

メインメモリデータベース vs. RAM ディスクデータベース： Linux 上でのベンチマークテスト

概要：メインメモリからデータをアクセスすれば、物理メディアからアクセスするよりも速いというのは明白です。新しいタイプのデータ・マネージメントシステム、メインメモリ・データベース(MMDB)はメモリ上のみの処理により飛躍的なパフォーマンスおよび使い易さをアピールします。しかしながら、データベース・キャッシングは同様のパフォーマンスを実現しないのでしょうか？また単にディスクアクセスの排除のみがゴールであれば、ファイルシステムをメモリ上に作成する RAM ディスクの上に従来のデータベースを実装すれば良いはずです。

本資料はオンメモリデータベースの eXtremeDB と db.linux 組込みデータベースを従来のディスクベースで実装したものと RAM ディスク上に実装したものを Red Hat Linux 6.2 上で動作し比較した際の考察を記述しております。db.linux を RAM 上に実装するとパフォーマンスは 74% 向上します。それでも従来のデータベースは MMDB よりは速くありません。パフォーマンスの差の原因は基本的な構造の違いにあります。ディスクに強く結び付けられたデータベースが抱えるオーバーヘッドにはデータの転送・重複、不要なりカバリ機能および皮肉にもディスクアクセスを回避するためのキャッシング・ロジックが含まれます。

ガイロジック株式会社

<http://www.gailogic.co.jp/db>

db@gailogic.co.jp

Copyright 2001, McObject LLC

はじめに

ディスクからデータを取り出すのではなく、データをメモリ上に保持していればアプリケーションのパフォーマンスが向上することは道理にかないます。ディスクアクセスは部品の実運動により遅延が伴う処理上の数少ないメカニカルな過程です。ソフトウェアの観点から見ても、ディスクアクセスにはパフォーマンスを著しく低下させる「システムコール」が伴います。データマネジメントシステム(DBMS)キャッシングおよびファイルシステム・キャッシングを行う上ではディスクアクセスを回避してパフォーマンスを向上したいと思うことは基本原則となっております。

すべてメモリ上に常駐するよう設計された新しいタイプのDBMSによってこの概念は最近拡張されています。メインメモリデータベース(MMDBs)の支持者はデータベースの速度と使い易さの革新的な改善を指摘し、メインメモリデータベースの手法はデータマネジメントおよびリアルタイムシステムの両分野にとって重要な一歩であると主張しています。

しかしながら、ここで明らかに解決しなければならない疑問が残ります。その疑問とは、キャッシュが利用可能ならば、データベースをすべてキャッシュに実装すればパフォーマンスが向上するのではないかということです。加えて、RAMドライブユーティリティはメモリにファイルシステムを構築するために存在します。確かに従来のデータベースをRAMディスクに実装すれば物理ディスクへのアクセスはなくなります。それでは、RAMディスクに実装された従来のデータベースのパフォーマンスはメインメモリデータベースのパフォーマンスと同様になるのでしょうか。

本資料ではこの理論が正しいかを検証します。検証方法として、30,000件のレコードのリードおよびライトに関するパフォーマンス計測用に2つの同様なデータベース構造およびアプリケーションを開発しました。テスト対象の2つのデータベースの主な違いは片方がメインメモリデータベースであるeXtremeDBで他方がディスクストレージを使用するように設計されているdb.linuxであるという点です。結果はRAMドライブに実装されたデータベースがディスク使用のデータベースよりも格段に速くなっても、メインメモリデータベースのパフォーマンスに到達することはありませんでした。以下において、ディスクベースのデータベース(RAMドライブに実装したものも含む)固有の各オーバーヘッド(キャッシング、データ転送など)がどのようにしてメインメモリデータベースのパフォーマンスとの差異の原因となっているかについて説明します。

メインメモリデータベースの出現

メインメモリデータベースはデータベースマネジメントの中では比較的新しい分野のデータベースです。この技術は当初ビジネスアプリケーションのパフォーマンスを向上するため、およびトラフィックピーク時に Web コマーシャルサイトをキャッシュするために生まれました。このようなエンタープライズ向けのものとして MMDB は従来の SQL/関係データベースと同様のものから、一部機能をメインメモリに実装したもの、すべてをメインメモリに実装したものと進化しました。

次にデータベース技術がターゲットとしたのは、組込みシステム開発でした。ネットワークスイッチ、ルータ、セットトップボックスその他の民生機器の開発者たちは徐々に商用のデータベースに新たな機能を追加していきました。必要とされるリアルタイム性能への対応をはじめ RAM や CPU 消費量の著しい低減、組込みシステム開発者が嗜好する言語である C/C++/Java などの第三代言語への対応などにより、メインメモリデータベースは組込み機器の市場セグメントに現れました。

eXtremeDB vs. db.linux の比較

McObject 社の eXtremeDB は組込みシステム市場向けに最初に作られたメインメモリデータベースです。アプリケーション内部からデータベースマネジメント機能を実現するために開発者が利用するように作られたものとしては、この eXtremeDB も db.linux、BerkleyDB、Empress、C-tree やその他のディスクベースのデータベースと同様です。マイクロソフト社の SQL サーバ、DB2 やオラクルなどのアプリケーションとは隔離して管理されているサーバ形式のものとは異なり、これらのデータベースはアプリケーションに「組み込まれて」いるのです。これらの各データベースはエンタープライズ向けのものに比べメモリフットプリントが小さく、またデータベース操作を正確に行うためのナビゲーション API を提供します。

本資料では eXtremeDB とディスクベースの組込み用データベースとして db.linux を比較しています。オープンソースである db.linux DBMS を選択したのは db.linux が 1986 年に db_VISTA という名前で最初にリリースされて以来長年の実績があることと、多くの人に使われているからです。また eXtremeDB と db.linux は同様のデータベース定義言語を持っていることも理由の 1 つです。

ベンチマークテストは Intel 社 Celeron 400MHz、RAM128MB 搭載の PC に Red Hat Linux6.2 を実装した環境で行いました。

データベースデザイン

同様なオブジェクト 30,000 件をデータベースにライトし、キー1つを通じてリードすることにより 2つのデータベースを比較するために、以下の簡単なデータベーススキーマは開発されました。

```

/*****
 *
 * Copyright (c) 2001 McObject, LLC. All Right Reserved.
 *
 *****/

#define int1      signed<1>
#define int2      signed<2>
#define int4      signed<4>
#define uint4     unsigned<4>
#define uint2     unsigned<2>
#define uint1     unsigned<1>

declare database mcs[1000000];

struct stuff {
    int2 a;
};
/*
 *
 */
class Measure {
    uint4 sensor_id;
    uint4 timestamp;
    string spectra;

    stuff thing;

    tree <sensor_id, timestamp> sensors;
};

```

図 1. eXtremeDB スキーマ

```

/*****
 *
 * Copyright (c) 2001 McObject, LLC. All Right Reserved.
 *
 *****/

struct stuff {
    short a;
};

database mcs [8192]
{
    data file "mcs.dat" contains Measure;
    key file "mcs.key" contains sensors;

    record Measure
    {
        long sensor_id;
        long m_timestamp;
        char spectra[1000];

        struct stuff thing;

        compound key sensors {
            sensor_id;
            m_timestamp;
        }
    }
}

```

図 2. db.linux スキーマ

2つのスキーマ間の差異で唯一意味があるものは'spectra'フィールドです。eXtremeDBの場合、'spectra'フィールドは'string'型として定義され、db.linuxではchar[1000]として定義されています。db.linuxの実装例では

実際にフィールドに何バイト格納されているかに関わらず、'spectra'フィールドに1000バイト消費されます。eXtremeDBではstringは可変長のフィールドです。db.linuxはeXtremeDBのstring型に直接相当するようなものはありませんが、db.linuxのネットワークモデルセットを使い、粒状度（パフォーマンスとスペース効率のトレードオフ）を調整することで可変長フィールドのエミュレートを行うことは可能です。このことは場合によっては大きな差異の原因となり同じ条件での比較を困難にする可能性があります。eXtremeDBには固定長キャラクターデータ型があります。敢えて可変長フィールドを使用したのは、可変長データ型こそまさにこの種のタスクのために設計されたものだからです。

(興味深い実験としては eXtremeDB を spectra で char[1000] を使うように、db.linux が可変長フィールドを使うように変更することが挙げられます。この実装を行うためのソースコードは Appendix A に記載しております。)
ベンチマーク アプリケーション

テストアプリケーション・プログラムの前半ではデータベースに 'Measure' クラス/レコードのインスタンス 30,000 件を入力します。

eXtremeDB がデータベース用にまたランダム文字列にメモリを割り当てます。またデータベースを開き、コネクションを確立します。

```
void *start_mem = malloc( DBSIZE );

if ( !start_mem ) {
    printf( "\nToo bad ..." );
    exit( 1 );
}

make_strings();

rc = mco_db_open( dbName, mcs_get_dictionary(), start_mem,
                 DBSIZE, (uint2) PAGESIZE );

if ( rc ) {
    printf( "\nerror creating database" );
    exit( 1 );
}

/* connect to the database, obtain a database handle */
mco_db_connect( dbName, &db );
```

図 3. eXtremeDB スタートアップ実装例

db.linux はランダム文字列用にメモリを割り当て、構造体 DB_TASK の初期化を行い、マルチスレッドアクセスを可能にする "s" シェアードモードでデータベースを開き、データ保全性確保のためにトランザクションを要求します。

```
make_strings();

stat = d_opentask(&task);

if((stat = d_dbuserid("rdmtest", &task))) return;

if((stat = d_open("mcs", "s", &task))) return;
```

図 4. db.linux スタートアップ実装例

ここから両方の実装例は 2 つのループに入ります。外側のループを 100 回繰り返し、内側のループを 300 回繰り返します。(合計 30,000 回)

eXtremeDB にレコードを追加するためにトランザクションが開始され、データベース中の新しいオブジェクト(Measure_new)用にスペースが用意されます。それから、sensor_id および timestamp フィールドがオブジェクトに置かれ、ランダム文字列が以前に生成されていた文字列のプールより抽出されオブジェクトに置かれます。そしてトランザクションを終了します。

```
for ( sensor_num = 0; sensor_num < SENSORS; sensor_num++ ) {
    for ( measure_num = 0; measure_num < MEASURES; measure_num++ ) {
        mco_trans_start(db, MCO_READ_WRITE, MCO_TRANS_FOREGROUND, &t);
        rc = Measure_new( t, &measure );
        if ( MCO_S_OK == rc ) {
            Measure_sensor_id_put(&measure, (uint4) sensor_num );
            Measure_timestamp_put(&measure, sensor_num + measure_num );
            get_random_string( str );
            Measure_spectra_put( &measure, str, (uint2) strlen(str) );
            rc = mco_trans_commit( t );
            if ( rc != 0 )
                goto repl;
        }
        else {
            mco_trans_rollback( t );
            printf( "\n\n\tOops, error allocating object: %d\n", rc );
            goto repl;
        }
    }
    putchar( \'.\' );
}
```

図 5. eXtremeDB ‘write’の実装例

db.linux の実装例にてトランザクションが開始しています。ライトロックが必要になります。次のコードでは source_id と timestamp の値をローカル・ストラクチャに割り当てています。さらに既に生成されている文字列のプールよりランダム文字列をコピーし、データベースにレコードをライトし(d_fillnew)トランザクションを終了します。

```

for( sensor_num = 0; sensor_num < NSENSORS; sensor_num++ ) {
    for( measure_num = 0; measure_num < NMEASURES; measure_num++ ) {
        if((stat = d_trbegin( "tid", &task )))
            break;
        if((stat = d_reclock(MEASURE, "w", &task, CURR_DB)))
            break;
        mr.sensor_id = sensor_num;
        mr.m_timestamp = measure_num + sensor_num;
        get_random_string( &mr.spectra[0] );
        if((stat = d_fillnew( MEASURE, &mr, &task, CURR_DB )))
            break;
        if( stat == S_OKAY ) {
            if((stat = d_trend( &task )))
                break;
            } else if((stat = d_trabort( &task ))) {
                break;
            }
            putchar('.');
        }
    }
}

```

図 6. db.linux ‘write’ 実装例

eXtremeDB はマルチスレッド・データベースであるので、リード・アクセスを含むすべてのデータベース操作は 1 トランザクションの範囲で実行される必要があります。したがって、データベースを開いているときにはオープンモードを特定する必要はありません。対照的に db.linux はシングルユーザ（ワンユーザ）・モードとマルチユーザ・モードが分かれています。db.linux のワンユーザ・モードでは、トランザクションは選択となりますが、マルチ・ユーザ・キャッシュの一貫性を保証するためにマルチユーザ・モードではトランザクションは必須となります。

次の 1 組のネストしたループは事前に作られた 30,000 件のオブジェクトのリード時のパフォーマンスを測定するために用意されています。

```

for ( sensor_num = 0; sensor_num < SENSORS; sensor_num++ ) {
    uint2 len;
    for ( measure_num = 0; measure_num < MEASURES; measure_num++ ) {
        mco_trans_start(db, MCO_READ_ONLY, MCO_TRANS_FOREGROUND, &t );
        rc = Measure_sensors_index_cursor( t, &csr );
        rc = Measure_sensors_find( t, &csr, MCO_EQ, sensor_num,
                                sensor_num + measure_num);

        if ( rc != 0 ) {
            rc = mco_trans_commit( t );
            goto rep2;
        }
        rc = Measure_from_cursor( t, &csr, &measure );
        /* read the spectra */
        rc = Measure_spectra_get( &measure, str, sizeof(str), &len );
        rc = Measure_sensor_id_get( &measure, &id );
        rc = Measure_timestamp_get( &measure, &ts );
        rc = mco_trans_commit( t );
    }
}

```

図 7. eXtremeDB ‘read’ 実装例

eXtremeDB の実装例では毎回のループでリードトランザクションを開始し、カーソルをインスタンス化してキー・フィールドを使用して **Measure** オブジェクトを見つけ出します。オブジェクトを見つけると、カーソルによりオブジェクト・ハンドルが初期化され、オブジェクトハンドルによりオブジェクトのフィールドがリードされます。そしてトランザクションが終了します。

```

for( sensor_num = 0; sensor_num < NSENSORS; sensor_num++ ) {
    for( measure_num = 0; measure_num < NMEASURES; measure_num++ ) {
        mr.sensor_id = sensor_num;
        mr.m_timestamp = measure_num + sensor_num;
        if((stat = d_relock(MEASURE, "r", &task, CURR_DB)))
            break;
        if((stat = d_keyfind( SENSORS, &mr, &task, CURR_DB )))
            break;
        if((stat = d_recread( &mr, &task, CURR_DB )))
            break;
        if((stat = d_refree(MEASURE, &task, CURR_DB)))
            break;
    }
}

```

図 8. db.linux ‘read’ 実装例

db.linux の実装例ではループが 2 つ用意されています。ループを 1 階繰り返すたびにキー検索用の値が構造体のフィールドに割り当てられます。db.linux ではリードオンリのアクセスではトランザクションは必要ありませんが、レコードの型

は明示的にロックされる必要があります。レコードのロックができると、キー参照用の値を保持している構造体は `d_keyfind` 関数に渡されます。キーの値が見つかると、レコードは `d_recread` により同じ構造体に読み込まれ、レコードのロックは解除されます。

上記から察せられるように、本文書ではキーの実装方法の違いはトランザクションおよびマルチユーザ（マルチスレッド）並行アクセスを中心に展開されています。（オブジェクト指向アプローチと `eXtremeDB` のデータベースアクセスなど哲学的な差異も存在しますが、オンメモリ・データベースとディスクベースのデータベースとの差異に関係する差異ではないので、ここでは無視します。）

`eXtremeDB` では並行処理はすべて非明示的に行われます。並行処理に必要なのは、データベースに対するアクセスはすべて1つのリードもしくはライトトランザクションの範囲で発生する必要があります。対照的に `db.linux` ではアプリケーションがリードもしくはライトのレコード型ロックを獲得することが必要となります。またこのことは必要に応じて、レコード型へのアクセスを試みる前に行う場合があります。`db.linux` は明示的なロックが必要なので、リードオンリのアクセスではトランザクションは不要です。

下記のグラフは `eXtremeDB` と `db.linux` のパフォーマンスを相対的に図示したものです。各データベースはマルチスレッド、トランザクションによる制御の環境で動作しています。`db.linux` では通常使用される環境と同様にデータベースファイルはディスクに保存されています。

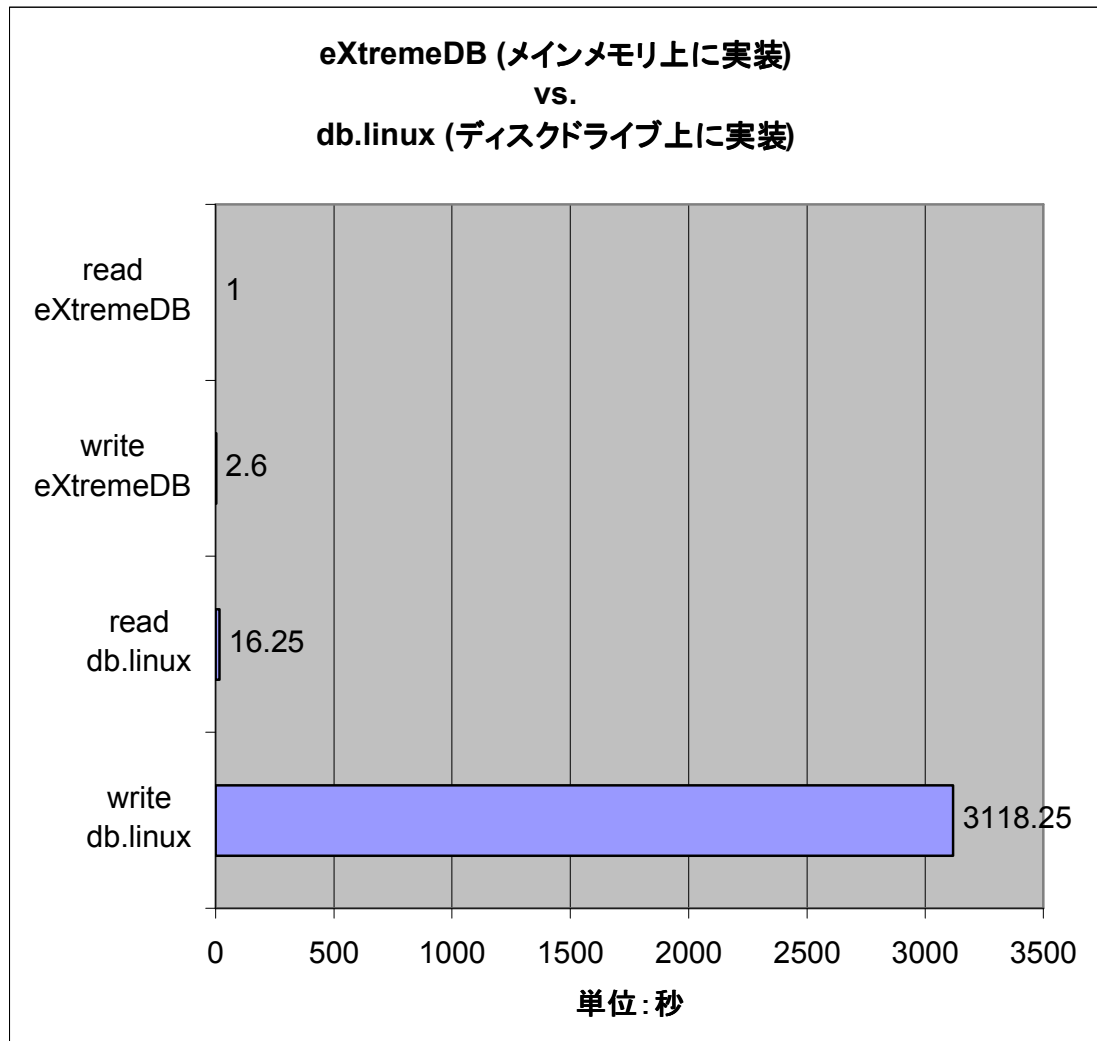


図 9. eXtremeDB とディスクベースのデータベース

メインメモリ上で処理していることが eXtremeDB の劇的なパフォーマンスを演出していることは明白です。それでは、RAM ディスクを使うことにより db.linux のパフォーマンスはオン・メモリ・データベースのパフォーマンスに比肩するまでになるでしょうか？

図 10 は上記と同様の方法で実装されている eXtremeDB と物理的なディスクアクセスを排除した RAM ディスクベースの db.linux のパフォーマンスを表したものです。(Red Hat Linux 6.2 上に実装した RAM ディスクの実装に関する詳細は Appendix B をご参照下さい。)

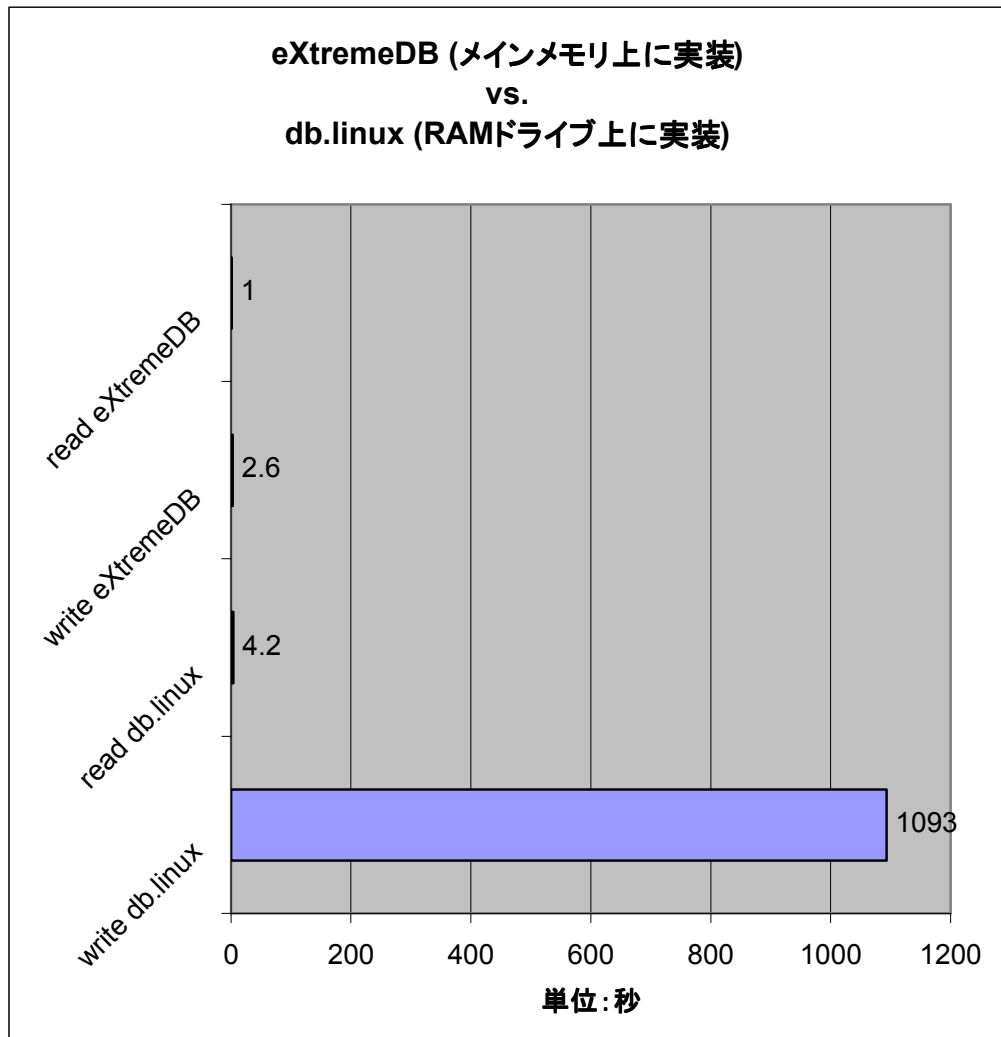


図 10. eXtremeDB と RAM ディスク上に実装したデータベース

図 10 により、RAM ディスク上に db.linix を実装することにより、db.linix のパフォーマンスがリードアクセスで約 4 倍、ライトアクセスで約 3 倍向上しているのがわかります。ディスクベースのデータベースファイルを RAM ディスクに移し替えばパフォーマンスが向上することはあきらかです。

しかしながら、当初よりメモリ上に実装される前提で設計されたデータベースの方がさらに優れたパフォーマンスを実現していることもまた上記同様あきらかです。メインメモリに実装されるデータベースは RAM ディスクベースのデータベースに比べ、ライトアクセスで 420 倍、リードアクセスで 4 倍以上のパフォーマンスをあげています。以下のセクションではこの両データベースのパフォーマンスの差異の原因を探って行きます。

分析ーオーバーヘッドの原因はどこにある？

RAM ディスク・アプローチにより物理的なディスクアクセスはなくなりました。それでも、ディスクベースのデータベースがオン・メモリ・データベースよりも遅いのはなぜでしょうか？この原因はディスクベースのデータベースにはメインメモリ処理には関係のない処理が伴うことにあります。また、このことは実装する場をRAMに変えても、内部的な機能に変わりがない事を表します。このような処理はもはや不要になった動作を前提として行われており、パフォーマンス上のオーバーヘッドを増大する結果になります。

キャッシング・オーバーヘッド

物理的にディスクにアクセスすることはパフォーマンスの重大な低下をともなうので、事実上すべてのディスクベースのデータベースはディスクアクセスを最小化するための技術を採用しています。このような技術の代表的なものとしては、データベース・キャッシングがあります。データベース・キャッシングではデータベース中最も頻繁に使用される部分をメモリに保持しようと最大限の努力が払われています。キャッシングのロジックにはキャッシュ同期化が含まれます。キャッシュ同期化とは、キャッシュ中のデータベースページのイメージがディスク上の物理データベースページと矛盾しないことを保証するものであり、アプリケーションが無効なデータをリードすることを防止します。

ディスクアクセスを最小化するための他の処理としては、キャッシュ照合があります。キャッシュ照合では、アプリケーションから要求されたデータがキャッシュ上にあるかないかを判断し、ない場合はディスクから取り出し、将来参照するために、そのデータをキャッシュ上に置きます。キャッシュ照合では、新しいページをキャッシュ上に置くために、キャッシュ上から退避させるデータの決定も行います。退避するデータが「ダーティ（汚れている＝更新したレコードを伴う）」な場合は、他のロジックが起動し、トランザクションが終了するまで他のアプリケーションによってダーティなデータが参照されないようにします。

1回のキャッシングによるオーバーヘッドは微々たるものですが、その合計となると、深刻な量となります。アプリケーションがディスクからレコードをリードするファンクション・コールを行うたびに上記のここの処理が伴います。

（db.linux の場合は `d_recfirst`, `d_recnext`, `d_findnm`, `d_keyfind` などが相当するファンクションコールになります。）上記のデモンストレーション・アプリケーションでは 90,000 回のファンクションコールが発生します。（このうち、`d_fillnew`, `d_keyfind`, `d_recread` とともに 30,000 回発生）対照的に eXtremeDB のようなオン・

メモリ・データベースのすべてのレコードは常にメモリ上にあるので、キャッシング動作は伴いません。

トランザクション処理オーバーヘッド

トランザクション処理ロジックは処理レイテンシの主要因です。電力の減少など悲劇的な状況では、ディスクベースのデータベースはシステムが再起動する際に、1つもしくは複数のログファイルから一部もしくは完全なトランザクションをコミットするかロールバックします。ディスクベースのデータベースは不可避免的にトランザクションログを記録し、トランザクション・ログファイルをフラッシュしトランザクションのコミット後にキャッシュをディスクに退避します。ディスクベースのデータベースには自身が RAM 上で実行されていることがわかりません。ログファイルがメモリ上のみに存在し、システムダウンが起きた際のリカバリには何の助けにならなくても、このような複雑な処理は変わらず行われます。

オン・メモリ・データベースはトランザクションの保全性もしくは、いわゆる ACID 準拠のトランザクションを提供する必要があります。オン・メモリ・データベースのアプリケーション・スレッドは一連の複数の更新処理を1つのユニットとしてコミットもしくはアボートできる必要があります。これを実現するために eXtremeDB は更新もしくは削除前のオブジェクトのイメージおよびトランザクション中に加わったデータベース・ページのリストを保持します。アプリケーションがトランザクションを終了（コミット）した際には更新前イメージのメモリおよびページ・リファレンスはメモリプールに戻ります（この処理は高速かつ効率的に行われます）。オン・メモリ・データベースがトランザクションをアボートしなければならないとき、例えばデータストリームの流入が割込まれたとき更新前のデータイメージはデータベースに戻され、新たに挿入されたページはメモリに戻されます。

悲劇的な状況が起きた場合—オン・メモリ・データベースのイメージは損なわれます。しかし、これこそ MMDB 向きのアプリケーションと言えることができます。システムが OFF のとき、もしくはなんらかの状況でオン・メモリ・イメージが無効になってしまったとき、再起動するとデータベースは単純に初期状態に戻ります。この例としては、断続的に衛星もしくはケーブルからセットトップ・ボックスにダウンロードされる番組ガイドアプリケーションや起動時にネットワーク上よりトポロジを見つけるネットワークスイッチ、アップストリームのサーバーにより初期化される無線アクセスポイントなどがあげられます。

このことは保存されているローカル・データが使えないということではありません。アプリケーションはストリーム（ソケット、パイプ、ファイルポインタなど）をオープンすることができ、また **eXtremeDB** に対してデータベース・イメージをストリームからリードするまたはストリームへライトするよう指示することができます。この機能はブート段階のデータの作成および維持に使うことができます。例えば、データベースの起動時などのケースがこれにあたります。ストリームのもう 1 つの末端として、他の処理へのパイプやファイル・システム・ポインタがあげられます（磁気、光学デバイス、フラッシュなどメディアは問いません）。しかし、**eXtremeDB** のトランザクション処理はこれらの機能とは独立して動作します。このことは最大限のデータ可用性を実現するために主記憶上の処理を主眼に置いているためです。

データ転送オーバーヘッド

ディスクベースのデータベースではデータは頻繁に転送され、コピーされます。実際に、アプリケーションはデータベースから幾度か削除されているデータのコピーとともに動作します。またこのコピーはプログラム変数に格納されています。アプリケーションがディスクベースのデータベースからデータをリードする際には、データの変更、データベースへのデータのライトバックなどの処理を伴います。

1. アプリケーションがデータベースの API を通じてデータベース・ランタイムに対してデータ・アイテム要求を出します（例：db.linux の `recread` 関数）。
2. データベース・ランタイムがファイルシステムに対して物理メディア（RAM ディスクの場合はメモリ上のデータ格納場所を指します。）からデータの取り出しを指示します。
3. ファイルシステムがデータのコピーを作成します。1 つのコピーはキャッシュ用に作られ、別にもう 1 つのコピーをデータベースに渡します。

4. データベースは自身のキャッシュにコピーを保持し、アプリケーションにもう1つのコピーを渡します。
5. アプリケーションはコピーに変更を施し、変更後のコピーをデータベース API（例：db.linux の `rewrite` 関数）を通じてデータベースに返します。
6. データベース・ランタイムが変更後のデータをコピーし、データベースキャッシュに渡します。
7. データベースキャッシュ中のコピーはファイルシステム・キャッシュにて更新され、その後ファイルシステムにライトされます。
8. 最後にデータは物理メディア（もしくはRAMディスク）にライトバックされます。

この例では、データのコピーの数は4（アプリケーション・コピー、データベース・キャッシュ、ファイルシステム・キャッシュ、ファイルシステム）、ファイルシステムからアプリケーション、またアプリケーションからファイルシステムの間の転送は6件でした。なお、この単純な例ではトランザクションのログ用に必要となる可能性があるコピーおよび転送の数は含まれていません。

対照的に、eXtremeDBのようなオン・メモリ・データベースではほとんどデータの転送を伴いません。アプリケーションが自身の用途のためにローカルのプログラム変数にデータのコピーを作成することは可能ですが、eXtremeDBはデータのコピーを必要としていません。そのかわり、eXtremeDBはアプリケーションに対してデータベース中のデータ・アイテムを直接指すポインタを参照用に渡します。このことにより、アプリケーションはデータと直接動作することができます。このような仕組みであっても、データはプロテクトされています。使用されるポインタはeXtremeDBのAPIだけを通じて使用することができるものなので、ポインタの使われ方は適正なものだけに限定されます。

OSに対する依存性

RAM ディスクベースのデータベースでは依然として、データベース中のデータへのアクセス用途としてデータベースの下位層に実装したファイルシステムを使用しています。したがって、RAM ディスクのデータベースはデータの格納位置を確定する際、`lseek()` のようなファイルシステムの関数に依存しています。`lseek()` (RAM ディスクをはじめ、いかなるディスク・ファイルシステムにおいても) の実装次第でデータベースのパフォーマンスも左右されますが、DBMS 自体はこのパフォーマンス決定要因に対して何の制御もすることができません。

対照的に、**eXtremeDB** はアクセスメソッドをコントロールし、最適化することが可能です。

`db.linux` は特にプロセス間通信(IPC)の点で OS への依存度が高くなっています。この場合 IPC が必要となるのは、並行アクセスの同期化を行う際および、1 つもしくは複数のクライアントの障害もしくはロック・マネージャ自身の障害からトランザクション・ログの回復を行う際です。IPC の実装状況により `db.linux` のパフォーマンスは左右されますが、IPC が最も良く実装された場合においても、重大なプロセス・オーバーヘッドが発生します。他の組込み用データベースでも `db.linux` 同様に IPC に依存したものもあります。

結論

本ホワイトペーパーによりあきらかになった点は主に以下の 2 点です。

- ディスクベースの DBMS を RAM ディスクに実装するとパフォーマンスが向上する。
- しかしながら、同じアプリケーションタスクおよび実行環境の上では、RAM ディスクに実装した DBMS でさえ、オン・メモリ・データベースのパフォーマンスに比べ重大な遅延を生じている。

このパフォーマンスの際に関する原因はオン・メモリか従来のデータベースかという構造上の問題に集約されます。皮肉にも、RAM ディスク上に実装した場合でさえ、ディスクベースのデータベースが遅いのはディスクアクセスを回避するために実装されたロジックでした。このロジックも RAM ディスク上では効果はありませんが、ディスク上と同様に動作します。障害からの回復するための技術など伝統的なデータベースの技術もメモリのみの実行環境では不要ですが、パフ

パフォーマンスを向上する目的でその機能のみ動作を止めることはできません。オン・メモリ・データベースはすべてのアプリケーションに適しているとは限りませんが、高速なデータ・アクセスを必要とする場合は比類ない選択肢となります。

本ホワイトペーパーの主題ではありませんが、オン・メモリ・データベースを利用する上でのベネフィットが上記の他に2つあります。1つはメモリ・フットプリントです。キャッシング機能をはじめ不要なロジックを排除することにより、必要となるデータ格納用のストレージ・スペースは格段に小さくなります。実際に上記のテストにおいて、eXtremeDBのRAMフットプリントは108KBで、データをすべて格納したときでも20.85MBでした。（元データのサイズは16.7MBです。）一方でdb.linuxの方はRAMフットプリントは323KBですべてのデータを格納したときには31.8MBでした。（元データのサイズは上記と同様16.7MBです。）もう1つのベネフィットはシンプルな構造の採用による高い信頼性です。伴う処理が少ないスリムな設計により、高いパフォーマンスを実現しています。

db.linux を用いて可変長文字列フィールドのエミュレーションを行うにあたり、データベース・スキーマを以下のように変更。

```
/*
 * Copyright (c) 2001 McObject, LLC. All Right Reserved.
 */
*****/

struct stuff {
    short a;
};

database mcs
{
    data file "mcs.dat" contains Measure;
    key file "mcs.key" contains sensors;

    record Measure
    {
        long sensor_id;
        long m_timestamp;

        struct stuff thing;

        compound key sensors {
            sensor_id;
            m_timestamp;
        }
    }
    record Text100 {
        char spectral100[100];
    }
    record Text200 {
        char spectra200[200];
    }
    record Text300 {
        char spectra300[300];
    }
    set Spectra {
        order last;
        owner Measure;
        member Text100;
        member Text200;
        member Text300;
    }
}
```

データベースにデータを格納する際、以下のスードコードを使用。

```
d_fillnew (MEASURE)
d_setor(SPECTRA)
char *p = spectra
do {
    if strlen(p) >= sizeof_spectra300
        strncpy( Text300.spectra300, p, sizeof_spectra300 )
        d_fillnew the Text300 record
        d_connect(SPECTRA)
        p += sizeof_spectra300
    else if strlen(p) >= sizeof_spectra200
        strncpy( Text200.spectra200, p, sizeof_spectra200 )
        d_fillnew the Text200 record
        d_connect(SPECTRA)
        p += sizeof_spectra200
    else if strlen(p) >= sizeof_spectra100
        strncpy( Text100.spectra100, p, sizeof_spectra100 )
        d_fillnew the Text100 record
        d_connect(SPECTRA)
        p = NULL
} while (p)
```

注意：上記のスードコードは大幅に簡略化されています。スードコードの目的の概要は以下のとおりです。文字列 `spectra` の最も大きなものを分割して適切なサイズのレコード `TextNNN` に格納し、`db.linux` の `SPECTRA` という名前のマルチ・メンバ・ネットワークモデルを使ってこれらのレコードのリンクリストを作成します。

データを取り出す際にはリンクリストが網羅され、分割された文字列を再び1つにつなぎ合わせます。

```
d_keyfind (MEASURE)
d_recread (MEASURE)
d_setor(SPECTRA)
char spectra[1000];
for( (stat = d_findfm(SPECTRA)); stat != S_EOS; (stat =
d_findnm(SPECTRA)) {
    d_recread( &text300rec )
    strcat( spectra, text300rec.spectra300 )
}
```

文字列を再組み立てするコードはセット・メンバ・レコードをリードし、文字列を1つにつなぎ合わせながら反復します。繰り返しますが、このスードコードは可変長文字列の主なロジックを説明するために大幅に簡略化されています。

Appendix B – RAM-Disk の設定

OS: Red Hat Linux 6.2

RAM ディスク設定手順:

1. /etc/lilo.config file: に下記の 1 行を追加

```
ramdisk=38000
```

lilo.conf: の一例

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinuz-2.2.5-15
        label=linux
        root=/dev/hda6
        read-only
        ramdisk=38000
```

2. /sbin/lilo and reboot とタイプする
3. RAM ディスクのマウントポイントを作成。（下記はその一例）

```
mkdir /tmp/ramdisk0
```

このディレクトリへの適切なアクセス権を付与する。

4. ブロックデバイス上にファイルシステムを作成。

```
/sbin/mke2fs /dev/ram0
```

df -k /dev/ram0 を動作し使用可能なメモリサイズを把握する。(ファイルシステムも多少のスペースを使います。).

5. RAM ディスクをマウントする

```
mount /dev/ram0 /tmp/ramdisk0
```

準備完了。