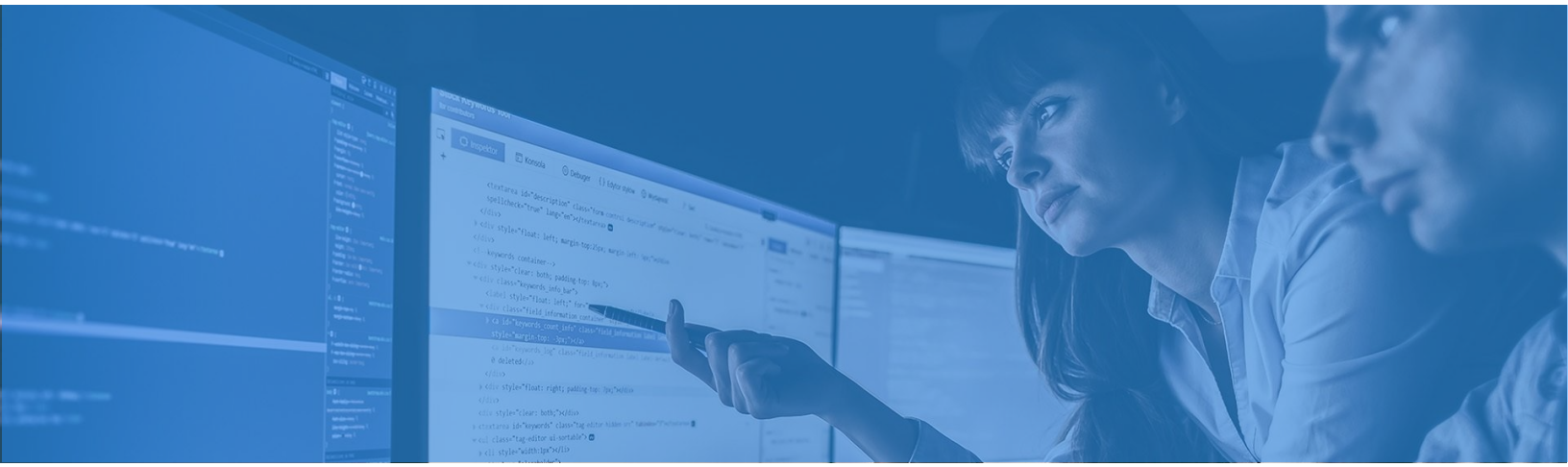

SQLite の限界を超えたその先にいる データベース Actian Zen

最先端のエッジ データ管理システムの開発者に向けて

IoT や複雑なインストルメンテーション領域における組み込み開発者の圧倒的多数は C、C++、または C# を使用してデータ処理およびローカル分析を行っています。その要因のひとつは、デバイスや内部システム コンポーネントだけでなく、もっと複雑なデジタル的に強化された機械のダイレクト I/O を、`inp()` および `outp()` ステートメントのいくつかのバリエーションを使用して処理しやすいことです。また、収集したデータの操作も、`fopen()`、`fclose()`、`fread()`、`fwrite()` などの使い慣れたファイル システム ステートメントを使用して簡単に行えます。これが最も楽なやり方です。プログラミングのクラスを受講している人（または、単に時間を割いてプログラミング方法を学習している人）はほとんど誰でも、これらのステートメントを使用してファイル システム レベルのデータを操作できます。

問題は、既存のファイル システムが非常にシンプルだということです。ファイル システム自体ではあまり多くの処理を行えません。ドキュメントおよびレコード管理、テーブルの作成、インデックス作成、ソート、および管理などとなると、役に立つステートメントはないので、`DIY`（自分でやる）しかありません。また、ここでは、まれなアクティビティやロケット科学レベルのアクティビティについて述べようとしているのでもありません。これらは、どのデータベース システムでも見られる日常のアクティビティです。

組み込みアプリケーションの開発者はデータベースを**嫌悪しており**、その理由を理解するのは難しいことではありません。従来のデータベースは大きすぎて IoT 環境やモバイル環境に配置できずにいます。それらはコストが高く、プロビジョニングが難しい場合があります。常に注意が必要となる傾向にあり、その種の注意は、費用のかかる DBA のみが管理できます。たしかに、いくつかの新しいオープン ソース データベースは、Oracle や Microsoft のデータベースよりはやや低コストです（ただし、それでもサービスやサポートは有償です）が、それらの大半はクラウド用に構築されています。IoT やモバイルのために構築作

業を行う開発者は、デバイス自体に組み込める、エッジ用やゲートウェイ用に設計されたものを求めています。

前時代の遺物から抜け出す

前述した前時代の IT 業界の遺物の中で、例外として注目を浴びてきた 1 つのオープン ソース データベースがあります。SQLite です。これはサーバーレス、移植可能、組み込み可能で、最も人気のあるプログラミング言語からアクセスできる、非常にコンパクトなデータベースです。

SQLite は 20 年以上にわたって存在しており、Web およびモバイル アプリケーションにおけるデータ キャッシュとしてほとんど至るところで利用されています。SQLite は、その名が示すように SQL（standard query language：標準クエリ言語）を利用すると共に、他の有名なオープン ソース SQL データベース（MySQL、Postgres、MariaDB など）と違って、サーバーレスかつデータベース管理者不要で動作できます。開発者は SQLite を配置することで、トランザクションを中心としたデータ管理を SQL に依存でき、DIY のあらゆるファイル システム操作を行わずに済ませられることに気付きました。また、開発者は、SQLite を配置した場合のパフォーマンス上の利点にも気付きました。データベースは、大きなデータセットのブロック読み取りと書き込みをネイティブのファイル管理システムよりも約 35% 速く実行できます。

SQLite はシンプルなファイル管理システムから「進化のはしご」を一段上っていたため、アプリケーション開発者は SQLite を使用することで、初期のエッジのさまざまな使用事例に対し、フラット ファイルで行うよりも効率的に取り組めました。しかし、「進化のはしご」の低い段階にいる多くの被造物と同様に、SQLite にも限界はあります。将来のエッジ使用事例の大半で、複数のチャネルと I/O データ転送速度、複数のアプリケーション、プロセスおよびスレッド、高速での大型データ セットの操作、ならびに保存時および転送時のデータの保護が必要になります。SQLite は、これらのニーズを満たすようには設計されていませんでした。

シングル ユーザーの組み込みモバイル アプリケーションに SQLite が最適であると考え、SQLite を使用してより洗練されたエッジ アプリケーションを構築しようとしてきた開発者は、これらすべてのニーズに対応しようとしたとき課題に直面しました。この課題に対する今日の対応策は、シンプルなファイル管理システムの欠点に対処していた当時の対応策である DIY（何でも自分でやる）と基本的には変わっていません。SQLite を拡張して複数のチャネル、複数のプロセス、高速のエッジ データ環境を処理できるようにするため、DIY には、自社開発したコードや GitHub ダウンロード、および、サーバー サポートやセキュリティ、接続機能用のアドオン コンポーネントを提供する零細の開発ショップからの購入物のマッシュアップが含まれるようになりました。これは、より洗練されたレベルの DIY ですが、DIY であることに変わりはありません。

進化のはしごを一段上る

カタツムリはアメーバよりも進化していますが、ハヤブサなどの進化レベルまでにはまだ遠く及んでいません。将来のエッジおよび組み込みアプリのニーズを満たすとなると、開発者には地を這うものでなく、空を飛翔できるものが必要です。

これを実現するのがActian Zen データベース製品ファミリです。Zen と SQLite の関係は、SQLite とフラット ファイル管理システムの関係に似ています。Actian Zen は、最先端のエッジ データ管理のニーズを満たすのに必要な機能のすべてを提供します。

Actian Zen Core エディションは、SQLite をデータベースとして魅力的なものにしていた特性と同じ特性を開発者に提供します。つまり、データベース管理者不要、サーバーレス オプション、移植可能性、および C、C++、C# など最も人気のあるプログラミング言語用の組み込みサポートが備わっています。Actian

Zen は、NoSQL および SQL API を追加することでアクセス オプションを拡張しているほか、SWIG（simplified wrapper and interface generator）のサポートによってさらにアクセスを拡張しています。SWIG により、開発者は Windows および Linux システム用の Python、Perl、PHP を使用して NoSQL API にアクセスできるようになります。

しかしながら、Actian Zen に組み入れられた進化的な飛躍は、Actian Zen Core エディション（フットプリントは 5MB 未満）に類似する、サーバーやゲットウェイで実行される Edge エディション（フットプリントは 50MB 未満）が備わったほか、さらに強力なデバイスやコンテナーで実行できる Enterprise エディションおよび Cloud エディション（フットプリントは 200MB 未満）が備わったことです。どのエディションも同じアーキテクチャに基づいているため、SQLite の世界とは違って互いに補完し、相互作用します。

Actian Zen を使って構築する

Actian Zen をコード ベースに組み込むには、32/64 ビットの C/C++ ライブラリ（btrieveC.h、btrieveCpp.h、btrieveC.lib、btrieveCpp.lib）をプロジェクトに追加するだけで済みます。Perl、Python、または PHP を使用したい場合は、SWIG 用のインターフェイス ライブラリを追加することも必要です。SQL による Zen の呼び出しは、SQLite を操作する際に使用する SQL 呼び出しとほぼ同じです。ただし、SQLite のパフォーマンスの限界を克服しようと奮闘している IoT およびモバイルの開発者にとって刺激的なのは、NoSQL API（Btrieve 2）を使用できる選択肢があることです。Btrieve 2 API は、ファイル管理システムと同じぐらい簡単に使用できますが、対象となるデータベースの全機能へのアクセスを提供します。

Btrieve 2 API を使用すると、Zen データベース エンジンとのやり取りはどれくらい簡単になるでしょうか。見てみましょう。まず、btrieveCpp.h ヘッダーをアプリケーション コードにインポートする必要があります。

```
#include "btrieveCpp.h"
```

次に、エンジンに接続するため、以下のように `btrieveClient` オブジェクトのインスタンスを作成する必要があります。

```
BtrieveClient btrieveClient (0x4232, 0);
```

これで、必要な設定はすべて完了です。最初のパラメーターは `serviceAgentIdentifier` で、エンジンに対するアプリケーションの各インスタンスを識別します。"AA"（0x4141）より大きい 2 バイト値を指定できます。この例では 0x4232、つまり "B2" です。2 番目のパラメーターは `clientIdentifier` で、これには 2 バイト整数を指定する必要があります。マルチスレッド アプリケーションを作成している場合は、各スレッドが、エンジンに対する一意のクライアント識別子を提供する必要があります。

次に、データ収集を行いましょう。

Zen のデータはファイルに格納されます。たとえば、ファイルには血圧（BP）計からのセンサー データが格納されているとします。各レコードは、以下のデータから成ります。

- 8 バイト タイムスタンプ - 血圧測定値の採取日時
- 2 バイト整数 - 収縮期の値
- 2 バイト整数 - 拡張期の値
- 1 バイト文字 - 評価コード

データの特性を考えると、13 バイトのレコードを保持するファイルを作成する必要があります。以下のスニペットは、血圧レコードを収容できる `Bprecord_t` 構造体を定義しています。`#pragma pack` ステートメントにより、レコードは実際に 13 バイトとなり、コンパイラによって余分なアライメント バイトが追加されない

ことが保証されます。

```
#pragma pack(1)
typedef struct {
  uint64_t timeTaken;
  uint16_t systolic;
  uint16_t diastolic;
  char EvalCode;
} BPrecord_t;
#pragma pack()
```

このようなレコード用のファイルを作成するには、btrieveFileAttributes オブジェクトを割り当て、SetFixedRecordLength プロパティを使用してレコード サイズを指定する必要があります。

```
Btrieve::StatusCode status;
BtrieveFileAttributes btrieveFileAttributes;
status = btrieveFileAttributes.SetFixedRecordLength(sizeof(BPrecord_t));
```

他のファイル属性（たとえば、ファイルをディスクに書き込む前に圧縮する属性）を追加することができますが、必須の属性はレコード サイズだけです。

次に、btrieveClient セッション オブジェクトで FileCreate() メソッドを使用してデータ ファイルを作成するように、Zen エンジンに指示する必要があります。また、ファイルが既存である場合に、上書きするかどうかを示す作成モードを指定することもできます。

```
static char* btrieveFileName = (char*)"Pressures.btr";
status = btrieveClient->FileCreate(&btrieveFileAttributes, btrieveFileName,
   Btrieve::CREATE_MODE_NO_OVERWRITE);
if ((status != Btrieve::STATUS_CODE_NO_ERROR) &
   (status != Btrieve::STATUS_CODE_FILE_ALREADY_EXISTS))
{
printf("Error: BtrieveClient::FileCreate():%d:%s.¥n", status,
   Btrieve::StatusCodeToString(status));
}
```

上の例には、Btrieve クラスの組み込み StatusCode 列挙を使用したエラー チェックが含まれています。簡潔にするため、後続のコード サンプルではエラー チェックを示しません。

ファイルにデータを挿入するには、まずそのファイルを開く必要があります。

```
BtrieveFile btrieveFile;
status = btrieveClient->FileOpen(btrieveFile, btrieveFileName, NULL,
   Btrieve::OPEN_MODE_NORMAL);
```

上のコードでは、btrieveFile オブジェクトを割り当て、btrieveClient セッションを使用して、以前に作成したファイルを開いています。3 番目の status パラメーター（この例では "NULL"）を使用して Btrieve オーナー ネームを渡すことができます。オーナー ネームは、データ ファイルをセキュリティ保護（および、任意で暗号化）するためのパスワードとして機能します。このファイルはオーナー ネームを持っていないため、NULL を入力値として渡します。

レコードを挿入するには、前に作成した Bprecord_t 構造体用のレコード バッファーを割り当て、このレコード構造体にデータ値を入れます。その後、RecordCreate メソッドを使用してレコードを挿入します。

```
Btrieve::StatusCode status = Btrieve::STATUS_CODE_NO_ERROR;
BPrecord_t record;
  // 現在のシステム時間を取得し、ミリ秒数に変換します
time_t now = time(0) * 1000000;
  // 時間を Btrieve 2 Timestamp 形式に変換します
record.timeTaken = Btrieve::UnixEpochMicrosecondsToTimestamp(now);
  // sysdata と diasdata は実行時に入力します
record.systolic = sysdata;
record.diastolic = diasdata;
  // systolic と diastolic から EvalCode を決定します
record.EvalCode = 'N'; // デフォルトは "N"（Normal：標準）です
if ((sysdata >= 120 and sysdata < 130) and (diasdata < 80))
    record.EvalCode = 'E'; // 血圧上昇
if ((sysdata >= 130 and sysdata < 140) or
  (diasdata >= 80 and diasdata < 90))
    record.EvalCode = 'H'; // 高血圧
if ((sysdata >= 140 and sysdata <= 180) or
  (diasdata >= 90 and diasdata <= 120))
    record.EvalCode = 'V'; // 超高血圧
if ((sysdata > 180) or (diasdata > 120))
    record.EvalCode = 'C'; // 高血圧性クリーゼ
  // レコードを挿入します
status = btrieveFile->RecordCreate((char*)& record, sizeof(record));
```

これらすべてのアクティビティは、標準の SQL API よりも Btrieve 2 API を使用した方がはるかに高速に実行され、SQLite が端緒すら実現できなかった IoT やモバイルのシナリオで一定水準のパフォーマンスを達成します。

最先端のエッジ データ管理で、データベースがファイル管理システムより非常に有利なのは、インデックスを作成してデータを迅速に取得できる機能があるためです。既存の Zen データ ファイルにインデックスを追加することで、特定の値によって、または特定の順序で迅速にレコードを取得できます。また、新しく作成したファイルに、レコードが挿入される前にインデックスを追加することもできます。

インデックス作成は次の 3 つの簡単なステップから成ります。

1. インデックス セグメントをセットアップする
2. インデックス属性を定義する
3. データ ファイルにインデックスを追加する

引き続きサンプル ファイルを使用して、レコードのタイムスタンプ部分にインデックスを追加します。ファイルを開いておかないとインデックスを追加できないので注意してください。

```
BtrieveKeySegment btrieveKeySegment;
BtrieveIndexAttributes btrieveIndexAttributes;
  // レコードの先頭 8 バイトにタイムスタンプのインデックス セグメントを作成します
status = btrieveKeySegment.SetField(0, 8, Btrieve::DATA_TYPE_TIMESTAMP);
  // Index オブジェクトにセグメントを追加します
status = btrieveIndexAttributes.AddKeySegment(&btrieveKeySegment);
  // 変更不可能なインデックス属性を指定します
status = btrieveIndexAttributes.SetModifiable(false);
  // インデックスを作成します
status = btrieveFile->IndexCreate(&btrieveIndexAttributes);
```

これで、インデックス セグメントが、レコードのオフセット 0 から始まる 8 バイトのフィールドとして定義されました。その 8 バイト内のデータは、Btrieve の TIMESTAMP として解釈されます。上記のサンプルで実行されていることを次に説明します。

- `AddKeySegment()` メソッドは、`btrieveKeySegment` インスタンスを `btrieveIndexAttributes` オブジェクトに追加して、単一セグメント キーを定義します。
- `SetModifiable()` メソッドは、インデックスを変更可能（true）または変更不可能（false）として指定します。他の属性を使用して、インデックスを一意にしたり、特定のインデックス番号を指定したりすることができます。
- `IndexCreate()` メソッドは、以前に開いた、`BtrieveFile` オブジェクトに関連付けられているデータ ファイルにインデックス 0 を追加します。インデックスは、現在ファイルにあるすべての値を伴って作成され、後続のすべての挿入/更新/削除操作で自動的に更新されます。

インデックスを作成した後、レコードを高速で取得するのは簡単です。インデックス定義に従って先頭または末尾のレコードを取得したり、指定された値と比較して特定のレコードを取得したりするためのメソッドがあります。この例では、タイムスタンプ値が最も高いレコード（挿入したばかりのレコードに相当する）を取得しています。

```
Btrieve::StatusCode status = Btrieve::STATUS_CODE_NO_ERROR;
BPrecord_t record;
  // 直近に挿入したレコードを取得します
if (btrieveFile->RecordRetrieveLast(Btrieve::INDEX_1, (char*)& record,
   sizeof(record), Btrieve::LOCK_MODE_NONE) != sizeof(record))
{
```

```
status = btrieveFile->GetLastStatusCode();

printf("Error: BtrieveFile::RecordRetrieve():%d:%s.¥n", status,
   Btrieve::StatusCodeToString(status));

}
```

メモ:レコード取得メソッドは、これまで見てきた他の呼び出しのようなステータス コードを返しません。代わりに、取得されたレコードの**サイズ**が関数呼び出しによって返されます。サイズが期待どおりに返されない場合は、`GetLastStatusCode()` メソッドを呼び出して、何が起こったのかを調べることができます。

レコード取得メソッドの最後の引数は、取得中にレコードをロックするオプションを提供します。

レコードを取得したら、それを更新するか削除するかを決定できます。まずレコードを取得しなければ、レコードを更新/削除することはできません。これらの操作には、`btrieveFile->RecordUpdate()` メソッドおよび `btrieveFile->RecordDelete()` メソッドが使用されます。

ファイルの作業が完了したら、`btrieveClient` オブジェクトで `FileClose()` メソッドを呼び出して、ファイルを閉じます。

```
status = btrieveClient->FileClose(btrieveFile));
```

あまり必要としないでしょうが、Btrieve 2 もデータ ファイルを削除するためのメソッドを提供しています。

```
status = btrieveClient->FileDelete(btrieveFileName);
```

アプリケーションを閉じる前に `Reset()` メソッドを呼び出して、エンジンとのセッションを解放する必要があります。

```
status = btrieveClient->Reset();
```

これで、基本的な CRUD(作成、読み取り、更新、削除)機能についての説明は終了です。これらの機能が非常に単純明快であることをご理解いただけたと思います。

株式会社エージーテック
本社:〒101-0054 東京都千代田区神田錦町1-21-1 ヒューリック神田橋ビル3F
TEL:03-3293-5300 (代表) FAX:03-3293-5270
カスタマーセンター TEL:03-3293-5283 MAIL:info@agtech.co.jp