

PSQL v12

PSQL Programmer's Guide



免責事項

Actian Corporation は、本ソフトウェアおよびドキュメントの使用を、利用者またはその会社に対して「現状のまま」で、かつ同梱の使用許諾契約書に記載の契約条件によってのみ許諾するものです。Actian Corporation は、いかなる場合にも本ソフトウェアおよび本マニュアルに記載された内容に関するその他の一切の保証を、明示的にも黙示的にも行いません。Actian Corporation は、市場性、権利、特定の目的に対する適合性、あるいは一連の取引業務や職業的な使用に関する問題などに対し、一切の保証を行わないことを明示するとともに、利用者およびその会社がこれに同意したものとします。

商標

Btrieve、Client/Server in a Box および Pervasive は Actian Corporation の登録商標です。Built on Pervasive Software、DataExchange、MicroKernel Database Engine、MicroKernel Database Architecture、Pervasive.SQL、Pervasive PSQL、Solution Network、Ultralight、ZDBA は Actian Corporation の商標です。

Apple、Macintosh、Mac、および OS X は、Apple Inc の登録商標です。

Microsoft、MS-DOS、Windows、Windows 95、Windows 98、Windows NT、Windows Me、Windows 2000、Windows 2003、Windows 2008、Windows 7、Windows 8、Windows 10、Windows Server 2003、Windows Server 2008、Windows Server 2012、Windows XP、Win32、Win32s、および Visual Basic は、Microsoft Corporation の登録商標です。

NetWare および Novell は Novell, Inc の登録商標です。NetWare Loadable Module、NLM、Novell DOS、Transaction Tracking System および TTS は Novell, Inc の商標です。

Oracle、Java、および Oracle または Java を含むすべての商標やロゴは、Oracle Corporation の商標または登録商標です。

すべての会社名および製品名は各社の商標または登録商標です。

© Copyright 2016 Actian Corporation. All rights reserved. このマニュアルの全文、一部に関わりなく複製、複写、配布をすることは、前もって発行者の書面による同意がない限り禁止します。

本製品には、Powerdog Industries により開発されたソフトウェアが含まれています。

© Copyright 1994 Powerdog Industries. All rights reserved. 本製品には、KeyWorks Software により開発されたソフトウェアが含まれています。

© Copyright 2002 KeyWorks Software. All rights reserved. 本製品には、DUNDAS SOFTWARE により開発されたソフトウェアが含まれています。

© Copyright 1997-2000 DUNDAS SOFTWARE LTD. All rights reserved. 本製品には、Apache Software Foundation Foundation (<http://www.apache.org/>) により開発されたソフトウェアが含まれています。

本製品ではフリー ソフトウェアの unixODBC Driver Manager を使用しています。これは Peter Harvey (pharvey@codebydesign.com) によって作成され、Nick Gorham (nick@easysoft.com) により変更および拡張されたものに Actian Corporation が一部修正を加えたものです。Actian Corporation は、unixODBC Driver Manager プロジェクトの LGPL 使用許諾契約書に従って、このプロジェクトの現在の保守管理者にそのコード変更を提供します。unixODBC Driver Manager の Web ページは www.unixodbc.org にあります。このプロジェクトに関する詳細については、現在の保守管理者である Nick Gorham (nick@easysoft.com) にお問い合わせください。

GNU Lesser General Public License (LGPL) は本製品の配布メディアに含まれています。LGPL は www.fsf.org/licenses/licenses/lgpl.html でも見ることができます。

PSQL Programmer's Guide

2016 年 7 月

目次

このマニュアルについて	xi
情報の参照先	xii
このマニュアルの読者	xiii
このマニュアルの構成	xiv
データベース アクセス方法	xiv
MicroKernel エンジンを使用したトランザクショナル プログラミング	xiv
リレーショナル プログラミング	xiv
付録	xv
表記上の規則	xvi
1 PSQL アクセス方法の概要	1
PSQL アクセス方法の概要	2
PSQL での SQL アクセス	3
2 開発者向けクイック スタート	5
アクセス方法の選択	6
データベース接続のクイック リファレンス	8
ADO.NET 接続	8
ADO/OLE DB 接続	8
JDBC 接続	8
Java クラス ライブラリ	9
DSN を使用しない接続	9
ODBC 情報	10
その他の SQL アクセス方法	10
アプリケーション開発のためのその他のリソース	11
概念情報	11
リファレンス情報	11
開発者向けオンライン リソース	11
サンプル コード	11
3 MicroKernel エンジンのアプリケーション開発	13
MicroKernel エンジン環境	14
ドキュメント	14
MicroKernel エンジンの設定に関する問題	15
4 MicroKernel エンジンの基礎	17
MicroKernel エンジンの概要	18
MicroKernel エンジン環境	19
ページ	20
ページ タイプ	20
ページ サイズ	20
ファイル タイプ	23
標準データ ファイル	23

データオンリー ファイル	23
キーオンリー ファイル	23
ラージ ファイル	24
長いファイル名	24
データ型	25
キー属性	26
キー属性の解説	26
キー仕様	37
データベース URI	40
構文	40
パラメーターの優先順位	41
特殊文字	41
備考	42
例	42
IPv6	43
ダブルバイト文字のサポート	44
レコード長	45
データの整合性	47
レコード ロック	47
トランザクション	47
トランザクション一貫性保守	48
システム データ	49
シャドウ ページング	49
ファイルのバックアップ	50
イベント ログギング	51
パフォーマンスの向上	52
システム トランザクション	52
メモリ管理	54
ページプリアロケーション	54
Extended オペレーション	54
ディスク使用量	55
空きスペース リスト	55
インデックス バランスの実行	55
データ圧縮	56
ブランク トランケーション	56
5 データベースの設計	57
データ ファイルについて	58
データ ファイルの作成	60
データ レイアウト	60
ファイル仕様およびキー仕様の構造体の作成	64
ページ レベル圧縮を用いたファイルの作成	66
Create オペレーションの呼び出し	66
Create Index オペレーション	67
論理レコード長の計算	69
ページ サイズの選択	70

ファイルサイズの予測	76
式および派生手順	76
データベースの最適化	79
重複キー	79
ページプリアロケーション	80
ブランク トランケーション	82
レコード圧縮	82
インデックス バランス	83
可変長部割り当てテーブル	83
キーオンリー ファイル	84
セキュリティの設定	85
オーナー ネーム	85
排他モード	86
SQL セキュリティ	86
6 言語インターフェイス モジュール	87
インターフェイス モジュールの概要	88
C/C++	90
インターフェイス モジュール	90
プログラミングの必要条件	91
Delphi	92
DOS (Btrieve)	93
インターフェイス モジュール	93
Pascal	95
Visual Basic	96
7 インターフェイス ライブラリ	99
インターフェイス ライブラリの概要	100
Linux および OS X	100
PSQL アプリケーションの配布	101
PSQL の配布規則	101
PSQL ActiveX ファイルの登録	101
PSQL を開発したアプリケーションと共にインストールする	101
8 レコードの処理	103
オペレーションのシーケンス	104
レコードへのアクセス	106
物理位置によるレコードへのアクセス	106
キー値によるレコードへのアクセス	107
可変長レコードの読み取り	108
チャンクによるレコードへのアクセス	109
レコードの挿入と更新	111
ミッションクリティカルな挿入と更新における信頼性の確保	111
重複不可キーの挿入	111
可変長レコードの挿入および更新	112
固定長部分の読み取りおよび更新	112

変更不可キーの更新	113
No-Currency-Change (NCC) オペレーション	113
マルチレコードのオペレーション	114
用語	114
背景	114
検証	115
最適化	115
パフォーマンスのヒント	118
キーの追加と削除	120
9 複数のクライアントのサポート	121
Btrieve クライアント	122
受動的並行性 (パッシブ コンカレンシー)	125
レコードのロック	127
ユーザー トランザクション	128
ロック	128
並行トランザクションのレコード ロック	130
暗黙ロック	133
ファイル ロック	134
複数並行制御ツールの例	135
例 1	135
例 2	138
複数ポジション ブロックの並行制御	140
複数ポジション ブロック	141
クライアント ID パラメーター	142
10 Btrieve アプリケーションのデバッグ	143
トレース ファイル	144
クライアント / サーバー環境における間接的な Chunk オペレーション	147
エンジンのシャットダウンと接続のリセット	148
ファイル内の無駄な領域の削減	149
11 Btrieve API プログラミング	151
Btrieve API プログラミングの基礎	152
Btrieve API フロー チャート	152
Visual Basic に関する注記	153
Delphi に関する注記	154
PSQL アプリケーションの起動	155
PSQL ソース モジュールの追加	155
Btrieve API のコード サンプル	156
ファイルの作成	156
レコードの挿入	161
レコードの更新	164
Step オペレーションの実行	166
Get オペレーションの実行	168

チャンク、BLOB、および可変長レコード	171
セグメント化されたインデックスの処理	175
Visual Basic のための Btrieve API 関数の宣言	177
12 データベースの作成	179
名前付きデータベース	180
バウンド データベース	181
データベース コンポーネントの作成	182
名前付け規則	183
一意名	183
有効な文字	183
名前の最大長	183
大文字と小文字の区別	184
データ辞書の作成	185
テーブルの作成	186
エイリアス	186
列の作成	187
インデックスの作成	188
インデックス セグメント	188
インデックス属性	189
13 リレーショナル データベース設計	191
データベース設計の概要	192
設計の段階	193
概念設計	193
論理設計	193
物理設計	195
14 データの挿入と削除	197
データの挿入および削除の概要	198
値の挿入	199
トランザクションの処理	200
データの削除	201
インデックスの削除	202
列の削除	203
テーブルの削除	204
データベース全体の削除	205
15 データの変更	207
データ変更の概要	208
テーブルの変更	209
デフォルト値の設定	210
UPDATE ステートメントの使用	211

16 データの取得	213
データ取得の概要	214
ビュー	215
ビューの機能	215
テンポラリ ビューとストアド ビュー	215
ビューの読み取り専用テーブル	216
マージ可能なビュー	217
選択リスト	218
ソートされた行とグループ化された行	219
結合	220
ほかのテーブルとのテーブルの結合	221
テーブルとのビューの結合	221
結合のタイプ	221
サブクエリ	223
サブクエリの制限	223
関連サブクエリ	223
制限句	224
制限句演算子	224
制限句の例	225
関数	227
集計関数	227
集合関数への引数	227
集計関数の規則	228
スカラー関数	228
17 ロジックの格納	229
ストアド プロシージャ	230
ストアド プロシージャと位置付け更新	230
ストアド プロシージャの宣言	230
ストアド プロシージャの呼び出し	231
ストアド プロシージャの削除	231
SQL 変数ステートメント	232
プロシージャ所有の変数	232
代入ステートメント	232
SQL 制御ステートメント	233
複合ステートメント	233
IF ステートメント	233
LEAVE ステートメント	233
LOOP ステートメント	234
WHILE ステートメント	234
SQL トリガー	235
トリガーのタイミングと順序	235
トリガー アクションの定義	236
18 データの管理	237
データ管理の概要	238

テーブル間の関係の定義	239
参照整合性の定義	239
キー	241
主キー	241
外部キー	242
参照制約	244
参照整合性規則	244
サンプル データベースの参照整合性	247
Course テーブルを作成する	247
Course テーブルに主キーを追加する	247
参照制約を使用して Student テーブルを作成する	247
データベース セキュリティの管理	248
データベース権限の理解	248
データベース セキュリティの確立	249
セキュリティの有効化	249
ユーザー グループとユーザーの作成	250
アクセス権の付与	250
ユーザーとユーザーグループの削除	251
アクセス権の取り消し	252
セキュリティの無効化	252
データベース セキュリティに関する情報の取得	252
並行制御	253
トランザクション処理	253
トランザクションの開始と終了	253
ネストされたトランザクションへのセーブポイントの使用	254
特に考慮すべき点	255
分離レベル	255
パッシブ コントロール	259
PSQL データベースのアトミシティ	260
プロシージャ内のトランザクション制御	260
A インターナショナル ソート規則を使用した照合順序のサンプル	261
ドイツ語のサンプル照合順序	262
未ソートのデータ	262
ソート済みのデータ	263
スペイン語のサンプル照合順序	264
未ソートのデータ	264
ソート済みのデータ	265
フランス語のサンプル照合順序	266
未ソートのデータ	266
ソート済みのデータ	267
B サンプル データベース テーブルと参照整合性	269
Demodata サンプル データベースの概要	270
Demodata サンプル データベースの構造	271
前提条件	271

エンティティの関係	271
Demodata サンプル データベースの参照整合性	273
Demodata サンプル データベースのテーブル設計	275
BILLING テーブル	275
CLASS テーブル	275
COURSE テーブル	276
DEPT テーブル	276
ENROLLS テーブル	276
FACULTY テーブル	277
PERSON テーブル	277
ROOM テーブル	278
STUDENT テーブル	278
TUITION テーブル	278

このマニュアルについて

PSQL ソフトウェア開発キット（SDK）を使用すれば、開発者は **MicroKernel** エンジン、リレーショナル エンジン、あるいはその両方を使用してアプリケーションを作成することができます。このマニュアルでは、PSQL インターフェイスを使用するアプリケーションの開発方法に関する基本事項について説明します。

情報の参照先

SDK のユーザー ドキュメントはデータベース エンジンのインストール時に一緒にインストールできます。このドキュメントには本マニュアルとその他の SDK マニュアルが含まれています。

探す情報

このマニュアルに含まれる情報

ADO および OLE DB を使用したプログラミング

Distributed Tuning Interface の使用法

API リファレンス マニュアル

ActiveX コントロール、OLE DB と ADO、Distributed Tuning Objects、および PSQL Direct Access Components for Delphi and C++ Builder

PSQL のインストールと実行

Java を使用したプログラミング

PSQL における管理者タスクの実行

PSQL エラー コード

その他

参照先

「[このマニュアルの構成](#)」

『PSQL Data Provider for .NET Guide』を参照してください。

『Distributed Tuning Interface Guide』を参照してください。

『Btrieve API Guide』を参照してください。

『ActiveX Controls Guide』、『OLE DB Provider Guide』、『Distributed Tuning Objects Guide』、および『PSQL Direct Access Components Guide』を参照してください。

『Getting Started With PSQL』を参照してください。

『JDBC Driver Guide』および『Java Class Library Guide』を参照してください。

『PSQL User's Guide』を参照してください。

『Status Codes and Messages』を参照してください。

索引を使って特定の必要な問題に関する情報を探してください。

ドキュメント ライブラリのビューアーは PSQL Control Center (PCC) に組み込まれました。ドキュメントは、PCC インターフェイスの [ようこそ] ビュー、[ヘルプ] メニュー、F1 (Windows) または Shift + F1 (Linux) キーを押すことによって開くことができます。

MicroKernel エンジンは、ハイパフォーマンスのデータ処理とプログラミングの生産性向上を目的として設計されています。ODBC によって、PSQL リレーショナル エンジンへリレーショナル アクセスすることができます。

このマニュアルの読者

このマニュアルでは、さまざまなデータベースへのアクセス方法や API を使用して PSQL アプリケーションを開発する方法を学習したい開発者向けに、手続き的な情報を提供しています。また、トランザクショナルおよびリレーショナルレベルでの PSQL データベースの設計と概念についても説明しています。

このマニュアルの構成

このマニュアルでは、以下の項目について説明します。

データベース アクセス方法

- 第1章「[PSQL アクセス方法の概要](#)」

この章では、PSQL アプリケーションを開発できる各種ビジュアル コンポーネントと API の概要を示します。

- 第2章「[開発者向けクイック スタート](#)」

この章では、アクセス方法の詳細を示します。

MicroKernel エンジンを使用したトランザクショナル プログラミング

- 第3章「[MicroKernel エンジンのアプリケーション開発](#)」

この章では、MicroKernel エンジン環境でのアプリケーションの開発と実行に関する情報を示します。

- 第4章「[MicroKernel エンジンの基礎](#)」

この章では、API および MicroKernel エンジンの機能について説明します。

- 第5章「[データベースの設計](#)」

この章では、データ ファイルの作成、システム パフォーマンスの向上、セキュリティの設定に関する情報を示します。

- 第6章「[言語インターフェイス モジュール](#)」

この章では、PSQL SDK インストール オプションに用意されている言語インターフェイス ソース モジュールを示します。

- 第7章「[インターフェイス ライブラリ](#)」

この章では、PSQL インターフェイス ライブラリと Glue DLL ファイルの配布条件について概説します。

- 第8章「[レコードの処理](#)」

この章では、レコードの挿入と更新、レコード内の位置の確立、キーの追加と削除に関する情報を示します。

- 第9章「[複数のクライアントのサポート](#)」

この章では、複数のユーザーとアプリケーションのサポートに関する基本的な概念について説明します。

- 第10章「[Btrieve アプリケーションのデバッグ](#)」

この章では、アプリケーションのトラブルシューティングに関するヒントを示します。

- 第11章「[Btrieve API プログラミング](#)」

この章では、Btrieve API を直接呼び出して PSQL アプリケーションの開発を開始する場合に役立つ情報を示します。

リレーショナル プログラミング

- 第12章「[データベースの作成](#)」

この章では、データベースの作成方法、つまり、データ辞書を作成し、データベースのテーブル、列およびインデックスを作成する方法について説明します。

- 第13章「[リレーショナル データベース設計](#)」

この章では、リレーショナル データベース設計の基本原則について概説します。開発プロセス全体にわたる完全なデータベース設計は、データベースの機能とパフォーマンスの成功に不可欠です。

- 第14章「[データの挿入と削除](#)」

この章では、PSQL アプリケーションまたは SQL ステートメントを使用してデータベースにデータを追加する方法について説明します。また、データベースから行、インデックス、列またはテーブルを削除したり、データベースが不要になったときにデータベース全体を削除する方法について説明します。

- 第15章「[データの変更](#)」

この章では、テーブル定義、列属性およびデータを変更する方法について説明します。これらのタスクを実行するには、対話形式のアプリケーションを使用して SQL ステートメントを入力します。

- 第16章「[データの取得](#)」

この章では、SELECT ステートメントを使用してデータを検索する方法について説明します。

- 第17章「[ロジックの格納](#)」

この章では、将来使用するために SQL プロシージャを格納する方法と SQL トリガーを作成する方法について説明します。

- 第18章「[データの管理](#)」

この章では、テーブル、データベースのセキュリティ管理、トランザクションの並行制御の関係の定義について説明します。また、PSQL データベースのアトミシティについても説明します。

付録

- 付録A「[インターナショナル ソート規則を使用した照合順序のサンプル](#)」

この付録では、MicroKernel エンジンに用意されている ISR テーブルを使用した言語固有の文字列のサンプル照合順序を示します。

- 付録B「[サンプル データベース テーブルと参照整合性](#)」

この付録では、大学のサンプル データベースのテーブル設計について説明します。

表記上の規則

特段の記述がない限り、コマンド構文、コード、およびコード例では、以下の表記が使用されます。

大文字小文字の 区別	通常、コマンドと予約語は、大文字で表記されます。本書で別途記述がない限り、これらの項目は大文字、小文字、あるいはその両方を使って入力できます。たとえば、MYPROG、myprog、またはMYprogと入力することができます。
太字	太字で表示される単語には次のようなものがあります。メニュー名、ダイアログ ボックス名、コマンド、オプション、ボタン、ステートメントなど。
固定幅フォント	固定幅フォントは、コマンド構文など、ユーザーが入力するテキストに使われます。
[]	省略可能な情報には、 <code>[log_name]</code> のように、角かっこが使用されます。角かっこで囲まれていない情報は必ず指定する必要があります。
	縦棒は、 <code>[file_name @file_name]</code> のように、入力する情報の選択肢を表します。
< >	< > は、 <code>/D=<5 6 7></code> のように、必須項目に対する選択肢を表します。
変数	<i>file name</i> のように斜体で表されている語は、適切な値に置き換える必要のある変数です。
...	<code>[parameter...]</code> のように、情報の後に省略記号が続く場合は、その情報を繰り返し使用できます。
::=	記号 <code>::=</code> は、ある項目が別の項目用語で定義されていることを意味します。たとえば、 <code>a::=b</code> は、項目 <i>a</i> が <i>b</i> で定義されていることを意味します。

PSQL アクセス方法の概要

1

この章では、PSQL プログラミングについて概説します。以下の項目が含まれます。

- 「[PSQL アクセス方法の概要](#)」
- 「[PSQL での SQL アクセス](#)」

PSQL アクセス方法の概要

以下に、PSQL のアクセス方法と API の要約を示します。

アクセス方法	説明	使用目的
Btrieve (MicroKernel エンジン)	独創的な Btrieve API	Btrieve データベース アプリケーションの作成
ADO (Microsoft IDE)	高レベルなビジュアルまたはコードベースのプログラミング	トランザクショナルまたはリレーショナル (SQL) アプリケーションのビジュアルプログラミング。これは、Microsoft 開発環境にお勧めできるプログラミング インターフェイスです。
PDAC (Embarcadero IDE)	PSQL Direct Access Components for Delphi and C++ Builder	Embarcadero データウェア コンポーネントの機能に取って代わるもので、Embarcadero Database Engine が不要になります。
ODBC (リレーショナル)	Microsoft Open Database Connectivity (ODBC)	SQL ベースのアプリケーションの作成
Java クラス ライブラリ	MicroKernel エンジン データ アクセス用の Java クラス ライブラリ	MicroKernel エンジンに接続する Java ベースのアプリケーションの作成
JDBC	Sun の Java Database Connectivity の実装	業界標準の API を使用した JDBC ベース SQL アプリケーションの作成。
Distributed Tuning Interface (DTI)	監視および管理用の PSQL API	アプリケーションからの管理機能およびユーティリティ機能の実行、アプリケーションのデータ辞書ファイルの作成と保守。
Distributed Tuning Objects (DTO)	監視と管理用の PSQL オブジェクト指向プログラミング インターフェイス	アプリケーションからの管理機能およびユーティリティ機能の実行、アプリケーションのデータ辞書ファイルの作成と保守。

PSQL での SQL アクセス

ODBC は SQL データベース用のアクセス方法です。これによって以下のような利点が得られます。

- パフォーマンスの向上
- 標準ベースの API
- PSQL のレプリケーション製品の中核

PSQL アプリケーションを開発するために、ODBC API を習得する必要はありません。開発を容易にする OLE DB または JDBC のような追加インターフェイスもあります。

開発者向けクイック スタート

2

PSQL ソフトウェア開発キット (SDK) は、2 つのデータベース ソリューションのそれぞれの長所を提供します。MicroKernel エンジンは高速データ トランザクションを提供し、また、リレーショナル エンジンは同じデータに対する完全機能型リレーショナル データ アクセスを提供し、報告および意思決定支援におけるパフォーマンスを向上させます。

この章では、以降で説明するアプリケーションの構築を PSQL で容易に行えるようにするためのヒントを示します。

- [「アクセス方法の選択」](#)
- [「データベース接続のクイック リファレンス」](#)
- [「アプリケーション開発のためのその他のリソース」](#)

アクセス方法の選択

開発戦略の選択肢には多くの要因が影響します。各種プラットフォーム上でツールを使用できるか、開発者が所定のプログラミング環境をよく理解しているか、また移植可能であるか、といった条件は、多くの場合、このプロセスにおいて決定的な役割を果たします。また一方で、開発者が柔軟性を持ち合わせている場合は、多岐にわたる要因を考慮する必要があります。

パフォーマンスは常に考慮の対象になります。ただし、実行時のパフォーマンスと開発期間とのバランスをとらなければなりません。すなわち、プログラムを短期間で完成するのと、使用時に短時間で実行できるようにするのとではどちらがより重要であるかということです。

データベースプログラミングでは、データベースのインターフェイスは、開発期間と実行時のパフォーマンスの両方に影響を与えます。多くの場合、SQL と Btrieve のどちらを選択するかはこれらの要因だけにに基づきます。

PSQL 製品を初めて使う場合、ADO NET/OLE DB、ActiveX コントロール、JDBC、PSQL Direct Access Components for Delphi and C++ Builder、その他サードパーティ製開発ツールなどのアクセス方法を使用して PSQL アプリケーションを開発することができます。

Btrieve API に直接書き込みを行う場合は、「[Btrieve API プログラミング](#)」を参照してください。この章では、いくつかのプログラミング言語でヒントとコード サンプルを示します。

表 1 では、さまざまな PSQL のアクセス方法を比較しています。

表 1 アプリケーション プログラミングのアクセス方法の比較

アクセス方法	特性	適応項目
Btrieve API	<ul style="list-style-type: none"> ◆ ほとんどの Windows プログラミング言語から DLL を呼び出し可能。 ◆ データベースの完全な機能セットを公開。 ◆ 最小サイズ。 ◆ 最大の柔軟性。 ◆ アプリケーションとデータ間の最短コード パス。 ◆ リレーショナル データベース管理システムにおける最小のコード オーバーヘッド。(ただし、データベース管理専用のアプリケーション コード数を増やす必要があります。) ◆ クライアント / サーバー機能。 ◆ BLOB サポート。 	<ul style="list-style-type: none"> ◆ サイズまたは実行時のパフォーマンスを重視するアプリケーション。
Java	<ul style="list-style-type: none"> ◆ シン クライアント。 ◆ プラットフォーム間の移植可能性。 ◆ インターネットまたはイントラネットの能力。 ◆ Winsock プロトコルと JNI プロトコルをサポート。 ◆ マシンと OS の独立性。インターネットと Web の能力。 ◆ 最小サイズ。 ◆ 優れた柔軟性。 ◆ インターフェイスに実装されている行セット、フィールドの抽象化。 ◆ 総合的なパフォーマンス。Java は、コードのオーバーヘッドの大きなインタプリタ言語です。 ◆ クライアント / サーバー機能、言語に固有のバージョン管理。 	<ul style="list-style-type: none"> ◆ Web アプレット。 ◆ インターネットベース アプリケーション。 ◆ 各種ハードウェアおよび OS プラットフォームで実行しなければならないアプリケーション。

表 1 アプリケーション プログラミングのアクセス方法の比較

アクセス方法	特性	適応項目
ADO/OLE DB	<ul style="list-style-type: none"> ◆ Visual Studio との優れた統合。 ◆ トランザクショナル、リレーショナルのいずれの状況でも動作。 ◆ インターネットまたはイントラネットに適応。 ◆ 行セット、フィールドの抽象化。 	<ul style="list-style-type: none"> ◆ Visual Studio を使用したアプリケーション開発。
ADO.NET	<ul style="list-style-type: none"> ◆ 優れた総合的パフォーマンス。 ◆ インターネットの能力。 ◆ XML サポート。 ◆ 効率的なスケーラブル アーキテクチャ。 	<ul style="list-style-type: none"> ◆ 実行時間が最重要である、管理された環境で実行するアプリケーション。
ActiveX	<ul style="list-style-type: none"> ◆ Visual Basic ネイティブ インターフェイス。 ◆ ほとんどの Windows プログラミング環境でサポート。 ◆ 優れた柔軟性。 ◆ 優れた総合的パフォーマンス。 ◆ インターネットの能力。 ◆ 行セット、フィールドの抽象化。 ◆ Extended オペレーション、テーブル結合機能。 ◆ クライアント / サーバー機能。 	<ul style="list-style-type: none"> ◆ 実行時のパフォーマンスとコーディングのしやすさとのバランスが重要なアプリケーション。 ◆ ダウンロード時のフットプリントが最小であることや、Java のマシン非依存性は必要としないが、インターネット上のデータへのアクセスを必要とするアプリケーション。
SQL/ODBC	<ul style="list-style-type: none"> ◆ データベースの実装からアプリケーション インターフェイスを抽象化。 ◆ ほとんどのプログラミング言語、多数のアプリケーションがサポート。 ◆ リレーショナル アクセスのみ。 ◆ ラージ。一般的にネイティブ DBMS API への直接インターフェイスより低速。アプリケーションに「汎用」インターフェイスを提供。 ◆ 完全なリレーショナル インプリメンテーション。 ◆ ネイティブ機能のサブセット。 ◆ ほとんどすべての Windows プログラミング環境と多数の市販のアプリケーションでサポートされる標準インターフェイス。 	<ul style="list-style-type: none"> ◆ ささまざまなデータ ストアへの異種アクセスを必要とするアプリケーション、または特定のデータ ストアに依存しないアプリケーション。 ◆ リレーショナル データ ストアの保守を重視するが、なおかつ、実行時のパフォーマンスが重要であるアプリケーション。
PSQL Direct Access Components	<ul style="list-style-type: none"> ◆ Delphi および C++ Builder の Embarcadero Database Engine に取って代わる。 ◆ トランザクショナルまたはリレーショナル コンテキストからデータにアクセスするクラス。 	<ul style="list-style-type: none"> ◆ Embarcadero IDE を使用したアプリケーション開発。

データベース接続のクイック リファレンス

このセクションでは、PSQL データベースへの接続方法に関する情報を提供します。

これらの例は、各アクセス方法の完全なドキュメントを補足するだけのものです。各アクセス方法には、詳細情報へのリンクがあります。

各サンプルは、PSQL に含まれる DEMODATA サンプル データベースの **Course** テーブルを参照します。

- [「ADO.NET 接続」](#)
- [「ADO/OLE DB 接続」](#)
- [「JDBC 接続」](#)
- [「Java クラス ライブラリ」](#)
- [「DSN を使用しない接続」](#)

ADO.NET 接続

ADO.NET の詳細については、以下を参照してください。

- 『PSQL Data Provider for .NET Guide』の [「データ プロバイダーの使用」](#)
- サンプルのヘッダーとファイル（ADO.NET のサンプルとヘッダー ファイルのオンライン ダウンロード）をインストールした場合に提供されるコード例。

ADO.NET DB 接続のサンプル コード

```
"ServerDSN=Demodata;UID=test;PWD=test;ServerName=localhost;"
```

ADO/OLE DB 接続

ADO/OLE DB の詳細については、以下を参照してください。

- [「OLE DB プロバイダーの概要」](#)
- [「PSQL OLE DB プロバイダーによるプログラミング」](#)

ADO/OLE DB 接続のサンプル コード

```
Dim rs As New ADODB.Recordset
rs.Open "Course", "Provider=PervasiveOLEDB;Data Source=DEMODATA",
    adOpenDynamic, adLockOptimistic, adCmdTableDirect

' データを使用した処理

rs.Close
```

JDBC 接続

JDBC の詳細については、以下を参照してください。

- [「PSQL JDBC ドライバーの概要」](#)
- [「PSQL JDBC 2 ドライバーを使用したプログラミング」](#)

JDBC 接続のサンプル コード

```
Class.forName("com.pervasive.jdbc.v2.Driver");
Connection con = DriverManager.getConnection("jdbc:pervasive://localhost:1583/
    DEMODATA");
PreparedStatement stmt = con.prepareStatement("SELECT * FROM Course ORDER BY Name");
ResultSet rs = stmt.executeQuery();
```


Java クラス ライブラリ

Java クラス ライブラリの詳細については、以下を参照してください。

- 「[PSQL Java インターフェイスの概要](#)」
- 「[Java クラス ライブラリを使ったプログラミング](#)」

JCL 接続文字列のサンプル

```
Session session = Driver.establishSession();
Database db = session.connectToDatabase();
db.setDictionaryLoc("c:¥¥PVS¥¥DEMOMDATA");
```

DSN を使用しない接続

PSQL により、アプリケーションは DSN を使用しない接続（DSN を使用しないで SQL エンジンに接続する）を実行することができます。

サーバーでローカルに実行、またはリモート クライアントから実行するには、次の手順が必要です。この方法は、ワークステーション / ワークグループ エンジンだけでなく、サーバー エンジンでも機能します。

- 1 SQLAllocEnv
- 2 SQLAllocConnect
- 3 SQLDriverConnect:"Driver={Pervasive ODBC Client Interface};ServerName=<解決するサーバー名>;dbq=@<サーバー側の DBName>;"

例

```
Driver={Pervasive ODBC Client Interface};ServerName=myserver;dbq=@DEMOMDATA;
```



メモ Pervasive.SQL 2000 (SP3) より前のリリースでは、DSN を使用しない接続は、エンジンに対しローカルに実行されるアプリケーション（つまり、エンジンが実行されているのと同じマシン上で実行）でのみサポートされていました。しかし、ドライバー文字列の形式が Pervasive.SQL 2000i (SP3) から変更されました。DSN を使用しない接続を行うアプリケーションは、Pervasive.SQL 2000 (SP2a) 以前が適用された環境で DSN なしで実行するには、すべて上記のように変更し、再コンパイルする必要があります。

ODBC 情報

PSQL ODBC インターフェイスの機能および制限事項については、『ODBC Guide』で説明します。このマニュアルは、PSQL サーバーおよびワークグループ製品に付属しています。

- ODBC の詳細については、『ODBC Guide』の「[PSQL ODBC リファレンス](#)」を参照してください。
- サポートされる SQL 構文については、『SQL Engine Reference』の「[SQL 構文リファレンス](#)」を参照してください。

その他の SQL アクセス方法

ADO/OLEDB

ADO/OLE DB のプログラミング情報については、『OLE DB Provider Guide』を参照してください。

JDBC

SQL エンジンの JDBC プログラミングについては、『JDBC Driver Guide』を参照してください。

- 「[PSQL JDBC ドライバーの概要](#)」
- 「[PSQL JDBC 2 ドライバーを使用したプログラミング](#)」

PDAC

PSQL Direct Access Components は Delphi および C++ Builder アプリケーションに使用します。詳細については、『PSQL Direct Access Components Guide』で以下のトピックを参照してください。

- 「[Direct Access Components の使用方法](#)」
- 「[Direct Access Components リファレンス](#)」

アプリケーション開発のためのその他のリソース

以下のトピックでは、PSQL の概念に関するさらなる情報を提供します。

概念情報

このマニュアルでは、MicroKernel エンジンとリレーショナル エンジンの両方の概念について説明します。

リファレンス情報

開発者のためのリファレンス情報は、さまざまなソフトウェア開発キット（SDK）のマニュアルに含まれています。データベース エンジンと同時にインストール可能な Eclipse ヘルプでは、「開発者リファレンス」を参照してください。

開発者向けオンライン リソース

Actian PSQL Web サイトで、開発者向けのオンラインの情報を得られます。

サンプル コード

SDK コンポーネントのインストール場所に、サンプル アプリケーションが保存されています。利用できるサンプルには次のものが含まれます。

- ADO/OLE DB プログラミング（Visual Basic または Visual C++ を使用）
- PSQL Direct Access Components（Delphi と C++ Builder を使用）
- Java プログラミング（PSQL Java クラス ライブラリまたは JDBC を使用）
- Distributed Tuning Interface（Visual C++ または Delphi 用）
- Distributed Tuning Objects（Visual Basic 用）

MicroKernel エンジンのアプリケーション開発

3

この章では、PSQL の MicroKernel エンジンでアプリケーションを設計する際に考慮する必要がある情報を示します。これらの概念について、以下の各セクションで説明します。

- 「[MicroKernel エンジン環境](#)」
- 「[MicroKernel エンジンの設定に関する問題](#)」

MicroKernel エンジン環境

エンド ユーザーが MicroKernel エンジン アプリケーションを実行する前に、エンド ユーザーのコンピュータで MicroKernel エンジンのバージョンを使用できるようにする必要があります。アプリケーションに不可欠な MicroKernel エンジン ソフトウェア バージョンと環境設定に関する情報をエンド ユーザーに提供する必要があります。

ドキュメント

エンド ユーザーが以下の PSQL マニュアルを参照できるようにしてください。

- 『Getting Started With PSQL』。このマニュアルでは、PSQL ソフトウェアのインストールについて説明しています。
- 『Status Codes and Messages』。このマニュアルでは、PSQL コンポーネントが返すことのできるステータス コードとシステム メッセージについて説明しています。
- 『PSQL User's Guide』。このマニュアルでは、PSQL ユーティリティについて説明しています。

PSQL OEM パートナーであれば、作成したアプリケーションにこれらのマニュアルを添付できます。

MicroKernel エンジンの設定に関する問題

エンド ユーザーは、MicroKernel エンジン アプリケーションについて以下の情報を知っておく必要があります。この情報を MicroKernel エンジン アプリケーションに添付するドキュメントに記載してください。

■ アプリケーションに必要なメモリ容量。

アプリケーションには、MicroKernel エンジン自体に必要な容量より多いメモリまたはディスク領域が必要となる場合があります。アプリケーションのディスク領域とメモリ必要量を確定し、この情報をユーザーに伝えてください。MicroKernel エンジンのシステム要件については、『Getting Started With PSQL』および PSQL Web サイトを参照してください。

■ アプリケーションにデフォルト以外の MicroKernel エンジン構成の設定が必要かどうか。特に、エンド ユーザーがこれらの MicroKernel エンジン オプションを変更する必要があるかどうかを考慮してください。

- **作成ファイルのバージョン。**アプリケーションには MicroKernel エンジンの旧バージョンとの後方互換性が必要ですか？そうであれば、このオプションに対応する値を設定するようにエンド ユーザーに指示してください。
- **インデックス バランス。**アプリケーションは、作成するすべてのファイルにインデックス バランス ファイル属性を設定しますか？そうであれば、エンド ユーザーはインデックス バランスをデフォルトのオフにして使用できます。そうでなければ、MicroKernel レベルでインデックス バランスをオンにするようにエンド ユーザーに指示する必要があります。詳細については、「[インデックス バランス](#)」を参照してください。
- **最大圧縮レコード サイズ。**アプリケーションは圧縮されたレコードを使用しますか？そうであれば、『Advanced Operations Guide』の「[レコードおよびページ圧縮](#)」と、このマニュアルの「[ページサイズの選択](#)」、「[ファイルサイズの予測](#)」、および「[レコード圧縮](#)」を参照してください。
- **システム データ。**データベース内のすべてのファイルは重複のないキーを持っていますか？そうであれば、これらのファイルはトランザクション一貫性保守の機能を利用できます。そうでなければ、エンド ユーザーはファイルをトランザクション一貫保守性のあるものにするために [システム データの作成] を [必要な場合] または [常時] に設定しなければならない場合があります。

設定オプションの説明については、『Advanced Operations Guide』を参照してください。

MicroKernel エンジンの基礎

4

この章では、MicroKernel エンジンの機能について、以下の各セクションで説明します。

- 「[MicroKernel エンジンの概要](#)」
- 「[ページ](#)」
- 「[ファイル タイプ](#)」
- 「[データ型](#)」
- 「[キー属性](#)」
- 「[データベース URI](#)」
- 「[ダブルバイト文字のサポート](#)」
- 「[レコード長](#)」
- 「[データの整合性](#)」
- 「[イベント ログ](#)」
- 「[パフォーマンスの向上](#)」
- 「[ディスク使用量](#)」

MicroKernel エンジンの概要

Btrieve API は、MicroKernel エンジンの低レベル インターフェイスで、データベース設計の機能面を具現するものですが、SQL、Java、ODBC などの高レベル インターフェイスには透過的です。たとえば、SQL インターフェイスはデータが物理的にどのように格納されるかに関係なく動作します。しかし、MicroKernel エンジンの開発者はページ サイズ、物理および論理カレンシー、型検査、妥当性検査などの低レベルの側面を考慮する必要があります。これらの低レベルの側面を考慮しても、Btrieve API には優れた柔軟性とデータに対する制御性があります。

MicroKernel エンジンは情報をファイルに格納します。このファイルのサイズは PSQL バージョン 9.5 以降では最大 256 GB です (バージョン 9.5 までの 9.x は 128 GB で、それ以前のバージョンでは 64 GB です)。各データ ファイル内には、データ バイトを収容するレコードがあります。1 つのファイルに最大 40 億個のレコードを収容できます。

レコード内のデータは、社員の名前、ID、住所、電話番号、賃率などを表します。しかし、MicroKernel エンジンはレコードを単なるバイトの集合として解釈します。つまり、レコード内で論理的に区別されている情報を認識しません。MicroKernel エンジンにおいては、ラスト ネーム、ファースト ネーム、社員 ID などはレコード内に存在しません。

MicroKernel エンジンがレコード内で認識できる唯一の区別されている情報の部分は、**キー**です。キーは、レコードに対する高速な直接アクセスと、キー値によるレコードのソート手段を提供します。MicroKernel エンジンには各ファイル内のレコードの構造を理解する方法がないため、以下の項目を識別することによって各キーを定義します。

- **番号。**これは、キーのリスト内のキーの順序です。バージョン 6.0 以降のファイルでは、キー番号間にギャップをとることができます。つまり、MicroKernel エンジンは連続番号の付いたキーを必要としません。キーを追加する場合、キー番号を指定したり、MicroKernel エンジンに使用可能な最小キー番号を割り当てさせることができます。キーを削除する場合、残りのキー番号をそのままにしたり、MicroKernel エンジンに連続番号を付け直させることができます。
- **位置。**これは、キーのレコードの先頭からのオフセットです。
- **長さ。**これは、キーに使用するバイト数です。
- **型。**これは、キーのデータ型です。
- **属性。**これは、MicroKernel エンジンにキー値を処理させる方法に関する追加情報を示します。MicroKernel エンジンは、セグメント、重複可能、変更可能、ソート順、大文字と小文字の区別、オルタネート コレーティング シーケンス、ヌル値などのキー属性をサポートします。

キーはいつでも作成または削除することができます。MicroKernel エンジンは、データ ファイルで定義されたキーごとに**インデックス**を作成します。インデックスは、データ ファイル自体の内部に格納されます。インデックスは、ファイル内の各キー値を実際のデータ内のオフセットへマップします。通常は、MicroKernel エンジンがデータのアクセスまたはソートを行う場合、ファイル内のすべてのレコードを検索しません。その代わりに、インデックスを検索し、その後要求に合うレコードだけを処理します。

インデックスは、データ ファイルの作成時、またはそれ以降いつでも作成できます。データ ファイルを作成するときは、MicroKernel エンジンがインデックスの作成に使用するキーを 1 つまたは複数定義できます。

ファイルを作成した後に、**外部インデックス**を定義することもできます。外部インデックス ファイルは、指定するキーでソートされたレコードを含む標準データ ファイルです。各レコードは、以下の項目から成ります。

- 元のデータ ファイルにおけるレコードの物理位置を識別する 4 バイト アドレス
- キー値

いつインデックスを作成するかに関係なく、ポジショニングの規則（どのレコードが現在のレコードで、どのレコードが次レコードかなどを管理するガイドライン）は同じです。

ファイルを作成すると同時にインデックスを作成する場合、MicroKernel エンジンはレコードがファイルに挿入された年代順に重複キー値を格納します。既に存在するファイルのインデックスを作成する場合、MicroKernel エンジンは、インデックス作成時の対応するレコードの物理順で重複キー値を格納します。MicroKernel エンジンがインデックスに重複キー値を格納する方法は、キーがリンク重複か繰り返し重複かによっても異なります。詳細については、「[重複可能性](#)」を参照してください。



メモ レコードの年代順は、レコードを更新してそのキー値を変更する場合、インデックスを削除して作成し直す場合、またはファイルを作成し直す場合に变化する可能性があります。したがって、ファイル内のレコードの順序がレコードの挿入された順序を常に反映するとは考えないでください。レコード挿入順序をトラッキングする場合は、AUTOINCREMENT キーを使用してください。

アプリケーションでインデックスが不要になれば、インデックスを削除できます。データ ページやその他のインデックス ページに対して、インデックスがファイル内で使用した領域が解放されます。(ただし、この空き領域はファイルに割り当てられたままになります。インデックスを削除した後に物理的なファイル サイズは減少しません。)

キーの定義に関する具体的な情報については、第 5 章「[データベースの設計](#)」を参照してください。

MicroKernel エンジン環境

エンド ユーザーが MicroKernel エンジン アプリケーションを実行する前に、エンド ユーザーのコンピューターで MicroKernel エンジンのバージョンを使用できるようにする必要があります。アプリケーションに不可欠な MicroKernel エンジン ソフトウェア バージョンと環境設定に関する情報をエンド ユーザーに提供する必要があります。

設定に関する注記

エンド ユーザーは、MicroKernel エンジン アプリケーションについて以下の情報を知っておく必要があります。この情報を MicroKernel エンジン アプリケーションに添付するドキュメントに記載してください。

■ アプリケーションに必要なメモリ容量。

アプリケーションには、MicroKernel エンジン自体に必要な容量より多いメモリまたはディスク領域が必要となる場合があります。アプリケーションのディスク領域とメモリ必要量を確定し、この情報をユーザーに伝えてください。MicroKernel エンジンのシステム要件については、『[Getting Started With PSQL](#)』および [PSQL Web サイト](#)を参照してください。

■ アプリケーションにデフォルト以外の MicroKernel エンジン構成の設定が必要かどうか。特に、エンド ユーザーがこれらの MicroKernel エンジン オプションを変更する必要があるかどうかを考慮してください。

- **作成ファイルのバージョン。**アプリケーションには MicroKernel エンジンの旧バージョンとの後方互換性が必要ですか？そうであれば、このオプションに対応する値を設定するようにエンド ユーザーに指示してください。
- **インデックス バランス。**アプリケーションは、作成するすべてのファイルにインデックス バランス ファイル属性を設定しますか？そうであれば、エンド ユーザーはインデックス バランスをデフォルトのオフにして使用できます。そうでなければ、MicroKernel レベルでインデックス バランスをオンにするようにエンド ユーザーに指示する必要があります。詳細については、「[インデックス バランス](#)」を参照してください。
- **最大圧縮レコード サイズ。**アプリケーションは圧縮されたレコードを使用しますか？そうであれば、『[Advanced Operations Guide](#)』の「[レコードおよびページ圧縮](#)」と、このマニュアルの「[ページサイズの選択](#)」、「[ファイルサイズの予測](#)」、および「[レコード圧縮](#)」を参照してください。
- **システム データ。**データベース内のすべてのファイルは重複のないキーを持っていますか？そうであれば、これらのファイルはトランザクション一貫性保守の機能を利用できます。そうでなければ、エンド ユーザーはファイルをトランザクション一貫保守性のあるものにするために [システム データの作成] を [必要な場合] または [常時] に設定しなければならない場合があります。

設定オプションの説明については、『[Advanced Operations Guide](#)』を参照してください。

ページ

ここでは、ページと MicroKernel エンジンによるページの処理方法に関する以下の情報を示します。

- 「[ページ タイプ](#)」
- 「[ページ サイズ](#)」

ページ タイプ

ファイルは、一連のページから構成されています。ページとは、データベースがメモリとディスクの間で転送する記憶容量の単位です。ファイルは、以下のページ タイプから構成されています。

ファイル コントロール レコード (FCR)	ファイルのファイル サイズ、ページ サイズ、その他の特性などのファイルに関する情報が含まれています。6.0 以降のすべてのデータ ファイル内の最初の 2 つのページは FCR ページです。MicroKernel エンジンは常に、FCR ページのうちの 1 ページを現在のページと見なします。現在の FCR ページには、最新のファイル情報が含まれています。
ページ アロケーション テーブル (PAT)	ファイル内のページを追跡するために使用される、MicroKernel エンジンの内部的な実装の一部。
データ	レコードの固定長部分が含まれています。MicroKernel エンジンは、1 つの固定長レコードを 2 つのデータ ページにまたがって分割しません。ファイルが可変長レコードを許可していないか、データ圧縮を使用しない場合、ファイルにはデータ ページはありますが可変ページはありません。
可変	レコードの可変長部分が含まれています。レコードの可変長部分が可変ページの残りの領域より長い場合、MicroKernel エンジンは複数の可変ページにわたって可変長部分を分割します。ファイルが可変長レコードを許可しているか、データ圧縮を使用する場合、ファイルにはデータ ページと可変ページの両方があります。
インデックス	レコードの検索で使用されるキー値が含まれています。
オルタネート コレーティング シーケンス (ACS)	ファイル内のキーのオルタネート コレーティング シーケンスが含まれています。

6.0 以降のすべてのファイルには、FCR ページと PAT ページがあります。標準ファイルにはデータ ページとインデックス ページも含まれており、オプションとして可変ページと ACS ページが含まれています。「[データオンリー ファイル](#)」にはインデックス ページが含まれていません。「[キーオンリー ファイル](#)」にはデータ ページが含まれていません。

ページ サイズ

ファイルを作成するときに固定ページ サイズを指定します。指定できるページ サイズやファイル オーバーヘッドなどは、ファイル形式をはじめさまざまな要因によって異なります。ページ サイズの説明については、第 5 章「[データベースの設計](#)」を参照してください。以下のセクションでは概要について説明します。

- 「[ページ サイズの基準](#)」
- 「[大きなページ サイズと小さなページ サイズ](#)」

ページ サイズの基準

指定するページ サイズは、以下の基準を満たす必要があります。

- ファイルのレコード長に相応なデータ ページを使用可能にする。
各データ ページには、一定のバイト数のオーバーヘッド情報が含まれています。表 19 を参照してください。その後、MicroKernel エンジンは各データ ページにできるだけ多くのレコードを格納しますが、ページにまたがってレコードの固定長部分を分割することはありません。

最適なページサイズでは、各データ ページの残余スペース量をできるだけ少なくしながら、最も多いレコードを収容できます。ページ サイズが大きいほど、通常はディスク領域の使用効率が高くなります。内部レコード長（ユーザー データ + レコード オーバーヘッド）が小さく、ページ サイズが大きいと、無駄な領域がかなりの量になる可能性があります。

- ファイルのキー定義に適合するインデックス ページを許可する。

各インデックス ページには、一定のバイト数のオーバーヘッド情報が含まれています。表 19 を参照してください。その後、ファイルのインデックス ページは 8 個のキーと各キーのオーバーヘッド情報を収容できる大きさがなければなりません（設定ごとのオーバーヘッドのバイト数の説明については、表 17、18、19、20 および 21 を参照してください）。

- ファイルが必要とするキー セグメント数を許可する。

「セグメント化」で説明したように、ファイルに対して定義するページ サイズはそのファイルに対して指定できるキー セグメント数を制限します。

- パフォーマンスを最適化する。

パフォーマンスを最適化するには、ページ サイズを 2 のべき乗のサイズ、つまり、512 バイト、1024 バイト、2048 バイト、4096 バイト、8192 バイト、16384 バイトに設定します。MicroKernel エンジンの内部キャッシュは一度に複数のサイズのページを格納できますが、2 のべき乗単位に分割されます。ページ サイズ 1536、2560、3072 および 3584 では、実際に、MicroKernel エンジン キャッシュ内のメモリが無駄になります。2 のべき乗のページ サイズを使用すると、キャッシュの使用効率が良くなります。

大きなページ サイズと小さなページ サイズ

現代のオペレーティング システムを最も効率よく使用するには、より大きなページ サイズを選択する必要があります。DOS が傑出したオペレーティング システムであった時代、つまり、セクターが 512 バイトで、すべての I/O が 512 の倍数で発生していたときは、小さなページ サイズが使用されていました。現在はそうではありません。32 ビットおよび 64 ビットのオペレーティング システムでは、いずれもデータを 4096 バイトあるいはそれ以上のブロック単位でキャッシュに移動させます。CD ROM ドライブは、2048 バイト単位で読み取られます。

MicroKernel エンジンのインデックスは、4096 バイトあるいはそれ以上のページ サイズを使用する場合に最も効率が良くなります。キーはノードごとにさらに多くの分岐を持つため、正しいレコード アドレスを検索するための読み取りは少なく済みます。このことは、アプリケーションがキーでランダムな読み取りを行っている場合に重要です。アプリケーションがキーまたはレコードでファイルに順次アクセスする場合は重要ではありません。

ページを小さくするもっともな理由は、競合を避けるためです。各ページのレコード数が少ないほど、さまざまなエンジンまたはトランザクションが同時に同じページを要求する可能性は低くなります。ファイルのレコード数が比較的少なく、レコードが小さい場合は、小さなページ サイズを選択できます。ファイルが大きいほど、競合の発生する可能性が低くなるようです。

大きなページ サイズのもう 1 つの潜在的な問題は、バージョン 7.0 以降のファイルに特有なものです。同じデータ ページに収容できるレコードまたは可変長セクションの数は最大 256 個です。短いレコード、圧縮されたレコード、または短い可変長セクションがある場合は、すべてのページにまだ数百バイト残っていても、すぐに制限に達する可能性があります。その結果、必要とされるよりもはるかに大きなファイルになります。レコード サイズがわかれば、このことがどの程度大きな問題かを計算できます。

ページ サイズの決定時に考慮する要素

- キーはページ サイズが大きいほど効率よく働きます。B ツリーのノードごとにより多くの分岐ができるため、B ツリーのレベルが少なくなります。レベルが少ないとは、ディスクの読み込みや書き込みが少ないことです。ディスク読み込みが少ないと、パフォーマンスは向上します。

- 並行処理は、クライアント トランザクションが使用されている場合は特に、**より小さなページ**の方が効率が良くなります。トランザクション時、MicroKernel エンジン は変更されるページをロックするので、ほかのすべてのクライアントは、トランザクションが終了または中止するまで、ロックされたページが解除されるのを待つ必要があります。たくさんのクライアントが並行して同一ページにアクセスを試みる場合、各ページで見つけられるデータが少ないほど効率が良くなります。
- ページへのランダム アクセスは、**より小さなサイズ**のページの方が効率が良くなります。これは、実際に使用するデータがキャッシュにあることが多いためです。再度データにアクセスした場合、十中八九まだキャッシュ内にあるでしょう。
- 大量のレコードにシーケンシャル アクセスする場合は、**大きなサイズ**のページの方が一度により多くのレコードを読み取れるため、効率が良くなります。ページを読み取るたびにほとんどすべてを使用するので、確実に読み取り回数が少なくなります。

データベース設計者はこれらの相反するニーズの中から選択しなければなりません。参照テーブルはめったに変更されませんが、ほとんど常に検索またはスキャンされるため、大きなページ サイズにする必要があります。トランザクション内で頻繁に追加および更新されるトランザクション ファイルは、小さいページ サイズにする必要があります。

すべての要因を慎重に検討することによって、どのようなページ サイズにするのかに対する正しい答えを導くことができます。ページ サイズの選択の詳細については、「[ページ サイズの選択](#)」を参照してください。

ファイル タイプ

Btrieve API は、PSQL 9.5 以降のデータ ファイルでは最大ファイル サイズ 256 GB (9.x から 9.5 までのバージョンでは 128 GB、それ以前のバージョンでは 64 GB) をサポートし、長いファイル名、および 3 つのデータ ファイル タイプをサポートしています。

- 「標準データ ファイル」
- 「データオンリー ファイル」
- 「キーオンリー ファイル」
- 「ラージ ファイル」
- 「長いファイル名」



メモ Btrieve 6.x およびそれ以前のユーザーのために、PSQL はファイルを 8.x および 7.x 形式で作成することができます。これらの新しい形式により、拡張性と新機能がもたらされました。

Btrieve 6.x およびそれ以前のバージョンは、PSQL 7.x または 8.x ファイルを開けません。しかし、PSQL v12 はバージョン 7.0 より前のファイルを開くことができます。PSQL v12 はバージョン 7.0 より前のファイルを開く際、ファイルを 7.0 または 8.0 の形式に**変換しません**。また、バージョン 8.0 より前の形式でファイルを作成するように PSQL を設定することもできます。これは、新しく作成された V8 より前のファイルを使用する場合に役立ちます。

標準データ ファイル

標準 7.x 以降のファイルには、FCR の 2 ページに続けて多数の PAT ページ、インデックス ページ、データ ページがあり、ファイルによっては可変ページや ACS ページもあります。固定長レコードまたは可変長レコードで使用する標準ファイルを作成できます。標準ファイルにはすべてのインデックス構造とデータ レコードが含まれているので、MicroKernel エンジンではファイル内のレコードに関するすべてのインデックス情報を動的に保守できます。

データオンリー ファイル

データオンリー ファイルを作成する場合には、キー情報を何も指定しないので、PSQL はファイルのインデックス ページを割り当てません。このため、初期のファイル サイズは標準ファイルの場合より小さくなります。データオンリー ファイルを作成後、ファイルにキーを追加できます。

キーオンリー ファイル

キーオンリー ファイルには、FCR ページに続けて多数の PAT ページとインデックス ページだけが含まれています。(また、ファイルに参照整合性制約を定義した場合は、1 ページ以上の可変ページが含まれます。)

キーオンリー ファイルはキーを 1 つだけ含み、レコード全体がそのキーと共に格納されるため、データ ページは不要です。キーオンリー ファイルは、レコードに単一のキーが含まれており、かつそのキーが各レコードの大部分を占有している場合に有効です。キーオンリー ファイルのもう 1 つの一般的な用途は、標準データ ファイルの外部インデックスとして使用する場合があります。

キーオンリー ファイルには以下の制限が適用されます。

- 各ファイルには 1 つのキーしか含められません。
- 定義できる最大レコード長は 253 バイト (バージョン 6.0 より前は 255 バイト) です。
- キーオンリー ファイルではデータ圧縮を行えません。

ラージ ファイル

MicroKernel エンジンでは、PSQL 9.5 以降では最大 256 GB（9.x から 9.5 までのバージョンでは 128 GB、それ以前のバージョンでは 64 GB）までのファイル サイズをサポートしています。ただし、多くのオペレーティング システムでは、単一のファイルでこれだけの大きなサイズをサポートしていません。オペレーティング システムのファイル サイズの制限より大きいファイルをサポートするために、MicroKernel エンジンでは大きなファイルをオペレーティング システムでサポートできるさらに小さなファイルに分割します。大きな論理ファイルは、**拡張ファイル**と呼ばれます。拡張ファイルを構成するさらに小さい物理ファイルは、**エクステンション ファイル**と呼ばれます。**ベース ファイル**は、大きすぎて 1 つの物理ファイルとしてサポートできなくなった元のデータ ファイルです。非拡張（非セグメント化）ファイルは、より効率的な I/O を提供するので、パフォーマンスが向上します。

PSQL 9.x 以降のファイルが自動的に 2 GB ごとに**拡張されない**ようにすることができます。セグメント操作の設定を変更するには、『Advanced Operations Guide』の「**PCC による設定**」に記述されているように、PCC（PSQL Control Center）の設定プロパティにアクセスします。ここで「**セグメント サイズを 2 GB に制限**」オプションを設定することができます。

このオプションが選択されていない場合、PSQL 9.x 以降のファイルが自動的に 2 GB にセグメント化されることはありません。バージョン 8.x およびそれ以前のデータ ファイルは、引き続き 2 GB に到達するごとに拡張されます。ファイルが既に拡張されている場合は、セグメント化されたままです。

設定プロパティに関わらず、すべてのファイルは現在のオペレーティング システムのファイル サイズ制限に基づいて引き続き拡張されます。

拡張ファイルなどのファイルのバックアップについては、「[ファイルのバックアップ](#)」を参照してください。

長いファイル名

MicroKernel エンジンでは 255 バイト以下の長いファイル名をサポートします。以下の項目もこの上限に従います。

- ローカライズされたマルチバイトまたはシングル バイト バージョンの文字列。
- リクエスターによって作成された UTF-8 UNICODE 形式の UNC バージョンのファイル名。

[スペースを含むファイル/ディレクトリ名] クライアント設定オプションが無効になっていない限り、ファイル名にはスペースを含めることができます。デフォルト設定は、**オン**です。『Advanced Operations Guide』の「[長いファイル名と埋め込みスペースのサポート](#)」を参照してください。

「**ラージ ファイル**」を使用する場合やアーカイブ ロギングまたは Continuous オペレーションの実行中など、MicroKernel エンジンが既存のファイル名に基づいて新しいファイルを作成する場合は（『Advanced Operations Guide』の第 9 章「[ログ、バックアップおよび復元](#)」を参照）、以下の例に示すように、新しいファイル名には元のファイル名のできるだけ多くの部分を含み、特有のファイル拡張子を使用します。

元のファイル名	Continuous オペレーションで作成されたファイル名
LONG-NAME-WITHOUT-ANY-DOTS	LONG-NAME-WITHOUT-ANY-DOTS.^^^
VERYLONGNAME.DOT.DOT.MKD	VERYLONGNAME.DOT.DOT.^^^

データ型

7.x 以降のファイル形式を使用する場合、キーを定義する際には以下のデータ型を使用できます。

AUTOINCREMENT	BFLOAT	CURRENCY
DATE	DECIMAL	FLOAT
INTEGER	LSTRING	MONEY
NUMERIC	NUMERICSA	NUMERICSTS
TIME	TIMESTAMP	UNSIGNED BINARY
ZSTRING	WSTRING	WZSTRING
NULL INDICATOR		

6.x ファイル形式を使用している場合は、上記のうち CURRENCY および TIMESTAMP を除くすべてのデータ型を使用してキーを定義できます。

6.x より前の形式を使用している場合、NUMERICSA および NUMERICSTS はデータ型やキー タイプとして使用できません。

データ型の詳細については、『SQL Engine Reference』の「[データ型](#)」を参照してください。

キー属性

以下のセクションでは、キーを定義するときに指定できる属性について説明します。

- 「[キー属性の解説](#)」
- 「[キー仕様](#)」

キー属性の解説

このセクションでは、キーに割り当てられる以下の属性について説明します。

- 「[セグメント化](#)」
- 「[重複可能性](#)」
- 「[変更可能性](#)」
- 「[ソート順序](#)」
- 「[大文字と小文字の区別](#)」
- 「[ヌル値](#)」
- 「[オルタネート コレーティング シーケンス](#)」

セグメント化

キーは、各レコード内の 1 つ以上の[セグメント](#)から構成できます。レコード内の任意の連続バイトをセグメントとすることができます。キー タイプとソート順序は、キー内のセグメントごとに変えることができます。

使用できるインデックス セグメントの数はファイルのページ サイズによって異なります。

ページ サイズ (バイト数)	キー セグメントの最大数 (ファイル バージョン別)		
	8.x 以前	9.0	9.5
512	8	8	切り上げ ²
1024	23	23	97
1536	24	24	切り上げ ²
2048	54	54	97
2560	54	54	切り上げ ²
3072	54	54	切り上げ ²
3584	54	54	切り上げ ²
4096	119	119	119 または 204 ³
8192	N/A ¹	119	119 または 420 ³
16384	N/A ¹	N/A ¹	119 または 420 ³

¹ N/A は「適用外」を意味します。

² 「切り上げ」は、ページ サイズを、ファイル バージョンでサポートされる次のサイズへ切り上げることを意味します。たとえば、512 は 1024 に切り上げられ、2560 は 4096 に切り上げるということです。

³ リレーショナル エンジンで使用できるインデックス セグメントの最大数は 119 です。MicroKernel エンジンの場合、最大数は、ページ サイズ 4096 では 204、ページ サイズ 8192 および 16384 では 420 です。

インデックス セグメントと MicroKernel エンジンに関する詳細については、ステータス コードの「26: 指定されたキーの数が不正です。」および「29: キー長が不正です。」を参照してください。

キーの合計長はキー セグメントの長さの合計であり、最大長は 255 バイトです。レコード内でキー セグメントは互いに重なり合っている場合でもかまいません。

セグメント化されたキーが重複不能キーである場合、セグメントを組み合わせで一意的な値をつくる必要がありますが、個々のセグメントには重複を含めることができます。このタイプのセグメント化されたキーを定義する場合、たとえ特定セグメントに重複がある可能性があっても、各セグメントにはキー レベル属性として `duplicates=no` を持ちます。特定セグメントが常に一意であるようにするには、セグメント化されたキー定義の他に個別の重複不能キーとしてそのセグメントを定義します。

MicroKernel エンジン呼び出しを行う場合、キー バッファの形式はキー番号で指定されるキーを収容できなければなりません。したがって、`keynumber=0` を定義した場合、キー 0 が 4 バイト整数ならば、キー バッファパラメータは以下のいずれかにすることができます。

- 4 バイト整数へのポインター
- 最初の要素または唯一の要素が 4 バイト整数である構造体へのポインター
- 4 バイト以上の文字列またはバイト配列へのポインター

基本的には、MicroKernel エンジンはキー バッファとして使用されるメモリの場所へのポインターを取得します。MicroKernel エンジンはそのメモリの場所に、`Get Equal` などのオペレーションで指定されたキー番号に対応するデータ値があることを期待します。また、MicroKernel エンジンはその場所にデータを書き込むことができ、書き込んだデータが指定されたキー番号に対応するキー値になります。このような場合は、キー値全体を収容できる十分なメモリの場所を割り当てる必要があります。

MicroKernel エンジンにとって、キーは、たとえ複数のセグメントから構成されていても、単一のデータの集合です。セグメント機能を使用すれば、連続していないデータのバイトを 1 つのキーとして結合できます。また、キーデータの個々の部分に異なるソート規則（サポートされるデータ型で指定されているとおり）を適用できます。一般的に、1 つのキー セグメントに関連付けられているデータ型がソート規則として使用されます。ソート規則は、2 つの値を比較してどちらの値が大きいか決定する方法を MicroKernel エンジンに指示します。データ型は、データの妥当性検査に使用されません。

MicroKernel エンジンは、常にキー セグメントでなくキー全体を処理します。キーを処理するには、キー全体を保持できる十分な大きさのキー バッファを設定します。アプリケーションの中には、すべての MicroKernel エンジン呼び出しで使用する汎用の 255 バイトのバッファを定義しているものがあります。キーの最大サイズは 255 バイトですから、十分なサイズです。このキー バッファにデータが返されたら、アプリケーションでは通常、キー セグメントと同じ型として宣言されているアプリケーションの変数または構造体に汎用バッファのデータをコピーします。別の方法として、キーに直接対応するキー バッファ パラメータ（単純変数または構造体変数）を渡します。

たとえば、レコードを読み取りたいが、キーのすべてのセグメントでなく最初のセグメントの値しかわからないものとして扱います。それでも、そのキーを使用してデータを検索することができます。ただし、すべてのセグメントに対応するキー バッファ全体を渡す必要があります。キー値の一部しかわからないので、`Get Equal` 呼び出しを使用できません。`Get Greater Or Equal` 呼び出しを使用する必要があります。この場合、わかっているだけのキー値でキー バッファを初期化し、次に不明のキー セグメントに低い値またはヌル値を指定します。

たとえば、データ値 `ulElement2`、`ulElement3`、`ulElement5` に相当する 3 つのセグメントから成る、キー 1 の定義があるとします。`ulElement2` に必要な値がわかっている場合、キー バッファを次のように初期化します。

```
SampleKey1.ulElement2 = < 検索する値 >;
SampleKey1.ulElement3 = 0;
SampleKey1.ulElement5 = 0;
```

次に、`Get Greater Or Equal` 呼び出しでキー バッファ パラメータとして `&SampleKey1` を渡します。MicroKernel エンジンが呼び出しを終了し、レコードが見つかった場合には、ステータス コード 0 が返され、対応するデータレコードが返されて、キー バッファに 3 つのセグメントすべてを含むキー値が設定されます。

重複可能性

PSQL は重複キー値を処理する方法として、リンク（デフォルト）および繰り返しの 2 つをサポートします。リンク重複キーでは、MicroKernel エンジンはインデックス ページの 1 組のポインターを使用して、同じキー値を持つレコードのうち年代順に最初と最後のレコードを識別します。さらに、MicroKernel エンジンはデータ ページの各レコード内の 1 組のポインターを使用して、同じキー値を持つレコードのうち年代順に前のレコードと次のレコードを識別します。キー値は、インデックス ページにのみ 1 回格納されます。

繰り返し重複キーでは、MicroKernel エンジンはインデックス ページの 1 つのポインターを使用して、データ ページの対応するレコードを識別します。キー値は、インデックス ページとデータ ページの両方に格納されます。重複キーの詳細については、「[重複キー](#)」を参照してください。

変更可能性

キーを変更可能キーとして定義すると、MicroKernel エンジンはレコードが挿入された後でもキーの値を変更できるようにします。キーの 1 つのセグメントが変更可能であれば、すべてのセグメントが変更可能でなければなりません。

ソート順序

デフォルトでは、MicroKernel エンジンはキー値を昇順（最小の値から最大の値へ）にソートします。しかし、MicroKernel エンジンがキー値を降順（最大の値から最小の値へ）に並べるように指定することができます。



メモ MicroKernel エンジンの Get オペレーション（Get Greater (8)、Get Greater or Equal (9)、Get Less Than (10) および Get Less Than or Equal (11)）で降順キーを使用するときは注意してください。この場合、Greater または Less はキーに関する順序を参照するため、降順キーの場合には、この順序は対応する昇順キーの逆になります。

降順キーで Get Greater オペレーション (8) を実行する場合、MicroKernel エンジンはキー バッファで指定するキー値より小さい最初のキー値に対応するレコードを返します。たとえば、10 件のレコードと整数型の降順キーを持つファイルについて考えてみましょう。10 件のレコードの降順キーに格納されている実際の値は、整数 0、1、2、3、4、5、6、7、8、9 です。現在のレコードのキー値が 5 で Get Greater オペレーションを実行した場合、MicroKernel エンジンはキー値 4 を含むレコードを返します。

同様に、降順キーで Get Less Than オペレーション (10) を実行した場合、MicroKernel エンジンはキー バッファで指定するキー値より次に大きい値を持つレコードを返します。前の例で、現在のレコードの降順キーの値が 5 で Get Less Than オペレーションを実行した場合、MicroKernel エンジンはキー値 6 を含むレコードを返します。

大文字と小文字の区別

デフォルトでは、MicroKernel エンジンは文字列キーをソートするときに大文字と小文字を区別します。つまり、小文字の前に大文字をソートします。キーを大文字小文字無視と定義すると、MicroKernel エンジンは大文字と小文字を区別せずに値をソートします。キーにオルタネート コレーティング シーケンス (ACS) がある場合、大文字と小文字の区別は適用されません。

ヌル値

PSQL v12 には、列のデータをヌル値として識別する方法が 2 つあります。ヌル値の元のタイプ（レガシー ヌルと呼びます）は、MicroKernel エンジンで長年使用されてきました。新しいタイプのヌル識別は、真のヌルと呼ばれます。このセクションでは、レガシー ヌルについて簡単に説明し、次に MicroKernel エンジンでの真のヌルの使用について詳細に説明します。

レガシー ヌル

ヌル値を許可するフィールドを定義する元の方法は「**擬似ヌル**」または「**レガシー ヌル**」と呼びます。これは、フィールド全体が特定のバイト値、一般的には ASCII ゼロで埋められている場合に、そのフィールドをヌルと見

なすという前提に基づいています。バイト値は、インデックス作成時に指定するキー定義に指定します。MicroKernel エンジンを使用する場合、MicroKernel エンジンがこの情報から行えることは、フィールドをインデックスに含めるか含めないかだけです。レガシーヌルは、その特殊な意味にもかかわらず、その他すべての値とまったく同様にソートされる値であるため、特別なソート規則はありません。

キー定義に「全セグメントヌル」(0x0008)のフラグが含まれている場合、キー内のすべてのセグメントがヌルと見なされると、そのキー値はインデックスに含められません。各セグメントがヌルと見なされるのは、フィールド内のすべてのバイトが「ヌルバイト」である場合です。同様に、キー定義に「一部セグメントヌル」(0x0200)のフラグが含まれている場合、キー内の1つ以上のセグメントがヌルと見なされると、そのキー値はインデックスに含められません。各セグメントがヌルと見なされる規則は前と同じです。

リレーショナルエンジンは、エンジンが定義するインデックス内でこれらのフラグを使用しません。リレーショナルエンジンは、テーブル間の結合を作成するために、インデックスを介してテーブル内のすべてのレコードにアクセスできる必要があるため、これらのフラグは使用しません。

真のヌル インデックス

Pervasive.SQL 2000 以降では、「真のヌル」と呼ぶ新しいタイプのヌル インジケータが導入されています。

真のヌルは、ヌルを許可する列の直前に1バイトのヌル インジケータ セグメント (NIS) を置くことにより MicroKernel エンジンに実装されます。これは、その列がヌルかどうかを示すために通常の列幅に追加された特別なバイトです。このバイトの値がゼロであると、このインジケータが関連付けられている列が通常の列、つまりヌルでないことを示します。このバイトがその他の値である場合は、その列の値がヌルであることを示します。

真のヌルを使用すると、レガシーヌルとは異なり、値がゼロの整数とヌルの整数を区別することができます。これはその他の数値フィールドにもあてはまります。このような区別が必要のない場合には、長さ0の文字列フィールドはヌルであると判別することができます。

リレーショナルエンジンは、列にインデックスが定義されているかどうかにかかわらず、真のヌル列を識別し使用することができます。ただし、基本のデータファイルは、キーに含まれるフィールドしか識別することができません。

「Create (14)」または「Create Index (31)」オペレーションのキー定義で、ヌル値を許可する列の前にヌル インジケータ セグメント (NIS) を追加することによって、MicroKernel エンジンのキー内に真のヌル フィールドを定義することができます。真のヌル キーに関する規則については、「[真のヌル キーの規則](#)」を参照してください。

MicroKernel エンジンでは NIS のオフセットに関する制約はありませんが、リレーショナル エンジンでは NIS はヌル許可列の直前にあるものと見なします。そのため、レコード内のフィールドを構造化する際は、NIS を使用する可能性のあるすべてのフィールドの前のバイトを、NIS を置く場所として空けておくことをお勧めします。こうしておくことにより、必要になった場合、これらのテーブルに SQL を介してアクセスする能力を維持することができます。

真のヌル キーの規則

この新しいキー タイプを使用するときは、次の規則に従う必要があります。

- 1 フィールド長は1である必要があります。
- 2 このフィールドは、インデックス内の同類のフィールドの前にある必要があります。言い換えると、複数セグメントのインデックスでは、同類のセグメントの直前に NIS が定義されている必要があります。NIS を最終セグメントまたは唯一のセグメントにすることはできません。
- 3 NIS の直後のフィールドは NIS の内容の影響を受けます。NIS がゼロの場合、直後のフィールドは非ヌルと見なされます。NIS がゼロ以外の場合、直後のフィールドはヌルと見なされます。
- 4 NIS のオフセットはその次のフィールドの直前のバイトである必要があります。これが、PSQL リレーショナル エンジンがこれらのフィールドの配置として要求する様式です。したがって、このインデックス用のデータ辞書を作成する場合、NIS は制御するフィールドの直前にある必要があります。ただし、トランザクショナル API にはこれを必要条件とするものは何もありません。

NIS 値

ゼロ以外の値はすべて、次のセグメントがヌルであることを表すインジケータと見なされます。デフォルトで、MicroKernel エンジンではゼロ以外の値の間の区別は行いません。PSQL リレーショナル エンジンでは現在、このフィールドがヌルであることを示すのに値 1 のみを使用します。ただし、異なるタイプのヌルを区別することはできます。これは、NIS に「大文字小文字無視」フラグを使用して行います。このキー フラグは通常、さまざまな文字列フィールドや文字フィールドにのみ適用可能であるため、NIS で使用した場合、DISTINCT に特別な意味を持たせるためにオーバーロードされます。つまり、異なる NIS 値は区別して扱われ、別個にソートされます。Actian Corporation は、将来の拡張のため最初の 15 の値を予約しています。アプリケーションでさまざまなタイプのヌルに特別な意味を持たせたい場合は、NIS に 16 より大きい値を使用してください。たとえば、より詳細なヌル定義は次のようになります。

- 適用外
- 未定
- 決定不能
- 検出不能
- 必須（未定）

NIS に DISTINCT フラグ（大文字小文字無視）を追加した場合、これらの非ゼロ値は異なる値として別個にソートされます。

真のヌル値のソート

真のヌルフィールドの値は非決定です。言い換えると、その値は不明です。この定義によれば、どの 2 つのヌル値も互いに等しくなく、ほかのすべてのキー値とも等しくありません。それでも、MicroKernel エンジンではこれらのヌル値を 1 つにまとめる必要があり、ヌルに等しいキー値を見つけることができなければなりません。これを行うため、MicroKernel エンジンでは比較の目的に合わせて、真のヌル値がそれぞれ等しいかのように解釈します。ソートしたり、インデックス内でヌル値の場所を検索したりする場合、真のヌル値は互いに等しいかのようにグループ化されます。しかし、重複のないインデックス内に値が既に存在するかどうかを判断しようとしているときは、真のヌルは互いに等しくなくなります。

NIS 内の非ゼロ値はすべて、直後のフィールドがヌルであることを示します。デフォルトの動作では、NIS 内の非ゼロ値はすべて同一値のように扱われ、ヌル許可フィールドがヌルであることを示していると解釈されます。したがって、NIS にさまざまな非ゼロ値を含み、それに続くヌル許可フィールドがさまざまな値を含むレコードを挿入した場合、これらは同一の値として解釈され、重複値の集まりとしてソートされます。

リンク重複キーと真のヌル

このセクションでは、リンク重複キーにいくつかのヌル値を挿入した結果について説明します。

リンク重複キーは、一意の値ごとに単一のキー エントリ、2 つのレコード アドレス ポインターを持ちます。レコード アドレス ポインターの 1 つは最初の重複レコード用で、もう 1 つは重複の連鎖の最後のレコード用です。各レコードには、連鎖内の前および次のレコードへのポインターから成る 8 バイトのオーバーヘッドがあります。連鎖の最後に新しい重複値が追加されるごとに、重複レコードは挿入された順に確実にリンクされます。インデックスに追加する目的では、すべてのヌル値は重複と見なされるので、1 つの連鎖に挿入順にリンクされます。各レコードが NIS および関連するヌル許可フィールドに異なるバイト値を持っていても、この連鎖内の最初と最後のレコードを指すキー エントリが 1 つだけ存在します。NIS キー セグメントが降順に定義されている場合、このキー エントリはインデックス内で先頭になります。それ以外は末尾になります。

繰り返し重複キーと真のヌル

繰り返し重複キーは、インデックス内に現れる各レコードの実際のキー エントリを含みます。レコード自体にオーバーヘッドはなく、レコードごとにそれ自体を指すキー エントリがあります。この種のインデックス内の重複値は、それらが指す物理レコード アドレスによってソートされます。つまり、重複値の順序は予測不能で、たくさんのクライアントによって任意のレコードが挿入および削除される高度な同時使用環境では特にそうなります。

真のヌル値は重複値であるかのように解釈され、ヌル許可フィールドのバイト値ではなく、レコード アドレスによってソートされます。したがって、繰り返し重複キーを使用する場合、真のヌル値を含むレコードはまとめてグループ化されますが、その形式は一定ではありません。NIS セグメントが降順の場合、インデックスの最初に現れ、それ以外の場合は最後に現れます。

重複のないキーと真のヌル

MicroKernel エンジンでは、重複フラグを 1 つも使用しないでインデックスを定義した場合、そのインデックスは重複のない値のみを持っている必要があります。しかし、真のヌル フィールドの値は決定不能であるため、重複と見なすことができません。このため、MicroKernel エンジンでは重複のないキーに複数の真のヌル値を入力できますが、Update オペレーションでそのキーに値が割り当てられたときに、キーの一意性が判断されるものとして扱います。ただし、これらの値をソートする目的では、MicroKernel エンジンではこれらを重複値であるかのようにまとめてグループ化します。したがって、真のヌル値を含むインデックスのセクションは繰り返し重複のインデックスと類似します。ヌルは物理レコード アドレスによってソートされ、その順序は予測できません。

変更不能キーと真のヌル

いったん変更不能キーに値を設定すると、変更することができません。しかし、真のヌル値は実際の値を持たないため、MicroKernel エンジンでは、真のヌル インデックスに定義されているフィールドのうち一部またはすべてに真のヌル値を持つレコードを挿入しておき、後で Update オペレーションを使用して、これらのフィールド値をヌルから非ヌルに変更することができます。ただし、いずれかのフィールドがいったん非ヌルになったら、変更不能性が適用され、そのフィールドを再度ヌルにしてももはや変更はできません。

Get オペレーションと真のヌル

真のヌル値が決定不能で互いに等しいとは見なされないとしても、真のヌル キー セグメントを持つレコードを検索することができます。

さまざまな Get オペレーションが、以下の手順を使用して真のヌル キーのアドレスを指定することができます。

- 1 NIS バイトに非ゼロ値を設定します。
- 2 キー バッファにキー全体を設定します。
- 3 真のヌル値が互いに等しいかのように Get オペレーションを実行します。

Get オペレーションの予想される動作を以下に示します。

- Get Equal および Get Greater Than or Equal は前方向でヌルを持つ最初のレコードを返します。
- Get Less Than or Equal は前方向から見てヌルを持つ最後のレコードを返します。
- Get Less Than はヌル値の前のレコードを返します。
- Get Greater Than はヌル値の後のレコードを返します。

これは、通常の実験値での Get オペレーションの動作と一貫性があります。

別個の (Distinct) 真のヌル

NIS バイト内の異なる値を区別することができます。前に示したように、デフォルトの動作では NIS 内の非ゼロ値はすべて同じものと見なされます。NIS にどのような値が含まれていても、ゼロでない限り、その後のヌル許可フィールドはヌルです。リレーショナル エンジンでは現在、作成するすべての真のヌル インデックス セグメントに対し、このデフォルトの動作を用いています。

しかし、テーブルに異なるタイプのヌル値を格納する場合には、NIS セグメントのキー定義に NOCASE フラグ (0x0400) を追加することができます。これ以後、これを DISTINCT フラグと呼びます。これを行うと、MicroKernel エンジンでは異なる NIS 値をそれぞれ他と区別して扱います。

別個の真のヌル セグメントは、それぞれの NIS 値によってグループに分けられます。前に述べたさまざまなタイプのインデックスを構築する際と同じ規則が適用されます。リンク重複キーは各別個の NIS 値に単一のエントリを持ち、そのタイプのヌルの繰り返しの先頭と末尾のポインターを持ちます。繰り返し重複および重複のないキーも、別個の NIS 値によってヌルレコードをグループ化します。降順キーでは、最も高い NIS 値がグループの先頭になり、ゼロ、非ヌル値へと小さい値へとソートされます。昇順キーでは、非ヌルレコードが先頭で、NIS 値 1、

2、のように続きます。Get オペレーションでは NIS 値に注意が払われます。NIS 値を 20 とするキー バッファァーを使用して GetEQ を実行する場合、別個の真のヌル インデックス内のすべての NIS 値が 1 であると、MicroKernel エンジンは一一致する値を見つけることができません。

現在は、リレーショナル エンジンもいずれの PSQL アクセス方法も、真のヌル インデックスを作成する際に DISTINCT フラグを使用していませんが、将来は使用する予定です。このため、PSQL でこれらのヌルのタイプに特定の意味を割り当てることが必要になる場合に備え、NIS 値の 2 から 16 は将来の使用のために予約されています。したがって、トランザクショナル Btrieve API を介してアクセスするレコードで個別のヌル値を使用する場合は、16 より大きい値を使用してください。

マルチ セグメントの真のヌル キー

2 つのヌルを許可する列を含む、マルチ セグメントの真のヌル インデックスについて考えてみます。キーは、実際には 4 つのセグメントのインデックスとして定義されます。最初のセグメントは NIS で、次に最初のヌル許可フィールド、2 番目の NIS、2 番目のヌル許可フィールドと続きます。以下のレコードがファイルに追加された場合にどうなるかを考えます。

```
"AAA", NULL  "BBB", NULL  "CCC", NULL  NULL, NULL
"AAA", "AAA" "BBB", "AAA" "CCC", "AAA" NULL, "AAA"
"AAA", "BBB" "BBB", "BBB" "CCC", "BBB" NULL, "BBB"
"AAA", "CCC" "BBB", "CCC" "CCC", "CCC" NULL, "CCC"
```

さらに "BBB", NULL のいくつかの組み合わせ

リレーショナル エンジンは常に、ヌル値が先頭に来る、真のヌル インデックス セグメントを作成します。各 NIS セグメントに降順フラグ (0x0040) を追加してこれを行います。降順フラグは各 NIS および 2 番目のヌル許可フィールドには使用されたが、1 番目のヌル許可フィールドには使用されていないと仮定します。そうすると、これらのレコードは次のようにソートされます。

```
1          NULL, NULL
2          NULL, "CCC"
3          NULL, "BBB"
4          NULL, "AAA"
5          "AAA", NULL
6          "AAA", "CCC"
7          "AAA", "BBB"
8          "AAA", "AAA"
9          "BBB", NULL
10         "BBB", NULL
11         "BBB", NULL
12         "BBB", "CCC"
13         "BBB", "BBB"
14         "BBB", "AAA"
15         "CCC", NULL
16         "CCC", "CCC"
17         "CCC", "BBB"
18         "CCC", "AAA"
```

ヌルは常に非ヌルより前に現れます。これは両方の NIS が降順であるためです。ただし、NIS がゼロ、つまりフィールドが非ヌルの場合、最初のフィールドは昇順で 2 番目のフィールドは降順でソートされます。

以下に、さまざまな Get オペレーションで何が返されるかを示します。

GetLT "BBB", NULL	が返すレコードは 8	"AAA", "AAA"
GetLE "BBB", NULL	が返すレコードは 11	"BBB", NULL
GetEQ "BBB", NULL	が返すレコードは 9	"BBB", NULL
GetGE "BBB", NULL	が返すレコードは 9	"BBB", NULL
GetGT "BBB", NULL	が返すレコードは 12	"BBB", "CCC "

GetLE にはファイルを逆順に調べていくという言外の意味が含まれているため、逆順で最初に「一致」したキー値のレコードを返します。GetEQ および GetGE には前方へ移動するという言外の意味があります。

インデックスからレコードを除外する

レガシー ヌルを使用すると、NIS を含むインデックスの各セグメントに「全セグメント ヌル」(0x0008) または「一部セグメント ヌル」(0x0200) のフラグも適用することができました。レコードを挿入すると、MicroKernel エンジン は NIS を使用してヌル許可フィールドがヌルかどうかを調べます。キー エントリがインデックスに含められるかどうかを決定するのにも同じ規則が使用されます。



メモ リレーショナル エンジンが作成したファイルはこれらのフラグを使用しません。

したがって、これらのファイルにどこかの時点で SQL からアクセスする可能性があり、目的が「列がヌル」であるレコードを見つけることである場合は、これらのフラグを使用しないでください。リレーショナル エンジン はヌルレコードを検索するのにインデックスを使用しますが、インデックスを介してヌルレコードにはアクセスできません。

Extended オペレーションでのヌル インジケータ セグメントの使用

Extended オペレーションを使用すると、アプリケーションは、そのテーブルのために作成されたインデックスがない場合でも、そのテーブルのフィールドにアクセスすることができます。任意のソースから得られる知識によってレコードのフィールドを実行中に定義することにより、レコード内のフィールドにフィルターを適用することができます。このように、Extended オペレーションで真のヌル フィールドを定義することが可能になり、MicroKernel エンジン はこれらのフィールドをインデックスにソートするのと同じ比較規則を適用することができます。

Extended オペレーションのフィルターは、キーを定義するのと同じように定義する必要があります。NIS のためのフィルター セグメントに続けてヌル許可フィールドのフィルター セグメントを含めます。ヌル値を検索する場合で、ヌル許可フィールドの内容が問題にならない場合であっても、ヌル許可フィールドを含める必要があります。GetNextExtended がインデックス パスに対して最適化されるには、MicroKernel エンジン は両方のフィルター セグメントを必要とします。これらが確実に含められるよう、ステータス 62 によって、NIS のためのフィルター式に続けて非 NIS のフィルター式がなかったことを示します。

NIS で使用できる比較演算子は、EQ または NE のみです。GT、GE、LT、および LE などのほかの比較演算子を使用すると、ステータス 62 が返されます。

Extended オペレーションのディスクリプターが不適切な形式あることにより発生するステータス 62 は、PSQL イベント ログにシステム エラーを追加します。これらのシステム エラーは表 2 にリストされており、ステータス 62 の理由を識別するのに役立ちます。

異なる NIS 値を明確に区別して扱いたい場合は、NIS フィルターの比較演算子に 128 を加えます。これは、大文字小文字無視で使用すると同じバイアス値です。インデックスを定義するときと同様に、非ゼロ値を明確に比較することを示す、つまりこれらがすべて同一として扱われるのではなく、互いに区別されることを示すための大文字小文字無視フラグがヌル インジケータ キー タイプに過負荷をかけてきました。

Extended オペレーションを使用して可能な限り最高のパフォーマンスを得たいなら、特定の限定されたキー値の範囲でキー パスを検索しようとするでしょう。まず最初に、GetGE を使用して範囲の先頭にカレンシーを確立します。次に GetNextExtended を実行します。また、GetLE に続けて GetPrevExtended を実行することができます。これらの Extended オペレーションは、フィルターに合致する値をもう見つけられない場合に自動的に検索を停止します。これは、Extended オペレーションの最適化と呼ばれます。フィルターで最適化を使用すれば、非常に多くのレコードをファイルから読まずに飛ばすことができるため、さらに効率的になります。最適検索を作成するためには、制限が存在する方向でインデックスを検索する必要があります。また、セグメントの結合には OR の代わりに AND を使用して、フィルターが正確にインデックスと一致するようにする必要があります。

昇順のインデックスで GetNextExtended を実行する場合、最適化フィルターが制限で検索を停止するのは、条件演算子が EQ、LT または LE の場合です。検索は、昇順のインデックスに従って特定の値より大きい値をファイルの終わり方向に向かって探す必要があります。同様に、降順のインデックスの場合、制限で検索を停止するのは条件演算子が EQ、GT または GE の場合です。検索条件に複数のフィールドがある場合、これはさらに複雑になります。簡単な考え方は、フィルターを最適化するには、最後のセグメントのみが EQ 以外の条件演算子を持

つことができるということです。これは、NIS を含みます。NIS の条件演算子が NE の場合、フィルターは、前のフィルター セグメントまでしか最適化することができません。

インデックスと正確に合致するということは、各フィルター式がインデックス内のセグメント順に従っており、同じオフセット、長さ、キー タイプ、大文字小文字区別（または DISTINCT フラグ）、および ACS 仕様を持つということです。これらがインデックスと合致していないと、Extended オペレーションは最適化できません。

真のヌルと SQL エンジン

真のヌルは、ヌル インジケータのキー タイプを使用し、前述の規則に従うことによって、リレーショナル エンジンに実装されます。MicroKernel エンジン アプリケーションも、このキー タイプを使用してヌル可能フィールドがヌルかどうかをその内容にかかわらず識別することができます。これにより、整数とほかの数値データ型のヌルを識別し、これらのヌル可能フィールドを完全に管理することができます。

真のヌルと Extended オペレーション

Extended オペレーションで発生するステータス 62 はディスクリプターが不正であることを示します。ディスクリプターの何が間違っているのかを判断するのがむずかしい場合があります。データベース エンジンでは PSQL イベント ログに、問題を正確に判断するのに使用できる行を追加しました。イベント ログのエントリは次のようなものです。

```
12-12-2008 11:12:45 W3MKDE      0000053C W3dbsmgr.exe  MY_COMPUTER      E
System Error:301.36.0  File:D:\WORK\TEST.MKD
```

システム エラー 301 から 318 までの番号は、以下の問題を示しています。

表 2 システム エラー コード

システム エラー	説明
301	ディスクリプター長が不正です。
302	ディスクリプター ID は、"EG" または "UC" でなければなりません。
303	フィールド タイプが無効です。
304	演算子の NOCASE フラグは、文字列およびヌル インジケータ タイプのみで使用できます。
305	演算子の ACS フラグ (0x08 および 0x20) は、文字列タイプのみで使用できます。
306	バイアスされていない演算子はゼロと同等です。
307	バイアスされていない演算子が 6 より大きいです。
308	無効な式のコネクタが見つかりました。0、1、2 のみで使用できます。
309	ACS が定義されていません。
310	最後の式には終端文字が必要です。
311	最後の式より前に終端文字が見つかりました。フィルター セグメント カウントが不正です。
312	抽出するレコード数がゼロです。
313	抽出するフィールド長がゼロです。
314	ヌル インジケータ セグメントの次には別のフィールドが続く必要があります。
315	ヌル インジケータ セグメントは AND で次のセグメントと結合する必要があります。
316	ヌル インジケータ セグメントは EQ または NE のみで使用できます。
317	ヌル インジケータ セグメントの次に別の NIS は続けられません。
318	ヌル インジケータ セグメントの次のフィールドは 255 バイト以下である必要があります。

オルタネート コレーティング シーケンス

オルタネート コレーティング シーケンス (ACS) を使用して、文字型のキー (STRING、LSTRING および ZSTRING) を標準 ASCII コレーティング シーケンスとは異なる順序でソートすることができます。1 つまたは複数の ACS を使用して、次のようにキーをソートすることができます。

- 独自のユーザー定義ソート順序による方法。この方法は、英数字 (A ~ Z、a ~ z、0 ~ 9) を非英数字 (# など) と混用するソート順序を必要とするような場合に使用します。
- スペイン語の *ll* のようなマルチバイト コレーティング要素、フランス語の *ô* のような区分発音符、ドイツ語の *ss* へ拡張する *ß* のような文字の拡張および短縮など、言語固有の照合順序に対応する国際的なソート規則 (ISR) による方法。

ファイルには、キーごとに異なる ACS を持つことができますが、1 つのキーには 1 つの ACS のみです。したがって、キーがセグメント化されている場合、各セグメントはそのキーに指定された ACS を使用するか、または ACS をまったく使用しないかのいずれかになります。あるファイルに、一部のセグメントにだけ ACS が指定されているキーがある場合、MicroKernel エンジンでは指定されたセグメントのみ、ACS を使用してソートします。

ユーザー定義 ACS

ASCII 標準とは別の方法で文字列値をソートする ACS を作成するには、次の表に示す形式を使用します。

表 3 ユーザー定義オルタネート コレーティング シーケンスの形式

位置 (オフセット)	長さ	説明
0	1	識別バイト。0xAC を指定します。
1	8	MicroKernel エンジンに ACS を識別させる、8 バイトの一意の名前。
9	256	256 バイトのマッピング。マッピング内の 1 バイトの位置はそれぞれ、マッピング内でのその位置のオフセットと同じ値を持つコードポイントに対応します。その位置にあるバイトの値は、コードポイントに割り当てられるコレーティング ウェイトです。たとえば、コードポイント 0x61 (<i>a</i>) をコードポイント 0x41 (<i>A</i>) と同じウェイトでソートさせるには、同じ値をオフセット 0x61 と 0x41 に設定します。

ACS は 16 進エディターで作成されるか、MicroKernel エンジン アプリケーションを作成するときに定義されるので、ユーザー定義 ACS はアプリケーション開発者には有用ですが、一般にエンド ユーザーによっては作成されません。

以下に、UPPER というコレーティング シーケンスを表す 9 バイト ヘッダーと 256 バイト本体を示します。ヘッダーは以下のとおりです。

```
AC 55 50 50 45 52 20 20 20
```

256 バイト本体は以下のようなものですが、左端の列にオフセットが付けてあります。

```
00: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
10: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
30: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40: 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50: 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60: 60 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
70: 50 51 52 53 54 55 56 57 58 59 5A 7B 7C 7D 7E 7F
80: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
90: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
B0: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
D0: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0: E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
F0: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

この ACS を構成するヘッダーと本体は、ファイル UPPER.ALT として PSQL に付属しています。UPPER.ALT は、大文字小文字に関係なくキーをソートすることができます（大文字と小文字を区別しないようにキーを定義することができますが、UPPER は独自の ACS を書くときに良い見本になります）。

例に示すオフセット 0x61 ~ 0x7A は、標準の ASCII コーディング シーケンスから変更されています。標準 ASCII コーディング シーケンスでは、オフセット 0x61 には値 0x61（小文字の *a* を表します）が含まれています。UPPER ACS を使用してキーをソートする場合、MicroKernel エンジンでは小文字の *a* (0x61) を、オフセット 0x61 のコーディング ウェイトである 0x41 を使ってソートします。このように、小文字 *a* は大文字 *A* (0x41) であるかのようにソートされます。したがって、ソートの目的のために、UPPER はキーのソート時に、すべての小文字をそれらに相当する大文字に変換します。

次に示す 256 バイトの本体は、ASCII スペース (0x20) の前の ASCII 文字が他のすべての ASCII 文字の後にソートされる点を除き、UPPER.ALT の本体と同じ機能を果たします。

```

00: E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
10: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
20: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
30: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
40: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
50: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
60: 40 41 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
70: 30 31 32 33 34 35 36 37 38 39 3A 5B 5C 5D 5E 5F
80: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
90: 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
A0: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
B0: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
C0: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
D0: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
E0: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
F0: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF

```

この本体では、別のコーディング ウェイトが割り当てられているので、文字のウェイトがもう ASCII 値と同じではありません。たとえば、ASCII スペース文字を表すオフセット 0x20 のコーディング ウェイトは 0x00 です。ASCII の大文字 *A* を表すオフセット 0x41 のコーディング ウェイトは 0x21 です。

大文字小文字に関係なくキーをソートするために、最後の例のオフセット 0x61 ~ 0x7A は変更されています。UPPER.ALT の本体と同様に、オフセット 0x61 はオフセット 0x41 と同じコーディング ウェイト、つまり 0x21 を持っています。同じコーディング ウェイトを持つことにより、オフセット 0x41 (*A*) はオフセット 0x61 (*a*) と同じようにソートされます。

インターナショナル ソート規則

ISO で定義された言語固有のコーディング シーケンスを使用して文字列をソートする ACS を指定するには、以下のように ISR テーブル名を指定する必要があります。

表 4 ISR テーブル名

ロケール / 言語	コード ページ	ISR テーブル名
アメリカ / 英語	437 MS-DOS Latin-US 850 MS-DOS Latin-1	PVSW_ENUS00437_0 PVSW_ENUS00850_0
フランス / フランス語	437 MS-DOS Latin-US 850 MS-DOS-Latin-1	PVSW_FRFR00437_0 PVSW_FRFR00850_0
ドイツ / ドイツ語	437 MS-DOS Latin-US 850 MS-DOS Latin-1	PVSW_DEDE00437_0 PVSW_DEDE00850_0

表 4 ISR テーブル名

ロケール / 言語	コード ページ	ISR テーブル名
スペイン / スペイン語	437 MS-DOS Latin-US 850 MS-DOS Latin-1	PVSW_ESES00437_0 PVSW_ESES00850_0
日本 / 日本語	932 シフト JIS	PVSW_JPJP00932_1

ISR テーブルは ISO の標準ロケール テーブルに基づいており、PSQL によって提供されます。ISR テーブルは PSQL のデータベース エンジンと一緒にインストールされた COLLATE.CFG ファイルに格納されています。複数のデータ ファイルが 1 つの ISR を共有できます。

サンプルの照合順序については、付録 A「[国際的なソート規則を使用した照合順序のサンプル](#)」を参照してください。

キー仕様

CREATE (14) または CREATE INDEX (31) を使用してインデックスを作成する場合、キー仕様構造体（インデックス セグメント ディスクリプター）が作成されます。各キー仕様は長さ 16 バイトで、次の情報を含みます。

表 5 キー仕様構造体

フィールド	データ型	長さ	NIS セグメント	説明
キー ポジション	Short Int	2	レコードの固定長部分でのオフセット	レコード内のキーの相対位置
キー長	Short Int	2	1	キーの長さ。常に 2 バイト
キー属性	Short Int	2	xxxxxxxxlxxxlxxxx FEDCBA9876543210	キー属性。属性の詳細については、次のセクションを参照してください。
予約済み	Byte	4	適用外	未使用
拡張データ型	Byte	1	255 (0xFF)	拡張データ型のうちの 1 つを指定します。新しいデータ型の NULL_INDICATOR が定義されました。
ヌル値（非インデックス値）	Byte	1	適用外	キーの除外値を指定します。
予約済み	Byte	2	適用外	未使用
手動割り当てキー番号	Byte	1		キー番号
ACS（オルタネート コレーティング シーケンス）番号	Byte	1	適用外	オルタネート コレーティング シーケンス (ACS) 番号

表 6 キー属性

属性	2 進数	16 進数	説明
重複	0000 0000 0000 0001	0x0001	
変更可能	0000 0000 0000 0010	0x0002	
バイナリ	0000 0000 0000 0100	0x0004	
ヌル キー（全セグメント）	0000 0000 0000 1000	0x0008	

表 6 キー属性

属性	2 進数	16 進数	説明
セグメント	0000 0000 0001 0000	0x0010	
ACS	0000 0000 0010 0000	0x0020	
ソート順序	0000 0000 0100 0000	0x0040	
繰り返し重複	0000 0000 1000 0000	0x0080	
拡張データ型	0000 0001 0000 0000	0x0100	
ヌルキー（一部セグメント）	0000 0010 0000 0000	0x0200	
大文字と小文字の区別 (Distinct)	0000 0100 0000 0000	0x0400	
既存の ACS	0000 1000 0000 0000	0x0008	内部使用のみ
予約済み	0001 0000 0000 0000	0x1000	
ページ圧縮	0010 0000 0000 0000	0x2000	「 ページ レベル圧縮を用いたファイルの作成 」を参照
ペンディング キー	1000 0000 0000 0000	0x8000	内部使用のみ

- セグメント キー特有のキー フラグ SEGMENTED (0x0010)、EXTENDED DATA TYPE (0x0100) は、NIS と共に ON に設定する必要があります。

フラグ none は OFF にする必要があります。

フラグ BINARY (0x0004)、ACS (0x0020) は無視されます。

制限と影響

真のヌルのサポートには、いくつかの制限があります。

- 参照整合性：現在 MicroKernel エンジンは、DELETE の CASCADE および RESTRICT アクション、UPDATE の RESTRICT アクションのみをサポートしています。SQL-92 は、delete および update の両方に CASCADE、RESTRICT、SET DEFAULT および SET NULL を定義しています。
- セグメント数の制限：ヌル許容列ごとに 2 つのセグメントを占めるため、キーのインデックス付けに使用するインデックス セグメントの数が増加しました。一方、データ ファイルごとのインデックス セグメントの最大数は同じです。

表 7 ページごとの最大インデックス セグメント数

ページ サイズ (バイト数)	キー セグメントの最大数 (ファイル バージョン別)		
	8.x 以前	9.0	9.5
512	8	8	切り上げ ²
1024	23	23	97
1536	24	24	切り上げ ²
2048	54	54	97
2560	54	54	切り上げ ²
3072	54	54	切り上げ ²

表 7 ページごとの最大インデックス セグメント数

ページ サイズ (バイト数)	キー セグメントの最大数 (ファイル バージョン別)		
	8.x 以前	9.0	9.5
3584	54	54	切り上げ ²
4096	119	119	119 または 204 ³
8192	N/A ¹	119	119 または 420 ³
16384	N/A ¹	N/A ¹	119 または 420 ³
¹ N/A は「適用外」を意味します。 ² 「切り上げ」は、ページ サイズを、ファイル バージョンでサポートされる次のサイズへ切り上げることを意味します。たとえば、512 は 1024 に切り上げられ、2560 は 4096 に切り上げるということです。 ³ リレーショナル エンジンで利用できるインデックス セグメントの最大数は 119 です。MicroKernel エンジンの場合、最大数は、ページ サイズ 4096 では 204、ページ サイズ 8192 および 16384 では 420 です。			

データベース URI

Btrieve ログイン API や、Create または Open オペレーションによる暗黙のログイン機能を使用する上での主要な概念は、データベース URI (Uniform Resource Indicator) です。これは、サーバー上のデータベース リソースのアドレスを記述する構文を提供します。

このセクションでは、Btrieve API で使用する URI の構文と意味を説明します。

構文

URI は次の構文を使用します。

```
access_method://user@host/dbname?parameters
```

表 8 データベース URI の要素

要素	定義
<i>access_method</i>	データベースのアクセスに使用する方法。この要素は必須です。現在、btrv のみがサポートされています。
<i>user@</i>	ユーザー名 (省略可能)。必要場合は、 <i>parameters</i> にユーザーのパスワードを指定します。"@ " 文字は、 <i>host</i> が指定されない場合でもユーザー名を区切るために使用する必要があります。
<i>host</i>	データベースが保存されているサーバー。 <i>host</i> が指定されていない場合は、ローカル マシンと見なされます。 <i>host</i> には、マシン名、IP アドレス、あるいは "localhost" キーワードを指定できます。 メモ : Linux または OS X システムのデータベースにアクセスする URI の場合、 <i>host</i> は必須要素です。
<i>dbname</i>	データベース名 (省略可能)。PSQL データベース エンジンの DBNAMES.CFG ファイル内のエントリに対応します。データベース名が指定されていない場合は、デフォルトのデータベース "DefaultDB" と見なされます。
<i>parameters</i>	追加オプションパラメーター。& (アンパサンド) 文字で区切ります。 <ul style="list-style-type: none"> ◆ table= テーブル - 特定の SQL テーブル名を指定します。テーブル名はデータベースの DDF に存在している必要があります。 ◆ dbfile= ファイル - ファイルの名前。ファイルの場所は、現在のデータベースに対する、DBNAMES.CFG 内のデータ ファイル ロケーションのエントリに関連しています。相対パスが指定されるので、ドライブ名の使用、絶対パスまたは UNC パスは不可です。データベース エンジンは完全なファイル名を解決します。いかなる方法であっても、PSQL クライアントによって「ファイル」が処理されることはありません。空白の埋め込みは可能です。それらの空白はデータベース エンジンによってエスケープされます。 ◆ file= ファイル - 特定のデータ ファイル名を指定します。PSQL クライアントはデータベース エンジンへ要求を送る前に、ファイルを標準の状態に変更し、入力名をその変更後の完全修飾 UNC 名で置き換えます。ドライブ名が使用されることがあり、その場合はクライアント側のドライブ名として解釈されます。スペースを含む UNC パスを使用することも可能です。 ◆ pwd= パスワード - クリア テキスト パスワード。PSQL クライアントは、転送する前に、クリア テキスト パスワードを暗号化パスワードに変更します。 ◆ prompt=[yes no] - データベース エンジンからステータス 170 (ユーザー名が不正であるか見つかからないため、ログインに失敗しました) または 171 (パスワードが不正なため、ログインに失敗しました) が返されたとき、ログイン ダイアログ ボックスのポップアップをどのように処理するかを PSQL クライアントに知らせます。prompt=yes と指定した場合、リクエストは [クライアント資格情報の入力要求] がオフに設定されている場合でも、常にログイン ダイアログ ボックスを表示します。prompt=no と指定した場合、リクエストは、アプリケーションはステータス 170/171 を直接受け取って、リクエストにダイアログ ボックスを表示させたくないものと見なします。これは、170 または 171 ステータス コードに対して、認証の入力要求をアプリケーションで処理したい場合に有効です。"yes" または "no" 以外の値は無視されます。リクエストは [クライアント資格情報の入力要求] 設定に基づいてログイン ダイアログ ボックスを表示します。このオプションは、クライアントの役割を果たしている Linux および OS X では無視されます。

パラメーターの優先順位

パラメーターの "file"、"table"、および "dbfile" のうち、2 つ以上のパラメーターが URI に指定された場合、データベース エンジンはそれらのパラメーターに対し優先順位を設定します。つまり、データベース エンジンは URI の解析後、優先順位が最も高いパラメーターを残します。同じ優先度を持つパラメーターが 2 つ以上指定された場合は、解析後には URI で最後に指定されているパラメーターが残ります。

優先順位は "file"、"table"、"dbfile" の順で設定されています。

優先順位の例

初期 URI 文字列	解析後 URI 文字列
btrv://?file=MyFile.btr&table=MyTable&dbfile=DataFile.btr	btrv://?file=MyFile.btr
btrv://?table=MyTable&dbfile=DataFile.btr	btrv://?table=MyTable
btrv://?dbfile=DataFile.btr&file=MyFile.btr	btrv://?file=MyFile.btr
btrv://?dbfile=DataFile.btr	btrv://?dbfile=DataFile.btr
btrv://?file=FileOne&file=FileTwo	btrv://?file=FileTwo
btrv://?table=TableOne&table=TableTwo&file=MyFile.btr	btrv://?file=MyFile.btr

特殊文字

ほかの URI と同様に、ある特定の英数字以外の文字は URI 構文で特殊な意味を持ちます。そのような文字の 1 つを URI のいずれかの要素で使用する場合は、その文字が実際のテキストではなく特殊文字であると識別されるよう、**エスケープ シーケンス**を使用する必要があります。エスケープ シーケンスは、特殊文字に相当する標準テキストを表す、別の特殊文字または文字の組み合わせです。

下記の表は、MicroKernel エンジン URI の構文でサポートされている特殊文字と、それに関連付けられたエスケープ シーケンス（パーセント記号と、指定された文字の 16 進値で表されます）を示しています。

表 9 データベース URI における特殊文字

文字	説明	16 進値
/	ディレクトリとサブディレクトリを区切ります。	%2F
?	ベース URI と関連パラメーターを分離します。	%3F
%	特殊文字を指定します。	%25
#	ブックマークまたはアンカーを示します。	%23
&	URI 内のパラメーターを区切ります。	%26
" "	二重引用符で囲まれている内容全体を示します。	%22
=	パラメーターとその値を区切ります。	%3D
空白	特別な意味はありませんが、予約されています。	%20
:	ホストとポートを分離します（予約されていますが、現在はサポートされていません）。コロンは IPv6 アドレスでも一部使用されます。「 IPv6 」を参照してください。	%3A

空白文字は URI 仕様で予約されていますが、これは区切り文字として使用されないため、引用符もエスケープ シーケンスもなしで使用できます。それ以外の上記の表内の記号は区切り文字として使用されるため、エスケープする必要があります。

例

このセクションでは、フィールド値の中で使用されている特殊文字を識別するためにエスケープシーケンスを使用している URI の例を示します。

表 10 エスケープシーケンスを含んでいる URI の例

URI	説明
btrv://Bob@myhost/demodata?pwd= This%20Is%20Bob	ユーザー名が "Bob" でパスワードが "This Is Bob"
btrv://Bob@myhost/demodata?pwd= This Is Bob	ユーザー名が "Bob" でパスワードが "This Is Bob"
btrv://myhost/mydb?file=c:/data%20files/pvsw/mydb/c.mkd	%20 は空白文字を表します。開くファイルは "C:¥data files¥pvsw¥mydb¥c.mkd" です。
btrv://Bob@myhost/demodata?pwd= mypass%20Is%20%26%3f	ユーザー名が "Bob" でパスワードが "mypass Is &?"

備考

空のユーザー名または空のパスワードは、ユーザー名やパスワードがないこととは異なるので注意してください。たとえば、btrv://@host/ には空のユーザー名が入っていますが、btrv://host/ にはユーザー名がありません。btrv://sam@host/?pwd= には、"sam" というユーザー名が入っており、パスワードは空です。

URI によっては *user:password* 構文を使用することもできます。ただし、ここで指定されるパスワードはその後クリアテキストとして転送されます。パスワードがクリアテキストとして転送されないようにするため、*user:password* 構文を使用してパスワードが提供された場合、PSQL データベース URI はそのパスワードを無視します。pwd= パラメーターを使用してパスワードを提供してください。このパスワードは PSQL クライアントによって転送される前に暗号化パスワードへ変更されます。

いくつかの URI では *user@host:port* 構文を使用するサーバーベースの命名機関を可能にすることもできます。PSQL データベース URI は *port* 要素の指定をサポートします。

例

URL (Uniform Resource Locator) は単に、インターネット上のファイルまたはリソースのアドレスです。データベース URI は同じ概念を利用してサーバー上のデータベースのアドレスを指定します。このセクションでは、PSQL データベースにおける、特に MicroKernel エンジン アクセスを使用する場合の URI の構文と意味の例を挙げます。

表 11 MicroKernel エンジン URI の例

例	説明
btrv://myhost/demodata	サーバー "myhost" 上のデータベース "demodata"。サーバーのオペレーティングシステムは、PSQL でサポートされる OS のいずれかになります。
btrv:///demodata	ローカルマシン上のデータベース "demodata"。ローカルマシンは Windows オペレーティングシステムを実行しています。Linux および OS X システムでは <i>host</i> 要素が必須です (上の例を参照)。
btrv://Bob@myhost/demodata	パスワードなしのユーザー名 "Bob" で、サーバー "myhost" 上のデータベース "demodata" にアクセスします。
btrv://Bob@myhost/mydb?pwd=a4	ユーザー名 "Bob"、パスワード "a4" で、サーバー "myhost" 上のデータベース "mydb" にアクセスします。
btrv://myhost/demodata?table=class	不特定のユーザーが、サーバー "myhost" 上のデータベース "demodata" 内のデータベーステーブル "class" にアクセスします。
btrv://myhost/?table=class	不特定のユーザーが、サーバー "myhost" 上のデフォルト データベース ("DefaultDB") 内のデータベーステーブル "class" にアクセスします。

表 11 MicroKernel エンジン URI の例

例	説明
btrv://myhost/mydb?file=f/mydb/a.mkd	不特定のユーザーが、サーバー "myhost" 上のデータベース "mydb" のセキュリティ資格情報を使用して、クライアントで見られるようなデータ ファイル "f/mydb/a.mkd" にアクセスします。 クライアントはドライブ "f" を変更するので注意してください。これは、クライアントで "f" をサーバー "myhost" へ割り当てる必要があることを意味します。
btrv://mydb?file=c/mydb/a.mkd	不特定のユーザーが、ローカル マシン上のデータベース "mydb" 下のデータ ファイル "c:/mydb/a.mkd" にアクセスします。 ドライブ "c:" はローカル マシン上のローカルドライブです。ローカル マシンは Windows オペレーティング システムを実行しています。
btrv://myhost/demodata?dbfile=class.mkd	不特定のユーザーが、サーバー "myhost" 上のデータベース "demodata" に定義されたデータ ディレクトリのうちの 1 つにあるデータ ファイル "class.mkd" にアクセスします。ファイル名が file= ではなく dbfile= で指定されているため、クライアント リクエスターはファイル名 class.mkd を標準の状態に変更しません。サーバー エンジンのみが class.mkd を標準の絶対パスに変更します。

IPv6

URI および UNC 構文では、コロンなど一部の特殊文字を使用できません。未加工の IPv6 アドレスではコロンを使用するので、UNC パスや URI 接続の処理には異なる方法を使用できます。PSQL は IPv6-literal.net 名および、角かっこ ([]) で囲まれた IPv6 アドレスをサポートします。

『Getting Started With PSQL』の「[IPv6](#)」を参照してください。

ダブルバイト文字のサポート

PSQL は、ファイルパス内にシフト JIS（日本工業規格）でコード化されたダブルバイト文字を受け入れます（シフト JIS は、日本語のコンピューターに一般に使用されるコード化の手法です）。また、レコードにシフト JIS ダブルバイト文字を格納し、「[インターナショナル ソート規則](#)」で説明した日本語 ISR テーブルでそれらの文字をソートします。他のマルチバイト文字はレコードに格納できますが、ISR テーブルは文化的に正しい規則に従ってこれらのレコードをソートする目的には現在使用できません。ダブルバイト文字を使用しても、PSQL アプリケーションのオペレーションに影響を与えません。

レコード長

すべてのレコードには、レコード長、あるレコードのキーを含むすべてのデータを取り込めるだけの十分な大きさが必要な固定長部分、および、データ ページにレコードを格納するのに必要なオーバーヘッドが含まれています。

物理レコード長を算出するために論理レコード長に追加しなければならないオーバーヘッドのバイト数については、「[レコードの圧縮を使用しない場合のレコード オーバーヘッドのバイト数](#)」および「[レコードの圧縮を使用した場合のレコード オーバーヘッドのバイト数](#)」を参照してください。

次の表は、固定長レコードの最大レコード サイズの一覧です。

表 12 固定長レコードの最大レコード サイズ (バイト単位)

ファイル バージョン	システムデータ不使用 ¹	システムデータ使用 ²
7.x	4088 (4096 - 8)	4080 (4088 - 8)
8.x	4086 (4096 - 10)	4078 (4086 - 8)
9.0 ~ 9.4	8182 (8192 - 10)	8174 (8182 - 8)
9.5 以上	16372 (16384 - 12)	16364 (16372 - 8)
¹ ページ オーバーヘッドおよびレコード オーバーヘッドは、最大ページ サイズから減算して最大レコード サイズを決定します。レコード オーバーヘッドは、各ファイル形式で 2 バイトです。		
² システム データは 8 バイトの追加オーバーヘッドが必要です。		

ファイルがシステム データを使用し、レコード長が上記の表に示した制限を超える場合、データベース エンジンではそのファイルに対し、データ圧縮を行うことに注意してください。

オプションとして、ファイル内のレコードには可変長部分を含めることができます。可変長レコードには、すべてのレコード内で同じサイズである固定長部分と、各レコードでサイズの異なる可変長部分があります。可変長レコードを使用するファイルを作成する場合、固定長の長さは各レコードの最大長です。最大レコード長は定義しません。

理論的には、可変長レコードの最大長はMicroKernel エンジンのファイルサイズの限界にのみ制限されます。PSQL バージョン 9.5 の場合は 256 GB です (9.x から 9.5 までのバージョンでは 128 GB、それ以前のバージョンでは 64 GB)。実際に、最大長は選択したオペレーティング システムやレコード アクセス方法のような要因により制限されます。レコード全体を検索、更新または挿入すると、データ バッファ長パラメーターは 16 ビット符号なし整数であるため、レコード長を 65535 までに制限します。

実行するオペレーションによって異なりますが、データ バッファは各種情報の転送に使用する MicroKernel エンジン関数パラメーターです。データ バッファには、レコード全体、レコードの一部、ファイル仕様などが含まれます。データ バッファの詳細については、第 5 章「[データベースの設計](#)」の表 15 を参照してください。



メモ データと内部ヘッダー情報の合計バイトは、64 KB (0x10000) バイトを超えることはできません。MicroKernel エンジンは、内部の目的で 1024 (0x400) バイトを予約します。つまり、64512 (0xFC00) バイトのデータを持つことができます。

ファイルが非常に大きいレコードを使用する場合は、ファイル内の可変長部割り当てテーブル (VAT) を考慮してください。リンク済みリストとして実装されている VAT は、レコードの可変長部分へのポインターの配列です。VAT は、非常に大きいレコードの各部分へのランダム アクセスを加速します。非常に大きいレコードの例として、バイナリ ラージオブジェクト (BLOB) やグラフィックスがあります。

非常に大きい可変長レコードを含むファイルについて、MicroKernel エンジンは多数の可変ページにまたがってレコードを分割し、**可変長部**と呼ぶリンク済みリストでこれらのページを接続します。アプリケーションがチャン

ク オペレーションを使用してレコードの一部にアクセスし、レコードの一部がレコード自体の先頭を越えたオフセットから始まる場合、MicroKernel エンジンはそのオフセットをシークするための可変長部リンク済みリストの読み取りにかなりの時間を費やすことがあります。そのようなシーク時間を制限するために、ファイルが VAT を使用するよう指定できます。MicroKernel エンジンは可変ページに VAT を格納します。VAT を含むファイルでは、可変長部分を持つ各レコードにそれ自体の VAT があります。

MicroKernel エンジンが VAT を使用するのには、ランダム アクセスを加速するためだけでなく、データ圧縮時に使用される圧縮バッファのサイズを制限するためでもあります。ファイルでデータ圧縮を使用する場合、そのファイルで VAT を使用する必要があります。

データの整合性

以下の機能は、マルチユーザー環境でファイルの整合性を保証しながら、並行アクセスをサポートします。

- 「レコード ロック」
- 「トランザクション」
- 「トランザクション一貫性保守」
- 「システム データ」
- 「シャドウ ページング」
- 「ファイルのバックアップ」

レコード ロック

アプリケーションは、明示的に、一度に1レコード（単一レコード ロック）または一度に複数のレコード（複数レコード ロック）をロックできます。アプリケーションは、レコード ロックを指定する場合、ウェイトまたはノーウェイト条件を適用することもできます。アプリケーションが現在使用できないレコードに対するノーウェイト ロックを要求する場合、つまり、レコードが既に別のアプリケーションでロックされているか、ファイル全体が排他トランザクションでロックされている場合、MicroKernel エンジンではロックを許可しません。

アプリケーションが使用できないレコードに対するウェイト ロックを要求すると、MicroKernel エンジンは**デッドロック状態**の有無を確認します。デッドロック検出ステータス コードを返す前に待機するように MicroKernel エンジン を設定することができます。これを行うと、MicroKernel エンジン を内部的に待機させてアプリケーションにはオペレーションを再試行させないので、マルチユーザー環境におけるパフォーマンスを向上させます。

トランザクション

ファイルに対して行う変更が多く、また、これらの変更をすべて行うか、またはまったく行わないかを確実にしなければならない場合は、**トランザクション**でこれらの変更を行うためのオペレーションを取り込みます。明示的なトランザクションを定義すれば、MicroKernel エンジンに複数のオペレーションをアトミックな単位として処理させることができます。ほかのユーザーは、トランザクションが終了するまでファイルに対して行われた変更がわかりません。MicroKernel エンジンでは、排他トランザクションと並行トランザクションの2種類のトランザクションをサポートします。

排他トランザクション

排他トランザクションでは、データ ファイルでレコードを挿入、更新または削除するときに MicroKernel エンジンがそのデータ ファイル全体をロックします。ほかのアプリケーションまたは同じアプリケーションの別のインスタンスはファイルを開いてそのレコードを読み取ることができますが、ファイルを変更することはできません。ファイルは、アプリケーションがトランザクションを終了または中止するまで、ロック状態のままになります。

並行トランザクション

並行トランザクションでは、MicroKernel エンジンでは、実行するオペレーションによってファイル内のレコードまたはページをロックします。MicroKernel エンジンでは、複数のアプリケーション（または同じアプリケーションの複数のインスタンス）が同じファイルの異なる部分に並行トランザクション内で変更を行えるようにしますが、それはこれらの変更が既にロックされているほかのファイル部分に影響を与えない場合にに限られます。レコードまたはページは、アプリケーションがトランザクションを終了または中断するまで、ロック状態のままになります。並行トランザクションは、6.0 以降のファイルのみに使用できます。

排他と並行

たとえ要求されたレコードを並行トランザクションが既にロックしていても、クライアントはこれまでどおりレコードを**読み取る**ことができます。ただし、これらのクライアントは排他トランザクション内から処理を行えません。また、要求されたレコードを含むファイルが排他トランザクションで現在ロックされているか、要求され

たレコードを並行トランザクションが既にロックしている場合、クライアントは読み取り操作にロック バイアスを適用できません。

クライアントが排他ロックでレコードを読み取ると、MicroKernel エンジン是个々のレコードだけをロックし、レコードが存在するページの残りの部分はロックされない状態のままになります。



メモ トランザクション内からファイルを開くだけでは、レコード、ページまたはファイルはロックされません。また、MicroKernel エンジンは、読み取り専用のフラグを立てたファイルや読み取り専用モードで開いたファイルはロックしません。

排他トランザクションを使用する場合、Begin Transaction (19 または 1019) オペレーションにノー ウェイト バイアスが付加されない限り、MicroKernel エンジンはロックされたファイルではかのクライアントを暗黙に待機させます。アプリケーションは、この暗黙の待機状態でハングしたように見えます。これらの排他トランザクションの寿命が短いと、待機時間に気が付かない場合があります。ただし、暗黙の待機を必要とする多くのクライアントの影響によって、大量の CPU 時間が使用されます。さらに、同じファイル内の複数のポジション ブロックはロックを共有します。

暗黙の待機を必要とする排他トランザクションも、ネットワーク帯域幅を無駄に使用しています。MicroKernel エンジンは、リクエスターに戻る前に約 1 秒間待機します。リクエスターは待機状態を認識し、MicroKernel エンジンに操作を返します。したがって、排他トランザクションは余計なネットワークトラフィックを発生させる可能性もあります。

余計な CPU サイクルとネットワークトラフィックの量は、ロック済みファイルで待機しているクライアント数と排他トランザクションに必要な時間と相まって、幾何級数的に増加します。

トランザクションー貫性保守

すべてのオペレーションを 1 つのトランザクション ログに記録することによって、「**トランザクションー貫性保守**」とアトミシティを保証するように MicroKernel エンジンを設定できます。トランザクションー貫性保守は、クライアントが End Transaction オペレーションを発行したときと、MicroKernel エンジンが正常終了したステータスコードをクライアントに返す前に、MicroKernel エンジンがログへの書き込みを終了することを保証する機能です。アトミシティは、特定のステートメントが終わりまで実行しない場合にそのステートメントがデータベース内に部分的または不明確な影響を残さないように保証し、それによってデータベースを安定した状態に保つことでデータベースの整合性を保証します。

トランザクションー貫性保守のオーバーヘッドなしでアトミシティが必要な場合は、PSQL V8 以降のリリースのトランザクション ログ機能を使用することができます。トランザクション ログの詳細については、『Advanced Operations Guide』を参照してください。

デフォルトのインストールでは、トランザクション ログは C:\ProgramData\Actian\PSQL\logs にあります。ログは、PSQL エンジンと同じマシン上に存在しなければなりません。この場所は、PCC でトランザクション ログディレクトリ設定オプションを使用して変更できます。MicroKernel エンジンを右クリックして [プロパティ] を選択し、[ディレクトリ] をクリックすると表示されます。

MicroKernel エンジンは、ログ セグメントと呼ぶ 1 つまたは複数の物理ファイルでトランザクション ログを保守します。現在のログ セグメントがユーザー定義のサイズ上限に達し、変更が待機状態であるファイルがなく、MicroKernel エンジンがシステム トランザクションを終了すると、MicroKernel エンジンは新しいログ セグメントを開始します。

すべてのトランザクション ログ セグメントには必ず .log というファイル拡張子が付きます。MicroKernel エンジンはログ セグメントのファイル名として、00000001.log、00000002.log、.... のように、8 桁の 16 進数を使った連続する番号を使用します。

特定ファイルのパフォーマンスを向上させるために、アクセラレイティド モードでファイルを開くことができます (バージョン 6.x MicroKernel エンジンはアクセラレイティド オープン要求を受け入れましたが、それらの要求をノーマル オープン要求と解釈していました)。アクセラレイティド モードでファイルを開くと、MicroKernel エンジンは、そのファイルに対するトランザクション ログを実行しません。



メモ システム障害が発生すると、起動時に削除されないログ セグメントが生成される場合があります。これらのセグメントは、データ ファイルに完全には書き込まれなかった変更を含みます。これらのログ セグメントを削除しないでください。どのファイルがこれらのログ セグメントに書き込まれているかはわかりません。これらのデータ ファイルは、次回開かれたときに自動的にロール フォワードするので、何もする必要はありません。

システム データ

PSQL は、7.x トランザクション ログ ファイル形式を使用します。MicroKernel エンジンがファイルでトランザクションのログを記録するには、MicroKernel エンジンがログ内のレコードを追跡する際に使用できる重複のない（重複不可能）キーである **ログ キー** がファイルに含まれている必要があります。1 つ以上の重複のない（重複不可能）キーを持つファイルの場合、MicroKernel エンジンがファイルで既に定義されている重複のないキーのうちの 1 つを使用します。

重複のないキーを持たないファイルの場合、MicroKernel エンジンがファイル作成時に **システム データ** を含めることができます。MicroKernel エンジンがファイルが 7.x 以降のファイル形式を使用している場合だけファイルにシステム データを含めます。その時点でファイルを作成する場合は、ファイル作成時にファイルにシステム データを含めるように MicroKernel エンジンが設定されます。システム データは、キー番号 125 で 8 バイトのバイナリ値として定義されます。たとえばファイルに重複のないユーザー定義キーがあっても、ユーザーがインデックスを削除することから保護するためにシステム定義ログ キーとも呼ぶシステム データを使用しなければならない場合があります。

ファイルがシステム データを使用し、レコード長が表 12 に示した制限を超える場合、データベース エンジンではそのファイルに対し、データ圧縮を行います。

MicroKernel エンジンがシステム データを追加するのはファイル作成時だけです。既存のファイルにシステム データを追加する場合は、[システム データの作成] 設定オプションをオンにしてから **Rebuild ユーティリティ** を使用します。



メモ MicroKernel エンジンがシステム データを追加すると、その結果、レコードが大きすぎてファイルの既存のページ サイズに収まらない場合があります。その場合、MicroKernel エンジンは、ファイルのページ サイズを自動的に次の適切な大きさに調整します。

シャドウ ページング

MicroKernel エンジンは、**シャドウ ページング** を使用して、システム障害が発生した場合に 6.0 以降のファイルが破壊されないようにします。クライアントがトランザクションの内部または外部のページの変更を要求すると、MicroKernel エンジンはデータ ファイル自体の空いた未使用の物理位置を選択し、シャドウ ページと呼ぶ新しいページ イメージをこの新しい場所へ書き込みます。1 回の MicroKernel エンジンの操作で、MicroKernel エンジンは元の論理ページとすべて同じサイズの複数のシャドウ ページを作成することがあります。

変更がコミットされる、つまり、オペレーションが終了するかトランザクションが終了すると、MicroKernel エンジンはシャドウ ページを現在のページにし、元のページが再利用可能になります。MicroKernel エンジンは、有効な再利用可能ページをトラッキングするために、ページ アロケーション テーブルというマップをファイルに格納します。変更がコミットされる前にシステム障害が発生した場合、MicroKernel エンジンは PAT を更新しないため、シャドウ ページを削除し、まだ元の状態のままになっている現在のページを復帰させて使用します。



メモ この説明では、シャドウ ページング プロセスを単純化しています。パフォーマンスの向上のため、MicroKernel エンジンは各オペレーションやユーザー トランザクションを個々にコミットせず、それらをシステム トランザクションと呼ぶまとまりにグループ化します。

クライアントがトランザクション内で動作している場合、元の論理ページに対応するシャドウ ページはそのクライアントにしか見えません。ほかのクライアントが同じ論理ページにアクセスしなければならない場合、元の（未変更の）ページが表示されます。つまり、ほかのクライアントには、最初のクライアントがまだコミットしていない変更は表示されません。元のファイルは常に有効でありかつ内部一貫性があるので、シャドウ ページングにより信頼性は向上します。

ファイルのバックアップ

定期的にファイルをバックアップすることは、データを保護する上で大切なステップです。

ファイルのバックアップの詳細については、『Advanced Operations Guide』の「[ログ、バックアップおよび復元](#)」を参照してください。

イベント ログギグ

イベント ログギグは PSQL の機能の 1 つで、MicroKernel エンジン、SQL インターフェイス、およびユーティリティ コンポーネントからの情報メッセージ、警告メッセージ、エラー メッセージを格納するために使用します。詳細については、『Advanced Operations Guide』の「[メッセージ ログの見直し](#)」を参照してください。

パフォーマンスの向上

MicroKernel エンジンには、パフォーマンスを向上させる以下の機能があります。

- 「システム トランザクション」
- 「メモリ管理」
- 「ページ プリアロケーション」
- 「Extended オペレーション」

システム トランザクション

パフォーマンスを向上し、データの回復を支援するために、MicroKernel エンジンは1つまたは複数のコミットされたオペレーション（トランザクションと非トランザクションの両方）を**システム トランザクション**と呼ぶオペレーションのまとまりに入れます。MicroKernel エンジンは、ファイルごとにシステム トランザクションを作成します。システム トランザクションには、同じエンジンで動作する1つまたは複数のクライアントからのオペレーションとユーザー トランザクションを含めることができます。



メモ システム トランザクションと、排他または並行トランザクションを混同しないでください。本書では、「トランザクション」という用語は排他トランザクションまたは並行トランザクション（ユーザー トランザクションとも呼びます）を指しています。ユーザー トランザクションはキャッシュ内のページに変更を加える方法に影響を与えるのに対して、システム トランザクションはキャッシュ内のダーティページをディスク上のファイルの一部にする方法に影響を与えます。MicroKernel エンジンは、システム トランザクションの起動とプロセスを制御します。

ユーザー トランザクションとシステム トランザクションはともにアトミック トランザクションです。つまり、これらのトランザクションは、すべての変更が行われるか、または変更が何も行われなかったといった具合に発生します。システム障害が発生すると、MicroKernel エンジンは障害の起きたシステム トランザクションに関連するすべてのファイルを再度開くときにそれらのファイルを回復します。障害の起きたシステム トランザクションに対して行われたすべての変更、つまり、終了した最後のシステム トランザクション以降にそのエンジン上のすべてのクライアントによって行われたファイルに対するすべてのオペレーションは失われます。ただし、ファイルは矛盾のない状態に復元されるので、システム障害の原因を解決した後にオペレーションを再試行することができます。

PSQL は、アクセラレイティド モードで開いたファイルを除くすべてのログ記録可能なファイルの**トランザクション一貫性保守**を保証します。（ファイルに少なくとも1つの重複のない重複不能キーが含まれている場合は、そのファイルをログに記録することができます。キーはシステム定義とすることができます。）トランザクション一貫性保守は、MicroKernel エンジンが End Transaction に対し正常終了のステータスコードをクライアントに返す前にトランザクション ログへの書き込みを終了することを保証する機能です。アクセラレイティド モードでファイルを開くと、MicroKernel エンジンはファイルをログに記録しません。したがって、MicroKernel エンジンはファイルにエントリを記録しません。したがって、MicroKernel エンジンはそのファイルのトランザクション一貫性保守を保証できません。

MicroKernel エンジンはファイルのシステム トランザクションをロールバックした後、次にファイルを開くときにログを再生します。これにより、システム トランザクションのロールバックによって、ログには格納されているがファイルに書き込まれていないコミット済みのオペレーションが復元されます。

各システム トランザクションは2つの段階、つまり、準備と書き込みから構成されています。

準備段階

準備段階では、MicroKernel エンジンは現在のシステム トランザクションですべてのオペレーションを実行しますが、ファイルにページを書き込みません。MicroKernel エンジンは必要に応じてキャッシュに登録されていないページをファイルから読み取り、キャッシュにだけ新しいページ イメージを作成します。

以下のアクションはどれも準備段階の終了を引き起こし、書き込み段階の開始を示します。

- MicroKernel エンジンがオペレーション バンドル制限に達した。
- MicroKernel エンジンが起動時間制限に達した。
- キャッシュ ページの合計数に対する書き込み準備されたページの比率がシステムのしきい値に達した。



メモ 一般に、準備段階は MicroKernel エンジン オペレーションの完了後に終了します。しかし、ユーザー トランザクションが完了する前に時間制限またはキャッシュしきい値に達する可能性があります。MicroKernel エンジンは、いずれにしても書き込み段階に切り替わります。

書き込み段階

書き込み段階では、MicroKernel エンジンは準備段階で準備されたすべてのページをディスクに書き込みます。MicroKernel はまず、すべてのデータ ページ、インデックス ページおよび可変ページを書き込みます。これらのページは実際にはシャドウ ページです。これらのページが書き込まれている間、ディスク上のファイルの一貫性は変わりません。

しかし、PAT ページは現在のページとしてシャドウ ページを指示するため、システム トランザクションの重要部分は PAT ページが書き込まれている間に発生します。この段階を保護するために、MicroKernel エンジンは FCR にフラグを書き込みます。すべての PAT ページが書き込まれると、最後の FCR が書き込まれるため、ファイルは一貫性を保ちます。この段階でシステム障害が発生すると、次にファイルが開かれたときに MicroKernel エンジンはシステム障害を認識し、ファイルを直前の状態にロールバックします。次に、トランザクション ログ ファイル内の一貫性保守可能なユーザー トランザクションはすべてファイルに書き込まれます。

システム トランザクションの頻度

頻度が低い

システム トランザクションの頻度を低くすると、ほとんどの構成ではパフォーマンスが向上します。その中には、ファイルが排他的に開かれる、クライアント / サーバー、シングルのエンジン ワークステーションおよびマルチエンジン ワークステーション環境があります。

MicroKernel エンジンがシステム トランザクションの起動回数を少なくすると、**ダーティページ**、つまり、書き込みが必要なページはメモリ内に長く存在します。アプリケーションが多数の変更操作を行っている場合、これらのページはディスクに書き込まれる前に何回も更新される可能性があります。つまり、ディスクの書き込みが少なくなります。実際に、最も効率の良いエンジンは必要なときだけ書き込まれるエンジンです。

到達するとシステム トランザクションが起動される制限には、オペレーション バンドル制限、起動時間制限、キャッシュ サイズの3つがあります。これらの制限に達すると、MicroKernel エンジンはシステム トランザクションを起動します。この設定の詳細については、『Advanced Operations Guide』を参照してください。

システム トランザクションの回数を減らす最も良い方法は、オペレーション バンドル制限と起動時間制限をさらに高い値に設定する方法です。キャッシュのサイズを増やすこともできます。

頻度が高い

MicroKernel エンジンがシステム トランザクションを実行する回数を少なくすることの欠点は、ディスクに書き込まなければならないマシン メモリ内のデータが常に多くなるという点です。停電などのシステム障害が発生すると、失われるデータが多くなります。MicroKernel エンジンはファイルを一貫性のある使用可能な状態に保つようにつくられていますが、その状態には最新の変更が含まれていない場合があります。もちろん、トランザクション一貫性保守でユーザー トランザクションを使用すると、このリスクが最も少なくなります。パフォーマンスの向上に対してシステム トランザクションの回数を減らすことのリスクをよく考慮してください。

たとえば、アプリケーションがワークステーション エンジンを使用し、低速または信頼性の低いネットワーク接続を通じてリモート ファイルを更新している場合、変更データができるだけ早くディスクに書き込まれるようにシステム トランザクションを頻繁に実行する必要があります。

メモリ管理

キャッシュは、読み取るページをバッファースするために MicroKernel エンジンが予約するメモリ領域です。アプリケーションがレコードを要求すると、MicroKernel エンジンはず、そのレコードを含むページが既にメモリ内にあるかどうかを確認するためにキャッシュを調べます。そうであれば、MicroKernel エンジンレコードをキャッシュからアプリケーションのデータ バッファに転送します。ページがキャッシュ内になければ、MicroKernel エンジンページをディスクからキャッシュ バッファに読み込んでから要求されたレコードをアプリケーションに転送します。MicroKernel エンジン キャッシュはローカル クライアントにより共有され、複数のオペレーション間で使用されます。

MicroKernel エンジンが新しいページをメモリに転送しようとするときにすべてのキャッシュ バッファがいっぱいであれば、MicroKernel エンジンがキャッシュ内のどのページを上書きするかを LRU (least-recently-used) アルゴリズムが決定します。LRU アルゴリズムは、最近参照されたページをメモリに保持することによって処理時間を短縮します。

アプリケーションがレコードを挿入または更新すると、MicroKernel エンジンはず対応するページのシャドウ イメージを作成し、キャッシュでそのページを変更し、次にそのページをディスクに書き込みます。変更されたページは、MicroKernel エンジンが新しいページでキャッシュ内のそのページのイメージを上書きできると LRU アルゴリズムが決定するまで、キャッシュ内に残ります。

一般に、キャッシュが大きいほどパフォーマンスが向上するのは、ある時点でメモリ内に保持するページ数を多くすることができるからです。MicroKernel エンジンを使用して、I/O キャッシュ バッファに予約するメモリ量を指定することができます。メモリ量を決定する場合は、アプリケーションの必要メモリ量、コンピューターにインストールされた総メモリ量、および、すべての並行 PSQL アプリケーションがアクセスする全ファイルの結合サイズを考慮してください。このキャッシュの設定は、[キャッシュ割当サイズ]で行います。

PSQL V8 以降のリリースでは、2 番目の動的 L2 キャッシュも使用できます。この動的キャッシュの設定は、[MicroKernel エンジンの最大メモリ使用量] (v9.1 以降では [MicroKernel の最大メモリ使用量])で行います。これらの設定の詳細については、『Advanced Operations Guide』を参照してください。



メモ 使用可能な物理メモリよりキャッシュを多くすると、実際にパフォーマンスが大きく低下する可能性があるのは、仮想メモリ内のキャッシュメモリの一部がディスク上にスワップされるからです。MicroKernel エンジン キャッシュは、オペレーティング システムがロードされた後で使用可能な物理メモリの約 60% に設定することをお勧めします。

たとえば、Windows NT 上でこの値を見つけるには、タスクバーで時計を右クリックし、[タスク マネージャー]を選択します。[パフォーマンス] タブを選択すると、ダイアログ ボックスの右下近くに使用可能な物理メモリが表示されます。

ページ プリアロケーション

ページ プリアロケーションは、MicroKernel エンジンがディスク領域を必要とするときにそのスペースを使用できるようにします。データ ファイルがディスク上の連続する領域を占有する場合、ファイル操作を高速化することができます。速度の向上は、非常に大きなファイルで最も顕著です。この機能の詳細については、「[ページ プリアロケーション](#)」を参照してください。

Extended オペレーション

Extended オペレーション - Get Next Extended (36)、Get Previous Extended (37)、Step Next Extended (38)、Step Previous Extended (39) および Insert Extended (40) - を使用すると、パフォーマンスを大幅に向上できます。Extended オペレーションは、アプリケーションにもよりますが、MicroKernel エンジンの要求数を 100 分の 1 以下に減らすことができます。これらのオペレーションには、不要なレコードがアプリケーションに送られないように返されたレコードにフィルターをかける機能があります。この最適化手法は、ネットワークを介してデータを送受信しなければならないクライアント / サーバー環境で最も良い結果をもたらします。

これらのオペレーションの詳細については、「[マルチレコードのオペレーション](#)」を参照してください。

ディスク使用量

MicroKernel エンジンには、必要なディスク使用量を最小限にするための以下の機能があります。

- 「[空きスペース リスト](#)」
- 「[インデックス バランスの実行](#)」
- 「[データ圧縮](#)」
- 「[ブランク トランケーション](#)」

空きスペース リスト

レコードを削除すると、そのレコードがこれまで占有したディスク領域が空きスペース リストに設定されます。新しいレコードを挿入すると、MicroKernel エンジンは新しい可変ページを作成する前に空きスペース リスト上のページを使用します。空きスペース スレッシュホールドは、ページが空きスペース リストに表示されるには、可変ページにどれだけの空きスペースが残っている必要があるかを MicroKernel エンジンに示します。

このような空き領域を再利用する方法では、ディスク領域を再利用するようにファイルを再編成する必要があります。また、空きスペース リストを使用すると、数ページにまたがる可変長レコードの断片化を抑えることができます。空きスペース スレッシュホールドを高く設定すると断片化を減らすことができますが、ファイルのためのディスク領域がより多く必要になります。

インデックス バランスの実行

インデックス ページがいっぱいになると、MicroKernel エンジンはデフォルトで新しいインデックス ページを自動的に作成し、いくつかの値をいっぱいになったインデックス ページから新しいインデックス ページへ移動します。[インデックス バランスの実行] オプションをオンにすると、既存のインデックス ページがいっぱいになるたびに新しいインデックス ページを作成しないようにすることができます。インデックス バランスを使用すると、MicroKernel エンジンはインデックス ページがいっぱいになるたびに、兄弟インデックス ページ内で利用可能な領域を探します。MicroKernel エンジンは次に、いっぱいになったインデックス ページから空き領域のあるインデックス ページへ値を振り分けます。

インデックス バランスは、インデックス ページの使用度を高めることにより、ページ数を減らし、同レベルのノード間でキーを均等に分配するため、読み取り操作でのパフォーマンスが向上します。しかし、この機能を使用すると、MicroKernel エンジンはより多くのインデックス ページを調べるために余分な時間が必要になったり、書き込み操作でより多くのディスク I/O が必要になったりする可能性もあります。インデックス バランスの正確な影響は状況によって異なりますが、インデックス バランスを使用した場合、書き込み操作のパフォーマンスは概して約 5 ～ 10% 低下します。

インデックス バランスは「定常状態」のファイルのパフォーマンスに影響を与えます。普通の変更操作は多少遅くなりますが、普通の取得操作は速くなります。平均的なインデックス ページにより多くのキーを収容することによって、これを行います。通常のインデックス ページは 50 ～ 65% まで満たされ、この場合、インデックス バランスされたページは 65 ～ 75% まで満たされます。つまり、検索するインデックス ページが少なくなります。

Create Index (31) でインデックスを作成すると、インデックス ページは 100% フルに近くなるため、これらのファイルが書き込みでなく読み取りについて最適化されます。



メモ ファイル内の File Flag フィールドに Index Balanced File ビットを設定する方法で、ファイル単位でインデックス バランスを指定することもできます。

[インデックス バランスの実行] オプションを有効にすると、アプリケーションが設定したバランス ファイルフラグの指定とは無関係に、MicroKernel エンジンはすべてのファイルでインデックス バランスを行います。インデックス バランス設定オプションの指定方法については、『PSQL User's Guide』を参照してください。

データ圧縮

MicroKernel エンジンではデータ圧縮を使用して、ファイルのレコードを挿入または更新する前にそれらのレコードを圧縮し、それらのレコードを取得するときにレコードを解凍します。圧縮されたレコードの最終の長さはそのレコードがファイルに書き込まれるまで決定できないので、MicroKernel エンジンでは常に圧縮されたファイルを可変長レコード ファイルとして指定します。ただし、固定長ファイルでデータ圧縮を使用すると、MicroKernel エンジンは Insert オペレーションと Update オペレーションで、データ ファイルに指定された固定レコード長より長いレコードを生成しません。

MicroKernel エンジンでは圧縮されたレコードを可変長レコードとして記録するので、頻繁な挿入、更新および削除を行う場合は個々のレコードがいくつかのデータ ページにわたってフラグメント化される場合があります。MicroKernel エンジンは 1 つのレコードを検索するために複数のファイル ページを読み取らなければならない場合があるので、この断片化によってアクセス時間が遅くなるおそれがあります。しかし、データ圧縮により、多数の繰り返し文字を含むレコードを格納する場合に必要なディスク容量を、大幅に削減することもできます。MicroKernel エンジンは、5 つ以上の同じ連続文字を 5 バイトに圧縮します。

この機能の詳細については、「[レコード圧縮](#)」を参照してください。

ブランク トランケーション

ブランク トランケーションはディスク容量を節約します。可変長のレコードを許し、かつデータ圧縮機能を使用しないファイルにのみ適用できます。この機能の詳細については、「[ブランク トランケーション](#)」を参照してください。

データベースの設計

5

この章では、データベースを設計するための形式とガイドラインを示します。以下の項目が含まれます。

- 「データ ファイルについて」
- 「データ ファイルの作成」
- 「論理レコード長の計算」
- 「ページ サイズの選択」
- 「ファイル サイズの予測」
- 「データベースの最適化」
- 「セキュリティの設定」

データ ファイルについて

MicroKernel エンジンでは、情報をデータ ファイルに保存します。各データ ファイル内には、レコードとインデックスの集合があります。レコードにはデータのバイトが格納されています。そのデータは、社員の名前、ID、住所、電話番号、賃率などを表します。インデックスは、レコードの一部に特定の値を含むレコードをすばやく見つける働きがあります。

MicroKernel エンジンでは、レコードを単なるバイトの集合と解釈します。これは、レコード内の情報で論理的に区別される部分、つまり、フィールドを認識しません。MicroKernel エンジンにおいては、ラスト ネーム、ファースト ネーム、社員 ID などはレコード内に存在しません。レコードは単にバイトの集合にすぎません。

MicroKernel エンジンではバイト指向であるため、たとえデータ型を宣言するキーに対してでも、レコード内のデータの変換、型検査または妥当性検査を行いません。データ ファイルとインターフェイスをとるアプリケーションは、そのファイル内のデータの形式と型に関するすべての情報を処理する必要があります。たとえば、アプリケーションは以下の形式に基づいてデータ構造を使用します。

レコード内の情報	長さ (バイト単位)	データ型
ラスト ネーム	25	ヌル終了文字列
ファースト ネーム	25	ヌル終了文字列
ミドル イニシャル	1	Char (バイト)
社員 ID	4	Long (4 バイト整数)
電話番号	13	ヌル終了文字列
月給	4	Float
合計レコード長	72 バイト	

ファイル内では、社員のレコードがバイトの集合として格納されます。以下のダイアグラムは、Cliff Jones のレコードのデータがファイルに格納されるときの状態を示したものです。(このダイアグラムは、文字列の ASCII 値を対応する英字または数字に置き換えています。整数とほかの数値は、正規 16 進表現から変更されていません。)

バイト位置	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
データ値	J	o	n	e	s	00	?	?	?	?	?	?	?	?	?	?

バイト位置	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
データ値	?	?	?	?	?	?	?	?	?	C	l	i	f	f	00	?

バイト位置	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
データ値	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

バイト位置	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
データ値	?	?	D	2	3	4	1	5	1	2	5	5	5	1	2	1

バイト位置	40	41	42	43	44	45	46	47
データ値	2	?	?	?	3	5	0	0

MicroKernel エンジンがファイル内で認識する情報で区別する唯一の部分は**キー**です。アプリケーション（またはユーザー）はレコード内のバイトの1つまたは複数の集合をキーとして指定できますが、バイトと各**キー セグメント**内で連続していなければなりません。

MicroKernel エンジンは指定されたキーの値に基づいてレコードをソートし、特定の順序でデータを返すための直接アクセスを行います。MicroKernel エンジンは、指定されたキー値に基づいて特定のレコードも検索できます。前の例では、各レコード内のラスト ネームを含む 25 バイトをファイル内のキーとして指定できます。アプリケーションはラスト ネームのキーで **Smith** という名前の全社員のリストを取得したり、全社員のリストの取得後にラスト ネームでソートされたそのリストを表示したりすることができます。

キーを使用すると、MicroKernel エンジンは情報にすばやくアクセスできます。MicroKernel エンジンは、データ ファイルで定義されたキーごとに**インデックス**を作成します。インデックスはデータ ファイル自体の中に格納され、そのファイル内の実際のデータへのポインターの集合が含まれています。キーの値は各ポインターに関連付けられています。

前の例では、ラスト ネーム キーのインデックスがラスト ネーム値をソートし、レコードがデータ ファイル内のどこにあるかを示すポインターを持っています。

インデックス	レコード		
Anderson	→ Anderson	Gayle	P10927365751255516550
Boerner	→ Nemet	Judit	L12345678901234567890
Bowling	→ Quaglino	Andy	X98765432109876543210
Harris	→ Harris	Ron	Q65748392019283764650
Nemet	→ Boerner	Clarissa	L82937465637298173640
Quaglino	→ Woodward	Nancy	B92736464838161537480
Woodward	→ Bowling	Mark	G92736465483892917370

通常、アプリケーションが情報にアクセスしたり、情報をソートする場合、MicroKernel エンジンはデータ ファイル内のすべてのデータを検索するわけではありません。その代わりに、インデックスを使用して検索を行い、アプリケーションの要求を満たすレコードだけを処理します。

データ ファイルの作成

MicroKernel エンジンは、データベース アプリケーションを最適化する際の大きな柔軟性を開発者に与えてくれます。そのような柔軟性を提供するため、MicroKernel エンジンは、MicroKernel エンジンの内部処理の多くを公開しています。MicroKernel エンジン初めて使用する場合、Create (14) オペレーションは非常に複雑に見えるかも知れませんが、作業の開始にあたってはこのオペレーションのすべての機能を必要とするわけではありません。ここでは、簡単なトランザクション ベースのデータ ファイルの作成を段階的に行うことで、基本的な条件に焦点を当てます。ここでは、必要に応じて簡素化を図るために、C インターフェイスの用語を使用します。



メモ 同じディレクトリに、ファイル名が同一で拡張子のみが異なるようなファイルを置かないでください。たとえば、同じディレクトリ内のデータ ファイルの 1 つに Invoice.btr、もう 1 つに Invoice.mkd という名前を付けてはいけません。このような制限が設けられているのは、データベース エンジンがさまざまな機能でファイル名のみを使用し、ファイルの拡張子を無視するためです。ファイルの識別にはファイル名のみが使用されるため、ファイルの拡張子だけが異なるファイルは、データベース エンジンでは同一のものであると認識されます。

データ レイアウト

ここでは、社員レコードを格納するデータ ファイルの例を使用します。アプリケーションは、一意の社員 ID または社員のラスト ネームを指定して社員情報を取得します。複数の社員が同じラスト ネームを持っている可能性があるため、データベースではラスト ネームの重複値を許します。これらの基準に基づくと、ファイルのデータ レイアウトは以下のようになります。

レコード内の情報	データ型	キーまたはインデックスの特性
ラスト ネーム	25 文字の文字列	重複可能
ファースト ネーム	25 文字の文字列	なし
ミドル イニシャル	1 文字の文字列	なし
社員 ID	4 バイト整数	重複不可
電話番号	13 文字の文字列	なし
月給	4 バイト浮動小数	なし

基本的なデータ レイアウトを設定したので、MicroKernel エンジンの用語と条件の適用を開始できます。この場合、実際にファイルを作成する前にキー構造とファイル構造に関する情報を決定します。これらの詳細情報を事前に処理しておかなければならないのは、Create (14) オペレーションがファイル情報、インデックス情報、キー情報をすべて一度に作成するからです。以降では、これらの詳細情報を処理する際に考慮しなければならない問題について説明します。

キー属性

まず、キーの特別な機能を決定します。以下の表に示すように、MicroKernel エンジンにはさまざまなキー属性があり、必要に応じ割り当てることができます。

表 13 キー属性

定数	説明
EXTTYPE_KEY	拡張データ型。文字列または符号なしバイナリ以外の MicroKernel エンジン データ型を格納します。標準バイナリ データ型よりもこの属性を使用してください。このキー属性には、標準バイナリおよび文字列データ型のほかに多数の属性を収容できます。
BIN	標準バイナリ データ型。今日までの経緯によりサポートされます。符号なし 2 進数を格納します。デフォルトのデータ型は文字列です。
DUP	リンク重複。ポインターでインデックス ページからデータ ページへリンクされる重複値を許します。詳細については、「重複キー」を参照してください。
REPEAT_DUPS_KEY	繰り返し重複。インデックス ページとデータ ページの両方に格納される重複値を許します。詳細については、「重複キー」を参照してください。
MOD	変更可能。レコードの挿入後にキー値の変更を行えます。
SEG	セグメント化。このキーが、現在のキー セグメントの後にセグメントを持つことを指定します。
NUL	ヌル キー（すべてのセグメント）。指定されたヌル値がキーのすべてのセグメントに含まれている場合は、インデックスからすべてのレコードを除外します（ファイルを作成するときにヌル値を指定します）。
MANUAL_KEY	ヌル キー（任意のセグメント）。指定されたヌル値がキーの任意のセグメントに含まれている場合は、インデックスからすべてのレコードを除外します（ファイルを作成するときにヌル値を指定します）。
DESC_KEY	降順のソート順序。キー値を降順（最大から最小へ）に並べます。デフォルトは昇順（最小から最大へ）です。
NOCASE_KEY	大文字と小文字の区別。大文字と小文字を区別せずに文字列値をソートします。キーにオルタネート コーレーティング シーケンス（ACS）がある場合は使用しないでください。ヌル インジケータ セグメントの場合、ゼロではないヌル値を明確に区別するために重ねて定義されます。
ALT	オルタネート コーレーティング シーケンス。ACS を使用して標準の ASCII 照合順序から異なる方法で文字列キーをソートします。各種キーは異なる ACS を使用できます。デフォルト ACS（ファイルで定義された最初の ACS）、ファイル内で定義された番号付き ACS、または、COLLATE.CFG システム ファイルで定義された名前付き ACS を指定できます。
NUMBERED_ACS	
NAMED_ACS	
簡素化を図るために、これらの定数は btrconst.h で定義され、C インターフェイスに合致しています。インターフェイスの中には、ほかの名前を使用したり、定数をまったく使用しないものがあります。ビット マスク、キー属性の 16 進値および 10 進値については、『Btrieve API Guide』を参照してください。	

定義するキーごとに、これらのキー属性を指定します。各キーは独自のキー仕様を持ちます。キーに複数のセグメントがある場合は、セグメントごとに仕様を提供する必要があります。これらの属性のいくつかは、同じキー内の異なるセグメントに対して異なる値を持つことができます。前の例で、キーはラスト ネームおよび社員 ID です。どちらのキーも拡張型を使用します。つまり、ラスト ネームは文字列、社員 ID は整数です。両方とも変更可能ですが、ラスト ネームだけは重複可能です。また、ラスト ネームは大文字と小文字を区別します。

キーに割り当てるデータ型に関して、MicroKernel エンジンは、入力レコードがそのキーに対し定義されたデータ型に従っているかどうかを確認しません。たとえば、ファイルで `TIMESTAMP` キーを定義したとします。そこに文字列を格納したり、または日付キーを定義し 2 月 30 日の値を格納したりすることができます。お使いの MicroKernel エンジン アプリケーションは問題なく機能しますが、同じデータにアクセスしようとする ODBC アプリケーションではうまくいかない可能性があります。それは、バイト 形式が異なり、タイムスタンプ値を生成

するためのアルゴリズムが異なるためです。データ型の説明については、『SQL Engine Reference』を参照してください。

ファイル属性

次に、ファイルの特別な機能を決定します。

以下の表に示すように、MicroKernel エンジンではさまざまな種類のファイル属性が指定できます。

表 14 ファイル属性

定数	説明
VAR_RECS	可変長レコード。可変長レコードを含むファイルで使用します。
BLANK_TRUNC	ブランク トランケーション。レコードの可変長部分の末尾の空白を削除することによって、ディスク領域を節減します。可変長のレコードを許し、かつデータ圧縮機能を使用しないファイルにのみ適用できます。詳細については、「 ブランク トランケーション 」を参照してください。
PRE_ALLOC	ページ プリアロケーション。ファイルの作成時にファイルで使用される連続ディスク領域を予約します。ファイルがディスク上の連続する領域を占有する場合、ファイル操作を高速化することができます。速度の向上は、非常に大きなファイルで最も顕著です。詳細については、「 ページ プリアロケーション 」を参照してください。
DATA_COMP	データ圧縮。レコードを挿入または更新する前に圧縮し、レコードの取得時に解凍します。詳細については、「 レコード圧縮 」を参照してください。
KEY_ONLY	キーオンリー ファイル。キーを 1 つだけ含み、レコード全体がそのキーと共に格納されるため、データ ページは不要です。キーオンリー ファイルは、レコードに単一のキーが含まれており、かつそのキーが各レコードの大部分を占有している場合に有効です。詳細については、「 キーオンリー ファイル 」を参照してください。
BALANCED_KEYS	インデックス バランス。いっぱいになったインデックス ページから領域が空いているインデックス ページへ値を振り分けます。インデックス バランスは読み取り操作でのパフォーマンスを高めますが、書き込み操作では余計な時間がかかる場合があります。詳細については、「 インデックス バランス 」を参照してください。
FREE_10 FREE_20 FREE_30	空きスペース スレッシュホールド。可変長レコードを削除して空いたディスク領域を再利用するためのスレッシュホールド パーセンテージを設定することで、ファイルを認識する必要がなくなり、数ページにわたる可変長レコードの断片化を減らします。 空きスペース スレッシュホールドが大きいと、レコードの可変長部分の断片化が減少しパフォーマンスが向上します。ただし、ディスク領域がさらに必要になります。パフォーマンスを高くしたいときは、空きスペース スレッシュホールドを 30 パーセントに増やします。
DUP_PTRS	重複ポインターを予約します。将来追加するリンク重複キーにポインター スペースをプリアロケートします。リンク重複キーを作成するための重複ポインターがない場合、MicroKernel エンジンは繰り返し重複キーを作成します。
INCLUDE_SYSTEM_DATA	システム データ。ファイル作成時にシステム データを取り込むことによって、MicroKernel エンジンがファイルにログオンして処理を行うことを許可します。これは、一意のキーを含まないファイルで有効です。
NO_INCLUDE_SYSTEM_DATA	
SPECIFY_KEY_NUMS	キー番号。MicroKernel エンジンが番号を自動的に割り当てるのではなく、特定の番号をキーに割り当てることができます。アプリケーションによっては、特定のキー番号を必要とするものがあります。

表 14 ファイル属性

定数	説明
VATS_SUPPORT	可変長部割り当てテーブル (VAT)。VAT (レコードの可変長部分へのポインターの配列) を使用して、ランダム アクセスを加速化したり、データ圧縮時に使用される圧縮バッファのサイズを制限します。詳細については、「 可変長部割り当てテーブル 」を参照してください。
簡素化を図るために、これらの定数は <code>btrconst.h</code> で定義され、C インターフェイスに合致しています。インターフェイスの中には、ほかの名前を使用したり、定数をまったく使用しないものがあります。ビット マスク、ファイル属性の 16 進値および 10 進値については、『 Btrieve API Guide 』を参照してください。	

データ ファイルの例でこれらのファイル属性を使用していないのは、レコードが小さなサイズの固定長レコードであるからです。

ファイル属性の定義については、「[ファイル タイプ](#)」を参照してください。Create オペレーションにおけるファイル属性の詳細については、『[Btrieve API Guide](#)』を参照してください。

ファイル仕様およびキー仕様の構造体の作成

Create オペレーションを使用する場合は、ファイルとキーの仕様情報をデータ バッファに渡します。以下の構造体では、社員データ ファイルの例を使用します。

表 15 ファイル仕様およびキー仕様のサンプル データ バッファ

説明		データ型 ¹	バイト番号	値の例 ²	
ファイル仕様					
論理固定レコード長（結合されたすべてのフィールドのサイズ：25 + 25 + 1 + 4 + 13 + 4）。手順については、「 論理レコード長の計算 」を参照してください。 ³		Short Int ⁴	0, 1	72	
ページ サイズ。	ファイル形式	Short Int	2, 3		
	6.0-9.0			512	
	6.0 以上			1,024	
	6.0-9.0			1,536	
	6.0 以上			2,048	
	6.0-9.0			3,072	
				3,584	
	6.0 以上			4,096	
	9.0 以上			8,192	
	9.5 以上			16,384	
	<p>ほとんどのファイルでは最小サイズの 4096 バイトが最も効率的です。微調整を行う場合の詳細については、「ページ サイズの選択」を参照してください。</p> <p>6.0 から 8.0 のファイル形式ではページ サイズ 512 の x 倍をサポートします。x は乗算の値が 4,096 以下になる数字です。</p> <p>9.0 ファイル形式ではページ サイズ 8,192 もサポートすることを除けば、以前のバージョンと同じページ サイズをサポートします。</p> <p>9.5 ファイル形式では、ページ サイズ 1,024 の 2⁰ 倍から 2⁴ 倍をサポートします。</p> <p>9.5 形式のファイルを作成する場合、指定された論理ページ サイズがそのファイル形式で有効ならば、MicroKernel は指定値の次に大きな有効値があるかどうかを調べ、存在する場合はその値に切り上げます。それ以外の値やファイル形式の場合、オペレーションはステータス 24 で失敗します。古いバージョンのファイル形式では切り上げは行われません。</p>				
キー数。（ファイル内のキー数：2）		Byte	4	2	
ファイル バージョン		Byte	5	0x60	バージョン 6.0
				0x70	バージョン 7.0
				0x80	バージョン 8.0
				0x90	バージョン 9.0
				0x95	バージョン 9.5
				0x00	データベースエンジンのデフォルトを使用
予約済み（Create オペレーションでは使用しません。）		予約済み	6- 9	0	

表 15 ファイル仕様およびキー仕様のサンプル データ バッファ

説明	データ型 ¹	バイト 番号	値の例 ²
ファイル フラグ。ファイル属性を指定します。ファイルの例では、ファイル フラグを使用していません。	Short Int	10, 11	0
追加ポインター数。将来のキーの追加のために予約する重複ポインター数を設定します。ファイル属性で予約重複ポインターを指定する場合に使用します。	Byte	12	0
予約済み (Create オペレーションでは使用しません。)	予約済み	13	0
ブリアロケート ページ数。事前に割り当てられるページ数を設定します。ファイル属性でページ ブリアロケーションを指定する場合に使用します。	Short Int	14, 15	0
キー 0 (ラスト ネーム) のキー仕様			
キー ポジション。レコード内のキーの最初のバイトの位置を指定します。レコード内の最初のバイトは 1 です。	Short Int	16, 17	1
キー長。バイト単位でキーの長さを指定します。	Short Int	18, 19	25
キー フラグ。キー属性を指定します。	Short Int	20, 21	EXTTYPE_KEY + NOCASE_KEY + DUP + MOD
Create には使用しません。	Byte	22-25	0
拡張キー タイプ。キー フラグで「拡張キー タイプを使用する」を指定する場合に使用します。拡張データ型のうちの 1 つを指定します。	Byte	26	ZSTRING
ヌル値 (レガシー ヌルのみ)。キー フラグで「ヌルキー (すべてのセグメント)」または「ヌルキー (任意のセグメント)」を指定する場合に使用します。キーの除外値を指定します。レガシー ヌルと真のヌルの概念については、「 ヌル値 」を参照してください。	Byte	27	0
Create には使用しません。	Byte	28, 29	0
手動割り当てキー番号。ファイル属性で「キー番号」を指定する場合に使用します。キー番号を割り当てます。	Byte	30	0
ACS 番号。キー フラグで「デフォルトの ACS を使用する」、「ファイル内の番号付きの ACS を使用する」または「名前付きの ACS を使用する」を指定する場合に使用します。使用する ACS 番号を指定します。	Byte	31	0
キー 1 (社員 ID) のキー仕様			
キー ポジション。(社員 ID は、ミドル イニシャルの後の最初のバイトから始まります。)	Short Int	32, 33	52
キー長。	Short Int	34, 35	4
キー フラグ。	Short Int	36, 37	EXTTYPE_KEY + MOD
Create には使用しません。	Byte	38-41	0
拡張キー タイプ。	Byte	42	INTEGER
ヌル値。	Byte	43	0
Create には使用しません。	Byte	44, 45	0

表 15 ファイル仕様およびキー仕様のサンプル データ バッファ

説明	データ型 ¹	バイト番号	値の例 ²
手動割り当てキー番号。	Byte	46	0
ACS 番号。	Byte	47	0
ページ圧縮のキー仕様			
物理ページサイズ ⁵	Char	A	512 (デフォルト値)
¹ 特に指定がない場合、すべてのデータ型は符号なしです。 ² 簡素化を図るため、数値以外の値の例は C アプリケーションの場合です。 ³ 可変長レコードを持つファイルの場合、論理レコード長はレコードの固定長部分のみを指します。 ⁴ Short Integer (Short Int) は「リトル エンディアン」のバイト順、つまり、Intel 系のコンピュータが採用している下位バイトから上位バイトへ記録する方式で格納する必要があります。 ⁵ ページ レベル圧縮でのみ使用します。ページ圧縮ファイルフラグ (表 6 を参照) と組み合わせて使用する必要があります。詳細については、「 ページ レベル圧縮を用いたファイルの作成 」を参照してください。			

ページ レベル圧縮を用いたファイルの作成

PSQL 9.5 以上では、Create オペレーションを使用してページ レベルの圧縮を用いたデータ ファイルを作成することができます。古いバージョンのデータ ファイルの場合は、論理ページを物理ページに割り当て、この割り当てをページ アロケーション テーブル (PAT) に格納します。物理ページのサイズは論理ページのサイズと同一です。

ファイルが圧縮されると、各論理ページが 1 つ以上の物理ページ単位に圧縮され、そのサイズは 1 論理ページよりも小さくなります。物理ページのサイズは、「物理ページサイズ」属性 (表 15 を参照) で指定されます。

ページ圧縮ファイルフラグ (表 6 を参照) は物理ページ サイズのキー仕様と組み合わせて使用され、ページ レベルの圧縮を適用したデータ ファイルを新規作成するよう MicroKernel へ通知します。論理ページサイズと物理ページサイズは次のように検証されます。

物理ページサイズに指定された値は、論理ページサイズに指定された値よりも大きくすることはできません。物理ページサイズに指定された値の方が大きい場合、MicroKernel は論理ページサイズと同じになるようその値を切り捨てます。論理ページサイズは物理ページサイズの倍数になっていなければなりません。倍数でない場合、その論理ページサイズの値は物理ページサイズの値のちょうど倍数になるよう切り捨てられます。このような操作の結果として、論理ページサイズと物理ページサイズの値が同じになった場合、ページ レベルの圧縮はこのファイルに適用されません。

Create オペレーションの呼び出し

Create オペレーション (14) では、以下の値が必要です。

- オペレーション コード。Create の場合は 14。
- ファイル仕様とキー仕様を含むデータ バッファ。
- データ バッファの長さ。
- ファイルの絶対パスを含むキー バッファ。
- 同じ名前のファイルが既に存在する場合に、MicroKernel エンジンが警告を出すかどうか (-1 = 警告、0 = 警告なし) を決定するための値を含むキー番号。

C における API 呼び出しは次のようになります。

Create オペレーション

```

/* データ バッファ構造体の定義 */
/* 以下の 3 つの構造体は、1 バイトのフィールドのアライメントを持っている必要があります。これは、コンパイラのプラグマを介して、またはコンパイラ オプションを介して行うことができます。*/
typedef struct
{
    BTI_SINT recLength;
    BTI_SINT pageSize;
    BTI_SINT indexCount;
    BTI_CHAR reserved[4];
    BTI_SINT flags;
    BTI_BYTE dupPointers;
    BTI_BYTE notUsed;
    BTI_SINT allocations;
} FILE_SPECS;

typedef struct
{
    BTI_SINT position;
    BTI_SINT length;
    BTI_SINT flags;
    BTI_CHAR reserved[4];
    BTI_CHAR type;
    BTI_CHAR null;
    BTI_CHAR notUsed[2];
    BTI_BYTE manualKeyNumber;
    BTI_BYTE acsNumber;
} KEY_SPECS;

typedef struct
{
    FILE_SPECS fileSpecs;
    KEY_SPECS keySpecs[2];
} FILE_CREATE_BUF;
/* データ バッファの作成 */
FILE_CREATE_BUF dataBuf;
memset (dataBuf, 0, size of (dataBuf)); /* dataBuf の初期化 */
dataBuf.recLength = 72;
dataBuf.pageSize = 4096;
dataBuf.indexCount = 2;
dataBuf.keySpecs[0].position = 1;
dataBuf.keySpecs[0].length = 25;
dataBuf.keySpecs[0].flags = EXTTYPE_KEY + NOCASE_KEY + DUP + MOD;
dataBuf.keySpecs[0].type = ZSTRING;
dataBuf.keySpecs[1].position = 52;
dataBuf.keySpecs[1].length = 4;
dataBuf.keySpecs[1].flags = EXTTYPE_KEY;
dataBuf.keySpecs[1].type = INTEGER;
/* ファイルの作成 */
strcpy((BTI_CHAR *)keyBuf, "c:¥¥sample¥¥sample2.mkd");
dataLen = sizeof(dataBuf);
status = BTRV(B_CREATE, posBlock, &dataBuf, &dataLen, keyBuf, 0);

```

Create Index オペレーション

あらかじめキーが定義されているファイルを作成すると、挿入、更新、または削除のたびにインデックスにデータが設定されます。これは、ほとんどのデータベースファイルに必要です。しかし、読み取る前にデータが完全

に取り込まれる類のデータベース ファイルがあります。これらには、テンポラリ ソート ファイルや、製品の一部として提供されるデータ実装済みのファイルが挙げられます。

このようなファイルの場合、レコードが書き込まれた後に **Create Index (31)** でキーを作成する方が処理が速くなる可能性があります。レコードの挿入をより速く実行できるように、インデックスを定義せずにファイルを作成する必要があります。その後、**Create Index** オペレーションによって、より効率的な方法でキーをソートし、インデックスを作成します。

このように作成されたインデックスは効率も良く、高速なアクセスができます。インデックス ページはキー順に読み込まれるので、**Create Index (31)** の実行時、**MicroKernel** エンジン各ページの終わりに空き領域を残す必要がありません。各ページは、100% 近くまでレコードが書き込まれます。それに対して、**Insert (2)** または **Update (3)** オペレーションでインデックス ページにレコードを書き込む場合、インデックス ページは半分に分割され、レコードの半分は新しいページにコピーされます。このプロセスにより、平均的なインデックス ページは約 50 ~ 65% まで満たされます。インデックス バランスを使用すると、65 ~ 75% まで満たされる場合があります。

Create Index (31) オペレーションのもう 1 つの利点として、作成された新しいインデックス ページはすべてファイルの終わりにまとめて書き込まれます。これは、大きなファイルの場合、読み取り処理間で読み取りヘッドの移動距離が短くなるということです。

この手法では、レコードが数千件しかない小さなファイルの場合、処理速度が上がらない可能性があります。このオペレーションの利点を生かすには、ファイルにさらに大きなインデックス フィールドが必要になります。さらに、すべてのインデックス ページを **MicroKernel** エンジン キャッシュに収容できる場合、この手法では速度は向上しません。しかし、常にインデックス ページのほんの一部しかキャッシュ内にないのであれば、多くの余分なインデックス ページの書き込みが省かれます。この技術により、多くのレコードを持つファイルの構築に必要な時間を大幅に短縮することができます。**Create Index (31)** では、ファイル内のインデックス ページ数が多いほど、一度に 1 レコードずつ挿入する場合よりも高速にインデックスを作成できます。

つまり、**PSQL** ファイルを最初から読み込む場合、すべてのインデックス ページを保持するためのキャッシュ メモリが不足することを避けることが重要です。その場合は、**Create (14)** を使用してインデックスを付けずにファイルを作成し、すべてのデータレコードが挿入されたときに **Create Index (31)** を使用します。

これらのオペレーションの詳細については、『**Btrieve API Guide**』を参照してください。

論理レコード長の計算

Create オペレーション（14）に**論理レコード長**を適用する必要があります。論理レコード長は、ファイル内の**固定長データ**のバイト数です。この値を算出するには、各レコードの固定長部分に格納しなければならないデータのバイト数を計算します。

たとえば、以下の表には、どのように社員ファイル例のデータ バイトを合計して論理レコード長を算出するかを示しています。

フィールド	長さ（バイト単位）
ラスト ネーム	25
ファースト ネーム	25
ミドル イニシャル	1
社員 ID	4
電話番号	13
賃率	4
論理レコード長	72

論理レコード長を計算する際に、可変長データをカウントしないのは、可変長の情報がファイル内の固定長レコードから離れて（可変ページ上に）格納されるからです。

最大論理レコード長は、表 16 に定義されているようにファイル形式によって異なります。

表 16 ファイル形式ごとの最大論理レコード長

PSQL バージョン	例
9.5	ページ サイズ - 10 - 2（レコード オーバーヘッド）
8.x から 9.x	ページ サイズ - 8 - 2（レコード オーバーヘッド）
6.x から 7.x	ページ サイズ - 6 - 2（レコード オーバーヘッド）
v6.x より前	ページ サイズ - 6 - 2（レコード オーバーヘッド）
メモ： 上記の例に示したレコード オーバーヘッドは、圧縮を使用しない（可変長レコードではなく）固定長レコード用です。	

ページ サイズの選択

データ ファイル内のすべてのページは同じサイズです。したがって、ファイル内のページのサイズを決定するときは、以下の問題に答える必要があります。

- 無駄にされるバイトを減らすためにレコードの固定長部分を保持するデータ ページの最適なサイズはいくつでしょうか。
- インデックス ページに最長のキー定義を保持できる最小サイズはいくつでしょうか（たとえファイルにキーを定義しない場合でも、トランザクション一貫性保守機能を有効にした場合、MicroKernel エンジンにキーを追加します）。

以降では、これらの問題の答えを導き出す方法を説明します。得られた解答で、ファイルに最も合うページ サイズを選択できます。

ディスク領域を最小限にするための最適なページ サイズ

ファイルの最適なページ サイズを決定する前に、まずファイルの物理レコード長を計算する必要があります。物理レコード長は、論理レコード長とファイルのデータ ページ上にレコードを格納するのに必要なオーバーヘッドとの合計です（ページサイズの一般的な情報については、「[ページ サイズ](#)」を参照してください）。

MicroKernel エンジンは常に、すべてのレコードに最低 2 バイトのオーバーヘッド情報をそのレコードの使用量カウンタとして格納しています。また、MicroKernel エンジンは、ファイル内でのレコードとキーの定義方法により、各レコード内に追加バイトを格納します。

次の表は、レコードの圧縮を使用しない場合、ファイルの特性によってレコード オーバーヘッドが何バイト必要になるかを示します。

表 17 レコードの圧縮を使用しない場合のレコード オーバーヘッドのバイト数

ファイルの特性	ファイル形式			
	6.x	7.x	8.x	9.0 および 9.5
使用回数	2	2	2	2
重複キー（キーごとに）	8	8	8	8
可変ポインター（可変長レコード）	4	4	6	6
レコード長（VAT ¹ を使用する場合）	4	4	4	4
ブランク トランケーションの使用（VAT 不使用 / VAT 使用）	2/4	2/4	2/4	2/4
システム データ	NA ²	8	8	8
¹ VAT：可変長部割り当てテーブル ² NA：適用外				

次の表は、レコードの圧縮を使用する場合、ファイルの特性によってレコード オーバーヘッドが何バイト必要になるかを示します。

表 18 レコードの圧縮を使用した場合のレコード オーバーヘッドのバイト数

ファイルの特性	ファイル形式			
	6.x	7.x	8.x	9.0 および 9.5
使用回数	2	2	2	2
重複キー（キーごとに）	8	8	8	8
可変ポインター	4	4	6	6
レコード長（VAT ¹ を使用する場合）	4	4	4	4
レコード圧縮フラグ	1	1	1	1
システム データ	NA ²	8	8	8
¹ VAT：可変長部割り当てテーブル ² NA：適用外				

次の表は、ページのタイプによってページ オーバーヘッドが何バイト必要になるかを示します。

表 19 ページ オーバーヘッド（バイト数）

ページのタイプ	ファイル形式				
	6.x	7.x	8.x	9.0	9.5
データ	6	6	8	8	10
インデックス	12	12	14	14	16
可変	12	12	16	16	18

次の表は、ファイルのレコードおよびキーの定義方法に基づいて、物理レコード長を算出するため、論理レコード長に追加しなければならないオーバーヘッドのバイト数を示しています。レコードのオーバーヘッドの一覧は、表 17 および表 18 にもあります。

表 20 物理レコード長のワークシート

説明	例
1 論理レコード長を算定します。手順については、「 論理レコード長の計算 」を参照してください。 このワークシートのファイルの例では、72 バイトの論理レコードを使用します。可変長レコードを持つファイルの場合、論理レコード長はレコードの固定長部分のみを指します。	72
2 レコード使用カウントに 2 を加算します。 圧縮レコードのエントリでは、使用カウント、可変ポインター、レコード圧縮フラグを加算する必要があります。 6.x および 7.x：7 バイト（2 + 4 + 1） v8.x 以降：9 バイト（2 + 6 + 1）	72 + 2 = 74

表 20 物理レコード長のワークシート

説明	例
3 リンク重複キーごとに、8 を加算します。 重複キーのためのバイト数を算出する場合、MicroKernel エンジンでは作成時に繰り返し重複として定義されているキーに対して重複ポインター スペースを割り当てません。デフォルトでは、ファイル作成時に作成された重複を許すキーがリンク重複キーです。圧縮レコードのエントリでは、重複キーのポインター用に 9 を加算します。 ファイル例には、1 つのリンク重複キーがあります。	74 + 8 = 82
4 予約重複ポインターごとに、8 を加算します。ファイル例には予約重複ポインターはありません。	82 + 0 = 82
5 ファイルが可変長レコードを許可している場合、8x より前のファイルでは 4 を加算し、8x 以降のファイルでは 6 を加算します。 ファイル例では可変長レコードを許可していません。	82 + 0 = 82
6 ファイルが VAT を使用している場合、4 を加算します。 ファイルの例では、VAT を使用していません。	82 + 0 = 82
7 ファイルがブランク トランケーションを使用する場合は、以下のいずれかを加算します。 ◆ ファイルが VAT を使用しない場合、2 を加算します。 ◆ ファイルが VAT を使用する場合、4 を加算します。 ファイルの例では、VAT を使用していません。	82 + 0 = 82
8 ファイルでシステム キーの作成にシステム データを使用する場合は 8 を加算します。 ファイルの例では、System Data を使用していません。	82 + 0 = 82
物理レコード長	82

物理レコード長を使用して、データ ページに対するファイルの最適ページ サイズを決定できます。

MicroKernel エンジンではデータ レコードの固定長部分をデータ ページに格納しますが、固定長部分をページをまたいで分割することはしません。また、MicroKernel エンジンでは各データ ページにオーバーヘッド情報を格納します（表 17 および 18 を参照してください）。ページ サイズを決定するときは、この追加のオーバーヘッドを計算する必要があります。

選択したページ サイズからオーバーヘッド情報のバイト数を差し引いたものが物理レコード長の正確な倍数にならない場合、ファイルには未使用領域が含まれています。次の式を使用して、効率のよいページ サイズを見つけることができます。

未使用バイト = (ページ サイズ - 表 17 および表 18 ごとのデータ ページ オーバーヘッド) mod
(物理レコード長)

ファイルによるディスク領域の使用量を最適化するには、未使用領域を最小にしてレコードをバッファーに格納できるページ サイズを選択します。サポートされるページ サイズはファイル形式により異なります。表 21 を参照してください。内部レコード長（ユーザー データ + レコード オーバーヘッド）が小さく、ページ サイズが大きいと、無駄な領域がかなりの量になる可能性があります。

最適なページ サイズの例

物理レコード長が 194 バイトの例を考えてみましょう。以下の表に、ページ サイズごとに、ページに格納できるレコード数とページに残る未使用領域のバイト数を示します。

表 21 物理レコード長の例 : 194 バイト

適用可能なファイル形式	ページ サイズ	1 ページあたりのレコード数	未使用バイト	
v8.x より前	512	2	118	$(512 - 6) \bmod 194$
8.x から 9.0			116	$(512 - 8) \bmod 194$
v8.x より前	1,024	5	48	$(1,024 - 6) \bmod 194$
8.x から 9.0			46	$(1,024 - 8) \bmod 194$
9.5			44	$(1,024 - 10) \bmod 194$
v8.x より前	1,536	7	172	$(1,536 - 6) \bmod 194$
8.x から 9.0			172	$(1,536 - 6) \bmod 194$
v8.x より前	2,048	10	102	$(2,048 - 6) \bmod 194$
8.x から 9.0			100	$(2,048 - 8) \bmod 194$
9.5			98	$(2,048 - 10) \bmod 194$
v8.x より前	2,560	13	32	$(2,560 - 6) \bmod 194$
8.x から 9.0			32	$(2,560 - 6) \bmod 194$
v8.x より前	3,072	15	156	$(3,072 - 6) \bmod 194$
8.x から 9.0			156	$(3,072 - 6) \bmod 194$
v8.x より前	3,584	18	86	$(3,584 - 6) \bmod 194$
8.x から 9.0			86	$(3,584 - 6) \bmod 194$
v8.x より前	4,096	21	16	$(4,096 - 6) \bmod 194$
8.x から 9.0			14	$(4,096 - 8) \bmod 194$
9.5			12	$(4,096 - 10) \bmod 194$
9.0	8,192	42	36	$(8,192 - 8) \bmod 194$
9.5			34	$(8,192 - 10) \bmod 194$
9.5	16,384	84	78	$(16,384 - 10) \bmod 194$

計画として、ページおよびレコード オーバーヘッドの量は、今後のアップグレードによるファイル形式で増える可能性があることに注意してください。現在のファイル形式でページにきっちり収まるレコード サイズを考えた場合、そのサイズが将来のバージョンのファイル形式に合うためには、もっと大きなページ サイズが必要になるかもしれません。

また、レコードおよびオーバーヘッドが指定されたページ サイズ内に収まらない場合、データベース エンジンが自動的にページ サイズを更新します。たとえば、9.x のファイルに対しページ サイズ 4,096 を指定し、レコードおよびオーバーヘッドの要件が 4,632 バイトだと仮定します。この場合、データベース エンジンはページ サイズ 8,192 を使用します。

データベースの設計

表に示すように、ページ サイズ 512 を選択すると 1 ページに 2 つのレコードしか格納できず、ファイル形式によって各ページの 114 ～ 118 バイトが未使用になります。しかし、ページ サイズ 4,096 を選択すると、1 ページあたり 21 個のレコードを格納でき、各ページの 16 バイトだけが未使用になります。同じ 21 レコードでも、ページ サイズ 512 では 2 KB 以上の無駄な領域ができてしまいます。

物理レコード長が非常に小さい場合は、ほとんどのページ サイズで無駄になる領域は非常にわずかです。ただし、v8.x より前のファイルではページごとに最大 256 レコードという制限があります。この場合、物理レコード長が小さいのに大きなページ サイズ（たとえば、4,096 バイト）を選択すると、無駄な領域が大きくなります。例として、表 22 に 8.x より前のファイルバージョンに対して 14 バイトのレコード長を設定した場合の動作を示します。

表 22 8.x より前のファイル バージョンの例：レコード長 14 バイト

ページ サイズ	1 ページあたりのレコード数	未使用バイト	
512	36	2	$(512 - 6) \bmod 14$
1,024	72	10	$(1,024 - 6) \bmod 14$
1,536	109	4	$(1,536 - 6) \bmod 14$
2,048	145	12	$(2,048 - 6) \bmod 14$
2,560	182	6	$(2,560 - 6) \bmod 14$
3,072	219	0	$(3,072 - 6) \bmod 14$
3,584	255	8	$(3,584 - 6) \bmod 14$
4,096	256	506	$(4,096 - 6) \bmod 14$

最大ページ サイズ

選択するページ サイズは、8 つのキー値にオーバーヘッドを加えた値を保持できるほどの大きさが必要です。ファイルに対して許容できる最小ページ サイズを見つけるには、表 23 で指定された値を追加します。

表では例として 9.5 ファイル形式を使用します。

表 23 最小ページ サイズ ワークシート

説明		例
1	バイト単位でファイル内の最大キーのサイズを決定します（社員ファイル例では、最大キーは 25 バイトです）。 一意のキーを定義していないファイルでは、システムで定義された、システム データとも呼ぶログ キーが最大キーである場合があります。そのサイズは 8 バイトです。	25
2	以下のうちの 1 つを追加します。 ◆ 重複を許さないキーまたは繰り返し重複を使用するキーの場合は、8 バイトを加算します。 ◆ リンク重複を使用するキーには 12 バイトを加算します（この例ではリンク重複を使用します）。	$25 + 12 = 37$
3	結果に 8 を掛けます（MicroKernel エンジンでは、1 ページに 8 個以上のキーのスペースが必要です）。	$37 * 8 = 296$
4	ファイル形式のインデックス ページ オーバーヘッドを追加します。 表 19 でインデックス ページの項目を参照してください。	$296 + 16 = 312$
最小ページ サイズ		312 バイト

計算結果以上の有効なページ サイズを選択します。ここで、選択するページ サイズがファイル作成後に作成されたキーのサイズを収容できるサイズでなければならないことに注意してください。キー セグメントの総数は、最小ページ サイズを指示する場合があります。たとえば、ページ サイズ 512 を使用するファイルには 8 個のキー セグメントしか定義できません。

表 24 最小ページ サイズ ワークシート

ページ サイズ	ファイル バージョンによるキー セグメント数			
	6.x および 7.x	8.x	9.0	9.5
512	8	8	8	N/A
1,024	23	23	23	97
1,536	24	24	24	N/A
2,048	54	54	54	97
2,560	54	54	54	N/A
3,072	54	54	54	N/A
3,584	54	54	54	N/A
4,096	119	119	119	204
8,192	N/A	N/A	119	420
16,384	N/A	N/A	N/A	420
N/A は「適用外」を意味します。				

ファイル サイズの予測

ページ数を予測できることから、ファイルを格納するのに必要なバイト数を見積もることが可能です。ただし、MicroKernel エンジンでは動的にページを操作するため、式を用いる場合、これらの式はファイル サイズを概算するだけであることを考慮してください。



メモ 以下の説明とファイル サイズを決定するための式は、データ圧縮を使用するファイルには適用されません。このようなファイルのレコード長は、各レコードに含まれる繰り返し文字の数によって異なるからです。

これらの式は、必要な最大記憶域に基づいていますが、一度に 1 つのタスクでしか、ファイルに対してレコードの更新や挿入を行わないことを前提としています。同時並行トランザクション中に、複数のタスクでファイルに対してレコードを更新したり挿入したりする場合には、ファイル サイズが増加します。

またこれらの式は、ファイルからまだレコードが削除されていないことを前提としています。ファイル内のレコードをいくつか削除しても、ファイルのサイズは変わりません。MicroKernel エンジンでは、削除されたレコードが占有していたページの割り当てを解除しません。むしろ、MicroKernel エンジンでは新しいレコードがファイルに挿入されるたびにそれらのページを再利用し、その後で新しいページを割り当てます。

計算の最終結果に小数値が含まれている場合は、小数値を次に大きい整数に切り上げます。

式および派生手順

次の式は、ファイルの格納に必要な最大バイト数を算出するために使用します。[手順 x を参照] は、この式の個々の要素を説明する手順への参照を示します。

ファイル サイズ (バイト単位) =

$$\begin{aligned} & (\text{ページ サイズ} * \\ & \quad (\text{データ ページ数 [手順 1 を参照]} + \\ & \quad \text{インデックス ページ数 [手順 2 を参照]} + \\ & \quad \text{可変ページ数 [手順 3 を参照]} + \\ & \quad \text{その他のページ数 [手順 4 を参照]} + \\ & \quad \text{シャドウ プール ページ数 [手順 5 を参照]})) \\ & + (\text{特殊ページ サイズ [手順 6 を参照]} * \\ & \quad (\text{PAT ページ数 [手順 7 を参照]} + \\ & \quad \text{FCR ページ数} + \text{予約済みページ数 [手順 8 を参照]})) \end{aligned}$$

ファイル サイズの算出には 2 種類のページ カテゴリを含める必要があります。標準ページ カテゴリには、データ ファイルが最初に作成されたときのページが含まれます (『Btrieve API Guide』の「[Create \(14\)](#)」も参照してください)。さらに、式には表 25 で示された特殊 (非標準) ページも含める必要があります。特殊ページはファイルのページサイズの倍数になるとは限りません。

1 以下の式を使用して、データ ページの数を算出します。

$$\begin{aligned} \text{データ ページ数} = \\ \#r / \\ ((PS - DPO) / PRL) \end{aligned}$$

この場合、

- #r : レコード数
- PS : ページサイズ
- DPO : データ ページ オーバーヘッド (表 17 および表 18 を参照)
- PRL : 物理レコード長 (表 20 を参照)

- 2 以下の式のうちの1つを使用して、定義されたキーごとにインデックス ページ数を計算します。

重複を許さないインデックス、または繰り返し重複キーを許すキーごとに、

$$\text{インデックス ページ数} = \left(\#r / \left((PS - IPO) / (KL + 8) \right) \right) * 2$$

この場合、

- #r : レコード数
- PS : ページ サイズ
- IPO : インデックス ページ オーバーヘッド
- KL : キー長

リンク重複キーを許すインデックスごとに、

$$\text{インデックス ページ数} = \left(\#UKV / \left((PS - IPO) / (KL + 12) \right) \right) * 2$$

この場合、

- #UKV : 一意のキー値の数
- PS : ページ サイズ
- IPO : インデックス ページ オーバーヘッド
- KL : キー長

B ツリー インデックス構造は、インデックス ページの使用率 50% 以上を保証します。したがって、インデックス ページの計算では、必要なインデックス ページの最小数に 2 をかけた値が最大サイズになります。

- 3 ファイルに可変長レコードが含まれている場合は、以下の式を使用して、可変ページの数を出します。

$$\text{可変ページ数} = (AVL * \#r) / (1 - (FST + (VPO/PS)))$$

この場合、

- AVL : 標準的なレコードの可変部分の平均の長さ
- #r : レコード数
- FST: ファイルが作成されときに指定された空きスペース スレッシュホールド (『Btrieve API Guide』の「[Create \(14\)](#)」も参照)
- VPO : 可変ページ オーバーヘッド (表 19 を参照)
- PS : ページ サイズ



メモ 可変長部分が同じページに収まるレコードの平均数を予測することは困難であるため、非常に大雑把な可変ページ数の予測しか得られません。

- 4 その他の標準ページ数を算出します。

- 使用するオルタネート コーディング シーケンスごとに 1 ページ
- ファイルに RI の制約がある場合は参照整合性 (RI) ページに 1 ページ

手順 1、2、3、および 4 の合計は、ファイルに格納される論理ページの概算総数を表します。

- 5 シャドウ ページ プールの概算ページ数を算出します。データベース エンジンに、シャドウ ページングにプールを使用します。以下の式を使用して、プール内のページ数を見積もります。

$$\text{シャドウ ページ プールのサイズ} = (\text{キー数} + 1) * (\text{挿入、更新および削除の数の平均数}) * (\text{並行 トランザクション数})$$

データベースの設計

この式は、タスクがトランザクションの外部で Insert、Update および Delete オペレーションを実行する場合に適用されます。タスクがトランザクション内でこれらのオペレーションを実行している場合は、トランザクション内で予想される Insert、Update、および Delete オペレーションの平均数に、式で算定されたトランザクション外部の値をかけます。タスクが同時並行トランザクションを実行している場合は、未使用ページのプールの予測サイズをさらに大きくする必要があります。

- 6 表 25 でファイルのバージョンおよびデータ ページ サイズを参照し、特殊ページ サイズを調べます。ファイル形式バージョンによって、FCR、予約済み、および PAT ページのページ サイズは、通常のデータ、インデックス、および可変ページ用のページ サイズとは異なっています。

表 25 ファイル形式ごとの特殊ページのページ サイズ

標準 ページ サイズ	ファイル形式 v6.x および 7.x		ファイル形式 8.x		ファイル形式 9.0 から 9.4		ファイル形式 9.5	
	特殊ページ サイズ	PAT ページ エントリ	特殊ページ サイズ	PAT ページ エントリ	特殊ページ サイズ	PAT ページ エントリ	特殊ページ サイズ	PAT ページ エントリ
512	512	N/A	2,048	320	2,048	320	N/A	N/A
1,024	1,024	N/A	2,048	320	2,048	320	4,096	480
1,536	1,536	N/A	3,072	480	3,072	480	N/A	N/A
2,048	2,048	N/A	4,096	640	4,096	640	4,096	480
2,560	2,560	N/A	5,120	800	5,120	800	N/A	N/A
3,072	3,072	N/A	6,144	960	6,144	960	N/A	N/A
3,584	3,584	N/A	7,168	1,120	7,168	1,120	N/A	N/A
4,096	4,096	N/A	8,192	1,280	8,192	1,280	8,192	1,280
8,192	N/A	N/A	N/A	N/A	N/A	N/A	16,384	16,000
16,384	N/A	N/A	N/A	N/A	N/A	N/A	16,384	16,000
N/A は「適用外」を意味します。								

- 7 ページアロケーション テーブル (PAT) ページ数を算出します (「[ページプリアロケーション](#)」も参照)。

各ファイルには少なくとも 2 ページの PAT ページがあります。ファイル内の PAT ページ数を算出するには、以下の式のうちの 1 つを使用します。

8.x より前のファイル形式の場合：

$$\text{PAT ページ数} = \frac{((\text{手順 1} \sim \text{3 のページの合計}) * 4) / (\text{ページ サイズ} - \text{オーバーヘッドの 8 バイト})}{2}$$

8.x 以降のファイル形式の場合：

$$\text{PAT ページ数} = \frac{2 * (\text{手順 1} \sim \text{3 のページの合計})}{\text{PAT エントリ数}}$$

PAT エントリ数については、表 25 でファイルバージョンとデータ ページ サイズを見てください。

- 8 ファイルコントロールレコード (FCR) ページ用に 2 ページを含めます (「[ファイルコントロールレコード \(FCR\)](#)」も参照)。8.x 以降のファイル形式を使用している場合は、予約済みページ用の 2 ページも含めます。

データベースの最適化

MicroKernel エンジンには、ディスク領域の節約とシステム パフォーマンスの向上を実現するいくつかの機能があります。これらの機能は以下のとおりです。

- 「重複キー」
- 「ページ プリアロケーション」
- 「ブランク トランケーション」
- 「レコード圧縮」
- 「インデックス バランス」
- 「可変長部割り当てテーブル」
- 「キーオンリー ファイル」

重複キー

キーを重複可能と定義すれば、MicroKernel エンジンは複数のレコードにそのキーの同じ値を持たせることができます。それ以外は、各レコードにそのキーの一意の値がなければなりません。セグメント化されたキーの 1 つのセグメントが重複可能であれば、すべてのセグメントが重複可能でなければなりません。

リンク重複キー

デフォルトでは、MicroKernel エンジンは v7.0 以降のファイルに重複キーを**リンク重複キー**として格納します。重複キー値を持つ最初のレコードがファイルに挿入されると、MicroKernel エンジンはインデックス ページにキー値を格納します。MicroKernel エンジンはまた、このキー値で最初のレコードと最後のレコードを識別するために 2 つのポインターを初期化します。さらに、MicroKernel エンジンはレコードの終わりにあるポインターのペアをデータ ページに格納します。これらのポインターは、同じキー値で前のレコードと次のレコードを識別します。ファイルを作成する場合、将来リンク重複キーを作成する際に使用するポインターを予約することができます。

データ ファイルを作成した後に重複キーの追加を予測し、リンク重複方法を使用するためのキーが必要であれば、ファイル内にポインターのための領域をプリアロケートすることができます。

繰り返し重複キー

リンク重複キーを作成するための領域がない場合、つまり、重複ポインターがない場合、MicroKernel エンジンは**繰り返し重複キー**を作成します。MicroKernel エンジンは、データ ページ上とインデックス ページ上に繰り返し重複キーのすべてのキー値を格納します。つまり、キーの値はデータ ページ上のレコード内に存在し、インデックス ページ上のキー エントリで反復されます。



メモ 6.0 より前の Btrieve のユーザーの場合、**リンク重複キー**は**永続インデックス**に対応し、**繰り返し重複キー**は**補足インデックス**に対応します。

Create (14) オペレーションまたは Create Index (31) オペレーションのキーの仕様ブロックにキー フラグのビット 7 (0x80) を設定することによって、キーを繰り返し重複キーとして定義できます。6.10 より前には、キーを繰り返し重複キーと定義できず、キー フラグのビット 7 はユーザーが定義できませんでした。6.0 以降では、リンク重複キーを作成するための領域がない場合、したがって、MicroKernel エンジンが繰り返し重複キーとしてキーを作成しなければならない場合、Stat オペレーション (15) がビット 7 を設定します。

5.x 形式を使用するファイルは、5.x では**補足キー属性**と呼ばれているこの同じキー フラグを使用して、キーが 5.x Create Supplemental Index オペレーション (31) で作成されたことを示します。



メモ 6.0 より前のファイルでは、補足インデックスだけが削除できます。永続インデックスは、その名前の示すとおり削除できません。6.0 以降のファイルでは、すべてのインデックスを削除できます。

リンクと繰り返し

各方法にはパフォーマンスの利点があります。

- リンク重複キーの方が検索がより速いのは、MicroKernel エンジンで読み取るページが少ないからです。
- 繰り返し重複キーを使用する方が、複数のユーザーが並行して同じページにアクセスする場合のインデックスページの競合は少なくなります。

リンク重複キーと繰り返し重複キーの間には、パフォーマンスのトレードオフがあります。一般に、キーの平均重複数が 2 つ以上であれば、リンク重複キーがディスク上で占有する領域は少なくなり、また、インデックスページが少ないために一般には検索が高速になります。ただし、重複キーを持つレコードがほとんどファイルに格納されておらず、キー長が非常に短い場合は、その逆になります。それは、リンク重複ツリー内のエントリがポインターに 8 バイトを必要とするのに対して、繰り返し重複キー エントリは 4 バイトを必要とするからです。

重複を持つキーがわずかしかなければ、繰り返し重複キーを使用して各データレコードに 8 バイトを追加保存するのが有利です。キー重複の平均数が 2 未満である場合は、いずれを選択してもパフォーマンス上の利点はそれほどありません。

いくつかの並行トランザクションが同時に同じファイル上でアクティブになると予想する場合、繰り返し重複キーの方が、これらのトランザクションが同じページにアクセスしない確率は高くなります。並行トランザクションでの書き込みに関連するすべてのページには、それらのページに対する暗黙ロックがあります。ある並行トランザクションで変更を行うためにページが必要であり、そのページが別の並行トランザクションに関与している場合、MicroKernel エンジンでは別のトランザクションが終了するまで待ちます。このような暗黙の待ち時間が頻繁に発生する場合、アプリケーションのパフォーマンスは低下します。

いずれのキー格納方法も、年代順をトラッキングするための簡便法としては推奨しません。リンク重複キーの場合、MicroKernel エンジンではキーを作成した後に挿入されるレコードの年代順を維持しますが、キーのインデックスを作成し直した場合、年代順は失われます。繰り返し重複キーについては、キーを作成してから新しいレコードを挿入するまでの間にレコードの削除がなかった場合のみ、MicroKernel エンジンではレコードの年代順を維持します。年代順をトラッキングするには、キー上で AUTOINCREMENT データ型を使用します。

ページ プリアロケーション

プリアロケーションは、MicroKernel エンジンがディスク領域を必要とするときに、その領域が利用可能であることを保証します。MicroKernel エンジンでは、データ ファイルを作成するときにファイルに最大 65,535 ページをプリアロケートできます。表 26 は、65,535 全ページ分のディスク領域を割り当てるものと仮定して、MicroKernel エンジンが各ページ サイズのファイルに対して割り当てる最大バイト数を示したものです。

表 26 ページ サイズごとのディスク領域割り当て

ページ サイズ	割り当てられたディスク領域 ¹
512	33,553,920
1,024	67,107,840
1,536	100,661,760
2,048	134,215,680
2,560	167,769,600
3,072	201,323,520
3,584	243,877,440
4,096	268,431,360
8,192	536,862,720
16,384	1,073,725,440
¹ 値は、ページ サイズに 65,535 をかけたものです。	

指定したページ数をブリアロケートするだけの十分な領域がディスクにない場合、MicroKernel エンジン はステータス コード 18（ディスクがいっぱい）を返し、ファイルを作成しません。

データ ファイルがディスク上の**連続する**領域を占有する場合、ファイル操作を高速化することができます。速度の向上は、非常に大きなファイルで最も顕著です。ファイルに連続するディスク領域をブリアロケートする場合、ファイルを作成するためのデバイスには必要なバイト数の連続する使用可能空き領域が必要です。MicroKernel エンジンは、ディスク上の領域が連続であるかどうかにかかわらず、指定するページ数をブリアロケートします。

ファイルに必要なデータ ページ数とインデックス ページ数を決定するには、この章の前半で説明した式を使用します。この部分の計算から出た剰余を次に大きい整数に切上げます。

ファイルのページをブリアロケートすると、そのファイルは実際にディスクのその領域を占有します。ほかのデータ ファイルは、そのファイルを削除または交換するまで、ブリアロケートされたディスク領域を使用できません。

レコードを挿入すると、MicroKernel エンジンはデータとインデックスのブリアロケートされた領域を使用します。ファイルにブリアロケートされたすべての領域が使用されている場合、MicroKernel エンジンは新しいレコードが挿入されるたびにファイルを拡張します。

API の Stat オペレーション（15）を発行すると、MicroKernel エンジンはファイルの作成時に割り当てたページ数と MicroKernel エンジンが現在使用しているページ数の差を返します。この差が常に、ブリアロケーションに指定したページ数より小さくなるのは、たとえレコードを何も挿入していなくても、ファイルの作成時に一定のページ数が使用されると MicroKernel エンジンが見なすからです。

ファイル ページは一度使用されたら、たとえそのページに格納されているすべてのレコードを削除しても、ページは使用中のままになります。Stat オペレーションが返す未使用ページ数は増えません。レコードを削除すると、MicroKernel エンジンはファイル内の空き領域のリストを保守し、新しいレコードを挿入したときに使用可能な領域を再利用します。

たとえ Stat オペレーションが返す未使用ページ数が 0 であっても、ファイルにはまだ使用可能な空き領域があります。以下のいずれかが真であれば、未使用ページ数が 0 である可能性があります。

- ファイルにはページをブリアロケートしなかった。
- ブリアロケートしたすべてのページは、ある時点で使用中であった。

ブランク トランケーション

空白を切り捨てることにした場合、MicroKernel エンジンではファイルにレコードを書き込むときにレコードの可変長部分の末尾の空白（ASCII のスペース コード 32、つまり 16 進数の 0x20）を格納しません。ブランク トランケーションは、レコードの固定長部分に影響しません。MicroKernel エンジンでは、データに埋め込まれている空白を削除しません。

切り捨てられた末尾の空白を含むレコードを読み取ると、MicroKernel エンジンではレコードを元の長さまで拡張します。MicroKernel エンジンがデータ バッファ長パラメーターで返す値には、拡張された空白が含まれています。ブランク トランケーションは、レコードの物理サイズに 2 バイトまたは 4 バイトのオーバーヘッドを追加し、固定長部分と一緒に格納します。ファイルが VAT を使用しない場合は 2、使用する場合 4 です。

レコード圧縮

ファイルを作成する場合、MicroKernel エンジンがファイルにデータ レコードを格納するときにデータ レコードを圧縮するかどうかを指定できます。レコード圧縮により、多数の繰り返し文字を含むレコードの格納に必要なスペースを大幅に削減できます。MicroKernel エンジンでは、5 つ以上の同じ連続文字を 5 バイトに圧縮します。

以下の環境におけるレコード圧縮の使用方法について考えてみましょう。

- 圧縮するレコードは、圧縮を使用することの利点が最大になるように構造化されます。
- ディスク利用度を高める必要性は、処理量の増大や圧縮されたファイルに必要なディスク アクセス時間より重要です。
- MicroKernel エンジンを実行しているコンピューターは、MicroKernel エンジンが圧縮バッファに使用する追加メモリを提供できます。



メモ データベース エンジンでは、システム データを使用しており、レコード長が許容最大サイズを超えるファイルについては自動的にレコード圧縮を使用します。表 12 を参照してください。

圧縮されたファイルに対してレコード I/O を実行する場合、MicroKernel エンジンでは圧縮バッファを利用して、レコードの圧縮および拡張処理用のメモリ ブロックを提供します。レコードの圧縮や拡張を行うのに十分なメモリを確保するため、MicroKernel エンジンでは、タスクが圧縮されたファイルに挿入する最長レコードの 2 倍の長さを格納できるだけのバッファ領域を必要とします。この要求は、MicroKernel エンジンがロードされた後にコンピューター内に残っている空きメモリ量に影響を与える可能性があります。たとえば、タスクが書き込む取得する最長レコードが 64 KB の長さであれば、MicroKernel エンジンはそのレコードの圧縮および拡張に 128 KB のメモリを必要とします。



メモ ファイルが VAT を使用している場合、MicroKernel エンジンが必要とするバッファ領域はファイルのページサイズの 16 倍です。たとえば、4 KB のレコードでは、レコードの圧縮と拡張に 64 KB のメモリが必要になります。

圧縮されたレコードの最終の長さはそのレコードがファイルに書き込まれるまで決定できないので、MicroKernel エンジンでは常に圧縮されたファイルを可変長レコード ファイルとして作成します。データ ページでは、MicroKernel エンジンでは重複キー ポインターごとに、ファイルが VAT を使用しない場合は 7 バイト、使用する場合は 11 バイトを追加し、さらに、重複キー ポインターごとに 8 バイトを格納します。MicroKernel エンジンでは次に、レコードを可変ページに格納します。レコードの圧縮されたイメージは可変長レコードとして格納されるので、タスクが頻繁な挿入、更新および削除を行う場合は、個々のレコードがいくつかのファイル ページにわたって断片化される場合があります。MicroKernel エンジンでは 1 つのレコードを取得するために複数のファイル ページを読み取らなければならない場合があるので、この断片化によってアクセス時間が遅くなるおそれがあります。

レコード圧縮オプションは、各レコードが多数の繰り返し文字を取り込む可能性がある場合に最も有効です。たとえば、レコードにいくつかのフィールドが含まれており、レコードをファイルに挿入するときにそれらのフィールドはすべてタスクによって空白に初期化される可能性があります。圧縮は、これらのフィールドがほかの値を含むフィールドによって分離される場合でなく、レコード内で 1 つにグループ化される場合に有効です。

レコード圧縮を使用するには、圧縮フラグを設定してファイルを作成しておく必要があります。キーオンリーファイルでは圧縮を行えません。

インデックス バランス

MicroKernel エンジンでは、**インデックス バランス**を使用することによってディスクをさらに節約できます。MicroKernel エンジンは、デフォルトではインデックス バランスを使用しないので、現在のインデックス ページがいっぱいになるたびに、MicroKernel エンジンは新しいインデックス ページを作成する必要があります。インデックス バランスが有効であれば、MicroKernel エンジンは現在のインデックス ページがいっぱいになるたびに、新しいインデックス ページを頻繁に作成しないようにすることができます。インデックス バランスを使用すると、MicroKernel エンジンは隣接するインデックス ページで使用可能な領域を探します。これらのページのうちの 1 つに領域がある場合、MicroKernel エンジンはいっぱいのインデックス ページから空き領域を持つページへキーを移動します。

インデックス バランス処理を実行すると、インデックス ページ数が少なくなるだけでなく、より密度の高いインデックスが作成され、ディスクの総利用率も上がり、大半の読み取り操作に対する応答が速くなります。ソート順でファイルにキーを追加する場合、インデックス バランスを使用していると、インデックス ページの利用率が 50% から 100% 近くまで増加します。ランダムにキーを追加する場合、最小のインデックス ページの利用率が 50% から 66% に増加します。

Insert オペレーションと Update オペレーションでは、バランス ロジックはファイル内のより多くのページを調べるように MicroKernel エンジンに要求し、より多くのディスク I/O を要求する可能性があります。余分なディスク I/O はファイル更新速度を下げます。インデックス バランスの正確な影響は状況によって異なりますが、インデックス バランスを使用した場合、書き込み操作のパフォーマンスは概して約 5 ~ 10% 低下します。

MicroKernel エンジンは 2 つのレベル、つまり、エンジン レベルとファイル レベルのインデックス バランスを提供するので、MicroKernel エンジン環境を微調整することができます。セットアップ中にインデックス バランス 設定オプションを指定すると、MicroKernel エンジンはすべてのファイルにインデックス バランスを適用します。インデックス バランス設定オプションの指定方法については、『PSQL User's Guide』を参照してください。

特定のファイルだけにインデックス バランスを行うように指定することもできます。そうするには、ファイル作成時にファイル フラグのビット 5 (0x20) を設定します。MicroKernel エンジンを起動したときにインデックス バランス設定オプションがオフであれば、MicroKernel エンジンはビット 5 のファイル フラグを設定したファイル上のインデックスだけにインデックス バランスを適用します。

MicroKernel エンジンを起動したときにインデックス バランス設定オプションがオンであった場合、MicroKernel エンジンはすべてのファイルのファイル フラグ内のビット 5 を無視します。この場合、MicroKernel エンジンはすべてのファイルにインデックス バランスを適用します。

ファイルは、インデックス バランスが有効であるかどうかに関係なく互換性があります。また、インデックス バランスが使用されたインデックス ページを含むファイルにアクセスするために、インデックス バランスを指定する必要はありません。MicroKernel エンジンのインデックス バランス オプションをオンにした場合、既存のファイル内のインデックス ページはいっぱいになるまで影響を受けません。MicroKernel エンジンは、このオプションを有効にした結果として既存のファイルに対し再度インデックス バランスを行いません。同様に、インデックス バランス オプションをオフにしても、既存のインデックスは影響を受けません。このオプションのオン、オフにより、MicroKernel エンジンがいっぱいになったインデックス ページを処理する方法が決まります。

可変長部割り当てテーブル

可変長部割り当てテーブルを使用すると、MicroKernel エンジンは非常に大きなレコード内に大きなオフセットで存在するデータに対してさらに高速にアクセスでき、データ圧縮を使用するファイル内のレコードを処理するときに MicroKernel エンジンが要求するバッファ サイズが大幅に削減されます。レコードの VAT を使用すると、MicroKernel エンジンはレコードの可変長部分をさらに小さな部分に分割し、次にこれらの部分をいくつかの可変長部に格納します。MicroKernel エンジンは、同量のレコード データを、レコードの可変長部に格納します。ただし、最後の可変長部は異なることがあります。MicroKernel エンジンは、ファイルのページ サイズに 8 をかけて、各可変長部に格納する量を計算します。最後の可変長部には、MicroKernel エンジンがほかの可変長部にデータを分割した後に残るデータが含まれています。



メモ 可変長部の長さを算出する式（ページサイズの 8 倍）は、MicroKernel エンジンの将来のバージョンでは変更される可能性があります。

VAT を使用し、4,096 バイトのページサイズを持つファイルでは、第 1 の可変長部にレコードの可変部分のオフセット 0 ～ 32,767 のバイトが格納され、第 2 の可変長部にオフセット 32,768 ～ 65,535 が格納され、以降同様に格納されます。MicroKernel エンジンが VAT を使用してレコード内の大きなオフセットまでのシークを加速できるのは、VAT を使用すると、レコードの下位オフセットのバイトを含んでいる可変長部をスキップできるからです。

アプリケーションでは、ファイルの作成時に VAT を使用するようにするかどうかを指定します。アプリケーションが非常に大きな、物理ページサイズの 8 倍を越えるレコードに対して **Chunk** オペレーションを使用し、順次でないランダムな方法でチャンクにアクセスする場合は、VAT によってアプリケーションのパフォーマンスが向上する可能性があります。アプリケーションがレコード全体に対する操作を行う場合は、VAT によってパフォーマンスは向上しません。この場合、MicroKernel エンジンはレコードを順次に読み書きし、MicroKernel エンジンはレコード内のどのバイトもスキップしないからです。

アプリケーションが **Chunk** オペレーションを使用するがレコードに順次アクセスする場合（たとえば、レコードの最初の 32 KB を読み、次の 32 KB を読み、という具合にレコードの最後まで読む）、VAT はパフォーマンスを向上させません。これは、MicroKernel エンジンがオペレーション間でレコード内の位置を保存することによって **Chunk** オペレーションが先頭からシークする必要性をなくしているためです。

VAT には別の利点もあります。MicroKernel エンジンが圧縮されたレコードを読み書きする場合、使用するバッファ サイズは、レコードの未圧縮サイズの最大 2 倍必要です。ファイルに VAT がある場合、そのバッファは 2 つの可変長部と同じ大きさ、つまり、物理ページサイズの 16 倍あれば済みます。

キーオンリー ファイル

キーオンリー ファイルでは、レコード全体がキーと共に格納されるため、データ ページは不要です。キーオンリー ファイルは、レコードに単一のキーが含まれており、かつそのキーがレコードの大部分を占有している場合に有効です。キーオンリー ファイルのもう 1 つの一般的な用途は、標準ファイルの外部インデックスとしての使用です。

キーオンリー ファイルには以下の制限が適用されます。

- 各ファイルには 1 つのキーしか含められません。
- 定義できる最大レコード長は 253 バイトです。
- キーオンリー ファイルではデータ圧縮を行えません。
- **Step** オペレーションは、キーオンリー ファイルでは機能しません。
- キーオンリー ファイルのレコードで **Get Position** を実行することはできますが、その位置は、レコードが更新されると変わります。

キーオンリー ファイルには、後ろに多数の PAT ページとインデックス ページが付いたファイル コントロール レコード ページしか含まれていません。キーオンリー ファイルに ACS がある場合、ACS ページもある可能性があります。ODBC を使用してファイルに参照整合性制約を定義した場合、ファイルには 1 つまたは複数の可変ページも含まれます。

セキュリティの設定

MicroKernel エンジンには、ファイル セキュリティの設定方法が 3 つあります。

- ファイルへのオーナー ネームの割り当て
- 排他モードでのファイルのオープン
- PSQL Control Center (PCC) のセキュリティ設定を使用

また、MicroKernel エンジンにはサーバー プラットフォーム上でネイティブ ファイル レベルのセキュリティが使用可能であればそれをサポートします。



メモ Windows 開発者: NTFS ファイル システムをサーバーにインストールすると、サーバー上でファイル レベル セキュリティを使用できます。FAT ファイル システムをインストールした場合は、ファイル システム セキュリティは使用できません。

MicroKernel エンジンには、データ セキュリティを向上させる以下の機能があります。

オーナー ネーム

MicroKernel エンジンでは、**Set Owner (29)** オペレーションを使用してオーナー ネームを割り当てることにより、ファイルに対するアクセスを制限することができます (『Btrieve API Guide』の「[Set Owner \(29\)](#)」を参照してください)。ファイルにオーナー ネームを割り当てると、MicroKernel エンジンはそのファイルに対するアクセスについてオーナー ネームを要求します。このため、オーナー ネームを指定しないユーザーまたはアプリケーションがファイルの内容に対して無許可でアクセスしたり変更することはできません。

同様に、ファイルに割り当てられているオーナー ネームがわかれば、ファイルからオーナー ネームをクリアできます。

オーナー ネームでは大文字と小文字が区別されます。また、短いものと長いものがあります。短いオーナー ネームは半角 8 文字までの範囲で指定できます。長いオーナー ネームは半角 24 文字までの範囲で指定できます。長いオーナー ネームに関する制限事項については、『Btrieve API Guide』の「[Set Owner \(29\)](#)」で「手順」を参照してください。

以下の方法でファイルに対するアクセスを制限できます。

- ユーザーはオーナー ネームを指定せずに読み取り専用アクセスを行うことができます。ただし、ユーザーまたはタスクはオーナー ネームを指定しないとファイルの内容を変更できません。そうしようとすると、MicroKernel エンジンエラーを返します。
- すべてのアクセス モードでユーザーにオーナー ネームを指定するように要求することができます。正しいオーナー ネームを指定しないと、MicroKernel エンジンがファイルに対するすべてのアクセスを制限します。

オーナー ネームを割り当てる場合、データベース エンジンがオーナー ネームを暗号化キーとしてディスク ファイル内のデータを暗号化するように要求することもできます。ディスク上のデータを暗号化すると、無許可のユーザーはデバッガーまたはファイル ダンプ ユーティリティを使用してデータを調べることができません。**Set Owner** オペレーションを使用し、暗号化を指定すると、ただちに暗号化が行われます。ファイル全体が暗号化されるまでは MicroKernel エンジンの制御下にあります。また、ファイル サイズが大きいほど、暗号化処理にかかる時間は長くなります。暗号化にはさらに処理時間が必要なため、データ セキュリティが重要である場合に限りこのオプションを選択する必要があります。

ファイルに割り当てられているオーナー ネームがわかれば、「[Clear Owner \(30\)](#)」オペレーションを使用してファイルから所有権の制限を削除できます。また、暗号化されたファイルで **Clear Owner** オペレーションを使用すると、データベース エンジンはそのファイルを復号化します。

排他モード

ファイルへのアクセスを 1 クライアントに制限するために、MicroKernel エンジンが排他モードでファイルを開くように指定できます。クライアントが排他モードでファイルを開くと、排他モードでファイルを開いたクライアントがそのファイルを閉じるまでほかのクライアントはファイルを開けません。

SQL セキュリティ

データベース URI (Uniform Resource Indicator) 文字列の詳細については、「[データベース URI](#)」を参照してください。PCC のセキュリティ設定へのアクセス方法は、『PSQL User Guide』を参照してください。

言語インターフェイス モジュール

6

この章では、PSQL SDK インストール オプションに用意されている言語インターフェイス ソース モジュールを示します。

各言語インターフェイス用のソース コードを提供しています。追加情報は、ソース モジュール自体にあります。

[Actian PSQL Web サイト](#)の開発者向けオンライン リソースをご覧くださいと、さまざまな言語インターフェイス用の記事やサンプル コードが盛り込まれています。

特定のインターフェイス モジュールの詳細については、以下をお読みください。

- 「[インターフェイス モジュールの概要](#)」
- 「[C/C++](#)」
- 「[Delphi](#)」
- 「[DOS \(Btrieve\)](#)」
- 「[Pascal](#)」
- 「[Visual Basic](#)」

インターフェイス モジュールの概要

プログラミング言語にインターフェイスがない場合は、コンパイラが複数の言語からの呼び出しが混在した状態をサポートするかどうかを確認してください。そうであれば、C インターフェイスを使用できる場合があります。

表 27 Btrieve 言語インターフェイス ソース モジュール

言語	コンパイラ	ソース モジュール
C/C++	<ul style="list-style-type: none"> ◆ Embarcadero、Microsoft、WATCOM などの大半の C/C++ コンパイラ。 このインターフェイスには、マルチプラットフォーム サポートがあります。 ◆ Embarcadero C++ Builder 	<ul style="list-style-type: none"> ◆ BlobHdr.h (Embarcadero または Phar Lap のみを使用する拡張 DOS プラットフォーム) ◆ BtiTypes.h (プラットフォーム依存データ型) ◆ BtrApi.h (Btrieve 関数プロトタイプ) ◆ BtrApi.c (すべてのプラットフォーム用の MicroKernel エンジン コード) ◆ BtrConst.h (共通 Btrieve 定数) ◆ BtrSamp.c (サンプル プログラム) ◆ CBBtrv.cpp ◆ CBBtrv.mak ◆ CBBMain.cpp ◆ CBBMain.dfm ◆ CBBMain.h
Delphi	<ul style="list-style-type: none"> ◆ Embarcadero Delphi 1 ◆ Embarcadero Delphi 3 以降 	<ul style="list-style-type: none"> ◆ BtrConst.pas (共通 Btrieve 定数) ◆ Btr32.dpr ◆ Btr32.dof ◆ BtrSam32.dfm ◆ BtrSam32.pas (サンプル プログラム) ◆ BtrApi32.pas ◆ BtrConst.pas (共通 Btrieve 定数)
Pascal	<ul style="list-style-type: none"> ◆ Borland Turbo Pascal 5 - 6 ◆ Borland Pascal 7 for DOS ◆ Extended DOS Pascal for Turbo Pascal 7 ◆ Borland Turbo Pascal 1.5 ◆ Borland Pascal 7 for Windows 	<ul style="list-style-type: none"> ◆ BtrApid.pas ◆ BtrSampd.pas (サンプル プログラム) ◆ BtrConst.pas (共通 Btrieve 定数) ◆ BlobHdr.pas ◆ BtrApiw.pas ◆ BtrSampw.pas (サンプル プログラム)
Visual Basic	<ul style="list-style-type: none"> ◆ Microsoft Visual Basic for Windows NT 以降 / Windows 98/ME 	<ul style="list-style-type: none"> ◆ BtSamp32.vbp ◆ BtrSam32.bas (サンプル プログラム) ◆ BtrFrm32.frm

以下の表に、Create や Stat などの Btrieve オペレーション用データ バッファで使用されるいくつかの共通データ型の比較を示します。

表 28 Btrieve データ バッファで使用される共通データ型

Assembly	C	COBOL	Delphi	Pascal	Visual Basic
doubleword	long ¹	PIC 9(4)	longint ¹	longint ¹	Long Integer
word	short int ¹	PIC 9(2)	smallint ¹	integer ¹	Integer
Byte	char	PIC X	char	char	String
Byte	Unsigned char	PIC X	Byte	Byte	Byte
¹ 整数値は、開発する環境により異なります。32 ビット環境では、整数は Long Integer と同じです。16 ビット環境では、整数は short int または small int と同じです。					

プログラミングの注意事項

Btrieve 関数を呼び出すと、常に、ステータス コードに対応する INTEGER 値が返されます。Btrieve 呼び出しの後、アプリケーションは常にこのステータス コードの値を確認します。ステータス コード 0 は、正常終了したオペレーションを示します。アプリケーションは、非ゼロのステータスを認識し、解決できなければなりません。

どの呼び出しでもすべてのパラメーターを提供する必要がありますが、MicroKernel はすべてのオペレーションにすべてのパラメーターを使用するわけではありません。各オペレーションに関連するパラメーターの詳細説明については、『Btrieve API Guide』を参照してください。

C/C++

ここでは、Btrieve API の C/C++ モジュール情報を示します。

C/C++ インターフェイスは、プラットフォーム依存型アプリケーションの作成を容易にします。このインターフェイスは、DOS、Windows、Linux、および OS X における開発をサポートします。これらのモジュールについては、表 27 でも説明しています。

インターフェイス モジュール

ここでは、C 言語インターフェイスを構成するモジュールについて詳述します。

BTRAPI.C

BTRAPI.C ファイルは、C アプリケーション インターフェイスの実際の実装です。BTRV および BTRVID を呼び出すすべてのアプリケーションをサポートします。これらの関数のいずれかで Btrieve 呼び出しを行う場合、BTRAPI.C をコンパイルし、そのオブジェクトをアプリケーションの他のモジュールとリンクします。

BTRAPI.C ファイルには、BTRAPI.H、BTRCONST.H、BLOBHDR.H および BTITYPES.H を取り込むようにコンパイラに指示する #include ディレクティブが含まれています。これらのファイルを取り込むことによって、BTRAPI.C はインターフェイスに関連するプラットフォームの独立性を与えるデータ型を利用します。

BTRAPI.H

BTRAPI.H ファイルには、Btrieve 関数のプロトタイプが含まれています。プロトタイプ定義では、BTITYPES.H ファイルで定義されているプラットフォーム依存データ型を使用します。BTRAPI.H は、BTRV 関数と BTRVID 関数を呼び出すすべてのアプリケーションをサポートします。

BTRCONST.H

BTRCONST.H ファイルには、Btrieve 固有の有効な定数が含まれています。これらの定数を使用すると、Btrieve オペレーション コード、ステータス コード、ファイル仕様フラグ、キー仕様フラグなどの多数の項目への参照を容易に標準化できます。

BTRCONST.H を利用せずに C アプリケーション インターフェイスを使用できますが、このファイルを取り込んでプログラミング作業を単純化することができます。

BTITYPES.H

BTITYPES.H ファイルは、プラットフォーム依存データ型を定義します。Btrieve 関数呼び出しで BTITYPES.H 内のデータ型を使用して、アプリケーションは各オペレーティング システム間で移植されます。

BTITYPES.H では、アプリケーションが動作するオペレーティング システムを指示する際に使用しなければならないスイッチについても説明しています。表 29 はこれらのオペレーティング システムのスイッチのリストを示したものです。

表 29 Btrieve API オペレーティング システム スイッチ

オペレーティング システム	アプリケーションの種類	スイッチ
DOS	16 ビット Tenberry Extender および BStub.exe を使用した 32 ビット Phar Lap 6 を使用した 32 ビット Embarcadero PowerPack を使用した 32 ビット	BTI_DOS BTI_DOS_32R BTI_DOS_32P BTI_DOS_32B
Linux	32 ビット	BTI_LINUX
Linux	64 ビット	BTI_LINUX_64

表 29 Btrieve API オペレーティング システム スイッチ

オペレーティング システム	アプリケーションの種類	スイッチ
OS X	64 ビット	BTI_MACOSX_64
Win32	32 ビット Windows	BTI_WIN_32
Win64	64 ビット Windows	BTI_WIN_64

BTRSAMP.C

BTRSAMP.C ソース ファイルは、表 29 で説明するオペレーティング システムでコンパイル、リンクおよび実行できるサンプル Btrieve プログラムです。

プログラミングの必要条件

C アプリケーション インターフェイスを使用してアプリケーション プラットフォームを独立型にする場合は、BTITYPES.H で説明しているデータ型を使う必要があります。これらのデータ型の使用方法については、BTRSAMP.C ファイルを参照してください。



メモ また、プログラムが動作するオペレーティング システムを識別するディレクティブを指定する必要があります。ディレクティブに使用できる値は、ヘッダー ファイル BTITYPES.H に列挙されています。お使いのコンパイラの適切なコマンド ライン オプションを使用して、ディレクティブを指定してください。

Delphi

Btrieve Delphi モジュールについては、表 27 で説明しています。

DOS (Btrieve)

ここでは、DOS アプリケーションが Btrieve API をどのように使用できるかについて説明します。

インターフェイス モジュール

Btrieve API を使用する DOS アプリケーション向けの言語インターフェースは、以下のモジュールによって構成されます。

BTRAPI.C

BTRAPI.C ファイルは、C アプリケーション インターフェイスの実装です。また、このファイルには次のような DOS インターフェイスも含まれます。

```
#if defined( BTI_DOS )
    BTI_API BTRVID(
        BTI_WORD operation,
        BTI_VOID_PTR posBlock,
        BTI_VOID_PTR dataBuffer,
        BTI_WORD_PTR dataLength,
        BTI_VOID_PTR keyBuffer,
        BTI_SINT keyNumber,
        BTI_BUFFER_PTR clientID )
```

BTRAPI.C は、BTRV および BTRVID を呼び出すすべてのアプリケーションをサポートします。これらの関数のいずれかで Btrieve 呼び出しを行う場合、BTRAPI.C をコンパイルし、そのオブジェクトをアプリケーションの他のモジュールとリンクします。

BTRAPI.C ファイルには、BTRAPI.H、BTRCONST.H、BLOBHDR.H および BTITYPES.H を取り込むようにコンパイラに指示する #include ディレクティブが含まれています。これらのファイルを取り込むことによって、BTRAPI.C はインターフェイスに関連するプラットフォームの独立性を与えるデータ型を利用します。

BTRAPI.H

BTRAPI.H ファイルには、Btrieve 関数のプロトタイプが含まれています。プロトタイプ定義では、BTITYPES.H ファイルで定義されているプラットフォーム依存データ型を使用します。BTRAPI.H は、BTRV 関数と BTRVID 関数を呼び出すすべてのアプリケーションをサポートします。

BTRCONST.H

BTRCONST.H ファイルには、Btrieve 固有の有効な定数が含まれています。これらの定数を使用すると、Btrieve オペレーション コード、ステータス コード、ファイル仕様フラグ、キー仕様フラグなどの多数の項目への参照を容易に標準化できます。

BTRCONST.H を利用せずに C アプリケーション インターフェイスを使用できますが、このファイルを取り込んでプログラミング作業を単純化することができます。

BTITYPES.H

BTITYPES.H ファイルは、プラットフォーム依存データ型を定義します。Btrieve 関数呼び出しで BTITYPES.H 内のデータ型を使用して、アプリケーションは各オペレーティング システム間で移植されます。

BTITYPES.H では、アプリケーションが動作する DOS オペレーティング システムを指示する際に使用しなければならないスイッチについても説明しています。次の表は、これらのスイッチの一覧を示します。

表 30 DOS アプリケーション用の Btrieve API オペレーティング システム スイッチ

オペレーティング システム	アプリケーションの種類	スイッチ
DOS	16 ビット	BTI_DOS
	Tenberry Extender および BStub.exe を使用した 32 ビット ¹	BTI_DOS_32R
	Phar Lap 6 を使用した 32 ビット	BTI_DOS_32P
	Embarcadero PowerPack を使用した 32 ビット	BTI_DOS_32B

Pascal

ここでは、Btrieve API の Pascal ソース モジュールについて説明します。

その後、Pascal の Btrieve API ソース モジュールについて説明します。

ソース モジュール

Pascal インターフェイスは、以下のソース モジュールから構成されています。

- BTRAPID.PAS – DOS 用 Btrieve 関数インターフェイス ユニット
- BTRCONST.PAS – 共通 Btrieve 定数ユニット
- BTRSAMPD.PAS – DOS 用 サンプル Btrieve プログラム

BBTRAPID.PAS

BTRAPID.PAS には、DOS 用の Pascal アプリケーション インターフェイスのソース コードの実装が含まれています。このファイルは、Btrieve 関数を呼び出すアプリケーションをサポートします。

Turbo Pascal が、MicroKernel エンジンを実正しくコンパイルして、アプリケーションの他のモジュールとリンクするためには、BTRAPID.PAS をコンパイルして Turbo Pascal ユニットを作成し、そのユニットをアプリケーションのソース コードの `uses` 句に記載します。

BTRCONST.PAS

BTRCONST.PAS ファイルには、Btrieve に固有な有効の定数が含まれています。これらの定数を使用すると、Btrieve オペレーション コード、ステータス コード、ファイル仕様フラグ、キー仕様フラグなどの多数の項目への参照を容易に標準化できます。

BTRCONST.PAS を使用する場合、そのファイルをコンパイルして Turbo Pascal ユニットを作成し、次にそのユニットをアプリケーションのソース コードの `uses` 句に示すことができます。

BTRCONST.PAS を利用せずに Pascal アプリケーション インターフェイスを使用できますが、このファイルを使用してプログラミング作業を単純化することができます。

BTRSAMPD.PAS

ソース ファイル BTRSAMPD.PAS は、コンパイル、リンクおよび実行できるサンプル Btrieve プログラムです。



メモ アプリケーションがバリエーション文字列を含む Pascal レコード構造を使用する場合は、たとえレコードがバック化されていなくても、Pascal レコード内の奇数長の要素が追加の記憶バイトを必要とする場合があります。このことは、Create (14) オペレーションのレコード長を定義する場合に必要な重要問題です。レコード タイプの詳細については、Pascal リファレンス マニュアルを参照してください。

Visual Basic

ここでは、Btrieve API の Visual Basic ソース モジュールについて説明します。

Visual Basic は 32 ビット アプリケーションをコンパイルする際、UDT（ユーザー定義データ型）のメンバーをそれぞれ、その特定のメンバーのサイズに応じて 8 ビット、16 ビット、32 ビットの境界に配置します。構造体と違い、データベース行はパック化されます。つまり、フィールド間に未使用スペースがないということです。配置をオフにする方法はないので、Visual Basic アプリケーションがデータベースにアクセスできるように構造体をパック化およびアンパック化する方法が必要です。PSQL Btrieve アライメント DLL、つまり PALN32.DLL は、この配列の問題を処理するように設計されています。

Visual Basic の場合、この言語はさまざまなビットの倍数で要素を配置します。以下の表に、各種データ型と、Visual Basic がそれらのデータ型を処理する方法を示します。

Visual Basic データ型	データ型定数	一般的なサイズ（バイト単位）	境界
Byte	FLD_BYTE	すべて	1 バイト（なし）
String	FLD_STRING	すべて	1 バイト（なし）
Boolean	FLD_LOGICAL	2	2 バイト
Integer	FLD_INTEGER	2	2 バイト
Currency	FLD_MONEY	4	4 バイト
Long	FLD_INTEGER	4	4 バイト
Single	FLD_IEEE	4	4 バイト
Double	FLD_IEEE	8	4 バイト

プログラムは、BTRCALL 関数を呼び出して Visual Basic 内の Btrieve 呼び出しにアクセスします。この関数へのアクセスは、プロジェクトに BTRAPI.BAS モジュールを取り込むことによって行います。必要な関数の残りの部分は、PALN32.DLL 内にあります。

➤ プロジェクトに PALN32.DLL を取り込むには

- [プロジェクト] > [参照設定] を選択し、PSQL Btrieve Alignment Library モジュールを確認します。このモジュールが表示されない場合は、まず、参照ボタンを選択してファイルを検索することによってリストにモジュールを追加します。

以下の表に、各関数とその関数が必要とする特定のモジュールを示します。

関数	用途	場所	パラメーター	戻り値
BTRCALL	Btrieve オペレーションを実行するには	BTRAPI.BAS	<ul style="list-style-type: none"> ◆ OP As Integer 『Btrieve API Guide』に示すような Btrieve オペレーション番号。 ◆ Pb\$ As String レコードを取得または格納するためか、Btrieve に構造体を渡すための文字列に、ポジションブロックを格納します。 ◆ Db As Any データバッファ。このパラメーターは、レコードを取得または格納するためか、Btrieve に構造体を渡すために使用します。 ◆ DL As Long データバッファの長さ。 ◆ Kb As Any キーバッファ。 ◆ Kl As Integer キーバッファの長さ。 ◆ Kn As Integer キー番号。 	<ul style="list-style-type: none"> ◆ Integer オペレーションから返される Btrieve ステータスコード。特定のコードの詳細については、『Status Codes and Messages』を参照してください。
RowToStruct	Visual Base UDT にバイト行を変換します。	PALN32.DLL	<ul style="list-style-type: none"> ◆ row (1 to n) As Byte パック化されたデータを取得する入力配列。 ◆ fld (1 to n) As FieldMap 個々のフィールドのデータ型を決定するための FieldMap 配列。 ◆ udt As Any データを格納する UDT。 ◆ udtSize As Long UDT のサイズ。LenB () を使用してこの値を生成します。 	<ul style="list-style-type: none"> ◆ Integer 問題なければ 0 です。それ以外はエラーが発生しました。
SetFieldMap	FieldMap 要素のメンバーを設定します。	PALN32.DLL	<ul style="list-style-type: none"> ◆ map As FieldMap FieldMap 配列の要素。 ◆ dataType As Integer フィールドタイプ、以下の定数上のパス。 - FLD_STRING - FLD_INTEGER - FLD_IEEE - FLD_MONEY - FLD_LOGICAL - FLD_BYTE - FLD_UNICODE¹ ◆ size As Long データベースに格納されているようなバイト単位のフィールドのサイズ。 	<ul style="list-style-type: none"> ◆ なし

関数	用途	場所	パラメーター	戻り値
SetFieldMap FromDDF	FieldMap タイプ の配列のすべての メンバーを設定 します。	PALN32.DLL	<ul style="list-style-type: none"> ◆ path As String データ ソースへの絶対パス名。 ◆ table As String テーブルの名前。 ◆ userName As String 予約済み。ヌル文字列 ("") を渡します。 ◆ passwd As String 予約済み。ヌル文字列 ("") を渡します。 ◆ map (1 to n) As FieldMap 記入する出力先 FieldMap 配列。この文字 列には、レコードのフィールド数として正 確な要素数が含まれていなければなりま せん。 ◆ unicode As Integer 文字列が ASCII としてレコードに格納さ れる場合は 0 です。それ以外は、ユニコー ドとして格納されます。 	<ul style="list-style-type: none"> ◆ Integer 問題なければ 0 で す。それ以外はエ ラーが発生しまし た。
StructToRow	バイト行に Visual Basic UDT を変換 します。	PALN32.DLL	<ul style="list-style-type: none"> ◆ row (1 to n) As Byte パック化されたデータを格納する出力配 列。 ◆ fld (1 to n) As FieldMap 個々のフィールドのデータ型を決定する ための FieldMap 配列。 ◆ udt As Any データを取得する UDT。 ◆ udtSize As Long UDT のサイズ。LenB () を使用してこの 値を生成します。 	<ul style="list-style-type: none"> ◆ Integer 問題なければ 0 で す。それ以外はエ ラーが発生しまし た。
¹ フィールド タイプ FLD_UNICODE は、データベース行（パック構造）だけでなく UDT（ユーザー定義データ型）内の両 方に UNICODE で格納される Visual Basic の String 型のフィールドを指定するのに使用します。フィールドに FLD_STRING 型が使用されると、データベース行ではデフォルトの ANSI コード ページ文字セットに変換されます。ただし、UDT（ユー ザー定義データ型）では UNICODE が使用されます。要するに、データベースに文字列フィールドを UNICODE で格納した い場合は、フィールド タイプに FLD_UNICODE を選択するということです。文字列フィールドを、デフォルトであるシス テムの ANSI コード ページ文字セットでデータベースに格納したい場合は、FLD_STRING を選択します。				

インターフェイス ライブラリ

7

この章では、以下の項目について説明します。

- 「[インターフェイス ライブラリの概要](#)」
- 「[PSQL アプリケーションの配布](#)」

インターフェイス ライブラリの概要

Windows アプリケーションから MicroKernel エンジンにアクセスする適切な方法は、コンパイル時に Btrieve Glue DLL を参照するライブラリにリンクすることです。Glue DLL は、インターフェイス DLL にアプリケーションを接着する役目を果たします。接着剤のように、Glue DLL はアプリケーションとインターフェイス DLL の間に介在する薄い層です。Glue DLL は、以下のアクションを正常に実行する役割を担っています。

- 1 インターフェイス DLL を読み込む。
- 2 インターフェイス DLL にバインドする、つまり、インターフェイス DLL からシンボルをインポートする。

どの段階でも、Glue DLL で障害状態が発生した場合は、アプリケーションが障害をユーザーに伝えるための適切なステータス コードを Glue DLL が発行します。

表 31 には、アプリケーションがリンクできるライブラリ、および読み込む DLL を示します。

表 31 MicroKernel エンジン プログラミング ライブラリ

オペレーティング システムとコンパイラ ¹	Glue DLL	リンク ライブラリ
Windows 32 ビット (Microsoft Visual C++, Watcom, Embarcadero)	W3BTRV7.DLL	W3BTRV7.LIB
Windows 64 ビット	W64BTRV.DLL	W64BTRV.LIB
¹ コンパイラ対応ライブラリは、さまざまなサブディレクトリ内にあります。Win32 アプリケーションをリンクするには、Microsoft コンパイラを使用する場合に ¥Win32 ディレクトリを使用し、Embarcadero または Watcom コンパイラを使用する場合に ¥Win32x ディレクトリを使用します。		

Linux および OS X

Linux および OS X には glue コンポーネントがありません。アプリケーションはインターフェイスを実装する共有ライブラリに対して直接リンクします。Linux 32 ビットと、Linux および OS X 64 ビット アプリケーション用の MicroKernel エンジンのリンク ライブラリはいずれも libpsqlmif です。

PSQL アプリケーションの配布

PSQL データベース エンジンでアプリケーションを開発する予定であれば、アプリケーションを配布する場合に以下の条件を心得ておいてください。

- 「[PSQL の配布規則](#)」
- 「[PSQL ActiveX ファイルの登録](#)」
- 「[PSQL を開発したアプリケーションと共にインストールする](#)」

PSQL の配布規則

PSQL でアプリケーションを開発した後、製品を配布する場合は Actian Corporation とのライセンス契約に注意してください。配布権に関するご質問は、弊社のマーケティング部署にお問い合わせください。

PSQL ActiveX ファイルの登録

次の表に、PSQL の ActiveX インターフェイスで構築されたアプリケーションの実行に必要なファイルを示します。

表 32 再配布可能なファイル

ファイル	場所	説明
ACBTR732.OCX	システム ディレクトリ	PSQL データ ソース コントロール
ACCTR732.OCX	システム ディレクトリ	PSQL バウンド コントロール
PEDTCONV.DLL	システム ディレクトリ	データ変換用 DLL
PBTRVD32.DLL	システム ディレクトリ	メタ データ処理用 DLL
SBTRV32.DLL	システム ディレクトリ	IDS 通信用 DLL
SWCOMP32.DLL	システム ディレクトリ	データ圧縮用 DLL

PSQL ActiveX ファイルと必要な DLL を配布した場合、ActiveX ファイルが正しく機能するように ActiveX ファイルを登録する必要があります。これらのファイルを登録するには次の 2 種類の方法があります。

- InstallShield などの現行のインストール用ユーティリティの多くは、インストール処理中に ActiveX コントロールを自動的に登録するよう変更することができます（詳細については、お使いになる特定のインストール用ユーティリティのマニュアルを参照してください）。
- ActiveX コントロールを登録するもう 1 つの方法は、再配布可能なファイル REGSVR32.EXE をインストールし、インストール中またはインストール後に実行する方法です。これは単純な ActiveX 登録用ユーティリティで、登録する ActiveX の名前をコマンド ライン パラメーターとして受け取ります（たとえば、REGSVR32 C:\MyInstall\ACBTR732.OCX）。

PSQL を開発したアプリケーションと共にインストールする

PSQL インストールのカスタマイズに関する情報は、『Installation Toolkit Handbook』を参照してください。

レコードの処理

8

この章では、以下の項目について説明します。

- 「オペレーションのシーケンス」
- 「レコードへのアクセス」
- 「レコードの挿入と更新」
- 「マルチレコードのオペレーション」
- 「キーの追加と削除」

オペレーションのシーケンス

Btrieve オペレーションの中には、Create (14)、Reset (28)、Version (26) などのように、いつでも発行できるものがあります。しかし、ほとんどの Btrieve オペレーションでは、Open オペレーション (0) を使用してファイルを開く必要があります。その場合、ファイル内に位置、つまりカレンシーを確立しないと、レコードを処理できません。

キーに基づいて論理カレンシーを確立したり、Step オペレーションを使用して物理カレンシーを確立したりすることができます。

- 物理カレンシーを確立するには、以下のオペレーションのうちの 1 つを使用します。
 - Step First (33)
 - Step Last (34)
- 論理カレンシーを確立するには、以下のオペレーションのうちの 1 つを使用します。
 - Get By Percentage (44)
 - Get Direct/Record (23)
 - Get Equal (5)
 - Get First (12)
 - Get Greater Than (8)
 - Get Greater Than or Equal (9)
 - Get Last (13)
 - Get Less Than (10)
 - Get Less Than or Equal (11)

カレンシーを確立した後、Insert (2)、Update (3)、Delete (4) などの対応する I/O オペレーションを発行できます。



メモ Btrieve ファイルで I/O を実行する場合は常に Btrieve オペレーションを使用してください。Btrieve ファイルで標準 I/O を実行しないでください。

カレンシーに基づいて、以下のようにファイル内を移動できます。

- 物理カレンシーに基づいて位置を確立したら、Step Next (24)、Step Next Extended (38)、Step Previous (35)、Step Previous Extended (39) のいずれかのオペレーションを使用します。Step オペレーションは、アプリケーションが特定の順序でレコードを取得する必要がある場合にすばやくファイル内を探索するのに有効です。Extended オペレーションは、多数のレコードを一度で処理するのに有効です。
- 論理カレンシーに基づいて位置を確立したら、オペレーション Get Next (6)、Get Next Extended (36)、Get Previous (7)、Get Previous Extended (37) のいずれかのオペレーションを使用します。Get オペレーションは、特定の順序でファイル内を探索するのに有効です。Extended オペレーションは、多数のレコードを一度で処理するのに有効です。

1 つのオペレーションで物理カレンシーを確立し、次に論理カレンシーを必要とするオペレーションを続けることはできません。たとえば、Step First オペレーションを発行した後に Get Next オペレーションを発行することはできません。

データオンリー ファイルでは、MicroKernel はインデックス ページの保守や作成を行いません。Step オペレーションと Get Direct/Record オペレーション (23) だけを使用してレコードにアクセスできますが、すべてのオペレーションでは物理位置でレコードを検索します。

キーオンリー ファイルでは、MicroKernel はデータ ページの保守や作成を行いません。Get オペレーションだけを使用してレコードにアクセスできますが、Get オペレーションでは論理カレンシーでレコードを検索します。

ファイルの処理を終了したら、Close オペレーション (1) を使用してファイルを閉じます。アプリケーションがいつでも終了できる状態にあれば、Stop オペレーション (25) を発行します。



メモ Stop オペレーションを実行できないと、MicroKernel はオペレーティング システムにそのリソースを返せません。この障害により、実際に、アプリケーションが動作しているコンピュータをクラッシュする可能性も含めて予測できないシステム動作が発生します。

レコードへのアクセス

Btrieve は、データへの物理アクセスと論理アクセスの両方を行います。物理アクセスでは、Btrieve はファイル内の物理レコード アドレスに基づいてレコードを取得します。論理アクセスでは、Btrieve はレコードに含まれているキー値に基づいてレコードを取得します。また、Btrieve ではレコード内のデータの「チャンク」にアクセスできます。

物理位置によるレコードへのアクセス

物理位置によるレコード アクセスは、以下の理由で高速になります。

- MicroKernel がインデックス ページを使用する必要がない。
- 物理レコードが存在するページはほとんどの場合キャッシュにあるため、通常、その前後のレコードは MicroKernel のメモリ キャッシュ内に既に存在している。

物理カレンシー

物理カレンシーは、物理位置でレコードにアクセスする際のポジショニングに影響を与えます。レコードを挿入する場合、レコードに含まれているキー値とは無関係に、MicroKernel はファイル内の最初の空きスペースにそのレコードを書き込みます。この場所はレコードの物理位置、つまり、アドレスと呼びます。レコードは、ファイルから削除するまでこの位置に残ります。Btrieve Step オペレーションは、物理位置を使用してレコードにアクセスします。

最後にアクセスされたレコードが、現在の物理レコードです。次の物理レコードは、現在の物理レコードに対してすぐ上位のアドレスを持つレコードです。直前の物理レコードは、すぐ下位のアドレスを持つレコードです。最初の物理レコードの直前の物理レコードはありません。同様に、最後の物理レコードの次の物理レコードはありません。

まとめると、現在の物理位置、次の物理位置、直前の物理位置がファイル内の物理カレンシーを構成します。

Step オペレーション

アプリケーションは、Step オペレーションを使用して、ファイル内の物理位置に基づいてレコードにアクセスできます。たとえば、Step First オペレーション (33) は、ファイル内の最初、つまり、最下位の物理位置に格納されているレコードを取得します。



メモ キーオンリー ファイルでは Step オペレーションを実行できません。

Step Next オペレーション (24) は、次に上位の物理位置に格納されているレコードを取得します。Step Previous オペレーション (35) は、ファイル内の次に下位の物理位置に格納されているレコードを取得します。Step Last オペレーション (34) は、ファイル内の最後、つまり、最上位の物理位置に格納されているレコードを取得します。

Step Next Extended (38) オペレーションと Step Previous Extended (39) オペレーションは、現在のレコードの後または前の物理位置から 1 つまたは複数のレコードを取得します。



メモ 各 Step オペレーションは物理カレンシーを再確立しますが、論理カレンシーについては、先に存在していたとしても破壊します。

キー値によるレコードへのアクセス

キー値でレコードにアクセスすると、指定されたキーに対するレコードの値に基づいてレコードを取得できます。

論理カレンシー

論理カレンシーは、キー値でレコードにアクセスする際のポジショニングに影響を与えます。ファイルにレコードを挿入すると、MicroKernel はレコード内の対応するキーに非ヌル値が存在する各 B ツリーを更新します。ファイルの各キーは、レコードの論理順序を決定します。この順序は、キーの定義済みのソート順序または ACS で決定されます。

最後にアクセスされたレコードが、現在の論理レコードです。このレコードは、必ずしも、最後に取得されたレコードではありません。最後のレコードは Get Direct/Chunk オペレーション (23) で取得されている可能性があります。このオペレーションは論理カレンシーを変更しません。次の論理レコードは、定められた論理順序で直前にあるレコードです。直前の論理レコードは、定められた論理順序で直前にあるレコードです。最初の論理レコードの直前の論理レコードはありません。同様に、最後の論理レコードの次の論理レコードはありません。

まとめると、現在の論理レコード、次の論理レコード、直前の論理レコードがファイル内の論理カレンシーを構成します。

no-currency-change (NCC) オプションを使用するオペレーションを実行する場合や、ヌル キー値を持つレコードを処理する場合を除き、現在の論理レコードは現在の物理レコードでもあります。たとえば、NCC Insert オペレーション (2) を実行し、挿入以前にあったものと同じ論理位置をファイル内に持つことができます。物理カレンシーが更新されます。

NCC オペレーションは、別のオペレーションを実行するために論理カレンシーを保存しなければならない場合に有効です。たとえば、レコードの挿入または更新を行い、オリジナルの論理カレンシーに基づいて Get Next オペレーション (6) を使用しなければならない場合があります。

NCC Insert オペレーション

```
status = BTRV( B_GET_FIRST, posBlock, dataBuf, &dataLen, keyBuf, keyNum); /* キーパスの最初のレコードを取得 */

for (i = 0; i < numRecords; i++)
{ status = BTRV( B_INSERT, posBlock, dataBuf, &dataLen, keyBuf, -1); /* キー番号-1 はカレンシー変更なしを示す */
} /* 複数レコードを挿入 */

status = BTRV( B_GET_FIRST, posBlock, dataBuf, &dataLen, keyBuf, keyNum); /* キーパスの最初のレコードの次のレコードを取得 */
```



メモ NCC オペレーションを使用する場合、MicroKernel はキー バッファー パラメーターに情報を返しません。論理カレンシーを保持する場合、NCC オペレーションの後にキー バッファー内の値を変更しないでください。それ以外は、次の Get オペレーションで予測できない結果が発生するおそれがあります。

Get オペレーション

アプリケーションは、Get オペレーションを使用して、指定されたキーの値に基づいてレコードを取得できます。対応する Get オペレーションは、ファイルから特定のレコードを取得したり、ある順序でレコードを取得できます。

たとえば、Get First オペレーション (12) はキー番号パラメーターで指定されたキーで最初のレコードを取得します。同様に、Get Last オペレーション (13) は、指定されたキーに基づいて論理順序に従って最後のレコードを取得します。Get Equal (5) や Get Less Than (10) などの Get オペレーションの中には、アプリケーションがキー バッファー パラメーターで指定するキー値に基づいてレコードを返すものがあります。

レコードの処理

➤Get オペレーションは論理カレンシーを確立します。アプリケーションは、以下の手順を実行することによってあるキーから別のキーへ変更できます。

- 1 Get オペレーションのうちの 1 つを発行することによってレコードを取得します。
- 2 レコードの 4 バイト物理アドレスを取得するために、Get Position オペレーション (22) を発行します。
- 3 Get Direct/Record オペレーション (23) を発行し、4 バイトの物理アドレスと変更するキー番号を MicroKernel に渡します。

Get Position (22) 以外の Get オペレーションは、論理カレンシーを確立するほか、物理カレンシーも確立します。したがって、Step Next (24) オペレーションまたは Step Previous (35) オペレーションを続けることができます。ただし、Step オペレーションを使用すると論理カレンシーが破壊されます。

➤Step オペレーションを使用した後に論理カレンシーを再確立するには、以下の手順で行います。

- 1 Step オペレーションを使用した直後に、Get Position オペレーション (22) を発行して取得されたレコードの 4 バイトの物理アドレスを取得します。
- 2 Get Direct/Record オペレーション (23) を発行し、4 バイトの位置と論理カレンシーを確立するキー番号を MicroKernel に渡します。

可変長レコードの読み取り

可変長の読み取りは、データ バッファ長パラメーターを使用してレコードを返すためのスペース量を MicroKernel に指示するという点で、固定長レコードの読み取りと同じです。このパラメーターをデータ バッファのサイズに設定すれば、データ バッファは最大のデータ量を収容できます。



メモ データ バッファ長を、データ バッファに割り当てられたバイト数より大きい値に設定しないでください。設定すると、アプリケーションの実行時にメモリが上書きされるおそれがあります。

読み取りオペレーションが成功した後、返されたレコードのサイズ、つまり固定長部分のサイズに可変長部分の実際のデータ量を足したサイズ（レコードの最大サイズではない）を反映して、データ バッファ長パラメータは変更されます。アプリケーションはこの値を使用して、データ バッファ内のデータ量を決定する必要があります。

たとえば、データ ファイル内に以下のレコードがあるとしましょう。

キー 0 : Owner 30 バイト ZSTRING	キー 1 : Account 8 バイト INTEGER	Balance (キーではない) 8 バイト	Comments (キーではない) 1000 バイト
John Q. Smith	263512477	1024.38	解説
Matthew Wilson	815728990	644.29	解説
Eleanor Public	234817031	3259.78	解説

以下に、Get Equal オペレーションの例を示します。



メモ アプリケーションの開発とデバッグを行う間、このオペレーションは読み取りオペレーションの直前と直後にデータ バッファ長を表示して、それが各時点で正しく設定されているかどうかを確認するのに役立ちます。

C での Get Equal オペレーション

```
/* B_GET_EQUAL を使用して key 1 = 263512477 のレコードを取得 */
memset(&dataBuf, 0, sizeof(dataBuf));
dataBufLen = sizeof(dataBuf); /* この値は 1047 */
account = 263512477;
*(BTI_LONG BTI_FAR *)&keyBuf[0] = account;
status = BTRV( B_GET_EQUAL, posBlock, &dataBuf, &dataBufLen, keyBuf, 1);
/* dataBufLen は現在 56 */
```

Visual Basic での Get Equal オペレーション

```
dataBufLen= length(dataBuf) ' この値は 1047
account% = 263512477
status = BTRV(B_GETEQUAL, PosBlock$, dataBuf, dataBufLen, account%, 1)
' dataBufLen は現在 56
```

返されたレコードがデータ バッファ長で指定された値より長いと、MicroKernel はデータ バッファ長で設定されたサイズに従ってできるだけ多くのデータを返し、ステータス コード 22 を返します。

チャンクによるレコードへのアクセス

Btrieve のデータ バッファ長パラメーターは 16 ビットの符号なし整数であるため、レコード長を 65,535 に制限します。Chunk オペレーションは、レコードの部分の読み取りまたは書き込みを行えるようにして、この制限をはるかに超えてレコード長を拡張します。チャンクは、オフセットと長さとして定義されます。オフセットは 64 GB までの大きさにすることができますが、長さは 65,535 バイトに制限されます。オペレーティング システムおよびデータ バッファ長パラメーターの制限は、Chunk オペレーションにも適用されます。ただし、Chunk オペレーションはレコードのどの部分にもアクセスできるので、この制限はレコード長に影響を与えず、1 つのオペレーションでアクセスできるチャンクの最大サイズにのみ影響を与えます。

たとえば、Chunk オペレーションで、アプリケーションは 3 つのチャンク取得呼び出しを行うことによって 150,000 バイト レコードを読み取ることができます。この例の各チャンクの長さは、50,000 バイトです。最初のチャンクはオフセット 0 から始まり、次のチャンクはオフセット 50,000 から始まり、最後のチャンクはオフセット 100,000 から始まります。

チャンクのオフセットと長さは、キー セグメント、レコードの固定長部分、可変長部などの MicroKernel が認識するレコードの内部構造に対応する必要はありません。また、チャンクは、アプリケーションが定義するフィールドなどのレコードの部分に一致する必要はありません。ただし、定義された部分をチャンクとして更新すると便利な場合があります。



メモ チャンクは、チャンクを定義しているオペレーションの実行中のみ有効です。

場合によっては、クライアント / サーバー環境で Chunk オペレーションを使用すると、クライアントのリクエスターはより小さなデータ バッファ長を設定してリクエスターのメモリ必要量を少なくすることができます。たとえば、アプリケーションがレコード全体のオペレーションを使用し、最高 50 KB 長のレコードにアクセスした場合、リクエスターはデータ バッファ長を最低でも 50 KB に設定しなければならず、それによって 50 KB の RAM を使用することになります。しかし、たとえば、アプリケーションが Chunk オペレーションを使用し、各チャンクのサイズを 10 KB に制限した場合、リクエスターはデータ バッファ長を 10 KB に設定することができます、それによって 40 KB の RAM が節約されることになります。

レコード内のカレンシー

レコード内のカレンシーが Chunk オペレーションに関連するのは、現在のレコード内のオフセットを追跡するからです。現在位置は、読み取りまたは書き込みが行われたチャンクの最後のバイトを 1 バイト越えたオフセットです。（これに関しては、最後のオペレーションでレコード全体を読み取ろうとし、MicroKernel がそのレコードの一部のみを返した場合でも同じです。レコードの一部のみを返す現象は、データ バッファ長が不十分であるときに発生します。）

レコードの処理

例外は、Truncate サブファンクションを使用する Update Chunk オペレーション (53) の場合です。この場合、MicroKernel は切り捨てられたレコードの終わりから 1 バイト先のオフセットを現在位置と定義します。

MicroKernel はレコード内カレンシーを追跡することにより、以下のことが行えます。

- Chunk オペレーションに対してネクストインレコード サブファンクションのバイアスを与える。
チャンクのオリジナルのオフセット、長さおよび個数を指定すると、MicroKernel はそれ以降のオフセットを計算します。
- チャンクにアクセスする際のパフォーマンスを向上する。
レコード内カレンシーは、ポジションブロックで最後にアクセスされた同じレコードを処理する限り、また、次のチャンク オフセットがレコード内の現在位置より大きい限り、Chunk オペレーションを高速化することができます。(つまり、レコード内カレンシーの利点を得るためにレコード内の直後のバイトにアクセスする必要はありません。)



メモ MicroKernel は、現在のレコードのレコード内カレンシーのみを保持します。物理カレンシーまたは論理カレンシーを変更すると、MicroKernel はレコード内カレンシーもリセットします。

Chunk オペレーション

Get Direct/Chunk オペレーション (23) と Update Chunk オペレーション (53) を使用して、チャンクにアクセスします。これらのオペレーションを使用するには、チャンクのオフセットと長さを定義するチャンク記述子構造を定義する必要があります。Get Direct/Chunk オペレーションの場合、チャンク記述子構造は MicroKernel がチャンクを返す先のアドレスも指定する必要があります。

Get Direct/Chunk オペレーションを使用する前に、Get Position オペレーション (22) を発行することによって現在のレコードの物理アドレスを取得する必要があります。1 つの Chunk オペレーションで、ネクストインレコード サブファンクション バイアスを使用してレコード内の複数のチャンクを取得または更新することができます。

レコードの挿入と更新

ほとんどの場合、レコードの挿入と更新は簡単なプロセスです。つまり、Insert オペレーション (2) または Update オペレーション (3) を使用し、データ バッファを使用してレコードを渡します。ここでは、挿入と更新に関連するいくつかの特別な場合について説明します。

ミッションクリティカルな挿入と更新における信頼性の確保

MicroKernel は非常に信頼性のあるデータ管理エンジンですが、システム障害を防ぐことができません。クライアント / サーバー アプリケーションでシステム障害がよく起こるのは、ネットワーク障害が発生する可能性があるからです。以下の MicroKernel 機能を利用して、信頼性を高めることができます。

- トランザクション一貫性保守。一貫性保守は、アプリケーションが End Transaction オペレーションから正常終了を示すステータス コードを受け取る前に、変更がディスクへコミットされていることを保証します。トランザクションは通常、複数の変更操作が 1 つのグループとして成功または失敗する必要がある場合、それらをグループ化するのに使用されます。また一方、変更がディスクへコミットされるタイミングはアプリケーションが管理するため、トランザクション一貫性保守は単一の操作にとっても役に立ちます。

Begin Transaction オペレーションと End Transaction オペレーションの内側で個々のミッション クリティカルな挿入操作および更新操作を「ラップ」し、MicroKernel の [トランザクション一貫性保守] 設定オプションを使用することを検討してください。Begin Transaction オペレーションと End Transaction オペレーションの詳細については、『Btrieve API Guide』を参照してください。トランザクション一貫性保守の詳細については、「[トランザクション一貫性保守](#)」を参照してください。



メモ アクセラレイティド モードでファイルを開くと、そのファイルに対するトランザクション ログは実行されなくなります。つまり、アクセラレイティド モードで開かれたファイル上で行われたオペレーションはトランザクション一貫性がありません。

- システム トランザクション回数。トランザクション一貫性保守を使用する別の方法として、MicroKernel のオペレーション バンドル制限と起動時間制限を使用してシステム トランザクションの回数を制御する方法があります。開いているファイルごとに、MicroKernel は一連のオペレーションを 1 つのシステム トランザクションにバンドルします。システム障害が発生すると、最新のトランザクション以前に行われた変更の内容がすべて失われますが、ファイルは矛盾のない状態に復元されるので、システム障害の原因を解決した後にオペレーションを再試行することができます。

[オペレーション バンドル制限] 設定と [起動時間制限] 設定をともに 1 に設定すると、MicroKernel は各オペレーションを個別のシステム トランザクションとしてコミットします。そうするとパフォーマンスが下がるので、この方法はパフォーマンスの低下を認めることのできるアプリケーションにのみ有効です。これを決定する 1 つの方法は、アプリケーションの実行中に CPU の利用度を測定する方法です。CPU の 50 ~ 100% を利用するアプリケーションは、このアプローチの良い候補ではありません。

重複不可キーの挿入

レコードを挿入する場合で、同じキー値が既に存在する可能性があり、それを重複させたくない場合は、次のいずれかの方法を実行します。

- Insert オペレーション (2) を実行する。MicroKernel がステータス コード 5 を返すと、キー値は存在し、挿入を行えません。
- Get Key バイアス (55) で Get Equal オペレーションを実行する。MicroKernel からステータス コード 4 が返された場合、キー値はまだ存在していないので、挿入を実行できます。

Insert オペレーションが単独で存在し、ファイル内の論理カレンシーに依存しない場合、各 Insert より前に Get Equal を実行すると、オーバーヘッドが増えます。しかし、挿入がグループになっている場合は、Get Equal オペ

レコードの処理

レーションがキー位置を指すインデックス ページをメモリに取り込むことで、これ以降に行われる Insert オペレーションが促進されます。

可変長レコードの挿入および更新

可変長データ ファイルを設計する場合は、アプリケーションがサポートするレコードの可変長部分の最大サイズを決定する必要があります。固定長部分と可変長部分の最大サイズを合計したサイズを収容するレコード構造を設定する必要があります。可変長レコードの読み取り、挿入および更新を行うときは、この構造体をデータ バッファとして使用します。

可変長レコードを挿入または更新する場合は、データ バッファ長パラメーターによって書き込むデータ量を **MicroKernel** に指示します。固定長部分のサイズに可変長部分の実際のデータ量を加えたサイズに、このパラメーターを設定します。データ バッファ長を、アプリケーションが可変長フィールドに対して許可する最大サイズに固定長を加えたサイズに設定しないでください。設定すると、**MicroKernel** は常に最大サイズを書き込みます。

たとえば、以下のレコードを挿入すると仮定します。

キー 0 : Owner 30 バイト ZSTRING	キー 1 : Account 8 バイト INTEGER	Balance (キーではない) 8 バイト	Comments (キーではない) 1,000 バイト
John Q. Smith	263512477	1024.38	解説

以下に、Insert オペレーションの例を示します。ここで、データ バッファ長は固定長部分に Comments フィールド内のデータ量 (8 バイト) を加えた長さとして計算されることに注意してください。加えるのは、Comments フィールドの最大サイズ (1,000 バイト) ではありません。

Insert オペレーション

```
#define MAX_COMMENT 1000 /* コメントの最大サイズ */
typedef struct
{
    char owner[30];
    int  number;
    int  balance;
} FixedData;
typedef struct
{
    FixedData fix;
    char      variable[MAX_COMMENT];
} DataBuffer;

DataBuffer account;
BTI_ULONG  dataBufLen;
BTI_SINT   status;

strcpy(account.fix.owner, "John Q. Smith");
account.fix.number = 263512477;
account.fix.balance = 102438;
strcpy (account.variable, "Comments");
dataBufLen = sizeof(FixedData) + strlen(account.variable) +1;
/* +1 はデータの後にヌル文字用 */
status = BTRV(B_INSERT, PosBlock, &account, &dataBufLen, keyBuffer, 0);
```

固定長部分の読み取りおよび更新

データ バッファ サイズを固定長に設定することによって、レコードの固定長部分だけを読み取ることが出来ます。**MicroKernel** は、固定長部分とステータス コード 22 だけを読み取りますが、そのとき **Update** オペレーションを使用し、固定長部分だけを渡すと、可変長部分は失われます。その代わりに、**Update Chunk** オペレーション (53) を使用すると、バイト オフセットと長さに基づいてレコードの一部が更新されます。バイト オフセットを

0 に設定し、その長さを固定長部分の長さに設定します。このオペレーションは固定長部分を更新し、可変長部分を残します。

変更不可キーの更新

変更不可と定義されているキー値を更新しようとする、MicroKernel はステータス コード 10 を返します。いずれにしてもキー値を変更する場合は、まずレコードを Delete (4) し、次にレコードを Insert (2) する必要があります。

No-Currency-Change (NCC) オペレーション

キー番号パラメーターに -1 (0xFF) を渡すことによって、No-Currency-Change (NCC) オペレーションと呼ぶ、標準の Insert または Update のバリエーションを実行することができます。NCC オペレーションは、アプリケーションが Get Next オペレーション (6) などの別のオペレーションを実行するために元の論理位置をファイルに保存しておく必要がある場合に有効です。

NCC Insert オペレーションを使用せずに同じ効果を実現するには、以下の手順を実行する必要があります。

- 1 Get Position (22) –現在の論理レコードの 4 バイト物理アドレスを取得します。手順 3 で使用するためにこの値を保存します。
- 2 Insert (2) –新しいレコードを挿入します。このオペレーションにより、新しい論理カレンシーおよび物理カレンシーが確立されます。
- 3 Get Direct/Record (23) –論理カレンシーと物理カレンシーが手順 1 の時点の状態になるよう、これらを再確立します。

NCC Insert オペレーションは論理カレンシーで標準の Insert と同じ効果がありますが、物理カレンシーでは別の効果があります。たとえば、これら 2 つの手順のいずれかに続けて Get Next (6) オペレーションを実行した場合は、どちらの手順でも結果は変わりませんが、Step Next (24) を実行した場合は、異なるレコードが返される可能性があります。

NCC Update オペレーションを使用せずにオリジナルの位置を保つには、以下の手順を実行する必要があります。

- 1 Get Next (6) –一次の論理レコードを確立します。
- 2 Get Position (22) –一次の論理レコードの 4 バイトの物理アドレスを取得します。手順 8 で使用するためにこの値を保存します。
- 3 Get Previous (7) –現在の論理レコードを再確立します。
- 4 Get Previous (7) –直前のレコードを確立します。
- 5 Get Position (22) –直前の論理レコードの 4 バイトの物理アドレスを取得します。手順 8 で使用するためにこの値を保存します。
- 6 Get Next (6) –現在の論理レコードを再確立します。
- 7 Update (3) –影響を受けたレコードを更新します。この標準の Update オペレーションが指定されたキーの値を変更する場合、新しい論理カレンシーも確立します。
- 8 Get Direct/Record (23) –手順 7 で更新されたレコードの前または後のレコードにカレンシーを設定します。アプリケーションが前方検索を続ける場合は、手順 2 で保存されたアドレスを Get Direct/Record オペレーションに渡します。アプリケーションが後方検索を続ける場合は、手順 5 で保存されたアドレスを渡します。

マルチレコードのオペレーション

PSQL は、複数のレコードまたは複数のレコードの一部にフィルタリングを行い、データを返すのに、高いパフォーマンスのメカニズムを提供します。このメカニズムは、Extended オペレーションと呼ばれ、4 つの特別なオペレーション コードでサポートされます。

- Get Next Extended (36)
- Get Previous Extended (37)
- Step Next Extended (38)
- Step Previous Extended (39)

これらのオペレーション のコードの記述方法については、『Btrieve API Guide』を参照してください。このセクションでは、これらのオペレーションを最高のパフォーマンスで使用するために、最適化する方法を説明します。

用語

次の語はほかの状況では別の意味を持つことがあります。このセクションでの用途では、次のような定義になります。

ディスクリプター

拡張式とも呼ばれます。Btrieve Extended オペレーションをどのように実行するかを示す、データ バッファ全体の内容です。

フィルター

拡張式の一部は選択されるレコードに適用される選択式を表します。

条件

単一の論理演算子を使用するフィルターの一部です。

コネクタ

後続の条件を接続する論理演算子です。AND、OR、または NONE のいずれかになります。

エクストラクタ

拡張式の一部で、どのデータを返すかを定義します。

キー

インデックス定義全体で、複数のセグメントを含むこともできます。Get オペレーションでは、MicroKernel がデータ ファイル内を単一のキー パスで移動することが必要です。

キー セグメント

複合インデックス、またはマルチセグメント キーとも呼ばれ、複数のセグメント定義ができます。各セグメントには、オフセット、長さ、データ型などを定義します。

背景

Extended オペレーションに対するフィルターの評価メカニズムは、非常に高速に設計されています。余計な処理を行わず直接的な方法で式を評価します。このアプローチにより、インターフェイスが独特な方法をとっていることに気付かれることでしょう。

- Extended オペレーションは初期位置を確立しません。現在の位置から前または後に移動するだけです。したがって、条件 (lastname = 'Jones' AND firstname = 'Tom' AND city = 'LasVegas') に該当するすべてのレコードを検索するためには、アプリケーションは Get Next Extended オペレーションを実行する前に Get Equal オペレーションを実行する必要があります。
- フィルターの評価は厳密に左から右へと行われます。たとえば、アプリケーションは、単一の Extended オペレーションを実行して、条件 ((Age = 32 AND Gender = "M") OR (Age = 30 AND Gender = "F")) を満たすすべてのレコードを取得することはできません。

この種の検索では、最も効率を上げるためにはファイルの中であちこち移動する必要がありますが、Extended オペレーションではこのような移動は行いません。Extended オペレーションでは、論理キーまたはレコードパスに沿って一度に1レコード移動します。上に挙げたような複合論理式は、PSQLの一部であるリレーショナル エンジンにあるような最適化プログラムを必要とします。その代わりに、MicroKernel は式を左から右へと評価し、フィルターの評価を可能な限り高速に行います。

上記の検索を行うには、呼び出しプログラムは2つの Get Extended 呼び出しを行います。それぞれの呼び出しの前に GetEqual オペレーションを実行してカーソルを先頭レコードに位置付けてから実行します。1つの Extended オペレーションで、上の例の4つの条件を使用する場合、(Age = 32 AND (Sex = "M" OR (Age = 30 AND Sex = "F"))) のように評価されます。言い換えると、MicroKernel は各レコードで最初の条件を評価し、次に論理演算子に注目して、次の条件を評価する必要があるかどうかを判断します。最初の条件が False の場合、AND 演算子は式全体が False であることを意味します。

検証

Extended オペレーションがステータス コード 62 (無効なディスクリプター) を返す場合がたくさんあります。最も一般的なものを以下に挙げます。

- ディスクリプターの長さが不十分です。これは、フィルター条件の数、各条件の長さ、ACS や ISR が使用されていれば何を使用しているか、および抽出するフィールドの数によって異なります。
- データ バッファ。長さは、ディスクリプターを完全に含むことができる長さが最低限必要です。
- 各条件は有効なデータ型である必要があります。Btrieve のキー タイプの1つである必要があります。
- 比較コードに使用するフラグは有効なもの (FILTER_NON_CASE_SENSITIVE、FILTER_BY_ACS、FILTER_BY_NAMED_ACS) である必要があり、文字列型フィールド (STRING_TYPE、LSTRING_TYPE、ZSTRING_TYPE、WSTRING_TYPE、WZSTRING_TYPE) にしか使用できません。
- 有効な比較コード (1-6) である必要があります。
- 有効なコネクタ (0-2) である必要があります。
- 参照する ACS や ISR はあらかじめ定義されている必要があります。
- 最後のフィルター条件には終端文字 (コネクタが 0) が必要です。
- ほかのすべてのフィルター条件は終端文字を含めてはいけません (コネクタが 1 または 2)。
- エクストラクタ レコード カウントが 0 ではありません。

最適化

Extended オペレーションの最適化というのは、現在のキー パスにフィルター条件を満たすレコードが残っている可能性がない場合に、MicroKernel がそのキー パスを使用してレコードを探すのをやめることができる、ということです。PSQL 2000i SP3 以降、MicroKernel は複数の条件で最適化を行うことができます。ただし、それらの条件が順次発生し、現在のキーのセグメントに合致している場合に限りです。

各条件の評価時に、MicroKernel は指定されたセグメントが最適化可能かどうかを決定します。最適化を実行するには、以下のすべてが真である必要があります。

- Step Extended オペレーション (38 または 39) ではなく、Get Extended オペレーション (36 または 37) である。
- 前のセグメントの評価時に一致するレコードが見つかったため、残りのセグメントの最適化を無効にできない。
- OR コネクタによって、現在の条件と後続のすべての条件が最適化不能になる。これは式が左から右へ評価されるためです。
- 条件は、現在のキー セグメントと同じレコード内のオフセットを参照している。
- 条件は以下の要件を満たす必要があります。
 - レコード内で、現在のキー セグメントと同じフィールド長を参照している。
 - または、データ型が文字列型の1つである場合は、キーの部分文字列である。
- 条件はキーと同じフィールド型を参照している。

レコードの処理

- 条件で、フィールドをレコード内の別のフィールドと比較していない (FILTER_BY_FIELD)。
- 条件とキーの大文字小文字の区別の設定が同じである
- データ型が文字列型の 1 つである場合、条件はキーと同じ ACS または ISR を持っている。
- 条件が最初のキー セグメントで最適化されない場合を除き、論理演算子は EQ (=) である。
- 最初のキー セグメントの場合、論理演算子には、方向が前方の場合は LT または LE (< または <=)、逆方向の場合は GT または GE (> または >=) も使用できます。これらの論理演算子では、1 つのフィルター条件のみが最適化されます。

実際の方向はオペレーションによって示される方向だけでなく、現在のキーが降順か昇順かにも左右されます。

表 33 Extended オペレーションの実際の方向

	昇順のキー セグメント	降順のキー セグメント
Get/Step Next	昇順 / 前方	降順 / 後方
Get/Step Prev	降順 / 後方	昇順 / 前方

以降のすべてのセグメントに対する最適化は、以下のいずれかによって無効にされます。

- これ以上最適化するセグメントがキーにない場合。
- 現在の条件に OR コネクタがある場合。
- 現在の条件は最適化されたが、条件が関連するキー セグメントの部分文字列である場合。
- 現在の条件は最適化されたが、論理演算子が EQ (=) でない場合。
- 現在の条件は最適化されず、前の条件が最適化された場合。つまり、複数の条件が AND で結合されている場合、最初の最適化条件となる最初のキー セグメントに一致するものは、フィルター内の最初の条件である必要はないということです。ただし、最初のキー セグメントに最適化できる条件がいったん見つかると、そのほかの最適化条件は最初の最適化条件の直後に続いて発生する必要があります。



メモ PSQL 2000i SP3 では、最適化条件をフィルター内で最初に発生させる必要があるという不具合がありました。したがって、最適化されない条件を最適化条件の前に置くという機能は SP3 より後でのみ有効です。この機能は SP3 より前でも有効でしたが、最適化することができるのは 1 つの条件だけでした。

例

後述の例では次のサンプル データを使用します。

表 34 マルチレコード オペレーション用のサンプル データ

レコード	Field 1	Field 2	Field 3	Field 4
1	AAA	AAA	AAA	XXX
2	AAA	BBB	BBB	OOO
3	AAA	CCC	CCC	XXX
4	BBB	AAA	AAA	OOO
5	BBB	AAA	BBB	XXX
6	BBB	AAA	CCC	OOO
7	BBB	BBB	AAA	XXX
8	BBB	BBB	BBB	OOO

表 34 マルチレコード オペレーション用のサンプル データ

レコード	Field 1	Field 2	Field 3	Field 4
9	BBB	BBB	CCC	XXX
10	BBB	CCC	AAA	OOO
11	BBB	CCC	BBB	XXX
12	BBB	CCC	CCC	OOO
13	CCC	AAA	CCC	XXX
14	CCC	CCC	AAA	OOO

上のテーブルは、Field 1、Field 2、および Field 3 がこの順で複合キーであるとします。アプリケーションでは、このファイルに対し、前述の 3 つのフィールドから成るセグメント キーを使用して GetFirst オペレーションを実行し、引き続き GetNextExtended オペレーションを実行します。以下の例には想定されるかっこが含まれていることに注意してください。フィルターが左から右へ評価される時、かっこはこの位置にのみ出現します。

キー セグメントに対して最適化を行うためには、フィルター条件に指定したオフセット、長さ、データ型、大文字小文字の区別、および ACS はキー定義と同一である必要があることを忘れないでください。

(Field1 = AAA AND (Field2 = AAA AND (Field3 = AAA)))

MicroKernel エンジン はレコード 1 を取得し、ステータス コード 64 (最適化限度を超えた) で検索を停止します。最適化条件を満たす最後に調べたレコードはレコード 1 で、これが現在のレコードになります。Pervasive.SQL 2000 SP3 より前のエンジンでは、1 つの条件しか最適化することができなかったため、現在のレコードはレコード 3 のままでした。

(Field1 = AAA OR (Field2 = AAA OR (Field3 = AAA)))

MicroKernel エンジン はレコード 1、2、3、4、5、6、7、10、13 および 14 を取得し、ステータス コード 9 (ファイルの終わりに達した) を返します。最初の条件に OR コネクタが含まれているため、どの条件も最適化されません。現在のレコードは、レコード 14 になります。

(Field1 = BBB AND (Field2 = BBB OR (Field3 = BBB)))

MicroKernel エンジン はレコード 5、7、8、9 および 11 を取得し、ステータス コード 64 を返します。OR コネクタが含まれていないため、最初の条件が最適化されましたが 2 番目の条件は最適化されませんでした。最適化条件を満たす最後に調べたレコードはレコード 12 で、これが現在のレコードになります。

(Field4 = OOO AND (Field2 = BBB AND (Field3 = BBB)))

MicroKernel エンジン はレコード 2 と 8 を取得し、ステータス コード 9 を返します。最初のキー セグメントに対してはどの条件も最適化されず、したがって次のセグメントも最適化されません。現在のレコードは、レコード 14 になります。

(Field1 = BBB AND (first byte of Field2 = B AND (Field3 = BBB)))

MicroKernel エンジン はレコード 8 を取得し、ステータス コード 64 を返します。最初の 2 つの条件は最適化されますが、2 番目の条件がサブ文字列であるため、3 番目の条件は最適化されません。最適化条件を満たす最後に調べたレコードは、レコード 9 です。Pervasive.SQL 2000 SP3 より前のエンジンは 1 つの条件しか最適化できなかったため、レコード 12 が現在のレコードになります。

(Field1 = BBB AND (Field2 = Field3))

これは、2 番目のオペランドが定数ではなくレコードの別のフィールドであることを示す、+64 のバイアスを比較コードに使用して行われます。MicroKernel エンジン はレコード 4、8、および 12 を取得し、ステータス コード 64 を返します。最初の条件は最適化されますが、2 番目の条件は定数と比較されないため最適化されません。最適化条件を満たす最後に調べたレコードは、レコード 12 です。

(Field1 <= BBB AND (Field2 <= BBB AND (Field3 <= BBB)))

MicroKernel エンジン はレコード 1、2、4、5、7 および 8 を取得し、ステータス コード 64 を返します。最初の条件は最適化されますが、論理演算子 OR を含まないため次の条件は最適化されません。最適化条件を満たす最後に調べたレコードは、レコード 12 です。

(Field1= BBB AND (Field2 < BBB AND (Field3 < BBB)))

MicroKernel エンジンではレコード 4 を取得し、ステータス コード 64 を返します。最初の条件は最適化されますが、2 番目の条件は論理演算子 EQ を含まないため最適化されません。最適化条件を満たす最後に調べたレコードは、レコード 12 です。

(Field1= BBB AND (Field2 = BBB AND (Field3 < BBB)))

MicroKernel エンジンではレコード 7 を取得し、ステータス コード 64 を返します。最初の 2 つの条件は EQ を使用しているため最適化されますが、3 番目の条件は最適化されません。最適化条件を満たす最後に調べたレコードは、レコード 9 です。Pervasive.SQL 2000 SP3 より前のエンジンは 1 つの条件しか最適化できなかったため、レコード 12 が現在のレコードになります。

(Field2>= AAA AND (Field2 <= BBB AND (Field1 >= AAA) AND (Field1 <= BBB)))

MicroKernel エンジンではレコード 1、2、4、5、6、7、8 および 9 を取得し、ステータス コード 64 を返します。最初の 3 つの条件は最初のキー セグメントに対して最適化されませんが、これらはすべて AND で結合されているため、4 番目の条件は検索を最適化するのに使用されます。2 番目の条件は、4 番目の条件の直後に発生すれば最適化することができます。しかし、キー セグメントに関連する位置がずれているため、最適化されません。1 つのキー セグメントのみが最適化されるため、最適化条件を満たす最後に調べたレコードは、レコード 12 になります。Pervasive.SQL 2000 SP3 には不具合があり、最適化可能な条件が最初に発生しない場合には最適化が妨げられます。したがって、SP3 エンジンでは同じレコードを取得しますが、ステータス コード 9 を返します。

パフォーマンスのヒント

このセクションでは、オペレーションの速度を上げる方法に関して説明します。

コネクタ

Extended オペレーションは論理式を左から右へ評価するため、この機能は、最も効率的な方法で必要なデータを抽出するために式を評価するのに使用することはできません。Extended オペレーションは、最初にカーソルをファイル内の適切な位置に設定して **Get** または **Step** オペレーションと共に使用するよう設計されています。したがって、最初に提案できることは次のようになります。

- 1 つのフィルター内に AND と OR コネクタを混在させないようにします。もしそうする場合は、キーのセグメントに一致するように AND 条件を最初に置き、少なくともエンジンがキー パスの少ない部分で検索を最適化できるようにします。

言い換えると、インデックスにないフィールドに対する OR で結合する条件は、インデックス付きのフィールドに対する最適化可能な条件の後に追加するのが適切です。たとえば、全国の電話帳から、テキサス州に住む "William"、"Bill"、"Billy" または "Billybob" という名前の人をすべて検索するとします。State フィールドのキーを使用し、GetEqual を使用してテキサス州の最初の人に現在のレコードを設定します。次に (State = "Texas" AND (FirstName = "William" OR (FirstName = "Bill" OR (FirstName = "Billy" OR (FirstName = "Billybob"))))) のようなフィルターを使って GetNextExtended を呼び出します。エクストラクタのリジェクト カウントが 10,000 で検索する最大レコード件数が 100 の場合、GetNextExtended はおよそ 10,000 レコードを見た後制御を戻します。しかし、テキサス州の 14,000,000 人を処理するには、ユタ州に到達してステータス コード 64 (フィルター制限に達しました) が返されるまで、同じ GetNextExtended オペレーションを何度も何度も繰り返す必要があります。この処理は各レコードを 1 度に 1 つずつアプリケーションに転送するよりはずっと速く行われます。

しかし、State と FirstName に複数のインデックスが存在した場合はどうでしょうか。上の GetNextExtended は動作はしますが、4 つの州と FirstName の組み合わせのそれぞれで GetEqual および GetNextExtended を行って両方のフィールドで最適化を行えば、ずっと速く処理できます。

したがって、インデックスが使用できない場合は、OR コネクタを使ったフィルターを使用することが唯一有効な方法であることがわかります。キーに一致するフィールドに対する AND コネクタが優先されます。

リジェクト カウント

もう 1 つ理解する必要のある問題は、リジェクト カウントの設定方法です。使用するアプリケーションが MicroKernel エンジンのサービスを受ける唯一のアプリケーションである場合は、ネットワーク トラフィックまたは内部的な通信処理を最低限に保つことができるため、最大のリジェクト カウントを使用することが最も効果

的です。ただし、並行処理が高い頻度で行われる環境でたくさんのアプリケーションが実行される場合は、リジェクト カウントの設定が大きすぎると重大な結果を招きます。

MicroKernel は、1 つのファイルに複数のアトミックな **Btrieve** オペレーションを実行中であっても、同時にたくさんの **Btrieve** 要求を処理することができます。したがって、読み取りスレッドがいくつでも許容されるだけでなく、1 つの書き込みスレッドが同じファイルにアクセスできます。ほとんどの **Btrieve** オペレーションは読み取り操作で、それらは実行に多くの時間を要しません。そのため、書き込み操作を行う場合は、すべての読み取り処理が終わるのを待ってから、レコードの挿入、更新、または削除を行う瞬間にファイルをロックします。この調整は、完了するのに長い時間がかかる読み取り操作を行わない限り、非常によく機能します。リジェクト カウントの値を大きく設定した **Extended** オペレーションで、レコードが見つからない場合にそうなります。これは読み取り処理を何度も何度も繰り返します。ほかの読み取り操作は問題なく完了しますが、書き込み操作は停滞し始めます。書き込みオペレーションがファイルに対して 100 回書き込みアクセスを試行すると、フラストレーション カウントと呼ばれる回数に達します。この時点で、書き込み操作はすべての新しい読み取りスレッドにブロックをかけます。その結果、このファイルに対するすべての **Btrieve** オペレーションは **Extended** オペレーションが完了するまで停止します。

- このため、並行処理が高い頻度で行われる環境で使用されるアプリケーションの場合は、リジェクト カウントは 100 から 1000 の間で使用してください。また、**Extended** オペレーションの最適化をするようにして、**MicroKernel** がレコードの読み取りと拒否を頻繁に行わなくてもよいようにしてください。

100 から 1000 のリジェクト カウントを使用したとしても、レコードをアプリケーションに戻してから拒否するよりも、**MicroKernel** に読み取りと拒否を実行させる方がよいです。

キーの追加と削除

Btrieve は、ファイルのキーを追加および削除するための操作を 2 つ用意しています。Create Index (31) と Drop Index (32) です。Create Index オペレーションは、ファイルが作成された後にファイルにキーを追加する場合に有効です。Drop Index オペレーションは、インデックス ページが損傷しているキーを削除する場合に有効です。キーを削除した後に、そのキーを再度追加することができますが、そうすると、MicroKernel によってインデックスが再構築されます。

キーを削除すると、特に他の指定をしない限り、MicroKernel はそのキーよりも大きな番号を持つすべてのキーの番号を付け替えます。MicroKernel は、より大きな番号を持つすべてのキーから 1 を引くことによって、キー番号の付け替えを行います。たとえば、キー番号 1、4、および 7 を持つファイルがあるとしましょう。キー 4 を削除すると、MicroKernel はキーの番号を 1 および 6 と付け直します。

MicroKernel によってキー番号を自動的に付け替えられたくない場合は、バイアス値 0x80 をキー番号パラメーターに設定する値に加算します。これにより、キー番号に空きを残しておくことができ、その結果、ファイル内のほかのキー番号に影響を及ぼすことなく、損傷したキーを削除し、そのキーを再構築することができます。インデックスを再構築するには、Create Index オペレーション (31) を使います。このオペレーションではキー番号を指定できます。



メモ 番号の付け替えを指示しないでキーを削除した場合、その後でユーザーが具体的なキー番号を割り当てずに影響を受けたファイルを複製すると、複製したファイルには元のファイルとは異なるキー番号が割り当てられます。

複数のクライアントのサポート

9

この章では、以下の項目について説明します。

- 「[Btrieve クライアント](#)」
- 「[受動的並行性 \(パッシブ コンカレンシー\)](#)」
- 「[レコードのロック](#)」
- 「[ユーザー トランザクション](#)」
- 「[複数並行制御ツールの例](#)」
- 「[複数ポジションブロックの並行制御](#)」
- 「[複数ポジションブロック](#)」
- 「[クライアント ID パラメーター](#)」

Btrieve クライアント

Btrieve クライアントとは、Btrieve 呼び出しを行うアプリケーション定義のエンティティです。各クライアントは Btrieve 呼び出しを実行でき、MicroKernel に登録されているファイルなどのリソースを個々に持ちます。また、MicroKernel はクライアントごとに排他トランザクションと並行トランザクションの両方の状態を保持します。

複数のクライアントを同時にサポートしなければならない場合は、パラメーターとしてクライアント ID を含む BTRVID 関数または BTRCALLID 関数を使用します。クライアント ID パラメーターは、MicroKernel がコンピューター上のクライアントを区別できるようにする 16 バイト構造体のアドレスです。以下に、クライアント ID を使用することが有効である場合の例を示します。

- すべて同時に処理されるいくつかのトランザクションを実行するマルチスレッド型アプリケーションを作成します。アプリケーションでは、Begin Transaction オペレーションごとに異なるクライアント ID を指定します。MicroKernel は、クライアント ID 別にトランザクションの状態を保持します。
- 2 つのクライアント ID を使用し、クライアント ID ごとにいくつかのファイルを開くアプリケーションを書きます。アプリケーションは BTRVID、BTRCALLID、または BTRCALLID32 を使って Reset オペレーションを実行できるので、MicroKernel は指定された 1 つのクライアント ID のファイルを閉じて、リソースを解放することができます。
- アプリケーション自体の複数のインスタンスが同時に動作できるようなアプリケーションを書きます。アプリケーションのデータの整合性をとるために、MicroKernel にはすべてのインスタンスが 1 つのクライアントであるように見えなければなりません。この場合、アプリケーションのどのインスタンスがその呼び出しを行っているかに関係なく、アプリケーションは各 Btrieve 呼び出しで同じクライアント ID パラメーターを提供します。
- Dynamic Data Exchange (DDE) サーバーの役割を果たすアプリケーションを書きます。Btrieve 呼び出しを行うサーバー アプリケーションは、サーバー アプリケーションへ要求を発信するアプリケーション間で返された情報を分配する必要があります。この場合、アプリケーションは各要求元のアプリケーションに異なるクライアント ID を割り当てることにより、複数のクライアント間に配信される情報を追跡するための手段を提供できます。

MicroKernel はいくつかの並行制御方法を提供し、いくつかの実装ツールを使用して、複数のクライアントが同じファイル内のレコードを並行してアクセスまたは変更しようとする場合に発生する可能性のある競合を解決します。

並行制御方法は以下のとおりです。

- 「[受動的並行性 \(パッシブ コンカレンシー\)](#)」
- 「[レコードのロック](#)」
- 「[ユーザー トランザクション](#)」

実装ツールは以下のとおりです。

- 明示的レコード ロック
- 暗黙レコード ロック
- 暗黙ページ ロック
- ファイル ロック

以降では、MicroKernel の並行制御方法について詳しく説明します。各セクションを読むとき、表 35 を参照してください。この表では、2 つのクライアントが同じファイルのアクセスまたは変更を行おうとする場合に発生する可能性のある競合のタイプをまとめています。この表 35 では、ローカル クライアントのアクションを説明しています。



メモ アプリケーションが BTRVID 関数を使用して同じアプリケーション内で複数のクライアントの定義と管理を行う場合、そのようなクライアントはローカル クライアントと見なされます。

この表では、クライアント 1 は表の左端の列の略語によって識別されるアクションを実行し、次にクライアント 2 が表の上端の行の略語によって識別されるアクションのうちの 1 つを**実行しよう**とします。

これらの略語によって表されるアクションについては、「**アクションコード**」で説明しています。

前提条件

表 35 は以下のことを前提としています。

- 表の特定のセルでは、クライアント 2 はクライアント 1 がアクションの実行を開始した後にアクションを実行しようとしています。最初のアクションが終了しないと、第 2 のアクションは開始できません。
- クライアント 2 が更新または削除操作を実行するセルの場合、クライアント 2 はクライアント 1 がアクションを実行する前に、影響を受けるレコードをあらかじめ読み取っているものとします。
- アクション MDR および MTDR のようにクライアント 2 のアクションが明示的に記述されていない限り、双方の操作が読み取りであっても、更新または削除であっても、クライアント 1 と 2 は常に同じレコードに対してアクションを実行します。
- クライアント 2 のアクションが明示的に記述されていない限り（アクション ITDP を参照してください）、クライアント 1 と 2 がともに挿入、更新、または削除操作を実行する場合、両クライアントは共通するページのうち少なくとも 1 ページを変更します。
- クライアント 1 の挿入操作の後でクライアント 2 が変更を実行する場合、変更されるレコードは挿入されたレコードではありませんが、両レコードはファイル内の 1 つ以上のデータ ページ、インデックス ページ、または可変ページを共有しています。

アクションコード

RNL	トランザクションでない処理または並行トランザクションにおける、ロック要求を伴わない読み取り。
RWL	トランザクションでない処理または並行トランザクションにおける、ロック要求を伴う読み取り。
INT	トランザクションでない処理における挿入。
ICT	並行トランザクションにおける挿入。
ITDP	並行トランザクションにおける挿入。同様に並行トランザクション内にいるクライアント 1 が挿入、更新、または削除によって変更したページとは異なるページを変更する。
MNT	トランザクションでない処理における変更（更新または削除）。
MDR	トランザクションでない処理における変更。クライアント 1 によって変更されたレコードとは異なるレコードを変更する。
MCT	並行トランザクションにおける変更。
MTDR	並行トランザクションにおける変更。クライアント 1 によって変更されたレコードとは異なるレコードを変更する。
EXT	排他トランザクションにおける読み取り、挿入または変更。

競合コード

適用外	適用されません（該当なし）。
NC	クライアント 1 とクライアント 2 のアクション間に競合またはブロックはありません。
RB	レコード レベルのブロック。クライアント 2 は、クライアント 1 でかけられたレコード ロックのためにブロックされます。

複数のクライアントのサポート

PB	ページレベルのブロック。クライアント 2 は、クライアント 1 でかけられたページ ロックのためにブロックされます。
FB	ファイルレベルのブロック。クライアント 2 は、クライアント 1 でかけられたファイル ロックのためにブロックされます。
RC	レコードの競合。初めクライアント 2 が読み取っていたレコードを後からクライアント 1 が変更したため、クライアント 2 は操作を実行できません。MicroKernel は、ステータス コード 80 を返します。

競合コード RB、PB、および FB の場合、クライアント 2 がノーウェイト タイプの操作（たとえば、ノーウェイト ロックを指定した読み取りや、500 バイアスを指定して開始された並行トランザクションでの挿入または変更など）を指定していない限り、MicroKernel はクライアント 2 のアクションを再試行します。ノーウェイトの操作の場合は、MicroKernel はエラー ステータス コードを返します。

表 35 ローカル クライアントに関して発生する可能性のあるファイル操作の競合

クライアント 2 のアクション										
	RNL	RWL	INT	ICT	ITDP	MNT	MDR	MCT	MTDR	EXT
クライアント 1 のアクション										
RNL	NC	NC	NC	NC	適用外	NC	適用外	NC	適用外	NC
RWL	NC	RB	NC	NC	適用外	RB	適用外	RB	適用外	RB
INT	NC	NC	NC	NC	適用外	NC	適用外	NC	適用外	NC
ICT	NC	NC	PB	PB	NC	PB	適用外	PB	適用外	PB
MNT	NC	NC	NC	NC	適用外	RC	NC	RC	NC	NC
MCT	NC	RB	PB	PB	NC	RB	PB	RB	PB	PB
EXT	NC	FB	FB	FB	適用外	FB	FB	FB	FB	FB

以下に、表 35 でアクション コードの組み合わせを解釈するための例を示します。

- EXT – RWL の組み合わせ。クライアント 1 は、排他トランザクション内からファイルのレコードを読み取ります。クライアント 2 は、トランザクションでない処理モードから、ノーウェイト ロック バイアスを指定してファイルからレコードを読み取ろうとすると、ステータス コード 85 (FB、ファイル レベルのブロック) を受け取ります。クライアント 2 がウェイト ロック バイアスを指定した場合、MicroKernel は操作を再試行します。
- ICT – ICT の組み合わせ。クライアント 1 は、並行トランザクション内からレコードを挿入します。クライアント 2 が同じファイルにレコードを挿入しようとした場合、MicroKernel は操作を再試行します。これは、この挿入操作で変更されるページのうちの 1 つが、クライアント 1 で実行された挿入操作によって既に変更されているからです。クライアント 2 が 500 バイアスを指定した並行トランザクションを開始した場合、MicroKernel はステータス コード 84 を返します（この表については、「[前提条件](#)」を参照してください）。
- ICT – ITDP の組み合わせ。この組み合わせは、クライアント 1 によって変更されたページをクライアント 2 が変更しないという点を除いては、ICT – ICT に似ています。この場合、クライアント 2 で試行された操作は正常終了します (NC、ブロックなし、競合なし)。
- MCT – MTDR の組み合わせ。クライアント 1 とクライアント 2 は異なるレコードを変更しますが、クライアント 2 はページ ロックでブロックされます。このブロックが発生するのは、変更されるレコードがファイル内のデータ ページ、インデックス ページまたは可変ページを共有しているからです（この表については、「[前提条件](#)」を参照してください）。

受動的並行性（パッシブ コンカレンシー）

アプリケーションが、トランザクション モード外または並行トランザクション内から単一レコードの読み取り操作および更新操作を実行する場合は、受動的並行性に依存して更新の競合を解決することができます。受動的並行性は MicroKernel によって自動的に適用されるものであり、アプリケーションまたはユーザーからの明示的な指示を必要としません。

受動的並行性の状態にある場合、MicroKernel はクライアントが操作にロック バイアスを適用しなくてもレコードを読み取れるようにします。最初のクライアントがレコードを読み取ってから、そのレコードの更新または削除を試みるまでの間に、2 番目のクライアントがそのレコードを変更した場合、MicroKernel はステータス コード 80 を返します。この場合、最初のクライアントが行う変更は、レコードの古いイメージに基づいています。そのため、最初のクライアントは更新または削除操作を実行する前に、レコードを再度読み取る必要があります。

受動的並行性により、開発者はわずかな変更だけでシングルユーザー環境からマルチユーザー環境へアプリケーションを直接移動できます。

表 36 と表 37 は、受動的並行性を使用している場合に 2 つのクライアントがどのように相互作用するかを示しています。トランザクションでない場合と並行トランザクションからの場合を示します。

表 36 受動的並行性（トランザクションでない処理の例）

クライアント 1	クライアント 2
1. ファイルを開きます。	
	2. ファイルを開きます。
3. レコード A を読み取ります。	
	4. レコード A を読み取ります。
5. レコード A を更新します。	
	6. レコード A を更新します。MicroKernel は、ステータス コード 80 を返します。
	7. レコード A を再度読み取ります。
	8. レコード A を更新します。

表 37 受動的並行性（並行トランザクションの例）

クライアント 1	クライアント 2
1. 並行トランザクションを開始します。	
	2. 並行トランザクションを開始します。
3. レコード A を読み取ります。	
4. レコード A を更新します。	
	5. レコード A を読み取ります。
6. トランザクションを終了します。	
	7. レコード A を更新します。MicroKernel は、競合ステータス コードを返します。
	8. レコード A を再度読み取ります。

表 37 受動的並行性（並行トランザクションの例）

クライアント 1	クライアント 2
	9. レコード A を更新します。
	10. トランザクションを終了します。



メモ クライアント 1 がレコード A の更新操作を既に実行した**後**でクライアント 2 がレコード A を読み取ったとしても、MicroKernel は手順 7 で競合エラーを正しく検出します。この競合が存在するのは、クライアント 1 が手順 6 でトランザクションを終了するまでレコード A に対して行った変更をコミットしないからです。クライアント 2 が手順 7 で更新を実行するときまでに、手順 5 で読み取ったレコード A のイメージは古くなってしまいます。

レコードのロック

多くの場合、クライアントは受動的並行性で実現されるものより強力な並行性制御を必要とします。そのため、**MicroKernel** では、クライアントが競合エラーを受け取らずにレコードの更新または削除を行えるようにすることができます。このエラーはステータス コード 80 で、このアプリケーションがレコードを読み取った後に別のクライアントがそのレコードを変更したことを示します。これを実現するには、クライアントはロックを要求してレコードを読み取る必要があります。**MicroKernel** がロックを許可した場合、ロックをかけたクライアントがロックを解除するまで、ほかのクライアントはレコードのロック、更新、削除は行えません。

したがって、操作が一時的にブロックされるためにクライアントが待機しなければならない場合があっても、レコードを更新または削除する機能は保証されます（たとえば、クライアントのレコードと同じデータ ページ上の別のレコードが、まだ実行されている並行トランザクション内の別のアプリケーションで変更される場合に、一時的なブロックが発生する可能性があります）。

クライアントはさまざまな種類のレコード ロックを明示的に要求できます。詳細については、「[ロック](#)」を参照してください。

ユーザー トランザクション

トランザクションによって、データが消失する可能性が低くなります。ファイルに加える変更が多く、また、これらの変更が確実に**すべて行われるか、まったく行われない**ようにする必要がある場合は、1つのトランザクションにこれらの変更用の操作を取り込みます。明示的なトランザクションを定義すれば、MicroKernel に複数の Btrieve オペレーションを1つのアトミック単位として処理させることができます。トランザクション内にオペレーション群を取り込むには、Begin Transaction オペレーション (19) と End Transaction オペレーション (20) でこれらのオペレーションを囲みます。

MicroKernel には、排他トランザクションと並行トランザクションの2種類のトランザクションがあります。どのタイプを使用するかは、変更するファイルに対するほかのクライアントからのアクセスをどのくらい厳しく制限するかにより決まります (MicroKernel では、ほかのアプリケーションやクライアントはトランザクションが終了するまで、どのような排他トランザクションまたは並行トランザクションにかかわる変更でも見ることはできません)。

タスクが排他トランザクション内のファイルで動作する場合、MicroKernel はトランザクションの期間中にファイル全体をロックします。ファイルが排他トランザクションでロックされると、ほかの非トランザクション クライアントはファイルを読み取れますが、変更することはできません。排他トランザクション内にいる別のクライアントも、最初のクライアントがトランザクションを終了してファイルのロックを解除するまで、ファイルのポジションブロックを必要とするオペレーション、たとえば、標準の Get オペレーションや Step オペレーションを実行できません。

アプリケーションが並行トランザクション内でファイルに操作をした場合、MicroKernel は以下のように、ファイル内で影響を受けるレコードとページのみをロックします。

- Get オペレーションまたは Step オペレーションが明確的に読み取りロック バイアスを指定して呼び出された場合、あるいは Begin Transaction オペレーションから読み取りロック バイアスを継承した場合、MicroKernel は Get オペレーションまたは Step オペレーションにおける1つまたは複数のレコードをロックします (ロックとロック バイアスについては、「[ロック](#)」を参照してください)。
- MicroKernel は、Insert、Update または Delete オペレーションで変更されるデータ ページ上のレコードをロックします。また、レコードが可変長レコードである場合、MicroKernel はそのレコードの各部分を含むすべての可変ページをロックします。最後に、MicroKernel は、Insert、Update または Delete オペレーションの結果として変更されるインデックス ページのエントリをロックします。インデックス ページの一部分の変更によって、エントリが別のページに移動することがあります。インデックス ページが分割または結合される場合が、その例です。これらの変更では、トランザクションが完了するまでインデックス ページの完全なページロックが保持されます。

排他トランザクションの場合と同様に、ほかのタスクは常に同時トランザクション内からロックされるデータを読み取ることができます。どのようなデータ ファイルでも、複数のタスクがそれぞれの並行トランザクションを操作して、トランザクション内で Insert、Update または Delete オペレーションを実行したり、読み取りロック バイアスを含む Get オペレーションまたは Step オペレーションを実行できます。これらの動作の唯一の制限は、2つのタスクが個々の並行トランザクションから同じレコードまたはページを同時にロックできないということです。

並行トランザクションには、以下の追加機能が適用されます。

- ロックされたページは、トランザクションの期間中ロックされたままになります。
- トランザクションで単にレコードを読み取るだけの場合、MicroKernel はレコードも対応するページもロックしません。
- ほかのクライアントは、並行トランザクション内でファイルに加えられた変更について、トランザクションが終了するまで認識できません。

ロック

レコード、ページまたはファイル全体でもロックできます。いったんロックされたら、そのロックに関与するクライアント以外は誰もレコード、ページ、またはファイルを変更できません。同様に、あるクライアントが所有

するロックは、以降で説明するように、別のクライアントによるレコード、ページまたはファイルのロックを防ぐことができます。

MicroKernel には、明示的ロックと暗黙ロックの 2 種類のロックがあります。クライアントが Btrieve オペレーションコードにロック要求を含めることによって明確にロックを要求する場合、そのロックを**明示的**ロックと呼びます。しかし、たとえクライアントが明示的にロックを要求しない場合でも、MicroKernel はクライアントが実行した動作の結果として影響を受けたレコードまたはページをロックできます。この場合、MicroKernel が行うロックを**暗黙**ロックと呼びます。（「[暗黙レコード ロック](#)」と「[暗黙ロック](#)」を参照してください。）



メモ 特に注記がない限り、**レコード ロック**は**明示的**レコード ロックを意味します。

レコードは、暗黙または明示的にロックできます。ページは暗黙にしかロックできません。ファイルは明示的にしかロックできません。

以降では、トランザクションでない処理環境とトランザクション環境の両方で適用される各種ロックについて説明します。

トランザクションでない処理環境における明示的レコード ロック

ここでは、トランザクションでない処理環境における明示的レコード ロックについて説明します。トランザクションがレコード ロックの使用にどのような影響を与えるかについては、「[並行トランザクションのレコード ロック](#)」を参照してください。

クライアントは、受動的並行性に依存しない方がよい場合があります。しかし、その同じクライアントが、クライアントにレコードの再読み取りを要求するステータス コード 80 を受け取らずに、読み取ったレコードを後で更新または削除できるようにしなければならない場合があります。クライアントはレコードに対する明示的レコード ロックを要求することによって、これらの要件を満たすことができます。アプリケーションがレコードの読み取り時にレコードをロックする場合、以下のバイアス値のうちの 1 つを対応する Btrieve Get または Step オペレーションコードに追加できます。

- 100 – 単一ウェイト レコード ロック
- 200 – 単一ノーウェイト レコード ロック
- 300 – 複数ウェイト レコード ロック
- 400 – 複数ノーウェイト レコード ロック

これらのロック バイアスは、Get オペレーションと Step オペレーションにのみ適用できます。トランザクションでない処理環境では、ほかのどのオペレーションにもロック バイアスを指定できません。



メモ 単一レコード ロックと複数レコード ロックには互換性がありません。したがって、クライアントは、ファイル内の同じポジション ブロックまたはカーソルに同時には両方のタイプのロックをかけることはできません。

単一レコード ロック

単一レコード ロックでは、クライアントは一度に 1 つのレコードしかロックできません。クライアントが単一レコード ロックを使って正常にレコードをロックしている場合、クライアントが以下のイベントのいずれかを完了するまでそのロックは有効です。

- ロックされたレコードを更新または削除する。
- 単一レコード ロックを使用してファイル内の別のレコードをロックする。
- Unlock オペレーション (27) を使用して明示的にレコードをロック解除する。
- ファイルを閉じる。
- Reset オペレーション (28) を発行して、開いているすべてのファイルを閉じる。

複数のクライアントのサポート

- 排他トランザクション中にファイル ロックを行う。

1 人のクライアントがレコードをロックすると、ほかのクライアントはそのレコードに対して Update (3) または Delete (4) オペレーションを実行できません。ただし、Get または Step オペレーションが以下の条件に従ってさえいれば、ほかのクライアントはこれらのオペレーションを使用してレコードを読み取ることができます。

- 明示的ロック バイアスが含まれていない。
- レコードを読み取るとそのレコードがロックされるようなトランザクションから実行されない。これは、たとえば MicroKernel がファイル全体をロックする排他トランザクションや、ロック バイアスで開始された並行トランザクションなどです。詳細については、「[並行トランザクションのレコード ロック](#)」と「[ファイル ロック](#)」を参照してください。

複数レコード ロック

複数レコード ロックを使用すると、クライアントは同じファイル内でいくつかのレコードを並行してロックできます。クライアントが複数レコード ロックを使って 1 つまたは複数のレコードを正常にロックしている場合、クライアントが以下のイベントのうち 1 つ以上を完了するまでそれらのロックは有効です。

- ロックされたレコードを削除する。
- Unlock オペレーション (27) を使用して明示的にレコード ロックを解除する。
- ファイルを閉じる。
- Reset オペレーション (28) を発行して、開いているすべてのファイルを閉じる。
- 排他トランザクション中にファイル ロックを行う。



メモ Update オペレーションは、複数レコード ロックを解除しません。

単一レコード ロックの場合と同様に、クライアントが複数レコード ロックで 1 つまたは複数のレコードをロックすると、ほかのクライアントはこれらのレコードに対して Update (3) または Delete (4) オペレーションを実行できません。「[ロック](#)」で説明しているように、ほかのクライアントは Get または Step オペレーションを使用して、これまでどおりロックされたレコードを読み取ることができます。

レコードが既にロックされている場合

別のクライアントによってレコードがロックされている、または、排他トランザクションによってファイル全体がロックされているために現在使用できないレコードに対し、クライアントがノーウェイト ロックを要求した場合、MicroKernel はステータス コード 84「レコードまたはページはロックされています」またはステータス コード 85「ファイルはロックされています」を返します。クライアントがウェイト ロックを要求し、そのレコードが現在使用できない場合、MicroKernel はオペレーションを再試行します。

並行トランザクションのレコード ロック

排他トランザクション (オペレーション 19) はファイル全体をロックするので、トランザクション内のレコード ロックは並行トランザクション (オペレーション 1019) にしか適用されません (トランザクションの種類の詳細については、「[ユーザー トランザクション](#)」を参照してください)。

MicroKernel では、クライアントは並行トランザクション内からファイル内の単一レコードまたは複数レコードをロックすることができます。クライアントは、以下の方法のいずれかでレコードをロックできます。

- 前にリストしたバイアス値のうちの 1 つを使用して、Get または Step オペレーションでレコード ロック バイアスを明示的に指定する (並行トランザクションのレコード ロック バイアス値は、非トランザクション レコード ロックのバイアス値と同じです)。
- Begin Concurrent Transaction オペレーション (1019) でレコード ロック バイアス値を指定する (この場合も、これらのバイアス値は、前にリストした非トランザクション レコード ロックのバイアス値と同じです)。

Begin Concurrent Transaction オペレーションでレコード ロック バイアス値を指定すると、そのトランザクション内の各オペレーションは、独自のバイアス値を持たない場合には、**Begin Concurrent Transaction** オペレーションからバイアス値を継承します。たとえば、**Get Next** オペレーション (06) が、先に実行されたバイアスのかけた **Begin Concurrent Transaction** オペレーション (1219) から 200 バイアスを継承するとしたら、その **Get Next** はノーウェイト ロックの読み取り操作 (206) として実行されます。

前に示したように、クライアントは並行トランザクション内で発生する個々の **Step** または **Get** オペレーションにバイアス値を追加することができます。この方法で追加されたバイアスは、継承されたバイアスより優先します。

並行トランザクションで単一レコード ロックと複数レコード ロックを解除するイベントは、トランザクションでない処理環境のイベントに似ています。単一レコード ロックについては、「**単一レコード ロック**」を参照してください。複数レコード ロックについては、次の例外を除き、「**複数レコード ロック**」を参照してください。

- **Close** オペレーションは、並行トランザクション内から設定された明示的レコード ロックを解除しない。**MicroKernel** のバージョン 7.0 では、たとえレコードがロックされていてもトランザクション内でファイルを閉じることができます。
- **End Transaction** または **Abort Transaction** オペレーションは、トランザクション内から得られたすべてのレコード ロックを解除する。

最後に、並行トランザクション内のクライアントがバイアスのかかっていない **Get** または **Step** オペレーションを使用して 1 つまたは複数のレコードを読み取る場合、**Begin Concurrent Transaction** オペレーションでロック バイアスが指定されていなければ、**MicroKernel** はロックを行いません。

暗黙レコード ロック

クライアントがトランザクションの外部または並行トランザクション内からレコードの更新または削除を行おうとすると、**MicroKernel** はクライアントの代わりにそのレコードを暗黙にロックしようとします。排他トランザクションで、暗黙のレコード ロックが不要なのは、**MicroKernel** が **Update** または **Delete** オペレーションを実行する前にファイル全体をロックするからです（「**ファイル ロック**」を参照してください）。

MicroKernel は、ほかのクライアントが以下の操作を行わない限り、クライアントに対して暗黙レコード ロックを与えることができます。

- レコードに明示的ロックをかける。
- レコードに暗黙ロックをかける。
- レコードを含んでいるファイルをロックする。



メモ **MicroKernel** では、単一のクライアントが同じレコードに対して明示的ロックと暗黙ロックの両方をかけることができます。

MicroKernel は、オペレーションの実行中、ファイルの整合性を確保するために必要な暗黙レコード ロックおよびその他すべてのロックを正常に取得できる場合のみ、指定された **Update** または **Delete** オペレーションを実行します（「**暗黙ロック**」を参照してください）。

オペレーションがトランザクションでない処理環境にある場合、**MicroKernel** は **Update** または **Delete** オペレーションの終了時に暗黙レコード ロックを解除します。オペレーションが並行トランザクションにある場合、**MicroKernel** はロックを維持します。その場合、クライアントがトランザクションを終了または中止するか、クライアントがリセットされる（これは **Abort Transaction** オペレーションを意味する）まで、ロックは有効です。暗黙レコード ロックを解除するために使用できる明示的な **Unlock** オペレーションはありません。

トランザクション中、暗黙ロックを維持することで、**MicroKernel** は別のクライアントが生成した新しいコミットされていないイメージがレコードに含まれている場合に、クライアントがロック バイアス値を指定した **Get** または **Step** オペレーションを介してそのレコードを明示的にロックする結果生じる競合を防ぐことができます。

MicroKernel が暗黙ロックを維持しなかった場合にどうなるかを考えてみましょう。クライアント 1 は並行トランザクション内からレコード A で更新を行うことによって、レコードのイメージを変更します。しかし、クライアント 1 は並行トランザクションを終了していないので、新しいイメージをコミットしていません。クライアント 2 は、レコード A を読み取ってロックしようとします。

複数のクライアントのサポート

暗黙ロックが維持されていなかったら、レコード A に対するクライアント 1 の暗黙レコード ロックはなくなっているため、クライアント 2 はレコードを正常に読み取ってロックできてしまいます。しかし、クライアント 1 が新しいイメージをコミットしていないので、クライアント 2 はレコード A の古いイメージを読み取ります。クライアント 1 がレコード A の変更されたイメージをコミットするオペレーションを終了し、クライアント 2 がレコード A を更新しようとする、そのレコードのクライアント 2 のイメージは無効となるので、MicroKernel はステータス コード 80 「MicroKernel でレコード レベルの矛盾が発生しました」を返します（表 38 の例を参照してください）。

クライアントがレコードを明示的または暗黙にロックしたか、そのレコードを含むファイル全体をロックした場合を考えてみましょう。別のクライアントが並行トランザクション内から問題のレコードの更新または削除を行おうとした場合、つまり、レコードを暗黙にロックしようとした場合、MicroKernel の実装のいくつかは待機し、ロックをかけてオペレーションをブロックしているクライアントがそのロックを解除するまで引き続きオペレーションを再試行します（どのバージョンの MicroKernel も、非トランザクションの更新または削除に対する再試行作業は試みません）。

Begin Concurrent Transaction オペレーションでバイアス値 500 を指定する（1519）と、MicroKernel はトランザクション内で Insert、Update および Delete オペレーションを再試行しなくなります。

ローカル クライアントの場合、MicroKernel はデッドロック検出を行います。ただし、バイアス 500 は再試行を抑止するので、MicroKernel はデッドロック検出を行う必要はありません。

Begin Transaction オペレーションでは、このバイアス値 500 をレコード ロックのバイアス値と組み合わせることができます。たとえば、 $1019 + 500 + 200$ （1719）を使用すると、Insert、Update および Delete オペレーションの再試行が抑止されると同時に、単一レコードの読み取りノーウェイト ロックが指定されます。

以下の例は、暗黙ロックの有効性を示したものです。この例では、暗黙ロックが存在しないと一時的に仮定しています。

表 38 暗黙ロックのない例

クライアント 1	クライアント 2
1. 並行トランザクションを開始します。	
2. レコード A を読み取ります。	
3. レコード A を更新します（関連するページをロックする、ただし、レコードに対する暗黙ロックなし）。	
	4. 単一レコード ロック（レコードの明示的ロック）を指定してレコード A を読み取ります。
5. トランザクションを終了します（ページ ロックを解除する）。	
	6. レコード A を更新します（競合、ステータス コード 80）。
	7. ロックを指定してレコード A を再度読み取ります。
	8. レコード A を更新します。

MicroKernel が手順 3 で暗黙レコード ロックを適用しないと仮定した場合、クライアント 2 は手順 4 でレコード A を正常に読み取り、ロックできるにもかかわらず、手順 6 でそのレコードを更新できません。これは、手順 4 でクライアント 2 がレコード A の有効なイメージを読み取っても、手順 6 に達するまでにそのイメージが有効でなくなってしまうからです。手順 5 で、クライアント 1 がレコード A の新しいイメージをコミットすることによって、手順 4 でクライアント 2 が読み取ったレコードのイメージを無効にします。

しかし、実際は、MicroKernel が手順 3 でレコード A を暗黙にロックします。つまり、MicroKernel は手順 4 でステータス コード 84 を返し、クライアント 1 が手順 5 を実行するまで読み取り操作を再試行するようにクライアント 2 に要求します。

前の例で手順 3 と 4 を逆にした場合にどうなるかを考えてみましょう。クライアント 2 は、レコード A に暗黙ロックをかけます。クライアント 1 は待機させられ、クライアント 2 が自身で読み取ったレコード A の更新を終えて、そのレコードに対する明示的ロックを解除するまで、Update オペレーションを再試行します。クライアント 1 が次にレコード A の更新を再試行すると、MicroKernel はステータスコード 80 を返します。このステータスは、クライアント 1 のレコード A のイメージが有効ではなくなったこと、つまり、クライアント 2 がレコード A を変更する前にクライアント 1 がそのレコードを読み取っていたことを示します。

暗黙ロック

複数のクライアントがファイルを同時に変更できるという大きな自由度があるのは、同じ MicroKernel でキャッシュを共有するからです。非トランザクションの変更 (Insert、Update または Delete オペレーション) は、ほかの非トランザクションの変更または別のクライアントによる並行トランザクションでの変更をブロックしません。並行トランザクション内で保留になっている変更は、その変更が同一レコードに影響を与えない限り、ほかの非トランザクションの変更も並行トランザクションにおける変更もブロックしません。

トランザクションの外側または並行トランザクションの中から Insert、Update、または Delete オペレーションが発生する場合は、MicroKernel がクライアントの代わりに、これらの操作中に変更されるレコードを暗黙にロックしようとします (排他トランザクションでは、MicroKernel は Update や Delete オペレーションを実行する前にファイル全体をロックするため、暗黙のレコード ロックおよびページ ロックは必要ありません。Insert オペレーションの場合は、クライアントがまだファイルをロックしていなければ MicroKernel がファイル ロックを要求します。「[ファイル ロック](#)」を参照してください)。暗黙レコード ロックと同様に、暗黙ページ ロックは MicroKernel エンジンが行うため、クライアントは明示的に要求しません。

変更または挿入が行われるデータ ページ上のレコードは常にロックされます。しかし、単一のオペレーションがいくつかのほかのレコードもロックしなければならない場合があります。たとえば、レコードに加えた変更が 1 つ以上のレコード キーに影響を与える場合、MicroKernel は影響を受けたキー値を含んでいるインデックス ページのレコードをロックする必要があります。また、MicroKernel は、オペレーション中に B ツリーのバランスを取る作用によって変更されるすべてのインデックス ページをロックする必要があります。変更がレコードの可変長部分に影響を与える場合、MicroKernel は可変ページ全体もロックする必要があります。

そのようなオペレーションがトランザクションでない処理環境で実行される場合、MicroKernel はオペレーションの終了時に暗黙レコード ロックを解除します。オペレーションが並行トランザクション内から行われる場合、MicroKernel はロックを保持します。その場合、クライアントがトランザクションを終了または中止するまで、あるいは、クライアントがリセットされる (これは Abort Transaction オペレーションを意味する) まで、ロックは有効です。明示的 Unlock オペレーションを使用して、暗黙レコード ロックや暗黙ページ ロックを解除することはできません。

並行トランザクションで発行された Insert、Update、または Delete オペレーションでレコードまたはページの変更が必要になった場合 (暗黙ページ ロックを必要とします) に、そのレコードまたはページが別の並行トランザクションによって現在ロックされているか、ファイル全体が排他トランザクションによってロックされている場合、MicroKernel は待機し、ロックをかけてオペレーションをブロックしているクライアントがそのロックを解除するまで引き続きオペレーションを再試行します。MicroKernel は、非トランザクションの更新や削除に対する再試行を試みません。

暗黙レコード ロックを取得できない場合は、クライアントは Begin Concurrent Transaction オペレーションでバイアス値 500 を使用することによって、オペレーションの再試行を抑止することができます。

暗黙ページ ロックと明示的レコード ロックまたは暗黙レコード ロックは、互いにブロックに影響を与えません。クライアントは、操作対象のレコードが入っているページを別のクライアントが暗黙にロックしていたとしても、そのページ上のレコードを読み取り、ロックすることができます。ただし、これは「[暗黙レコード ロック](#)」で説明しているように、ロックするレコードが更新されたレコードと同じでない場合に限りです。逆に、クライアントはレコードを更新または削除した場合、それによって影響を受けるレコードが入っているデータ ページの中に、別のクライアントによってロックされたレコードが既に含まれていても、そのデータ ページを暗黙にロックすることができます。

ファイル ロック

クライアントは排他トランザクション内で初めてファイルにアクセスするとき、ファイル ロックの取得を試みます。



メモ 前の文章が示すように、MicroKernel はクライアントが **Begin Transaction** オペレーションを実行する際にはファイルをロックしません。ロックが発生するのは、**Begin Transaction** オペレーションの実行後、クライアントがレコードを読み取ったり変更したりするときだけです。

ファイル ロックは、その名前が示すように、ファイル全体をロックします。クライアントのファイル ロックは、そのクライアントがトランザクションを終了または中止するか、そのクライアントがリセットされる（これは **Abort Transaction** オペレーションを意味する）まで有効です。

クライアントが排他トランザクションでファイルをロックしようとしたとき、既に別のトランザクションがそのファイルにロック（レコード、ページ、または ファイル ロック）をかけていた場合には、MicroKernel は待機し、ロックをかけてオペレーションをブロックしているクライアントがそのロックを解除するまで引き続きオペレーションを再試行します。また、ローカル クライアントがオペレーションをブロックし、MicroKernel がデッドロックの状況を検出すると、MicroKernel はステータス コード 78「MicroKernel でデッドロックを検出しました。」を返します。

ファイル ロックを取得できない場合は、クライアントは **Begin Exclusive Transaction** オペレーションでノーウェイト ロック バイアス値 200 または 400（219 または 419）を指定することによって、オペレーションの再試行を抑制することができます。このような方法でクライアントがトランザクションを開始したとき、MicroKernel はファイル ロックを与えられない場合には、ステータス コード 84 または 85 を返します。

バイアス値 200 および 400 は、レコード ロックから派生したものです。しかし、レコード ロック環境における単一ロックおよび複数ロックという概念は、排他トランザクション環境では何も意味しません。事実上、ファイルがロックされるときにファイル内のすべてのレコードがロックされます。排他トランザクション環境では、バイアスの「ノーウェイト」の意味だけが残されます。

MicroKernel は、ウェイト ロック バイアス（100 または 300）を **Begin Exclusive Transaction** オペレーションで受け付けます（それぞれオペレーション 119、319 になります）が、**Begin Transaction** オペレーションのデフォルトモードはウェイト モードであるため、このようなバイアス値の加算には意味がありません。

排他トランザクションでは、ファイルのどの部分にでも最初にアクセスが生じたら、MicroKernel はファイル全体をロックします。そのため、MicroKernel は以下の例外を除き、排他トランザクション内で実行される **Get** または **Step** オペレーションのオペレーション コードに明示的に追加されたレコード ロック バイアス値を無視します。

クライアントが **Begin Transaction** オペレーションをウェイト モード（オペレーション 19、119、または 319）で実行しているときに、そのトランザクションの最初の読み取り（**Get** または **Step** オペレーション）にバイアス 200 または 400（ノーウェイト ロック バイアス）をかけた場合、ノーウェイト バイアスが **Begin Transaction** オペレーションのウェイト モードよりも優先されます。そのため、クライアントがこのバイアスのかかった読み取り操作を実行してもファイルをロックできない場合、たとえば、別のクライアントがそのファイルのレコードを既にロックしている場合は、MicroKernel は待機せず（デフォルト）、デッドロックの有無を確認しません。これは、クライアントが読み取り操作の再試行を無制限に行うことを前提としているからです。これと同じ状況で、再試行を実行する MicroKernel のほかのバージョンも、ファイル ロックを再試行したりデッドロックの有無を確認しないという指示として、ノーウェイト バイアスを自動的に認識します。



メモ 排他トランザクション内から実行される **Get** または **Step** オペレーションに対するバイアス値 200 と 400 は、待機しないという意味しかありません。つまり、並行トランザクション内からの場合と同様に、それらの値は明示的なレコード ロックを要求しません。

ファイル ロックは、レコード ロックとページ ロックのどちらも両立しません。したがって、別のクライアントがファイルにレコード ロックまたはページ ロックをかけている場合、MicroKernel はクライアントにそのファイルのロックを与えません。逆に、別のクライアントが既にファイルにロックをかけている場合、MicroKernel はクライアントにレコード ロックまたはページ ロックを与えません。

複数並行制御ツールの例

以下の例は、さまざまな並行制御機構の使用例を示したものです。

例 1

例 1 は、明示的レコード ロック、暗黙レコード ロック、明示的ページ ロック、受動的並行性の相互関係を示したものです。この例で操作される 2 つのレコード、レコード A とレコード B は、同じデータ ページに存在し、ファイルにはキーが 1 つだけあるものとします。各手順の詳細説明については、例の後を参照してください。

次の表は、暗黙レコード ロック、明示的レコード ロック、暗黙ページ ロック、受動的並行性の相互関係を示したものです。

表 39 レコード ロック、ページ ロック、並行性の相互関係

クライアント 1	クライアント 2	クライアント 3（非トランザクション）
1. 複数ノーウェイト ロック バイアスを指定して並行トランザクションを開始する（1419）		
	2. 単一ウェイト ロック バイアスを指定して並行トランザクションを開始する（1119）	
3. 単一ノーウェイト ロック バイアスを指定した Get Equal を使用してレコード A を読み取る（205）		
	4. Get Equal（5）を使用してレコード B を読み取る（単一ウェイト ロック バイアスを継承）	
		5. Get Equal（5）を使用してレコード B を読み取る
		6. レコード B の削除（4）を試みる：MicroKernel がステータス コード 84 を返すため、クライアント 3 は再試行しなければならない
	7. レコード B を更新する（3）	
8. レコード A の更新（3）を試みる：MicroKernel が再試行しなければならない		
	9. トランザクションを終了する（20）	
10. レコード A の更新（3）を再試行する：正常終了		
		11. レコード B の削除（4）を再試行する：MicroKernel がステータス コード 80 を返すため、クライアント 3 はレコード B を再度読み取らなければならない
		12. Get Equal（5）を使用してレコード B を再度読み取る

表 39 レコード ロック、ページ ロック、並行性の相互関係

クライアント 1	クライアント 2	クライアント 3 (非トランザクション)
		13. レコード B の削除 (4) を再試行する : MicroKernel はステータス コード 84 を返す
14. トランザクションを終了する		
		15. レコード B の削除 (4) を再試行する : 正常終了

手順 1 で、クライアント 1 の **Begin Concurrent Transaction** オペレーションには一般的なバイアス値 400 (複数レコード ノーウェイト ロック) を指定します。このバイアスは、このトランザクション内にあるバイアスのかかっていない各 **Get** または **Step** オペレーションに継承されます。この時点では、MicroKernel はファイル、ファイルのページまたはレコードにロックを適用していません。

手順 2 で、クライアント 2 の **Begin Transaction** オペレーションには一般的なバイアス値 100 (単一レコード ウェイト ロック) を指定します。このバイアスは、このトランザクション内にあるバイアスのかかっていない各 **Get** または **Step** オペレーションに継承されます。MicroKernel はまだ、ファイル、ファイルのページまたはレコードにロックを適用していません。

手順 3 で、クライアント 1 の **Get Equal** オペレーションはバイアス値 200 (単一レコード ノーウェイト ロック) を指定しています。MicroKernel は、継承された 400 (複数ノーウェイト レコード ロック) ではなくこのバイアス値を受け入れます。これは、継承されたバイアス値よりも、個々のオペレーションに指定されたバイアス値が優先されるからです。

手順 4 で、クライアント 2 の **Get Equal** オペレーション (5) は独自のバイアス値を指定していません。したがって、このオペレーションはクライアント 2 の **Begin Concurrent Transaction** オペレーション (1119) から単一ウェイト ロック バイアス値 100 を継承します。たとえレコード A とレコード B が同じページ上にあっても、双方のロック要求 (手順 3 と手順 4) は成功します。なぜなら、レコード ロック要求は指定されたレコードだけをロックするものであり、そのレコードが位置するデータ ページも、関連するどのインデックス ページもロックしないからです。

手順 5 で、クライアント 3 のロックを要求しない非トランザクションの **Get Equal** オペレーション (5) は成功します。これは、要求したレコードが存在する限り、非トランザクションの読み取りは常に成功するからです。

手順 6 で、クライアント 3 はレコード B の削除 (4) を行おうとしますが、クライアント 2 がそのレコードに明示的ロックをかけているので、レコード B に対してレコードの削除に必要な暗黙レコード ロックを取得できません。その結果、MicroKernel はクライアント 3 にステータスコード 84 (レコードまたはページがロックされている) を返します。そうすると、クライアント 3 は制御を放棄して、必要であれば後で **Delete** オペレーションを再試行する必要があります。

手順 7 で、クライアント 2 はまずレコード B に対する暗黙レコード ロックの取得に成功します。レコード B は、手順 4 で手順 2 から単一ウェイト ロック バイアスを継承するため、既にクライアント 2 で明示的にロックされていますが、明示的ロックと暗黙ロックの両方が同じクライアントに属しているので、問題はありません。それと同時に、クライアント 2 は、レコード B を含んでいるデータ ページとレコード B のキー値を含んでいるインデックス ページに対するページ ロックの取得にも成功します。



メモ クライアント 2 によってロックされたデータ ページには、クライアント 1 で明示的にロックされたレコード A が含まれていますが、「**暗黙ロック**」で説明しているように、レコード ロックはページ ロックをブロックしません。

MicroKernel は手順 7 で **Update** オペレーション (3) そのものを実行するときに、変更されたデータ ページとインデックス ページの**コミット**されていない新しいイメージをシャドウ ページとしてファイルに書き込みます。この時点で、MicroKernel はレコード B に対するクライアント 2 の明示的ロックを解除しますが、クライアント 2 は今し方取得した暗黙ページ ロックに加えて、レコード B の暗黙レコード ロックも保持します。クライアント 2

が手順 7 で Update オペレーション (3) を終了した後も、クライアント 3 がまだレコード B に対する暗黙レコード ロックを取得できないのは、クライアント 2 がレコードに対する暗黙レコード ロックを今も保持しているからです。クライアント 3 は、その再試行を続ける必要があります。

クライアントがリモートである場合、クライアント 2 は実際にファイルを更新する前に、クライアント 2 のローカル クライアント間の並行制御に必要なページ ロックを設定するほか、ファイルを保留中の変更の状態とします。

手順 8 で、クライアントはまずレコード A に対する暗黙レコード ロックをうまく取得します。たとえレコード A のデータ ページが既にクライアント 2 でロックされていても、ページ ロックはレコード ロックをブロックしないので、ロックの競合はありません（「[暗黙ロック](#)」を参照してください）。次に、クライアント 1 はレコード A を含んでいるデータ ページの暗黙ページ ロックを取得しようとします。この試行は、データ ページが手順 7 でクライアント 2 によって既にロックされているため失敗します。Begin Concurrent Transaction オペレーション (1419) は 500 バイアスを指定していないため、MicroKernel はオペレーションを再試行します（クライアントがローカルである場合、MicroKernel はデッドロック検出も行います）。

もしもクライアント 1 が 500 バイアスを追加して Begin Transaction オペレーション (1919) を発行していたら、MicroKernel は直ちにユーザーに制御を返したでしょう。

クライアントがリモートである場合、クライアント 1 は手順 7 でクライアント 2 によって設定された保留中の変更の状態に直面するため、MicroKernel はオペレーションを再試行します。

手順 9 で、そのトランザクションを終了することによって、クライアント 2 はレコード B に対する暗黙レコード ロックを解除し、手順 5 でロックしたデータ インデックス ページに対する暗黙ページ ロックを解除します。この時点で、MicroKernel はクライアント 2 がトランザクションで作成した新しいページ イメージをすべてコミットします。これで、これらのイメージがファイルの有効な部分になります。

クライアントがリモートである場合、クライアント 2 はファイルに対する保留中の変更の状態をクリアすると共に、ロックを解除します。

手順 10 で、クライアント 1 が続行している更新の再試行がようやく成功するのは、クライアント 2 がレコード A のデータ ページとインデックス ページをロックしなくなったからです。

手順 11 で、クライアント 2 が手順 9 でトランザクションを終了し、それによってすべてのロックを解除したにもかかわらず、クライアント 3 はまだレコード B を削除できません。現状では、クライアント 3 がレコードを削除しようとする、MicroKernel の受動的並行制御はステータス コード 80（競合）を返します。これは、クライアント 3 が手順 5 で最初にレコード B を読み取った後で、クライアント 2 がこのレコードを変更したからです。この時点で、クライアント 3 はレコードを再度読み取らないと、Delete オペレーションを再試行できません。

手順 12 で、クライアント 3 はレコード B を再度読み取ることにより、クライアント 2 が手順 7 でレコードに加えた変更を反映し、手順 9 でコミットされたイメージを取得します。

手順 13 で、クライアント 3 はレコード B を削除しようとしても再びうまくいかず、MicroKernel からステータス コード 84 が返されます。このステータス コードは、クライアント 1 がレコード A を更新するために、レコード B を含むデータ ページとインデックス ページの暗黙ページ ロックを持っているという事実を反映しています。最初に述べたように、同じデータ ページにレコード A とレコード B が含まれており、同じインデックス ページにこれらのレコードのキー値が含まれているものと仮定します。

手順 14 で、クライアント 1 はトランザクションを終了することによって、変更をコミットし、暗黙ページ ロックを解除します。

手順 15 で、クライアント 3 はようやくレコード B を削除することができます。

例 2

例 2 は、ファイル ロックと受動的並行制御の相互関係を示したものです。各手順の詳細説明については、例の後を参照してください。

表 40 ファイル ロックと受動的並行性の相互関係

クライアント 1	クライアント 2
1. ファイル 1 を開く (0)	
2. ファイル 2 を開く (0)	
3. ファイル 3 を開く (0)	
4. 単一レコード ロックを使用してファイル 3 のレコード E を取得する (105)	
	5. ファイル 1 を開く (0)
6. 排他トランザクションを開始する (19)	
7. ファイル 1 のレコード B を取得する (5)	
	8. ファイル 1 のレコード A を取得する (5)
	9. ファイル 1 のレコード A を更新する (3) (ステータス コード 85、再試行)
10. ファイル 2 のレコード C を取得する (5)	
11. ファイル 2 のレコード C を更新する (3)	
12. ファイル 1 のレコード B を削除する (4)	
13. トランザクションを終了する (20)	
	14. 手順 9 を再試行する (成功)

手順 4 で、クライアント 1 はファイル 3 のレコード E に対する明示的レコード ロックを取得します。

手順 6 で、クライアント 1 は排他トランザクションを開始します。クライアント 1 が 3 つのファイルを開いていますが、MicroKernel はこれらのファイルをまだロックしていませんし、Begin Transaction オペレーションを実行した結果として、ファイル 3 のレコード E に対する明示的ロックを解除することはありません。

手順 7 で、クライアント 1 はファイルにアクセスした結果としてファイル 1 に対するファイル ロックを取得します（「[ファイル ロック](#)」を参照してください）。もし、直前の手順（たとえば手順 5 と 6 の間）でクライアント 2 がロック バイアスを持つオペレーションを使用してファイル 1 からレコードを読み取っていたら、手順 7 は失敗します。

手順 8 で、クライアント 2 はファイル 1 からレコード A を正常に読み取ります。この読み取りが成功するのは、ロックを要求しないからです。ただし、Get Equal オペレーション (5) がロック バイアスを指定して発行されていたら、クライアント 1 がファイル 1 を現在ロックしているため、オペレーションは失敗します。

手順 9 では、クライアント 1 がファイルをロックしているため、クライアント 2 は暗黙レコード ロックを取得できません。したがって、MicroKernel はクライアント 2 にステータス コード 85（ファイルがロックされている）を返します。クライアント 2 は直ちに制御を放棄し、クライアント 1 が手順 13 でトランザクションを終了または中止するまで、手順 9 を再試行しなければなりません（「[レコードが既にロックされている場合](#)」を参照してください）。

手順 10 で、クライアント 1 はファイル 2 に対するファイル ロックを取得します。

手順 13 で、クライアント 1 はファイル 1 と 2 に対するファイル ロックを解除します。



メモ クライアント 1 は排他トランザクションでファイル 3 にアクセスしていないので、ファイル 3 をロックしませんでした。事実上、手順 13 以降でも、クライアント 1 はファイル 3 のレコード E に対する明示的レコード ロックを保持します。クライアント 1 は、ファイル 3 にアクセスしてトランザクションでファイル全体をロックしていた場合のみ、レコード E を解除したでしょう。

手順 14 で、ようやく手順 9 におけるクライアント 2 の Update オペレーションの再試行が成功します。

複数ポジション ブロックの並行制御

MicroKernel では、同じクライアントが同じファイル内で複数のポジション ブロック (カーソル) を使用できます。

並行トランザクションまたは排他トランザクションの内側では、複数ポジション ブロックは変更されたページの同じビューを共有します。複数ポジション ブロック セット内の各ポジション ブロックは、そのセットのほかのポジション ブロックによって行われた変更であれば、**変更がコミットされる前であっても**、直ちにわかります。

複数ポジション ブロックはすべてのロック、つまり、明示的レコード ロック、暗黙レコード ロック、暗黙ページ ロック、ファイル ロックを共有します。したがって、どのクライアントについても、あるポジション ブロックのロックによって別のポジション ブロックが同じファイル内で別のロックを取得できなくなることはありません。

クライアントがトランザクションを終了または中止すると、MicroKernel はそのクライアントの暗黙ロックとファイル ロックをすべて解除します。ただし、クライアントの明示的レコード ロックがトランザクション内から与えられたものであるかどうかにかかわらず、MicroKernel はファイル内の各ポジション ブロックが要求したときだけクライアントの明示的レコード ロックを解除します。

たとえば、キー値として -2 を持つ Unlock オペレーション (27) は、指定されたポジション ブロックに属する複数レコード ロックのみを解除します。Close オペレーション (1) は、その実行時に指定された同じポジション ブロックに対して取得されたロックのみを解除します。同様に、クライアントがトランザクション内でポジション ブロックの最初のレコード ロック、ページ ロック、またはファイル ロックを取得したとき、MicroKernel はそのポジション ブロックに対して取得していた明示的レコード ロックのみを解除します。「[例 2](#)」で、クライアント 1 がファイル 1、ファイル 2、ファイル 3 を開くのではなく、同じファイルを 3 回開いていた場合、また、クライアント 1 が最初の 2 つのポジション ブロックのみを使用してファイルにアクセスしていた場合、3 つ目のポジション ブロック用に取得した明示的ロックは、End Transaction オペレーション以降も変化しません。

複数ポジション ブロック

BTRV 関数を使用するアプリケーションで、同じファイルに2つのアクティブなポジションブロックを持ち、両方のポジションブロックから同じレコードに対して複数レコード ロックを指定した読み取りを発行した場合、どちらのポジションブロックも成功ステータスを受け取ります。ただし、キー番号に -1 とデータ バッファに物理位置を指定するか、キー番号に -2 を指定するかしてレコードのロックを解除する場合は、両方のポジションブロックがロック解除の呼び出しを発行した場合にのみ、レコードのロックは解除されます。1つのポジションブロックだけがロック解除呼び出しを行った場合（どの呼び出しかは問題ではない）、別のユーザーはレコードをロックしようとする、ステータスコード 84 を受け取ります。両方のポジションブロックがロック解除を発行した後、第2のユーザーはレコードをロックできます。

この動作は単一レコード ロックの場合にも当てはまりますが、この場合のロック解除コマンドは、キー番号の特定も、データ バッファへの物理位置の指定も必要としません。ただし、別のユーザーがレコードをロックするには、両方のポジションブロックがロック解除を発行する必要があります。

各カーソル（つまり、各ポジションブロック）はロックを取得します。MicroKernel では同じクライアントのカーソルが同じレコードをロックできますが、各カーソルがロック解除を発行しなければ、レコードのロックは完全に解除されません。

クライアント ID パラメーター

BTRV 関数でなく BTRVID 関数を使ってアプリケーションを開発する場合は、クライアント ID と呼ばれる追加パラメーターを指定する必要があります。これにより、アプリケーションは複数のクライアント ID を **Btrieve** に割り当てて、ほかのクライアントの状態に影響を与えることなく、1 人のクライアントのオペレーションを実行することが可能になります。

たとえば、2 つのアプリケーションが **Windows** で実行されており、そのアプリケーションそれぞれが 3 つの異なるクライアント ID を使用していると仮定します。この場合、アクティブ クライアント数は 6 個になります。これが同じアプリケーション（および各インスタンス内の同じクライアント ID の値）の 2 つのインスタンスであるか、2 種類のアプリケーションであるかは問題ではありません。**Btrieve** は、これら 6 つのクライアント ID をそれぞれ区別します。

Btrieve アプリケーションのデバッグ

10

この章では、Btrieve アプリケーションのデバッグに役立つと思われる情報を示します。以下の項目が含まれています。

- 「[トレース ファイル](#)」
- 「[クライアント / サーバー環境における間接的な Chunk オペレーション](#)」
- 「[エンジンのシャットダウンと接続のリセット](#)」
- 「[ファイル内の無駄な領域の削減](#)」

トレース ファイル

MicroKernel の [トレース オペレーションの実行] 設定オプションを選択すると、各 Btrieve API 呼び出しをトレースし、その結果をファイルに保存できます。これは、アプリケーションのデバッグに有効です。以下に、サンプルトレース ファイルを示します。

BUTIL STAT 呼び出しの MicroKernel トレース ファイル

```
MicroKernel Database Engine [Server Edition] for Windows NT trace file
Created : Wed Dec 17 18:19:09
<In> 0198 Opcode : 0026 Crs ID : 0xffffffff Db Length : 00005 Keynum : ff
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 00 00 00 00 00 - .....
KBuf: ?? - .
<Out>0198 Status : 0000 Crs ID : 0xffffffff Db Length : 00005 Keynum : ff
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 07 00 00 00 54 - ....T
KBuf: ?? - .
-----
<In> 0199 Opcode : 0000 Crs ID : 0xffffffff Db Length : 00001 Keynum : fe
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 4e 4f 54 53 48 4f 57 4e - 00 NOTSHOWN.
KBuf: 5c 5c 4e 54 34 53 52 56 - 2d 4a 55 44 49 54 5c 43 ¥¥NT4SRV-JUDIT¥C
24 5c 64 65 6d 6f 64 61 - 74 61 5c 74 75 69 74 69 $¥demodata¥tuiti
<Out>0199 Status : 0000 Crs ID : 0x00000002 Db Length : 00001 Keynum : fe
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 4e 4f 54 53 48 4f 57 4e - 00 NOTSHOWN.
KBuf: 5c 5c 4e 54 34 53 52 56 - 2d 4a 55 44 49 54 5c 43 ¥¥NT4SRV-JUDIT¥C
24 5c 64 65 6d 6f 64 61 - 74 61 5c 74 75 69 74 69 $¥demodata¥tuiti
-----
<In> 0200 Opcode : 0015 Crs ID : 0x00000002 Db Length : 00028 Keynum : fe
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 00 00 00 00 00 01 07 00 - 00 00 00 00 00 00 00 03 .....
c3 3f 00 10 00 00 00 00 - b4 fe 36 03 .?.....6.
KBuf: 00 00 00 00 1c 00 00 00 - da fe 36 03 00 00 00 00 .....6.....
00 01 07 00 00 00 00 00 - 00 00 00 03 c3 3f 00 10 .....?..
<Out>0200 Status : 0000 Crs ID : 0x00000002 Db Length : 00007 Keynum : fe
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 03 00 0e 00 04 05 01 - .....
KBuf: 00 00 00 00 07 00 00 00 - da fe 36 03 03 00 0e 00 .....6.....
04 05 01 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
-----
<In> 0201 Opcode : 0015 Crs ID : 0x00000002 Db Length : 33455 Keynum : ff
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 2b 00 cb ff ff ff ff ff - ff ff ff ff ff ff ff 00 +.....
00 0e 00 04 05 01 00 00 - 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 - 00 00 00 14 4e 54 34 53 .....NT4S
52 56 2d 4a 55 44 49 54 - 5c 75 6e 6b 6e 6f 77 6e RV-JUDIT¥unknown
KBuf: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
<Out>0201 Status : 0000 Crs ID : 0x00000002 Db Length : 00064 Keynum : ff
Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 0e 00 00 10 03 70 08 00 - 00 00 01 12 00 00 00 00 .....p.....
01 00 04 00 00 01 08 00 - 00 00 0f 00 00 00 00 00 .....
05 00 05 00 03 05 04 00 - 00 00 0a 00 00 00 01 00 .....
0a 00 01 00 03 01 02 00 - 00 00 00 00 00 00 02 00 .....
KBuf: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
```



```

-----
<In> 0202  Opcode : 0065  Crs ID : 0x00000002  Db Length : 00268  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      45 78 53 74 01 00 00 00 - 00 00 00 00 01 00 00 00  ExSt.....
           00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  .....
           00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  .....
           00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  .....
KBuf:      ?? - .
<Out>0202  Status : 0000  Crs ID : 0x00000002  Db Length : 00035  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      01 00 00 00 01 00 00 00 - 17 00 00 00 43 3a 5c 44  .....C:¥D
           45 4d 4f 44 41 54 41 5c - 54 55 49 54 49 4f 4e 2e  EMODATA¥TUITION.
           4d 4b 44 - MKD
KBuf:      ?? - .
-----
<In> 0203  Opcode : 0065  Crs ID : 0x00000002  Db Length : 00008  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      45 78 53 74 02 00 00 00 - ExSt....
KBuf:      ?? - .
<Out>0203  Status : 0000  Crs ID : 0x00000002  Db Length : 00008  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      00 00 01 00 00 00 07 00 - .....
KBuf:      ?? - .
-----
<In> 0204  Opcode : 0001  Crs ID : 0x00000002  Db Length : 00000  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      ?? - .
KBuf:      ?? - .
<Out>0204  Status : 0000  Crs ID : 0x00000000  Db Length : 00000  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      ?? - .
KBuf:      ?? - .
-----
<In> 0205  Opcode : 0028  Crs ID : 0xffbc000c  Db Length : 00000  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      -
KBuf:      -
<Out>0205  Status : 0000  Crs ID : 0xffbc000c  Db Length : 00000  Keynum : 00
Clnt ID : 00 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:      -
KBuf:      -
-----

```

オペレーションごとに、MicroKernel に渡した値と MicroKernel から返された値を示しています。入力値は <In> で表され、出力値は <Out> で表されています。それぞれの後ろに、オペレーションが処理された順序を示す番号が続くので、<Out> 0016 は <In> 0016 の結果となります。Opcode フィールドは、実行されたオペレーションコードを示します。Status フィールドは、返されたステータスコードを示します。

Crs ID は、MicroKernel が要求に割り当てたカーソル ID、つまり、ハンドルです。この情報は、複数のクライアントをサポートしたり、複数のカーソルで単一のクライアントをサポートするアプリケーションのデバッグに有効な場合があります。

Db Length は、データ バッファ長を反映します。Keynum は、キー番号を反映します。Clnt ID は、BTRVID 関数と BTRCALLID 関数で使用するクライアント ID パラメーターを反映します。DBuf は、データ バッファの内容を反映します。KBuf は、キー バッファの内容を反映します。トレース ファイルは、MicroKernel の設定に応じて、データ バッファおよびキー バッファの内容を切り詰めます。



メモ パフォーマンスの低下を回避するために、[トレース オペレーションの実行] 設定を短時間だけオンに切り替えて、どのオペレーションが **MicroKernel** で処理されているかを判断します。

クライアント / サーバー環境における間接的な Chunk オペレーション

Get Direct/Chunk オペレーション (23) を行おうとすると、アプリケーションはステータス コード 62 を受け取ることがあります。アプリケーションでは間接ランダム ディスクリプター オプション (サブファンクション 0x80000001) を指定しているのに、トレースを有効にすると、実際には MicroKernel が直接ランダム ディスクリプター オプション (サブファンクション 0x80000000) を受信していることがわかります。

間接チャンク オプションでは、アプリケーションはデータの取得後、そのデータが Btrieve 呼び出しの実際のデータ バッファー パラメーターに返されるのではなく、データを格納したいアプリケーションのメモリ ブロックに返されるよう、データ アドレスへのポインターを指定することができます。しかし、MicroKernel がアプリケーションのメモリへの直接アクセスを行わない環境でアプリケーションが動作しているので、Btrieve クライアント リクエスターは間接チャンク要求を直接チャンク要求に変換してから、その要求を MicroKernel に送ります。

すべてのアプリケーションは、アプリケーションと MicroKernel の間の通信に必ずプロセス間通信 (IPC) を使用します。IPC が要求されるため、MicroKernel はアプリケーションのメモリにアクセスしないので、クライアント リクエスターは単一の連続メモリ ブロックを割り当て、そのメモリ ブロックを指すようにすべてのデータ ポインターを調整します。MicroKernel から戻ると、リクエスターは間接オプションに対応する形式にデータ バッファーを変換し、返されたデータ チャンクをアプリケーションのメモリ ブロック内の指定された間接アドレスへ移動します。

エンジンのシャットダウンと接続のリセット

Windows 98/ME または Windows NT 以降をターゲットにするマルチスレッド コンソール アプリケーションを開発している場合は、起こり得る CTRL-C キーストロークを処理するようにコントロール ハンドラー ルーチンを設定する必要があります。このコントロールハンドラー ルーチンでは、**Reset** オペレーション (28) または **Stop** オペレーション (25) のいずれかを発行することによって、すべての Btrieve クライアントをクリーンアップする必要があります。クリーンアップ プロセスは、アプリケーションが CTRL-C イベントをオペレーティング システムに渡す前に終了しなければなりません。

システムがスレッドを終了したときにアプリケーションがアクティブのままになっている場合、**MicroKernel** はアプリケーションとの接続をクリーンアップできず、さらに多くのシステム リソースを強制的に割り当てます。このため、パフォーマンスの低下が発生し、エンジンのシャットダウンに必要な時間が大幅に増えます。コントロール ハンドラー ルーチンの詳細については、**Microsoft** のドキュメントを参照してください。

ファイル内の無駄な領域の削減

MicroKernel は、必要に応じてディスク領域を割り当てます。アプリケーションが新しいレコードを挿入するときにファイル内に十分なスペースがない場合、MicroKernel は追加のデータ ページとインデックス ページをファイルに自動的に割り当てます。割り当てられた各ページのサイズは、ファイルが作成されたときのページサイズと同じです。MicroKernel は、新しいファイル サイズを反映するためにディレクトリ構造も更新します。

MicroKernel がファイルにスペースを割り当てると、そのスペースはファイルが存在する間割り当てられたままになります。

➤ 多数のレコードが削除されたファイルが必要とするスペースを減らすために、以下のように Maintenance ユーティリティを使用します。

- 1 元のファイルと同じ特性を持つ新しいファイルを作成します。

対話形式の Maintenance ユーティリティでは、[オプション] メニューの [情報エディターの表示] を選択し、[ファイル情報エディター] で元のファイルの [情報のロード] を行い、[ファイルの作成] を行います。コマンド ライン ベースの Maintenance ユーティリティでは、コマンドは CLONE です。

- 2 以下の方法のうちの 1 つを使用して新しいファイルにレコードを挿入します。

- 元のファイルからレコードを読み取り、それらのレコードを新しいファイルに挿入する、小さなアプリケーションを作成します。
- コマンド ラインベースの Maintenance ユーティリティでは、SAVE コマンドを使用し、次に LOAD コマンドを使用します。別の方法として、COPY コマンドを使用することができます。
- 対話形式の Maintenance ユーティリティでは、[データ] メニューから [保存] コマンドを使用し、次に [ロード] コマンドを使用します。別の方法として、[データ] メニューから [コピー] コマンドを使用できます。

- 3 新しいファイルの名前を変更し、次にディスクからオリジナル ファイルを削除します。

この章では、Btrieve API を直接呼び出して PSQL アプリケーションの開発を開始する場合に役立つ情報を示します。最も一般的なプログラミング作業には、Visual Basic と Delphi のサンプル コードとサンプル構造体が付属しています。

以下の項目が含まれます。

- 「[Btrieve API プログラミングの基礎](#)」
- 「[Visual Basic に関する注記](#)」
- 「[Delphi に関する注記](#)」
- 「[PSQL アプリケーションの起動](#)」
- 「[Btrieve API のコード サンプル](#)」
- 「[Visual Basic のための Btrieve API 関数の宣言](#)」

Btrieve API プログラミングの基礎

以下のフロー チャートは、レコードの挿入、更新および削除にアプリケーションでどの Btrieve オペレーションを使用したらよいかを示しています。Btrieve の詳細については、『Btrieve API Guide』を参照してください。

Btrieve API フロー チャート

レコードの挿入

- 1 OPEN (0) でファイルを開く
- 2 INSERT (2) でレコードを追加する (繰り返し)
- 3 CLOSE (1) でファイルを閉じる
- 4 STOP (25) でリソースを解放する

レコードの更新

- 1 OPEN (0) でファイルを開く
- 2 GET EQUAL (5) またはその他の単一レコード取得オペレーションを実行して既存のレコードを検索し物理カレンシーを設定する
- 3 レコードを変更する
- 4 UPDATE (3) で更新する
- 5 CLOSE (1) でファイルを閉じる
- 6 STOP (25) でリソースを解放する

レコードの削除

- 1 OPEN (0) でファイルを開く
- 2 GET EQUAL (5) またはその他の単一レコード取得オペレーションを実行して既存のレコードを検索し物理カレンシーを設定する
- 3 DELETE (4) でレコードを削除する
- 4 CLOSE (1) でファイルを閉じる
- 5 STOP (25) でリソースを解放する

Visual Basic に関する注記

以下に、Visual Basic で PSQL アプリケーションを開発するときに注意が必要なことをいくつか示します。

- Visual Basic では、ユーザーが定義したデータ型にバイト配置の問題があることがわかっています。また、この問題に関する情報と PAln32.DLL、Btrieve Alignment DLL の使用方法については、Btrieve API に関するセクション「[Visual Basic](#)」を参照してください。
- 結果のレコードのタイプごとにレコード クラスを作成すると、以下のステップに示すようなデータ アクセスが容易に行えます。

- a. Record というクラスを作成します。
- b. レコードのレイアウトを定義する構造体を作成します。

```
Type Byte2
    field1 byte
    field2 byte
End Type
Type Rec
    Size As Byte2
    Name As String*30    'SQL マスク = x30
    Salary As String*10  'SQL マスク = zzzzzzzz.99
End Type
```

- c. iAsciiFlag=1 と ispadding=0 を使用してデータを Rec のインスタンスへ読み込みます。

```
Dim instofRec As New Rec
```

- d. ドット表記でデータにアクセスします。

```
instofRec.Name="Jeff"
```

- e. レコード クラスを使用してすべての instofRec データ操作を行います。

Delphi に関する注記

以下に、Delphi で PSQL アプリケーションを開発するときに考慮が必要なことを示します。

- Pascal の旧バージョンと違い、Delphi の長さ指定子のない**文字列型**は動的でヌル終端です。つまり、文字列バッファに値を割り当てるまで、メモリが文字列バッファに割り当てられている保証はありません。また、**文字列型**を使用する場合は、予想された結果を十分に保持できる大きさの文字列型を埋め込む必要があります。以下の例に示すように、StringOfChar () 関数を使用して、Btrieve からの予想された戻り値を取り込める大きさの空白文字列を割り当てることができます。

```
CustKeyBuffer: string; // 長い文字列型
CustBufLen   : word;
    // BTRV() はキー長として常に 255 を渡すため、
    // MAX_KEY_LEN は 255
CustKeyBuffer := StringOfChar(' ', MAX_KEY_LEN);
CustBufLen := SizeOf(CustRec);
Status := BTRV(B_GET_FIRST, CustPosBlock, CustRec,
    CustBufLen, CustKeyBuffer, 0);
    {CustKeyBuffer にはキーの値が入っている}
    {取得したレコード}
```

- この章に示すすべての Delphi サンプルがエラー レポートを示しているわけではありません。しかし、すべての呼び出しの後の戻りコードを確認する必要があります。
- この章のサンプルを実行しようとする場合、INTERNAL_FORMAT スタイルを使用するフェッチについては、クエリ内のフィールドの順序はデータ レコードからフェッチするメンバーの順序に合っていなければなりません。FillGridHeaders () ルーチンを使用する場合、クエリがフィールドを一覧する順序と同じ順序でグリッドを詰める必要があります。

PSQL アプリケーションの起動

PSQL アプリケーションを開発する場合は、特定のプログラミング インターフェイスに対応するソース モジュールを組み入れる必要があります。

- BTRCONST と BTRAPI32 – Btrieve アプリケーションに必要なソース モジュール

PSQL ソース モジュールの追加

アプリケーションを開発しているプログラミング インターフェイスに Btrieve ソース モジュールを組み込む必要があります。

➤Visual Basic プロジェクトに Btrieve ソース モジュールを追加するには

- 1 Visual Basic で新しいプロジェクトを起動します。
- 2 既存の標準モジュールをプロジェクトに追加します。
- 3 PSQL ソース モジュールを追加します。

➤Delphi プロジェクトに Btrieve ソース モジュールを追加するには

- 1 Delphi で新しいプロジェクトを起動します。
- 2 [プロジェクト] メニューから [オプション] を選択します。
- 3 [ディレクトリ] タブをクリックします。
- 4 [検索パス] データ フィールドに「<パス>¥INTFYDELPHI」を挿入します（パス部分は PSQL SDK コンポーネントのインストール先です）。
- 5 USES 句にソース モジュールを組み入れます。

Btrieve API のコード サンプル

ここでは、Btrieve API で実行できる以下のタスクの Visual Basic、Delphi および C/C++ のコード サンプルを示します。

- 「ファイルの作成」
- 「レコードの挿入」
- 「レコードの更新」
- 「Step オペレーションの実行」
- 「Get オペレーションの実行」
- 「チャンク、BLOB、および可変長レコード」
- 「セグメント化されたインデックスの処理」

ファイルの作成

Create (14) オペレーションを使用して、アプリケーション内からファイルを作成することができます。ファイルを作成するには、新しい Btrieve ファイルの作成に必要な情報を含む構造体を作成する必要があります。

この API の詳細については、『Btrieve API Guide』を参照してください。

サンプル コード

以下のサンプル コードは、Create オペレーションでファイルを作成する方法を示しています。

Visual Basic (ファイルの作成)

以下のサブルーチンは、Orders というファイルを作成します。

```
Sub CreateOrdersFile(OrdFileLocation As String)

    ' 次の構文はファイル仕様を設定します
    OrdFixedRecSize = Len(OrdRecBuf)
    FileDefinitionBuffer.RecLen = OrdFixedRecSize
    FileDefinitionBuffer.PageSize = 4096
    FileDefinitionBuffer.IndxCnt = 2
    FileDefinitionBuffer.FileFlags = VARIABLELENGTH

    ' キー 0、注文番号
    FileDefinitionBuffer.KeyBuf0.KeyPos = 1
    FileDefinitionBuffer.KeyBuf0.KeyLen = 4
    FileDefinitionBuffer.KeyBuf0.KeyFlags = EXTTYPE + MODIFIABLE
    FileDefinitionBuffer.KeyBuf0.KeyType = Chr$(BAUTOINC)

    ' キー 1、連絡先番号
    FileDefinitionBuffer.KeyBuf1.KeyPos = 5
    FileDefinitionBuffer.KeyBuf1.KeyLen = 4
    FileDefinitionBuffer.KeyBuf1.KeyFlags = EXTTYPE + MODIFIABLE + DUP
    FileDefinitionBuffer.KeyBuf1.KeyType = Chr$(BUNSGBIN)

    BufLen = Len(FileDefinitionBuffer)
    OrdFileLocation = OrdFileLocation & " "
    KeyBufLen = Len(OrdKeyBuffer)
    CopyMemory OrdKeyBuffer, OrdFileLocation, Len(OrdFileLocation)
    giStatus = BTRCALL(BCREATE, OrdPosBlock, FileDefinitionBuffer, BufLen, _
        ByValOrdFileLocation, KeyBufLen, 0)
End Sub
```

Delphi (ファイルの作成)

以下のルーチンは、Customer という可変長ファイルを作成します。

```
function CreateCustomerFile(FileName: String): SmallInt;
var
    CustRec : CustomerRecordType;           // ユーザー定義のレコード構造体
    CustBufLen : word;
    CustPosBlock : TPositionBlock;          // [1..128] のバイト配列
    CustFileLocation : String[255];
    CustFixedRecSize : LongInt;
    FileDefinitionBuffer : FileCreateBuffer; // ファイル作成用の構造体
    FilebufLen : Word;
    KeyNum : ShortInt;

begin
    { 次の構文はファイル仕様を定義します }
    { レコードの固定長部分のみのサイズを計算 }
    CustFixedRecSize := SizeOf(CustRec) -
        SizeOf(CustRec.Notes);
    FileDefinitionBuffer.fileSpec.recLen := CustFixedRecSize;
    FileDefinitionBuffer.fileSpec.pageSize := 4096;
    FileDefinitionBuffer.fileSpec.indexCount := 4;
    FileDefinitionBuffer.fileSpec.fileFlags := VARIABLELENGTH;

    { キー仕様の定義、キー 0 連絡先番号 }
    FileDefinitionBuffer.keyspecArray[0].Position := 1;
    FileDefinitionBuffer.keyspecArray[0].Length := 4; { 4 バイト整数 }
    FileDefinitionBuffer.keyspecArray[0].Flags := KFLG_EXTTYPE_KEY + KFLG_MODX;
    FileDefinitionBuffer.keyspecArray[0].KeyType := AUTOINCREMENT_TYPE;

    { キー 1、連絡先名 }
    FileDefinitionBuffer.keyspecArray[1].Position := 5;
    FileDefinitionBuffer.keyspecArray[1].Length := 30;
    FileDefinitionBuffer.keyspecArray[1].Flags := KFLG_EXTTYPE_KEY + KFLG_MODX +
        KFLG_DUP;
    FileDefinitionBuffer.keyspecArray[1].KeyType := STRING_TYPE;

    { キー 2、会社名 }
    FileDefinitionBuffer.keyspecArray[2].Position := 35;
    FileDefinitionBuffer.keyspecArray[2].Length := 60;
    FileDefinitionBuffer.keyspecArray[2].Flags := KFLG_EXTTYPE_KEY + KFLG_MODX +
        KFLG_DUP;
    FileDefinitionBuffer.keyspecArray[2].KeyType := STRING_TYPE;

    { キー 3、販売員、次回の連絡日 }
    FileDefinitionBuffer.keyspecArray[3].Position := 220;
    FileDefinitionBuffer.keyspecArray[3].Length := 4;
    FileDefinitionBuffer.keyspecArray[3].Flags := KFLG_EXTTYPE_KEY + KFLG_MODX +
        KFLG_DUP + KFLG_SEG;
    FileDefinitionBuffer.keyspecArray[3].KeyType := LSTRING_TYPE;
    FileDefinitionBuffer.keyspecArray[4].Position := 223;
    FileDefinitionBuffer.keyspecArray[4].Length := 4;
    FileDefinitionBuffer.keyspecArray[4].Flags := KFLG_EXTTYPE_KEY + KFLG_MODX +
        KFLG_DUP;
    FileDefinitionBuffer.keyspecArray[4].KeyType := DATE_TYPE;

    CustFileLocation := FileName + #0; { 作成するファイルのパスおよびファイル名 }
    FilebufLen := sizeof(FileDefinitionBuffer);
```

Btrieve API プログラミング

```
KeyNum          := 0;
FillChar(CustPosBlock, SizeOf(CustPosBlock), #0);

Result := BTRV(B_CREATE,          //OpCode 14
               CustPosBlock,      // ポジション ブロック ("cursor" または "handle")
               FileDefinitionBuffer, // 新規ファイルの定義
               FileBufLen,        // 定義の長さ
               CustFileLocation[1], // パスとファイル名
               keyNum);           // 0 (ゼロ) は既存のファイルを上書きする
end;    {CreateCustomerFile}
```

C/C++ (レコードの作成)

```
BTI_SINT CreateCustomerFile(LPCTSTR szCustomerFileName)
{
    Customer_Record_Type CustRec;          // ユーザー定義のレコード構造体
    char                  CustPosBlock[POS_BLOCK_SIZE]; // customer ファイル内の "Cursor"
    char                  CustFileLocation[255];
    size_t                CustFixedRecSize;
    FileDescriptionType   FileDefBuf;      // ファイル作成用の構造体
    BTI_WORD              FilebufLen;
    char                  KeyNum;           // 1 バイトの符号付き整数
    BTI_SINT              iStatus;

    /* レコードの固定長部分のサイズを計算 */
    CustFixedRecSize      = sizeof(CustRec) - sizeof(CustRec.Notes);
    FileDefBuf.RecLen     = CustFixedRecSize;
    FileDefBuf.PageSize   = 4096;
    FileDefBuf.IndxCnt    = 4;
    FileDefBuf.DupPointers = 4;
    FileDefBuf.FileFlags  = VAR_RECS | BALANCED_KEYS;
    /* キー仕様の定義、キー 0 連絡先番号 */
    FileDefBuf.KeyBuf[0].KeyPos = 1;
    FileDefBuf.KeyBuf[0].KeyLen = 4;
    FileDefBuf.KeyBuf[0].KeyFlags = EXTTYPE_KEY | MOD;
    FileDefBuf.KeyBuf[0].KeyType = AUTOINCREMENT_TYPE;
    /* キー 1 - 連絡先名 */
    FileDefBuf.KeyBuf[1].KeyPos = 5;
    FileDefBuf.KeyBuf[1].KeyLen = 30;
    FileDefBuf.KeyBuf[1].KeyFlags = EXTTYPE_KEY | DUP | MOD ;
    FileDefBuf.KeyBuf[1].KeyType = STRING_TYPE;
    /* キー 2 - 会社名 */
    FileDefBuf.KeyBuf[2].KeyPos = 35;
    FileDefBuf.KeyBuf[2].KeyLen = 60;
    FileDefBuf.KeyBuf[2].KeyFlags = EXTTYPE_KEY | DUP | MOD ;
    FileDefBuf.KeyBuf[2].KeyType = STRING_TYPE;
    /* キー 3 - 販売員、次回の連絡日 */
    FileDefBuf.KeyBuf[3].KeyPos = 220;
    FileDefBuf.KeyBuf[3].KeyLen = 3;
    FileDefBuf.KeyBuf[3].KeyFlags = EXTTYPE_KEY | DUP | MOD | SEG;
    FileDefBuf.KeyBuf[3].KeyType = STRING_TYPE;

    FileDefBuf.KeyBuf[4].KeyPos = 223;
    FileDefBuf.KeyBuf[4].KeyLen = 4;
    FileDefBuf.KeyBuf[4].KeyFlags = EXTTYPE_KEY | DUP | MOD ;
    FileDefBuf.KeyBuf[4].KeyType = DATE_TYPE;

    //
```

```
//-----
FileBufLen = sizeof(FileDefBuf);
KeyNum = 0; // 上書き処理
strcpy(CustFileLocation, szCustomerFileName);
iStatus = BTRV(B_CREATE,
               CustPosBlock,
               &FileDefBuf,
               &FileBufLen,
               CustFileLocation,
               KeyNum);
return(iStatus);
} // CreateCustomerFile()
```

サンプル構造体（ファイルの作成）

以下のサンプル構造体はそれぞれ、前の Visual Basic、Delphi および C/C++ のコード サンプルで使用する構造体です。

Visual Basic（ファイルの作成）ーサンプル構造体

```
Declare Function BTRCALL Lib "w3btrv7.dll" (ByVal OP, ByVal Pb$, _
      Db As Any, DL As Long, Kb As Any, ByVal Kl, ByVal Kn) As Integer
```

```
Declare Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" _
      (hvpDest As Any, hvpSource As Any, ByVal cbCopy As Long)
```

```
Type OrderRecordBufferType
    OrderNumber As typ_byte4      ' 4 バイト unsigned
    ContactNumber As typ_byte4    ' 4 バイト unsigned
    OrderDate As DateType
    OrderTotal As typ_byte4      ' 4 バイト real
    NotUsed As String * 64
End Type
```

```
Type OrderKeyBufferType
    BufferValue(255) As Byte
    OrderNumber As typ_byte4
    CustNumber As typ_byte4
    NotUsed As String * 64
End Type
```

```
Type FileSpec
    RecLen As Integer
    PageSize As Integer
    IndxCnt As Integer
    NotUsed As String * 4
    FileFlags As Integer
    Reserved As String * 2
    Allocation As Integer
    KeyBuf0 As KeySpec
    KeyBuf1 As KeySpec
    KeyBuf2 As KeySpec
    KeyBuf3 As KeySpec
    KeyBuf4 As KeySpec
    KeyBuf5 As KeySpec
End Type
```

```
Global FileDefinitionBuffer As FileSpec
```

Btrieve API プログラミング

```
{ 以下は Order テーブル変数 }
Global OrdPosBlock      As Byte(0 to 127)
Global OrdRecPos        As typ_byte4
Global OrdRecBuf        As OrderRecordBufferType
Global OrdKeyBuffer     As OrderKeyBufferType
Global OrdFixedRecSize  As Long
Global OrdFileLocation  As String
```

Delphi（ファイルの作成）－サンプル構造体

```
type
  CustomerRecordType = packed record
    必要に応じた処理
  end;    //CustomerRecordType

type
  TPositionBlock = array[0..127] of byte;

type
  BTI_KEY_DESC = packed record
    Position   : BTI_SINT;
    Length     : BTI_SINT;
    KeyFlags   : BTI_SINT;
    NumUnique  : BTI_LONG;
    ExtKeyType : BTI_BYTE;
    NullVal    : BTI_BYTE;
    Reserv     : array [0..1] of BTI_CHAR;
    KeyNumber  : BTI_UBYTE;
    ACSNumber  : BTI_UBYTE;
  end;    {BTI_KEY_DESC}

  BTI_KEY_ARRAY = array [0..MAX_KEY_SEG - 1] of BTI_KEY_DESC;
  BTI_ACS        = array [0..ACS_SIZE - 1] of BTI_CHAR;

type
  FileCreateBuffer = packed record
    RecLen      : BTI_SINT;
    PageSize    : BTI_SINT;
    NumKeys     : BTI_SINT;
    Reserved1   : BTI_LONG;
    FileFlags   : BTI_SINT;
    DupPointers : BTI_BYTE;
    Reserved2   : BTI_BYTE;
    Alloc       : BTI_SINT;
    Keys        : BTI_KEY_ARRAY;
    ACS         : BTI_ACS;
  end;    {BTI_FILE_DESC}
```

ここで留意すべき点は、定義を簡単にするために、オルタネート コーレーティング シーケンス (ACS) をキー配列全体の後に置くということです。Btrieve は、最後のキー セグメントの直後に ACS が続くことを期待しているため、ACS を構造体内の該当位置まで移動させる必要があります。

C/C++（ファイルの作成）－サンプル構造体

```
struct CustRec
{
    必要に応じた処理
}    //CustRec
```



```

struct date_type
{
    BTI_BYTE   day;
    BTI_BYTE   month;
    BTI_SINT   year;
}; //date_type

struct KeySpec
{
    BTI_SINT   KeyPos;
    BTI_SINT   KeyLen;
    BTI_SINT   KeyFlags;
    BTI_LONG   KeyTot;
    BTI_CHAR   KeyType;
    BTI_CHAR   NulValue;
    BTI_CHAR   NotUsed[2];
    BTI_BYTE   KeyNumber;
    BTI_BYTE   ACSNum;
}; //KeySpec

struct FileDescriptionType
{
    BTI_SINT   RecLen;
    BTI_SINT   PageSize;
    BTI_SINT   IndxCnt;
    BTI_LONG   RecTot;
    BTI_SINT   FileFlags;
    BTI_BYTE   DupPointers;
    BTI_BYTE   NotUsed;
    BTI_SINT   Allocation;
    KeySpec   KeyBuf[119];
}; //FileDescriptionType

```

レコードの挿入

Insert (2) オペレーションは、ファイルにレコードを挿入します。この API への呼び出しを行うには以下の前提条件を満たす必要があります。

- 対象となるファイルが開いている必要があります。
- 挿入するレコードは適切なレコード長を持つ必要があります。また、キー値は対象となるファイルで定義されているキーと一致していなければなりません。

データバッファに挿入する行を設定して BINSERT を呼び出すことにより、行を挿入できます。この API の詳細については、『Btrieve API Guide』を参照してください。以下のサンプルコードとサンプル構造体は、Visual Basic、Delphi および C/C++ で Insert オペレーションを実行する方法を示しています。

サンプル コード

以下のサンプルコードは、Insert オペレーションでレコードを挿入する方法を示しています。

- 「[Visual Basic \(レコードの挿入\)](#)」
- 「[Delphi \(レコードの挿入\)](#)」
- 「[C/C++ \(レコードの挿入\)](#)」

Visual Basic (レコードの挿入)

```
FillCustBufFromCustomerEdit
```

```
InsertCustomerRecord ' BtrCallModule プロシージャ
```

```

Sub FillCustBufFromCustomerEdit()
    Dim tmplong As Long
    Dim StrDay As String * 2
    Dim StrMonth As String * 2
    Dim StrYear As String * 4

    tmplong = CLng(FormCustomerEdit.EdContactNumber.Text)
    CustRecBuf.ContactNumber = ToType4(tmplong)
    ' この関数は、「サンプル構造体 (レコードの挿入)」のサンプルを参照してください
    CustRecBuf.ContactName = FormCustomerEdit.EdContactName.Text
    CustRecBuf.CompanyName = FormCustomerEdit.EdCompanyName.Text
    CustRecBuf.Address1 = FormCustomerEdit.EdAddress1.Text
    CustRecBuf.Address2 = FormCustomerEdit.EdAddress2.Text
    CustRecBuf.City = FormCustomerEdit.EdCity.Text
    CustRecBuf.State = FormCustomerEdit.EdState.Text
    CustRecBuf.ZipCode = FormCustomerEdit.EdZip.Text
    CustRecBuf.Country = FormCustomerEdit.EdCountry.Text
    CustRecBuf.SalesRep = FormCustomerEdit.EdSalesRep.Text
    StrDay = Mid$(FormCustomerEdit.EdContactDate.Text, 1, 2)
    StrMonth = Mid$(FormCustomerEdit.EdContactDate.Text, 4, 2)
    StrYear = Mid$(FormCustomerEdit.EdContactDate.Text, 7, 4)
    CustRecBuf.NextContact.Day = CByte(StrDay)
    CustRecBuf.NextContact.Month = CByte(StrMonth)
    CustRecBuf.NextContact.Year = CInt(StrYear)
    CustRecBuf.PhoneNumber = FormCustomerEdit.EdPhone.Text
    CustRecBuf.Notes = Trim(FormCustomerEdit.EdNotes.Text) & Chr$(0)
    FormCustomerEdit.EdRecLength = Str(CustBufLength)
End Sub

Sub InsertCustomerRecord()
    Dim lCustBufLength As Long
    Dim iKeyNum As Integer
    Dim iKeyBufLen As Integer
    lCustBufLength = Len(CustRecBuf) - MaxNoteFieldSize + Len(Trim(CustRecBuf.Notes))
    ' CustBufLength = 238
    iKeyBufLen = KEY_BUF_LEN
    iKeyNum = CustKeyBuffer.CurrentKeyNumber
    giStatus = BTRCALL(BINSERT, CustPosBlock, CustRecBuf, _
        lCustBufLength, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub

```

Delphi (レコードの挿入)

```

function InsertCustomerRecord(var CustPosBlock : TPositionBlock;
    CustRec      : CustomerRecordType)
    : SmallInt;

var
    CustBufLen    : Word;
    KeyNum        : ShortInt;
    CustKeyBuffer : String[255];
begin
    { 可変長レコードの全体のサイズを計算 }
    CustBufLen := SizeOf(CustRec) - SizeOf(CustRec.Notes) + Length(CustRec.Notes);
    KeyNum := -1; { 挿入時の " カレンシー変更なし " を指定 }
    FillChar(CustKeyBuffer, SizeOf(CustKeyBuffer), #0); { 実行は不要 }
    Result := BTRV(B_INSERT,
        CustPosBlock, //OpCode 2
        // 既に開いているポジション ブロック

```

```

        CustRec,          // 挿入するレコード
        CustBufLen,       // 新規レコードの長さ
        CustKeyBuffer[1], // NCC insert には不要
        KeyNum);
end;    {InsertCustomerRecord}

```

C/C++ (レコードの挿入)

```

BTI_SINT InsertCustomerRecord(char          CustPosBlock[POS_BLOCK_SIZE],
                               Customer_Record_Type CustRec)
{
    BTI_WORD  CustBufLen;
    char      KeyNum;      // 1 バイトの符号付きバイト
    char      CustKeyBuffer[255];
    BTI_SINT  iStatus;

    /* 可変長レコードの全体のサイズを計算 */
    CustBufLen = sizeof(CustRec) - sizeof(CustRec.Notes) + strlen(CustRec.Notes);
    KeyNum = -1;    // insert 中の " カレンシー変更なし " を指定
    memset(CustKeyBuffer, sizeof(CustKeyBuffer), 0);    // 実行は不要
    iStatus = BTRV(B_INSERT,          //OpCode 2
                  CustPosBlock,       // 既に開いているポジション ブロック
                  &CustRec,           // 挿入するレコード
                  &CustBufLen,       // 新規レコードの長さ
                  CustKeyBuffer,     // NCC insert には不要
                  KeyNum);
    PrintStatus("B_INSERT: status = %d", iStatus);
    return(iStatus);
} // InsertCustomerRecord()

```

サンプル構造体 (レコードの挿入)

以下のサンプル構造体はそれぞれ、前の Visual Basic、Delphi および C/C++ のコード サンプルで使用される構造体です。

Visual Basic (レコードの挿入) – サンプル構造体

```

Global Const BINSERT = 2

' 以下は Customer テーブルのデータ構造
Type CustomerRecordBufferType
    ContactNumber As typ_byte4
    ContactName As String * 30
    CompanyName As String * 60
    Address1 As String * 30
    Address2 As String * 30
    City As String * 30
    State As String * 2
    ZipCode As String * 10
    Country As String * 3
    PhoneNumber As String * 20
    SalesRep As String * 3
    NextContact As DateType
    NotUsed As String * 12
    Notes As String * MaxNoteFieldSize
End Type

```

Btrieve API プログラミング

```
' 以下は Customer ファイルの変数
Global CustPosBlock As Byte(0 to 127)
Global CustRecBuf As CustomerRecordBufferType
Global CustKeyBuffer As CustomerKeyBufferType
Global CustFixedRecSize As Long
Global CustFileLocation As String
Global CustPosition As typ_byte4
Global CustPosPercent As typ_byte4

Function ToInt(vValue As typ_byte4) As Long
    Dim iInt As Long
    CopyMemory iInt, vValue, 4
    ToInt = iInt
End Function

Function ToType4(vValue As Long) As typ_byte4
    Dim tmpTyp4 As typ_byte4
    CopyMemory tmpTyp4, vValue, 4
    ToType4 = tmpTyp4
End Function
```

Delphi (レコードの挿入) – サンプル構造体

```
type
    CustomerRecordType = packed record
        必要に応じた処理
    end;    //CustomerRecordType
```

C/C++ (レコードの挿入) – サンプル構造体

```
struct CustRec
{
    必要に応じた処理
}    //CustRec
```

レコードの更新

Update (3) オペレーションは、既存のレコード内の情報を変更します。この Btrieve 呼び出しを行うには、ファイルが開いており、物理カレンシーが確立していなければなりません。トランザクション内でレコードを更新する場合は、レコードの取得もトランザクション内で行う必要があります。

この API の詳細については、『Btrieve API Guide』を参照してください。以下のサンプルコードとサンプル構造体は、Visual Basic、Delphi および C/C++ で Update オペレーションを実行する方法を示しています。

サンプルコード

以下のサンプルコードは、Update オペレーションでファイルを変更する方法を示しています。

- 「[Visual Basic \(レコードの更新\)](#)」
- 「[Delphi \(レコードの更新\)](#)」
- 「[C/C++ \(レコードの更新\)](#)」

Visual Basic (レコードの更新)

```
FillCustBufFromCustomerEdit
UpdateCustomerRecord      'BtrCallModule プロシージャ

Sub FillCustBufFromCustomerEdit()
```

```

Dim tmplong As Long
Dim StrDay As String * 2
Dim StrMonth As String * 2
Dim StrYear As String * 4

tmplong = CLng(FormCustomerEdit.EdContactNumber.Text)
CustRecBuf.ContactNumber = ToType4(tmplong)
CustRecBuf.ContactName = FormCustomerEdit.EdContactName.Text
CustRecBuf.CompanyName = FormCustomerEdit.EdCompanyName.Text
CustRecBuf.Address1 = FormCustomerEdit.EdAddress1.Text
CustRecBuf.Address2 = FormCustomerEdit.EdAddress2.Text
CustRecBuf.City = FormCustomerEdit.EdCity.Text
CustRecBuf.State = FormCustomerEdit.EdState.Text
CustRecBuf.ZipCode = FormCustomerEdit.EdZip.Text
CustRecBuf.Country = FormCustomerEdit.EdCountry.Text
CustRecBuf.SalesRep = FormCustomerEdit.EdSalesRep.Text
StrDay = Mid$(FormCustomerEdit.EdContactDate.Text, 1, 2)
StrMonth = Mid$(FormCustomerEdit.EdContactDate.Text, 4, 2)
StrYear = Mid$(FormCustomerEdit.EdContactDate.Text, 7, 4)
CustRecBuf.NextContact.Day = CByte(StrDay)
CustRecBuf.NextContact.Month = CByte(StrMonth)
CustRecBuf.NextContact.Year = CInt(StrYear)
CustRecBuf.PhoneNumber = FormCustomerEdit.EdPhone.Text
CustRecBuf.Notes = Trim(FormCustomerEdit.EdNotes.Text) & Chr$(0)
FormCustomerEdit.EdRecLength = Str(CustBufLength)
End Sub

Sub UpdateCustomerRecord()
    Dim lCustBufLength As Long
    Dim iKeyBufLen As Integer
    Dim iKeyNum As Integer

    ' 次の構文は customer レコードを更新します
    lCustBufLength = Len(CustRecBuf) - MaxNoteFieldSize + Len(Trim(CustRecBuf.Notes))
    iKeyBufLen = KEY_BUF_LEN
    iKeyNum = CustKeyBuffer.CurrentKeyNumber
    giStatus = BTRCALL(bUpdate, CustPosBlock, CustRecBuf, _
                lCustBufLength, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub

```

Delphi (レコードの更新)

```

function UpdateCustomerRecord(var CustPosBlock: TPositionBlock;
                               CustRec      : CustomerRecordType)
                               : SmallInt;
var
    CustBufLen      : Word;
    KeyNum          : ShortInt;
    CustKeyBuffer   : String[255];
begin
    { 可変長レコードの全体のサイズを計算 }
    CustBufLen := SizeOf(CustRec) - SizeOf(CustRec.Notes) + Length(CustRec.Notes);
    KeyNum := -1; { 更新時の "カレンシー変更なし" を指定 }
    FillChar(CustKeyBuffer, SizeOf(CustKeyBuffer), #0); { 実行は不要 }
    Result := BTRV(B_UPDATE,           //OpCode 3
               CustPosBlock,           // 既に開いているポジション ブロック
               CustRec,                 // 新規レコード
               CustBufLen,              // 新規レコードの長さ

```

Btrieve API プログラミング

```
                CustKeyBuffer[1],    // NCC update には不要
                KeyNum);
end;    {UpdateCustomerRecord}
```

C/C++ (レコードの更新)

```
BTI_SINT UpdateCustomerRecord(char                CustPosBlock[POS_BLOCK_SIZE],
                               Customer_Record_Type CustRec)
{
    BTI_WORD  CustBufLen;
    char      KeyNum;    // 1 バイトの符号付きバイト
    char      CustKeyBuffer[255];
    BTI_SINT  iStatus;

    /* 可変長レコードの全体のサイズを計算 */
    CustBufLen = sizeof(CustRec) - sizeof(CustRec.Notes) + strlen(CustRec.Notes);
    KeyNum = -1;    // 更新時の " カレンシー変更なし " を指定
    memset(CustKeyBuffer, sizeof(CustKeyBuffer), 0);    // 実行は不要
    iStatus = BTRV(B_UPDATE,                //OpCode 3
                   CustPosBlock,            // 既に開いているポジション ブロック
                   &CustRec,                // 挿入するレコード
                   &CustBufLen,            // 新規レコードの長さ
                   CustKeyBuffer,          // NCC insert には不要
                   KeyNum);
    PrintStatus("B_UPDATE: status = %d", iStatus);
    return(iStatus);
} //UpdateCustomerRecord()
```

サンプル構造体 (レコードの更新)

以下のサンプル構造体はそれぞれ、前の Visual Basic、Delphi および C/C++ のコード サンプルで使用する構造体です。

Visual Basic (レコードの更新) – サンプル構造体

```
Global Const bUpdate = 3
```

Insert オペレーションについては、「[サンプル構造体 \(レコードの挿入\)](#)」を参照してください。

Delphi (レコードの更新) – サンプル構造体

```
type
    CustomerRecordType = packed record
        必要に応じた処理
    end;    //CustomerRecordType
```

C/C++ (レコードの更新) – サンプル構造体

```
struct CustRec
{
    必要に応じた処理
} //CustRec
```

Step オペレーションの実行

Step オペレーション (Step First、Step Next、Step Last、Step Previous) では、レコードを取得してデータ バッファに入れることができます。レコードを取得するためにキー パスは使用されません。これらの API の詳細については、『Btrieve API Guide』を参照してください。

以下のサンプル コードとサンプル構造体は、Delphi および C/C++ で Step オペレーションを実行する方法を示しています。



メモ レコードが返される順序に決して依存しないでください。MicroKernel は、いつでもファイル内のレコードを移動できます。特定の順序でレコードを必要とする場合は、Get オペレーションを使用してください。

サンプル コード

以下のサンプル コードは、Step オペレーションでレコードを取得する方法を示しています。

Delphi (Step オペレーション)

以下のコード例は、ファイル内の最初の物理位置を返します。

```
{ ファイルから最初の物理レコードを取得 }
CustBufLen := SizeOf(CustRec);           // データ レコードの最大サイズ
Status      := BTRV(B_STEP_FIRST,       // OpCode 33
                  CustPosBLock,         // 既に開いているポジション ブロック
                  CustRec,              // レコードが返されるバッファ
                  CustBufLen,           // 返される最大長
                  CustKeyBuffer[1],     // Step では不要
                  CustKeyNumber);       // Step では不要

{ ファイルから 2 番目のレコードを取得 (順序は保証されない) }
CustBufLen := SizeOf(CustRec);           // リセット - 前の Step によって変更されている
Status      := BTRV(B_STEP_NEXT,        // OpCode 24
                  CustPosBLock,
                  CustRec,
                  CustBufLen,
                  CustKeyBuffer[1],
                  CustKeyNumber);

{ 先頭レコードに戻る }
CustBufLen := SizeOf(CustRec);           // リセット - 前の Step によって変更されている
Status      := BTRV(B_STEP_PREV,        // OpCode 35
                  CustPosBLock,
                  CustRec,
                  CustBufLen,
                  CustKeyBuffer[1],
                  CustKeyNumber);
```

C/C++ (Step オペレーション)

```
/* ファイルから最初の物理レコードを取得 */
CustBufLen = sizeof(CustRec);           // データ レコードの最大サイズ
iStatus = BTRV(B_STEP_FIRST,           // OpCode 33
              CustPosBLock,             // 既に開いているポジション ブロック
              &CustRec,                 // レコードが返されるバッファ
              CustBufLen,               // 返される最大長
              CustKeyBuffer,           // Step では不要
              KeyNum);                 // Step では不要

/* ファイルから 2 番目のレコードを取得 (順序は保証されない) */
CustBufLen = sizeof(CustRec);           // リセット - 前の Step によって変更されている
iStatus = BTRV(B_STEP_NEXT,            // OpCode 24
              CustPosBLock,
              &CustRec,
```

```

        &CustBufLen,
        CustKeyBuffer,
        KeyNum);
    /* 先頭レコードに戻る */
    CustBufLen = sizeof(CustRec);      // リセット - 前の Step によって変更されている
    iStatus = BTRV(B_STEP_PREVIOUS,    // OpCode 35
        CustPosBlock,
        &CustRec,
        &CustBufLen,
        CustKeyBuffer,
        KeyNum);

```

サンプル構造体

以下のサンプル構造体はそれぞれ、前の Delphi および C/C++ のコード サンプルで使用される構造体です。

Delphi (Step オペレーション) - サンプル構造体

```

type
    CustomerRecordType = packed record
        必要に応じた処理
    end;    //CustomerRecordType

```

C/C++ (Step オペレーション) - サンプル構造体

```

struct CustRec
{
    必要に応じた処理
}    //CustRec

```

Get オペレーションの実行

Get オペレーションでは、レコードを取得できます。これらのオペレーションは、どの行を返すかを指定するためにキー バッファ パラメーターを必要とします。これらの API の詳細については、『Btrieve API Guide』を参照してください。

以下のサンプル コードとサンプル構造体は、Visual Basic、Delphi および C/C++ でいくつかの Get オペレーションを実行する方法を示しています。

サンプル コード

以下のサンプル コードは、Get オペレーションでファイルを取得する方法を示しています。

Visual Basic (Get オペレーション)

```

Sub LoadContactBox(RecPosition As typ_byte4)
    FormBrowseCustomers.lstContact.Clear
    GetDirectCustomerRecord ' BtrCallModule プロシージャ
    If giStatus = 0 Then

        ' 次の構文は contact リスト ボックス文字列を作成します
        FormatContListBoxString
        If giStatus = 0 Then
            PosIndex = 0
            PosArray(PosIndex) = RecPosition
            FirstRecPosition = RecPosition
        End If
    Else

```



```

    FormBrowseCustomers.lblMsg.Caption = "didn't get record"
End If

' 次の構文はリスト ボックスの残りを埋めます
While giStatus = 0 And PosIndex < CustMaxNumRows - 1
    GetNextCustomerRecord ' BtrCallModule プロシージャ
    If giStatus = 0 Then
        ' contact リスト ボックス文字列を作成
        FormatContListBoxString

        ' 次の構文はレコード位置を返します
        GetPositionCustomerRecord ' BtrCallModule プロシージャ
        If giStatus = 0 Then
            PosIndex = PosIndex + 1
            PosArray(PosIndex) = RecPosition

        ' 次の構文はレコード位置配列へのポインターを構築します

        Select Case PosIndex
            Case 1
                SecondRecPosition = RecPosition
            Case 10
                SecToLastRecPosition = RecPosition
            Case 11
                LastRecPosition = RecPosition
        End Select
    End If
End If
Wend
If FormBrowseCustomers.lstContact.ListCount <> 0 Then
    FormBrowseCustomers.lstContact.ListIndex = 0
End If
End Sub

Sub GetDirectCustomerRecord()
    Dim iKeyBufLen As Integer
    Dim iKeyNum As Integer

    ' 次の構文はレコード位置によって直接レコードを取得します

    BufLen = Len(CustRecBuf)
    iKeyBufLen = MaxKeyBufLen
    iKeyNum = CustKeyBuffer.CurrentKeyNumber

    ' 次の構文はデータ バッファにアドレスを設定します

    CustRecBuf.Notes = "" ' 取得の前に可変長領域をクリア
    LSet CustRecBuf = RecPosition
    giStatus = BTRCALL(BGETDIRECT, CustPosBlock, _
    CustRecBuf, BufLen, CustKeyBuffer, iKeyBufLen, iKeyNum)
    DBLen = BufLen
End Sub

Sub GetNextCustomerRecord()
    Dim iKeyNum As Integer
    Dim iKeyBufLen As Integer

    ' 次の構文は次の customer レコードを返します

```

Btrieve API プログラミング

```
BufLen = Len(CustRecBuf)
iKeyBufLen = KEY_BUF_LEN
iKeyNum = CustKeyBuffer.CurrentKeyNumber
giStatus = BTRCALL(BGETNEXT, CustPosBlock, CustRecBuf, _
    BufLen, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub
```

```
Sub GetPositionCustomerRecord()
    Dim iKeyBufLen As Integer
    Dim iKeyNum As Integer
```

・ 次の構文はレコード位置を返します

```
BufLen = MaxDataBufLen
iKeyBufLen = KEY_BUF_LEN
iKeyNum = CustKeyBuffer.CurrentKeyNumber
giStatus = BTRCALL(BGETPOS, CustPosBlock, RecPosition, _
    BufLen, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub
```

Delphi (Get オペレーション)

```
var
    CustKeyBuffer: LongInt;
begin
    CustBufLen := SizeOf(CustRec);
    CustKeyNumber := 0; { Contact ID 順 }

    { 次の構文は指定したソート順を使用してファイルから最初のレコードを返します }
    CustBufLen := SizeOf(CustRec);           // データ レコードの最大サイズ
    Status := BTRV(B_GET_FIRST,           //OpCode 12
        CustPosBlock,           // 既に開いているポジション ブロック
        CustRec,               // レコードが返されるバッファー
        CustBufLen,           // 返される最大長
        CustKeyBuffer,         // レコードから抽出したキー値を返す
        CustKeyNumber);       // 取得時に使用するインデックス順

    { 次の構文は指定したソート順でファイル内の次のレコードを返します }
    CustBufLen := SizeOf(CustRec);           // リセット - 前の Get によって変更されている
    Status := BTRV(B_GET_NEXT,           //OpCode 6
        CustPosBlock,
        CustRec,
        CustBufLen,
        CustKeyBuffer[1],
        CustKeyNumber);

    { 次の構文はファイル内の前のレコードを返します }
    CustBufLen := SizeOf(CustRec);           // リセット - 前の Step によって変更されている
    Status := BTRV(B_GET_PREV,           //OpCode 7
        CustPosBlock,
        CustRec,
        CustBufLen,
        CustKeyBuffer[1],
        CustKeyNumber);
```

C/C++ (Get オペレーション)

```

/* ファイルから最初の論理レコードを取得 */
CustBufLen = sizeof(CustRec);      // データ レコードの最大サイズ
iStatus = BTRV(B_GET_FIRST,        // OpCode 12
               CustPosBBlock,      // 既に開いているポジション ブロック
               &CustRec,           // レコードが返されるバッファ
               &CustBufLen,        // 返される最大長
               CustKeyBuffer,      // レコードから抽出したキー値を返す
               CustKeyNumber);     // 取得時に使用するインデックス順

/* ファイルから 2 番目のレコードを取得選択したキー順 */
CustBufLen = sizeof(CustRec);      // リセット - 前の Get によって変更されている
iStatus = BTRV(B_GET_NEXT,         // OpCode 6
               CustPosBBlock,
               &CustRec,
               &CustBufLen,
               CustKeyBuffer,
               CustKeyNumber);

/* 先頭レコードに戻る */
CustBufLen = sizeof(CustRec);      // リセット - 前の Get によって変更されている
iStatus = BTRV(B_GET_PREVIOUS,     // OpCode 7
               CustPosBBlock,
               &CustRec,
               &CustBufLen,
               CustKeyBuffer,
               CustKeyNumber);

```

サンプル構造体 (Get オペレーション)

以下のサンプル構造体はそれぞれ、前の Visual Basic および Delphi のコード サンプルで使用する構造体です。

Visual Basic (Get オペレーション) - サンプル構造体

```

Global Const BGETNEXT = 6
Global Const BGETDIRECT = 23
Global Const BGETPOS = 22

```

Delphi (Get オペレーション) - サンプル構造体

```

type
  CustomerRecordType = packed record
    必要に応じた処理
  end; //CustomerRecordType

```

C/C++ (Get オペレーション) - サンプル構造体

```

struct CustRec
{
    必要に応じた処理
} //CustRec

```

チャンク、BLOB、および可変長レコード

Btrieve のチャンク オペレーションを使用すると、可変長レコード部分および BLOB 部分の読み書きが行えます。最大レコード長は 64 GB ですが、固定レコード長の最大は 64 KB (65,535 バイト) です。最初の 65,535 バイトを越えた先にあるレコードの各部分にアクセスしたいときに、チャンクを使用します。

サンプル コード

以下のサンプル コードは、チャンク、バイナリ ラージ オブジェクト (BLOB) および可変長レコードを処理する方法を示しています。

Visual Basic (チャンク /BLOB/ 可変長レコード)

```
Private Sub LoadImageFromBtrieve()

' 次の構文は、Btrieve に格納されているイメージを
' 出力イメージ テキスト ボックスで指定したファイルに返します

Dim lBytes As Long
Dim lBytesread As Long
Dim sLine As String
Dim lBytesToRead As Long
Dim iKey As Integer
Dim lAddressMode As Long
Dim lNumberOfChunks As Long
Dim lChunkOffset As Long
Dim lChunkAddress As Long
Dim lChunkLength As Long
Dim iNumChunksRead As Integer

GetEqualGraphicRecord ' レコードと BLOB の一部を取得
On Error GoTo FileNotFound

FormCustomerGraphic.MousePointer = 11
lNumberOfChunks = 0
On Error GoTo BMPOpenError
Open txtOutputImage.Text For Binary Access Write As #1
lBytesread = (BufLen - 68) ' 読み取ったバイト数からグラフィック レコードの
                          ' 固定長を差し引いた値を保存する
                          ' グラフィック レコードの最初のチャンクの固定長は
                          ' 68 です (上の GetEqualGraphicRecord)

sLine = Right(ChunkReadBuffer.ChunkArray, lBytesread)
Put #1, , sLine ' 最初のチャンクを bmp ファイルに書き出す
iNumChunksRead = 1
If giStatus = 22 And (BufLen = MaxChunkSize) Then
    GetPositionGraphicRecord ' チャンクを取得する前に
                          ' 現在のレコードの位置を取得する必要がある
Do
    lNumberOfChunks = 1
    lChunkOffset = 0
    lChunkAddress = 0
    lChunkLength = MaxChunkSize
    iNumChunksRead = iNumChunksRead + 1
    ChunkGetBuffer.RecordAddress = GrphPosition

' H80000000 (ランダム チャンクの取得)

' H40000000 (ネクスト イン レコード バイアス) はレコード内カレンシーを使用させる

    ChunkGetBuffer.AddressMode = ToType4((&H80000000 + &H40000000))
    ChunkGetBuffer.NumberOfChunks = ToType4(lNumberOfChunks)
    ChunkGetBuffer.ChunkOffset = ToType4(lChunkOffset)
    ChunkGetBuffer.ChunkAddress = ToType4(lChunkAddress)
    ChunkGetBuffer.ChunkLength = ToType4(lChunkLength)
```

- ・ 前の構文は読み取りバッファを使用します。
- ・ 最初のチャンクの取得 GetEqualGraphicRecord で
- ・ レコードの固定長が読み取られているため、以降の
- ・ チャンクの取得ではバッファ全体を使用します。

- ・ 次の構文は読み取りバッファと取得バッファを読み込みます

```

CopyMemory ChunkReadBuffer, ChunkGetBuffer, Len (ChunkGetBuffer)
GetGraphicChunk
If giStatus = 0 Then      ' レコードの終わりを越えて読むと 103 が返される
    If MaxChunkSize <> BufLen Then
        sLine = Left (ChunkReadBuffer.ChunkArray, BufLen)
        lBytesread = lBytesread + (BufLen)
    Else
        sLine = ChunkReadBuffer.ChunkArray
        Bytesread = lBytesread + MaxChunkSize
    End If
    If Len(sLine) > 0 Then
        Put #1, , sLine
    End If
End If
Loop While (giStatus = 0)
End If
Close #1
On Error Resume Next
Set Image1.Picture = LoadPicture(txtOutputImage.Text)
FormCustomerGraphic.MousePointer = 0
NumChunks.Text = iNumChunksRead
NumBytes.Text = lBytesread
LastStatus.Text = giStatus
On Error GoTo 0
Exit Sub

'InvalidPicture:

MsgBox Err.Number & ":" & Err.Description & vbCrLf & "Load from disk and save",
    vbOKOnly, "Invalid Picture in Graphic file"
Resume Next

FileNotFound:
MsgBox Err.Number & ":" & Err.Description, vbOKOnly, "Graphic Load Error"
FormCustomerGraphic.MousePointer = 0
On Error GoTo 0

BMPOpenError:
MsgBox "Directory for temporary imaging work does not exist." & vbCrLf & _
    "Please select a valid directory for image out.", vbOKOnly, "User path error"

Screen.MousePointer = vbDefault
On Error GoTo 0
End Sub

Sub GetGraphicChunk()
    Dim sKeyBuffer As String
    Dim iKeyBufLen As Integer

    BufLen = Len (ChunkReadBuffer)

```

Btrieve API プログラミング

```
sKeyBuffer = Space$(KEY_BUF_LEN)
iKeyBufLen = KEY_BUF_LEN

{ 次の構文では、キー番号にチャンク モードの -2 を設定する必要があります }

giStatus = BTRCALL(BGETDIRECT, GrphPosBlock, ChunkReadBuffer, _
                    BufLen, ByVal sKeyBuffer, iKeyBufLen, -2)
End Sub
```

サンプル構造体 (チャンク /BLOB/ 可変長レコード)

次のサンプル構造体は、前の Visual Basic のコード サンプルで使用される構造体です。

Visual Basic (チャンク /BLOB/ 可変長レコード) - サンプル構造体

```
Type GraphicRecordBufferType
    ContactNumber As typ_byte4
    NotUsed As String * 64
End Type

Type GraphicKeyBufferType
    BufferValue(255) As Byte
    CurrentKeyNumber As Integer
    ContactNumber As typ_byte4
    NotUsed As String * 64
End Type

Type ChunkReadDescriptorNext
    ChunkArray As String * MaxChunkSize
End Type

Type ChunkGetDescriptor
    RecordAddress As typ_byte4
    AddressMode As typ_byte4
    NumberOfChunks As typ_byte4
    ChunkOffset As typ_byte4
    ChunkLength As typ_byte4
    ChunkAddress As typ_byte4
End Type

Global ChunkGetBuffer As ChunkGetDescriptor
Global ChunkReadBuffer As ChunkReadDescriptorNext

' グラフィック テーブル変数
Global GrphPosBlock As Byte(0 to 127)
Global GrphRecBuf As GraphicRecordBufferType
Global GrphKeyBuffer As GraphicKeyBufferType
Global GrphFixedRecSize As Long
Global GrphFileLocation As String
Global GrphKeyNumber As Integer
Global GrphPosition As typ_byte4
```

セグメント化されたインデックスの処理

以下のサンプル コードは、セグメント化されたインデックスを処理する方法を示しています。

サンプル コード

Visual Basic (セグメント化されたインデックス)

```
Sub CreateCustomerFile(CustFileLocation As String)

    ' 次の構文は customer ファイルを作成し、ファイル仕様を設定します

    CustFixedRecSize = Len(CustRecBuf) - Len(CustRecBuf.Notes)
    FileDefinitionBuffer.RecLen = CustFixedRecSize
    FileDefinitionBuffer.PageSize = 4096
    FileDefinitionBuffer.IndxCnt = 4
    FileDefinitionBuffer.FileFlags = VARIABLELENGTH

    ' 以下はキー仕様を定義します

    ' キー 0、連絡先番号
    FileDefinitionBuffer.KeyBuf0.KeyPos = 1
    FileDefinitionBuffer.KeyBuf0.KeyLen = 4
    FileDefinitionBuffer.KeyBuf0.KeyFlags = EXTTYPE + MODIFIABLE
    FileDefinitionBuffer.KeyBuf0.KeyType = Chr$(BAUTOINC)

    ' キー 1、連絡先名
    FileDefinitionBuffer.KeyBuf1.KeyPos = 5
    FileDefinitionBuffer.KeyBuf1.KeyLen = 30
    FileDefinitionBuffer.KeyBuf1.KeyFlags = EXTTYPE + MODIFIABLE + DUP
    FileDefinitionBuffer.KeyBuf1.KeyType = Chr$(BSTRING)

    ' キー 2、連絡先名
    FileDefinitionBuffer.KeyBuf2.KeyPos = 35
    FileDefinitionBuffer.KeyBuf2.KeyLen = 60
    FileDefinitionBuffer.KeyBuf2.KeyFlags = EXTTYPE + MODIFIABLE + DUP
    FileDefinitionBuffer.KeyBuf2.KeyType = Chr$(BSTRING)

    ' キー 3、販売員、次回の連絡日

    ' これはセグメント キー
    FileDefinitionBuffer.KeyBuf3.KeyPos = 220
    FileDefinitionBuffer.KeyBuf3.KeyLen = 3
    FileDefinitionBuffer.KeyBuf3.KeyFlags = EXTTYPE + _ MODIFIABLE + DUP + SEGMENT
    FileDefinitionBuffer.KeyBuf3.KeyType = Chr$(BSTRING)
    FileDefinitionBuffer.KeyBuf4.KeyPos = 223
    FileDefinitionBuffer.KeyBuf4.KeyLen = 4
    FileDefinitionBuffer.KeyBuf4.KeyFlags = EXTTYPE + MODIFIABLE + DUP
    FileDefinitionBuffer.KeyBuf4.KeyType = Chr$(BDATE)

    BufLen = Len(FileDefinitionBuffer)
    CustFileLocation = CustFileLocation & " "
    KeyBufLen = Len(CustFileLocation)
    giStatus = BTRCALL(BCREATE, CustPosBlock, FileDefinitionBuffer, _
        BufLen, ByVal CustFileLocation, KeyBufLen, 0)
End Sub
```

Delphi (セグメント化されたインデックス)

セグメント化されたインデックスの作成に関するコードを参照するには、「[Delphi \(ファイルの作成\)](#)」コード サンプルの「キー 3」を参照してください。

```
var
  CustKeyBuffer : record                // セグメント化されたキー バッファ
    SalesRep      : array[0..2] of Char;
    NextContact   : DateType;         //Btrieve データ構造
  end;

  CustBufLen : Word;
  CustKeyNumber : ShortInt;
begin
  CustKeyNumber := 3;                  //SalesRep/Date 順
  CustKeyBuffer.SalesRep := 'TO';      // イニシャル TO の人物を検索
  CustKeyBuffer.NextContact.Day := 9;  //NextContact が 9/9/98
  CustKeyBuffer.NextContact.Month := 9;
  CustKeyBuffer.NextContact.Year := 1998;

  CustBufLen := SizeOf(CustRec);

  { 次の構文は、指定したソート順 (KeyNum) を使用してファイルから先頭レコードを取得します }
  Status := BTRV(B_GET_EQUAL,        //OpCode 5
    CustPosBLock,    // 既に開いているポジション ブロック
    CustRec,         // レコードが返されるバッファ
    CustBufLen,      // 返される最大長
    CustKeyBuffer,   // レコードから抽出したキー値を返す
    CustKeyNumber);  // 取得時に使用するインデックス順
```

C/C++ (セグメント化されたインデックス)

```
struct    // セグメント化されたキー バッファ
{
  char      SalesRep[3];
  date_type NextContact;    //Btrieve データ構造
} CustKeyBuffer;
BTI_WORD  CustBufLen;
char      CustKeyNumber;

CustKeyNumber = 3;          // SalesRep/Date 順
CustKeyBuffer.SalesRep = "TO"; // イニシャル TO の人物を検索
CustKeyBuffer.NextContact.Day = 9; // NextContact が 9/9/98
CustKeyBuffer.NextContact.Month = 9;
CustKeyBuffer.NextContact.Year = 1998;
CustBufLen = sizeof(CustRec);

/* 指定したソート順 (KeyNum) を使用してファイルから先頭レコードを取得する */
iStatus = BTRV(B_GET_EQUAL,    //OpCode 5
  CustPosBLock,    // 既に開いているポジション ブロック
  &CustRec,        // レコードが返されるバッファ
  &CustBufLen,     // 返される最大長
  CustKeyBuffer,   // 検索するレコードを指定
  CustKeyNumber);  // 取得時に使用するインデックス順
```

Visual Basic のための Btrieve API 関数の宣言

以下に、Visual Basic のための Btrieve API 関数の宣言を示します。

```
Declare Function BTRCALL Lib "w3btrv7.dll" (ByVal OP, ByVal Pb$, Db As Any, DL As Long, Kb As Any, ByVal Kl, ByVal Kn) As Integer
```

```
Declare Function BTRCALLID Lib "w3btrv7.dll" (ByVal OP, ByVal Pb$, Db As Any, DL As Long, Kb As Any, ByVal Kl, ByVal Kn, ByVal ID) As Integer
```

```
Declare Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" (hpvDest As Any, hpvSource As Any, ByVal cbCopy As Long)
```

PSQL データベースは、以下の 2 つの部分から構成されています。

- データを記述するデータ辞書
- データを物理的に取り込むデータ ファイル

この章では、名前付きデータベースとバウンド データベースについて説明するほか、データベースの作成方法についても説明します。それ以降のセクションでは、データベースの作成に伴うデータ辞書の作成、データベースのテーブル、列およびインデックスの作成方法について説明します。

- 「名前付きデータベース」
- 「バウンド データベース」
- 「データベース コンポーネントの作成」
- 「名前付け規則」
- 「データ辞書の作成」
- 「テーブルの作成」
- 「列の作成」
- 「インデックスの作成」

名前付きデータベース

名前付きデータベースには論理名があり、ユーザーはその論理名の実際の場所がわからなくても識別できます。データベースに名前を付ける際は、その名前を特定の辞書ディレクトリのパスおよび 1 つまたは複数のデータファイルのパスに関連付けるようにします。データベース名を使って **PSQL** にログインするとき、**PSQL** ではその名前を使って、データベースの辞書とデータ ファイルを検索します。データベースに名前を付けないと、以下のことを行えません。

- トリガーの定義
- 主キーと外部キーの定義
- データベースのバインド
- データベースの整合性制約の停止

既存のアンバウンド データベースに名前を付けたり、新しいバウンド データベースを作成したりするには、**PSQL Control Center** を使用します。詳細については、『**PSQL User's Guide**』を参照してください。

バウンド データベース

データベースをバインドすると、データのアクセスに使用する方法とは無関係に、MicroKernel はデータベースの定義済みセキュリティ、参照整合性 (RI) およびトリガーを設定できます。MicroKernel は、以下のようにこれらの整合性のコントロールを設定します。

- **バウンド** データベース上でセキュリティを定義すると、Btrieve ユーザーはそのデータベースにアクセスできません。
- **アンバウンド** データベース上でセキュリティを定義すると、Btrieve ユーザーはそのデータベースにアクセスできます。
- **バウンド** データベース上でセキュリティを定義しないと、Btrieve ユーザーは以下のようにデータ ファイルにアクセスできます。

バウンド ファイルの制約	Btrieve を使用するアクセスのレベル
RI 制約の定義あり	ユーザーは、RI 制約内ですべてのものにアクセスして実行できます。
INSERT トリガーの定義あり	読み取り専用、更新および削除アクセス
UPDATE トリガーの定義あり	読み取り専用、挿入および削除アクセス
DELETE トリガーの定義あり	読み取り専用、挿入および更新アクセス

バウンド ファイルに複数の制約が存在する場合、アクセス レベルは制限される最大限の制約に従います。たとえば、ファイルに INSERT トリガーと UPDATE トリガーが定義されている場合は、読み取り専用および削除アクセスが行えます。



メモ たとえデータベースをバインドしなくても、データ ファイルにトリガーがあるか、外部キーがあるか、あるいは外部キーで参照される主キーがある場合、PSQL はデータ ファイルにバウンドというスタンプを自動的に付けます。したがって、データ ファイルはアンバウンド データベースの一部であっても、バインドされていることになります。そのような場合、MicroKernel はそのファイルがバウンド データベースの一部であるかのようにファイルに整合性制約を設定します。

バウンド データベース内の辞書ファイルとデータ ファイルは、ほかの名前付きデータベースでは参照できません。また、バウンド データ ファイルはデータベース内のほかのテーブルでは参照できません。

バウンド データベースを作成するか、既存のデータベースをバインドする場合、PSQL はすべての辞書ファイルとデータ ファイルにバウンド データベースの名前を付けます。また、PSQL はすべてのデータ ファイルにそのデータ ファイルに関連するテーブルの名前を付けます。また、データベースに新しいテーブルまたは辞書ファイルを追加すると、PSQL はそれらを自動的にバインドします。

データベース コンポーネントの作成

データベースを作成するには、PSQL Control Center を使用します。『PSQL User's Guide』を参照してください。

データベースにテーブルを作成するには、PSQL Control Center を使用するか、『SQL Engine Reference』で定義されている CREATE TABLE 構文を使用します。CREATE TABLE ステートメントを発行する場合は、列を定義するコマンドを取り込む必要があります。また、参照整合性 (RI) 制約を定義するコマンドを取り込むこともできます。

名前付け規則

データベースを作成する場合、PSQL では各データベース コンポーネントに記述名を付けることができます。ユーザーとアプリケーションは、これらの名前でデータベースのコンポーネントを参照します。ここでは、データベース コンポーネントに名前を付けるときに従うべき規則について概説します。

詳細については、『Advanced Operations Guide』の「[識別子の種類別の制限](#)」を参照してください。

一意名

以下のデータベース コンポーネントは、辞書に一意名を持っていなければなりません。

- テーブル
- ビュー
- インデックス
- キー
- ユーザー名
- グループ名
- ストアド プロシージャ
- トリガー
- 1つのテーブル内の列名

パラメーターと変数の名前は、SQL ステートメント内で一意でなければなりません。PSQL キーワードは予約語であるため、データベース コンポーネントに名前を付けるためにそれらのキーワードを使用したり、パラメーター名や変数で使用することはできません。予約キーワードのリストについては、『SQL Engine Reference』の「[SQL の予約語](#)」を参照してください。

異なるテーブルで列名が重複している場合、関連するテーブル名またはエイリアス名を列名の前に置くことによって、各テーブル内に列名の修飾を行うことができます。たとえば、Student テーブルの ID 列を Student.ID として参照できます。これは、**完全修飾された列名**であり、テーブル名 (Student) は**列修飾子**です。

有効な文字

以下に、SQL レベルでのデータベース コンポーネントの名前に対する有効な文字と、変数およびパラメーター名に対する有効な文字を示します。

- a ~ z
- A ~ Z
- 0 ~ 9
- _ (アンダスコア)
- ^ (キャレット)
- ~ (チルダ)
- \$ (ドル記号)



メモ データベース コンポーネントの名前の先頭は文字でなければなりません。データベース コンポーネントの名前、またはこれらの規則に従わないパラメーター名を指定する場合は、"name" のように二重引用符で囲んで名前を指定します。

名前の最大長

PSQL では、辞書内のデータベース コンポーネント名の最大長に制限があります。『Advanced Operations Guide』の「[識別子の種類別の制限](#)」、および『SQL Engine Reference』の「[PSQL 機能の制限 / 条件](#)」を参照してください。

大文字と小文字の区別

PSQL は、データベース コンポーネント名を定義する場合に大文字と小文字を区別します。*TaBLel* という名前のテーブルを作成する場合、PSQL はテーブル名を *TaBLel* として辞書に格納します。ユーザー名、ユーザー グループ名およびパスワードを例外として、PSQL はコンポーネント名を定義した後に大文字と小文字を区別しません。テーブル *TaBLel* を定義した後、そのテーブルを *tablel* として参照できます。

ユーザー名、ユーザー グループ名およびパスワードは、PSQL で大文字と小文字を区別します。たとえば、マスター ユーザーとしてログインする場合、ユーザー名を *Master* として指定する必要があります。

データを取得する場合、作成された状態に基づいて、PSQL はテーブル、ビュー、エイリアスおよび列名を表示します。

```
SELECT *  
FROM Course#
```

PSQL は、以下のように列名を返します。

```
"Name", "Description", "Credit_Hours", "Dept_Name"
```


データ辞書の作成

PSQL は、辞書を使用してデータベースに関する情報を格納します。辞書は、データベースのテーブルとビューを記述するいくつかのシステム テーブルから構成されています。

システム テーブルには、インデックス定義、例の特性、保全性とセキュリティ情報などの数種類のデータベース情報が含まれています。表 41 は PSQL が作成するシステム テーブルを示しています。『SQL Engine Reference』のツール、「[システム テーブル](#)」も参照してください。

表 41 PSQL システム テーブル

操作	結果テーブル
データ辞書の作成	X\$File、X\$Field、X\$Index
列の属性の指定	X\$Attrib
ストアド SQL プロシージャの作成	X\$Proc
データベース セキュリティの定義	X\$User、X\$Rights
参照制約の定義	X\$Relate
ビューの定義	X\$View
トリガーの定義	X\$Trigger、X\$Depend

システム テーブルはデータベースの一部であるため、システム テーブルに照会してそれらの内容を決定できません。適切な権利があれば、システム テーブルを作成したり、それらの内容を変更することもできます。



メモ PSQL は、システム テーブル内のいくつかのデータを表示しません。たとえば、ストアド ビューおよびストアド プロシージャの名前以外の情報は、PSQL でしか使用できません。また、ユーザー パスワードなどのいくつかのデータは暗号化された形式で表示されます。

各システム テーブルの内容をすべて参照する場合は、『SQL Engine Reference』を参照してください。

辞書を作成すると、データベースにテーブル、列およびインデックスを追加できます。

➤ 名前付きデータベースの作成は、以下の手順で行います。



メモ 参照整合性やトリガーなどのいくつかの機能を使用するには、名前付きデータベースが必要です。

- 1 新しい辞書テーブルを格納するためのディレクトリを作成します。
- 2 名前付きデータベースを追加するには、PSQL Control Center を使用します。『PSQL User's Guide』の「[新規データベースを作成するには](#)」を参照してください。

➤ 名前なしデータベースの辞書の作成は、以下の手順で行います。

- 1 PSQL DDF Builder を実行します。
- 2 データ辞書ファイル (DDF) の作成については、『DDF Builder User's Guide』の情報を確認してください。「[DDF Builder について](#)」を参照してください。

テーブルの作成

テーブルを作成する場合は、テーブルに名前を付ける必要があります。各テーブル名は、データベース内で固有の名前である必要があります。テーブル名を付ける規則の詳細については、「[名前付け規則](#)」を参照してください。

どのテーブルをデータベースに作成するかを決定する場合は、さまざまなユーザーがビューを使用してさまざまな組み合わせでデータを見ることができるように考慮してください。ビューはテーブルに似ており、多くの目的、たとえば、データの取得、更新、削除などの目的でテーブルとして処理することができます。しかし、ビューは必ずしも 1 つのテーブルだけに関連付けられているわけではありません。ビューは、複数のテーブルから情報を組み合わせることができます。詳細については、「[データの取得](#)」を参照してください。

PSQL Control Center を使用してテーブルを作成することができます。『PSQL User's Guide』の「[新規テーブルのために Table Editor を起動するには](#)」を参照してください。

エイリアス

以下のステートメント要素内のテーブル名にエイリアス（**エイリアス名**とも呼ぶ）を割り当てることができます。

- SELECT または DELETE ステートメントの FROM 句
- INSERT ステートメントの INTO 句
- UPDATE ステートメント内のテーブルのリスト



メモ エイリアスは、エイリアスを使用するステートメントにのみ適用されます。PSQL は、データ辞書にエイリアスを格納しません。

エイリアスは、最大 20 文字の組み合わせとすることができます。テーブル名とエイリアス名は常に空白で区切ります。エイリアスと列名はピリオド (.) で区切ります。一度特定のテーブルのエイリアスを指定したら、ステートメント内であればどこでも、テーブルの列名の修飾に使用することができます。

以下の例では、Student テーブルにエイリアス名 *s* を、Enrolls テーブルにエイリアス名 *e* を指定しています。

```
SELECT s.ID, e.Grade
      FROM Student s, Enrolls e
      WHERE s.ID = e.Student_ID#
```

エイリアスを使用して以下のことが行えます。

- 長いテーブル名を置き換える。

対話形式で作業している場合、エイリアスを使用すると、特に列名を修飾しなければならないときにキーボード入力時間を節減できます。たとえば、以下のステートメントではエイリアスとして、Student テーブルには *s*、Enrolls テーブルには *e*、Class テーブルには *c1* を割り当てています。この例では、エイリアスを使用して、選択リストと WHERE 条件の各列のソースを区別しています。

```
SELECT s.ID, e.Grade, c1.ID
      FROM Student s, Enrolls e, Class c1
      WHERE (s.ID = e.Student_ID) AND
            (e.Class_ID = c1.ID)#
```

- ステートメントを読みやすくします。単一のテーブル名を持つステートメントでも、エイリアスはステートメントを読みやすくすることができます。
- 関連されたサブクエリ内の外側のクエリのテーブルを使用します。

```
SELECT s.ID, e.Grade, c1.ID
      FROM Student s, Enrolls e, Class c1
      WHERE (s.ID = e.Student_ID) AND
            (e.Class_ID = c1.ID) AND
            e.Grade >= (SELECT MAX (e2.Grade)
                        FROM Enrolls e2
                        WHERE e2.Class_ID = e.Class_ID)#
```

列の作成

CREATE TABLE ステートメントを使用してテーブルを作成する際に列を作成するか、ALTER TABLE ステートメントを使用して既存のテーブルに列を追加することができます。いずれの場合も、以下の特性を指定する必要があります。

- 列名一列を識別します。各列名は、テーブル内で固有の名前にする必要があり、また、20 文字を超えることはできません。PSQL は、データベースの列名を定義する場合に大文字と小文字を区別しますが、列名を定義した後は大文字と小文字を区別しません。たとえば、*ColuMNI* という列を作成する場合、名前は *ColuMNI* として辞書に格納されます。それ以降は、*column1* としてその名前を参照できます。列に名前を付ける規則の詳細については、「[名前付け規則](#)」を参照してください。
- データ型一文字列や数字など、予想するデータの種類の、割り当てるディスク保存領域を識別します。データ型の詳細については、『[Btrieve API Guide](#)』を参照してください。

インデックスの作成

インデックスは、特定の値を検索する操作、または特定の値によって並べ替える操作を最適化します。これらの操作のいずれかを頻繁に実行するすべての列に対し、インデックスを定義します。インデックスは、クエリの最適化において、特定の行または行のグループに対する高速の取得方法を提供します。PSQL は、参照整合性 (RI) 付きのインデックスも使用します。インデックスは結合におけるパフォーマンスを向上し、クエリを最適化しやすくします。RI の詳細については、『PSQL User's Guide』を参照してください。

PSQL データベースでは、MicroKernel エンジンが定義する物理ファイルの一部としてインデックスを作成し、管理します。MicroKernel エンジン は、Insert、Update、または Delete オペレーションのすべての管理を行います。これらのアクティビティは、すべての PSQL アプリケーションに対して透過的です。

インデックスを作成するには、CREATE INDEX ステートメントを使用します。この方法では、名前付きインデックスを作成します。名前付きインデックスを作成した後、そのインデックスを削除できます。インデックスの削除の詳細については、第 14 章「[データの挿入と削除](#)」を参照してください。

インデックスを使用して行をソートしたり個々の行を高速に取得できますが、データベースのディスク保存領域が増加し、Insert、Update、または Delete オペレーションにおけるパフォーマンスが多少低下します。インデックスを定義するときは、これらの相殺条件を考慮してください。

次の例では、CREATE INDEX ステートメントを使用して、既に存在するテーブルにインデックスを追加します。

```
CREATE INDEX DeptHours ON Course (Dept_Name, Credit_Hours) #
```



メモ 多数のデータを含むファイルで CREATE INDEX ステートメントを使用する場合は、実行が終了するまでにある程度の時間がかかり、その間はほかのユーザーがそのファイル内のデータにアクセスできないことに注意してください。

CREATE TABLE ステートメントと CREATE INDEX ステートメントの詳細については、『SQL Engine Reference』を参照してください。

インデックス セグメント

同じテーブル内の単一の列または列のグループ上にインデックスを作成できます。複数の列を含むインデックスをセグメント化されたインデックスと呼び、各列をインデックス セグメントと呼びます。

たとえば、サンプルデータベースの Person テーブルには以下の 3 つのインデックスがあります。

- Last Name 列と First Name 列から成るセグメント化されたインデックス
- Perm_State + Perm_City 列
- ID 列

インデックス セグメントの数は、データ ファイルのページ サイズの影響を受けます。PAGESIZE キーワードの使用法の詳細については、『Btrieve API Guide』を参照してください。テーブルに対して作成できるインデックスの最大数は、データ ファイルのページ サイズと各インデックス内のセグメント数により異なります。表 42 に示すように、ページ サイズが 4096 バイトより小さいデータ ファイルには、ページ サイズ 4096 のデータ ファイルと同じ個数のインデックス セグメントを収容できません。使用するインデックス セグメントの数はファイルのページ サイズによって異なります。

表 42 データ ファイルあたりの最大インデックス セグメント数

ページ サイズ (バイト数)	キー セグメントの最大数 (ファイル バージョン別)		
	8.x 以前	9.0	9.5
512	8	8	切り上げ ²
1,024	23	23	97
1,536	24	24	切り上げ ²
2,048	54	54	97
2,560	54	54	切り上げ ²
3,072	54	54	切り上げ ²
3,584	54	54	切り上げ ²
4,096	119	119	204
8,192	N/A ¹	119	420
16,384	N/A ¹	N/A ¹	420
¹ N/A は「適用外」を意味します。 ² 「切り上げ」は、ページ サイズを、ファイル バージョンでサポートされる次のサイズへ切り上げることを意味します。たとえば、512 は 1,024 に切り上げられ、2,560 は 4,096 に切り上げるということです。			

『Status Codes and Messages』 マニュアルで、インデックス セグメントと MicroKernel エンジンに関連情報について、ステータス コード 26 " 指定されたキーの数不正です " およびステータス コード 29 " キー長不正です " を参照してください。

ページ サイズと固定レコード長を使用して、データが格納されている効率性、たとえば、ページあたりの無駄に使用されているバイト数などを計算することができます。ページあたりのレコード数を少なくすることによって、ページレベル ロックでのロックが問題となる並行処理を改善することができます。

デフォルトでは、PSQL はすべてのテーブルをページ サイズ 4096 バイトで作成します。ただし、CREATE TABLE ステートメントの PAGESIZE キーワードを使用してより小さなページ サイズを指定したり、MicroKernel Database エンジンを使用してテーブルを作成し、そのテーブルにより小さなページ サイズを指定することができます。

テーブルに対して定義されたインデックス セグメントの総数を計算する場合、セグメント化されていないインデックスは 1 つのインデックス セグメントとしてカウントされます。たとえば、テーブルに 3 つのインデックスが定義されていて、そのうちの 1 つに 2 つのセグメントがある場合、インデックス セグメントの総数は 4 です。

PSQL Control Center を使用して、定義されたインデックス セグメント数とデータ ファイルのページ サイズを表示できます。このユーティリティの詳細については、『PSQL User's Guide』を参照してください。

インデックス属性

インデックスを作成する場合は、インデックスに一連の特性、つまり、**属性**を割り当てることができます。インデックス属性は、インデックスの変更可能性と、テーブルに定義するインデックスを PSQL がどのようにソートするかを決定します。インデックス定義を作成または変更するたびに、インデックス属性を指定するパラメータを取り込むことができます。

データベースの作成

インデックスは、以下の属性を持つことができます。

大文字と小文字の区別	PSQL がソート中に大文字と小文字をどのように評価するかを決定します。デフォルトでは、PSQL は大文字と小文字を区別するインデックスを作成します。大文字と小文字を区別するインデックスを作成するには、インデックスを作成するときに CASE キーワードを指定します。
ソート順	PSQL がどのようにインデックス列の値をソートするかを決定します。デフォルトの設定で、PSQL ではインデックス列の値を昇順（小さいものから大きなものへ）にソートします。降順にソートするインデックスを作成するには、インデックスを作成するときに DESC キーワードを指定します。
重複不可	PSQL を使用して複数の行が同じインデックス列の値を持つことができるかどうかを決定します。デフォルトでは、PSQL は一意でないインデックスを作成します。一意の値を必要とするインデックスを作成するには、インデックスを作成するときに UNIQUE キーワードを指定します。
変更可能性	PSQL が対応する行をソートした後でインデックス列の値を変更できるかどうかを決定します。デフォルトでは、PSQL が行を格納すると、PSQL でインデックス列の値を変更できません。変更可能なインデックスを作成するには、インデックスを作成するときに MOD キーワードを指定します。
セグメント化	<p>インデックスがセグメント化されるかどうか、つまり、1つのインデックスに結合された列のグループからインデックスを構成するかどうかを指示します。デフォルトでは、PSQL はセグメント化されないインデックスを作成します。CREATE TABLE ステートメントを使用してセグメント化されたインデックスを作成するには、インデックスの最後のセグメントを除き、作成する各インデックス セグメントに SEG キーワードを指定します（SEG キーワードは、指定された次の列が作成するインデックスのセグメントであることを指示します）。</p> <p>CREATE INDEX コマンドでは一度に 1 つのインデックスだけしか作成できないため、SEG キーワードを使用してセグメント インデックスを指定する必要はありません。複数の列を指定する場合、PSQL は列を指定する順に列を使用してセグメント化されたインデックスを作成します。</p>
一部のみ	<p>列とオーバーヘッドの合計サイズが 255 バイト以上のときに、PSQL が CHAR 列または VARCHAR 列の一部を使用するかどうか、最後または唯一のインデックス列として設計するかどうかを示します。</p> <p>デフォルトでは、PSQL は部分インデックスを作成しません。CREATE INDEX ステートメントを使用して部分インデックスを作成するには、PARTIAL キーワードを指定します。</p>

重複不可能性と変更可能性は、インデックス全体だけに適用されます。重複不可能性または変更可能性は、インデックス全体に適用しなければ、単一のインデックス セグメントに適用することはできません。たとえば、セグメント化されたインデックスを作成し、インデックス セグメントのうちの 1 つに **MOD** キーワードを指定する場合、すべてのセグメントに対して **MOD** キーワードを指定する必要があります。

それに対して、インデックス全体に影響を与えずに個々のインデックス セグメントに大文字と小文字の区別、ソート順序、セグメント化を適用できます。たとえば、大文字と小文字を区別するインデックスに大文字と小文字を区別しないインデックス セグメントを作成できます。

以下の条件を満たせば、部分インデックスはインデックス内に定義された最後の列にのみ適用されます。

- その列がインデックスに定義された唯一の列である、または、インデックスに定義された最後の列であること
- 最後のインデックス列のデータ型が **CHAR** または **VARCHAR** であること
- 列のオーバーヘッドを含めたインデックスの合計サイズが 255 バイト以上であること

インデックスの作成および使用可能な属性の詳細については、『SQL Engine Reference』の「[CREATE INDEX](#)」を参照してください。

リレーショナル データベース設計

13

この章では、以下の項目について説明します。

- 「データベース設計の概要」
- 「設計の段階」

データベース設計の概要

この章では、リレーショナル データベース設計の基本原則について概説します。開発プロセス全体にわたる完全なデータベース設計は、データベースの機能とパフォーマンスの成功に不可欠です。

サンプル データベースの DEMODATA は PSQL の一部として提供されており、データベースの概念と技法を図解するためにマニュアルで頻繁に使用されます。テーブル、行、列などの基本的なリレーショナル データベース概念の定義については、PSQL オンライン ヘルプの用語集を参照してください。

設計の段階

リレーショナル データベースの基本的な構造を理解すれば、データベース設計プロセスを開始することができます。データベースの設計は、ビジネスに応じたデータベース構造の開発と精製に関連するプロセスです。

データベースの設計には、以下の 3 つの段階があります。

- 1 概念データベース設計
- 2 論理データベース設計
- 3 物理データベース設計

概念設計

データベース設計サイクルの最初のステップは、業務に必要なデータを定義することです。以下のような質問に答えることで、概念設計をより明確に定義できます。

- 対象となる業務では現在、どのような種類の情報を使用しているか。
- 対象となる業務では、どのような種類の情報が必要か。
- 設計しようとしているシステムからどのような情報が必要か。
- 業務を遂行する場合の前提条件は何か。
- 対象となる業務の制限は何か。
- どのようなレポートを生成する必要があるか。
- この情報で何を行うのか。
- 設計しようとしているシステムでは、どのようなセキュリティが必要か。
- 今後、どのような種類の情報を拡大していく必要があるか。

業務の目標を明確に定め、データベースを使用することになるスタッフからの意見を収集することは不可欠なプロセスです。この情報で、テーブルと列を効率的に定義できます。

論理設計

論理データベース設計により、以降のビジネスの情報要求の定義と評価を容易に行えます。論理データベース設計では、追跡しなければならない情報とこれらの各情報間の関係を記述します。

論理設計を終了すると、ユーザーと共にデータベースの設計が完全かつ正確であるかどうかを検証できます。ユーザーは、追跡しなければならないすべての情報が設計に含まれているかどうかを判断したり、設計が業務の流れに応じた情報の関係を反映しているかどうかを判断することができます。

論理データベース設計の作成は、以下の手順で行います。

- 1 概念設計で決定したようなビジネスで要求される情報に基づいて必要なテーブルを定義します。
- 2 テーブル間の関係を決定します（詳細については、「[テーブルの関係](#)」を参照してください）。
- 3 各テーブルの内容（列）を決定します。
- 4 テーブルを少なくとも第 3 正規形に正規化します（詳細については、「[正規化](#)」を参照してください）。
- 5 主キーを決定します（詳細については、「[キー](#)」を参照してください）。
- 6 各列の値を決定します。

テーブルの関係

リレーショナル データベースで、共通の列を共有することによってテーブルを相互に関連付けます。この列は複数のテーブルに存在し、テーブルの結合を可能にします。テーブルの関係には 3 つのタイプがあります。つまり、1 対 1、1 対多、多対多の関係です。

「1 対 1 の関係」は、あるテーブルの各行が第 2 テーブルの 1 つの行にのみ関連付けられる場合に生じます。たとえば、ある大学では 1 つの部屋に 1 人の教職員を割り当てることに決めています。したがって、1 つの部屋には一度に 1 人の教官しか割り当てることができません。またその大学では、1 つの学部には 1 人の学部長しか任命できないことに決めています。したがって、1 人しか学部長になれません。

「1 対多の関係」は、あるテーブルの各行が別のテーブルの多数の行に関連付けられる場合に生じます。たとえば、1 人の教官が多数のクラスを教えることができます。

「多対多の関係」は、あるテーブルの 1 つの行が第 2 テーブルの多数の行に関連付けられる場合に生じます。同様に、これらの関連付けられた行は第 1 テーブルの多数の行とも関連付けられます。たとえば、1 人の学生が多数の講座に登録できると同時に、それらの講座も多数の学生を受け入れることができます。

正規化

正規化は、データベース内の冗長性を低下させて安定性を高めるプロセスです。正規化は、特定のデータがどのテーブルに属しているか、また、ほかのデータとの関係はどうであるかを決定します。その結果、対象となるデータベースは、プロセスまたはアプリケーション駆動型でなく、さらに安定したデータベース実装を提供するデータ駆動型の設計になります。

データベースを正規化する場合、以下の列を排除します。

- 複数のアトミックでない値を含む列
- 重複または反復する列
- テーブル名で修飾されていない列
- 冗長データを含む列
- ほかの列から派生可能な列

第 1 正規形

第 1 正規形の列には、以下の特性があります。

- 1 つのアトミック値しか含まれていない。
- 反復しない。

正規化の第 1 の規則は、「重複する列または複数の値を含む列を、新しいテーブルへ移動しなければならない」というものです。

第 1 正規形に正規化されたテーブルには、いくつかの利点があります。たとえば、サンプル データベースの Billing テーブルでは、第 1 正規形によって以下が実行されます。

- 新しい列を追加しなくても、学生ごとに任意の数のトランザクションを作成できます。
- 1 つの列（トランザクション番号）しか検索しないため、トランザクションのデータをすばやく照会またはソートできます。
- 空の列は格納されないため、ディスク領域を効率よく使用できます。

第 2 正規形

第 1 正規形のテーブルで、そのテーブルのキーに関する情報を提供する列のみを含んでいるテーブルは、第 2 正規形です。

正規化の第 2 の規則を実施するには、現在のテーブルの主キーに依存しない列を新しいテーブルへ移動しなければなりません。

テーブルに冗長データが含まれている場合、そのテーブルは第 2 正規形に違反します。このため、一貫性のないデータが生じ、データベースの整合性を欠くことになります。たとえば、学生が住所を変更した場合には、新しい住所を反映させるよう、既存のすべての行を更新する必要があります。古い住所を含んでいる行は、一貫性のないデータとなってしまいます。

データが冗長であるかどうかを判断するには、トランザクションを追加しても変化しないデータを確認します。**Student Name** や **Street** のような列はトランザクションに関係なく、主キーの **Student ID** によって異なります。したがって、この情報は **Student** テーブルに格納し、トランザクション テーブルには格納しません。

第2正規形に正規化されたテーブルにも、いくつかの利点があります。たとえば、サンプル データベースの **Billing** テーブルでは、第2正規形によって以下が実行されます。

- 学生情報を1行だけで更新できます。
- 必要な学生情報を消去せずに、学生に関するトランザクションを削除できます。
- 反復するデータおよび冗長データは格納されないため、ディスク領域をさらに効率よく使用できます。

第3正規形

テーブルに独立した列だけが含まれているとき、そのテーブルは第3正規形です。

正規化の第3の規則は、「既存の列から派生できる列を除去しなければならない」というものです。たとえば、ある学生の場合、**Date of Birth** 列が既にあるならば、**Age** 列を含める必要はありません。年齢は誕生日から計算できるからです。

第3正規形のテーブルには必要な列だけが含まれており、不要なデータが格納されないことから、ディスク領域をさらに効率よく使用できます。

要約すると、第1、第2、および第3正規形の規則が示していることは、各列の値は主キー全体に関する事実でなければならない、つまり主キーにほかならないということです。

キー

ODBC キーは、テーブルの参照整合性 (RI) の制約が定義されている列または列のグループです。つまり、キーまたは複数のキーの組み合わせは、行に含まれるデータの識別子として機能します。

参照整合性とキーの詳細については、『**Advanced Operations Guide**』を参照してください。

物理設計

物理データベース設計は、論理設計をより洗練されたものにすることです。物理データベース設計は、論理設計をリレーショナル データベース管理システムにマップします。この段階で、ユーザーがデータベースにアクセスする方法を調べます。データベース設計サイクルのこのステップでは、以下の種類の情報を決定します。

- よく利用するデータ。
- データ アクセスのためのインデックスを必要とする列。
- 柔軟性と拡張のためのスペースを必要とする領域。
- データベースを非正規化することでパフォーマンスが向上するかどうか(データベースを非正規化するには、パフォーマンスを満たすために冗長性を再導入します)。正規化の詳細については、「**正規化**」を参照してください。

データの挿入と削除

14

この章では、以下の項目について説明します。

- 「[データの挿入および削除の概要](#)」
- 「[値の挿入](#)」
- 「[トランザクションの処理](#)」
- 「[データの削除](#)」
- 「[インデックスの削除](#)」
- 「[列の削除](#)」
- 「[テーブルの削除](#)」
- 「[データベース全体の削除](#)」

データの挿入および削除の概要

データ辞書、テーブルおよび列を作成した後、SQL Data Manager を使用してデータベースにデータを追加できます。SQL ステートメントを使用すると、以下を行うことができます。

- 挿入するリテラル値を指定する。
- ほかのテーブルからデータを選択し、その結果の値を行全体または指定された列に挿入する。

リテラル値を挿入するには、その値が指定された列のデータ型および長さに適合していなければなりません。

データベースから行、インデックス、列またはテーブルを削除（**drop**）できます。さらに、不要になったデータベース全体を削除することもできます。

値の挿入

INSERT ステートメントで VALUES 句を使用して、データベースに挿入するリテラル値を指定できます。以下の例では、サンプルデータベースの **Course** テーブルに新しい行を挿入しています。

```
INSERT INTO Course
VALUES ('ART 103', 'Principles of Color', 3, 'Art');
```

この例では、ステートメントはテーブルの各列の値を順番どおりに挿入するため、列名の **Name**、**Description**、**Credit_Hours**、および **Dept_Name** を列挙することは任意です。しかし、ステートメントが行全体ではなく選択した行にのみデータを挿入する場合や、テーブルに定義されている順序とは異なる順序で列にデータを挿入する場合は、列のリストが必要になります。

INSERT ステートメントの詳細については、『SQL Engine Reference』で「[INSERT](#)」を参照してください。

トランザクションの処理

テーブルにデータを挿入しようとしたとき、そのデータが無効である場合には、PSQL がエラーを返します。エラーが発生する前に挿入されたデータはすべてロールバックされます。この結果、データベースを安定した状態に保つことができます。

PSQL データベースでトランザクション処理を使用して、論理的に関連付けられた一連のステートメントをグループ化することができます。トランザクションの中でセーブポイントを使用すると、トランザクションを効果的にネストさせることができます。あるネスト レベルのステートメントが失敗した場合、そのネスト レベルにある一連のステートメントがセーブポイントにロールバックされます。トランザクション処理とセーブポイントの詳細については、『SQL Engine Reference』の以下のトピックを参照してください。

- 「[START TRANSACTION](#)」
- 「[COMMIT](#)」
- 「[ROLLBACK](#)」
- 「[SAVEPOINT](#)」
- 「[RELEASE SAVEPOINT](#)」

データの削除

DELETE ステートメントには、位置付けと検索の 2 種類があります。

DELETE ステートメントを使用して、テーブルまたは更新可能なビューから 1 つまたは複数の行を削除できます。PSQL で削除する特定の行を指定するには、DELETE ステートメントの WHERE 句を使用します。

```
DELETE FROM Class
WHERE ID = 005#
```

位置付け DELETE ステートメントは、開いている SQL カーソルに関連するビューの現在の行を削除します。

```
DELETE WHERE CURRENT OF mycursor;
```

DELETE ステートメントの詳細については、『SQL Engine Reference』で「[DELETE](#)」を参照してください。

インデックスの削除

名前付きインデックスが不要になった場合は、DROP INDEX ステートメントを使用して削除します。

```
DROP INDEX DeptHours#
```

DROP INDEX ステートメントの詳細については、『SQL Engine Reference』で「[DROP INDEX](#)」を参照してください。

列の削除

テーブルから列を削除するには、ALTER TABLE ステートメントを使用します。

```
ALTER TABLE Faculty  
    DROP Rsch_Grant_Amount#
```

この例では、Faculty テーブルから Rsch_Grant_Amount 列を削除し、データ辞書から列の定義を削除します。

ALTER TABLE ステートメントの詳細については、『SQL Engine Reference』で「[ALTER TABLE](#)」を参照してください。



メモ 多数のデータを含むファイルで ALTER TABLE ステートメントを使用する場合は、実行が終了するまでにある程度の時間がかかります。実行中はほかのユーザーはこのファイル内のデータにアクセスできなくなることに注意してください。

テーブルの削除

データベースからテーブルを削除するには、DROP TABLE ステートメントを使用します。

```
DROP TABLE Student#
```

この例では、データ辞書から InactiveStudents テーブル定義を削除し、さらに対応するデータ ファイル (INACT.MKD) を削除します。

DROP TABLE ステートメントの詳細については、『SQL Engine Reference』で「[DROP TABLE](#)」を参照してください。



メモ システム テーブルは削除できません。システム テーブルを網羅したリストについては、『SQL Engine Reference』を参照してください。

データベース全体の削除

特定のデータベースが不要になった場合は、PSQL Control Center の SQL Data Manager を使用してそのデータベースを削除できます。詳細については、『PSQL User's Guide』を参照してください。

データの変更

15

この章では、以下の項目について説明します。

- 「[データ変更の概要](#)」
- 「[テーブルの変更](#)」
- 「[デフォルト値の設定](#)」
- 「[UPDATE ステートメントの使用](#)」

データ変更の概要

データベースを作成した後、以下のようにデータベースを変更できます。

- テーブルを作成した後、テーブル定義を変更できます。
- 列を作成した後、オプションの列属性を設定できます。
- データベースにデータを追加した後、データを変更できます。

SQL Data Manager を使用してこれらのタスクを実行できます。対話形式アプリケーションの詳細については、『PSQL User's Guide』を参照してください。SQL ステートメントの詳細については、『SQL Engine Reference』を参照してください。

テーブルの変更

テーブルの作成後にテーブル定義を変更するには、ALTER TABLE ステートメントを使用します。ALTER TABLE ステートメントでは、列の追加と削除、主キーおよび外部キーの追加と削除、テーブルのデータ ファイルのパス名の変更を行うことができます。

次の例は、サンプル データベースの Tuition テーブルに Emergency_Phone という数値型の列を追加します。

```
ALTER TABLE Tuition ADD Emergency_Phone NUMERIC(10,0) #
```

列の詳細については、第 14 章「[データの挿入と削除](#)」を参照してください。主キーと外部キーの詳細については、第 18 章「[データの管理](#)」を参照してください。

デフォルト値の設定

行を挿入しても列に値を指定しないと、PSQL がデフォルト値を挿入します。デフォルト値は、各行の列に有効な値が確実に入るようにします。

サンプル データベースの **Person** テーブルでは、すべての学生は同一の州内に住んでいます。**State** 列に **TX** などのデフォルト値を設定することにより、その列に対し最も可能性の高い値が常に入力されるようにします。

列のデフォルト値を設定するには、**CREATE TABLE** ステートメントで **DEFAULT** ステートメントを使用します。

```
CREATE TABLE MyTable(c1 CHAR(3) DEFAULT 'TX', ID INTEGER) #  
SELECT * FROM MyTable#
```

SELECT ステートメントの結果

```
"c1", "ID"
```

```
"TX", "1234"
```

2 列から 1 行フェッチされました。

UPDATE ステートメントの使用

既にテーブル内に存在する行の中のデータを変更するには、UPDATE ステートメントを使用します。UPDATE ステートメントでは、行の特定の列を変更できます。また、UPDATE ステートメントの WHERE 句を使用して、PSQL がどの行を変更するかを指定できます。これを検索更新 (Searched Update) と呼びます。SQL で宣言されたカーソルと位置付け UPDATE ステートメントを使用して、データをフェッチする宣言済みカーソルの現在の行を更新できます。

```
UPDATE Course
  SET Credit_Hours = 4
  WHERE Course.Name = 'Math' #
```

この例では、Math というコース名を含む行を検索し、Credit Hours 列の値を 4 に変更するよう PSQL に指示しています。

前の例で示されているように、UPDATE ステートメントの SET 句の右側に定数を置くことによって、列を更新できます。

この章では、SELECT ステートメントを使用して以下のタスクを実行する方法について説明します。

- 「[データ取得の概要](#)」
- 「[ビュー](#)」
- 「[選択リスト](#)」
- 「[ソートされた行とグループ化された行](#)」
- 「[結合](#)」
- 「[サブクエリ](#)」
- 「[制限句](#)」
- 「[関数](#)」

データ取得の概要

データベースにデータを入力したら、SELECT ステートメントでそのデータを取得し、表示することができます。PSQL は、要求するデータを**結果テーブル**で返します。SQL ステートメントでは、以下のことが行えます。

- テンポラリ ビューまたはパーマネント（ストアド）ビューを作成する。
- データベース内の 1 つまたは複数のテーブルから取り出す列を列挙する、**選択リスト**を指定する。
- 行をソートする方法を指定する。
- 行をサブセットにグループ化する場合の基準を指定する。
- テーブルにテンポラリ名（エイリアス）を割り当てる。
- 1 つまたは複数のテーブルからデータを取得し、1 つの結果テーブルにデータを表示する（結合）。
- SELECT ステートメントで**サブクエリ**を指定する。
- PSQL が選択する行を制限するために**制限句**を指定する。

ビュー

ビューは、データベース内のデータを調べるためのメカニズムです。複数のテーブルのデータを結合したり、1つのテーブルの特定の列だけを含めたりすることができます。ビューはテーブルに似ていますが、データベースのテーブルの列に基づいて選択した一連の列または計算結果から構成されています。したがって、ビューには、複数のテーブル内の列からのデータ、または実際にテーブル内にまったくないデータ、たとえば、SELECT COUNT (*) FROM Person などが含まれている場合があります。

ビューの機能

以下に、ビューのいくつかの機能を示します。

- ビューの列は、可変長の列が最後の列でなければならないということ以外、任意の順序で配列することができます。可変長の列は1つしか指定できません。
- 制限句を使用して、PSQL がビューに返す行のセットを指定できます。制限句は、データをビューに取り込む必要のある条件を指定します。詳細については、「[制限句](#)」を参照してください。
- データベースにアクセスするユーザーとアプリケーションごとに、ビューの設計とカスタマイズが行えます。これらのビューの定義は、後で呼び出すためにデータ辞書に格納できます。
- ビューが読み取り専用のビューでない限り、データを取得、更新、または削除する際に、テーブル リストにストアド ビュー名をいくつでも含めることができます。読み取り専用のビューでは、データの取得しか行えません。
- ストアド ビューでは、ビューの計算列と定数に見出しを付け、ビューからデータを取得するときにこれらの見出しの名前を列名のリストで使用しなければなりません。

テンポラリ ビューとストアド ビュー

SELECT ステートメントを使用して、テンポラリ ビューまたはストアド ビューを作成できます。テンポラリ ビューは1回だけ使用して、その後で解放するものです。PSQL はストアド ビューの定義をデータ辞書 (XSProc) に保管するので、後でそのビューを呼び出すことができます。CREATE VIEW ステートメントを使用すると、ストアド ビューの作成と名前付けが行えます。

各ビューはデータベース内で一意であり、また 20 文字を超えることはできません。ビューに名前を付ける規則の詳細については、第 14 章「[データの挿入と削除](#)」を参照してください。

PSQL は、データベース要素名を定義する場合に大文字と小文字を区別します。PhoNE という名前のストアド ビューを作成すると、PSQL は辞書にビュー名を PhoNE として格納します。ビュー名の定義後、PSQL は大文字小文字を区別しません。ストアド ビュー PhoNE を定義した後、そのビューを *phone* で参照することができます。ストアド ビューを使用すると、以下の機能が実現します。

- 頻繁に実行するクエリを格納し、後で使用するために名前を付けることができます。以下の例では、Department テーブルに基づいて Phones というストアド ビューを作成します。

```
CREATE VIEW Phones (PName, PPhone)
AS SELECT Name, Phone_Number
FROM Department#
```

- データの取得、更新および削除を行う場合、テーブル リストでストアド ビューの名前を指定できます。ストアド ビューはデータベース内のテーブルであるかのように動作しますが、実際には使用する都度、PSQL エンジンによって内部で再構築されます。以下の例では、ストアド ビュー Phones を参照することによって、Department テーブル内の History Department の電話番号を更新します。

```
UPDATE Phones
SET PPhone = '5125552426'
WHERE PName = 'History'##
```

- 見出しを指定できます。見出しでは、辞書の列に対して定義した名前とは異なる列名を指定します。以下の例では、ストアド ビュー Phones に見出し Department と Telephone を指定します。

データの取得

```
CREATE VIEW Dept_Phones (Department, Telephone)
AS SELECT Name, Phone_Number
FROM Department#
```

以下の例に示すように、ビューの以降のクエリ内で見出しを使用できます。

```
SELECT Telephone
FROM Dept_Phones#
```

選択リストに単純な列名が含まれている場合に、見出しを指定しないと、PSQL はその列名を列の見出しとして使用します。

ビューに含める定数と計算列に名前を付けるには、見出しを使用する必要があります。以下の例では、見出し **Student** と **Total** を作成します。

```
CREATE VIEW Accounts (Student, Total)
AS SELECT Student_ID, SUM (Amount_Paid)
FROM Billing
GROUP BY Student_ID#
```

重複する列名を持つ複数のテーブルから **SELECT *** を指定する場合にも、見出しを使用しなければなりません。

- データベースにアクセスするユーザーまたはアプリケーションごとに、カスタマイズされたビューを作成できます。これらのビューの定義は、後で呼び出すためにデータ辞書に格納できます。

ビューの読み取り専用テーブル

読み取り専用のテーブルを含んでいるビューの行を挿入、更新、または削除することはできません。（ここでいう「更新」は、挿入、更新、および削除を指します。テーブルが読み取り専用ならば、テーブルを更新できません。）テーブルの中には、読み取り専用と指定されているビュー内にあるかどうかにかかわらず、読み取り専用のものがあります。そのようなテーブルは本質的に読み取り専用であり、更新できません。テーブルが以下の基準のうちのいずれかを満たした場合、そのテーブルは読み取り専用です。

- データベースはセキュリティが有効で、現在のユーザーまたはユーザー グループはデータベースまたはテーブルに定義されている **SELECT** 権しかありません。
- データ ファイルには、たとえば、DOS または Windows の **ATTRIB** コマンドや Linux および OS X の **chmod** コマンドを使用して、物理ファイル レベルで読み取り専用のフラグが付けられています。
- ビューを作成して以下の項目を組み込む **SELECT** 句を実行します。
 - 選択リストの集計関数
 - **GROUP BY** 句または **HAVING** 句
 - **UNION**
 - **DISTINCT** キーワード
- ビューを作成する **SELECT** ステートメントを実行すると、テーブルには以下の特性が組み込まれます。
 - テーブルは、**SELECT** ステートメントの **FROM** 句にあるマージできないビューに表示されます。
 - テーブルはシステム テーブルです。たとえ読み取り専用モードがビューのオープン モードを無効にする場合でも、システム テーブルは常にビュー内で読み取り専用として開かれます。
 - テーブルからの列が計算列に現れるか、スカラー関数が選択リストに現れます。
 - テーブルは、最も外側のクエリと相関関係を持たないサブクエリの **FROM** 句に現れます。サブクエリは、最も外側のクエリと直接または間接に相関関係を持たせることができます。サブクエリは、もしテーブルから列への参照が含まれており、最も外側の **FROM** 句中に、その特定の発生がある場合、最も外側のクエリと直接相関関係を持ちます。サブクエリは、あるサブクエリと相関関係があつて、そのサブクエリが最も外側のクエリと直接または間接に相関関係がある場合に、最も外側のクエリと間接的に相関関係があります。
 - オープン モードは読み取り専用です。

- FOR UPDATE を指定せずに、以下のキーワードで位置付け UPDATE ステートメントを実行します。
ORDER BY
SCROLL

マージ可能なビュー

ビューは、ベース テーブルと列だけを使用して SELECT ステートメントを書き換えることができる場合にマージ可能です。

たとえば、何人の学生が 1 クラスにいるかを知りたい場合、それを計算するビューを定義できます。以下のようにビュー `NumberPerClass` を定義します。

```
CREATE VIEW NumberPerClass (Class_Name, Number_of_Students)
AS SELECT Name, COUNT (Last_Name)
FROM Person, Class, Enrolls
WHERE Person.ID = Enrolls.Student_ID
AND Class.ID = Enrolls.Class_ID
GROUP BY Name#
```

以下のようにビュー `NumberPerClass` を定義します。

```
SELECT *
FROM NumberPerClass#
```

この場合、ビュー `NumberPerClass` がマージ可能なのは、以下のように SELECT ステートメントを書き換えることができるからです。

```
SELECT Name, COUNT (Last_Name)
FROM Person, Class, Enrolls
WHERE Person.ID = Enrolls.Student_ID
AND Class.ID = Enrolls.Class_ID
GROUP BY NAME#
```

以下のように SELECT ステートメントを書きたい場合、ビュー `NumberPerClass` はマージ不能です。

```
SELECT COUNT (Name)
FROM NumberPerClass
WHERE Number_of_Students > 50#
```

このステートメントは、ビュー `NumberPerClass` に対して無効です。ベース テーブルとベース 列だけでは、このビューを書き換えることができません。

ビューに以下の特性が含まれていない場合、ビューはマージ可能です。

- ビューがマージ不能なビューを参照する。
- ビューが選択リスト内に集計関数を持つか、DISTINCT キーワードを持っており、また、選択リスト内に集計を持つ SELECT ステートメントの FROM 句に現れる。
- ビューが DISTINCT キーワードを持ち、SELECT ステートメントの FROM 句に現れるが、そのステートメントはその FROM 句の中に複数の項目があり、その選択リストに集計を持たず、かつ DISTINCT キーワードを持っていない。
- ビューがその選択リストに集計を持ち、SELECT ステートメントの FROM 句に現れるが、そのステートメントはその FROM 句の中に複数の項目があるか、または WHERE 句の制限がある。

選択リスト

SELECT ステートメントを使用してデータを取得する場合は、結果テーブルに組み込む列のリスト、つまり、選択リストを指定します。テーブル内のすべての列を取得する場合、列のリストの代わりにアスタリスク (*) を使用できます。



メモ リストの代わりに * を使用することは避けてください。* を使用すると、テーブル内の列の数または列のサイズが変化した場合にアプリケーションに潜在的な問題が生ずるおそれがあります。また、アプリケーションは一般に不必要なデータを返します。

以下の例では、Class テーブルから 3 つの列を選択します。

```
SELECT Name, Section, Max_Size
FROM Class;
```

以下の例では、Class テーブルからすべての列を選択します。

```
SELECT * FROM Class;
```

データを取得する場合、PSQL はクエリで名前を指定した方法に基づいて列名を表示します。

- 列名を明示的に指定すると、PSQL は入力されたとおりに列名を返します。以下の例では、列名をすべて小文字で指定します。

```
SELECT name, section, max_size FROM Class#
```

PSQL は、以下のように列名を返します。

```
"Name", "Section", "Max_Size"
```

これらの列名は、返されたデータの見出しです。データ自体ではありません。

以下の例では、テーブル Department と Faculty のエイリアスを定義します。

```
SELECT d.Name, f.ID FROM Department d, Faculty f;
```

PSQL は、以下のように列名を返します。

```
"Name", "ID"
```

- * を使用して列名を指定する場合、以下の例に示すように列名はすべて大文字で表示されます。

```
SELECT * FROM Department;
```

PSQL は、以下のように列名を返します。

```
"Name", "Phone_Number", "Building_Name", "Room_Number", "Head_Of_Dept"
```

以下の例では、テーブル Department と Faculty のエイリアスを定義します。

```
SELECT * FROM Department d, Faculty f;
```

PSQL は、以下のように列名を返します。

```
"Name"
"Phone_Number"
"Building_Name"
"Room_Number"
"Head_Of_Dept"
"ID"
"Dept_Name"
"Designation"
"Salary"
"Building_Name"
"Room_Number"
"Rsched_Grant_Amount"
```

ソートされた行とグループ化された行

結果テーブルにどのようなデータを組み入れるかを決定すると、データの順序を指定できます。ORDER BY 句を使用してデータをソートしたり、GROUP BY 句を使用してある列単位で行をグループ化することができます。データをグループ化すると、集計関数を使用してグループ単位でデータを要約することもできます。集計関数の詳細については、「[集計関数](#)」を参照してください。

以下の例では、サンプル データベースの Person テーブルにラスト ネームですべての行の順序を指定します。

```
SELECT *  
  FROM Person  
 ORDER BY Last_Name#
```

以下の例では、Room テーブル内の Building Name 列で結果をグループ化します。この例では、2 つの集計関数 COUNT と SUM も使用します。

```
SELECT Building_Name, COUNT(Number), SUM(Capacity)  
  FROM Room  
 GROUP BY Building_Name;
```

結合

結合は、列を複数のテーブルから 1 つのビューに結合するステートメントから発生します。データが読み取り専用でなければ、このビューからデータの取得、挿入、更新または削除を行えます。



メモ ここでは、主に、SELECT ステートメントによるテーブルの結合について説明します。ただし、1 つのステートメントを複数のテーブルに適用することによって、INSERT、UPDATE および DELETE ステートメントで結合を作成することもできます。『SQL Engine Reference』では、これら SQL ステートメントおよび結合ビューの最適化の方法について説明しています。

FROM 句に各テーブル名またはビュー名を表示することによって、テーブルからデータを取得できます。1 つまたは複数の結合条件を指定するには、WHERE 句を使用します。結合条件では、1 つのテーブルから列の値を参照する式を、別のテーブルから列の値を参照する式と比較します。

データが正しく正規化されると、ほとんどの結合は指定されたキー値に基づいて値を関連付けます。それにより、参照整合性の関係でデータを抽出できます。たとえば、どの教授がどのクラスを教えているかを知りたい場合、Faculty ID に基づいて結合を作成できます。Faculty ID は、Class テーブル内の外部キーであり、また、Person テーブル内の主キーです。

```
SELECT DISTINCT Class.Name, Person.Last_Name
FROM Class, Person, Faculty
WHERE Class.Faculty_ID = Person.ID
      AND Class.Faculty_ID = Faculty.ID;
```

この例では、共通の列 Faculty ID の共通の値に基づいて 2 つのテーブルを結合します。

また、データ型の列の間で数値の比較を行うことでもテーブルを結合できます。たとえば、<、> または = を使用して列を比較できます。Faculty テーブルの以下の自己結合は、各教職員より給与が高いすべての教職員を識別します。このため、Faculty テーブルに含まれているレコードよりかなり多くのレコードが生成されます。

```
SELECT A.ID, A.Salary, B.ID, B.Salary
FROM Faculty A, Faculty B
WHERE B.Salary > A.Salary;
```

日付や時刻などの同様の比較を行えば、多数の有効かつ意味のある結果を生成できます。

列を結合する場合は、可能であれば同じデータ型の列を選択します。たとえば、2 つの NUMERIC 列を比較する方が、NUMERIC 列を INTEGER 列と比較するより効率的です。2 つの列が同じデータ型でなく、ともに数値または文字列であれば、PSQL は両方のテーブルをスキャンし、結合条件を結果に対する制限として適用します。

WHERE 句で文字列型の列を使用する場合、結合条件の 1 つの列を計算の文字列の列とすることができます。そうすれば、複数の文字列を連結し、結合条件でそれらの文字列を別のテーブルからの 1 つの文字列と比較することができます。

PSQL が結合を処理する方法は、結合条件にインデックス列が含まれているかどうかにより異なります。

- インデックスとして定義されている列が結合条件に含まれている場合、パフォーマンスは向上します。インデックスで対応するテーブル内の行をソートする場合、PSQL は制限句の条件に合う行だけを選択します。
- 結合条件にインデックスとして定義されている列が含まれていない場合、パフォーマンスは低下します。PSQL は各テーブルの各行を読んで制限句の条件に合致する行を選択します。パフォーマンスを向上するために、結合を実行する前にテーブルのうちの 1 つでインデックスを作成することができます。これは特に、クエリが頻繁に実行するクエリである場合に有効です。

ほかのテーブルとのテーブルの結合

SELECT ステートメントを使用して結合を指定するには、FROM 句 を使用して関連するテーブルを一覧表示し、WHERE 句を使用して結合の条件と制限を指定します。以下の例では、エイリアスを使用してステートメントの単純化も行います。

```
SELECT Student_ID, Class_ID, Name
FROM Enrolls e, Class cl
WHERE e.Class_ID = cl.ID;
```

次の例では、3 つのテーブルを結合します。

```
SELECT p.ID, Last_Name, Name
FROM Person p, Enrolls e, Class cl
WHERE p.ID = e.Student_ID AND e.Class_ID = cl.ID;
```

次の例では、英語で 3.0 より低い成績をとった学生の一覧を取得します。

```
SELECT First_Name, p.Last_Name
FROM Person p, Student s, Enrolls e, Class cl
WHERE s.ID = e.Student_ID
AND e.Class_ID = cl.ID
AND s.ID = p.ID
AND cl.Name = 'ACC 101'
AND e.Grade < 3.0;
```

この例では、WHERE 句の最初の 3 つの条件で 4 つのテーブル間の結合を指定します。次の 2 つの条件は、ブール演算子 AND で接続された制限句です。

テーブルとのビューの結合

ビューを 1 つまたは複数のテーブルと結合するには、FROM 句にビュー名を取り込みます。指定するビューには、1 つのテーブルまたはいくつかの結合されたテーブルから列を取り込むことができます。

結合のタイプ

PSQL は、等結合、不等号結合、ヌル結合、カルテシアン結合、自己結合、左外部結合、右外部結合、および全外部結合をサポートします。

結合の構文の詳細については、以下のトピックを参照してください。

- 『SQL Engine Reference』「[SELECT](#)」
- 『SQL Engine Reference』「[JOIN](#)」

等結合

等結合は、2 つの結合列を等価と定義するときに発生します。以下のステートメントは、等結合を定義します。

```
SELECT First_Name, Last_Name, Degree, Residency
FROM Person p, Student s, Tuition t
WHERE p.ID = s.ID AND s.Tuition_ID = t.ID;
```

不等号結合

比較演算に基づいてテーブルを結合できます。不等号結合では以下の演算子を使用できます。

<	より小さい
>	より大きい
<=	小さいかまたは等しい
>=	大きいかまたは等しい

データの取得

以下の WHERE 句は、 \geq 演算子を使用する結合を示しています。

```
SELECT Name, Section, Max_Size, Capacity, r.Building_Name, Number
FROM Class cl, Room r
WHERE Capacity >= Max_Size;
```

カルテシアン結合

カルテシアン結合は、あるテーブル内の各行を別のテーブル内の各行に関連付けます。PSQL は、一方のテーブルの各行に対しもう一方のテーブルのすべての行を読みます。

大きなテーブルでは、カルテシアン結合に時間がかかりかかる可能性があります。PSQL は、このタイプの結合を行うために以下の行数を読み取らなければならないからです。

(あるテーブル内の行数) * (ほかのテーブル内の行数)

たとえば、あるテーブルに 600 行含まれており、もう 1 つのテーブルに 30 行含まれている場合、PSQL は各テーブルのカルテシアン結合を作成するのに 18,000 行を読み取ります。

以下のステートメントは、サンプル データベース内の Person テーブルと Course テーブルにカルテシアン結合を生成します。

```
SELECT s.ID, Major, t.ID, Degree, Residency, Cost_Per_Credit
FROM Student s, Tuition t#
```

自己結合

自己結合では、FROM 句でテーブル名を何回も指定できます。自己結合を指定する場合は、テーブル名の各インスタンスにエイリアスを割り当て、PSQL が結合でテーブルの各発生セグメントを区別できるようにします。

以下の例では、Jason Knibb という人と同じ州に定住所がある人のすべてをリストします。このクエリは、ID、名前、名字、現在の電話番号、電子メールアドレスを返します。

```
SELECT p2.ID, p2.First_Name, p2.Last_Name, p2.Phone, p2.Email_Address
FROM Person p1, Person p2
WHERE p1.First_Name = 'Jason' AND p1.Last_Name = 'Knibb'
AND p1.Perm_State = p2.Perm_State
```

左部、右部、完全外部結合

外部結合の詳細については、『SQL Engine Reference』に記載されています。『SQL Engine Reference』の「[SELECT](#)」および「[JOIN](#)」を参照してください。

サブクエリ

ネストされたクエリとも呼ぶサブクエリは、以下のうちの1つの中に含まれている SELECT ステートメントです。

- 別の SELECT ステートメントの WHERE 句または HAVING 句。
- UPDATE または DELETE ステートメントの WHERE 句。

サブクエリを使用すると、ネストされた SELECT ステートメントの出力に SELECT、UPDATE または DELETE ステートメントの結果の基準を置くことができます。

相関関係を持つサブクエリ以外のサブクエリを発行すると、PSQL はステートメント全体を構文解析し、最も内側のサブクエリを最初に実行します。Pervasive PSQL は、最も内側のサブクエリの結果を次のレベルのサブクエリに対する入力として使用します。以下同様に行います。

サブクエリで使用できる式の詳細については、『SQL Engine Reference』を参照してください。

サブクエリの制限

WHERE 句のサブクエリは、検索基準の一部になります。SELECT、UPDATE および DELETE ステートメントでサブクエリを使用する場合は、以下の制限が適用されます。

- サブクエリを小カッコで囲まなければならない。
- サブクエリに UNION 句を含めることはできない。
- 外側のクエリの WHERE 句で ANY、ALL、EXISTS または NOT EXISTS キーワードを使用しない限り、サブクエリの選択リストには列名の式を1つしか組み込めない。

ステートメントにいくつかのレベルのサブクエリをネストできます。ネストできるサブクエリ数は、PSQL が使用できるメモリ量で決定されます。

相関サブクエリ

相関サブクエリには、外側のクエリの FROM 句のテーブルから列を参照する WHERE または HAVING 句が含まれています。この列を相関列と呼びます。外側のクエリからの結果と比較してサブクエリからの結果をテストするか、クエリの中の特定値の有無をテストするには、相関サブクエリを使用しなければなりません。

相関列は外側のクエリから発するので、その値は外側のクエリがフェッチされるたびに変化します。次に PSQL はこの変化した値に基づいて内側のクエリの式を評価します。

以下の例に、教室で実際に使用される時間より多い履修単位時間を持つコースの名前を示します。

```
SELECT c.Name, c.Credit_Hours
FROM Course c
WHERE c.Name IN
  (SELECT cl.Name
   FROM Class cl
   WHERE c.Name = cl.Name AND c.Credit_Hours >
    (HOUR (Finish_Time - Start_Time) + 1))#
```

上記のステートメントは、パフォーマンスを向上させるために簡単なクエリに書き換えることができます。

```
SELECT c.Name, c.Credit_Hours
FROM Class cl, Course c
WHERE cl.Name = c.Name AND c.Credit_Hours >
  (HOUR (Finish_Time - Start_Time) + 1)#
```

制限句

制限句は、演算子と式からなる ASCII テキスト文字列です。制限句は、ビューの列の値に対する選択基準を指定し、ビューに取り込む行数を制限します。**WHERE** や **HAVING** などの句の構文では、制限句を使用しなければなりません。制限句は、以下の条件を指定できます。

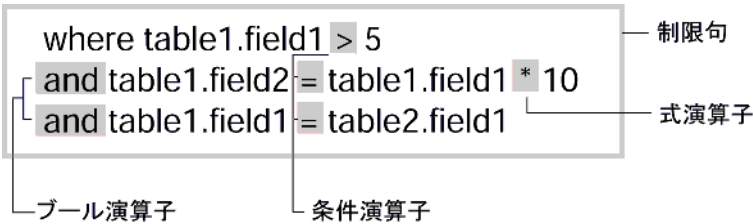
- 制限条件 — 列の値を参照する式を、同じテーブル内の列の値を参照する定数または別の式と比較します。
- 結合条件 — あるテーブルからの列の値を参照する式を、別のテーブルから列の値を参照する式と比較します。

制限句には、複数の条件を含めることができます。制限句には、データベース内のほかのテーブルの内容に検索基準を置いている **SELECT** サブクエリも含めることができます。サブクエリを含む条件には、**EXISTS**、**NOT EXISTS**、**ALL**、**ANY** および **SOME** キーワードまたは **IN** 範囲演算子を含めることができます。

SELECT、**UPDATE** または **DELETE** ステートメント内の **WHERE** または **HAVING** 句を使用して、制限句を指定できます。

図 1 には、制限句の例を示し、制限句の要素を図示しています。

図 1 制限句の例



制限句演算子

制限句は 3 つのタイプの演算子を使用できます。

- ブール演算子 — 制限句の条件を連結します。
- 条件演算子 — 式を連結して条件を形成します。条件演算子は、関係演算子または範囲演算子とすることができます。
- 式演算子 — 2 つの式を連結して別の式を作ります。式の演算子は、算術演算子または文字列演算子のいずれかです。

ブール演算子

ブール演算子は、論理条件を指定します。

表 43 ブール演算子

演算子	説明
AND	AND で連結されているすべての検索条件が True であれば、制限をパスします。
OR	OR で連結された条件の少なくとも 1 つが True であれば、制限をパスします。
NOT	条件が False であれば、制限をパスします。

条件演算子

条件演算子は、関係演算子または範囲演算子とすることができます。

- 関係演算子 — 列の値を別の列の値または定数と比較します。列の値が **True** であれば、PSQL はその行を選択します。
- 範囲演算子 — 列の値をその列の指定された範囲の値と比較します。列の値が **True** であれば、制限を通過し、PSQL はその行を選択します。

表 44 に関係演算子を示します。

表 44 関係条件演算子

演算子	説明	演算子	説明
<	より小さい	>=	大きいまたは等しい
>	より大きい	!=	等しくない
=	等しい	<>	等しくない
<=	小さいまたは等しい		

表 45 に条件演算子を示します。

表 45 範囲条件演算子

演算子	説明
IN	値は指定されたリストに存在します。
NOT IN	値は指定されたリストに存在しません。
BETWEEN	値は指定された範囲内に存在します。
NOT BETWEEN	値は指定された範囲内に存在しません。
IS NULL	値は列に対して定義された NULL 値です。
IS NOT NULL	値は列に対して定義された NULL 値ではありません。
LIKE	値は指定された文字列に一致します。実際の文字に対して 2 つのワイルドカード文字を代入できます。パーセント記号 (%) は、 <i>n</i> 文字（ここで、 <i>n</i> は 0 でもよい）の並びを表します。アンダースコア (_) は、単一文字を表します。
NOT LIKE	値は指定された文字列に一致しません。

IN 演算子と NOT IN 演算子を使用すると、第 2 の式は列名または定数の代わりにサブクエリとすることができます。

式の演算子

式の演算子を使用して、算術演算子または文字列演算子で計算列の式を作成できます。詳細については、「[関数](#)」を参照してください。

制限句の例

以下の例では、制限句演算子のいくつかを実証します。

データの取得

OR と 等号 (=)

以下の例では、関係 EQUAL TO 演算子とブール OR 演算子を使用します。また、State 列の値が Texas または New Mexico であるすべての行を選択します。

```
SELECT Last_Name, First_Name, State
FROM Person
WHERE State = 'TX' OR State = 'NM' #
```

IN

以下の例では、IN 演算子を使用します。これにより、名が Bill と Roosevelt のレコードがテーブル Person から選択されます。

```
SELECT * FROM Person WHERE First_name IN ('Roosevelt', 'Bill') #
```

LIKE

以下の例では、LIKE 演算子を使用します。

```
SELECT ID, First_Name, Last_Name, Zip
FROM Person
WHERE Zip LIKE '787%';
```

この例では、Person テーブルから ZIP コードが "787" で始まるレコードを取得します。

関数

一度データベースにデータを取り込むと、データに対して関数（集計関数）を使用して、列値のセットに結果を返すことができます。または、1 つまたは複数のパラメーターを入力として受け入れ、スカラー関数を使用して、1 つの値を返すことができます。

集計関数

集計関数は、一連の列の値に対する 1 つの結果を返す関数です。PSQL では、表 46 に示す集計関数を使用できます。

表 46 集計関数

関数	説明
AVG	値のグループの平均を算定します。オペランドが DECIMAL でない場合、AVG は 8 バイトの FLOAT を返します。オペランドが DECIMAL である場合、AVG は 10 バイトの DECIMAL を返します。
COUNT	指定されたグループ内の行数をカウントします。COUNT は常に、4 バイトの INTEGER を返します。
DISTINCT	DISTINCT キーワードは SELECT ステートメントで使用して、結果から重複する値を削除するよう PSQL に指示します。DISTINCT を使用すると、SELECT ステートメントの条件を満たす一意の行をすべて検索できます。
MAX	値のグループの最大値を返します。MAX は、オペランドと同じデータ型とサイズを返します。
MIN	値のグループの最小値を返します。MIN は、オペランドと同じデータ型とサイズを返します。
SUM	値のグループの合計を算定します。オペランドが DECIMAL でない場合、SUM は 8 バイトの FLOAT を返します。オペランドが DECIMAL である場合、SUM は 10 バイトの DECIMAL を返します。

これらの各関数の詳細については、『SQL Engine Reference』を参照してください。

集合関数への引数

AVG 関数と SUM 関数の場合、関数への引数は数値列名でなければなりません。COUNT 関数、MIN 関数および MAX 関数は、数値列または非数値列に結果を示すことができます。

集計関数の参照をネストすることはできません。たとえば、以下の参照は無効です。

```
SUM(AVG(Cost_Per_Credit))
```

以下の例に示すように、式で集計関数を使用できます。

```
AVG(Cost_Per_Credit) + 20
```

グループ集計関数への引数として式を使用することもできます。たとえば、以下の式は有効です。

```
AVG(Cost_Per_Credit + 20)
```

集計関数は、ヌル列の値を有効値として扱います。たとえば、40 行のデータと 5 行のヌル値を含むテーブルで、COUNT 関数は 45 を返します。

DISTINCT キーワードを使用して、PSQL がすべてのヌル列の値を 1 つの値として扱うようにすることができます。以下の例では、Grade 列で列の平均値を計算します。

```
AVG(DISTINCT Grade)
```

DISTINCT キーワードは、AVG 関数、COUNT 関数および SUM 関数に影響を与えます。このキーワードは、MIN 関数と MAX 関数に影響を与えません。

集計関数の規則

以下のように、SELECT ステートメントで集計関数を使用できます。

- 選択リストの項目
- HAVING 句

一般に、GROUP BY 句を含む SELECT ステートメントで集計関数を使用して、ある行のグループの集計値を算定します。ただし、SELECT ステートメントに GROUP BY 句が含まれておらず、その句で集計関数を使用したい場合、選択リストのすべての項目は集計関数でなければなりません。

SELECT ステートメントに GROUP BY 句が含まれている場合、GROUP BY 句に指定する列は、集計関数でなく単一の列である選択項目でなければなりません。ただし、GROUP BY 句にも示されていない選択項目はすべて集計関数でなければなりません。

以下の例では、各学生が支払った金額を決定できる結果テーブルを返します。

```
SELECT Student_ID, SUM(Amount_Paid)
FROM Billing
GROUP BY Student_ID;
```

GROUP BY 句で使用される HAVING 句に集計関数を組み込むこともできます。GROUP BY 句を持つ HAVING 句を使用すると、PSQL から返される行のグループが制限されます。PSQL は GROUP BY 句で指定された各行グループの列に対して集計関数を実行し、グループ化列の値が等しい行セットごとに 1 つの結果を返します。

以下の例では、PSQL は 15 時間を超える履修単位時間で現在受講登録されている学生についてのみ行グループを返します。

```
SELECT Student_ID, SUM(Credit_Hours)
FROM Enrolls e, Class cl, Course c
WHERE e.Class_ID = cl.ID AND cl.Name = c.Name
GROUP BY Student_ID
HAVING SUM(Credit_Hours) > 15;
```

スカラー関数

CONCAT や CURDATE のようなスカラー関数は、1 つまたは複数のパラメーターを入力として受け入れ、1 つの値を返します。たとえば、LENGTH 関数は文字列の列の値の長さを返します。式で計算列を使用できる PSQL ステートメントでスカラー関数を使用できます。

使用できる式演算子のタイプは、関数が返す結果のタイプにより異なります。たとえば、関数が数値を返す場合、算術演算子を使用できます。関数が文字列を返す場合、文字列演算子を使用できます。

スカラー関数をネストできますが、以下の例に示すように、ネストされた各関数は次のレベルのスカラー関数へ対応するパラメーターとしての結果を返します。

```
SELECT RIGHT (LEFT (Last_Name, 3), 1)
FROM Person;
```

PSQL はまず、LEFT 関数を実行します。Last Name 列の値が Baldwin である場合、LEFT 関数から生ずる文字列は Bal です。この文字列は RIGHT 関数のパラメーターで、この関数は文字列の右端の文字として「l」を返します。

数値を計算する計算列の中に数値結果を返すスカラー関数を使用できます。文字列値を式として別の文字列関数へ返すスカラー関数も使用できますが、文字列の結果の合計長は 255 バイト以内でなければなりません。

PSQL で使用できるスカラー関数の詳細については、『SQL Engine Reference』で「[ビット演算子](#)」を参照してください。

この章では、将来使用するために SQL プロシージャを格納する方法とトリガーを作成する方法について説明します。ストアド ビューの詳細については、第 16 章「[データの取得](#)」を参照してください。

以下の項目について説明します。

- 「[ストアド プロシージャ](#)」
- 「[SQL 変数ステートメント](#)」
- 「[SQL 制御ステートメント](#)」
- 「[SQL トリガー](#)」

ストアド プロシージャ

ストアド プロシージャを使用して、論理的に関連付けされたプログラミング ステップを一般的なプロセスにグループ化し、次にそのプロセスを1つのステートメントで呼び出すことができます。また、パラメーターを渡すことによって、異なる値でこのプロセスを実行できます。

SQL ストアド プロシージャを呼び出すと、ホスト言語のプログラムと SQL エンジンとの間で内部的な通信を行うことなく、プロシージャ全体が実行されます。ストアド プロシージャを単独で呼び出したり、ほかのプロシージャまたはトリガーの本体の一部として呼び出すことができます。トリガーの詳細については、「[SQL トリガー](#)」を参照してください。

ストアド プロシージャ内の SQL 変数ステートメントを使用して、ステートメントからステートメントへ値を内部に格納することができます。これらのステートメントの詳細については、「[SQL 変数ステートメント](#)」を参照してください。

ストアド プロシージャ内の SQL 制御ステートメントを使用して、プロシージャの実行フローを制御することができます。これらのステートメントの詳細については、この章の後半の「[SQL 制御ステートメント](#)」を参照してください。

ストアド プロシージャと位置付け更新

ストアド プロシージャと位置付け更新の例に示します。

```
DROP PROCEDURE curs1
CREATE PROCEDURE curs1 (in :Arg1 char(4) ) AS
BEGIN
    DECLARE :alpha char(10) DEFAULT 'BA';
    DECLARE :beta INTEGER DEFAULT 100;

    DECLARE degdel CURSOR FOR
        SELECT degree, cost_per_credit FROM tuition
            WHERE Degree = :Arg1 AND cost_per_credit = 100
    FOR UPDATE;
    OPEN degdel;
    FETCH NEXT FROM degdel INTO :alpha,:beta
    DELETE WHERE CURRENT OF degdel;
    CLOSE degdel;
END

CALL curs1('BA')
```

ストアド プロシージャの宣言

ストアド プロシージャを定義するには、CREATE PROCEDURE ステートメントを使用します。

```
CREATE PROCEDURE EnrollStudent (in :Stud_id integer, in :Class_Id integer);

BEGIN
    INSERT INTO Enrolls VALUES (:Stud_id, :Class_Id, 0.0);
END
```

ストアド プロシージャ名の最大サイズは 30 文字です。パラメーター リストの両側には小かっこを付ける必要があります。パラメーター名は有効な SQL 識別子とすることができます。

ストアドプロシージャは、辞書内で一意名を持っていなければなりません。

CREATE PROCEDURE ステートメントの構文の詳細については、『SQL Engine Reference』で「[CREATE PROCEDURE](#)」を参照してください。

ストアド プロシージャの呼び出し

ストアド プロシージャを定義するには、CREATE ステートメントを使用します。

```
CALL EnrollStudent (274410958, 50);
```

すべてのパラメーターに値を定義する必要があります。CALL ステートメントで関連する引数を使用するか、または CREATE PROCEDURE ステートメントの関連するデフォルトの句を使用して、パラメーターに値を指定することができます。CALL ステートメント内のパラメーターの引数値は、関連するデフォルト値に優先します。

以下の2つの方法のいずれかで、CALL ステートメントに呼び出し値を指定できます。

- 場所引数 — プロシージャが作成されたときのリスト内のパラメーターの序数の位置に基づいて、パラメーター値を暗黙に指定することができます。
- キーワード引数 — 値が割り当てられているパラメーターの名前を使用して、パラメーター値を明示的に指定できます。

場所またはキーワードの引数リストでは、パラメーター値を2回指定できません。同一呼び出しで場所引数とキーワード引数の両方を使用する場合、キーワード引数は場所引数を介して値を受け取るパラメーターを参照することはできません。キーワード引数を使用する場合、同じパラメーター名を2度使用することはできません。

CALL ステートメントの構文の詳細については、『SQL Engine Reference』で「[CALL](#)」を参照してください。

ストアド プロシージャの削除

ストアド プロシージャを削除するには、DROP PROCEDURE ステートメントを使用します。

```
DROP PROCEDURE EnrollStudent;
```

このステートメントの構文の詳細については、『SQL Engine Reference』で「[DROP PROCEDURE](#)」を参照してください。

SQL 変数ステートメント

SQL 変数ステートメントを使用して、ステートメント間で内部的に値を格納することができます。SQL 変数ステートメントには、以下のステートメントがあります。

■ Assignment ステートメント

これらのステートメントはストアド プロシージャ内で使用できます。

プロシージャ所有の変数

ストアド プロシージャ内で定義する SQL 変数は、**プロシージャ所有の変数**です。その適用範囲は、SQL 変数が定義されるプロシージャです。したがって、そのプロシージャ内でしかその変数を参照できません。プロシージャが別のプロシージャを呼び出す場合、呼び出し側プロシージャのプロシージャ所有の変数は被呼び出し側プロシージャで直接使用できません。その代わり、その変数をパラメーターで渡す必要があります。同じストアド プロシージャでプロシージャ所有の変数を 2 回以上宣言することはできません。

ストアド プロシージャの本体が複合ステートメントの場合、そのプロシージャで宣言された SQL 変数名は、そのプロシージャのパラメーター リスト内のパラメーター名と同じにすることはできません。複合ステートメントの詳細については、「[複合ステートメント](#)」を参照してください。

代入ステートメント

代入ステートメントは、SQL 変数の値を初期化または変更します。値の式は、定数、演算子、この SQL 変数またはほかの SQL 変数に関連する計算された式とすることができます。

```
SET :CourseName = 'HIS305';
```

値の式は、SELECT ステートメントとすることもできます。

```
SET :MaxEnrollment = (SELECT Max_Size FROM Class  
    WHERE ID = classId);
```

このステートメントの構文の詳細については、『SQL Engine Reference』で「[SET](#)」を参照してください。

SQL 制御ステートメント

ストアド プロシージャの本体内容でのみ制御ステートメントを使用できます。これらのステートメントは、プロシージャの実行を制御します。制御ステートメントは以下のとおりです。

- 複合ステートメント (BEGIN...END)
- IF ステートメント (IF...THEN...ELSE)
- LEAVE ステートメント
- Loop ステートメント (LOOP および WHILE)

複合ステートメント

複合ステートメントは、ほかのステートメントをグループ化します。

```
BEGIN
    DECLARE :NumEnrolled INTEGER;
    DECLARE :MaxEnrollment INTEGER;

    DECLARE :failEnrollment CONDITION
        FOR SQLSTATE '09000';

    SET :NumEnrolled = (SELECT COUNT (*)
        FROM Enrolls
        WHERE Class_ID = classId);

    SET :MaxEnrollment = (SELECT Max_Size
        FROM Class
        WHERE ID = classId);

    IF (:NumEnrolled >= :MaxEnrollment) THEN
        SIGNAL :failEnrollment ELSE
        SET :NumEnrolled = :NumEnrolled + 1;
    END IF;
END
```

ストアド プロシージャまたはトリガーの本体内容で複合ステートメントを使用できます。トリガーの詳細については、「[SQL トリガー](#)」を参照してください。

ほかの複合ステートメント内に複合ステートメントをネストできますが、最も外側の複合ステートメントだけに DECLARE ステートメントを取り込むことができます。

複合ステートメントの構文の詳細については、『SQL Engine Reference』で「[BEGIN \[ATOMIC\]](#)」を参照してください。

IF ステートメント

IF ステートメントは、条件の真の値に基づいて条件付き実行を行います。

```
IF (:counter = :NumRooms) THEN
    LEAVE Fetch_Loop;
END IF;
```

IF ステートメントの構文の詳細については、『SQL Engine Reference』で「[IF](#)」を参照してください。

LEAVE ステートメント

LEAVE ステートメントは、複合ステートメントまたは Loop ステートメントから離れることによって実行を続けます。

```
LEAVE Fetch_Loop
```

LEAVE ステートメントは、ラベル付き複合ステートメント内またはラベル付き Loop ステートメント内に現れるはずですが、LEAVE ステートメントからのステートメント ラベルは、LEAVE を含むラベル付きステートメントのラベルと同じでなければなりません。このラベルは、**対応ラベル**と呼ばれています。



メモ 複合ステートメントには、Loop ステートメントを取り込むことができます。Loop ステートメントを埋め込むことができるので、LEAVE ステートメントのステートメント ラベルは埋め込みループのラベルまたはストアード プロシージャの本体のラベルに一致します。

LEAVE ステートメントの構文の詳細については、『SQL Engine Reference』で「[LEAVE](#)」を参照してください。

LOOP ステートメント

LOOP ステートメントは、ステートメント ブロックの実行を繰り返します。

```
FETCH_LOOP:
LOOP
    FETCH NEXT cRooms INTO CurrentCapacity;

    IF (:counter = :NumRooms) THEN
        LEAVE FETCH_LOOP;
    END IF;

    SET :counter = :counter + 1;
    SET :TotalCapacity = :TotalCapacity + :CurrentCapacity;
END LOOP;
```

SQL ステートメント リスト内の各ステートメントがエラーなく実行され、また、PSQL に LEAVE ステートメントが発生しないか、ハンドラーを呼び出す場合は、LOOP ステートメントの実行が繰り返されます。LOOP ステートメントは、与えられた条件が真である間に実行が継続されるという点で、WHILE ステートメントに似ています。

LOOP ステートメントに開始ラベルがある場合、このステートメントは**ラベル付き LOOP ステートメント**と呼ばれます。終了ラベルを指定する場合、そのラベルは開始ラベルと同じでなければなりません。

LOOP ステートメントの構文の詳細については、『SQL Engine Reference』で「[LOOP](#)」を参照してください。

WHILE ステートメント

WHILE ステートメントは、指定された条件が真である間にステートメント ブロックの実行を繰り返します。

```
FETCH_LOOP:
WHILE (:counter < :NumRooms) DO
    FETCH NEXT cRooms INTO :CurrentCapacity;
    IF (SQLSTATE = '02000') THEN
        LEAVE FETCH_LOOP;
    END IF;

    SET :counter = :counter + 1;
    SET :TotalCapacity = :TotalCapacity + :CurrentCapacity;
END WHILE;
```

PSQL は、ブール値の式を評価します。その値が真であれば、PSQL は SQL ステートメント リストを実行します。SQL ステートメント リスト内の各ステートメントがエラーなく実行され、また、LEAVE ステートメントが発生しない場合は、Loop ステートメントの実行が繰り返されます。ブール値の式が偽または不明である場合、PSQL は Loop ステートメントの実行を終了します。

WHILE ステートメントに開始ラベルがある場合、このステートメントは**ラベル付き WHILE ステートメント**と呼ばれます。終了ラベルを指定する場合、そのラベルは開始ラベルと同じでなければなりません。WHILE ステートメントの構文の詳細については、『SQL Engine Reference』で「[WHILE](#)」を参照してください。

SQL トリガー

トリガーは、データベースに対して一貫性のある規則を強制するために、テーブルに定義されたアクションのことです。トリガーは、ユーザーがそのテーブルで SQL データ変更ステートメントを実行するときに、DBMS に対して実行するアクションが適切かどうかを確認する辞書オブジェクトです。

トリガーを宣言するには、**CREATE TRIGGER** ステートメントを使用します。

```
CREATE TRIGGER CheckCourseLimit;
```

トリガー名の最大サイズは 30 文字です。

トリガーを削除するには、**DROP TRIGGER** ステートメントを使用します。

```
DROP TRIGGER CheckCourseLimit;
```

トリガーを直接呼び出すことはできません。トリガーは、関連するトリガーを持つテーブル上の INSERT、UPDATE または DELETE アクションの結果として呼び出されます。これらのステートメントの構文の詳細については、『SQL Engine Reference』で以下のトピックを参照してください。

- 「[CREATE TRIGGER](#)」
- 「[DROP TRIGGER](#)」
- 「[INSERT](#)」
- 「[UPDATE](#)」
- 「[DELETE](#)」



メモ トリガーが回避されるのを防止するため、PSQL はトリガーを含むデータ ファイルをバウンド データ ファイルとして区別します。これにより、Btrieve ユーザーのアクセスが制限され、PSQL データベースでトリガーを発生させるアクションが実行されないようにします。詳細については、『SQL Engine Reference』を参照してください。

トリガーのタイミングと順序

トリガーは所定のイベントに対して自動的に実行するので、いつどのような順序でトリガーを実行するかを指定できることが大切です。トリガーを作成するときはその時機と順序を指定します。

トリガー アクションの時機の指定

トリガーに関連するイベントが発生すると、そのトリガーはイベントの前または後にトリガーを実行しなければなりません。たとえば、INSERT ステートメントがトリガーを呼び出した場合、トリガーは INSERT ステートメントの実行前または後に実行しなければなりません。

```
CREATE TABLE Tuitionidtable (primary key(id), id ubigint)#
CREATE TRIGGER InsTrig
  BEFORE INSERT ON Tuition
  REFERENCING NEW AS Indata
  FOR EACH ROW
  INSERT INTO Tuitionidtable VALUES (Indata.ID);
```

トリガー アクションの時機として **BEFORE** または **AFTER** を指定してください。トリガー アクションは、行ごとに 1 回実行します。**BEFORE** を指定すると、トリガーは行オペレーションの前に実行します。**AFTER** を指定すると、トリガーは行オペレーションの後に実行します。



メモ PSQL は、RI の制約を設定することによってはトリガーを呼び出しません。また、RI の制約によってシステムがテーブル上でカスケードされた削除を行う可能性もある場合、テーブルでは DELETE トリガーが定義されないことがあります。

トリガー順序の指定

イベントが同じ指定時機に複数のトリガーを呼び出す場合があります。たとえば、INSERT ステートメントは、その実行後に実行するよう定義された複数のトリガーを呼び出すことができます。これらのトリガーは同時に実行できないので、トリガーの実行順序を指定する必要があります。

以下の CREATE TRIGGER ステートメントが 1 番を指定しているので、テーブルに対して定義された以降の BEFORE INSERT トリガーはすべて 1 より大きい一意の番号を得なければなりません。

```
CREATE TRIGGER CheckCourseLimit
    BEFORE INSERT
    ON Enrolls
    ORDER 1
```

符号なし整数で順序の値を指定しますが、この整数はそのテーブル、時間およびイベントに対して一意でなければなりません。現在の順序に新しいトリガーを追加する可能性がある場合は、これに適応できるように、番号に空を残しておくようにします。

トリガーの順序を指定しないと、トリガーはそのテーブル、時間およびイベントに対して現在定義されているトリガーの順序の値よりも大きい一意の順序の値で作成されます。

トリガー アクションの定義

トリガーアクションは、行ごとに 1 回実行します。トリガー アクションの構文は以下のとおりです。

```
CREATE TRIGGER InsTrig
    BEFORE INSERT ON Tuition
    REFERENCING NEW AS Indata
    FOR EACH ROW
    INSERT INTO Tuitionidtable VALUES (Indata.ID);
```

トリガー アクションに WHEN 句が含まれている場合、ブール式が真であれば、トリガーされた SQL ステートメントが実行します。式が真でなければ、トリガーされた SQL ステートメントは実行しません。WHEN 句が存在しないと、トリガーされた SQL ステートメントは無条件で実行します。

トリガーされた SQL ステートメントは、ストアド プロシージャ呼び出し (CALL *procedure_name*) などの単一の SQL ステートメントか、複合ステートメント (BEGIN...END) とすることができます。



メモ トリガーのアクションはトリガーのタイトル テーブルを変更しないようにする必要があります。

トリガー アクションで、古い行イメージの列 (DELETE または UPDATE の場合) または新規の行イメージの列 (INSERT または UPDATE の場合) を参照しなければならない場合は、以下のようにトリガー宣言に REFERENCING 句を挿入する必要があります。

```
REFERENCING NEW AS N
```

REFERENCING 句を使用して、トリガーによって変更されるデータの情報を保持できます。

この章では、以下の項目について説明します。

- 「[データ管理の概要](#)」
- 「[テーブル間の関係の定義](#)」
- 「[キー](#)」
- 「[参照制約](#)」
- 「[サンプル データベースの参照整合性](#)」
- 「[データベース セキュリティの管理](#)」
- 「[並行制御](#)」
- 「[PSQL データベースのアトミシティ](#)」

データ管理の概要

この章では、以下の項目について説明します。

- テーブル間の関係の定義
- データベース セキュリティの管理
- 並行性の制御
- SQL データベースのアトミシティ

多くの場合、SQL ステートメントを使用して、これらのデータベース管理作業を行うことができます。

SQL Data Manager を使用して SQL ステートメントを入力することもできます。SQL Data Manager の使用方法の詳細については、『PSQL User's Guide』を参照してください。

テーブル間の関係の定義

PSQL と共に参照整合性 (RI) を使用して、データベース内でそれぞれのテーブルがどのように関係しているかを定義することができます。RI は、あるテーブルの列 (または列のグループ) が別のテーブルの列 (または列のグループ) を参照しているとき、これらの列に対する変更は同期することを保証します。RI はテーブル間の関係を定義する一連の規則を提供します。これらの規則は**参照制約**として知られています (参照制約は簡略的に関係とも呼ばれます)。

データベース内のテーブルに参照制約を定義すると、MicroKernel エンジンが、これらのテーブルにアクセスするすべてのアプリケーションにわたって制約を強制します。これにより、アプリケーションはテーブルを変更するごとに個別にテーブルの参照をチェックすることから解放されます。

RI を使用するにはデータベースに名前を付ける必要があります。いったん参照制約を定義すると、影響を受けるファイルはそれぞれデータベース名を含みます。誰かがファイルを更新しようとするとき、MicroKernel エンジンがデータベース名を使用して適用できる RI 定義を含むデータ辞書を探し、その RI 制約に対して更新をチェックします。これにより PSQL アプリケーションが RI を危うくすることを防止します。MicroKernel エンジンが参照整合性制約に合致しない更新を阻止するからです。

データベース内のテーブルに参照整合性を定義するには、CREATE TABLE および ALTER TABLE ステートメントを使用します。これらのステートメントの構文説明については、『SQL Engine Reference』の以下のトピックを参照してください。

- 「[CREATE TABLE](#)」
- 「[ALTER TABLE](#)」

参照整合性の定義

次の定義は、参照整合性を理解するのに役立ちます。

- 親テーブルは、外部キーによって参照される主キーを含むテーブルです。
- 親行は、主キーが外部キー値と一致する、親テーブル内にある行です。
- あるテーブルの行の削除により、別のテーブルの行の削除が起こる場合、**連鎖削除テーブル**が発生します。テーブルが連鎖削除かどうかは次の条件により決定されます。
 - 自己参照テーブルはそれ自体に対し連鎖削除になります。
 - 従属テーブルは、削除規則に関係なく、常にその親に対して連鎖削除です。
 - あるテーブルの親テーブルとさらにその親テーブルの削除規則が CASCADE の場合、このテーブルと、親の親テーブルは連鎖削除になります。
- 従属テーブルは、1 つまたは複数の外部キーを持つテーブルです。これらの外部キーは、それぞれ同一または異なるテーブルの主キーを参照することができます。従属テーブルは複数の外部キーを持つことができます。

従属テーブルの外部キー値は、それぞれ関連する親テーブルに一致する主キー値がある必要があります。言い換えると、外部キーが特定の値を持つ場合、外部キーの親テーブル内のいずれかの行の主キー値がその値を持つ必要があります。

従属テーブルに行を挿入する試みは、次の場合に失敗します。それぞれの参照制約の親テーブルが、挿入しようとする従属テーブルの外部キー値と一致する主キー値を持たない場合です。外部キーが現在参照している親テーブルの行を削除しようとするとき、参照制約をどのように定義したかによって、失敗するか、または従属行まで削除することになります。

- 従属行は、従属テーブル内の行で、その外部キー値は、関連付けられている親行の一致する主キー値に依存します。
- 孤立行は、親テーブルの主キーに対応するインデックスに存在しない外部キー値を持つ、従属テーブルの行です。従属キー値は対応する親キー値を持ちません。
- 参照は、主キーを参照する外部キーです。

データの管理

- 参照パスは、従属テーブルと親テーブルの間の参照で構成される特定のセットです。
- 子孫は、参照パス上にある従属テーブルです。これは、パスの元の親テーブルから削除された 1 つまたは複数の参照です。
- 自己参照テーブルは、それ自体の親テーブルであり、その主キーを参照する外部キーを含むテーブルです。
- サイクルは、あるテーブルの親テーブルが同時にそのテーブルの子テーブルにもなっている参照パスです。

キー

RI を使用するには、キーを定義する必要があります。キーには、主キーと外部キーの 2 つのタイプがあります。

主キーとは、テーブル内の各行を一意に識別する列または列のグループのことです。キー値は常に一意であるため、行の重複の検出または防止に使用することができます。

外部キーは、テーブルの関係における従属テーブルと親テーブルで共通の列または列のセットです。親テーブルは、主キーとして定義された一致する列または列のセットを持つ必要があります。外部キーは親テーブルの主キーを参照します。これは、1 つのテーブルから別のテーブルへの列の関係で、MicroKernel エンジンに参照制約を行わせる機能を提供します。

主キー

優れた主キーは次のような特性をもちます。

- これは必須で、非ヌル値を格納する必要があります。
- 一意であること。たとえば、**Student** または **Faculty** テーブルの **ID** 列は、それぞれ一意に定義されているため、優れたキーと言えます。人の名前を使用することは、複数の人が同一名である可能性があるため、あまり実用的ではありません。また、データベースは名前のバリエーション（たとえば、**Andrew** に対する **Andy** や **Jennifer** に対する **Jen**）を同一のものとして検出することができません。
- 安定性があること。学生の **ID** は、個人を一意に識別するだけでなく、人の名前が変更される可能性があるのに対し変更されることがないので、優れたキーです。
- 短いこと。文字数が少ないこと。小さな列はストレージのわずかなスペースしか占めず、データベースの検索が早く、入力ミスが少なくなります。たとえば、9 桁の **ID** 列は 30 文字の名前列より簡単にアクセスできます。

主キーの作成

テーブルに外部キーを作成することにより、参照制約を作成します。ただし、外部キーを作成する前に、外部キーが参照する親テーブルの主キーを作成する必要があります。

テーブルは主キーを 1 つだけ持つことができます。次のいずれかを使用して主キーを作成することができます。

- **CREATE TABLE** ステートメントの **PRIMARY KEY** 句
- **ALTER TABLE** ステートメントの **ADD PRIMARY KEY** 句

次の例は、サンプル データベースの **Person** テーブルに、主キー **ID** を作成します。

```
ALTER TABLE Person
  ADD PRIMARY KEY (ID);
```

主キーを作成する場合、**PSQL** は、一意、非ヌル、変更不可能なインデックスを使用して主キーをテーブル上に実装することを忘れないでください。指定した列にそのようなインデックスが存在しない場合、**PSQL** は、これらの特性を持ち、主キー定義に指定された列を含む、名前のないインデックスを追加します。

主キーの削除

主キーを削除できるのは、それに依存する外部キーをすべて削除した後だけです。テーブルから主キーを削除するには、**ALTER TABLE** ステートメントで **DROP PRIMARY KEY** 句を使用します。テーブルには主キーが 1 つしかないため、次の例に示すように、主キーを削除するときに列名を指定する必要はありません。

```
ALTER TABLE Person
  DROP PRIMARY KEY;
```

主キーの変更

テーブルの主キーを変更するには、次の手順を行います。

- 1 ALTER TABLE ステートメントで DROP PRIMARY KEY 句を使用して既存の主キーを削除します。



メモ このことにより、主キーに使用された列やインデックスが削除されることはありません。主キー定義を削除するだけです。主キーを削除するには、その主キーを参照する外部キーがあってははいけません。

- 2 ALTER TABLE ステートメントで ADD PRIMARY KEY 句を使用して新しい主キーを作成します。

外部キー

外部キーは、テーブルの関係における従属テーブルと親テーブルで共通の列または列のセットです。親テーブルは、主キーとして定義された一致する列または列のセットを持つ必要があります。外部キーを作成すると、従属テーブルとその親テーブルの間に参照制約またはデータ リンクを作成することになります。この参照制約には親テーブルの従属行を削除または更新する規則を含めることができます。

外部キー名は省略可能です。外部キー名を指定しない場合、PSQL は外部キー定義の最初の列の名前を使用して外部キーを作成しようとします。外部キーとその他のデータベース要素の名前付け規則については、「[名前付け規則](#)」を参照してください。

PSQL のキーワードは予約語であるため、これらはデータベース要素の名前付けには使用できません。PSQL のキーワードの一覧は、『SQL Engine Reference』の「[SQL の予約語](#)」を参照してください。

既存のテーブルに外部キーを作成する

既存のテーブルに外部キーを作成するには、次の手順を行います。

- 1 参照する親テーブルに主キーが存在することを確認します。
主キーと外部キーのすべての列は同一のデータ型と長さで、一連の列の順序は両方の定義で同じである必要があります。
- 2 PSQL は、外部キー定義に指定された列または列のグループに非ヌル インデックスを作成します。テーブル定義に既にそのようなインデックスが存在する場合、PSQL はそのインデックスを使用します。存在しなければ、PSQL は非ヌルで、一意ではなく、変更可能なインデックス属性を持つ名前のないインデックスを作成します。
- 3 ALTER TABLE ステートメントで ADD CONSTRAINT 句を使用して新しい外部キーを作成します。

たとえば、次のステートメントは、サンプル データベースの Faculty テーブルの Dept_Name 列に Faculty_Dept という名前の外部キーを作成します。外部キーは Department テーブルに作成された主キーを参照し、削除制限規則を指定します。

```
ALTER TABLE Faculty
    ADD CONSTRAINT Faculty_Dept FOREIGN KEY (Dept_Name)
    REFERENCES Department
    ON DELETE RESTRICT;
```

テーブル作成時に外部キーを作成する

テーブル作成時に外部キーを作成するには、次の手順を行います。

- 1 参照する親テーブルに主キーが存在することを確認します。
主キーと外部キーのすべての列は同一のデータ型と長さで、一連の列の順序は両方の定義で同じである必要があります。

- 2 PSQL は、外部キー定義に指定された列または列のグループに非ヌル インデックスを作成します。テーブル定義に既にそのようなインデックスが存在する場合、PSQL はそのインデックスを使用します。存在しなければ、PSQL は非ヌルで、一意ではなく、変更可能なインデックス属性を持つ名前のないインデックスを作成します。
- 3 CREATE TABLE ステートメントを使用してテーブルを作成し、FOREIGN KEY 句を含めます。

たとえば、次のステートメントは、Course テーブルの Dept_Name 列に Course_in_Dept という外部キーを作成します。

```
CREATE TABLE Course
  (Name CHAR(7) CASE,
   Description CHAR(50) CASE,
   Credit_Hours USMALLINT,
   Dept_Name CHAR(20) CASE) #

ALTER TABLE Course
  ADD CONSTRAINT Course_in_Dept
  FOREIGN KEY (Dept_Name)
  REFERENCES DEPARTMENT (Name)
  ON DELETE RESTRICT
```

外部キーの削除

テーブルから外部キーを削除するには、ALTER TABLE ステートメントで DROP CONSTRAINT 句を使用します。テーブルには複数の外部キーがある可能性があるため、外部キーの名前を指定する必要があります。

```
ALTER TABLE Course
  DROP CONSTRAINT Course_in_Dept;
```

参照制約

参照制約を定義するデータベースは次の条件を満たしている必要があります。

- データベースにはデータベース名がある必要があります。
- データベースは、単一のワークステーション ドライブまたは単一のマップされたネットワーク ドライブに存在する必要があります。
- データ ファイルは 6.x 以降の MicroKernel エンジン形式である必要があります。

5.x 以降のデータ ファイルの 6.x または 7.x 形式への変換については、『Advanced Operations Guide』を参照してください。

データベースが参照整合性をサポートするためには、外部キーの概念をサポートする必要があります。外部キーは 1 つのテーブル（従属テーブルと呼ばれる）の 1 つの列または一連の列で、別のテーブル（親テーブルと呼ばれる）の主キーを参照するのに使用します。RI 規則はすべての外部キー値が有効な主キー値を参照することを必要とします。たとえば、学生は存在しない講座に登録することはできません。

CREATE TABLE または ALTER TABLE ステートメントを使用して、名前付きデータベースのテーブルにキーを定義することができます。次のセクションでは、キーの作成と変更方法について説明します。また、参照制約の例も用意されています。

データベースに参照制約を定義した後は、更新を行うアプリケーションは参照規則に従わないと失敗します。たとえば、アプリケーションが対応する親行を親テーブルに挿入する前に従属テーブルに行を挿入しようとすると、これは失敗します。詳細については、「[参照整合性規則](#)」を参照してください。



メモ ファイルに参照制約が定義されている場合、これはバウンド データ ファイルです。ユーザーがこのファイルに Btrieve を使用してアクセスしようとすると、アクセスできますが、RI 制約の範囲内のアクションを実行するのに限られます。バウンド データ ファイルについての詳細は、「[データベース権限の理解](#)」を参照してください。

参照整合性規則

データベース テーブルに参照制約を定義した場合、従属テーブルの行の挿入と更新、および親テーブルの行の更新と削除に一定の規則が適用されます。PSQL は、次のように制限規則とカスケード規則をサポートします。

- 従属テーブルへの挿入 — 各外部キー定義の親テーブルは、挿入する外部キーに対応する主キー値を持つ必要があります。親テーブルが対応する値を持たない場合、その挿入処理は失敗します。
- 従属テーブルの更新 — 各外部キー定義の親テーブルは、外部キーに対応する主キー値（外部キーの新しい値）を持つ必要があります。親テーブルが対応する値を持たない場合、その更新処理は失敗します。
- 親テーブルでの更新 — これは許可されません。主キー値を更新することはできません。このような処理を実行するには、更新したい行を削除し、その後新しいキー値を持つ同一行を挿入します。
- 親テーブルでの削除 — この処理についてカスケード規則または制限規則のいずれかを指定することができます。カスケードとは、従属テーブルが、削除される主キー値に一致する外部キー値を持つ場合、その一致する値を持つ行がすべて従属テーブルから削除されることを意味します。

制限規則とは、従属テーブルが、削除される主キー値と一致する外部キーを持つ場合、親テーブルの削除処理が失敗することを意味します。カスケード処理は再帰的です。従属テーブルが、カスケード外部キーの親テーブルで主キーを持つ場合、処理はその一連のデータで繰り返されます。

挿入規則

挿入規則は制限規則です。挿入される行の外部キーは、それぞれ親テーブルの主キー値と等価である必要があります。親テーブルは、挿入しようとする行の外部キーの親行を持っている必要があります、そうでない場合は挿入は失敗します。PSQL は、MicroKernel エンジンが、従属テーブルに自動的に挿入規則を適用するようにします。

更新規則

更新規則は制限規則でもあります。外部キー値は、親テーブルの対応する主キー値に更新される必要があります。親テーブルが外部キー値に対応する親行を持たない場合、更新は失敗します。

テーブルに外部キーを定義する際に明示的に更新規則として制限規則を指定することもできますが、指定しなかった場合、PSQL は MicroKernel エンジンに対しデフォルトでこの規則を順守させます。

削除規則

外部キーを定義する際に、削除規則として制限またはカスケードを明示的に指定することができます。明示的に削除規則を指定しなかった場合、PSQL は削除規則として制限をデフォルトと見なします。

- 削除規則として制限を指定した場合、PSQL は MicroKernel エンジンに、親テーブルから削除しようとする行のそれぞれについて、その行が別のテーブルの外部キーの親行であるかどうかを調べさせます。親行である場合、PSQL はステータス コードを返し、その行を削除しません。その親行を削除する前に、まず参照テーブルの対応する行をすべて削除する必要があります。
- 削除規則としてカスケードを指定した場合、PSQL は MicroKernel エンジンに、親テーブルから削除しようとする行のそれぞれについて、その行が別のテーブルの外部キーの親行であるかどうかを調べさせます。次に、MicroKernel エンジンはそのテーブルの子孫についてそれぞれ削除規則をチェックします。子孫のいずれかの削除規則が制限である場合、削除は失敗します。すべての子孫の削除規則がカスケードである場合、PSQL は元の親テーブルへの参照パス上のすべての従属行を削除します。

次のガイドラインは、外部キーの削除規則を決定します。

- 2 つ以上のテーブルのサイクルでは、テーブルそれ自体に対して連鎖削除できません。したがって、サイクル内の少なくとも 2 つの従属テーブルはカスケード削除規則であってはいけません。
- 1 つのテーブルから別のテーブルへのすべてのパスで最後の削除規則は同じである必要があります。
- 外部キーの削除規則がカスケードの場合、外部キーを含むテーブルは、削除トリガーが定義されていないけません。
- 外部キーを持つテーブルに削除トリガーが定義されている場合、削除規則は制限規則である必要があります。

PSQL はこれらのガイドラインを参照制約の定義されているデータベース上で強制します。これらのガイドラインに違反する削除規則を宣言しようとする、PSQL はエラーの発生を示すステータス コードを返します。

PSQL は、テーブルから従属行を削除する際、発生し得る例外を回避するために削除規則のガイドラインを強制します。これらのガイドラインがなければ発生する例外を次に示します。

連鎖削除サイクルの例外

2 つ以上のテーブルのサイクルでは、テーブルそれ自体に対して連鎖削除できません。したがって、サイクル内の少なくとも 2 つの従属テーブルは制限削除規則である必要があります。

次のステートメントを実行するとします。

```
DELETE FROM Faculty
```

Faculty と Department テーブルの関係により、Faculty からの行の削除は、まず Faculty から、次に Department から行を削除します。Department の名前の制限規則により、ここでカスケード削除は停止します。

PSQL が Faculty テーブルから行を削除する順によって、結果に矛盾が生じることがあります。ID が 181831941 の行を削除しようすると、その削除処理は失敗します。Department の Name 列の制限規則により、PSQL は、主キーの値が Mathematics と等しい Department テーブルの最初の行を削除することができません。これは、Faculty の 2 番目の行がこの行の主キーを参照し続けるためです。

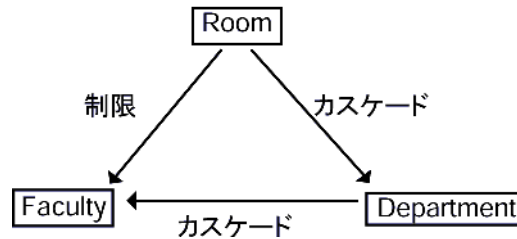
そうではなく、PSQL が、主キーが 179321805 と 310082269 に等しい Faculty の行を最初に削除した場合、Faculty と Department のすべての行が削除されます。

この例の DELETE ステートメントの結果には一貫性があるので、行は削除されません。

複数のパスにおける例外

複数の連鎖削除パスからの削除規則は同一である必要があります。図2はこのガイドラインを使用しないと発生する可能性のある1つの例外を示しています。図中の矢印は従属テーブルを指しています。

図2 複数のパスの例外



Faculty は Room に対し、異なる削除規則を持つ複数の連鎖削除パスで連鎖削除になっています。次のステートメントを実行するとします。

```
DELETE FROM Room
  WHERE Building_Name = 'Bhargava Building'
  AND Number = 302;
```

操作が成功するかどうかは、PSQL が Faculty と Department の削除規則を確実にするため、これらにアクセスする順序に依存します。

- 最初に Faculty にアクセスする場合、Room と Faculty の関係が制限規則であるため、削除処理は失敗します。
- 最初に Department にアクセスする場合、Department と Faculty で順次処理され、削除処理は成功します。

問題を回避するため、PSQL は、Faculty へ導く両方のパスに対する削除規則が同一であることを保証します。

サンプル データベースの参照整合性

このセクションでは、サンプル データベースのテーブルと参照制約定義を説明します。

Course テーブルを作成する

次のステートメントは Course テーブルを作成します。

```
CREATE TABLE Course
  (Name CHAR(7) CASE,
   Description CHAR(50) CASE,
   Credit_Hours USMALLINT,
   Dept_Name CHAR(20))
```

Course テーブルに主キーを追加する

次のステートメントは、Course テーブルに主キー (Name) を追加します。

```
ALTER TABLE Course
  ADD PRIMARY KEY (Name);
```

参照制約を使用して Student テーブルを作成する

次のステートメントは Student テーブルを作成し、参照制約を定義します。

```
CREATE TABLE Student
  (ID UBIGINT,
   PRIMARY KEY (ID),
   Cumulative_GPA NUMERICSTS(5,3),
   Tuition_ID INTEGER,
   Transfer_Credits NUMERICSA(4,0),
   Major CHAR(20) CASE,
   Minor CHAR(20) CASE,
   Scholarship_Amount DECIMAL(10,2),
   Cumulative_Hours INTEGER)

CREATE UNIQUE INDEX Tuition_ID ON Student(ID)

ALTER TABLE Student ADD CONSTRAINT S_Tuition
  FOREIGN KEY (Tuition_ID)
  REFERENCES Tuition
  ON DELETE RESTRICT
```

データベース セキュリティの管理

PSQL のセキュリティ オプションを使用すると、特定のユーザーに対し、データ列の操作を制限することができます。これらの制限は、ユーザーがテーブルの特定の列しか見えないようにすることから、すべての列を見ることができるが更新できないようにすることまで、さまざまな範囲で行うことができます。PSQL は、データベース許可について、オペレーティング システムのファイルおよびディレクトリ権限を想定しません。デフォルトで、PSQL を使用してデータベースにアクセスするすべてのユーザーは、データに読み書きする完全なアクセス権を持ちます。このアクセスを制限し、PSQL を使用してデータベースを不当な更新またはアクセスから保護するために、データベース セキュリティを有効にし、定義する必要があります

PSQL のセキュリティ ステートメントにより、データベースへのアクセスを制限する次の動作を行うことができます。

- データベースのセキュリティを有効にする。
- ユーザーおよびユーザー グループを識別し、パスワードを割り当てる。
- ユーザーおよびユーザー グループに権限を付与する。
- ユーザーおよびユーザー グループの権限を取り消す。
- データベースのセキュリティを無効にする。
- データベースに定義されたセキュリティに関する情報を取得する。

データベース権限の理解

表 47 は、ユーザーおよびユーザー グループに付与することができる権限を示しています。

表 47 データベース権

アクセス権	説明
Login	ユーザーがデータベースにログインすることを許可します。ユーザーとパスワードを作成するときにこのアクセス権を割り当てます。ただし、Login 権はユーザーがデータにアクセスすることは許可しません。ユーザーがデータにアクセスできるようにするには、ほかのアクセス権を割り当てる必要があります。ユーザー グループに Login 権を割り当てることはできません。
Create Table	ユーザーが新規テーブル定義を作成できるようにします。ユーザーはテーブル作成時に自動的にテーブルへのフル アクセス権を持ちますが、Master ユーザーは後でテーブルの読み取り、書き込み、変更のアクセス権を取り消すことができます。Create Table 権はグローバル権とも呼ばれます。このアクセス権がデータ辞書全体にも適用されるからです。
Select	ユーザーがテーブルの情報を照会することを許可します。Select 権は特定の列にもテーブル全体にも与えることができます。
Update	ユーザーに指定した列またはテーブルの情報を更新するアクセス権を与えます。Update 権は特定の列にもテーブル全体にも与えることができます。
Insert	ユーザーがテーブルに新しい行を追加できるようにします。Insert 権はテーブル レベルでのみ付与することができます。
Delete	ユーザーがテーブルから情報を削除できるようにします。Delete 権はテーブル レベルでのみ付与することができます。
Alter	ユーザーがテーブル定義を変更できるようにします。Alter 権はテーブル レベルでのみ付与することができます。
References	ユーザーがテーブルを参照する外部キー参照を作成できるようにします。References 権は、参照制約を定義するのに必要です。
All	Select、Update、Insert、Delete、Alter および References 権を含みます。

あるタイプのアクセス権を、データベース全体または特定のデータベース要素に割り当てることができます。たとえば、Update 権をユーザーまたはユーザー グループに割り当てた場合、これを一定のテーブルまたはテーブル内の列に制限することができます。それに反して、Create Table 権をユーザーまたはユーザー グループに割り当てた場合、そのユーザーまたはユーザー グループはデータベース全体に Create Table 権を持ちます。単一のテーブルまたは列に対して Create Table 権を適用することはできません。

Create Table と Login 権はデータベース全体に適用される一方、そのほかのアクセス権はすべてテーブルに適用されます。さらに、Select および Update 権をテーブル内の個々の列に適用することができます。

データベース セキュリティの確立

次の 9 つの手順はデータベースのセキュリティを確立する一般的な方法を表します。

- 1 セキュリティを確立するデータベースにログインします。
データベースへのログインの詳細については、『[PSQL User's Guide](#)』を参照してください。
- 2 マスター ユーザーを作成し、SET SECURITY ステートメントを使用してマスター パスワードを指定し、データベースのセキュリティを有効にします。
マスターとしてセキュリティを有効にすると、マスター ユーザーの名前は *Master*（大文字小文字を区別）となり、セキュリティを有効にしたときに指定したパスワードがマスター パスワード（大文字小文字を区別）となります。詳細については、「[セキュリティの有効化](#)」を参照してください。
- 3 任意：PUBLIC グループに最低限のアクセス権のセットを定義します。
すべてのユーザーは自動的に PUBLIC グループに所属します。詳細については、「[PUBLIC グループにアクセス権を付与する](#)」を参照してください。
- 4 任意：CREATE GROUP ステートメントを使用してユーザー グループを作成します。
システムで必要な数のグループを作成することができます。ただし、1 人のユーザーは PUBLIC 以外は 1 つのグループにしか所属できません。詳細については、「[ユーザー グループの作成](#)」を参照してください。
- 5 任意：GRANT CREATETAB および GRANT（アクセス権）ステートメントを使用して、各ユーザー グループへのアクセス権を付与します。詳細については、「[ユーザー グループへのアクセス権の付与](#)」を参照してください。
- 6 GRANT LOGIN ステートメントを使用してユーザー名とパスワードを指定し、ユーザーに Login 権を付与します。選択により、各ユーザーをユーザー グループに割り当てることができます。詳細については、「[ユーザーの作成](#)」を参照してください。
- 7 GRANT CREATETAB と GRANT（アクセス権）ステートメントを使用して、ユーザー グループのメンバーでない作成済みユーザーにアクセス権を与えます。詳細については、「[ユーザーへのアクセス権の付与](#)」を参照してください。
- 8 任意：不当な Btrieve アクセスからファイルを保護するために、データベースをバウンド データベースにします。バウンド データベースの詳細については、「[データベース権限の理解](#)」を参照してください。

セキュリティの有効化

セキュリティを有効にするには SET SECURITY ステートメントを使用できます。それに応え、PSQL はマスター ユーザーを作成します。マスター ユーザーはデータベースに対し完全な読み書きのアクセス権を持ちます。SET SECURITY ステートメントで指定したパスワードはデータベースのマスター パスワードになります。

次の例はデータベースのセキュリティを有効にし、マスター ユーザーのパスワードに Secure を指定します。

```
SET SECURITY = Secure;
```

パスワードでは大文字小文字が区別されます。

データの管理

セキュリティを有効にすると、PSQL は X\$User と X\$Rights というシステム テーブルを作成します。セキュリティを有効にすると、マスター ユーザーを除くすべてのユーザーは、明示的にほかのユーザーを作成してログイン権を与えない限り、データベースにアクセスできません。

ユーザー グループとユーザーの作成

セキュリティを有効にした後、データベースは 1 人のユーザー (Master) と 1 つのユーザー グループ (PUBLIC) を持ちます。ほかのユーザーにデータベースへのアクセスを提供するには、マスター ユーザーとしてデータベースにログインし、名前とパスワードを使用してユーザーを作成します。またユーザーをユーザー グループに組織することもできます。

PSQL はユーザー名の大文字小文字を区別します。したがって、マスター ユーザーとしてログインする場合、ユーザー名を Master と指定する必要があります。

ユーザー グループの作成

セキュリティ管理を単純化するために、ユーザーをユーザー グループに組織することができます。システムに必要な数のユーザー グループを作成することができます。ただし、1 人のユーザーは、PUBLIC に加えて 1 つのグループにしか所属できません。ユーザーは、いったん追加されたグループに所属すると、グループのアクセス権を継承します。そのユーザーに個別のアクセス権を与えることはできません。グループ内のユーザーのアクセス権は、グループ全体に定義されたアクセス権と異なるものにするにはできません。ユーザーに固有のアクセス権を与えるには、そのユーザーのためだけの特別なグループを作成します。

ユーザー グループを作成するには、CREATE GROUP ステートメントを使用します。

```
CREATE GROUP Accounting;
```

一度に複数のユーザー グループを作成することもできます。

```
CREATE GROUP Accounting, Registrar, Payroll;
```

ユーザー グループ名は大文字と小文字を区別し、30 文字以内で、データベースに対して一意である必要があります。ユーザー グループ名を付ける規則の詳細については、『Advanced Operations Guide』を参照してください。

ユーザーの作成

データベースにユーザーを作成するとき、PSQL は、対応するユーザー名とパスワードをデータベースのセキュリティ テーブルに記録します。ユーザーを作成するには GRANT LOGIN TO ステートメントを使用します。次の例は、ユーザー Cathy を作成し、パスワードとして Passwd を割り当てます。

```
GRANT LOGIN TO Cathy:Passwd;
```



メモ PSQL はパスワードを暗号化形式で格納します。したがって、X\$User テーブルに照会してユーザーのパスワードを表示することはできません。

ユーザー作成時にユーザーをユーザー グループに割り当てることもできます。たとえば、ユーザー Cathy を Accounting グループに割り当てるには次のステートメントを使用します。

```
GRANT LOGIN TO Cathy :Passwd IN GROUP Accounting;
```

ユーザー名とパスワードは大文字と小文字を区別します。ユーザー名とパスワードに対して許容される長さと文字については、『Advanced Operations Guide』の「[識別子の種類別の制限](#)」を参照してください。

アクセス権の付与

このセクションでは、ユーザー グループと個々のユーザーにアクセス権を与える方法を説明します。

PUBLIC グループにアクセス権を付与する

すべてのユーザーは自動的に PUBLIC グループに所属します。PUBLIC グループは特別なユーザー グループで、特定のデータベースのすべてのユーザーの最低限のアクセス権のセットを定義するのに使用します。PUBLIC グループに割り当てられたユーザーより少ないアクセス権を持つユーザーはいません。PUBLIC グループからユーザーを削除することはできません。PUBLIC グループに与えられているアクセス権をユーザーから取り消すことはできません。

デフォルトで、PUBLIC グループにはアクセス権が何もありません。PUBLIC グループのアクセス権を変更するには、GRANT (アクセス権) ステートメントを使用します。たとえば、次のステートメントはサンプル データベースのすべてのユーザーに、データベース内の Department、Course、Class テーブルを照会することを許可します。

```
GRANT SELECT ON Department, Course, Class TO PUBLIC;
```

PUBLIC グループにアクセス権を与えたら、別のグループを作成してより高いレベルのアクセス権を定義することができます。ユーザーをグループに所属させないことによって、個々のユーザーに、ほかのユーザーまたはユーザー グループとは異なるアクセス権を追加することもできます。

ユーザー グループへのアクセス権の付与

ユーザー グループにアクセス権を割り当て、そのグループにユーザー名とパスワードを追加することができます。こうすると、各ユーザーのアクセス権を個々に割り当てる手間を省くことができます。また、グループにセキュリティ権を割り当てた場合、セキュリティ管理はより簡単になります。グループ全体に 1 度に新しいアクセス権を与えたり既存のアクセス権を取り消したりすることにより、多数のユーザーのアクセス権を変更することができます。

ユーザー グループにアクセス権を与えるには、GRANT (アクセス権) ステートメントを使用します。たとえば、次のステートメントは、Accounting グループのすべてのユーザーが、サンプル データベースの Billing テーブルの定義を変更することを許可します。

```
GRANT ALTER ON Billing TO Accounting;
```



メモ Alter 権を与えることは、Select、Update、Insert、Delete 権を暗黙的に与えることを忘れないでください。

ユーザーへのアクセス権の付与

ユーザーを作成すると、そのユーザーはデータベースにログインできます。ただし、そのユーザーは、アクセス権を持つユーザー グループに所属させるか、そのユーザーにアクセス権を与えるかのいずれかをしなければ、データにアクセスすることはできません。

ユーザーにアクセス権を与えるには、GRANT (アクセス権) ステートメントを使用します。次の例は、ユーザー John に、サンプル データベースの Billing テーブルに行を挿入することを許可します。

```
GRANT INSERT ON Billing TO John;
```



メモ Insert 権を与えることは、Select、Update、Delete 権を暗黙的に与えることになります。

ユーザーとユーザーグループの削除

ユーザーを削除するには、REVOKE LOGIN ステートメントを使用します。

```
REVOKE LOGIN FROM Bill;
```

このステートメントは、データ辞書からユーザー Bill を削除します。ユーザーを削除すると、データベースのセキュリティを無効にしない限り、そのユーザーはデータベースにアクセスできません。

データの管理

次の例のように、複数のユーザーを一度に削除することもできます。

```
REVOKE LOGIN FROM Bill, Cathy, Susan;
```

ユーザー グループを削除するには、次の手順に従います。

- 1 次の例のように、グループからすべてのユーザーを削除します。

```
REVOKE LOGIN FROM Cathy, John, Susan;
```

- 2 グループを削除するには、DROP GROUP ステートメントを使用します。次の例ではグループ Accounting が削除されます。

```
DROP GROUP Accounting;
```

アクセス権の取り消し

ユーザーのアクセス権を取り消すには、REVOKE ステートメントを使用します。次の例は、サンプル データベースの Billing テーブルから、ユーザー Ron の Select 権を取り消します。

```
REVOKE SELECT  
    ON Billing  
    FROM Ron;
```

セキュリティの無効化

データベースのセキュリティを無効にするには、次の手順を行います。

- 1 マスター ユーザーとしてデータベースにログインします。
- 2 次のように NULL キーワードを指定し、SET SECURITY ステートメントを発行します。

```
SET SECURITY = NULL;
```

データベースのセキュリティを無効にすると、PSQL はデータベースからシステム テーブルの X\$User と X\$Rights を削除し、関連する DDF ファイルも削除します。



メモ USER.DDF と RIGHTS.DDF データ辞書ファイルを単純に削除するだけではセキュリティを無効にすることはできません。これらを削除してデータベースにアクセスを試みると、PSQL はエラーを返してデータベースへのアクセスを拒否します。

データベース セキュリティに関する情報の取得

データベース セキュリティを設定すると、PSQL はシステム テーブル X\$User と X\$Rights を作成します。システム テーブルはデータベースの一部であるため、適切なアクセス権があれば、それらにクエリを実行できます。

各システム テーブルの内容をすべて参照する場合は、『SQL Engine Reference』の「[システム テーブル](#)」を参照してください。

並行制御

MicroKernel エンジンとその自動修復機能は、データベースの物理的な整合性を制御します。PSQL は、MicroKernel エンジンのトランザクションとレコード ロック機能を使用して、論理的なデータの整合性を提供します。PSQL は MicroKernel エンジンと共に、以下のタイプの並行制御を提供します。

- トランザクションの分離レベル
- 明示的ロック
- パッシブ コントロール

トランザクション処理

トランザクション処理は、単一のテーブル内であっても複数のテーブルにまたがっていても、論理的に関連する一連のデータベース変更を識別できるようにし、これを 1 つの単位として完了させるように要求します。トランザクション処理には 2 つの重要な概念があります。

- 作業の論理的な単位、つまりトランザクションは、データベースの整合性を確実にするために 1 つの操作として扱う必要のある別個の操作のセットです。トランザクション中にミスを犯したり、問題があった場合、ROLLBACK WORK ステートメントを発行して既に行った変更を元に戻すことができます。

たとえば、教務係は学生の口座と支払われた金額の記入を 1 度の操作で行い、それから 2 番目の操作で残額を更新します。これらの操作をグループ化することにより、学生の会計は正確になります。

- ロック単位は、トランザクションが完了するまでほかのタスクがブロックされるデータの総数です（タスクは PSQL のセッションです）。ロックにより変更しようとしているデータがほかのタスクによって変更されるのを防ぎます。ほかのタスクもデータを変更した場合、PSQL は一貫性のある以前の状態にロールバックすることができません。したがって、トランザクション内では所定のロック単位に一度に 1 つのタスクしかアクセスできません。ただし、同一タスクに属する複数のカーソルはロック単位に同時にアクセスできます。

START TRANSACTION ステートメントがトランザクションを開始します。トランザクション中に完了したいすべてのステートメントを発行したら、COMMIT WORK ステートメントを発行してトランザクションを終了します。COMMIT WORK ステートメントはすべての変更を保存し、これを恒久的なものにします。



メモ START TRANSACTION および COMMIT WORK は、ストアード プロシージャでのみ使用できます。これら 2 つの SQL ステートメントの詳細については、『SQL Engine Reference』を参照してください。

操作の 1 つでエラーが発生した場合、トランザクションをロールバックし、エラーを修正した後、再試行することができます。たとえば、いくつかのテーブルに関連する更新を行う必要があるけれども、更新の 1 つが失敗した場合、既に行った更新をロールバックすることができるので、データは矛盾しません。

2 つのタスクが ログイン セッションを共有していて、セッションを開いたタスクが、もう 1 つのタスクがトランザクションを完了する前にログアウトした場合にも PSQL は自動的にロールバック操作を行います。

トランザクションの開始と終了

トランザクションを開始するには、ストアード プロシージャ内で START TRANSACTION ステートメントを発行します。トランザクション中に完了したいすべてのステートメントを発行したら、COMMIT WORK ステートメントを発行して変更をすべて保存し、トランザクションを終了します。

```
START TRANSACTION;
UPDATE Billing B
  SET Amount_Owed = Amount_Owed - Amount_Paid
  WHERE Student_ID IN
    (SELECT DISTINCT E.Student_ID
     FROM Enrolls E, Billing B
      WHERE E.Student_ID = B.Student_ID);
COMMIT WORK;
```

START TRANSACTION ステートメントの詳細については『SQL Engine Reference』を参照してください。

ネストされたトランザクションへのセーブポイントの使用

SQL トランザクションでは、セーブポイントと呼ばれるマーカーを定義することができます。セーブポイントを使用すると、トランザクション内のセーブポイント以降の変更を元に戻して最後のコミットを要求する前までの変更を継続して追加したり、トランザクション全体を中止することができます。

トランザクションを開始するには、START TRANSACTION ステートメントを使用します。ROLLBACK または COMMIT WORK ステートメントを発行するまでトランザクションはアクティブです。

セーブポイントを設定するには、SAVEPOINT ステートメントを使用します。

```
SAVEPOINT SP1;
```

セーブポイントにロールバックするには、ROLLBACK TO SAVEPOINT ステートメントを使用します。

```
ROLLBACK TO SAVEPOINT SP1;
```

セーブポイント名は、現在の SQL トランザクション内の現在アクティブなセーブポイントを指定する必要があります。このセーブポイントを設定した後の変更はキャンセルされます。

セーブポイントを削除するには、RELEASE SAVEPOINT ステートメントを使用します。

```
RELEASE SAVEPOINT SP1;
```

このステートメントは SQL トランザクションがアクティブな場合のみ使用できます。

COMMIT WORK ステートメントを発行した場合、現在の SQL トランザクションによって定義されたセーブポイントはすべて破棄され、トランザクションがコミットされます。



メモ ROLLBACK TO SAVEPOINT と ROLLBACK WORK を混同しないでください。前者は指定したセーブポイントまでの操作をキャンセルし、一方後者は最も外側のトランザクションとその中にあるセーブポイントをすべてキャンセルします。

セーブポイントはトランザクションをネストする方法を提供します。これによりアプリケーションは、一連のステートメントが正常に完了するのを待つ間、トランザクション内の前の操作を保存することができます。たとえば、この目的で WHILE ループを使用することができます。最初の試行で失敗する可能性のある一連のステートメントの開始前にセーブポイントを設定することができます。トランザクションが進行する前に、このサブトランザクションが正常に完了する必要があります。失敗した場合、サブトランザクションはセーブポイントにロールバックし、そこから再試行できます。サブトランザクションが成功した場合、トランザクションの残りの部分が続行されます。

SAVEPOINT ステートメントを発行するときは SQL トランザクションがアクティブである必要があります。



メモ MicroKernel は各トランザクションが内部的にネストするレベルを合計 255 まで許可します。ただし、PSQL は INSERT、UPDATE、DELETE ステートメントでアトミシティを保証するために内部的にこれらのレベルをいくつか使用します。したがって、1 つのセッションでは事実上 253 を越えるセーブポイントを一度にアクティブにすることはできません。トランザクション中に INSERT、UPDATE、DELETE ステートメントが含まれていると、トリガーによってこの制限はさらに厳しくなります。この制限に達した場合は、セーブポイントの数か、トランザクションに含まれるアトミック ステートメントの数を減らします。

セーブポイント内でロールバックされた操作は、外側のトランザクション（1 つまたは複数）が正常に完了してもコミットされません。ただし、セーブポイント内で完了した操作は、最も外側のトランザクションによって、物理的にデータベースにコミットされる前にコミットされます。

たとえば、サンプルデータベースで、学生をいくつかのクラスに登録するトランザクションを開始するとします。最初の 2 つのクラスで学生を正常に登録したとしても、3 番目のクラスで失敗する可能性があります。これは、クラスが定員を満たしていたり、学生が登録している別のクラスと衝突するためです。学生をこのクラスに登録するのに失敗したとしても、前の 2 つのクラスへの登録をやり直したいとは思えないでしょう。

次のストアード プロシージャは、まず最初にセーブポイント SP1 を設定し、次に Enrolls テーブルにレコードを追加して学生をクラスに登録します。それからクラスへの現在の登録を決定し、クラスの最大定員と比較します。比較に失敗した場合、SP1 にロールバックします。成功した場合はセーブポイント SP1 を解放します。

```
CREATE PROCEDURE Enroll_student( IN :student ubigint, IN :classnum integer);
BEGIN
    DECLARE :CurrentEnrollment INTEGER;
    DECLARE :MaxEnrollment INTEGER;
    SAVEPOINT SP1;
    INSERT INTO Enrolls VALUES (:student, :classnum, 0.0);
    SELECT COUNT(*) INTO :CurrentEnrollment FROM Enrolls WHERE class_id = :classnum;
    SELECT Max_size INTO :MaxEnrollment FROM Class WHERE ID = :classnum;
    IF :CurrentEnrollment >= :MaxEnrollment
    THEN
        ROLLBACK to SAVEPOINT SP1;
    ELSE
        RELEASE SAVEPOINT SP1;
    END IF;
END;
```



メモ SQL レベルで操作する場合、トランザクションはインターフェイスによって異なる方法で制御されます。ODBC では、トランザクションは *SQLSetConnectOption* API の *SQL_AUTOCOMMIT* オプションを使用することにより、関連する *SQLTransact* API も使用して制御されます。

これらのステートメントの構文についての詳細は、『SQL Engine Reference』の各ステートメントの項を参照してください。

特に考慮すべき点

トランザクションは、次の操作には影響しません。

- 辞書定義の作成または変更を行うオペレーション。したがって、ALTER TABLE、CREATE GROUP、CREATE INDEX、CREATE PROCEDURE、CREATE TABLE、CREATE TRIGGER、および CREATE VIEW の各ステートメントの結果はロールバックできません。
- 辞書定義を削除するオペレーション。したがって、DROP DICTIONARY、DROP GROUP、DROP INDEX、DROP PROCEDURE、DROP TABLE、DROP TRIGGER および DROP VIEW の各ステートメントの結果はロールバックできません。
- セキュリティ権の割り当てまたは削除を行うオペレーション。したがって、CREATE GROUP、DROP GROUP、GRANT (アクセス権)、GRANT CREATETAB、GRANT LOGIN、REVOKE (アクセス権)、REVOKE CREATETAB および REVOKE LOGIN の各ステートメントの結果はロールバックできません。

トランザクション内でこれらの操作のいずれかを試行し PSQL がステートメントを完了した場合、結果をロールバックすることはできません。

トランザクション中に、既にあるテーブルを参照している場合、トランザクション中にそのテーブルを変更または削除することはできません。つまり、辞書定義を変更することはできません。たとえば、トランザクションを開始し、Student テーブルにレコードを挿入し、Student テーブルを変更しようとする、ALTER ステートメントは失敗します。このトランザクションから操作をコミットし、それからテーブルを変更する必要があります。

分離レベル

同様にトランザクション中にあるほかのユーザーからトランザクションが分離する範囲を定義することにより、分離レベルはトランザクション ロック単位の適用範囲を決定します。分離レベルを使用すると、PSQL は、指定した分離レベルに応じて自動的にページまたはテーブルをロックします。これらの自動ロックは、PSQL が内部

データの管理

的に制御するものですが、**暗黙ロック**または**トランザクション ロック**と呼びます。アプリケーションが明示的に指定したロックは明示的ロックと呼びます。以前はレコード ロックと呼んでいました。

PSQL はトランザクションのために 2 つの分離レベルを提供します。

- 排他（アクセスするデータ ファイル全体をロックします）。ODBC の分離レベル `SQL_TXN_SERIALIZABLE` に相当します。
- カーソル安定性（アクセスする行またはページをロックします）。ODBC の分離レベル `SQL_TXN_READ_COMMITTED` に相当します。

分離レベルは、ODBC API の `SQLSetConnectOption` を使用して設定します。

排他的分離レベル（SQL_TXN_SERIALIZABLE）

排他的分離レベルを使用する場合、ロック単位はデータ ファイル全体です。排他トランザクション内で 1 つまたは複数のファイルにアクセスすると、ファイルは、トランザクション内のほかのユーザーが行う同様のアクセスからロックされます。このタイプのロックは、同時に同一テーブルにアクセスを試みるアプリケーションが非常に少ない場合や、トランザクションが行われている間にファイルの大部分がロックされるような場合に最も有効です。

PSQL は、トランザクションが終了するとファイルのロックを解除します。排他トランザクション中にテーブルにアクセスする場合、次の状態になります。

- トランザクションが終了するまで、ほかのトランザクション中のタスクは、そのテーブルに対する行の読み込み、更新、削除、挿入を行えません。
- ほかのトランザクション中でないタスクは、そのテーブルの行を読むことができますが、更新、削除、挿入はできません。
- 同一タスク内の複数のカーソルはテーブル内のどの行も読むことができます。ただし、特定のカーソルで更新、削除、挿入操作を実行すると、PSQL はそのカーソルのためにデータ ファイル全体をロックします。

排他的分離レベルを使用して結合ビューを介してテーブルにアクセスする場合、PSQL はビュー内でアクセスされたすべてのファイルをロックします。

カーソル安定性分離レベル（SQL_TXN_READ_COMMITTED）

MicroKernel エンジンではデータ ファイルを一連のデータ ページとインデックス ページとして保持します。カーソル安定性分離レベルを使用する場合、ロック単位はデータ ファイルではなく、データ ページまたはインデックス ページです。カーソル安定性トランザクション内でレコードを読み込むと、PSQL はこれらのレコードが含まれるデータ ページをロックし更新可能にします。しかし、複数のトランザクション中のタスクによってテーブルが並行アクセスされることは許可します。これらのレコード ロックは、ほかのレコードのセットを読み込む場合にのみ解放されます。PSQL はレベル カーソル安定性をサポートします。これによりアプリケーションが同時に複数のレコードをフェッチできるためです。

さらに、データ ページまたはインデックス ページに対する変更は、次の読み込み操作を発行したとしても、トランザクションの継続中これらのレコードをロックします。操作をコミットまたはロールバックするまで、トランザクション中のほかのユーザーはこれらのロックされたレコードにアクセスすることはできません。ただし、ほかのアプリケーションは、それぞれのトランザクション内から、同一ファイルの別のページをロックすることはできます。

カーソル安定性トランザクション中にファイルにアクセスする場合、PSQL はデータ ページおよびインデックス ページを次のようにロックします。

- 行を読むことはできますが、更新したり削除したりすることはできません。PSQL は、次の行読み込み操作が行われるか、トランザクションを終了するまで、その行のあるデータ ページをロックします。
- 行内のインデックスでない列の更新、インデックスを含まない行のテーブルからの削除、インデックスを含まない新しい行のテーブルへの挿入を行うことができます。PSQL は、それに続く読み込み操作にかかわらず、残りのトランザクションの間中、その行のあるデータ ページをロックします。

- 行内のインデックス列の更新、インデックスを含む行のテーブルからの削除、インデックスを含む新しい行のテーブルへの挿入を行うことができます。PSQL は、それに続く読み込み操作にかかわらず、残りのトランザクションの間中、影響を受けるインデックス ページをデータ ページと同様ロックします。

カーソル安定性は、ほかのユーザーが同一データ ファイルのほかのデータ ページにアクセスすることを許可しながら、読み込んだデータを確実に安定した状態に保つことができます。カーソル安定性分離レベルでは、一度に読み込める行の数を制限することにより、一度にロックされるデータ ページ数が少なくなり、一般的にすべてのタスクでより優れた並行性を実現できます。これにより、ロックするページが少ないため、ほかのネットワーク ユーザーは、データ ファイルのより多くのページにアクセスできます。

ただし、アプリケーションが多数の行をスキャンまたは更新する場合、影響するテーブルからほかのユーザーを完全にロックする可能性が高くなります。したがって、小さなトランザクションで読み込み、書き込み、コミットを行う場合にカーソル安定性を使用するのが最も良い方法です。

カーソル安定性はサブクエリ内のレコードをロックしません。カーソル安定性は、行が返された状態が変更されないことを保証するのではなく、実際に返された行が変更されないことを保証します。

トランザクションと分離レベル

トランザクション内でデータにアクセスする場合はいつでも、PSQL はアクセスされたページまたはファイルをそのアプリケーションのためにロックします。ほかのアプリケーションは、ロックが解除されるまで、ロックされたページまたはファイルに書き込むことはできません。

カーソル安定性分離レベルを使用すると、結合ビューでテーブルにアクセスする場合、PSQL はビュー内のすべてのテーブルのアクセスされたページをロックします。カーソル安定性分離レベルを使用すると、結合ビューでテーブルにアクセスする場合、PSQL はビュー内のすべてのテーブルのアクセスされたページをロックします。

PSQL はノーウェイト トランザクションを実行します。別のタスクがロックしているレコードに、トランザクション内からアクセスした場合、PSQL はページまたはテーブルがロックされているか、デッドロックが検出されたことを知らせます。いずれの場合にも、トランザクションをロールバックし、再試行してください。PSQL では、同一のアプリケーション内から同一のデータ ファイルにアクセスする複数のカーソルを使用できます。

次の手順は、2 つのアプリケーションがトランザクション内から同一テーブルにアクセスする場合にどのように相互作用するかを示しています。手順には番号が付けられていて、発生した順を示します。

タスク 1	タスク 2
1. ビューをアクティブにします。	
	2. ビューをアクティブにします。
3. トランザクションを開始します。	
	4. トランザクションを開始します。
5. レコードをフェッチします。	
	6. 同一データ ファイルからレコードのフェッチを試行します。
	7. 両方のタスクがカーソル安定性を使用していて、タスク 2 が、既にタスク 1 がロックしているのと同じレコードをフェッチしようとする、ステータス コード 84 (レコードまたはページがロックされている) を受け取ります。どちらか一方のタスクが排他トランザクションを使用している場合は、ステータスコード 85 (ファイルがロックされている) を受け取ります。
	8. 必要に応じフェッチを再試行します。
9. レコードを更新します。	
10. トランザクションを終了します。	
	11. フェッチに成功します。

タスク 1	タスク 2
	12. レコードを更新します。
	13. トランザクションを終了します。

トランザクションは、ほかのアプリケーションの更新に対し、一時的にレコード、ページ、またはテーブルをロックするため、アプリケーションはトランザクション中にオペレーター入力のための中断を行ってはいけません。これは、オペレーターが応答するかトランザクションが終了されるまで、トランザクションからアクセスされているレコード、ページまたはテーブルを、ほかのどのアプリケーションも更新できないためです。



メモ カーソル安定性トランザクション内でのレコードの読み込みは、それに続く更新処理が競合なしに成功することを保証するものではありません。これは、PSQL が更新を完了するのに必要とするインデックスページを、ほかのアプリケーションが既にロックしていることがあるためです。

デッドロックの回避

デッドロック状態は、2つのアプリケーションが、一方が既にロックしたテーブル、データ ページ、インデックス ページ、またはレコードに対し操作を再試行する場合に発生します。デッドロックの発生を最小限に抑えるには、アプリケーションでトランザクションのコミットを頻繁に行います。アプリケーションから操作の再試行を行わないでください。PSQL はエラーを返す前に妥当な回数の再試行を行います。

排他的分離レベル下のデッドロック状態

排他的分離レベルを使用する場合、PSQL は、データ ファイル全体をほかのアプリケーションの更新からロックします。したがって、アプリケーションが同じ順序でデータ ファイルにアクセスしない場合、次の表のようにデッドロックが起こる可能性があります。

タスク 1	タスク 2
1. トランザクションを開始します。	
	2. トランザクションを開始します。
3. ファイル 1 からフェッチします。	
	4. ファイル 2 からフェッチします。
5. ファイル 2 からフェッチします。	
6. ロックのステータス コードを受け取ります。	
7. 手順 5 を再試行します。	
	8. ファイル 1 からフェッチします。
	9. ロックのステータス コードを受け取ります。
	10. 手順 8 を再試行します。

カーソル安定性分離レベル下のデッドロック状態

カーソル安定性分離レベルを使用する場合、アプリケーションがアクセスしているファイルのレコードまたはページ（アプリケーションがロックしていないレコードまたはページ）を、ほかのアプリケーションが読み込み、更新することができます。

パッシブ コントロール

アプリケーションが単一レコードのフェッチを行い、論理的に関連しない一連の更新処理を行う場合、PSQL の並行制御であるパッシブ メソッドを使用することができます。この方法を使用すると、トランザクションやレコード ロックを行わずに、レコードをフェッチ、更新、または削除することができます。これらの操作は楽観的更新および削除と呼ばれます。

タスクがトランザクションも明示的レコード ロックも使用しないで更新および削除操作を行う場合、デフォルトで、そのタスクはほかのタスクの変更を上書きできません。このデータの整合性を確実にするこの機能は、パッシブ コントロールで、楽観的並行制御と呼ばれることもあります。パッシブ コントロールでは、タスクはどのような種類のロックも行いません。既にフェッチしてあるレコードを別のタスクが変更した場合、更新または削除オペレーションを実行する前に、そのレコードを再度フェッチする必要があります。

パッシブ コントロールの下では、レコードをフェッチしてから更新または削除操作をする間に別のアプリケーションがそのレコードを更新または削除した場合、競合のステータス コードが返されます。これは、最初にデータをフェッチしてから、別のアプリケーションがそのデータに変更を加えたことを示します。競合のステータスコードを受け取った場合、更新または削除操作を実行する前にもう一度そのレコードをフェッチする必要があります。

パッシブ コントロールを使用すると、シングル ユーザー システムで設計されたアプリケーションを、ロック呼び出しを実装することなくネットワーク上で実行することができます。ただし、パッシブ コントロールは、負荷の軽いネットワーク環境で使用されるか、データがほとんど変化しないような場合にのみ有効です。負荷の高いネットワーク環境や変化の激しいデータの場合、パッシブ コントロールは有効ではありません。

PSQL データベースのアトミシティ

アトミシティの原則は、所定のステートメントの実行が完了しなかった場合、データベースに部分的またはあいまいな影響を残してはいけない、ということです。たとえば、あるステートメントが5レコードの内3レコードを挿入したあとに失敗して、その3レコードを元に戻さなかった場合、操作を再試行するときにデータベースの一貫性は失われています。ステートメントがアトミックで実行の完了に失敗した場合、すべての変更はロールバックされ、データベースの一貫性は保たれます。この例では、5レコードの挿入で1つでも失敗した場合は、1つのレコードも挿入されてはいけません。

アトミシティの規則は、複数のレコードまたはテーブルを変更するステートメントで特に重要です。また、アトミシティの規則は、失敗した操作の再試行をより簡単にします。前の試行による部分的な影響も残っていないことが保証されるからです。

PSQL は2つの方法でアトミシティを実施します。

- 1 UPDATE、INSERT、DELETE ステートメントはアトミックと定義されています。PSQL は複数のレコードまたは複数のテーブルの変更操作が失敗した場合、変更の影響がデータベースにまったく残らないことを保証します。

プロシージャの内部または外部のいずれで実行されたかにかかわらず、Update、Insert、Delete 操作でアトミシティは保証されます。

- 2 ストアド プロシージャを作成する際、ATOMIC として指定することができます。このようなプロシージャは、その実行全体にアトミシティの規則を適用します。したがって、ATOMIC プロシージャ内の UPDATE、INSERT、DELETE ステートメントがアトミックに実行されるだけでなく、そのプロシージャ内のほかのどのステートメントが失敗した場合でも、そのプロシージャの実行による影響はロールバックされます。

プロシージャ内のトランザクション制御

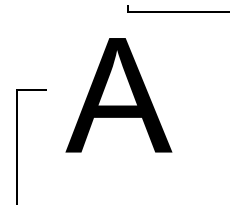
トリガーは常に外部のデータ変更ステートメント (INSERT、DELETE、または UPDATE) によって開始され、すべてのデータ変更ステートメントはアトミックであるため、次のステートメントは、トリガー内またはトリガーによって起動されるプロシージャ内では使用できません。

- START TRANSACTION
- COMMIT WORK
- ROLLBACK WORK (RELEASE SAVEPOINT および ROLLBACK TO SAVEPOINT を含む)

言い換えると、トリガーは ATOMIC 複合ステートメントと同じ規則に従います。

ユーザーが起動した COMMIT WORK、ROLLBACK WORK、RELEASE SAVEPOINT、ROLLBACK TO SAVEPOINT ステートメントは、(アトミシティの目的で) システムが開始したトランザクションを終了させることはありません。

国際的なソート規則を使用した照合順序のサンプル



この付録では、Btrieve に用意されている ISR テーブルを使用した言語固有のサンプル照合順序を示します。

この付録では、以下の言語でサンプル照合順序を示します。

- 「[ドイツ語のサンプル照合順序](#)」
- 「[スペイン語のサンプル照合順序](#)」
- 「[フランス語のサンプル照合順序](#)」

ドイツ語のサンプル照合順序

ここでは、ドイツ語の文字セットを使用する未ソートの文字列とソート済みの文字列のサンプルを示します。

- 「[未ソートのデータ](#)」
- 「[ソート済みのデータ](#)」

未ソートのデータ

Datei	abzüglich	Abriß	Äffin
Ähre	Rubin	aufwärmen	Jacke
ächten	Bafög	Behörde	berüchtigt
beschießen	zugereiste	Beschluß	Blitzgerät
Bürger	Abgänger	Dämlich	darüber
daß	Aufwasch	absägen	Defekt
dösen	drängeln	drüber	dürr
Efeu	Effekt	einfädeln	einschlägig
dunkel	englisch	Ente	einsetzen
Engländer	entführen	Bergführer	Haselnuß
Füllen	für	Zöllner	füßen
hätte	gefährden	gefangen	Gegenüber
gesinnt	Härte	Haß	Fußgänger
häßlich	hatte	Gewäschshaus	Kahl
Höhe	Jaguar	jäh	Jähzorn
Jux	Käfer	Kaff	Käfig
Kreisförmig	Kreißaal	Lüftchen	Jahr
luxuriös	Pflügen	pfütze	einhüllen
Reißbrett	Reißer	Prügel	Zögern
Abgang	Raub	Regreß	Zobel
Säge	Führer	Führung	regulär
schnüffler	Rübe	Zoll	Rübli
säen	Rätsel	Salz	Schnörkel
Abschluß	strategisch	Gespann	dünnkel
Gewähr	Zone	entblößen	Zugegen
Däne	Straßenkreuzung	Zügel	

ソート済みのデータ

Abgang	drängeln	Gewähr	regulär
Abgänger	drüber	Härte	Reißbrett
Abriß	dunkel	Haselnuß	Reißer
absägen	Dünkel	Haß	Rübe
Abschluß	dürr	häßlich	Rubin
abzüglich	Efeu	hatte	Rübli
ächten	Effekt	hätte	säen
Äffin	einfädeln	Höhe	Säge
Ähre	einhüllen	Jacke	Salz
aufwärmen	einschlägig	Jaguar	Schnörkel
Aufwasch	einsetzen	jäh	schnüffler
Bafög	Engländer	Jahr	Straßenkreuzung
Behörde	englisch	Jähzorn	strategisch
Bergführer	entblößen	Jux	Zobel
berüchtigt	Ente	Käfer	Zögern
bescheißen	entführen	Kaff	Zoll
Beschluß	Führer	Käfig	Zöllner
Blitzgerät	Führung	Kahl	Zone
Bürger	Füllen	Kreisförmig	zugegen
Dämlich	für	Kreißaal	Zügel
Däne	fußen	Lüftchen	Zugereiste
darüber	Fußgänger	luxuriös	
daß	gefährden	pflügen	
Datei	gefangen	Pfütze	
Defekt	Gegenüber	Prügel	
dösen	gesinnt	Rätsel	

スペイン語のサンプル照合順序

ここでは、スペイン語の文字セットを使用するソート済みの文字列と未ソートの文字列のサンプルを示します。

- 「[未ソートのデータ](#)」
- 「[ソート済みのデータ](#)」

未ソートのデータ

acción	añal	añoso	baja
abdomen	bético	betún	Borgoña
búsqueda	acá	zarigüeya	cañada
abdicación	cañamo	caos	cartón
cigüeña	clarión	cónsul	cúpola
chaqué	chófer	descortés	desparej
desparapajo	desteñir	educación	elaboración
émbolo	epítome	hórreo	época
estúpido	Eucaristía	flúido	horrendo
barbárico	garañón	garguero	gruñido
hélice	heróina	gárgara	garanon
herionómano	fréir	helio	horrible
íglú	ígneo	intentar	interés
ínterin	acompañanta	interior	jicara
jinete	judicial	lactar ⁷	lácteo
lúpulo	lustar	llana	llegada
llorar	judío	máquina	maraña
living	maravilla	lívido	marqués
llama	manómetro	marquesina	fábula
mí	miasma	obstáculo	obstante
opiata	ordeñar	ordinal	pabellón
pábilo	penumbra	peña	peor
perímetro	período	rábano	réplica
república	señorita	rabia	xilófono
periódico	sórdido	peón	tea
xilografía	tiña	tío	típico
zoo	ópera	tipo	tirón
té	sordina	repleto	según
segunda	tísico	manoseado	titán
señoría	títere	bebé	

ソート済みのデータ

abdicación	caos	época	hórreo
abdomen	cartón	estúpido	horrible
acá	cigüeña	Eucaristía	iglú
acción	clarión	fábula	ígneo
acompañanta	cónsul	flúido	intentar
añal	cúpola	fréir	interés
añoso	chaqué	garanon	ínterin
baja	chófer	garañón	interior
barbárico	descortés	gárgara	jícara
bebé	desparej	garguero	jinete
bético	desparapajo	gruñido	judicial
betún	desteñir	hélice	judío
Borgoña	educación	helio	lactar'
búsqueda	elaboración	heróina	lácteo
cañada	émbolo	herionómano	lívido
cañamo	epítome	horrendo	living
lúpulo	pábilo	tío	llegada
lustar	penumbra	típico	llorar
llama	peña	tipo	manómetro
llana	peón	tirón	manoseado
máquina	rábano	xilografía	
maraña	rabia	zarigüeya	
maravilla	repleto	zoo	
marqués	réplica		
marquesina	república		
mí	según		
miasma	segunda		
obstáculo	señoría		
obstante	señorita		
ópera	sórdido		
opiata	sordina		
ordeñar	té		
ordinal	tea		
pabellón	tiña		
peor	tísico		
perímetro	titán		
periódico	títtere		
período	xilófono		

フランス語のサンプル照合順序

ここでは、フランス語の文字セットを使用するソート済みの文字列と未ソートの文字列のサンプルを示します。

- 「未ソートのデータ」
- 「ソート済みのデータ」

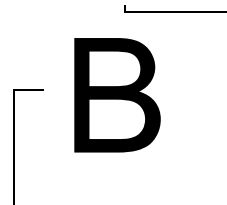
未ソートのデータ

ou	lésé	péché	999
OÙ	haïe	coop	caennais
lèse	dû	côlon	bohème
gêné	lamé	pêche	LÈS
cæsium	resumé	Bohémien	pêcher
les	CÔTÉ	résumé	Ålborg
cañon	du	Haie	pécher
cote	colon	l'âme	resume
élève	Canon	lame	Bohème
0000	relève	gène	casanier
élevé	COTÉ	relevé	Grossist
Copenhagen	côte	McArthur	Aalborg
Größe	cølibat	PÉCHÉ	COOP
gêne	révélé	révèle	Noël
île	aïeul	nôtre	notre
août	@@@@@	CÔTE	COTE
côté	coté	aide	air
modelé	MODÈLE	maçon	MÂCON
pèche	pêché	pechère	péchère

ソート済みのデータ

@@@@	coop	Haie	OÙ
0000	COOP	haïe	pèche
999	Copenhagen	île	pêche
Aalborg	cote	lame	péché
aide	COTE	l'âme	PÉCHÉ
aïeul	côte	lamé	pêché
air	CÔTE	les	pécher
Ålborg	coté	LÈS	pêcher
août	COTÉ	lèse	pechère
bohème	côté	lésé	péchère
Bohème	CÔTÉ	MÂCON	relève
Bohémien	du	maçon	relevé
caennais	dû	McArthur	resume
cæsium	élève	MODÈLE	resumé
Canon	élevé	modelé	résumé
cañon	gène	Noël	révèle
Casanier	gêne	NOËL	révélé
cølibat	géné	notre	
colon	Größe	nôtre	
côlon	Grossist	ou	

サンプル データベース テーブル と参照整合性



この付録では、以下の項目について説明します。

- 「[Demodata サンプル データベースの概要](#)」
- 「[Demodata サンプル データベースの構造](#)」
- 「[Demodata サンプル データベースの参照整合性](#)」
- 「[Demodata サンプル データベースのテーブル設計](#)」

Demodata サンプル データベースの概要

サンプル データベースの DEMODATA は PSQL 製品の一部として提供されており、データベースの概念と技法を図解するためにマニュアルで頻繁に使用されます。PSQL に関しては既によく理解されていると思いますが、この付録の情報をもう一度見直して、新しいサンプル データベースに慣れてください。

大学関連の環境で仕事をしていなくても、これらのサンプル データベースの例をテンプレートおよび参考として利用すれば、独自の情報システムの設計と開発を容易に行えます。ここに示す例は実際の生活の場面を反映しているので、この例に示すサンプル クエリなど機能を利用することができます。

Demodata サンプル データベースの構造

データベースの物理構造は、リレーショナル データベースの要素であるテーブル、列、行、キー、インデックスで構成されています。

このデータベースに含まれる 10 個のテーブル間にはさまざまな関係があります。このデータベースには、学生、教職員、授業、登録などに関するデータが含まれています。

前提条件

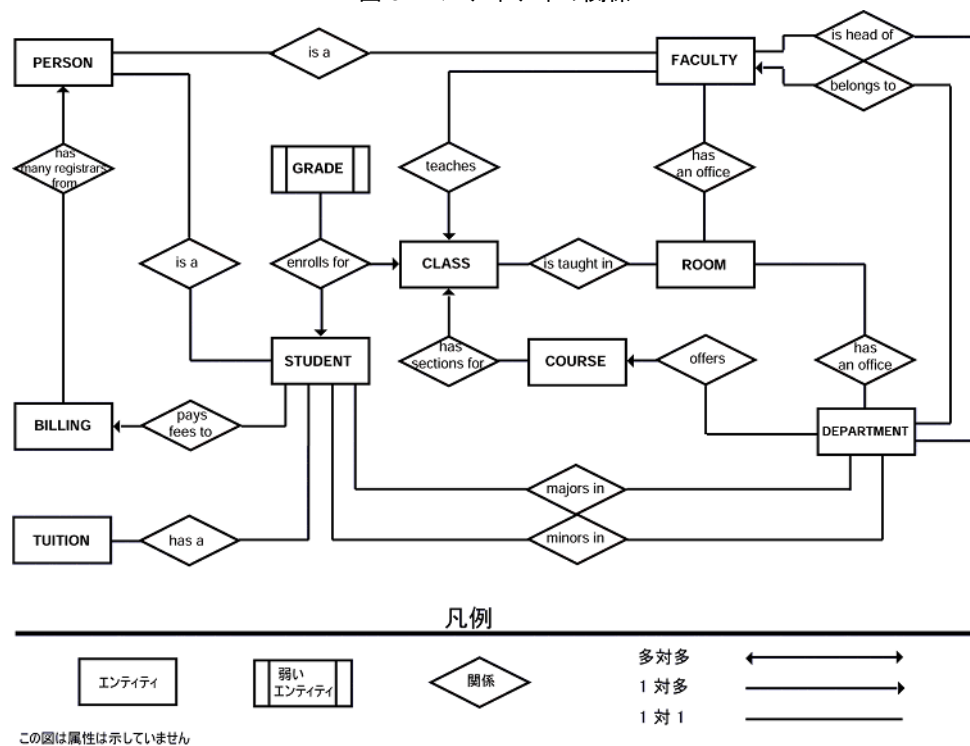
以下に、データベースを構築したときのいくつかの前提条件を示します。

- データベースの適用範囲は 1 学期です。
- 学生は、同じコースを何回も受講できません。たとえば、学生は Algebra I の Sections 1 と 2 に受講登録できません。
- 教職員は学生でもかまいませんが、同じ授業での講義と受講登録は行えません。
- どのコースも 1 つの学部だけから提供されます。
- 学生が評価を受けるには、学生を授業に受講登録し、その授業を教える教職員を任命しなければなりません。
- 教職員が所属する学部は 1 つですが、複数の学部で講義を行うことができます。
- すべての学生は、アメリカの社会保険番号基準に基づいた学生 ID を持っています。
- すべての教職員は、アメリカの社会保険番号基準に基づいた教職員 ID を持っています。
- その他すべての職員は、アメリカの社会保険番号基準に基づいた個人 ID を持っています。
- 教室は、同じビル内で固有です。
- 2 つの授業を同じ教室で同時に教えることはできません。
- 教職員は、与えられた時間に 1 つの授業しか教えることができません。
- 授業に受講登録するための前提条件は必要ありません。
- 学部は専攻を意味します。
- 1 つの授業は、学期を通じて 1 人の教職員しか担当できません。
- 電話番号または郵便番号と州には相関関係がありません。
- 教務係は、教職員または学生であってははいけません。
- ある人がデータベースに入力されると、すべての質問に答えなければならない調査か、あるいは、質問にまったく答える必要がない調査を行うことができます。
- コースの履修単位時間は、必ずしも、授業が行われる時間数と同じではありません。
- 電子メール アドレスは、一意のアドレスである必要はありません。

エンティティの関係

エンティティは、データベース内の主なコンポーネントを記述するオブジェクトです。データベースを設計する場合、エンティティとそれらの相互の関係を定義してから先に進むことが大切です。University データベースでは、CLASSES、STUDENTS、FACULTY、GRADES などがエンティティです。エンティティとそれらの相互関係については、図 3 で概説しています。

図 3 エンティティの関係



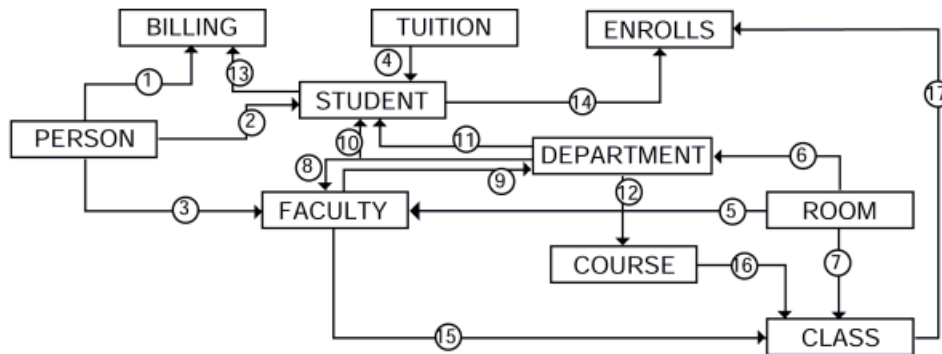
GRADES は弱いエンティティです。このエンティティは授業を受ける学生に依存しているので、その存在は他のエンティティの有効性に依存しています。STUDENT テーブルと FACULTY テーブルが共通の情報を作成するのは、学生が教職員になったり、教職員が学生になる場合があるからです。共通の情報は、PERSON テーブルにあります。

Demodata サンプル データベースの参照整合性

ここでは、サンプルデータベースの参照整合性（RI）設計について説明します。

図 4 は、University データベース内の様々なテーブル間に存在する参照制約を示したものです。ボックスはテーブルを表します。一方向の方向矢印は、親テーブルから参照用テーブルまでの参照制約を意味します。たとえば、制約番号 16 では、親テーブル Course で主キーを参照する Class テーブルに外部キーが存在します。

図 4 サンプル データベースの RI 構造



メモ 図 4 は、依存関係グラフの役割も果たします。このグラフは、物理設計を行うときにどのテーブルを先に作成しなければならないかを示します。

RI に参加するテーブル、列、キーは、以下のように定義されます。

表 48 RI に関連するテーブルと列

制約	参照用テーブル	外部キー	参照されるテーブル	主キー
1	BILLING	Registrar_ID	PERSON	ID
2	STUDENT	ID	PERSON	ID
3	FACULTY	ID	PERSON	ID
4	STUDENT	Tuition_ID	TUITION	ID
5	FACULTY	Building_Name、Room_Number	ROOM	Building_Name、Number
6	DEPARTMENT	Building_Name、Room_Number	ROOM	Building_Name、Number
7	CLASS	Building_Name、Room_Number	ROOM	Building_Name、Number
8	FACULTY	Dept_Name	DEPARTMENT	Name
9	DEPARTMENT	Head_Of_Dept	FACULTY	ID
10	STUDENT	Major	DEPARTMENT	Name
11	STUDENT	Minor	DEPARTMENT	Name
12	COURSE	Dept_Name	DEPARTMENT	Name
13	BILLING	Student_ID	STUDENT	ID

表 48 RI に関連するテーブルと列

制約	参照用テーブル	外部キー	参照されるテーブル	主キー
14	ENROLLS	Student_ID	STUDENT	ID
15	CLASS	Faculty_ID	FACULTY	ID
16	CLASS	Course_Name	COURSE	Name
17	ENROLLS	Class_ID	CLASS	ID

Demodata サンプル データベースのテーブル設計

以下に、サンプル データベースのテーブルへの手引きを示します。以下の情報は、各テーブルに収録されています。

- テーブル内の列
- 各列のデータ型
- 列のサイズまたは長さ（バイト数）
- キー（列がキーでない場合は空白）
- インデックス（列にインデックスがない場合は空白）
- 「BILLING テーブル」
- 「CLASS テーブル」
- 「COURSE テーブル」
- 「DEPT テーブル」
- 「ENROLLS テーブル」
- 「FACULTY テーブル」
- 「PERSON テーブル」
- 「ROOM テーブル」
- 「STUDENT テーブル」
- 「TUITION テーブル」

BILLING テーブル

列	データ型	サイズ	キー
Student_ID	UBIGINT	8	主キー、外部キー
Transaction_Number	USMALLINT	2	主キー
Log	TIMESTAMP	8	
Amount_Owed	DECIMAL	7.2	
Amount_Paid	DECIMAL	7.2	
Registrar_ID	UBIGINT	8	外部キー
Comments	LONGVARCHAR	65500	

CLASS テーブル

列	データ型	サイズ	キー
ID	IDENTITY	4	主キー
Name	CHARACTER	7	外部キー
Section	CHARACTER	3	
Max_Size	USMALLINT	2	
Start_Date	DATE	4	

列	データ型	サイズ	キー
Start_Time	TIME	4	
Finish_Time	TIME	4	
Building_Name	CHARACTER	25	外部キー
Room_Number	UINTeger	4	外部キー
Faculty_ID	UBIGINT	8	外部キー

COURSE テーブル

列	データ型	サイズ	キー
Name	CHARACTER	7	主キー
Description	CHARACTER	50	
Credit_Hours	USMALLINT	2	
Dept_Name	CHARACTER	20	外部キー

DEPT テーブル

列	データ型	サイズ	キー
Name	CHARACTER	20	主キー
Phone_Number	DECIMAL	10.0	
Building_Name	CHARACTER	25	外部キー
Room_Number	UINTeger	4	外部キー
Head_of_Dept	UBIGINT	8	外部キー

ENROLLS テーブル

列	データ型	サイズ	キー
Student_ID	UBIGINT	8	主キー、外部キー
Class_ID	INTEGER	4	主キー、外部キー
Grade	REAL	4	

FACULTY テーブル

列	データ型	サイズ	キー
ID	UBIGINT	8	主キー、外部キー
Dept_Name	CHARACTER	20	外部キー
Designation	CHARACTER	10	
Salary	CURRENCY	8	
Building_Name	CHARACTER	25	外部キー
Room_Number	UINTeger	4	外部キー
Rsch_Grant_Money	FLOAT	8	

PERSON テーブル

列	データ型	サイズ	キー
ID	UBIGINT	8	主キー
First_Name	VARCHAR	15	
Last_Name	VARCHAR	25	
Perm_Street	VARCHAR	30	
Perm_City	VARCHAR	30	
Perm_State	VARCHAR	2	
Perm_Zip	VARCHAR	10	
Perm_Country	VARCHAR	20	
Street	VARCHAR	30	
City	VARCHAR	30	
State	VARCHAR	2	
Zip	VARCHAR	10	
Phone	DECIMAL	10.0	
Emergency_Phone	CHARACTER	20	
Unlisted	BIT	1	
Date_Of_Birth	DATE	4	
Email_Address	VARCHAR	30	
Sex	BIT	1	
Citizenship	VARCHAR	20	
Survey	BIT	1	

サンプル データベース テーブルと参照整合性

列	データ型	サイズ	キー
Smoker	BIT	1	
Married	BIT	1	
Children	BIT	1	
Disability	BIT	1	
Scholarship	BIT	1	
Comments	LONGVARCHAR	65500	

ROOM テーブル

列	データ型	サイズ	キー
Building_Name	CHARACTER	25	主キー
Number	INTEGER	4	主キー
Capacity	USMALLINT	2	
Type	CHARACTER	20	

STUDENT テーブル

列	データ型	サイズ	キー
ID	UBIGINT	8	主キー、外部キー
Cumulative_GPA	DECIMAL	5.3	
Tuition_ID	INTEGER	4	外部キー
Transfer_Credits	DECIMAL	4.0	
Major	CHARACTER	20	外部キー
Minor	CHARACTER	20	外部キー
Scholarship_Money	DECIMAL	19.2	
Cumulative_Hours	SMALLINT	2	

TUITION テーブル

列	データ型	サイズ	キー
ID	INTEGER	4	主キー
Degree	VARCHAR	4	
Residency	BIT	1	

列	データ型	サイズ	キー
Cost_Per_Credit	REAL	4	
Comments	LONGVARCHAR	65500	

