／Ｏコマンドをデータベースサーバ２０４に送られるデータベースステートメントへとコンバートする。特定されたＩ／Ｏオペレーションを実行するためにトランスレーションエンジン３０８によって送られた一連のステートメントは、トランザクション開始（begin transaction）ステートメントにより先行され、トランザクション終了（close transaction）ステートメントで終わる。結果として、データベースサーバ２０４によるそれらのステートメントの実行中に何らかの失敗が起これば、そのトランザクションの部分としてデータベースサーバ２０４によりなされる変更はすべてその失敗の時点までロールバックされることになる。

【００９３】

トランザクションの失敗を引起す事態は、Ｉ／Ｏコマンドの発生元のシステムに基づいて変化し得る。たとえば、ＯＳファイルシステムは署名の概念をサポートすることができ、ここであるファイルのソースを特定するデジタル「署名」はそのファイルに付加される。署名されたファイルをストアするために開始されたトランザクションは、たとえば、そのストアされているファイルの署名が予測どおりの署名でない場合、失敗するおそれがある。

【００９４】

オンザフライ・インテリジェントファイルのコンバート

本発明の一実施例によれば、ファイルは、関係データベースに挿入される前に処理され、それらがその関係データベースから検索されると再び処理される。図９はインバウンドおよびアウトバウンドのファイル処理を行なうのに用いられるＤＢファイルサーバ３０８の機能的構成要素を示すブロック図である。

【００９５】

図９を参照して、トランスレーションエンジン３０８は、レンダリングユニット９０４およびパーシングユニット９０２を含む。一般に、パーシングユニット９０２は、ファイルのインバウンド処理を行なう役割を担い、レンダリングユニット９０４はファイルのアウトバウンド処理を行なう役割を担う。これらの機能的ユニットの各々について、ここでより詳細に述べることにする。

【００９６】

インバウンドのファイル処理

インバウンドのファイルは、ＤＢファイルＡＰＩを介してＤＢファイルサーバ４０８に渡される。インバウンドのファイルを受取ると、パーシングユニット９０２はそのファイルのファイルタイプを特定し、その後、そのファイルタイプに基づいてそのファイルをパーシングする。パーシング処理中、パーシングユニット９０２はパーシングされているファイルから構造化された情報を抽出する。この構造化された情報とは、たとえば、パーシングされているファイルについての情報、またはこのファイルの論理的に別個の構成要素もしくはフィールドを表わすデータを含み得る。この構造化情報は、構造化情報の発生元のファイルとともにデータベース中にストアされる。その後、データベースサーバに対してクエリが出され、このように抽出された構造化情報が特定のサーチ条件を満たすか否か

に基づいてファイルを選択かつ検索し得る。

【００９７】

ある文書のパーシングを行なうためにパーシングユニット９０２により用いられる特定の技術、およびそれにより生成される構造化されたデータは、パーシングユニット９０２に渡された文書のタイプに基づいて変化することになる。したがって、何らかのパーシングオペレーションを行なう前に、パーシングユニット９０２はこの文書のファイルタイプを特定する。あるファイルのファイルタイプを決定するには、さまざまな要因が考慮され得る。たとえば、ＤＯＳまたはWindows（Ｒ）のオペレーティングシステムでは、あるファイルのファイルタイプは、そのファイルのファイル名中の拡張子により示されることが多い。すなわち、ファイル名が「．ｔｘｔ」で終わる場合、パーサユニット９０２はそのファイルをテキストファイルであると分類し、テキストファイル特有のパーシング技術をそのファイルに与える。同様に、ファイル名が「．ｄｏｃ」で終わる場合は、パーサユニット９０２はそのファイルをMicrosoft Word文書であると分類し、Microsoft Word特有のパーシング技術をそのファイルに与える。これに対して、Macintoshオペレーティングシステム（Macintosh Operating System）は、あるファイルに対するファイルタイプ情報をそのファイルとは別に保持される属性としてストアする。

【００９８】

あるファイルのファイルタイプを決定するためにパーシングユニット９０２が考慮し得る他の因子として、たとえば、そのファイルが位置付けられるディレクトリが挙げられる。したがって、パーサユニット９０２は、WordPerfect文書として＼WordPerfect＼文書ディレクトリにストアされるすべてのファイルを、それらのファイルのファイル名にかかわらず、分類およびパーシングするよう構成され得る。

【００９９】

これに代えて、インバウンドのファイルのファイルタイプとリクエスト元のエンティティが要求するファイルタイプとの両者が、ＤＢファイルサーバ４０８に対して提供される情報によって特定されるかまたはそれを通じて推定される場合もある。たとえば、あるウェブブラウザがメッセージを送る場合、そのメッセージは典型的にはそのブラウザについての情報（たとえばブラウザのタイプ、バージョンなど）を含む。あるウェブブラウザがＨＴＴＰプロトコルサーバを通じてあるファイルをリクエストする場合、この情報はＤＢファイルサーバ４０８に伝達される。この情報に基づいて、レンダリングユニット９０４はそのブラウザの能力（capability）についての情報を調べ、またそれらの能力から最良のファイルタイプを推定してブラウザへ運ぶこともできる。

【０１００】

上述したように、パーシングユニット９０２により用いられる特定のパーシング技術、およびこのように生成された構造化データのタイプは、パーシングされているファイルのタイプに基づいて変化することになる。たとえば、パーシングユニット９０２により生成された構造化データは、埋込メタデータ、導出（derived）メタデータおよびシステムメタデータを含み得る。埋込メタデータはファイル自体に埋込まれた情報である。導出メタデータは、ファイル内に含まれておらずそのファイルを解析することにより導出され得る情報である。システムメタデータは、ファイルの発生元のシステムにより提供されたファイルについてのデータである。

【０１０１】

たとえば、アプリケーション４１０がMicrosoft Word文書をパーシングユニット９０２に渡すとする。パーシングユニット９０２はその文書をパーシングしてそのファイル内に埋込まれているファイルについての情報を抽出する。Microsoft Word文書に埋込まれる情報としては、たとえば、文書の作者、文書が割当てられるカテゴリ、および文書についてのコメントを示すデータが挙げられる。

【０１０２】

Word文書についての埋込情報の場所を突きとめ抽出するのに加えて、パーサ９０２はその文書についての情報の導出もし得る。たとえば、パーサ９０２はこのWord文書をスキャン

してこの文書に含まれるページ数、段落数および語数を決定し得る。最後に、この文書が発生したシステムは、この文書のサイズ、作成日、最終変更日、およびファイルタイプを示すデータをパーシングユニット９０２に供給し得る。

【０１０３】

ある文書のファイルタイプがより構造化されるほど、この文書から構造化されたデータの特定のアイテムを抽出しやすくなる。たとえば、ＨＴＭＬ文書は典型的には、特定のフィールド（タイトル、ヘッディング１、ヘッディング２など）の初めと終わりとを特定するデリミタまたは「タグ」を有する。これらのデリミタはパーシングユニット９０２により用いられ、ＨＴＭＬ文書をパーシングすることによって、区切られたフィールドのいくつかまたはすべてに対してメタデータのアイテムをもたらし得る。同様に、ＸＭＬファイルは高度に構造化されており、ＸＭＬパーサはＸＭＬ文書中に含まれるフィールドのいくつかまたはすべてに対するメタデータの別のアイテムを抽出することができるであろう。

【０１０４】

一旦あるファイルに対してパーシングユニット９０２が構造化データを生成していれば、ＤＢファイルサーバ４０８はデータベースサーバ２０４にデータベースコマンドを発し、そのファイルをファイルテーブル（たとえばファイルテーブル７１０）のある行に挿入させる。一実施例によれば、このように発せられたデータベースコマンドはこのファイルをその行のある欄におけるＢＬＯＢとしてストアし、そのファイルについて生成された構造化データのさまざまなアイテムを同じ行の他の欄にストアする。

【０１０５】

これに代えて、あるファイルに対する構造化データアイテムのいくつかまたはすべてをファイルテーブルの外部にストアすることもできる。このような状況下で、あるファイルに関連付けられた構造化データをストアする行は、そのファイルを特定するデータを典型的に含むことになる。たとえば、Word文書がファイルテーブルの行Ｒ２０にストアされ、そのWord文書に対するシステムメタデータ（たとえば作成日、変更日など）がシステム属性テーブルの行Ｒ３４にストアされると想定する。このような状況において、ファイルテーブルのＲ２０およびシステム属性テーブルのＲ３４の両者が典型的にはWord文書に対する固有の識別子をストアするFileID欄を含むことになる。それから、クエリにより、ファイルテーブル中の行とシステム属性テーブル中の行とをFileID値に基づいて結合する結合ステートメントを発することで、そのファイルとそのファイルについてのシステムメタデータとの両方を検索することができる。ファイル「クラス」に関連付けられたテーブル中のファイル属性をストアするための技術について以下により詳細に説明する。

【０１０６】

アウトバウンドのファイル処理

アウトバウンドのファイルは、データベースサーバ２０４に送られたデータベースコマンドに応答して検索される情報に基づいてレンダリングユニット９０４により構築される。一旦構築されると、アウトバウンドのファイルはＤＢファイルＡＰＩを通じてそれをリクエストしたエンティティへと運ばれる。

【０１０７】

重要なことには、レンダリングユニット９０４により生じたアウトバウンドファイルのファイルタイプ（ターゲットファイルタイプ）は、必ずしもそのアウトバウンドファイルを構築するのに用いられたデータを生じたファイルと同じファイルタイプ（ソースファイルタイプ）でなくてもよい。たとえば、レンダリングユニット９０４はもともとデータベース内にWordファイルとしてストアされていたデータに基づいてテキストファイルを構築し得る。

【０１０８】

さらに、アウトバウンドファイルを要求するエンティティは、そのアウトバウンドファイルが構築されるもとのファイルを生じたエンティティとは全く別のプロトコルを用いて全く別のプラットフォーム上にあってもよい。たとえば、プロトコルサーバ４０６ｂがＩＭＡＰ４サーバインターフェイスを実現し、プロトコルサーバ４０６ａがＨＴＴＰサーバイ

ンターフェイスを実現する場合を想定する。これらの状況下で、Ｅメールアプリケーションから発生するＥメール文書はプロトコルサーバ４０６ｂを通じてデータベース内にストアされ、プロトコルサーバ４０６ａを通じてWebブラウザによりデータベースから検索され得る。この筋書きでは、パーシングユニット９０２がこのＥメールのファイルタイプ（たとえばＲＦＣ８２２）に関連付けられたパーシング技術を呼出し、レンダリングユニットがデータベースから検索されたＥメールデータからＨＴＭＬ文書を構築するレンダリングルーチンを呼出すであろう。

【０１０９】

パーサおよびレンダラの登録

上述したように、あるファイルに施されるパーシング技術はそのファイルのタイプにより指示される。同様に、あるファイルに施されるレンダリング技術はそのファイルのソースタイプとそのファイルのターゲットタイプとの両者により指示される。すべてのコンピュータプラットホームにわたって存在するファイルタイプの数は莫大である。したがって、すべての公知のファイルタイプを扱うパーシングユニット９０２を築くのも、ファイルタイプからファイルタイプへのすべての可能なコンバートを扱うレンダリングユニット９０４を築くのも実用的でない。

【０１１０】

本発明の一実施例によれば、ファイルタイプの急増により引き起こされる問題は、タイプ特有のパーシングモジュールをパーシングユニット９０２に登録可能にすることにより、またタイプ特有のレンダリングモジュールをレンダリングユニット９０４に登録可能にすることにより、対処される。タイプ特有のパーシングモジュールとは、ある特定のファイルタイプに対してパーシング技術を実現するモジュールのことである。たとえば、Word文書は、Word文書パーシングモジュールを用いてパーシングされるのに対し、ＰＯＰ３のＥメール文書はＰＯＰ３　Ｅメールパーシングモジュールを用いてパーシングされる。

【０１１１】

タイプ特有のパーシングモジュールと同様、タイプ特有のレンダリングモジュールとは、１つ以上のソースファイルタイプに関連付けられたデータを１つ以上のターゲットファイルタイプにコンバートするための技術を実現するモジュールのことである。たとえば、タイプ特有のレンダリングモジュールは、Word文書をテキスト文書にコンバートするために設けられ得る。

【０１１２】

ソースファイルタイプとターゲットファイルタイプとが同じであってもコンバートが必要となる場合もある。たとえば、パーシングされデータベース内に挿入されると、ＸＭＬ文書の内容は単一のＢＬＯＢには保持されず、多数のテーブルの多数の欄にわたって広がり得る。その場合は、たとえそのデータがもはやＸＭＬファイルとしてストアされていなくとも、ＸＭＬがそのデータのソースファイルタイプである。タイプ特有のレンダリングモジュールは、そのデータからＸＭＬ文書を構築するために設けられ得る。

【０１１３】

パーシングユニット９０２がインバウンドのファイルを受けると、パーシングユニット９０２はそのファイルのファイルタイプを決定し、タイプ特有のパーシングモジュールがそのファイルタイプに対して登録されているか否かを判定する。タイプ特有のパーシングモジュールがそのファイルタイプに登録されていれば、パーシングユニット９０２はそのタイプ特有のパーシングモジュールにより与えられたパーシングルーチンをコールする。それらのパーシングルーチンはインバウンドファイルをパーシングしてメタデータを生成し、このメタデータはこの後そのファイルとともにデータベース内にストアされる。タイプ特有のパーシングモジュールがそのファイルタイプに対して登録されていなければ、パーシングユニット９０２はエラーを生じるか、あるいは汎用のパーシング技術をそのファイルに施し得る。この汎用のパーシング技術にはファイルの内容についての知識が何もないので、そのファイルに対して生成できる有用なメタデータに関してこの汎用のパーシング技術が制限されることになる。

【０１１４】
レンダリングユニット９０４がファイルリクエストを受けると、レンダリングユニット９０４はデータベースコマンドを発し、そのファイルに関連付けられるデータを検索する。そのデータは、ファイルのソースファイルタイプを示すメタデータを含む。レンダリングユニット９０４はこの後、タイプ特有のレンダリングモジュールがそのソースファイルタイプに対して登録されているか否かを判定する。タイプ特有のレンダリングモジュールがそのソースファイルタイプに対して登録されていれば、そのタイプ特有のレンダリングモジュールにより与えられたレンダリングルーチンを呼出してファイルを構築し、こうして構築されたファイルを、そのファイルをリクエストしているエンティティへ与える。
【０１１５】
タイプ特有のレンダリングモジュールによってどのターゲットファイルタイプを選択すべきかを決定するためにさまざまな因子が用いられ得る。ファイルをリクエストしているエンティティが、それが要求するファイルのタイプを明示的に示す場合もある。たとえばテキスト編集者はテキストファイルのみ扱うことができる。テキスト編集者はソースファイルタイプがWord文書であるファイルをリクエストできる。このリクエストに応答して、Word特有のレンダリングモジュールが呼出され、これが、要求されたターゲットファイルタイプに基づいて、このWord文書をテキストファイルにコンバートする。このテキストファイルはその後テキスト編集者へ運ばれる。
【０１１６】
他には、ファイルをリクエストしているエンティティが多数のファイルタイプをサポートし得る場合もある。一実施例によれば、タイプ特有のレンダリングモジュールは、（１）リクエスト元のエンティティとタイプ特有のレンダリングモジュールとの両者によりサポートされるファイルタイプのセットを特定し、（２）そのセットにおいて最良のターゲットファイルタイプを選択する、という論理を組込む。この最良のターゲットファイルの選択には、問題のファイル特有の特徴を含む、さまざまな因子を考慮に入れることができる。
【０１１７】
たとえば、（１）ＤＢファイルサーバ４０８があるファイルに対するリクエストを受け、（２）そのファイルのソースファイルタイプがそのファイルは「ＢＭＰ」イメージであると示し、（３）このリクエストが「ＧＩＦ」、「ＴＩＦ」および「ＪＰＧ」イメージをサポートするエンティティにより開始されており、（４）ＢＭＰソースタイプ特有のレンダリングモジュールが「ＧＩＦ」、「ＪＰＧ」および「ＰＣＸ」のターゲットファイルタイプをサポートする、と仮定する。このような状況下で、ＢＭＰソースタイプ特有のレンダリングモジュールは、「ＧＩＦ」および「ＪＰＧ」の両者が可能性のあるターゲットファイルタイプであると決定する。これら２つの可能性のあるターゲットファイルタイプから選択するために、ＢＭＰソースタイプ特有のレンダリングモジュールは、そのファイルについての情報（その解像度および色の深みを含む）を考慮に入れ得る。この情報に基づいて、ＢＭＰソースタイプ特有のレンダリングモジュールはＪＰＧが最良のターゲットファイルタイプであると決定でき、このＢＭＰファイルをＪＰＧファイルにコンバートするよう進むことができる。その結果生じるＪＰＧファイルはこの後、リクエスト元のエンティティに運ばれる。
【０１１８】
一実施例によれば、タイプ特有のパーシングおよびレンダリングモジュールは、データベーステーブルにモジュールの能力を示す情報をストアすることにより登録される。たとえば、タイプ特有のレンダリングモジュールに対するエントリは、ソースファイルタイプがＸＭＬでありリクエスト元のエンティティがWindows（Ｒ）に基づいたWebブラウザである場合に用いるべきであることを示し得る。タイプ特有のパーシングモジュールに対するエントリは、ソースファイルタイプが.GIFイメージである場合にそれを用いるべきであるということを示し得る。
【０１１９】

ＤＢファイルサーバ４０８がＤＢファイルＡＰＩを通じてあるファイルに関係するコマンドを受けると、ＤＢファイルサーバ４０８は発生時のファイルタイプと、そのコマンドを発したエンティティのアイデンティティとを決定する。ＤＢファイルサーバ４０８はこの後データベースサーバ２０４にデータベースコマンドを発し、これによってデータベースサーバ２０４に、登録されたモジュールのテーブルをスキャンさせ、現状で用いるのに適切なモジュールを選択させる。インバウンドのファイルの場合、適切なパーシングモジュールが呼出され、データベースに挿入される前にファイルをパーシングする。アウトバウンドのファイルの場合、適切なレンダリングモジュールが呼出され、データベースから検索されたデータからアウトバウンドのファイルを構築する。

【０１２０】

本発明のある実施例によれば、ＤＢファイルシステムにより、オブジェクト指向技術を用いてファイルのクラスを規定することが可能になり、ここで各ファイルタイプは１つのファイルクラスに属し、ファイルクラスは他のファイルクラスからの属性を継承することができる。このようなシステムにおいて、あるファイルのファイルクラスはそのファイルに対する適切なパーサおよびレンダラを決定するのに用いられる因子となり得る。ファイルクラスの使用については以下により詳細に述べることにする。

【０１２１】

ストアドクエリディレクトリ

上記で説明したように、階層ディレクトリ構造は、各行が１つのファイルに対応するファイルテーブル７１０を用いて、データベースシステムにおいて実現され得る。特定されたファイルに関連付けられた行の場所をファイルのパス名に基づいて効率的に突きとめるために階層インデックス５１０が採用され得る。

【０１２２】

図５および図７に示す実施例において、各ディレクトリの子ファイルが明示的に列挙される。特に、各ディレクトリの子ファイルはそのディレクトリに関連付けられるインデックスエントリのDir__entry__listにおいて列挙される。たとえば、インデックスエントリ５１２はWindows（Ｒ）ディレクトリ６１４に対応し、インデックスエントリ５１２のDir__entry__listは「Word」および「Access」をWindows（Ｒ）ディレクトリ６１４の子ファイルとして明示的に列挙する。

【０１２３】

本発明の一局面によれば、いくつかまたはすべてのディレクトリの子ファイルが、明示的に列挙されるのではなく、ストアドクエリのサーチ結果に基づいて動的に決定される、ファイルシステムが提供される。このようなディレクトリはここではストアドクエリディレクトリと称す。

【０１２４】

たとえば、ファイルシステムのユーザが拡張子.docを有するすべてのファイルを単一のディレクトリにグループ分けしたいとする。従来のファイルシステムでは、ユーザはディレクトリを作成し、拡張子.docを有するすべてのファイルをサーチし、その後このサーチで見つかったファイルを新たに作成したディレクトリへ移動させるか、新たに作成したディレクトリとサーチで見つけたファイルとの間にハードリンクを作成するか、のいずれかを行なう。残念ながら、この新たに作成されたディレクトリの内容はサーチが行なわれた時点でのシステムの状態を正確に反映しているにすぎない。仮に、.doc拡張子を有さない名称に変えたとしても、フィールドはディレクトリ内に残ることになる。さらに、新規ディレクトリが確立された後に他のディレクトリ内に作成された.doc拡張子を有するファイルは、この新規ディレクトリには含まれない。

【０１２５】

新規ディレクトリのメンバーシップを統計的に規定するのではなく、このディレクトリのメンバーシップはストアドクエリにより規定され得る。拡張子.docを有するファイルを選択するストアドクエリは以下のように現われ得る。

【０１２６】

Ｑ１：
SELECT* from files__table
但し、
files__table.Extension = “doc”
図７を参照して、テーブル７１０に対して実行されると、クエリＱ１は、Example.doc」と題された２つの文書に対する行である、行Ｒ４および行Ｒ１２を選択する。

【０１２７】
本発明の一実施例によれば、クエリＱ１などのクエリを階層インデックス５１０におけるディレクトリエントリにリンクするための機構が設けられる。階層インデックス５１０の横断中、そのようなリンクを含むディレクトリエントリに遭遇すると、このリンクにより特定されるクエリが実行される。このクエリにより選択された各ファイルは、ちょうどそのファイルがディレクトリエントリをストアするデータベーステーブル中の明示的なエントリであったかのように、ディレクトリエントリに関連付けられるディレクトリの子として扱われる。

【０１２８】
たとえば、ユーザがWord６１６の子であるディレクトリ「Documents」の作成を望み、このドキュメントディレクトリが拡張子.docを有するすべてのファイルを含むことを望むとする。本発明の一実施例によれば、このユーザは、このディレクトリに属することになるファイルに対する選択条件を特定する、クエリを設計する。この例では、ユーザはクエリＱ１を生成し得る。このクエリはこの後データベースシステム内にストアされる。

【０１２９】
他のタイプのディレクトリと同様、Documentディレクトリに対する行がファイルテーブル７１０に加えられ、このDocumentディレクトリに対するインデックスエントリが階層インデックス５１０に加えられる。さらに、Wordディレクトリに対するインデックスエントリのDir__Entry__listは、新規なDocumentディレクトリがWordディレクトリの子であることを示すように更新される。Dir__Entry__listにおける子を明示的にリストするのではなく、このDocumentディレクトリに対する新規ディレクトリエントリは、ストアドクエリに対するリンクを含む。

【０１３０】
図１０および図１１はそれぞれ、Documentsディレクトリに対して適切なエントリが作成された後の階層インデックス５１０とファイルテーブル７１０との状態を示す。図１０を参照して、Documentsディレクトリに対してインデックスエントリ１００４が作成されている。Documentsディレクトリの子はストアドクエリの結果セットに基づいて動的に決定されるので、インデックスエントリ１００４のDir__entry__listフィールドはヌル（null）である。子ファイルを静的に列挙する代わりに、インデックスエントリ１００４は、Documentsディレクトリの子ファイルを決定するために実行されることになっているストアドクエリ１００２へのリンクを含む。

【０１３１】
Documentsディレクトリに対するインデックスエントリ１００４の作成に加えて、Wordディレクトリに対する既存のインデックスエントリ５１４は、DocumentsがWordディレクトリの子であることを示すように更新される。具体的には、インデックスエントリ５１４にDir__entry__listアレイエントリが加えられ、名称「Documents」、Documentsディレクトリに対するインデックスエントリのRowID（すなわちＹ７）、およびDocumentsディレクトリのFileID（すなわちＸ１３）を特定する。

【０１３２】
図示した実施例では、階層インデックス５１０に２つの欄が加えられている。具体的には、ストアドクエリディレクトリ（ＳＱＤ：Stored Query Directory）欄は、ディレクトリエントリがストアドクエリディレクトリに対するものであるかを示すフラグを含む。ストアドクエリディレクトリに対するディレクトリエントリにおいて、クエリポインタ（ＱＰ：Query Pointer）欄に、ディレクトリに関連付けられるストアドクエリへのリンクが記

憶される。ストアドクエリディレクトリ以外のディレクトリに対するディレクトリエントリにおいては、ＱＰ欄はヌルである。

【０１３３】

リンクの性質は各実現例ごとに変化し得る。たとえば、ある実現例によれば、このリンクは、ストアドクエリがストアされているストレージ場所に対するポインタであり得る。別の実現例によれば、このリンクは、単に、ストアドクエリテーブル中のストアドクエリを調べるのに用いられ得る固有のストアドクエリ識別子であり得る。本発明は、何らかの特定のタイプのリンクに限定されることはない。

【０１３４】

図１１を参照して、ここには、Documentsディレクトリに対する行（Ｒ１３）を含むように更新されたファイルテーブル７１０が図示される。一実施例によれば、従来のディレクトリに対して保持されたものと同じメタデータがDocumentsディレクトリに対してもまた保持される。たとえば、行Ｒ１３は、作成日、最終変更日などを含み得る。

【０１３５】

図１２はファイル階層構造のブロック図である。図１２に示す階層構造は図６のものと同じであるが、Documentsディレクトリ１２０２が加えられている。何らかのアプリケーションがDocumentsディレクトリ１２０２の内容の表示をリクエストすると、データベースはそのDocumentsディレクトリ１２０２に関連付けられたクエリを実行する。クエリは、このクエリを満足するファイルを選択する。このクエリの結果はその後、Documentsディレクトリ１２０２の内容としてアプリケーションに提示される。図１２に示された時点では、ファイルシステムはDocumentsディレクトリ１２０２に関連付けられたクエリを満足するファイルを２つしか含まない。これら２つのファイルはともにExample.docと題されている。したがって、これら２つのExample.docファイル６１８および６２２はDocumentsディレクトリ１２０２の子として示される。

【０１３６】

ＯＳファイルシステムの多くにおいて、同じディレクトリは同じ名称の２つの異なるファイルをストアできない。したがって、Documentsディレクトリ１２０２内にExample.docと題された２つのファイルがに存在すると、ＯＳファイルシステムの規則が破られるおそれがある。この問題に対処するためにさまざまな技術が用いられ得る。たとえば、ＤＢファイルシステムは各ファイル名に文字を付加して固有のファイル名を作ることができる。したがって、Example.doc６１８はExample.doc１として提示され得るのに対し、Example.doc６２２はExample.doc２として提示される。特定の情報を伝えない文字を付加するのではなく、意味を伝えるように付加文字を選択してもよい。たとえば、付加する文字により、そのファイルが静的に位置づけられるディレクトリへのパスを示してもよい。すなわち、Example.doc６１８はExample.doc__Windows（Ｒ）__Wordと表わすことができ、一方、Example.doc６２２はExample.doc__VMS__App４と表わされる。これに代えて、単にストアドクエリディレクトリにＯＳファイルシステムの規則を破らせることも可能である。

【０１３７】

図１０に示す実施例では、所与のディレクトリの子ファイルはすべて静的に規定されるか、またはすべてストアドクエリにより規定されるかのいずれかである。しかしながら、本発明の一実施例によれば、ディレクトリはいくつかの静的に規定された子ファイルと、ストアドクエリにより規定されたいくつかの子ファイルとを有し得る。たとえば、ヌルのDir__entry__listを有するのではなく、インデックスエントリ１００４は、１つ以上の子ファイルを静的に特定するDir__entry__listを有し得る。したがって、アプリケーションがデータベースシステムにDocumentsディレクトリの子を特定するように要請すると、データベースサーバは静的に規定された子ファイルとストアドクエリ１００２を満足する子ファイルとの集合をリストすることになる。

【０１３８】

重要なことには、あるディレクトリの子ファイルを特定するストアドクエリが他のディレクトリおよび文書を選択してもよい。そのような他のディレクトリのいくつかまたはすべ

ては、それら自体がストアドクエリディレクトリであり得る。ある状況下では、ある特定のディレクトリのストアドクエリがその特定のディレクトリ自体を選択し、そのディレクトリを自身の子にするという場合もある。

【０１３９】

ストアドクエリディレクトリの子ファイルはオンザフライで決定されるので、子ファイルのリスティングは常にデータベースの現状を反映するものとなろう。たとえば、「Documents」ストアドクエリディレクトリが上述のとおり作成されたとする。拡張子.docを有する新規ファイルが作成されるたびに、そのファイルは自動的にDocumentsディレクトリの子になる。同様に、あるファイルの拡張子が.docから.txtに変わると、そのファイルは自動的にDocumentsディレクトリの子としての資格をなくすことになる。

【０１４０】

一実施例によれば、ストアドクエリディレクトリに関連付けられたクエリは、ディレクトリの子ファイルとなる特定のデータベース記録を選択し得る。たとえば、「Employees（従業員）」と題されたディレクトリは、データベース内のEmployeesテーブルからすべての行を選択するストアドクエリにリンクされ得る。あるアプリケーションが仮想従業員ファイルのうちの１つの検索をリクエストすると、レンダラは対応の従業員記録からのデータを用いて、リクエストを出しているアプリケーションが予期するファイルタイプのファイルを生成する。

【０１４１】

ストアされたクエリ文書

ストアドクエリをディレクトリの子ファイルを特定するのに用いることができるのと同様に、ストアドクエリはまた、文書の内容を特定するのに用いることもできる。図７および図１１を参照して、これらの図はBody（本体）欄を有するファイルテーブル７１０を示している。ディレクトリに対し、Body欄はヌルである。文書に対し、Body欄は文書を含むBLOBを含む。ストアドクエリによって内容が特定されるファイルに対して、Body欄はストアドクエリに対するリンクを含んでいてもよい。アプリケーションがストアされたクエリ文書の検索を要求すると、ストアされたクエリ文書に関連付けられる行にリンクされたストアドクエリが実行される。文書の内容はそこで、クエリの結果のセットに基づいて構成される。ある実施例によれば、クエリ結果から文書を構成するプロセスは上述のようにレンダラによって行なわれる。

【０１４２】

ストアドクエリの結果によってその内容が完全に決められる文書に対するサポートをもたらすことに加えて、サポートはまた、ある部分はクエリの結果によって決められるが他の部分はそうではない文書に対してもたらされてもよい。たとえば、文書ディレクトリにおける行のBody欄はBLOBを含んでいてもよく、その際、別の欄はストアドクエリへのリンクを含む。その行に関連付けられるファイルに対するリクエストを受取った際、クエリは実行されてもよく、クエリの結果はファイルをレンダリングする際にBLOBと組合されてもよい。

【０１４３】

複数レベルのストアドクエリディレクトリ

上述のように、ストアドクエリはディレクトリの子ファイルを動的に選択するのに用いられてもよい。ディレクトリの子ファイルはすべて、ファイル階層構造における同じレベル（すなわち、ストアドクエリと関連付けられるディレクトリの真下のレベル）に属する。ある実施例によれば、あるディレクトリに関連付けられるストアドクエリはディレクトリの下の複数のレベルを規定し得る。複数のレベルを規定するクエリに関連付けられるディレクトリはここで、複数レベルのストアドクエリディレクトリと称するものとする。

【０１４４】

たとえば、複数レベルのストアドクエリディレクトリは、従業員テーブルにおけるすべての従業員記録を選択し、これらの従業員記録を部門および地域ごとにグループ分けするクエリと関連付けられていてもよい。これらの条件のもとで、各グループ分けキー（部門お

よび地域）および従業員記録に対して別個の階層レベルを設けてもよい。具体的には、このようなクエリの結果はファイル階層構造における３つの異なるレベルとして表わされてもよい。ディレクトリの子ファイルは第１のグループ分け基準によって定められる。この例においては、第１のグループ分け基準は「部門」(department)である。よって、ディレクトリの子ファイルは、さまざまな部門の値、すなわち「Dept1」、「Dept2」および「Dept3」であってもよい。これらの子ファイルはそれら自体がディレクトリとして表わされる。

【０１４５】

部門ディレクトリの子ファイルは第２のグループ分け基準によって定められることになる。この例においては、第２のグループ分け基準は「地域」(region)である。したがって、各部門ディレクトリは「North」、「South」、「East」、「West」などの地域値の各々に対する子ファイルを有することとなる。地域ファイルもまたディレクトリとして表わされる。最後に、各地域ディレクトリの子ファイルは、地域ディレクトリに関連付けられるある特定の部門／地域の組合せに対応するファイルとなる。たとえば、＼Dept1＼Eastディレクトリの子はEast地域におけるDepartment1における従業員であることとなる。

【０１４６】

ストアドクエリディレクトリの子ファイルに対する
ファイルオペレーションの取扱い

上記のように、ストアドクエリディレクトリの子ファイルは、従来のディレクトリの子ファイルと同様の態様でアプリケーションに対して示される。しかしながら、従来のディレクトリの子ファイルに対して行なわれ得るあるファイルオペレーションは、ストアドクエリディレクトリの子ファイルに対して行なわれると特殊な問題点を生じることとなる。

【０１４７】

たとえば、ユーザがストアドクエリディレクトリの子ファイルを別のディレクトリに移動すべきであることを特定する入力をしたと仮定する。子ファイルはディレクトリに関連付けられるストアドクエリにおいて特定される基準を満たしているという事実によってストアドクエリディレクトリに属しているため、このオペレーションは問題を生じる。ファイルがその基準をもはや満たさなくなるような態様でファイルを変更するのでなければ、そのファイルはストアドクエリディレクトリの子ファイルとしての資格を有し続けることになる。

【０１４８】

あるファイルをストアドクエリディレクトリに移動する試みがなされる際に同様の問題が生じる。そのファイルが既にストアドクエリディレクトリの子ではないのであれば、そのファイルはストアドクエリディレクトリに関連付けられるストアドクエリを満たさない。ストアドクエリにより特定される基準をファイルが満たすようにする態様でそのファイルを変更するのでなければ、そのファイルはストアドクエリディレクトリの子とされるべきではない。

【０１４９】

これらの問題点を解決するのにさまざまなアプローチをとることができる。たとえば、ファイルをストアドクエリディレクトリの中へまたはその中から移動することを試みるオペレーションに応答してエラーを出すようにＤＢファイルシステムを構成してもよい。代わりに、ＤＢファイルシステムはこのような試みに応答して、問題のファイル（またはファイルとして表わされているデータベース記録）を削除してもよい。

【０１５０】

さらに別のアプローチでは、ストアドクエリディレクトリの中へ移動されたファイルはこれらがディレクトリに関連付けられるストアドクエリの基準を満たすように自動的に変更されてもよい。たとえば、ストアドクエリディレクトリに関連付けられるストアドクエリが既婚のすべての従業員を選択しているものと仮定する。ある従業員記録に対応するファイルがそのストアドクエリディレクトリに移動されると、その従業員記録の「既婚」フィールドが更新され、その従業員が既婚であることを示す。

【０１５１】
同様に、ストアドクエリディレクトリの外へ移動されたファイルはこれらがディレクトリに関連付けられるストアドクエリの基準をもはや満たさないように自動的に変更されてもよい。たとえば、「既婚の従業員」のストアドクエリディレクトリにおけるファイルがそのディレクトリの外へ移動された場合、対応する従業員記録の「既婚」フィールドが更新されその従業員が既婚ではないことを示すようにする。
【０１５２】
ストアドクエリの基準を満たさないファイルを対応のストアドクエリディレクトリの中へ移動する試みがなされた場合、別のアプローチとしては、ストアドクエリディレクトリのインデックスエントリを更新してそのファイルをストアドクエリディレクトリの子として統計的に確立することが挙げられる。こうした状況のもとでは、ストアドクエリディレクトリは、ストアドクエリを満たしていることから子ファイルであるいくつかの子ファイルと、ストアドクエリディレクトリに手動で移動されたために子ファイルとなった他の子ファイルとを有することとなる。
【０１５３】
プログラム的に規定されたファイル
ストアドクエリディレクトリおよびストアされたクエリ文書はプログラム的に規定されたファイルの例である。プログラム的に規定されたファイルとは、ファイルシステムに対してファイルとして表わされたエンティティ（たとえば文書またはディレクトリ）であるが、その内容および／または子ファイルがコードを実行することによって定められるものである。ファイルの内容を定めるために実行されるコードとは、ストアされたクエリファイルの場合のようにストアされたデータベースクエリを含んでいてもよく、および／または他のコードを含んでいてもよい。一実施例によれば、プログラム的に規定されるファイルに関連付けられるコードは以下のルーチンを実現する。
【０１５４】
```
resolve_filename( filename):child_file_handle;
list_directory;
fetch;
put;
delete;
```
resolve_filenameルーチンは、「filename」の名を有しかつプログラム的に規定されたファイルの子であるファイルのファイルハンドル(file handle)を戻す。list_directoryルーチンは、プログラム的に規定されたファイルのすべての子ファイルのリストを戻す。fetchルーチンは、プログラム的に規定されたファイルの内容を検索する。putルーチンは、プログラム的に規定されたファイルの中へデータを挿入する。deleteルーチンは、プログラム的に規定されたファイルを削除する。
【０１５５】
一実施例によれば、「resolve_pathname(path):file_handle」ルーチンも提供される。resolve_pathnameルーチンはパスを受け、パスにおける各ファイル名(filename)に対してresolve_filename機能を反復的にコールする。
【０１５６】
一実施例によれば、ＤＢファイルシステムは、従来のファイル（すなわち、プログラム的に規定されていないファイル）に対する、上に列挙したルーチンを実現するオブジェクトクラスをもたらす。説明の目的で、そのオブジェクトクラスはここで「ディレクトリクラス」と称することにする。プログラム的に規定されたファイルを実現するため、ディレクトリクラスのサブクラスが確立される。そのサブクラスはディレクトリクラスのルーチンを継承するが、これらのルーチンの実現をプログラマがオーバーライドすることを可能にする。サブクラスによりもたらされる実現は、プログラム的に規定されたファイルにかかわるファイルオペレーションに応答してＤＢファイルシステムによって行なわれるオペレーションを決める。

【０１５７】
ファイルシステム内でのイベント通知
この発明の一局面によれば、あるファイルシステムイベントの発生の際にユーザが先を見越して(proactively)通知されるファイルシステムが提供される。これらは先を見越して通知されるため、関心のあるイベントが起こったことを示す条件を検出するため繰返されるポーリングのオーバーヘッドを引起さずにすむ。ファイルシステムイベントの発生の際に通知を受けるという能力は、たとえば、ユーザにとってある特定のファイルシステムイベントが重要な意味を有している場合などに非常に有用である。
【０１５８】
たとえば、ある文書の複数のコピーが異なる場所において維持され（「キャッシュされ」）、その文書に対してより効率のよいアクセスをもたらすようにすることは一般的である。こうした条件のもとで、そのコピーのうちの１つが更新された場合、残りのコピーは古くなってしまう（すなわち、これらのコピーはもはやその文書の現在の状態を反映していない）。以下説明するイベント通知手法を用いて、１つのコピーが更新された際に、他のコピーが存在するサイトではその更新について先を見越して通知を受けることができる。これらのサイトにおけるプロセスまたはユーザはそこで、その状況下で適当である何らかの処置をとることができる。キャッシュの場合、適当な処置とはたとえば、文書のキャッシュされたバージョンを更新されたバージョンで置き換えることであるかもしれない。
【０１５９】
別の例としては、ある特定のユーザがある会社の技術文書のすべてをそれらが出版される前に見直す責任がある場合がある。その会社のテクニカルライタは、すべての技術文書をそのユーザによる見直しのための準備が整った際に「見直し準備完了」ディレクトリの中へ記憶するように指示を受けているかもしれない。事前対応型の(proactive)通知システムがなければ、技術文書を「見直し準備完了」ディレクトリに単に記憶するだけでは新しい文書の見直しの準備ができたことをユーザに気づかせることにはならない。むしろ、テクニカルライタがそのユーザにその文書が見直されるための準備が整ったことを知らせるか、またはユーザが「見直し準備完了」ディレクトリを定期的にチェックするなどの何らかの追加の作業が必要となる。これに対し、ここに説明するイベント通知手法を実現するファイルシステムでは、技術文書を「見直し準備完了」ディレクトリの中へ入れるという行為により、新しい技術文書が見直しされる準備が整ったことをユーザに通知するためのユーザに対するメッセージの生成を引起すことができる。
【０１６０】
この発明の一実施例によれば、ファイルシステムイベントに対して先を見越してメッセージを生成するためのルールを定義付けてもよい。このようなイベントには、たとえば、ある特定のディレクトリにおけるファイルの記憶または作成、ある特定のディレクトリにおけるファイルの削除、ある特定のディレクトリからのファイルの移動、ある特定のファイルの変更または削除およびある特定のディレクトリへファイルをリンクすることなどが含まれる。これらのファイルシステムオペレーションは単に代表として表わすものである。事前対応型の通知ルールが作成され得る特定のオペレーションは実現例ごとに異なり得る。この発明はどんなある特定のセットのファイルシステムオペレーションに対してイベント通知サポートをもたらすことにも限定されない。
【０１６１】
一実施例によれば、event__idがファイルシステムイベントに割当てられる。そこで、あるevent__idおよび１以上の加入者の組を特定する通知ルールが作成されてもよい。あるルールがファイルシステムに一旦登録されると、ルールのevent__idによって識別されるファイルシステムイベントの発生に応答して、そのルールにおいて識別される消費者の組に自動的にメッセージが送られる。
【０１６２】
たとえば、あるユーザはいつファイルがある特定のディレクトリに追加されるかを知ることについて関心を登録するかもしれない。この関心を記録するため、データベースサーバ

は、（１）「登録ルール」テーブルの中に行を挿入し、（２）ディレクトリに関連付けられるフラグを設定して、少なくとも１つのルールがそのディレクトリに対して登録されたことを示す。登録されたルールのテーブルに挿入される行はエンティティを識別しそのエンティティが関心を持っているイベントを示す。行はまた、そのエンティティと通信するのに用いるべきプロトコルなどの追加の情報を含んでいてもよい。あるルールがディレクトリに当てはまることを示すフラグは、ディレクトリに関連付けられるファイルのテーブルの行において、またはディレクトリに関連付けられる階層インデックスエントリにおいて、またはその両方において記憶されてもよい。

【０１６３】

ファイルをディレクトリに挿入する際、データベースサーバはディレクトリと関連付けられるフラグを検査してそのディレクトリに対して何らかのルールが登録されているかどうかを判定する。そのディレクトリに対してルールが登録されている場合、登録されたルールのテーブルをサーチしてそのディレクトリに当てはまる特定のルールを見出す。登録されたルールがディレクトリに対して行なわれている特定のオペレーションに当てはまるルールを含んでいる場合、これらのルールに識別される関心を持っているエンティティにメッセージが送られる。エンティティに対してメッセージを送るのに用いられるプロトコルはエンティティごとに異なり得る。たとえば、あるエンティティに対しては、メッセージはＣＯＲＢＡを介して送られてもよく、その一方、他のエンティティに対しては、メッセージはＨＴＴＰを介するＨＴＭＬページの形で送られてもよい。

【０１６４】

一実施例によれば、通知機構は、その内容のすべてがここに引用により援用される、１９９７年１０月３１日にチャンドラ（Chandra）他によって提出された「データベースシステムにおけるメッセージ待ち行列のための装置および方法」（APPARATUS AND METHOD FOR MESSAGE QUEUING IN A DATABASE SYSTEM）と題された米国特許出願第０８／９６１，５９７号に記載される待ち行列機構のような待ち行列機構を用いて、上述のように、データベース実現型ファイルシステムとともに実現される。

【０１６５】

そのような実施例の１つによれば、データベースサーバの外部で実行されるイベントサーバがデータベースサーバによって管理される待ち行列に対して加入者として登録される。イベントサーバが加入する待ち行列はここでファイルイベント待ち行列と称することとする。ある特定のファイルシステムイベントに関心があるエンティティはその関心をイベントサーバに登録する。イベントサーバはデータベースＡＰＩを介してデータベースサーバと通信し、関心を持っているエンティティと、それらのエンティティがサポートするプロトコルを介して通信する。

【０１６６】

データベースサーバがファイルシステムに関連するオペレーションを行なう際、データベースサーバファイルイベント待ち行列の中に、オペレーションに関連付けられるevent__idを示すメッセージを入れる。待ち行列機構は、イベントサーバがファイルイベント待ち行列の中に関心を登録したことを判定し、イベントサーバにメッセージを送信する。イベントサーバは関心を持っているエンティティのリストをサーチしていずれかのエンティティがそのメッセージにおいて識別されるイベントに関心を登録していないかどうかを判定する。イベントサーバは次に、そのイベントに関心を登録したすべてのエンティティに対してファイルシステムイベントの発生を示すメッセージを送信する。

【０１６７】

関心を持っているエンティティへメッセージを転送するのにイベントサーバを用いる実施例において、イベントサーバはある特定の最大数のユーザをサポートするように構成されてもよい。関心を持っているユーザの数が最大数を超えた場合、追加のイベントサーバを開始して追加のユーザに対してサービスをもたらす。単一のイベントサーバのケースと同様に、複数イベントサーバのシステムにおける各イベントサーバはファイルイベント待ち行列への加入者として登録される。

【０１６８】
代替の実施例によれば、ファイルシステムイベントに関心を持っているエンティティはファイルイベント待ち行列への加入者として直接登録される。登録情報の一部として、エンティティはそれらが関心を持っているファイルシステムイベントのevent__idを示す。待ち行列機構がファイルイベント待ち行列の中にメッセージを入れる際、待ち行列機構はすべての待ち行列加入者に自動的にメッセージを送るわけではない。むしろ、待ち行列機構は登録情報を検査してどのエンティティがそのメッセージに関連付けられる特定のイベントに関心を登録しているかを判定し、それらのエンティティのみに選択的にメッセージを送る。データベースＡＰＩをサポートとしていないエンティティの場合、登録情報にはこれらのエンティティがサポートするプロトコルについての情報が含まれる。待ち行列機構はこれらのエンティティに対し、それらの登録情報にリストされているプロトコルを用いてファイルイベントメッセージを送信する。

【０１６９】
ファイルシステムイベント通知はさまざまなコンテキストにおいて適用され得る。たとえば、時には第１のマシンに第２のマシンに存在するファイルのキャッシュを記憶することが望ましいことがある。そのようなファイルキャッシュを実現する現在利用可能な機構の１つはMicrosoft Windows（Ｒ）オペレーティングシステムにより提供される「ブリーフケース」機能である。ブリーフケース機能により、ユーザがあるマシン上で特殊なフォルダ（「ブリーフケース」）を作成し、そのブリーフケースの中へ他のマシン上に記憶されているファイルをコピーすることが可能となる。各々のブリーフケースは「更新」オプションを有しており、これが選択されると、ブリーフケース内のファイルのコピーと元の場所にあるファイルのコピーとをファイルシステムに比較させる。もしファイルが同じ変更日を有していない場合、ファイルシステムはユーザがその２つのコピーを同期化するのを可能にする（典型的に、より新しいコピーをより古いコピーに上書きすることによって）。

【０１７０】
ブリーフケース機構とは異なり、ファイルシステムイベント通知機構は、ファイルキャッシュを先を見越して更新することを可能にし、これによってファイルキャッシュが常に、元の場所にあるファイルの現在の状態を反映するようにする。たとえば、ファイルキャッシュを管理するプロセスは、キャッシュに含まれているファイルの元のコピーに対する更新について関心を登録してもよい。これにより、元のファイルのいずれかが更新された際にはプロセスは自動的にこれを知らされることとなり、即刻これに応答して更新されたファイルをファイルキャッシュの中へコピーすることができる。同様に、ファイルシステムイベント通知機構を用いて、第１のマシン上に第２のマシン上に存在する１つ以上のディレクトリをミラー化してもよい。ファイルシステムイベント通知機構をこのような態様で用いるため、ミラー化された(mirrored)ディレクトリを維持するためのプロセスは最初にディレクトリおよびその中に含まれるすべてのファイルのコピーを作り、次に、ディレクトリおよびディレクトリに含まれるファイルに対して加えられた変更についてその関心を登録する。変更がディレクトリに加えられたことを知らされると、プロセスはディレクトリのコピーに対し対応する変更を加える。同様に、ミラー化されたディレクトリ内のファイルのいずれかに対する変更を知らされた際には、プロセスはファイルのコピーに対して対応する変更を加える。

【０１７１】
たとえば、ミラー化されたディレクトリからミラー化されていないディレクトリへファイルが移動された場合、プロセスはミラー化されたディレクトリからファイルのコピーを削除し、そのファイルについての関心の登録を解除する。したがって、プロセスはファイルが更新された際も引続き通知されるということはない。同様に、ミラー化されていないディレクトリからミラー化されているディレクトリへファイルが移動された場合、プロセスはディレクトリが変わったことを知らされることとなる。そのメッセージに応答して、プロセスは新しいファイルを識別し、ミラー化されたディレクトリ内に新しいファイルのコ

ピーを作り、その新しいファイルについてその関心を登録する。
【０１７２】
ファイルシステムにおけるバージョン管理
職場においては、大勢の人が長期間にわたってともに作業することになる大型の仕事を「プロジェクト」と称する。プロジェクトに取組んでいる際、社員は典型的に多数の文書を作成し、その各々は何らかの態様でそのプロジェクトに関係がある。
【０１７３】
同様に、コンピュータシステム内では、ユーザはしばしば、すべてがあるプロジェクトに関係のある多数の電子文書を作成する。たとえば世界中の多数のサイトに位置しているプログラマがそれぞれ、同じコンピュータプログラムの異なる部分に取組んでいるかもしれない。そのコンピュータプログラムに対して彼らが生成する電子文書は典型的にソースコードファイルを含むが、単一のプロジェクトに属する。すなわち、この議論の文脈では、プロジェクトとは関連するファイルの集まりのことである。
【０１７４】
典型的に、プロジェクトのファイルは特定のフォルダの中へ整理されることとなる。たとえば、図１３は、プロジェクト「Big Project」に関連するファイルがどのようにさまざまなフォルダに整理されているかの一例を示している。図１３を参照して、Big Projectと題されたフォルダ１３０２は、そのプロジェクトに関連するすべてのファイル（ディレクトリおよび文書）を保持するように作成されたものである。Big Project１３０２のすぐ下の子ファイルはフォルダsource code１３０４およびフォルダdocs１３０６である。source code１３０４は、ロサンゼルスに位置するプログラマのsource code１３１６およびsource code１３１８を記憶するためのLA code１３１２と、サンフランシスコに位置するプログラマのsource code１３２０を記憶するためのSF code１３１４との２つのディレクトリを含む。docs１３０６は、specs１３０８およびuser manual１３１０の２つのフォルダを含む。specs１３０８はspecs１３２２およびspecs１３２４を含む。user manual１３１０はＵＭ１３２６を含む。
【０１７５】
しばしば、あるプロジェクト内のファイルは同じプロジェクト内の他のファイルへの参照(reference)（たとえばＨＴＭＬリンク）を含んでいるであろう。これらの参照は典型的に、他の文書をその文書の完全なパス名を用いて識別している。したがって、文書がディレクトリ階層構造におけるある場所から別の場所へ移動された場合、あるいはその文書の名称が変更された場合、その文書へのすべての参照が無効となってしまう。
【０１７６】
文書間参照の存在により、ファイルの新しいバージョンは典型的に、それらが置換するより古いバージョンと同じ名前で同じ場所に記憶される。従来のファイルシステムでは、このプロセスによってファイルのより古いバージョンは上書きされ、これを回復するのが不可能になる。残念ながら、ファイルのより古いバージョンを回復することが望ましい場合は多々ある。たとえば、より新しいバージョンから重大な情報がうっかり削除されてしまったかもしれない。もしより古いバージョンを回復することが不可能であれば、ユーザはその失われた資料を再現するのに、それも再現できるのであればの話であるが、かなりのリソースを費やさなければならないかもしれない。さらに、多くの場合、ファイルに対する変更履歴を復元することが可能であったり、ある特定の変更がいつ加えられたものであるかを判断することが可能であったり、またはある時点で何が変更されたかを判断することが可能であることは望ましい。
【０１７７】
この発明の一局面によれば、ファイルの新しいバージョンがより古いバージョンを上書きすることなく、より古いバージョンと同じ名称を用いてディレクトリ階層構造における同じ場所に保存されるバージョニング(versioning)機構が提供される。より古いバージョンを上書きするのではなく、より古いバージョンは保持され、ユーザは選択的にファイルのより古いバージョンを検索することができる。さらに、より古いバージョンはディレクト

リ階層構造におけるそれらの元の場所において保持される。以下により詳細に説明するように、ファイルシステムがディレクトリ階層構造内の同じ場所において同じファイルの複数のバージョンを同じ名称で保持することを可能にする新規のディレクトリバージョニング手法が提供される。

【０１７８】

新しいバージョンの作成によって元のバージョンの名称または場所が変更されないため、ファイルの最初のバージョンに対するどんな参照も、ファイルのより新しいバージョンが作成された場合でもファイルの最初のバージョンを示し続けることとなる。したがって、文書内に含まれるファイル間参照は、参照された文書のより新しいバージョンが作成されたとしても、引続き参照された文書の正しいバージョンを指す。バージョニングプロセスにおいてファイル間参照が有効なままである（すなわち、引続き、参照されたファイルの正しいバージョンを参照する）という事実は、ファイル検索の効率にかなり有益な影響がある。具体的には、参照されたファイルの適切なバージョンを探すためにルックアップオペレーションを行なうことを必要とするのではなく、参照されたファイルは他のファイル内に含まれるそれらへの参照をたどることによって直接検索することができる。

【０１７９】

同様に、ある特定の時点におけるディレクトリの内容を判定するプロセスにルックアップオペレーションが関与する必要がない。ディレクトリはそれら自体がバージョン付け（versioned）されているため、ディレクトリのある特定のバージョンを選択することは単にディレクトリのメンバを選択することになる。ディレクトリの選択されたバージョンは、ディレクトリのそのバージョンに属する正しいファイルへの、よってファイルの正しいバージョンへの、直接リンクを含むことになる。

【０１８０】

また、バージョンごとにファイルの名称が変わる場合でも同じファイルのバージョン間の関係を追跡するための手法が提供される。以下により詳細に説明するように、ファイルの名称に加えて、各ファイルの各バージョンに対してFileIDおよびバージョンナンバが維持される。２つのファイルが同じFileIDを有している場合、それらは異なる名称を有していたとしても同じファイルの異なるバージョンである。

【０１８１】

この発明の一局面によれば、ユーザが見たいと思うプロジェクトの「ビュー」(view)をユーザが選択することを可能にする機構が提供される。プロジェクトのビューはある特定の時点において存在していた状態でのプロジェクトのファイルを表わす。たとえば、ユーザに提示されるデフォルトビューはすべてのファイルの最新のバージョンを表わしてもよい。別のビューでは、１日前の時点で最新であったファイルのバージョンを表わしているかもしれない。別のビューでは、一週間前の時点で最新であったファイルのバージョンを表わしていてもよい。

【０１８２】

一実施例によれば、あるプロジェクトにおける各ファイルとともにバージョンナンバを記憶することによってバージョン追跡機構が提供される。たとえば、ファイルテーブル７１０などの、ファイルテーブルを用いるデータベースシステムにおいて実現されるファイルシステムにおいて、あるファイルに関連付けられる行の１つの欄はそのファイルに対するバージョンナンバを記憶してもよい。ファイルが作成されるたびに、ファイルに対する行がファイルテーブル７１０の中に挿入され、予め定められた最初のバージョンナンバ（たとえば１）がその行のバージョン欄に記憶される。

【０１８３】

ファイルが更新されると、ファイルの前のバージョンは上書きされない。その代わり、ファイルの新しいバージョンのために新しい行がファイルテーブルに挿入される。新しいバージョンのための行には元の行と同じFileID、NameおよびCreation Dateが含まれるが、より高いバージョンナンバ（たとえば２）、新しいModification Dateおよび場合によっては異なるファイルサイズなどが含まれる。さらに、ファイルの内容を記憶するBLOBは更

新を反映することとなるが、元のエントリのBLOBは変わらない。

【０１８４】

一実施例によれば、ファイルとそのファイルが存在するディレクトリとがともにあるプロジェクトに属している場合、ファイルへの変更によってディレクトリの新しいバージョンが実効的に作成される。これにより、ディレクトリにおけるファイルの更新により、ファイルの新しいバージョンのためのファイルテーブルの行が作成されるだけでなく、ディレクトリの新しいバージョンのためのファイルテーブルの行も作成されることとなる。階層インデックスを用いる一実施例において、ディレクトリの新しいバージョンに対するインデックスエントリもまた階層インデックスに追加されることとなる。

【０１８５】

もしあるディレクトリと親ディレクトリとがともに同じプロジェクトに属しているなら、ディレクトリの新しいバージョンの作成によって親ディレクトリの新しいバージョンが実効的に作成される。これにより、ディレクトリの親ディレクトリに対するファイルテーブルおよび階層インデックスにも新しい行が追加される。このプロセスは続けられ、あるプロジェクトに属しかつファイル階層構造において更新されたファイルの上に存在するすべてのディレクトリに対して新しいバージョンが作られることとなる。

【０１８６】

バージョニング機構がプロジェクトに属するファイルの更新にどのように応答するかを示すため、図１３に示されるすべてのファイルがバージョン１であると仮定し、かつcode１３２０に対して更新が行なわれたと仮定する。図１４に示されるように、バージョニング機構はcode１３２０の元のバージョンを削除することなく新しいバージョンのcode１３２０′を作成することによって更新に応答する。code１３２０はSF codeディレクトリ１３１４に属し、そのため元のバージョンを削除することなく新しいバージョンのSF codeディレクトリ１３１４′が作成される。SF codeディレクトリ１３１４はsource codeディレクトリ１３０４に属するため新しいバージョンのsource codeディレクトリ１３０４′が元のバージョンを削除することなく作成される。最後に、source codeディレクトリ１３０４はbig projectディレクトリ１３０２に属するため、新しいバージョンのbig project１３０２′が元のバージョンを削除することなく作成される。

【０１８７】

図１４に示されるように、親ファイルの新しいバージョンが子ファイルの新しいバージョンに応答して作成される際、親ファイルの新しいバージョンは更新されたファイルの元のバージョンではなく、更新されたファイルの新しいバージョンが子であることを除いて、引続き更新前に有していたのと同じ子を有する。たとえば、新しいバージョンのcode１３２０′は新しいバージョンのSF code１３１４′の子である。新しいバージョンのSF code１３１４′は、新しいバージョンのsource code１３０４′の子である。しかしながら、元のsource code１３０４の変わらない子ファイル（たとえばLA code１３１２）は引続き新しいバージョンのsource code１３０４′の子ファイルであり続ける。同様に、新しいバージョンのsource code１３０４′は新しいバージョンのbig project１３０２′の子であるが、元のbig projectの変わらない子ファイル（たとえばdocs１３０６）はbig project１３０２の新しいバージョンの子ファイルであり続ける。

【０１８８】

ファイルシステムが階層インデックスを用いて実現される実施例では、ディレクトリの新しいバージョンに対して作成されたインデックスエントリは、更新された子ファイルに対するアレイエントリが子ファイルの新しいバージョンに対するアレイエントリで置換されることを除いて、ディレクトリの前のバージョンに対するインデックスエントリと同じDir__entry__listを含むことになる。更新された子ファイルが子ディレクトリであった場合、新しいディレクトリに対するDir__entry__listアレイエントリは、子ディレクトリの新しいバージョンに対するインデックスエントリの、階層インデックス内の、RowIDを含むこととなる。

【０１８９】

あるプロジェクトに属するファイルがそのプロジェクトにおける１つのディレクトリからそのプロジェクトにおける別のディレクトリに移動された場合、ファイルそのものは変更されていないため、ファイルの新しいバージョンは作成されない。しかしながら、ファイルが移動された元のディレクトリおよびファイルが入れられたディレクトリはともに変更されている。このため、これらのディレクトリおよび同じプロジェクトにあるこれらのディレクトリのすべての先祖ディレクトリに対して新しいバージョンが作成される。図１５は、LA code１３１２からSF code１３１４へ移動される図１３のcode１３１８に応答して作られることになる新しいディレクトリを示す。具体的には、新しいバージョンのLA code１３１２′およびSF code１３１４′が作成されることになる。新しいバージョンのLA code１３１２′はその子としてcode１３１８を有さない。むしろ、code１３１８は新しいバージョンのSF code１３１４′の子となる。新しいsource codeディレクトリ１３０４′が作成され、新しいバージョンのLA code１３１２′およびSF code１３１４′にリンクされる。新しいbig projectディレクトリ１３０２′が作成され、新しいsource code１３０４′および元のdocsディレクトリ１３０６にリンクされる。

【０１９０】

上述のバージョニング手法を用いて、あるプロジェクト（たとえばbig project１３０２）に対して変更が加えられる度にそのプロジェクトのルートディレクトリの新しいバージョンが作成される。ルートプロジェクトディレクトリの各バージョンから派生するリンクはある特定の時点においてそのプロジェクトに属していたすべてのファイルを互いにリンクし、このようにリンクされたファイルのバージョンはそのある特定の時点において存在していたバージョンである。たとえば、図１４を参照して、big project１３０２から派生するリンクはcode１３２０に対する更新の前に存在していた状態でのプロジェクトを反映している。big project１３０２′から派生するリンクは、code１３２０に対する更新の直後に存在していた状態でのプロジェクトを反映する。同様に、図１５において、big project１３０２から派生するリンクは、code１３１８をLA code１３１２からSF code１３１４へ移動する前に存在していた状態でのプロジェクトを反映する。big project１３０２′から派生するリンクは、code１３１８をLA code１３１２からSF code１３１４へ移動した直後に存在していた状態でのプロジェクトを反映する。

【０１９１】

タグ付け

残念なことに、上述のバージョニング手法により、特にプロジェクトのより上位のレベルにおけるディレクトリの、ファイルバージョンの大幅な急増が起きる。状況によっては、このような急増は必要ではなく望ましくもないかもしれない。したがって、この発明の一実施例によれば、ファイルのバージョンに「タグ付けする」ための機構が提供される。ファイルのバージョンのタグ付けによりファイルのそのバージョンを保持すべきであることを示す。すなわち、より新しいバージョンが作成される際にファイルのより古いバージョンを常に保持するのではなく、ファイルのより古いバージョンはタグ付けされている場合にのみ保持される。そうでなければ、これらはより新しいバージョンが作成される際に置換される（上書きされる）。

【０１９２】

図１３を参照して、code１３２０がタグ付けされていないものと仮定する。code１３２０が更新された場合、codeの新しいバージョンは単にcodeの古いバージョンで置換される。code１３２０がタグ付けされている場合にのみ、図１４に示されるように、code１３２０、SF code１３１４、source code１３０４およびbig project１３０２の別個の新しいバージョンが作られることとなる。

【０１９３】

多くの場合、タグはあるプロジェクト内のすべてのファイルに対して同時に適用されることになる。たとえば、あるソフトウェアプログラムのある特定のバージョンがリリースされた場合、プログラムのリリースされたバージョンを作成するのに用いられたすべてのソースコードはその時点でタグ付けされてもよい。これにより、ソースコードファイルへの

その後の改訂にかかわらず、リリースされたバージョンに関連付けられる正確に同じソースコードの組が後に参照するために利用可能となる。

【０１９４】

タグが常に全体としてのプロジェクトに適用される実施例において、単一のタグがルートプロジェクトディレクトリに対して維持されてもよい。タグ付けされているルートプロジェクトディレクトリのバージョンを用いてあるファイルの場所を確認する場合、そのファイルに対するいかなる変更もそのファイルの新しいバージョンを作成することにつながり、その一方でそのファイルの元のバージョンが保持される。逆に、タグ付けされていないルートプロジェクトディレクトリのバージョンを用いてファイルの場所が確認される場合、そのファイルに対するいかなる変更も単にファイルの前のバージョンを上書きすることになる。

【０１９５】

別の実施例によれば、タグをファイルに適用することはファイル階層構造においてそのファイルより下にあるすべてのファイルにタグを実効的に適用する。たとえば、タグがLA code１３１２に適用されるものと仮定する。code１３１８がLA code１３１２から外へ移動される場合、LA code１３１２の新しいバージョンが作成される。code１３１８が更新される場合、code１３１８およびLA code１３１２の双方の新しいバージョンが作成される。このような実施例において、すべてのタグ付きファイルを通してファイル階層構造をトラバースすることによってファイルの場所が確認される場合、そのファイルに対するどんな変更によってもファイルの新しいバージョンが作成されることになる。階層構造におけるタグ付けされたいずれのファイルもトラバースすることなくファイルの場所が確認される場合、そのファイルに対するどんな変更もそのファイルの前のバージョンを上書きすることとなる。

【０１９６】

パージカウント

タグ付けの代わりにまたはタグ付けに加えて用いることができるバージョンの急増を低減するための別の手法には、パージカウントを維持することが含まれる。パージカウントは、いずれかの所与のファイルに対して保持されることとなるバージョンの最大数を示す。既にバージョンの数がパージカウントに達しているファイルに対して新しいバージョンが作成される場合、そのファイルの新しいバージョンはそのファイルの保持される最も古いバージョンを上書きする。パージカウントはファイルごとのシステム、プロジェクトごとのベースまたはファイルごとのベースで実現されてもよい。ファイルごとのシステムのベースで実現される場合、単一のパージカウントがファイルシステムにおいて維持されるすべてのファイルに適用される。プロジェクトごとのベースでは、所与のプロジェクトにおけるすべてのファイルは同じパージカウントを有するが、異なるプロジェクトは異なるパージカウントを有し得る。ファイルごとのベースでは、各ファイルに対して異なるパージカウントが特定され得る。

【０１９７】

タグ付けと組合せて用いられる場合、パージカウント機構はさまざまな態様で実現され得る。一実施例によれば、タグ付けされたファイルは、ファイルの新しいバージョンを作成することによってパージカウントを超えることになるかどうかを判定する目的では無視され、タグ付けされたファイルはパージカウント機構によっては削除されることは決してない。たとえば、あるファイルに対するパージカウントが５であり、すなわちそのファイルの５つのバージョンが存在すると仮定し、かつこれらの５つのバージョンのうちの１つにタグ付けがされていると仮定する。そのファイルに対して更新がなされると、パージカウント機構は、現在そのファイルの既存のタグなしバージョンは４つしかないと判断し、よって、既存のバージョンのいずれをも削除することなくファイルの別のバージョンを作成する。同じファイルが再び更新された場合は、パージカウント機構は、ファイルの既存のタグなしバージョンは５つであると判定し、よって、新しいバージョンを作成することに応答してファイルの最も古いタグなしバージョンを削除する。

【０１９８】
プロジェクト間リンク
各リンクはソースファイル（そのリンクが拡張される元のファイル）およびターゲットファイル（そのリンクが指し示すファイル）を有する。ファイル階層構造において、リンクのソースファイルはしばしばディレクトリであり、リンクのターゲットファイルはディレクトリ内のファイルである。しかしながら、リンクのすべてがディレクトリとその子との間のものであるわけではない。たとえばＨＴＭＬファイルは、グラフィック画像および他のＨＴＭＬファイルへのハイパーリンクを含み得る。階層インデックスを用いて実現されるファイルシステムでは、これらのハイパーリンクはディレクトリ－文書間リンクと同様の態様で扱うことができる。

【０１９９】
ファイルシステムのビューにより、ファイルシステムにおける各プロジェクトがある特定の時点においてどのように存在していたかが示される。しかしながら、あるビューにおける１つのプロジェクトに関連付けられるその時点は、同じビューにおける別のプロジェクトに関連付けられる時点とは異なるかもしれない。このことにより、リンクのソースファイルがリンクのターゲットファイルとは異なるプロジェクトに属する場合に問題が生じる。たとえば、ビューが、ファイルＦ１を含むプロジェクトＰ１に対する時間Ｔ１とファイルＦ２を含むプロジェクトＰ２に対する後の時間Ｔ２とを特定すると仮定する。さらに、ファイルＦ２がファイルＦ１へのリンクを有すると仮定する。Ｆ２のＴ２バージョンに含まれるリンクはＰ１のＴ２バージョンへ行くのであり、Ｐ１のＴ１バージョンに行くのではない。しかしながら、そのビューはＰ１に対するＴ１を特定するため、そのビューを介してＰ１におけるいずれのファイルに対して行なわれるどんなオペレーションに対してもＰ１のＴ１バージョンが用いられるべきである。

【０２００】
この発明の一実施例によれば、各リンクに対して「プロジェクト間境界」フラグが維持される。リンクのプロジェクト間境界フラグは、そのリンクのソースファイルおよびターゲットファイルが同じプロジェクトにあるかどうかを示す。階層インデックス５１０などの階層インデックスを用いるファイルシステムにおいて、プロジェクト間境界フラグはたとえば、インデックスエントリのDir__entry__listの各アレイエントリに記憶されてもよい。

【０２０１】
ファイル階層構造のトラバース(traversal)において、すべてのリンクのプロジェクト間境界フラグはそのリンクをたどる前に検査される。あるリンクのプロジェクト間境界フラグが設定されている場合、ソース側ファイルが属しているプロジェクトの要求されるバージョン時間はターゲット側ファイルが属しているプロジェクトの要求されるバージョン時間と比較される。所望のバージョン時間が同じである場合、そのリンクはトラバースされる。所望のバージョン時間が同じではない場合、ターゲット側ファイルが属しているプロジェクトの要求されるバージョン時間に対応するターゲットファイルのバージョンを探してサーチが行われる。

【０２０２】
たとえば、Ｆ２とＦ１との間のリンクのプロジェクト間境界フラグが設定されることとなる。これにより、Ｐ２の要求されるバージョン時間とＰ１の要求されるバージョン時間とが比較される。Ｐ２の要求されるバージョン時間はＴ２であり、これはＰ１の要求されるバージョン時間であるＴ１と同じではない。したがって、Ｐ１はリンクをたどることによってその場所を確認することはできないであろう。その代わり、時間Ｔ１に対応するＰ１のバージョンの場所を確認するためにサーチが行われることとなる。

【０２０３】
代替の実施例によれば、プロジェクト間境界フラグは全く維持されない。代わりに、リンクに遭遇するたびに、ソースファイルの要求されるバージョン時間がターゲットファイルの要求されるバージョン時間と比較される。ソースファイルとターゲットファイルとが同

じプロジェクトにある場合、または同じ要求されるバージョン時間を有する異なるプロジェクトにある場合、そのリンクをたどる。そうでなければ、ターゲットファイルの正しいバージョンを探してサーチが行なわれる。

【０２０４】

オブジェクト指向ファイルシステム

近年、オブジェクト指向プログラミングが標準のプログラミング規範となっている。オブジェクト指向プログラミングでは、世界はオブジェクトの観点からモデル化される。オブジェクトとは、記録を操る手続きおよび機能と組合される記録である。あるオブジェクトクラスにおけるすべてのオブジェクトは同じフィールド（「属性」）を有し、同じ手続きおよび機能（「方法」）により操られる。オブジェクトはそれが属するオブジェクトクラスの「インスタンス」であるといわれる。

【０２０５】

ときおり、アプリケーションは、類似であるが同一ではないオブジェクトクラスの使用を必要とすることがある。たとえば、イルカと犬との両方をモデル化するのに用いられるオブジェクトクラスには鼻、口、長さおよび年齢の属性が含まれるかもしれない。しかしながら、犬オブジェクトクラスは毛色属性を必要とする一方、イルカオブジェクトクラスはひれの大きさの属性を必要とするかもしれない。

【０２０６】

あるアプリケーションが複数の類似の属性を必要とする状況におけるプログラミングを容易にするため、オブジェクト指向プログラミングでは「継承」をサポートする。継承がなければ、プログラマは犬オブジェクトクラスに対して１つのコードのセットを書き、イルカオブジェクトクラスに対して第２のコードのセットを書かなければならなくなる。双方のオブジェクトクラスに共通した属性および方法を実現するコードは双方のオブジェクトクラスに重複して現われることとなる。このような態様でコードが重複しているのは、特に、共通の属性および方法の数が独特の属性の数よりはるかに多い場合に非常に効率が悪い。さらに、オブジェクトクラス間のコード重複によりコードを改訂するプロセスが複雑になる。これは、その属性を有するすべてのオブジェクトクラス間で整合性を維持するために、共通の属性に対して加えられた変更はコードにおける複数の位置において複製されなければならないためである。

【０２０７】

継承により、オブジェクトクラス間に階層構造を確立することが可能となる。所与のオブジェクトクラスの属性および方法は、自動的に階層構造における所与のオブジェクトクラスに基づいたオブジェクトクラスの属性および方法となる。たとえば、「動物」オブジェクトクラスは関連付けられた方法とともに、鼻、口、長さおよび年齢属性を有するものとして定義付けられ得る。これらの属性および方法をイルカおよび犬オブジェクトクラスに追加するため、プログラマはイルカおよび犬オブジェクトクラスが動物オブジェクトクラスを「継承する」のを特定することができる。このような状況の下で、イルカおよび犬オブジェクトクラスは動物オブジェクトクラスの「サブクラス」であるといえ、動物オブジェクトクラスは犬およびイルカオブジェクトクラスの「親」クラスであるといわれる。

【０２０８】

この発明の一局面によれば、ファイルシステムに対して、継承を含むオブジェクト指向規範を適用するための機構が提供される。具体的には、ファイルシステムにおける各ファイルはあるクラスに属する。ファイルシステムのクラスは、とりわけ、ファイルシステムがそのファイルについて記憶している情報のタイプを定める。一実施例によれば、ベースクラスが設けられる。ファイルシステムのユーザはそこで他のクラスを登録してもよく、これはベースクラスまたはいずれかの前に登録したクラスのサブクラスとして定義付けられてもよい。

【０２０９】

新しいファイルクラスがファイルシステムに登録される際、ファイルシステムは新しいタイプのファイルおよび新しいタイプのファイルシステムとの対話をサポートするよう実効

的に拡張される。たとえば、ほとんどの電子メールアプリケーションは電子メール文書が「優先度」プロパティを有していることを期待する。ファイルシステムが優先度プロパティのための記憶をもたらしていない場合、電子メールアプリケーションはそのファイルシステムに記憶される電子メール文書に対して正しく動作しないかもしれない。同様に、あるオペレーティングシステムは、あるタイプのシステム情報がファイルとともに記憶されていることを期待するかもしれない。ファイルシステムがその情報を記憶していない場合、オペレーティングシステムは問題に遭遇し得る。ある特定のタイプのシステムまたはプロトコル（たとえば、特定のオペレーティングシステム、ＦＴＰ、ＨＴＰＰ、ＩＭＡＰ４など）をサポートするのに必要とされるすべての属性を含むクラスを登録することによって、そのシステムまたはプロトコルとの正確かつ透過的な対話が可能となる。

【０２１０】

クラスを登録するために、そのクラスについての情報がもたらされ、これはそのクラスの親クラスを識別し親クラスが有していない属性でそのクラスが有しているどんな属性をも記述するデータを含む。その情報はまた、そのクラスのインスタンスに対して動作する特定の方法を特定してもよい。

【０２１１】

ユーザがファイルクラスを登録することを可能にし、ファイルクラス間での継承をサポートし、ファイルが属するクラスに基づいてファイルについての情報を記憶するオブジェクト指向ファイルシステムは、ファイルシステムそのものが実現されるコンテキストに応じてさまざまな態様で実現され得る。一実施例によれば、オブジェクト指向ファイルシステムは上述のようにデータベース実現型ファイルシステムのコンテキストにおいて提供される。オブジェクト指向ファイルシステムのさまざまな局面をデータベース実現型の実施例に関連して説明するが、ここで説明するオブジェクト指向ファイルシステム手法はそのような実施例に限定されるものではない。

【０２１２】

オブジェクト指向ファイルシステムのデータベース実現

一実施例によれば、データベース実現型ファイルシステムはベースクラスを設けており、そのベースクラスのサブクラスをファイルシステムに登録することが可能である。図１６を参照して、ファイルクラスの例示的なセットが示される。ベースクラスは「Files」と題され、名称、作成日および変更日を含むすべてのファイルに一般的に共通である属性を含む。同様に、Filesクラスの方法には、すべてのファイルに対して行われ得るオペレーションのための方法が含まれる。

【０２１３】

一実施例によれば、Filesクラスの属性は、データベース実現型ファイルシステムがともに用いられることになるオペレーティングシステムによって維持されるすべての属性の合併である。たとえば、図３に示されるようにサーバ２０４によって維持されるデータベースにおいてファイルシステムが実現されていると仮定する。そのファイルシステムに記憶されるファイルはオペレーティングシステム３０４ａおよびオペレーティングシステム３０４ｂから生じたものであるが、これらのオペレーティングシステムは必ずしも同じファイル属性のセットをサポートするわけではない。このため、データベースサーバ２０４によって実現されるファイルシステムのFilesクラスの属性のセットは２つのオペレーティングシステム３０４ａおよび３０４ｂによってサポートされる属性のセットの合併となる。

【０２１４】

代替の実施例によれば、Filesクラスの属性はデータベース実現型ファイルシステムがともに用いられるオペレーティングシステムによって維持されるすべての属性の交差である。そのような実施例においては、Filesクラスのサブクラスを各オペレーティングシステムに対して登録することができる。所与のオペレーティングシステムに対して登録されたサブクラスは、ベースのFilesクラスに既に含まれていない所与のオペレーティングシステムによってサポートされる属性のすべてを追加することによってベースのFilesクラス

を拡張することとなる。

【０２１５】

図１６に例示される実施例では、「Document」クラスおよび「Folder」クラスの、Filesの２つのサブクラスが登録されている。DocumentクラスはFilesクラスの属性および方法のすべてを継承し、かつ文書ファイルに特有の属性を追加する。例示される実施例では、Documentクラスは属性「サイズ」を追加する。

【０２１６】

Folderクラスは、Filesクラスの属性および方法のすべてを継承し、フォルダファイル（すなわち、他のファイルを含むことが可能である、ディレクトリなどのファイル）に特有である属性および方法を追加する。例示される実施例では、Folderクラスは新しい属性「max__children」および新しい方法「dir__list」を導入している。max__children属性はたとえば、所与のフォルダ内に含まれ得る子ファイルの最大数を示していてもよい。dir__list方法はたとえば、所与のフォルダの子ファイルのすべてのリストを提供するようにしてもよい。

【０２１７】

図１６に例示されるクラス階層構造では、Documentクラスは、e-mailおよびTextの２つの登録されたサブクラスを有する。これらのサブクラスは両方ともDocumentクラスの属性および方法のすべてを継承する。さらに、e-mailクラスは、Read__flag、優先度および送信者の３つの追加のプロパティを含む。Textクラスは１つの追加の属性であるCR__Flagと追加の方法Typeとを有する。CR__Flagは、テキスト文書が「復帰」(carriage return)記号を含むかどうかを示すフラグであってもよい。Type方法は、コンピュータモニタなどの入出力デバイスへテキスト文書を出力する。

【０２１８】

ファイルクラスおよびファイル形式

ファイルの内部構造はファイルの「形式」と称される。典型的に、ファイルの形式はファイルを作成するアプリケーションにより決められる。たとえば、あるワードプロセッサにより作成された文書は別のワードプロセッサによって作成された別の文書と同じ意味内容を有していても、全く異なる形式を有しているかもしれない。いくつかのファイルシステムでは、文書形式とファイル名拡張子との間にマッピングが維持されている。たとえば、.docで終わるファイル名を有するすべてのファイルはある特定のワードプロセッサにより作成されたファイルであると推定され、よって、そのワードプロセッサによって強いられる内部構造を有するものと推定される。他のファイルシステムでは、文書の形式についての情報はその文書に関連付けられる別個のメタファイルにおいて維持される。

【０２１９】

ファイル形式とは対照的に、ここに説明するファイルクラス機構は文書の内部構造に関連しない。むしろ、ファイルのファイルクラスはファイルシステムがそのファイルに対してどんな情報を維持するか、かつファイルシステムがファイルにどんなオペレーションを行なえるかを決める。たとえば、多数のワードプロセッサによって作成された文書はすべてDocumentクラスのインスタンスであり得る。このため、ファイルシステムは文書の内部構造が完全に異なっていても、文書について同じ属性情報を維持し、文書に対して同じオペレーションを行なうことを可能にする。

【０２２０】

クラステーブル

一実施例によれば、オブジェクト指向ファイルシステムは、ファイルの各クラスに対して関係テーブルが作成される関係データベースシステムにおいて実現される。図１７は、図１６に例示されるクラスに対して作成され得るテーブルの一例である。具体的には、Filesテーブル１７０２、documentテーブル１７０４、E-mailテーブル１７０６、Textテーブル１７０８およびFolderテーブル１７０８はそれぞれ、Filesクラス、Documentクラス、E-mailクラス、TextクラスおよびFolderクラスに対応する。

【０２２１】

一実施例によれば、所与のクラスに対するクラステーブルは、（１）その所与のクラスに属するファイルおよび（２）その所与のクラスのいずれかの子孫(descendant)クラスに属するファイルのための行を含む。たとえば、例示されるシステムにおいて、Filesクラスはベースクラスである。したがって、ファイルシステムにおけるすべてのファイルはFilesクラスまたはその子孫クラスのメンバとなる。したがって、Filesテーブルはファイルシステムにおけるすべてのファイルに対する行を含むこととなる。一方、E-mailクラスおよびTextクラスはDocumentクラスの子孫であるが、FilesクラスおよびFolderクラスはそうではない。したがって、Documentテーブル１７０４は、クラスDocument、E-mailまたはTextのすべてのファイルに対する行を含むが、クラスFilesまたはFolderのものであるファイルに対する行は含まない。

【０２２２】

各クラスに対するテーブルは、そのクラスにより導入された属性に対する値を記憶する欄を含む。たとえば、DocumentクラスはFilesクラスの属性を継承し、これらの属性にサイズ属性を追加する。したがって、Documentテーブルには、サイズ属性に対するサイズ値を記憶するための欄が含まれる。同様に、E-mailクラスはDocumentクラスの属性を継承し、read_flag、優先度および送信者属性を導入する。したがって、E-mailテーブル１７０６には、read_flag値、優先度値および送信者値を記憶するための欄が含まれる。

【０２２３】

図１７に示されるファイルシステムにおいて５つのファイルが記憶されている。File1と名づけられたファイルはFilesテーブル１７０２におけるRowID X1に記憶される。File1のFileIDはＦ１である。File1のクラスはFileクラスであり、これは行Ｘ１のClass欄に記憶される値によって示されるとおりである。File1はFilesクラスのインスタンスであるため、Filesテーブル１７０４はFile1に対する情報を含む唯一のクラステーブルである。したがって、File1に対して記憶される唯一の属性値はFilesクラスに関連付けられる属性に対する値である。

【０２２４】

File2と名付けられたファイルはFilesテーブル１７０２におけるRowID X2に記憶される。File2のFileIDはＦ２である。File2のクラスはDocumentクラスであり、これは行Ｘ２のClass欄に記憶される値によって示されるとおりである。File2はDocumentクラスのインスタンスであるため、Filesテーブル１７０２およびDocumentテーブル１７０４はFile2に対する情報を含む。すなわち、File2に対して記憶される属性値は、Filesクラスから継承された属性を含む、Documentクラスと関連付けられる属性に対する値である。

【０２２５】

File3と名付けられるファイルはFilesテーブル１７０２におけるRowID X3に記憶される。File3のFileIDはＦ３である。File3のクラスはE-mailクラスであり、これは行Ｘ３のClass欄に記憶される値によって示されるとおりである。File3はE-mailクラスのインスタンスであるため、Filesテーブル１７０２、Documentテーブル１７０４およびE-mailテーブル１７０６はすべてFile3に対する情報を含む。すなわち、File3に対して記憶される属性値は、DocumentクラスおよびFilesクラスから継承された属性を含む、E-mailクラスに関連付けられる属性に対する値である。

【０２２６】

File4と名付けられたファイルはFilesテーブル１７０２におけるRowID X4に記憶される。File4のFileIDはＦ４である。File4のクラスはTextクラスであり、これは行Ｘ４のClass欄に記憶される値によって示されるとおりである。File4はTextクラスのインスタンスであるため、Filesテーブル１７０２、Document１７０４およびTextテーブル１７０８はFile4に対する情報を含む。すなわち、File4に対して記憶される属性値は、DocumentクラスおよびFilesクラスから継承された属性を含む、Textクラスに関連付けられる属性に対する値である。

【０２２７】

File5と名付けられたファイルはFilesテーブル１７０２におけるRowID X5に記憶される。

File5のFileIDはＦ５である。File5のクラスはFolderクラスであり、これは行Ｘ５のClass欄に記憶される値によって示されるとおりである。File5はFolderクラスのインスタンスであるため、Filesテーブル１７０２およびFolderテーブル１７０８はFile5に対する情報を含む。すなわち、File5に対して記憶される属性値は、Filesクラスから継承される属性を含む、Folderクラスに関連付けられる属性に対する値である。

【０２２８】

この発明の一実施例によれば、クラステーブル内のファイルは上に図５および図８に関連して説明したように階層インデックスをトラバースすることによってアクセスされる。階層インデックスのトラバースにより（パス名導出において行われるように）、ターゲットファイルに対応するFilesテーブル１７０２内の行のRowIDが生成される。その行から、Filesクラス属性に対する属性値が検索される。しかしながら、他のクラスに属するファイルに関しては、追加の属性は他のクラステーブルから検索されなければならないかもしれない。たとえば、File3に対し、作成日および変更日はFilesテーブル１７０２の行Ｘ３から検索され得る。しかしながら、File3のサイズを検索するには、Documentテーブル１７０４の行Ｙ２にアクセスしなければならない。File3に対する優先度情報を検索するには、E-mailテーブル１７０６の行Ｑ１にアクセスしなければならない。

【０２２９】

あるファイルに属するさまざまな属性値の検索を容易にするため、これらの属性を含む行は互いにリンクされる。例示される実施例では、リンクは「Derived RowID」とラベル付けされた欄に記憶される。ある特定のクラスに対するテーブルにおけるある特定のファイルに対する行のDerived RowID欄において記憶される値は、そのある特定のクラスのサブクラスに対するテーブルに存在するそのある特定のファイルに対する行を指し示す。たとえば、File3に対するFilesテーブル行Ｘ３のDerived RowID欄は値Ｙ２を含む。Ｙ２はDocumentテーブル１７０４におけるFile3に対する行のRowIDである。同様に、Document行Ｙ２のDerived RowID欄は値Ｑ１を含む。Ｑ１はE-mailテーブル１７０６におけるFile3に対する行のRowIDである。

【０２３０】

例示される実施例では、ある特定のファイルに対する行間のリンクは片方向であり、親クラスに対するテーブルにおける行からサブクラスのテーブルにおける行へ行く。これらの片方向リンクにより、ベーステーブル（すなわちファイルテーブル）における行から始まるサーチが容易となるが、これはほとんどの条件下で当てはまる。しかしながら、サーチの開始点が別のテーブルの行である場合、親クラステーブルにおける関連のある行はリンクによってその場所を確認することができない。これらの関連のある行を探すため、関心のあるファイルのFileIDに基づいてこれらのテーブルのサーチが行なわれてもよい。

【０２３１】

たとえば、ユーザがDocumentテーブル１７０４の行Ｙ２を検索し、File3に対する他の属性値のすべてを検索することを望んだと仮定する。E-mailに特有の属性値を含む行は、行Ｙ２のDerived RowID欄におけるポインタをたどることによって見つけられるかも知れず、これはE-mailテーブル１７０６における行Ｑ１を指し示す。しかしながら、残りの属性を探すためには、Filesテーブル１７０２をFileID F3に基づいてサーチする。このようなサーチにより行Ｘ３が見出されることとなり、これはFile3の残りの属性値を含む。

【０２３２】

代替の実施例によれば、関連のある行間のリンクは、すべての関連のある行がFileIDルックアップなしでその場所を確認することが可能となる態様で実現されてもよい。たとえば、各クラステーブルはまた、親クラステーブルにおける関連のある行のRowIDを含むParent RowID欄を有していてもよい。したがって、Documentテーブル１７０４の行Ｙ２に対するParent RowID欄はFilesテーブル１７０２における行Ｘ３を指し示すこととなる。代わりに、片方向リンクの連鎖における最後の行が、Filesテーブルにおける関連のある行へ戻るポインタを含んでいてもよい。さらに別の選択肢としては、各クラステーブルに対して、Filesテーブルにおける関連のある行へ戻るポインタを含む欄を設けることを含む。

したがって、Textテーブル１７０８の行Ｒ１およびDocumentテーブル１７０４の行Ｙ３はともに、Filesテーブル１７０２の行Ｘ４へ戻るポインタを含むことになる。

【０２３３】

サブクラス登録

上に述べたように、新しいクラスを登録することによってファイルシステムのクラス階層構造を拡張するための機構が提供される。一般的に、クラス登録プロセスにおいて提供される情報は、新しいクラスの親クラスを識別するデータと新しいクラスによって追加される属性を記述するデータとを含む。任意に、データはまた、新しいクラスのインスタンスに対して行なうことができる新しい方法を識別するのに用いられるデータを含んでいてもよい。

【０２３４】

登録情報は数多くの手法のうちのいずれを用いてファイルシステムに提供されてもよい。たとえば、ユーザに、登録されたクラスのすべてを表わすアイコンを含むグラフィックユーザインターフェイスを提示してもよく、ユーザはユーザインターフェイスによって表わされるコントロールを操作して、（１）クラスのうちの１つを新しいクラスの親として選択し、（２）新しいクラスに名を付け、（３）新しいクラスに対して追加の属性を定義付け、（４）新しいクラスに対して行われ得る新しい方法を定義付けてもよい。代わりに、ユーザはファイルシステムに対して、新しいクラスに対する登録情報を含むファイルを与えてもよい。ファイルシステムはそのファイルをパーシングして情報を識別し抽出し、その情報に基づいて新しいクラスに対するクラスファイルを作る。

【０２３５】

この発明の一実施例によれば、クラス登録情報がExtensible Markup Language（ＸＭＬ）ファイルの形でファイルシステムにもたらされる。ＸＭＬ形式はwww.oasis-open.org/cover/xml.html＃contentsおよびそこにリストされるサイトにおいて詳細に説明される。一般的に、ＸＭＬ言語は、フィールドを指名しフィールドの始まりおよび終わりをマークするタグとこれらのフィールドに対する値とを含む。たとえば「Folder」ファイルクラスに対する登録情報を含むＸＭＬ文書は以下の情報を含んでいるかもしれない。

【０２３６】

```
〈typename〉
folder
〈/typename〉
〈inherits_from〉
files
〈/inherits_from〉
〈dbi_classname〉
my_folder_methods
〈/dbi_classname〉
〈prop_def〉
    〈name〉
    max_children
    〈/name〉
    〈type〉
    integer
    〈/type〉
〈/prop_def〉
```

このファイルクラス登録文書を受取ったことに応答して、ファイルシステムは、新しいクラスFolderに対するテーブルを作成する。このようにして作成された新しいテーブルは、登録情報において定義付けられる属性の各々に対する欄を含む。この例においては、max_children属性のみが定義付けられている。max_children属性に対して特定されるデータタイプは「整数」である。したがって、Folderテーブルは、整数値を保持するmax_chi

ldren欄とともに作成される。属性の名称およびタイプに加えて、各属性に対してさまざまな他の情報がもたらされてもよい。たとえば、登録情報は、属性値に対する範囲または最大長さを示していてもよく、その欄にインデックスをつけるべきであるかまたはその欄が一意性または参照制約を受けるべきであるかを示していてもよい。

【０２３７】

登録情報はまた、新しいクラスファイルによってサポートされるどんな方法についての情報も含む。一実施例によれば、新しい方法はこれらの方法に関連付けられるルーチンを含むファイルを識別することによって特定される。一実施例によれば、各ファイルクラスに関連付けられるルーチンはJAVA（Ｒ）クラスにおいて実現される。第１のファイルクラスが第２のファイルクラスのサブクラスである場合、第１のファイルクラスに関連付けられる方法を実現するJAVA（Ｒ）クラスは第２のファイルクラスの方法を実現するJAVA（Ｒ）クラスのサブクラスである。

【０２３８】

上に挙げたＸＭＬの例では、登録情報のdbi__classnameフィールドがFolderファイルクラスに対するJAVA（Ｒ）クラスファイルを特定する。具体的には、登録情報はdbi__classnameフィールドに対してファイル名「my__folder__methods」をもたらし、my__folder__methods JAVA（Ｒ）クラスがFolderクラスの継承されていない方法に対するルーチンを実現することを示す。FolderクラスはFilesクラスのサブクラスであるため、my__folder__methodsクラスはFilesクラスに対する方法を実現するJAVA（Ｒ）クラスのサブクラスとなる。したがって、my__folder__methodsクラスはFiles方法を継承することとなる。

【０２３９】

親ファイルクラスによってサポートされていない新しい方法を定義付けることに加え、子ファイルクラスに対するルーチンは親クラスにおいて定義付けられる方法の実現をオーバーライドできる。たとえば、図１６に示されるFilesクラスは「記憶」方法を提供する。Folderクラスはその記憶方法を継承する。しかしながら、Filesクラスに対してもたらされる記憶方法の実現は、フォルダを記憶するのに必要とされる実現ではないかもしれない。したがって、Folderクラスは記憶方法のそれ自身の実現をもたらしてもよく、これによりFilesクラスによってもたらされる実現をオーバーライドする。

【０２４０】

ファイルのクラスの判定

ファイルシステムがファイルに対してオペレーションを行なうように求められた際、ファイルシステムはそのファイルが属するファイルのある特定のクラスに対する要求されたオペレーションを実現するルーチンを呼出す。上述のように、その同じオペレーションは、たとえばサブクラスがその親クラスによってもたらされた実現をオーバーライドした際には異なるファイルクラスに対して異なった態様で実現され得る。すなわち、正しいオペレーションが行なわれることを確実にするため、ファイルシステムはまず、オペレーションが行なわれるべきファイルのクラスを識別しなければならない。

【０２４１】

ファイルシステムにおいて既に記憶されているファイルに対しては、ファイルのクラスを識別するタスクは些細なことかもしれない。たとえば、図１７に示される実施例では、Filesテーブル１７０２は、どの所与の行に対しても、その行と関連付けられるファイルのクラスを示すデータを記憶するClass欄を含む。したがって、File3に対して「移動」オペレーションを行なうことのリクエストを受取った場合、行Ｘ３のClass欄を検査してFile3がE-mailのタイプのものであることを判定する。これにより、「移動」のE-mailの実現が実行されるべきである。「移動」のE-mail実現は、E-mailファイルクラスがその継承した「移動」方法の実現をオーバーライドする場合にはE-mailファイルクラスに対してもたらされる実現となる。そうでなければ、「移動」のE-mail実現はE-mailクラスによって継承された実現である。

【０２４２】

ファイルのクラスを識別するタスクは、ファイルが既にファイルシステムに記憶されてい

ない場合にはより困難であり得る。たとえば、ファイルシステムがファイルシステムに既に存在していないファイルを記憶することを求められた際、ファイルシステムはファイルテーブルを検査することによってクラス判定を行なうことができない。このような条件下では、ファイルのタイプを識別するのにさまざまな手法を用いてもよい。一実施例によれば、ファイルのタイプはファイルオペレーションリクエストにおいて明白にもたらされ得る。たとえば、オペレーティングシステムのコマンド行を介して発行されたコマンドに応答してリクエストがなされた場合、コマンド行引き数(command-line arguments)のうちの１つを用いてファイルのファイルタイプを示してもよい。たとえば、コマンドは「move a:＼mydocs＼file2c:＼yourdocs/class=document」と入力されてもよい。

【０２４３】

ファイルのクラスを判定するための別の手法には、ファイルの名称に含まれる情報に基づいてクラスを判定することが含まれる。たとえば、ある拡張子（たとえばdoc.wpd.pwpなど）を有するすべてのファイルはある特定のファイルクラス（たとえばDocument）のメンバとしてすべて扱われてもよい。したがって、ファイルシステムがこれらのファイルに対してオペレーションを行なうことを求められると、そのある特定のファイルクラスに関連付けられる方法実現が用いられる。

【０２４４】

ファイルのクラスを判定するためのさらに別の手法には、ファイルシステム階層構造内のファイルの場所に基づいてクラスを判定することが含まれる。たとえば、ある特定のディレクトリまたはディレクトリのセット内で作成されるすべてのファイルは、ファイルがどのように名付けられているかにかかわらずある特定のファイルクラスに属するものと推定され得る。これらおよび他の手法をさまざまな態様で組合せてもよい。たとえば、ある特定の拡張子を有するファイルは、そのファイルが第２のクラスと関連付けられるディレクトリに記憶されているのでなければ、第１のクラスのメンバとして扱われ得る。ファイルが第２のクラスに関連付けられるディレクトリに記憶されている場合、ファイルが別のファイルクラスのメンバであることをファイルオペレーションリクエストが明示的に識別しているのでなければ、ファイルは第２のクラスのメンバとして扱われる。

【０２４５】

ハードウェアの外観

図１８はこの発明の実施例が実現され得るコンピュータシステム１８００を示すブロック図である。コンピュータシステム１８００は、バス１８０２または情報を通信するための他の通信機構と、バス１８０２に結合され情報を処理するためのプロセッサ１８０４とを含む。コンピュータシステム１８００はまた、ランダムアクセスメモリ（ＲＡＭ）または他の動的記憶装置などのメインメモリ１８０６を含み、これはバス１８０２に結合され情報と、プロセッサ１８０４によって実行されるべき命令とを記憶する。メインメモリ１８０６はまた、プロセッサ１８０４によって実行されるべき命令の実行の間に一時変数または他の中間情報を記憶するのに用いられてもよい。コンピュータシステム１８００はさらに、バス１８０２に結合されプロセッサ１８０４のための静的情報および命令を記憶するための読取専用メモリ（ＲＯＭ）１８０８または他の静的記憶装置を含む。磁気ディスクまたは光ディスクなどのストレージデバイス１８１０が設けられ、バス１８０２に結合されて情報および命令を記憶する。

【０２４６】

コンピュータシステム１８００は、コンピュータユーザに情報を表示するための、陰極線管（ＣＲＴ）などのディスプレイ１８１２にバス１８０２を介して結合されてもよい。英数字キーおよび他のキーを含む入力デバイス１８１４は、バス１８０２に結合されプロセッサ１８０４へ情報およびコマンド選択を通信する。別のタイプのユーザ入力デバイスは、マウス、トラックボールまたはカーソル方向キーなどのカーソル制御１８１６であり、これはプロセッサ１８０４へ方向情報およびコマンド選択を通信し、ディスプレイ１８１２上でのカーソル移動を制御する。この入力デバイスは典型的に、第１の軸（たとえばｘ）および第２の軸（たとえばｙ）の２軸での２つの自由度を有し、これはデバイスが平面

における位置を特定することを可能にする。

【０２４７】

この発明は、ここに説明される手法を実現するためのコンピュータシステム１８００の使用に関する。この発明の一実施例によれば、これらの手法は、プロセッサ１８０４がメインメモリ１８０６に含まれる１つ以上の命令の１つ以上のシーケンスを実行することに応答してコンピュータシステム１８００によって実現される。このような命令は、ストレージデバイス１８１０などの別のコンピュータ可読媒体からメインメモリ１８０６に読込まれてもよい。メインメモリ１８０６に含まれる命令のシーケンスの実行により、プロセッサ１８０４がここに説明するプロセスステップを行なうこととなる。代替の実施例では、この発明を実現するのに布線回路をソフトウェア命令の代わりに、またはソフトウェア命令と組合せて用いてもよい。このように、この発明の実施例はハードウェア回路およびソフトウェアのいずれの特定の組合せにも限定されるものではない。

【０２４８】

ここで用いられる用語「コンピュータ可読媒体」は、実行のためにプロセッサ１８０４に命令を提供することにかかわるすべての媒体を指していう。このような媒体は数多くの形態をとることができ、これには不揮発性媒体、揮発性媒体および伝送媒体が含まれるがこれらに限定されるものではない。不揮発性媒体はたとえば、ストレージデバイス１８１０などの光ディスクまたは磁気ディスクを含む。揮発性媒体には、メインメモリ１８０６などの動的メモリが含まれる。伝送媒体には、バス１８０２をなす配線を含む、同軸ケーブル、銅線および光ファイバが含まれる。伝送媒体にはまた、電波および赤外線データ通信において生成されるもののような、音波または光波の形態をとっていてもよい。

【０２４９】

一般的な形態のコンピュータ可読媒体には、たとえば、フロッピー（Ｒ）ディスク、フレキシブルディスク、ハードディスク、磁気テープまたは他の磁気媒体すべて、ＣＤ－ＲＯＭ、他の光学媒体すべて、穿孔カード、紙テープ、孔のパターンを有する他の物理媒体すべて、ＲＡＭ、ＰＲＯＭおよびＥＰＲＯＭ、FLASH-EPROM他のメモリチップすべてまたはカートリッジ、以下に説明するような搬送波またはコンピュータが読むことができる他の媒体すべてが含まれる。

【０２５０】

さまざまな形態のコンピュータ可読媒体が、実行のためにプロセッサ１８０４に１つ以上の命令の１つ以上のシーケンスを与えることに関係し得る。たとえば、命令は初めに、遠隔地のコンピュータの磁気ディスクに担持されているかもしれない。遠隔地のコンピュータはその動的メモリに命令をロードして、命令をモデムを用いて電話回線を介して送信することができる。コンピュータシステム１８００が有するモデムが電話回線上のデータを受信し、赤外線送信機を用いてそのデータを赤外線信号に変換することができる。赤外線検出器は赤外線信号に載ったデータを受信し、適切な回路はそのデータをバス１８０２上に出力することができる。バス１８０２はデータをメインメモリ１８０６に運び、そこからプロセッサ１８０４が命令を取出し実行する。メインメモリ１８０６によって受信された命令は、任意にプロセッサ１８０４による実行の前または後のいずれかにおいてストレージデバイス１８１０に記憶されてもよい。

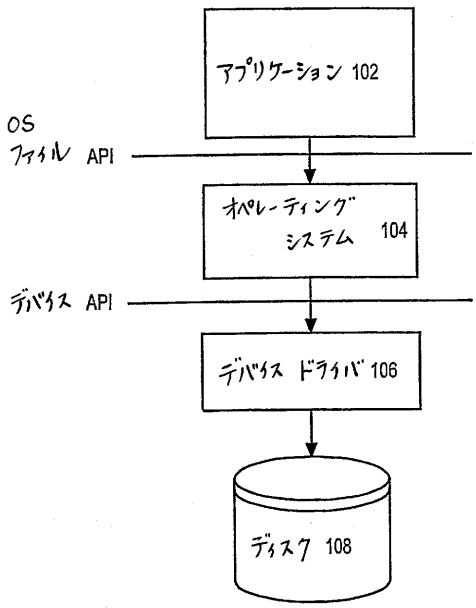
【０２５１】

コンピュータシステム１８００はまた、バス１８０２に結合される通信インターフェイス１８１８を含む。通信インターフェイス１８１８は、ローカルネットワーク１８２２に接続されるネットワークリンク１８２０に結合される双方向データ通信を提供する。たとえば、通信インターフェイス１８１８は、対応するタイプの電話回線に対するデータ通信接続をもたらすモデムまたはサービス総合デジタルネットワーク（ＩＳＤＮ）カードであってもよい。別の例としては、通信インターフェイス１８１８は、互換性のあるＬＡＮへのデータ通信接続をもたらすためのローカルエリアネットワーク（ＬＡＮ）カードであってもよい。無線リンクもまた実現され得る。このような実現例のいずれにおいても、通信インターフェイス１８１８は、さまざまなタイプの情報を表わすデジタルデータストリーム

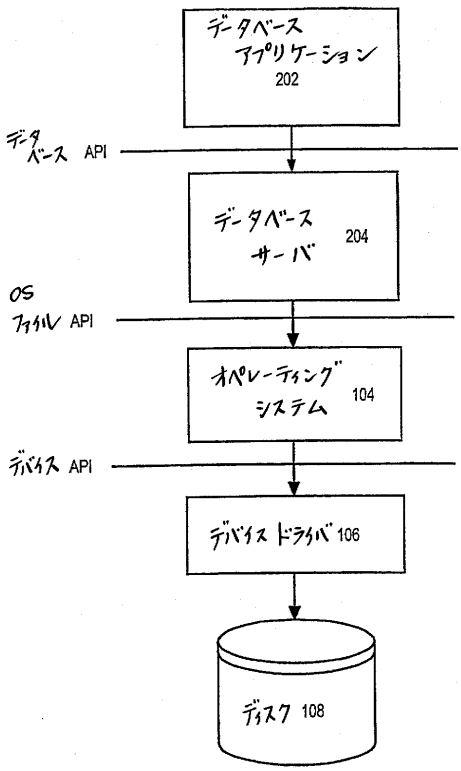
を担持する電気信号、電磁信号または光信号を送受信する。
【０２５２】
ネットワークリンク１８２０は典型的に、１つ以上のネットワークを介して他のデータデバイスに対するデータ通信を提供する。たとえば、ネットワークリンク１８２０は、ローカルネットワーク１８２２を介して、ホストコンピュータ１８２４に対してまたはインターネットサービスプロバイダ（ＩＳＰ）１８２６によって操作されるデータ機器に対して接続をもたらし得る。そのＩＳＰ１８２６は、現在一般的に「インターネット」１８２８と称されるワールドワイドパケットデータ通信ネットワークを介するデータ通信サービスを提供する。ローカルネットワーク１８２２およびインターネット１８２８はともに、デジタルデータストリームを担持する電気信号、電磁信号または光信号を用いる。さまざまなネットワークを介する信号と、デジタルデータをコンピュータシステム１８００と授受するネットワークリンク１８２０上のおよび通信インターフェイス１８１８を介する信号とは、情報を運ぶ搬送波の例示的な形態である。
【０２５３】
コンピュータシステム１８００は、ネットワーク、ネットワークリンク１８２０および通信インターフェイス１８１８を介して、プログラムコードを含む、メッセージを送信しデータを受信することができる。インターネットの例では、サーバ１８３０がインターネット１８２８、ＩＳＰ１８２６、ローカルネットワーク１８２２および通信インターフェイス１８１８を介してアプリケーションプログラムに対する要求されたコードを送信するかもしれない。この発明によれば、ダウンロードされたそのようなアプリケーションの１つがここに説明される手法を実現する。
【０２５４】
受信されたコードは、受信されるとともにプロセッサ１８０４によって実行されてもよく、および／または後に実行するためにストレージデバイス１８１０または他の不揮発性ストレージに記憶されてもよい。このような態様で、コンピュータシステム１８００は搬送波の形態であるアプリケーションコードを得てもよい。
【０２５５】
前述の明細書において、この発明をその特定の実施例に関連して説明した。しかしながら、この発明のより広い精神および範囲から逸脱することなくこれにさまざまな変更および修正を加えてもよいことが明らかになるであろう。明細書および図面はしたがって、例示的な意味でみなすべきであり、限定的な意味でみなすものではない。
【図面の簡単な説明】
【図１】　従来のアプリケーションにおいて、オペレーティングシステムによって提供されるファイルシステムを通じてデータが記憶される様子を示すブロック図である。
【図２】　従来のデータベースアプリケーションにおいて、データベースシステムによって提供されるデータベースＡＰＩを通じてデータが記憶される様子を示すブロック図である。
【図３】　データベースＡＰＩおよびＯＳファイルシステムＡＰＩを含む種々のインターフェイスを通じて同じデータの組にアクセス可能な、システムを示すブロック図である。
【図４】　トランスレーションエンジン３０８をより詳細に示すブロック図である。
【図５】　階層インデックスを示すブロック図である。
【図６】　階層インデックスによってエミュレートすることのできるファイル階層構造を示すブロック図である。
【図７】　本発明に一実施例に従った、関係データベース内にファイルを記憶するのに使用することのできるファイルテーブルを示すブロック図である。
【図８】　階層インデックスを使用してパス名を導出するステップを示すフローチャートである。
【図９】　データベースファイルサーバをより詳細に示すブロック図である。
【図１０】　ストアドクエリディレクトリのためのエントリを含む階層インデックスのブロック図である。

【図１１】　ストアドクエリディレクトリのための行を含むファイルテーブルのブロック図である。
【図１２】　ストアドクエリディレクトリを含むファイル階層構造を示すブロック図である。
【図１３】　ファイル階層構造を示すブロック図である。
【図１４】　図１３に示すファイル階層構造が、ここに説明するバージョニング技術の一実施例に従って、ドキュメントの更新に応答して更新される様子を示す、ブロック図である。
【図１５】　図１３に示すファイル階層構造が、ここに説明するバージョニング技術の一実施例に従って、ドキュメントがあるフォルダから別のフォルダへと移動するのに応答して更新される様子を示す、ブロック図である。
【図１６】　本発明の一実施例に従ったファイルクラスのクラス階層構造を示すブロック図である。
【図１７】　本発明の一実施例に従った、図１６のファイルクラス階層構造を実現するデータベース実現型ファイルシステムにおいて使用される、関係テーブルを示すブロック図である。
【図１８】　本発明の実施例がそれにおいて実現され得る、コンピュータシステムを示すブロック図である。

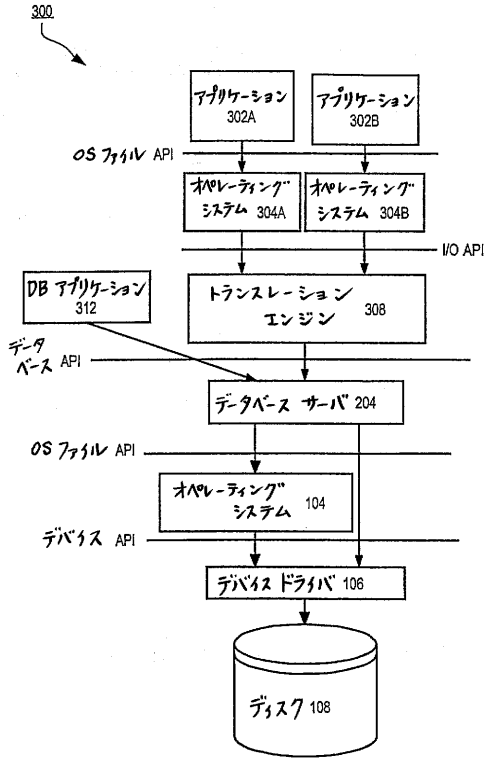
【図１】



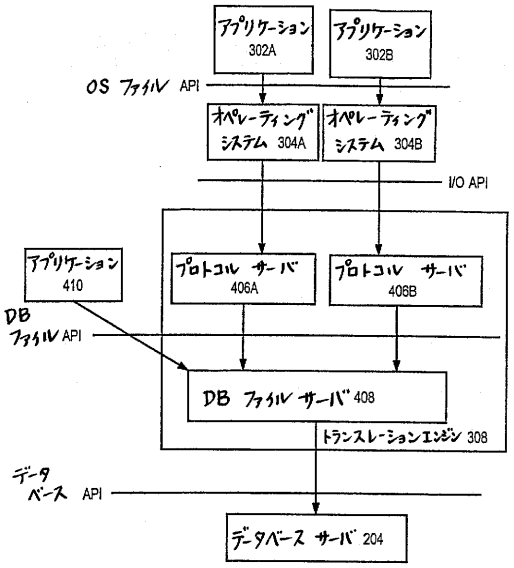
【図２】



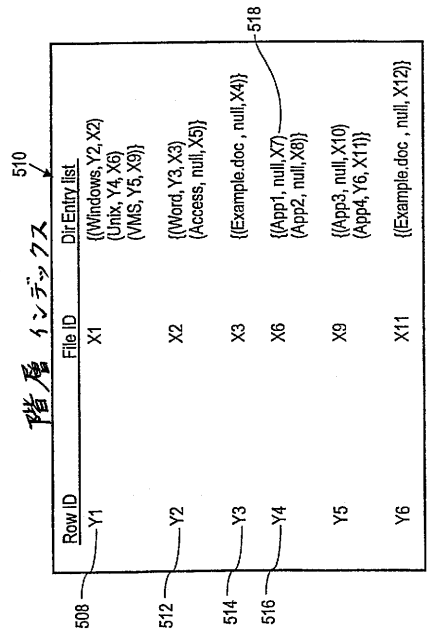
【図３】



【図４】



【図５】



【図６】

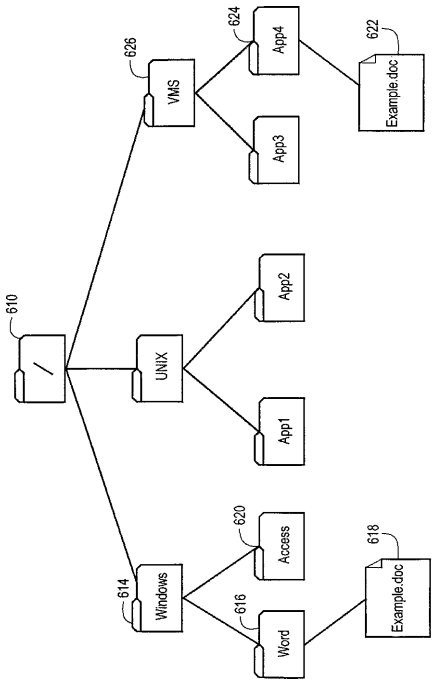
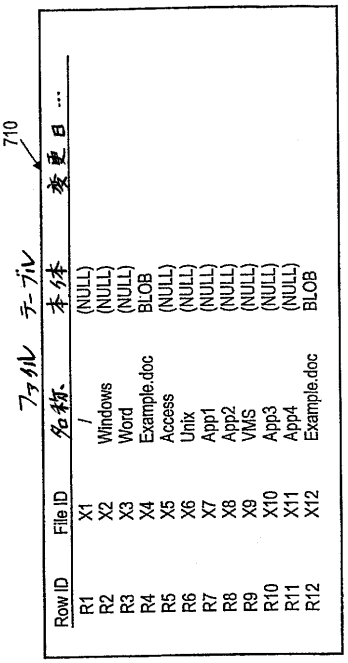
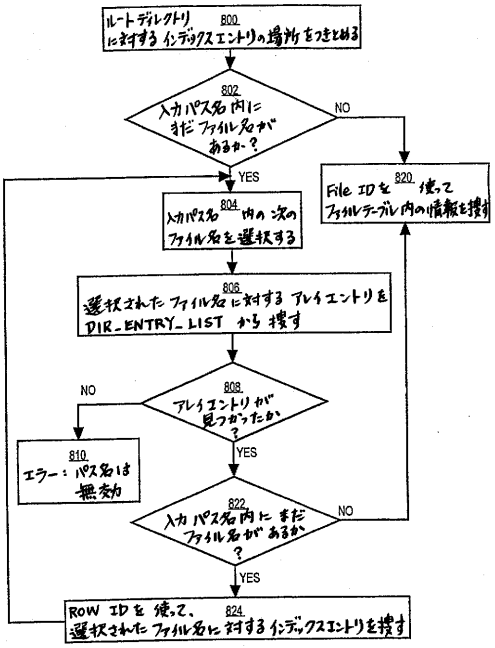


Fig. 6

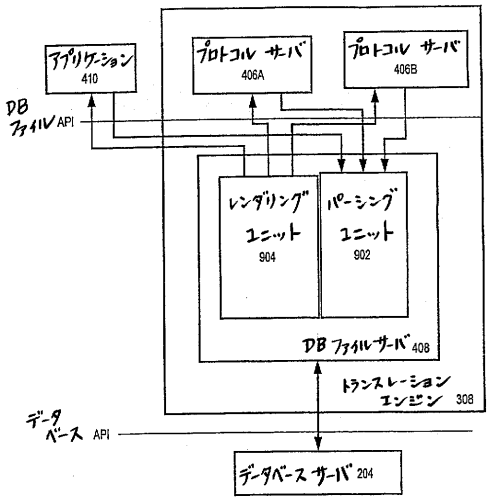
【図７】


710
ファイル テーブル
Row ID
File ID
名称
本体
変更日 ...
R1 X1 / (NULL)
R2 X2 Windows (NULL)
R3 X3 Word (NULL)
R4 X4 Example.doc BLOB
R5 X5 Access (NULL)
R6 X6 Unix (NULL)
R7 X7 App1 (NULL)
R8 X8 App2 (NULL)
R9 X9 VMS (NULL)
R10 X10 App3 (NULL)
R11 X11 App4 (NULL)
R12 X12 Example.doc BLOB

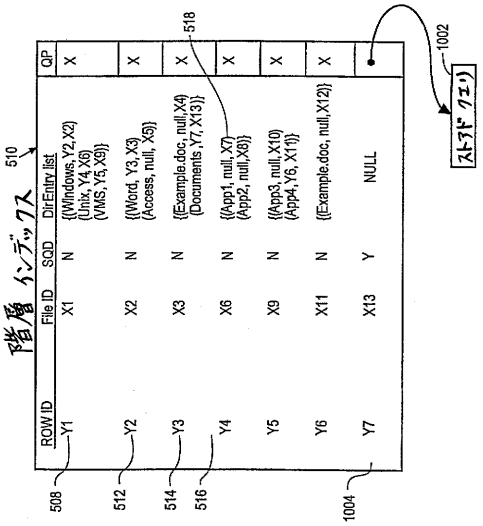
【図８】


ルートディレクトリ 800
にあるインデックスエントリの場所をつきとめる
802
入力パス名内にまだファイル名があるか？
NO
YES
804
入力パス名内の次のファイル名を選択する
820
File IDを使ってファイルテーブル内の情報を探す
806
選択されたファイル名に対するエントリを DIR_ENTRY_LIST から探す
808
エントリが見つかったか？
NO
YES
810
エラー：パス名は無効
822
入力パス名内にまだファイル名があるか？
NO
YES
824
ROW IDを使って、選択されたファイル名に対するインデックスエントリを探す

【図９】


アプリケーション 410
プロトコル サーバ 406A
プロトコル サーバ 406B
DB ファイル API
レンダリング ユニット 904
パーシング ユニット 902
DB ファイルサーバ 408
トランスレーション エンジン 308
データベース API
データベース サーバ 204

【図１０】


510
階層インデックス
518
1002
ストアドクエリ
QP
ROW ID
File ID
SQD
DirEntry list
Y1 X1 N {(Windows,Y2,X2) (Unix, Y4,X6) (VMS,Y5,X8)}
Y2 X2 N {(Word, Y3, X3) (Access, null, X5)}
Y3 X3 N {(Example.doc, null,X4) (Documents,Y7, X13)}
Y4 X6 N {(App1, null, X7) (App2, null,X8)}
Y5 X9 N {(App3, null,X10) (App4, Y6, X11)}
Y6 X11 N {(Example.doc, null,X12)}
Y7 X13 Y NULL
508
512
514
516
1004

【図１１】

710

ファイル テーブル

Row ID	File ID	名称	本体	変更日 ...
R1	X1	/	(NULL)	
R2	X2	Windows	(NULL)	
R3	X3	Word	(NULL)	
R4	X4	Example.doc	BLOB	
R5	X5	Access	(NULL)	
R6	X6	Unix	(NULL)	
R7	X7	App1	(NULL)	
R8	X8	App2	(NULL)	
R9	X9	VMS	(NULL)	
R10	X10	App3	(NULL)	
R11	X11	App4	(NULL)	
R12	X12	Example.doc	BLOB	
R13	X13	Documents	(NULL)	

【図１２】

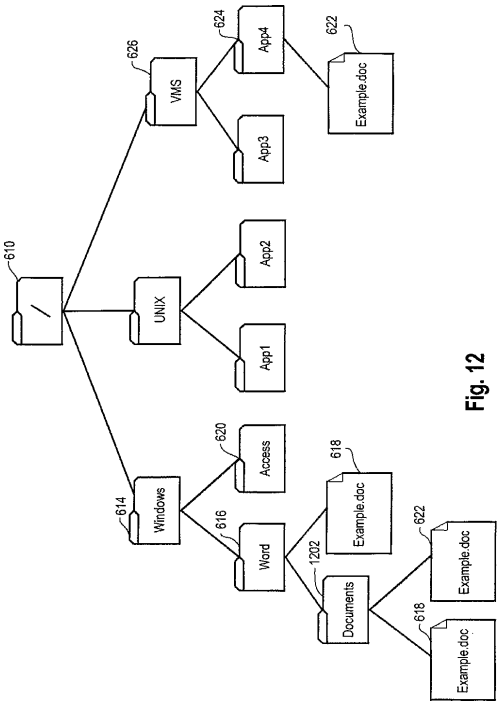


Fig. 12

【図１３】

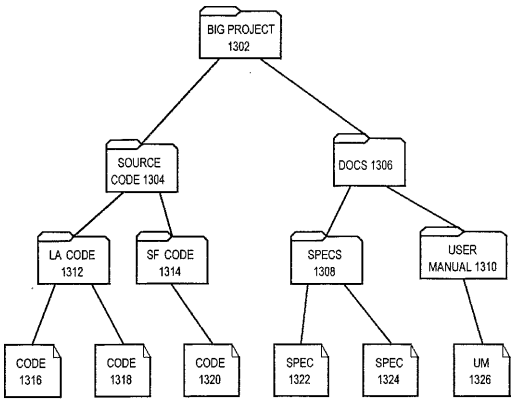


FIG. 13

【図１４】

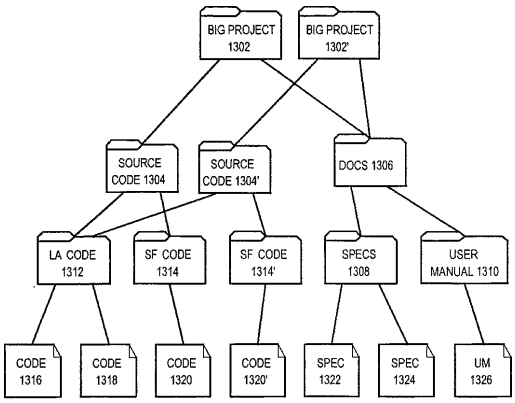


FIG. 14

【図１５】

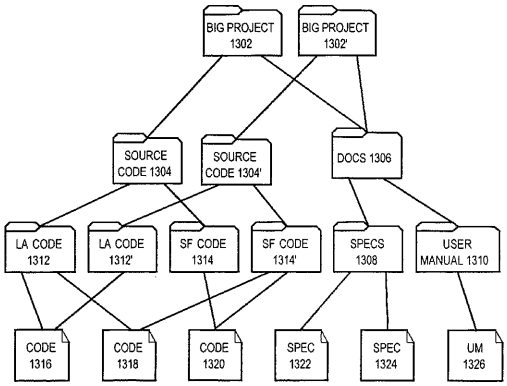
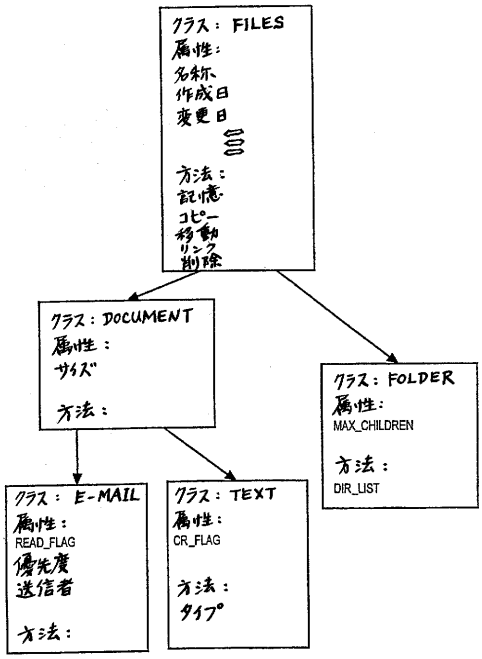
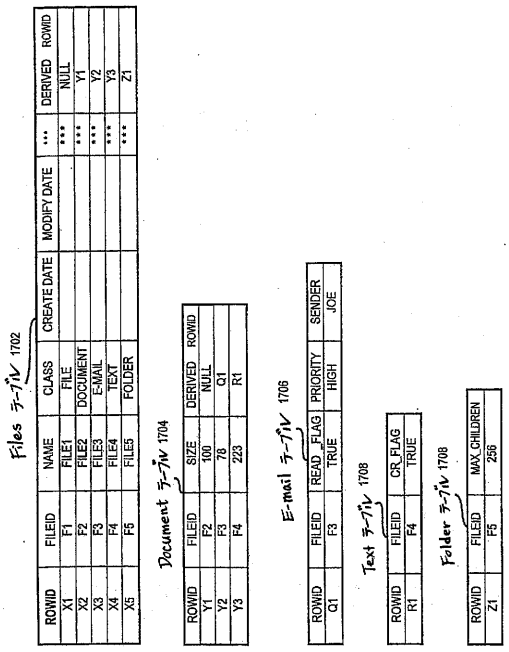


FIG. 15

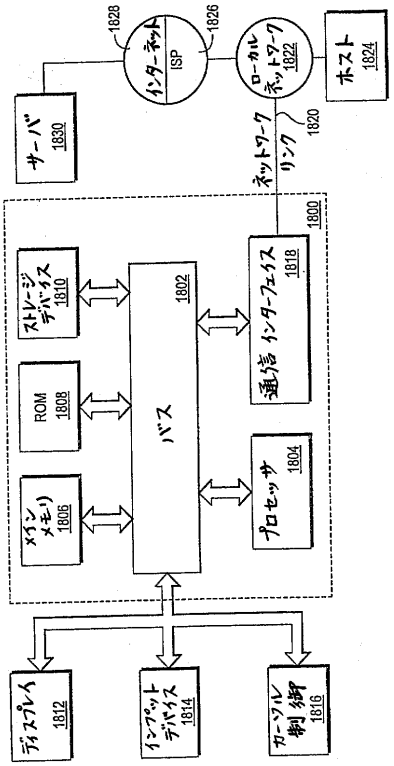
【図１６】



【図１７】



【図１８】



フロントページの続き

(31)優先権主張番号 09/571,060
(32)優先日 平成12年5月15日(2000.5.15)
(33)優先権主張国 米国(US)
(31)優先権主張番号 09/571,492
(32)優先日 平成12年5月15日(2000.5.15)
(33)優先権主張国 米国(US)
(31)優先権主張番号 09/571,496
(32)優先日 平成12年5月15日(2000.5.15)
(33)優先権主張国 米国(US)
(31)優先権主張番号 09/571,508
(32)優先日 平成12年5月15日(2000.5.15)
(33)優先権主張国 米国(US)
(31)優先権主張番号 09/571,568
(32)優先日 平成12年5月15日(2000.5.15)
(33)優先権主張国 米国(US)
(31)優先権主張番号 09/571,696
(32)優先日 平成12年5月15日(2000.5.15)
(33)優先権主張国 米国(US)

(74)代理人 100098316
弁理士 野田 久登
(74)代理人 100109162
弁理士 酒井 將行
(72)発明者 セドラー, エリック
アメリカ合衆国、９４３０６ カリフォルニア州、パロ・アルト、ティムロット・レーン、８４１
(72)発明者 ロバーツ, マイケル
アメリカ合衆国、９４３０６ カリフォルニア州、パロ・アルト、アシュトン・アベニュ、５７０

合議体
審判長 和田 志郎
審判官 稲葉 和生
審判官 山田 正文

(56)参考文献 特開平１０－２４７１５５（ＪＰ，Ａ）
水吉 俊幸, Ｃ／Ｓシステムの性能を引き出すためのネットワーク設計（Ｉ）, 日経オープンシステム, 日本, 日経ＢＰ社, １９９５年4月１５日, 第２５号, ｐ. ２８７～２９８
すずき ひろのぶ, Ｌｉｎｕｘで理解するＯＳ講座 Ｎｏ. 2, ＳｏｆｔｗａｒｅＤｅｓｉｇｎ, 日本, 株式会社技術評論社, １９９９年2月１８日, 第１００号, ｐ. ７６～８０
木村 憲雄, ネットワーク・ファイル・システムの研究, ＯＰＥＮ ＤＥＳＩＧＮ Ｎｏ. ５ 第2版, 日本, ＣＱ出版株式会社, １９９６年6月２０日, ｐ. １００～１３７

(58)調査した分野(Int.Cl., ＤＢ名)
G06F12/00