Windows および Linux における SQL Anywhere の I/O 要件

28 March 2011

もくじ

1 はじめに 3

2 信頼できるストレージの必要性 3
2.1 SQL Anywhere におけるリカバリ 3
2.2 トランザクションの永続性. 4

3 ストレージ・スタック 5
3.1 概要 5
3.2 ディスク・ドライブ 5
3.2.1 Windows 6
3.2.2 Linux 6
3.3 ストレージ・コントローラ 7
3.3.1 Windows 7
3.3.2 Linux 7
3.4 ストレージ・ドライバ 8
3.4.1 Windows 8
3.4.2 Linux 9
3.5 ファイルシステム 10
3.5.1 Windows 11
3.5.2 Linux 11

4 システムの構成 12
4.1 Windows 12
4 . 1 . 1 12
4.2 Linux 13
4.2.1 デバイス名の特定 13
4 . 2 . 2 14

5 まとめ 14
5.1 Windows 15
5.2 Linux 15

1 はじめに

データベース・サーバは、コミットされたトランザクションを存続させるためにデータを確実に安定ストレージに到達させ、電源停止時にも適切にリカバリできなければなりません。オペレーティング・システムおよびストレージ・デバイスでは、パフォーマンスを高めるために書き込み操作がキャッシュされリオーダされますが、適切に構成されていないシステムでデータベース・サーバを実行すると、データが失われたりファイルが壊れたりする可能性があります。このドキュメントでは、永続的なストレージの要件と SQL Anywhere の I/O セマンティックを理解するために必要な予備知識を提供します。

2 信頼できるストレージの必要性

2.1 SQL Anywhere におけるリカバリ

データベース・サーバのクラッシュ、オペレーティング・システムのクラッシュ、あるいは電源停止による異常シャットダウンが発生した場合、SQL Anywhere はデータベース・ファイルを再び利用できるようにするためにリカバリを実行しなければなりません。このような場合、SQL Anywhere では 2 段階のリカバリ戦略が実施されます。第 1 段階では、データベース・ファイル内のページが最後のチェックポイント時点の状態に戻されます。第 2 段階では、最後のチェックポイント以降にコミットされたすべてのトランザクションが、トランザクション・ログに記録された操作に従ってリプレイされます。このリカバリが成功するか否かは、正常動作時のデータベース・サーバの動作にかかっています。

正常動作時には、最後のチェックポイント以降に初めてデータベース・ファイル内のページが変更されたときに、特殊なアクションが実行されます。すなわち、データベース・サーバは、何らかの変更を許可する前に、ページのコピーを保管する必要があります。この変更前のコピーは更新前イメージと呼ばれ、データベース・ファイル自体に保管されます。最後のチェックポイント以降に作成されたすべての更新前イメージの集合からチェックポイント・ログが形成され、チェックポイント・ログは、次のチェックポイントが完了した時点で破棄されます。この一連の処理はチェックポイントのたびに繰り返されます。データベース・サーバが異常終了した場合は、チェックポイント・ログによって、リカバリの第 1 段階に必要なメカニズムが提供されます。

SQL Anywhere では、変更されたページを更新前イメージより先にディスクに書き込むことはできません。このような処理を行うとリカバリできなくなる可能性があるからです。この要件を満たすために、データベース・サーバは、更新前イメージの書き込み後にオペレーティング・システムの *flush* 操作を発行し、*flush* 操作が完了するまで新たに変更されたページを書き込みません。SQL Anywhere の強固なリカバリ・スキームを確保するためには、オペレーティング・システムで提供される *flush* 操作において、*flush* より前に発行されたすべての書き込み操作が *flush* 操作の完了時には安定ストレージ上に存在することが保証されなければなりません。

この点についてわかりやすく説明するために、あるテーブルからローが削除され、そのテーブル・ページが更新された場合について考えます。この変更が更新前イメージより先にディスクに書き込まれ、その瞬間にオペレーティング・システムがクラッシュしたとします。リカバリ中、その他すべての更新前イメージが適用され、データベースのほとんどは元に戻りますが、

最後に更新されたページは戻りません。リカバリの際は、トランザクション・ログに記録されたすべての操作がリプレイ時に正常に実行されなければならないという要件があります。これは、その操作を記録した文が最初の実行時に成功しているからです。この例の場合、トランザクション・ログを適用すると、ローとインデックス・エントリを削除する delete 文になりますが、(ページはロールバックされているため) インデックス・エントリは存在しても、ローはテーブル・ページ上に存在しないため、この delete は失敗します。そして、このデータの矛盾した性質により、リカバリは失敗します。

データベースへの書き込みをディスクにフラッシュすることも重要ですが、特定の時点でファイル・メタデータをディスクにフラッシュすることも必要です。このファイル・メタデータ (すなわちファイルに関するファイルシステム情報) も物理的にディスクに保管されます。ファイル・メタデータには、ファイルが保管されているディスク上の場所やファイルのサイズといった情報が含まれます。SQL Anywhere は、安定メディアへのメタデータの保管が必要になった時点で *flush* 要求を発行します。

ここで、なぜファイル・メタデータも安定メディアに書き込むことが重要なのかを示す例を挙げます。データベース・ファイルがいっぱい (空きページがない) のため、新しいページを割り当てる必要があるとします。SQL Anywhere は、このデータを保管するためにファイルを拡張しなければなりません。このファイルが数ページ分拡張され、新しいページ上のデータがデータベース内の一部のページから参照されようになったとします。この状況でドライブへの電源が停止した場合、新しいページの書き込みが安定メディアに到達していても、リカバリ時にデータベース・ファイルが短くなるという問題が生じることがあります。更新後のファイル・サイズを保持しているファイルシステム情報がディスクにコミットされなかった場合、このファイルは最後に保管された時点の長さに戻されます。この例の場合は、ファイルの末尾より後にあるページを参照しているデータが存在することになります。SQL Anywhere は、これらの新しいページを参照しているものを含む書き込みの前に *flush* 要求を行うことにより、このような問題を回避します。SQL Anywhere はフラッシュ要求の完了を待ってから、電源が停止した場合にデータベースを矛盾した状態にする可能性のある書き込みを実行します。

2.2 トランザクションの永続性

SQL Anywhere は “ACID”[1] に準拠したデータベースであり、正常にコミットされたトランザクションは永続的でなければなりません。つまり、電源が停止した場合も、データベースの再起動後にはトランザクションの効果が存続するということです。そのために、SQL Anywhere は、コミットが発行されるたびに、トランザクションをディスク上のトランザクション・ログに物理的に書き込む必要があります。SQL Anywhere は、最後のチェックポイント以降に実行されたトランザクション・ログ内の操作をリプレイすることでリカバリの第 2 段階を実行し、

1 ACID は Atomicity (不可分性)、Consistency (一貫性)、Isolation (隔離性)、Durability (永続性) を表す略語

データベースを電源停止前と同じ状態に安全にリストアします。

トランザクションがコミットされると、SQL Anywhere は、トランザクション全体の操作を確実に安定ストレージ上のトランザクション・ログに記録するために、フラッシュ操作と、「ライトスルー」と呼ばれる I/O メカニズム (次の項を参照) を組み合わせて使用します。ただし、オペレーティング・システムでこれらの操作の信頼性が保証されなければ、SQL Anywhere はトランザクションの永続性を保証することができません。

3 ストレージ・スタック

3.1 概要

オペレーティング・システムのストレージ・スタックは多くの層で構成されています。これらの層が 1 つでも正しく構成されていないと、スタック全体が信頼できないものになる可能性があります。ストレージ・スタックを簡単に表すと、上位層から順に次のようになっています。

- ファイルシステム
- ストレージ・ドライバ
- ストレージ・コントローラ
- ディスク・ドライブ

SQL Anywhere データベース、トランザクション・ログ、および dbspace はファイルシステム上の通常のファイルに過ぎません。ファイルシステムはオペレーティング・システムで提供され、ファイルやディレクトリに対する操作を、ストレージ・ドライバに対して発行可能な I/O 要求へと変換する役割を担います。ストレージ・ドライバはこれらの要求を受け取ると、それをストレージ・コントローラ (ハードウェア) に直接転送し、最終的に要求はディスク・ドライブに渡されます。以降の項では、これらの各層がストレージ・スタックの信頼性にどのように影響するかについて下位層から順に考察していきます。

3.2 ディスク・ドライブ

ディスク・ドライブは不揮発性ストレージです。すなわち、ディスクに書き込まれた情報は電源が失われても存続します。従来、ディスク・ドライブは回転メディアを使用して実装されてきましたが、現在は SSD (Solid State Drive ：ソリッド・ステート・ドライブ) が一般的になりつつあります。使用テクノロジに関係なく、ほとんどのモデム・ディスクは、パフォーマンスを高めるために、書き込みキャッシュとして高速 (ただし揮発性) DRAM バッファを採用しています。回転ディスクの場合、書き込みキャッシュにより、ドライブは I/O を遅らせてリオーダできるため、プラッタの回転が目的位置に到達するまで待機することにより生じる自然な遅延が軽減されます。SSD では、書き込みキャッシュを採用することで、SSD が内部的に使用する大容量のブロックサイズの書き込みに OS の I/O を再マッピングするための再書き込みによってパフォーマンスが低下するのを防ぎます。

どちらの場合も、I/O が書き込みキャッシュに正常に保管された直後に I/O の完了がレポートされます。書き込みキャッシュ内の I/O が物理メディアに書き込まれる前に電源が停止すると、その I/O は失われてしまいます。

このように、ディスクは I/O が揮発性の書き込みキャッシュに保管された時点で I/O の完了をレポートするため、ストレージ・スタックの上位層では、特定のデータが本当に不揮発性メディアに保管されていることを保証する手段が必要になります。SCSI および ATA コマンド・セットには、ディスク・キャッシュを明示的にフラッシュするコマンドが用意されています。また、FUA (*Force Unit Access* ：強制ユニットアクセス) と呼ばれる I/O メカニズムも備えています。FUA ビットは I/O 単位で設定されます。FUA ビットが設定された書き込みは、安定メディアに到達するまでその I/O が完了したとみなさないようディスクに通知します。FUA を使用するよう構成された I/O は、書き込み (write) がキャッシュを経由 (*through*) してそのまま物理メディアに送られることから「ライトスルー (writethrough)」とも呼ばれます。

パフォーマンスへの配慮から、SQL Anywhere はトランザクション・ログへのフラッシュをコミットごとに発行しません。代わりに、コミットされたトランザクションがディスクに存在することが FUA によって保証されることを前提としています。そのため、基礎となるオペレーティング・システムで信頼できる FUA 操作が提供されていない状況で書き込みキャッシュを使用すると、SQL Anywhere でトランザクションの永続性が保証されません。

3.2.1 Windows

Windows 環境では、*FlushFileBuffers()* 呼び出しによりディスク・フラッシュを要求できます。SQL Anywhere は重要な場面でこの呼び出しを使用することで、データの信頼性を保証するよう OS に要求します。残念ながら、特定の I/O ドライバにバグがあることから、この関数を呼び出しても、フラッシュ・コマンドが必ずしもディスクに渡されるとはかぎりません。

また、SQL Anywhere は、データベースおよびトランザクション・ログを開くときに *CreateFile()* に FILE_FLAG_WRITE_THROUGH フラグを渡して、FUA を使用するよう I/O に要求します。ただし、FUA の処理はすべてのディスク・メーカーおよび Windows バージョンで同じわけではありません。実際、一部の ATA ベースの実装では FUA ビット全体が破棄される状況が観察されており、これによって SQL Anywhere の信頼性は低下します。

Windows 2000 Service Pack 3 (SP3) より古いバージョンの Microsoft Windows では、ディスクへの FUA ビットの伝播が一貫していないことがわかっています。Windows では、FUA ビットを伝播させるためにレジストリ・キーを設定することが必要な場合もあります。

SCSI 実装では FUA ビットが適切に考慮されます。

3.2.2 Linux

Linux 環境では、*fsync* 呼び出しによってディスク・フラッシュを要求します。ただし、Linux では、I/O スタックの上位層に構成や実装の不備があると、*fsync* を呼び出してもフラッシュ・コマンドがディスクに送信されません。

多くの場合、ファイルの破損を防ぐためには、ディスク上の書き込みキャッシュをすべて無効にしなければなりません。

Linux では、I/O での FUA のサポートをユーザランド・プロセスから要求するメソッドは公開されていないため、トランザクションの永続性を保証するために、ディスク・キャッシュを無効にする必要があります。

3.3 ストレージ・コントローラ

ストレージ・コントローラは、オペレーティング・システムのコマンドを 1 つ以上のディスクに対する実際の操作にマッピングするハードウェアです。一般に、市販 PC では、マザーボードに単純なストレージ・コントローラが組み込まれており、これによって少数の個々のディスクとの通信が可能になります。「サーバクラス」のマシンには、RAID や仮想ドライブ構成など、さらに高度な機能を実装する専用のストレージ・コントローラが搭載されているものがあります。また、一部のハードウェアベースの RAID コントローラには、バッテリ・バックアップ書き込みキャッシュを備えているものもあり、この場合、コントローラが書き込みキャッシュ内のデータの永続性を保証できます。このようなコントローラを使用する場合、個々のディスク自体のキャッシュが無効になっていれば、コントローラのキャッシュを安全に有効にできます。

現在、一部の市販 PC メーカーからファームウェアベースの RAID コントローラが提供されていますが、市場ではこれがハードウェア RAID コントローラと混同されています。これらのコントローラは「フェイク RAID」または「ホスト RAID」と呼ばれます。実行する操作は最も基本的な操作に限られ、RAID 処理を実際に実装するドライバがオペレーティング・システムによって提供されることを前提としています。

「フェイクRAID」の意味合いは、次に説明するようにプラットフォームによって異なります。

3.3.1 Windows

Windows では、OS にソフトウェア RAID が組み込まれています。ただし、Windows のストレージ・ドライバに関する項で説明するように、ストレージ・コントローラ・ドライバの選択が問題になることもあります。この環境でも、信頼できる動作を得るために、ディスク・キャッシュ機能を無効にすることが必要になる場合があります。

3.3.2 Linux

Linux では、デバイス・マッパ・レベルでソフトウェア RAID が処理されます (第 3.4.2 項を参照)。この層は、多くの状況でディスク・キャッシュをフラッシュするために要求をストリップすることで知られています。通常、Linux 環境では、このような構成に使用されるデバイスには、“md” プレフィックスで始まる名前が付けられます。ご使用のデバイス名が “md” で始まっている場合は、RAID を使用しないようシステムの構成を変更するか、基礎となるディスクで書き込みキャッシュを無効にしてください。

3.4 ストレージ・ドライバ

I/O スタックにおいてストレージ・コントローラのすぐ上位にある層がストレージ・ドライバです。ストレージ・ドライバは使用されているストレージ・コントローラに固有で、カーネルで使用される汎用ブロック I/O 操作とコントローラのネイティブ・コマンド・セット (SCSI または ATA) 間の変換を担います。

3.4.1 Windows

前に説明したとおり、Windows 環境では、SQL Anywhere はデータベース・ファイルを開く際にフラグ FILE_FLAG_WRITE_THROUGH を使用します。これは、書き込みが実際に物理メディアに保管されるまで、データベース・サーバに対して書き込み完了通知を行わないことを意味します。また、FILE_FLAG_WRITE_THROUGH の処理はすべてのハードウェアおよび Windows のバージョンで共通なわけではありません。実際、このフラグが設定されていても、ディスクが書き込みをバッファリングし続けた結果、書き込みがばらばらの順序でディスクに到達することもあります。このような書き込みを正しい順序で行わなければならないときもありますが、ファイル・メタデータが安定メディア上に存在していなければならないときもあります。そのために、SQL Anywhere は FlushFileBuffers() を呼び出すことにより、未処理のすべての書き込みおよびメタデータをディスクに強制的にフラッシュします。FlushFileBuffers が呼び出されると、キャッシュの同期 (SCSI) またはキャッシュのフラッシュ (IDE/ATAPI) コマンドがストレージ・デバイスに送信されます。これにより、それ以降のすべての書き込みにおいて、フラッシュ要求前のすべての書き込みが安定メディア上に存在することが保証されます。また、安定メディアに保管されたメタデータには、その時点におけるデータベース・ファイルの状態が正しく反映されます。

これをなぜファイルシステムに関する項ではなく「ストレージ・ドライバ」の項で説明するかというと、一部のストレージ・ドライバでキャッシュのフラッシュ要求が無視されることがわかっているからです。このようなストレージ・デバイスはバッテリ・バックアップと組み合わせて使用されていると推測できますが、断定はできません。キャッシュのフラッシュ・コマンドが実行されると、ディスク・キャッシュ全体が安定メディアにフラッシュされます。これには犠牲が伴う可能性があり、パフォーマンスへの影響も考えられます。すなわち、これらのドライブ／ドライバの組み合わせは、リカバリの可能性を代償にパフォーマンスを向上させます。SQL Anywhere を実行する際にキャッシュのフラッシュ要求を発行できないストレージ・ドライバを使用するのは危険です。このような状況では、ACID 要件を満たすことができず、データベース・ファイルが破損する可能性もあります。システムにより、要求時にディスク・キャッシュが実行されることが非常に重要です。ご使用のシステムがこれらの要件を満たしているかどうかについては、各ハードウェアおよびソフトウェアのベンダにお問い合わせください。

3.4.2 Linux

Linux には 2 種類の ATA ドライバ・セットが含まれていることから、問題は多少複雑になります。「古い」／安定したセットは主に旧式の並列 IDE デバイスで使用されています。一方、新しいセットは “libata” と呼ばれ、主に SATA デバイスで使用されていますが、旧式ドライバの多くが新しいモデルへと書き換えられています。新しいモデルでは、ストレージ・スタックの上位層を単純化し、すべてのブロック・デバイスを SCSI デバイスと同様に認識できるよう、SCSI コマンドを ATA コマンドにマッピングします。使用ディスクが SATA ディスクであることがわかっており、ディスク名が (/dev/hda ではなく) /dev/sda である場合、使用ドライバは新しい libata ベースのドライバであると考えられます。

3.4.2.1 デバイス・マッパ

デバイス・マッパ層では、ハイエンド・ハードウェア・ストレージ・コントローラで提供される機能がソフトウェアでエミュレートされます。前に説明しましたが、デバイス・マッパ層は、ソフトウェア RAID と LVM (Logical Volume Manager : 論理ボリューム・マネージャ) をサポートするために使用されます。ソフトウェア RAID により、Linux カーネルは、特殊なハードウェアがなくても、標準 RAID レベルに定義されているアルゴリズムを使用して、複数の物理ディスクを単一のディスクとして表すことができます。

LVM は、仮想ボリューム (ディスクなど) およびパーティションの作成を可能にします。LVM は、新しい物理ディスクを論理ボリュームに追加する機能 (サイズを増加) やデータを破壊することなくパーティションのサイズを変更する機能など、非常に有益な機能を備えています。LVM がシステム管理に役立つことから、一般的な Linux ディストリビューションの多くでは、LVM の使用がインストール時にデフォルトで選択されます。
LVM によって利便性がもたらされる一方、デバイス・マッパ層はストレージの信頼性の妨げにもなります。ついこの間まで、デバイス・マッパ層では、ディスク・キャッシュをフラッシュするために、ストレージ・スタックの上位層からの要求がすべてストリップされていました。現在、この問題は、Linux カーネル 2.6.29 以上で、単一ディスクのみが関与する限られた構成セットでのみ解決されています。従来のデバイス・マッパ層を通じたフラッシュ要求の不確実な動作を踏まえれば、SQL Anywhere を実行するシステムでは、LVM およびソフトウェア RAID の使用を避けることをおすすめします。

3.4.2.2 ブロック I/O 層

ブロック I/O 層は次の 5 つの主要コンポーネントで構成されています。

1. 未処理の I/O 要求のキュー
2. ブロックベースの I/O への、デバイスに依存しない単純なインタフェース
3. キュー内の要求をマージおよびリオーダすることでユーザ構成可能なシステム応答性要件を維持する一連の I/O スケジューラ
4. 最近使用されたファイルシステム・ブロックのキャッシュ

5. ストレージ・ドライバが汎用ブロック I/O 要求を実際の I/O 操作にマッピングするために使用するプラガブル・バックエンド

ブロック I/O 層は、ディスクに対してブロックを読み書きするための単純なインタフェースを提供します。興味深いことに、ディスク・ドライブの書き込みキャッシュをフラッシュするためのメソッドは直接公開されておらず、代わりに、より汎用な I/O バリア操作が公開されています。I/O バリアが処理待ち I/O のキューに挿入されると、バリアより前にあるすべての操作が完了するまで、バリアより後にある操作は完了しないことが保証されます。フラッシュは、書き込みに続いてバリア操作をキューに投入し、書き込みが完了するまで待機することにより、簡単にシミュレートできます。バリアより前にあるすべての操作が完了するまで書き込みは完了できないため、書き込みが完了した時点で、SQL Anywhere はキャッシュ内のすべてのデータが永続ストレージに永久的にフラッシュされたことを保証できます。現在提供されているすべてのストレージ・ドライバは、ディスクの書き込みキャッシュをフラッシュするためのコマンドを発行することにより、バリア操作を実装しています。

Linux は多様な I/O スケジューラを備えています。これらのスケジューラは、“/proc” ファイルシステム内のファイルを変更することにより、ユーザが実行時に選択することができます。各スケジューラによって実装されるポリシーはそれぞれ異なります。*deadline* および *anticipatory* スケジューラは、I/O スループットを最大化することを目的としており、*CFQ* スケジューラはシステム上の複数のアプリケーションに I/O 帯域幅を均等に分配することを目的としています。また、*no-op* スケジューラは送信された要求をそのままストレージ・ドライバに渡します。どの I/O スケジューラを選択してもストレージ・スタックの信頼性に影響することはないため、さまざまな I/O スケジューラを試し、実際の作業負荷において最高のパフォーマンスが得られるものを選択することをおすすめします。

システム全体のパフォーマンスを高めるために、ブロック I/O 層を通過するすべての I/O がキャッシュされます。このキャッシュは一般に「Linux ページ・キャッシュ」と呼ばれ、未使用のメモリを使用するよう動的に拡張され、アプリケーションでメモリが必要になった時点で自動的に縮小します。Linux は、*pdflush* と呼ばれるバックグラウンド・タスクにより、ページ・キャッシュを定期的にフラッシュします。また、ページ・キャッシュは、アプリケーションから *fsync* が発行されたときにも明示的にフラッシュされます。このキャッシュのフラッシュにはバリア操作が関与しないため、常に確実に行うことができます。SQL Anywhere はすでにデータベース・ページのキャッシュを独自に維持しているため、デフォルトでは、ダイレクト I/O を使用してこのキャッシュの使用を回避します。これにより、SQL Anywhere と Linux カーネル間でのメモリの争奪が低減され、通常はパフォーマンスが向上します。必要に応じて、データベース・サーバの開始時に “-u” オプションを指定することにより、ダイレクト I/O を使用しないよう SQL Anywhere を構成することもできます。

3.5 ファイルシステム

SQL Anywhere は、データベースに通常のファイルを使用します。そのため、データベースに対する操作およびデータベースから行われる操作はすべて、ファイルシステム・ドライバを通過することになります。ファイルシステムは、ファイル・データとメタデータの 2 種類のデータを管理します。ファイル・データはアプリケーションから書き込まれる実際のデータで、SQL Anywhere の場合は、データベース・ファイルに保持されている実際のデータに該当します。

メタデータはファイルシステムでファイルを管理するために必要なデータで、作成日時、ファイル・サイズ、パーミッション、ファイル名、ファイルに割り当てられた一連のディスク・ブロックなどが含まれます。

3.5.1 Windows

Windows オペレーティング・システムと SQL Anywhere はどちらも FAT および NTFS ファイルシステムをサポートしています。NTFS は、データベース・システムのトランザクション・ログと同様のメカニズムを使用してディスクに対する変更を管理するジャーナル・ファイルシステムです。NTFS は、データベース・システムと同様に、基礎となるハードウェアによってデータが適切にメディアにフラッシュされることを前提としています。電源停止が発生した場合に、不要なキャッシュが原因で、基本的にファイルシステム自体が古い状態に戻る可能性があります。さらに深刻な状況では、ファイルシステム自体が矛盾した状態になり、破損してしまうこともあります。

3.5.2 Linux

Linux は幅広いファイルシステムをサポートしています。この項では、最も一般的な Ext2、Ext3、Ext4、および XFS について取り上げます。これら 4 つのファイルシステムのうち、Ext3、Ext4、および XFS はジャーナル・ファイルシステム、Ext2 は非ジャーナル・ファイルシステムです。ジャーナル・ファイルシステムは、使用されているデータ構造のディスク上表現に対する変更を管理するために、データベース・システムと同様のロギング機能を内部的に実行します。ファイルシステムがクリーンにアンマウントされなかった場合でも、次回マウント時に矛盾のない状態に戻すことができます。ジャーナルを持たない Ext2 は、正常にアンマウントされないと破損する可能性があり、この場合は *fsck* ユーティリティを使用して修復する必要があります。

Ext3 および Ext4 には、次の 3 種類のジャーナル・モードがあります。

- data=journaled – このモードでは、データおよびメタデータのすべての変更がジャーナルに書き込まれます。このオプションでは、すべてのデータを 2 回書き込む必要があるため、パフォーマンスはあまり高くありません。また、ダイレクト I/O を使用できないことから、SQL Anywhere のパフォーマンスがさらに低下します。

- data=ordered – このモードでは、メタデータの変更のみがジャーナルに書き込まれますが、処理待ちのデータ変更はジャーナルのコミット前にすべてディスクにフラッシュされます。このモードでは、data=journaled と同レベルの一貫性が保証され、しかもパフォーマンスは大幅に優れています。Ext3 および Ext4 では、これがデフォルトのジャーナル・モードです。

- data=writeback – このモードでは、メタデータの変更のみがジャーナルに書き込まれますが、処理待ちのデータ変更はジャーナルのコミット前に強制的にディスクに書き込まれません。このモードでは、オペレーティング・システムのクラッシュや電源停止が発生した場合に、リカバリ後にファイルの新しい部分に古いブロックが保持される可能性があります。これはセキュリティ・リスクの原因になりますが、

アプリケーションが *fsync* によってデータを適切にフラッシュするかぎり、信頼性には影響しません。

data=ordered および data=writeback モードが正しく機能するかどうかは、バリア操作をブロック I/O 層に発行するファイルシステムにかかっています。意外にも、Ext3 および Ext4 はデフォルトでバリアを使用するよう構成されていないため、このオプションを明示的に有効にする必要があります。Ext3 および Ext4 でバリアのサポートを有効にするには、*/etc/fstab* で、ファイルシステムのマウント・オプションに “barrier=1” オプションを追加し、再起動します。

XFS では、data=writeback とほぼ同等の単一ジャーナル・モードのみがサポートされます。バリアの使用はデフォルトで有効になっていますが、ファイルシステムに “/nobarrier” オプションを指定して無効にすることもできます。XFS を使用する場合は、*/etc/fstab* でファイルシステムのマウント・オプションを調べ、“/nobarrier” オプションが使用されていないことを確認してください。

SQL Anywhere の信頼性を確保するためには、*fsync* の呼び出しにより、*fsync* より前に発行されたすべての書き込みが、この呼び出しが戻ったときには安定ストレージ上に保管されていることが保証されなければなりません。ディスクの書き込みキャッシュが使用されていない場合、Ext2、Ext3、Ext4、および XFS ではこの要件が満たされます。書き込みキャッシュが使用されている場合、ファイルシステムはキャッシュをフラッシュするために、*fsync* でバリア操作を発行する必要があります。Ext2 はいかなるバリア操作も発行しないため、安全ではありません。

従来、data=ordered または data=writeback モードの XFS、Ext3、および Ext4 では、ファイルのメタデータに対する変更がある場合のみ *fsync* でバリア操作が発行されていました。メタデータの変更を伴わないファイルのインプレース更新では、*fsync* 呼び出しで、これらのファイルシステムからバリアが発行されません。Ext3 および Ext4 についてはカーネル 2.6.32 から、また XFS についてはカーネル 2.6.33 から、バリアのサポートに必要なオプションを使用してファイルシステムがマウントされていれば、これらのファイルシステムから *fsync* で常にバリア操作が発行されるようになっています。これより古いバージョンのカーネルを使用しているか、またはバリアのサポートを有効にしていない場合は、ドライブの書き込みキャッシュを有効にできず、SQL Anywhere が信頼できるストレージであるための要件を維持することができません。

4 システムの構成

ディスクによる不要なキャッシュは、電源が停止した場合にデータの破損や矛盾の原因となる重要な要因です。多くの問題は、ディスクの書き込みキャッシュを無効にすることで軽減できます。

4.1 Windows

4.1.1 書き込みキャッシュの無効化

- デバイス マネージャに切り換えます (Windows 7 および Windows Vista の場合は [コントロール パネル] → [デバイス マネージャ] を選択。Windows XPの場合は[コントロールパネル] →

[管理ツール] → [コンピュータの管理] → [デバイス マネージャ]を選択)。

- [ディスク ドライブ]を右クリックして[プロパティ]を選択します。
- [ポリシー]タブをクリックします。
- [ディスクの書き込みキャッシュを有効にする]チェックボックスをオフにします。

4.2 Linux

4.2.1 デバイス名の特定

df コマンドを使用すると、SQL Anywhere データベースが存在するデバイスを特定することができます。一般に、SQL Anywhere データベースは複数の dbspace ファイルとトランザクション・ログ・ファイル (および多くの場合はミラー・ログ・ファイル) で構成されています。次に示す例では、メイン・データベース・ファイル、すなわちシステム dbspace のみが使用されています。デバイスを特定するには、唯一の引数として *df* に SQL Anywhere データベース・ファイルを渡します。これにより、出力の第 1 カラムで、このファイルが存在しているパーティションを表すデバイス・ファイルが与えられます。

```
$ df /opt/sqlanywhere11/demo.db
Filesystem          1K-blocks      Used Available Use% Mounted on
/dev/sda1           921150000 394688772 526461228  43% /
```

この例でレポートされたデバイス名は /dev/sda1 です。実際のブロック・デバイス名を特定するには、レポートされたデバイス名のサフィックスからすべての数字を除去します。この例の基礎となるブロック・デバイスのデバイス名は /dev/sda です。

与えられたデバイス名から、使用デバイスの種類を判断することができます。デバイス名が /dev/sd で始まっている場合、SCSI ディスクを使用しているか、または SATA ディスクと新しい libata ベースのドライバを使用しています。/dev/hd で始まっている場合は、ATA ベースのデバイスと旧式の「並列 IDE」ベースのドライバを使用しています。また、/dev/md で始まっている場合は、システムでデバイス・マッパ層によってソフトウェア RAID が実装されています。さらに、/dev/mapper/ で始まっている場合は、システムでデバイス・マッパ層によって LVM が実装されています。ソフトウェア RAID または LVM を使用している場合、ドライブの書き込みキャッシュを無効にするには、基礎となる実際のデバイスを特定する必要があります。

ソフトウェア RAID の場合、基礎となるデバイスは、/sys ファイルシステムを調べれば特定できます。たとえば、*df* コマンドにより、データベース・ファイルがデバイス md0 上にあることがわかったとします。この場合、/sys/block/md0/md/ には、プレフィックス dev- を持つ、基礎となる実際のデバイスへのシンボリックリンクが含まれています。

```
$ ls -d /sys/block/md127/md/dev-sd*
/sys/block/md127/md/dev-sda  /sys/block/md127/md/dev-sdb
```

LVM を使用している場合は、*pvs* ツールにより、マシン上のすべての物理ボリュームと、それが割り当てられているボリューム・グループを列挙することができます。

```
$ df /opt/sqlanywhere11/demo.db
Filesystem                    1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                              28376956  14314528  12597700  54% /

$ pvs
  PV         VG         Fmt  Attr PSize   PFree
  /dev/sda2  VolGroup00 lvm2 a-   29.88G      0
  /dev/sdb1  HomeGroup  lvm2 a-   10.00G      0
  /dev/sdb2  HomeGroup  lvm2 a-  989.99G      0
```

この例では、データベース・ファイルはデバイス /dev/VolGroup00-LogVol00 に存在しています。また、*pvs* の出力から、このデバイスが物理デバイス /dev/sda 上にあると判断できます。

4.2.2 書き込みキャッシュの無効化

書き込みキャッシュを無効にするには、まず、前項の手順に従って、SQL Anywhere データベースが存在している基礎となる物理デバイスを特定します。書き込みキャッシュを無効にするためのツールは、使用デバイスの種類によって異なります。デバイス名が /dev/sd で始まっている場合は、次のように *sdparm* ツールを使用します。

```
$ sdparm --clear=WCE /dev/sda
```

再起動後にも変更を存続させるには、次のコマンドを使用します。

```
$ sdparm --clear=WCE --save /dev/sda
```

デバイス名が /dev/hd で始まっている場合は、次のように *hdparm* ツールを使用します。

```
$ hdparm -W 0 /dev/hda
```

sdparm とは異なり、*hdparm* コマンドには、再起動後に変更を存続させるためのメカニズムがありません。この変更を存続させるには、ディストリビューションのドキュメントに従って init スクリプトを作成し、上記の *hdparm* コマンドを含めます。

5 まとめ

SQL Anywhere が信頼できるストレージであるために、オペレーティング・システムで提供されるストレージ・スタックに求める要件は 1 つのみです。すなわち、*flush* 呼び出しにおいて、この呼び出し前に発行されたすべての書き込みが呼び出し完了時には安定ストレージに保管されていることが保証されなければなりません。書き込みキャッシュを有効にしてディスクを使用する場合、ストレージ・スタックはさまざまな条件により不安定になる可能性があります。

5.1 Windows

次のいずれかの条件に当てはまる場合、前述の手順で書き込みキャッシュを無効にすると、システムのリカバリ可能性が高まります。

- FUA ビットが考慮されない
- キャッシュのフラッシュ要求が考慮されない

5.2 Linux

システムの構成が次のいずれかの条件に当てはまる場合、前述の手順でディスクの書き込みキャッシュを無効にする必要があります。ディスクの書き込みキャッシュを安全に有効にできるのは、次のどの条件にも該当しない場合のみです。

- Ext2 ファイルシステムを使用している
- barrier=1 マウント・オプションを指定しないで Ext3 または Ext4 ファイルシステムを使用している
- 2.6.32 より古い Linux カーネルで Ext3 または Ext4 ファイルシステムを使用している
- /nobarrier マウント・オプションを指定して XFS ファイルシステムを使用している
- 2.6.33 より古い Linux カーネルで XFS ファイルシステムを使用している
- 2.6.29 より古いカーネルで LVM、ソフトウェア RAID、フェイク RAID、またはホスト RAID を使用している

法的注意