New Journal System

ロホン・リバルディエール

original presentation by Laurent Ribardière

新しくなった 4D のジャーナルシステム

はじめに

v14 では，ジャーナル（ログ）ファイルの内部フォーマットが新しくなりました。この資料では，新旧のジャーナル形式を説明し，内部フォーマットの変更を検討することになった経緯，その目的，既存のデータベースが受ける影響を考察します。結論からいえば，ジャーナルファイルの内部フォーマットが変更されたことにより，既存のデータベースは，アップグレード前にストラクチャを修正しなければならことになりました。テーブル数の多いシステムや，同じストラクチャを複数のサイトに配付しているソリューションでは，しっかり前もって対策を立てることが必要です。今回の変更により， 4D に数々の新しい可能性が開かれました。今後，新しいフォーマットを活かした強力で便利なソリューションが提供される予定です。

この資料は，2013 年 4D Summit（ネバダ州ラスベカス）および 4D Summit Euro（パリ）でのロホン・リバルディエールのプレゼンテーションを基に，4D Japan が構成しました。

壊れたジャーナルとデータ復旧

理想的な世界では，データベースはいつまでも順調に動き，クラッシュすることはありません。しかし，現実の世界は残酷です。4D Server はクラッシュするかもしれませんし，ハードディスクは壊れるかもしれませんし，誰かが電源ケーブルにつまづいてプラグがコンセントから抜けてしまうかもしれないのです。そんなとき，運悪く，データベースがキャッシュの書き込み中だったとすればどうでしょうか。最悪，そのデータファイルは壊れたものになり，2 度と再起動できないかもしれません。そんな事態に備えるのがバックアップおよびジャーナルファイルです。**バックアップ**があれば，それを作成した時点までデータベースは復元できます。さらに，**ジャーナル**（ログ）があれば，事故が起きる直前のデータまでを取り戻すことができます。

では，データファイルだけでなく，ジャーナルファイルも同時に壊れたとしたらどうでしょうか。（本来，こうしたことを避けるために，データファイルとジャーナルファイルは別々のディスクに作成することが勧められています。しかし，現実にこういうことは起こり得ます。）

ジャーナルファイルが壊れた，といっても，実際には，ある部分が読み取れない，あるいは不完全なだけで，実際には，かなりのデータは「無傷」かもしれません。一般的なドキュメントであれば，少々の被害があったとはいえ，すべてが失われたわけではない，と思うかもしれません。しかし，ジャーナルファイルは違います。壊れたジャーナルは，たとえ部分的にリカバリーできたとしても，もう統合（復旧）には使用できません。

4D は，壊れたジャーナル，つまり少しでも内容が欠損しているログファイルをデータファイルに統合することが絶対にできないようになっています。デベロッパーの中には，それを理不尽と感じる人がいるかもしれません。しかし，ジャーナルファイルの内容を知れば，不完全なジャーナルをデータファイルに統合できない理由がもっともなものであることが分かります。

レコード番号とジャーナリング

ジャーナルには，レコードデータに対するすべてのオペレーションが記録されています。つまり，追加・更新・削除・トランザクションそして実際のレコードデータです。また，4D では**レコード番号**でレコードを特定しています。したがって，ジャーナルファイルの中には，簡単にいって，レコード#1 を作成，レコード#2 を更新 …といた内容が書かれています。4D は，番号が分かれば，データファイル内におけるレコードの位置が分かるようになっています。

ご存知のように，レコード番号は再利用できることになっています。つまり，レコードが削除された場合，その同じ番号が別のレコードに与えられる，ということです。ある瞬間に別々のレコードが同じレコード番号を持つことはありませんが，時間の経過とともに，レコード番号によって参照されるレコードが変わることはあります。

ジャーナルファイルには，レコード番号，オペレーション，値などが，順序どおりに記録されています。ですから，これを順序どおりに再生すれば，結果的に同じ番号のレコードには同じ値が登録されます。しかし，もし，ひとつでも削除や作成のオペレーションをスキップすれば，以後のレコード番号がすべて狂ってしまうかもしれません。データは合っているかもしれませ

んが，間違った場所（レコード）に保存されてしまうのです。結局，そのデータベースはデタラメなものです。

4D 本社のクオリティ責任者であるジャン・ラゲーエは，かつて「力技」で壊れたジャーナルからデータをサルベージしたことがあります。まず，バイナリエディターでジャーナルを開き，取り出せる限りのログを復元しました。その後，破損した箇所の内容を調査し，前後でレコード番号の対応関係が保たれるよう，ジャーナルから復元できないデータはすべてマニュアル入力したのです。1 日がかりで，ついにデータの復元に成功しました。見方を変えるならば，そうでもしない限り，壊れたジャーナルからデータを復元することはできない，ということです。

プライマリーキーとジャーナリング

壊れたジャーナルがデータファイルに統合できない，もっといえば，ジャーナルファイルを選択的に統合することができないのは，レコードの特定をレコード番号に頼っているからです。もし，レコード番号の代わりに**プライマリーキー**でレコードを特定すれば，もっと柔軟にジャーナルファイルを扱うことができ，結果的にデータベースの堅牢性を高めることができます。レコード番号とは違い，プライマリーキーは，追加・削除の順序や回数とは無関係だからです。

そのようなわけで，v14 では，レコード番号の代わりにプライマリーキーがジャーナルファイルに記録されるようになりました。プライマリーキーとは，データベースのテーブルに登録された **1 件のレコードを特定する一意識別子**（Unique Identifier）のことです。一般的なデータベースでは，1 対 N リレーションの 1 レコードを特定するため，あるいは単純に 1 件のレコードを特定するためにプライマリーキーを使用します。対照的に，4D は，そのようなときにはレコード番号（例: GOTO RECORD）やインデックスフィールド（例: Find index key）が使用され，プライマリーキーは特に必要ありませんでした。そもそも，バージョン 2004 以前は，プライマリーキーというものが存在しませんでした。しかし，v14 では，そうした認識を改めなければなりません。v14 以降，基本的にどのテーブルもプライマリーキーが設定されていることが期待されています。実際，新しく作成されたテーブルには，自動的にプライマリーキー

が作成されるようになっています。一方，プライマリーキーが設定されていないデータベースは，ジャーナルファイルを作成することができません。

4D は，v11 以降，プライマリーキーを追加しようと思えば，追加することができました。しかし，現実の問題として，ほとんどの 4D データベースには，プライマリーマリーキーが設定されていません。ですから，アップグレード後もジャーナルファイルを使用するためには，ストラクチャを修正することが必要です。テーブル数の多いシステムや，同じストラクチャを複数のサイトに配付しているソリューションであれば，しっかり前もって対策を立てなければなりません。しかし，具体的な作業の流れを考慮する前に，プライマリーキーをジャーナルに記録することのメリットをもういちど振り返ってみましょう。

最大のメリット: 危機管理

前述したように，レコード番号に基づいたジャーナルファイルには，**部分的な統合ができない**，という本質的な制限があります。壊れたジャーナルを「できるだけ」統合したり，統合したくない（例えば誤入力）内容を飛ばして統合することは構造的に無理です。一方，プライマリーキーに基づいたジャーナルファイルであれば，もっと柔軟にジャーナルを扱うことができます。v14 の後期リリースでは，MSC に新しい「リカバリーモード」が用意される予定です。このモードでは，**壊れたジャーナルファイルの中から選択的にオペレーションを統合する**ことができます。それが可能なのは，ジャーナルにプライマリーキーが記録されているからです。また，v14 では，**新しいミラーリングコマンド**が追加され，ミラーサーバー運用のバリーエーションが増えました。これも，新しいジャーナルのおかげです。最後に，v14 の後期リリースでは，待望の**フルオートマチック・ミラーサーバー**が提供できることになりました。いくつかのパラメーター（サーバーの場所，ミラーリングの頻度など）を設定するだけで，自動的にサーバーからジャーナルを取り出し，常にアップデートされるサーバーです。しかも，そのようなサーバーをいくつもセットアップすることができます。ですから，ストラクチャを変更することには一定の苦労が伴いますが，それだけの価値があると考えられています。

ジャーナルファイルの内容

4D のジャーナルファイル（旧称・ログファイル）のフォーマットは，v14 で新しくなりました。ジャーナルは，バイナリファイルなので，自分で解読しようと思えば，HEX エディターが必要になりますが，実際の構造は比較的シンプルなものです。ジャーナルに記録されるオペレーションは，どんなものであっても，最初にちいさな（4 バイト）ヘッダータグがあり，それがジャーナルのエントリーであることをチェックできるようになっています。その後には，8 バイトのオペレーション番号が続きます。これは，オペレーション毎にインクリメントされる一意の管理番号です。その後は，データサイズ（8 バイト）およびオペレーションのタイプ（追加・削除・更新など）が 4 バイト，それから実際のデータが続きます。データサイズは，オペレーションのタイプによってさまざまです。最後にデータサイズ（8 バイト）がもう一度，記録され，フッタータグで終わりとなります。データのサイズが 2 度，記録されるのは，**ジャーナルを逆から読むこともできるようにするため**です。これにより，たとえば，ジャーナルファイルの末尾から初めて，最後のオペレーションをみつけることができます。あるいは，途中が破損しているジャーナルファイルの中からできる限りのデータを取り出すこともできます。なぜならば，ジャーナルファイルのサイズに関係なく，特定のオペレーションが記録されているおおまかな位置を計算でき，その前後からどんなエントリーでも素早くみつけることができるからです。この仕組みは，分散ミラーリング（後述）システムにも活かされています。

新フォーマットのメリット

旧フォーマットのジャーナルには，レコード番号が記録されていました。そのため，レコード番号の削除や追加のエントリーがひとつでも欠けていれば，以降のログにおけるレコード番号がすべて狂います。以前のバージョンでは，カウンターが連続していないオペレーションに遭遇した時点で，ログの統合（再生）を中止し，それ以降のログは決して統合できないようになっていました。もし，何らかの理由（バグ，ネットワークやディスクの障害，その他）により，

ログファイルを部分的に読めないことがあれば，ログファイルの統合は断念し，最後のバックアップから再出発しなければなりません。

新フォーマットでは，レコード番号の代わりにプライマリーキーが記録されるようになりました。そのため，たとえジャーナルを途中で飛ばしたとしても，**最悪，レコード数が合わないか，その値が古いままになるだけ**で，以前のように以降のオペレーションがまったく狂うようなことはありません。**運が良ければ，すべてのデータが完全に復元できる**かもしれません（飛ばされたオペレーションが，後のオペレーションで上書きされるものだった場合）。

別の点として，レコード番号に基づく以前のログファイルでは，レコードを削除した後，同じレコード番号が別のレコードに割り当てられたため，ジャーナルをみただけではどちらのレコードを削除するべきなのか，知ることができませんでした。プライマリーキーに基づくログファイルであれば，時間の経過と関係なく，削除されるべきレコードを特定することができます。

直接，プライマリーキーとは関係ないことですが，v14 ではジャーナルファイルをより柔軟に扱えるようにするため，もうひとつの点が変更されました。それは，グローバルオペレーションカウンターのインクリメント条件です。

グローバルオペレーションカウンター

前述したように，ジャーナルには，オペレーション（レコードの追加・削除・更新など）のタイプやデータとともに，実行された順序を示す一意の管理番号が記録されています。それは，**オペレーション毎にインクリメントされる 64 ビット整数のカウンター**です。

4D で何らかのデータベースオペレーションが発生すると，まずその情報がジャーナルに書き込まれ，続いてそのデータがキャッシュに書き込まれ，やがてキャッシュがデータファイルに書き込まれます。データファイルには，最後に実行されたオペレーション番号が記録されているので，たとえ書き込み前あるいは書き込み中にクラッシュが発生したとしても，ジャーナルのカウンターとデータのカウンターを比較すれば，まだデータファイルに書き込まれていない

オペレーションを特定することができます。たとえば，データベースがクラッシュし，データファイルに書き込まれたオペレーション番号が 100，ジャーナルに記録された最後のオペレーション番号が 120 であれば，101 番から 120 番までのオペレーションを再生すれば良いことが分かります。

v14 では，このグローバルオペレーション番号をインクリメントする条件が見直され，**ほんとうにデータを書き換えるオペレーションだけがカウンターをインクリメントする**ようになりました。v13 までは，データベースを開いたり，閉じたりしただけでも，オペレーション番号がインクリメントされました。これにより，新しいミラーリングのスタイル，**連鎖ミラーリング**ができるようになりました。

連鎖ミラーリング

前述したように，旧フォーマットのジャーナルは，データファイルを開閉しただけでも，グローバルオペレーションカウンターがインクリメントされました。そのため，メインサーバー（あるいはミラーサーバー）でデータファイルやジャーナルファイルを開き直しただけでも，ログファイルの符号が一致しない（新しすぎる，古すぎる）状況になり，ログファイルの使用を継続することができなくなりました。（身に覚えのある方は多いのではないでしょうか）

v14 のジャーナルでは，たとえ 50 回，データベースを閉じたり開いたりしても，データを改変しない限り，グローバルカウンターがインクリメントされることがありません。そのため，ジャーナルをミラーサーバーに統合した後，別のミラーサーバーに転送し，そこでも統合する，といったことができます。これが**連鎖ミラーリング**です。

以前のジャーナルでは，ミラーサーバーをひとつしか設置できませんでした。新フォーマットでは，ミラーサーバーはいくつでも追加でき，システムの堅牢性を高めることができます。

ミラーリングとバックアップ

以前のバージョンでは，ミラーサーバーでバックアップを実行することは許されていませんでした。バックアップを実行すれば，グローバルカウンターがインクリメントされてしまい，次のジャーナルが統合できなくなるからです。しかし，この制限は v14 で取り払われました。v14 では，ミラーサーバーでもバックアップを実行することができます。もちろん，ジャーナルファイルを作成することもできます。実際，v14 では，ミラーリングを実施しているのであれば，むしろ**メインサーバーのバックアップを実行しない**ことが勧められています。バックアップは，システムをブロックする要因になるからです。ミラーサーバーで定期的にバックアップを実行し，ジャーナルも記録しているのであれば，それはメインサーバーをバックアップしているのと同じことです。そうすることは，サーバーの負担を軽減することにも寄与します。

トランザクションとジャーナリング

旧フォーマットのジャーナルでは，まずトランザクションの開始とともにエントリーが追加され，続いてトランザクション中のオペレーション（追加・削除・更新）が逐一記録され，最後にトランザクションの終了つまり確定またはキャンセルがひとつのオペレーションとして記録されました。また，ミラーリングをセットアップする場合，まずジャーナルを閉じてから（New log file），そのジャーナルをミラーサーバーに転送して統合しました（INTEGRATE LOG FILE）。このとき，もし，トランザクションが実行中であれば，メモリ（あるいは一時ファイル）に置かれた進行中のトランザクションは，その開始からの内容が次のログファイルに持ち越されました。つまり，閉じられたほうに記録されたオペレーションには，無効フラグが立てられ，次のファイルにオペレーションが再記録されます。しかし，このフォーマットには，次のようなデメリットがありました。

- ミラーリングを実施するためには，頻繁にジャーナルを閉じなければならない
- おおきなトランザクションはひとつのジャーナルファイルに収まりきらないかもしれない
- 多数のトランザクションが実行される場合，ジャーナルを閉じるタイミングがない

フルオートマチック・ミラーリング，また v14 の新しいミラーリングコマンド（INTEGRATE MIRROR LOG FILE）は，**開始オペレーション番号を指定**することにより，ジャーナルを途中から統合できることがポイントになっています。新フォーマットのジャーナルは，両方向に読むことができ，目的のオペレーションを素早くみつけられるため，そうしたこともできるようになりました。

しかし，そうなると今度はトランザクションが問題になります。旧フォーマットのように，トランザクション中の各オペレーションをジャーナルに記録すれば，指定したオペレーションの範囲がトランザクションの途中，あるいは一部ということになるかもしれません。しかし，トランザクションは不可分の操作であるはずです。そこで，v14 のトランザクションは，**コミットされた時点ですべてのオペレーションがジャーナルに記録される**ようになりました。

新フォーマットでは，プロセスがトランザクションを確定した時点ですべてのオペレーションがジャーナルに記録されます。逆に，キャンセルされたトランザクションは一切，ジャーナルに記録が残りません。確定されたトランザクション内の各オペレーションは，それが一連の動作であることを保証する仕方で書き込まれています。もちろん，他のプロセスのオペレーションが途中に挿入されることはありますが（ジャーナリング中に他のプロセスをブロックしないため），それでもトランザクションが断片的に扱われることはありません。

これにより，少ない分量でジャーナルが処理されたとしても，効率的にトランザクションが統合できるようになりました。また，最終的にキャンセルされる一時的なトランザクションがジャーナルに記録されない分，ジャーナルファイルがコンパクトなものになり，そのようなオペレーションをログの統合で再生する必要がない，というメリットがあります。

分散ミラーリング

フルオートマチック・ミラーリングのシナリオは，このようなものです。ミラーサーバーは，メインサーバーに接続し，定期的に最新のジャーナルをリクエストします。つまり，最後に統合したオペレーション番号を指定し，また次回，要求するべきオペレーション番号を受け取り

ます。メインサーバーは，ジャーナルファイル（そうしている間にもログは追加中）を逆から，あるいは途中から読み，要求されたログを返します。このとき，次回に備えて最後に返したオペレーション番号も控えておきます。ミラーサーバーは，受け取ったログを統合し，次回のミラーリングに備えます。メインサーバーは，ミラーリングを実行するたびにジャーナルを閉じなくても良く，複数のミラーサーバーから別々の開始オペレーションを要求されたとしても，それに応じることができます。これが**分散ミラーリング**です。

前述した連鎖ミラーリングは，ミラーからミラーへ，**同じジャーナル**をリレーしてゆきますが，分散ミラーリングでは，それぞれのミラーサーバーが直接，メインサーバーに**別々のジャーナル（開始番号）**を要求することができます。さらに，各ミラーサーバーにジャーナルが統合される間隔を変えることができます。v14 の後期リリースでは，これを全自動で実行するフルオートマチック・ミラーサーバーが提供される予定です。連鎖ミラーリングは，途中のサーバーで障害やエラーが発生すれば，後続のミラーがすべて影響を受けるため，分散（平行・非同期）ミラーリングのほうが優れているということができます。

実際，複数のミラーサーバーを設置する場合，それぞれのミラーサーバーは，できるだけ違った環境で運用されるのが理想的です。ミラーサーバーには，基本的にふたつの目的があります。ひとつは，なるべくダウンタイムを抑えてサービス復帰を早めるため，もうひとつは，完全なデータを保護するためです。ミラーとメインを同じ場所に置けば，前者を実現することができます。しかし，後者を重視するのであれば，ミラーとメインは別々の場所に置くべきです。分散ミラーリングは，そうしたジレンマを解決することができます。

あるいは，ユーザーにより誤入力（たとえばレコード削除）も，それが発覚した時点でまだ統合されていないミラーサーバーがあれば，そこでロールバックを実行し，問題の操作を飛ばして残りのオペレーションを統合する，といった対処が考えられます。そのためには，各ミラーにジャーナルが統合されるタイミングをずらしておくことが効果的です。もちろん，すべてのミラーサーバーに問題のあるジャーナルが統合された後であっても，バックアップからの復元はできます。しかし，それは最後の手段であるはずです。

別データファイルのジャーナルを統合

4D がデータファイルを作成すると，そのファイルにはいくつかの識別子（符号）が記録されます。ストラクチャとの組み合わせをチェックするための識別子，ジャーナルとの組み合わせをチェックするための識別子などです。通常，ジャーナルファイルは，識別子が対応するデータファイルでなければ，統合ができないようになっています。事故を防止するためです。しかし，敢えて**別のデータファイルで作成されたジャーナル**を統合したいケースがあるかもしれません。v14 では，そういうこともできるようになりました。

たとえば，こんなケースが考えられます。ミラーサーバーがセットアップされており，メインサーバーは順調に稼働を続けています。しかし，一部のデータに問題がありそうなことが判明しました。デベロッパーは，メインサーバーを止めずに，ミラーサーバーのほうでデータファイルを修復します。しかし，修復を実行すれば，新しいデータファイルが作成されるため，その識別子とメインサーバーで運用されているジャーナルは互換性がありません。本来，ジャーナルを統合できるのは，そのファイルを作成した当時のデータファイルか，そのコピーだけだからです。

v14 がベータ版だったころ， INTEGRATE LOG FILE に新しいパラメーターが増えた時期がありました。それは，データファイル識別子を無視し，ジャーナルを統合するためのものでした。しかし，最終的には，別コマンド，INTEGRATE MIRROR LOG FILE にその仕様が移されました。INTEGERATE LOG FILE は，識別子が一致しない別データファイルのジャーナル統合を阻みます。つまり，こちらのコマンドは，以前と動作が変わりません。

一方，INTEGRATE MIRROR LOG FILE は，データファイルの識別子が一致しない別データファイルのジャーナルでも統合することができます。もっとも，ある程度のセキュリティは保証されています。テーブルに識別子があるので，ログのデータが間違ったテーブルに統合されることはありませんし，統合を開始するオペレーション番号を指定する必要があるので，複数のジャーナルを間違った順番で統合するようなことはできないからです。

ジャーナルに含めないテーブル

以前のバージョンでは「ログファイルを使用する」プロパティを有効にした場合，すべてのテーブルがジャーナリングの対象になりました。v14 では，一部のテーブルをジャーナルに含めないように設定することができます。たとえば，集計や解析などのために使用され，途中のオペレーションを記録する必要性が低い一時テーブル，郵便番号データのように別の提供元から再現できるテーブルなどは，ジャーナルから除外することにより，全体のパフォーマンスを向上させることができます。ジャーナルは，データファイルとは違うディスクに作成されているはずですが，それでも，多少はパフォーマンスに影響があるからです。あるいは，エラーログなども，ジャーナリングに含めなくても良いケースがあるかもしれません。 ジャーナルに含めるテーブルを取捨選択することにより，ジャーナルファイルのサイズを抑えることができます。

ミラーサーバーの読み書き

以前のバージョンでは，ジャーナルの統合以外の方法でミラーサーバーのデータを更新することは御法度でした。グローバルオペレーションカウンターやレコード番号が狂ってしまい，ログの統合ができなくなるからです。v14 では，ミラーサーバーのデータを更新しても大丈夫な場合があります。たとえば，ジャーナルに含めないテーブルは，ジャーナルの統合で上書きされないので，ミラーサーバーで更新しても問題ありません。あるいは，ジャーナルに含まれているテーブルであっても，それがジャーナルで上書きされないレコード，あるいは上書きされても構わないレコードであれば，ミラーサーバーで更新することができます。

ミラーのジャーナルをメインに統合できるか

v14 では，ミラーサーバーもジャーナルファイルを作成することができるようになりました。そこで双方向にジャーナルを交換できるのではないか，と考えるかもしれません。たとえば，支店（あるいは営業担当者）のデータベースで単独にデータを更新し，そのジャーナルをメイ

ンのデータベースに統合する，といったシナリオです（それぞれのストラクチャは一緒）。新しいジャーナルであれば，確かにそういうこともできますが，もともとジャーナルは双方向の統合を想定したものではないことを踏まえておかなければなりません。

新フォーマットのジャーナルでは，プライマリーキがレコードを特定します。また，新コマンドを使用すれば，データファイルとジャーナルファイルの符号を無視することができます。ですから，別々に作成されたジャーナルを互いに統合することもできないわけではありませんが，その場合，ジャーナルの統合によってデータベースの論理的整合性が失われないよう，気を配る必要があります。もし，各支店（あるいは営業担当者）の役割が明確に分かれており，同じレコードにアクセスすることがないのであれば，問題はありませんが，ビジネスアプリケーションでそのようなケースはむしろ例外的であると思われます。

アップグレード対策

v14 のジャーナルには，レコード番号の代わりにプライマリーキーが記録されます。言い方を換えるならば，**プライマリーキーのないテーブルは，ログファイルに含めることができません**。ログファイルに含められない，ということは，ジャーナルからの復元もできないという意味です。アップグレード後もジャーナルを有効にするためには，プライマリーキーを作成する必要があります。もっとも，これまで特にプライマリーキーを意識していなかったとしても，ビジネスアプリケーションの性格上，すでに何らかの一意識別子がデータベースの中でもう使用されているかもしれません。そうであれば，そのフィールドを正式なプライマリーキーとして指定する，という対策も考えられます。具体的には，そのフィールドやフィールドの組み合わせにプライマリーキー属性を設定することにより，ストラクチャレベルでそれをプライマリーキーに設定するという意味です。たとえば，**テーブルにひとつだけ存在する重複不可フィールド**はプライマリーキーにできるかもしれません。あるいは，**N 対 1 リレーションの 1 フィールド**も，プライマリーキーの候補として挙げることができます。そのようなフィールドは，すでにアプリケーションの中で「キー」の役目を果たしているかもしれませんが，ジャーナルを使用するためには，プライマリーキー属性を設定しなければなりません。なお，プライマリーキー

属性を設定した時点で，そのフィールドは（対応するプロパティ値に関係なく）自動的に重複不可・NULL 不可となります。しかし，勧められているのは，既存フィールドの中からプライマリーキーを選ぶのではなく，新しい自動 UUID フィールドを追加するという対策です。その理由は後述します。

既存のフィールドは使用しないほうが良い?

データベースには，「プライマリーキーっぽい」フィールドがすでに存在するかもしれません。たとえば，請求書番号，顧客番号といったフィールドです。あるいは，患者番号＋カルテ番号＋日付＋時間といった具合に，いくつもの文字列を連結してキーとなる識別コード値を（トリガで）生成しているかもしれません。理論的には，そのようなフィールドを「正式な」プライマリーキーに昇格させることもできます。実際，プライマリーキー管理ウィザードも，そのようなフィールドをプライマリーキー候補としていくつか挙げるようになっています。しかし，総合的にみれば，**既存のフィールドをプライマリーキーにすることは勧められません**。

プライマリーキーの条件を思い出してください。それは，NULL ではない，そして重複しない，というものでした。請求書番号や顧客番号といったフィールドは，確かにその条件が満たされているように思えます。タイムスタンプなどの文字列もそうです。しかし，v14 のプライマリーキーは，バックアップからの復元に使用されるキーでもあるので，**絶対に値を変更しない**，という条件をこれに加えなければなりません。請求書番号などは，現場で訂正できるようになっているかもしれません。そうであれば，そのフィールドは v14 のプライマリーキーには相応しくない，ということになります。

もちろん，既存のフィールドをプライマリーキーに指定することが絶対に悪いわけではありません。ただ，そうするのであれば，そのフィールドは，**重複不可・NULL 不可・変更不可**でなければいけない，ということを忘れないでください。

おすすめなのは自動プライマリーキー

自動インクリメントは，シーケンス番号と同じように，1 番から始まって自動的に値がインクリメントされるスタンプ番号です。カウンターをリセットしない限り，重複する番号が振られることはありません。これに対し，**UUID**（バージョン 4）は，特別なアルゴリズムによって生成され，重複することのないとされるとてもおおきな整数です。UUID が 128 ビット（16 バイト）であるのに対し，倍長整数は 32 ビット（4 バイト）なので，フィールドサイズが 4 分の 1，インデックスや値の比較も倍長整数のほうが高速だろう，と考えるかもしれません。しかし，実際に 64 ビット版の 4D Server で比較してみると，UUID 型の検索は，倍長整数型の 1-2%程度，遅いだけであることが確認できます。つまり，UUID が占有するメモリのサイズは，間違いなく倍長整数の 4 倍ですが，計算速度はそれほど違わないのです。4D は，32 ビット整数であっても，内部的に 64 ビットの演算を実行しているため，倍長整数であっても，実際には 64 ビットの値を比較しています。逆に，UUID の一意性は，ほとんどの場合，前半の 64 ビットを比較したところで判明するものなので，結局のところ，両者のスピードに決定的な差はないということになります。倍長整数型のプライマリーキーは，複数のデータファイルを統合する状況が発生したときには，値が重複するので問題となります。それに対し，UUID はアルゴリズムの性格上，重複する値が発生する可能性はまずないので，将来，データベースを統合する必要が発生したときにも安心，というメリットがあります。ですから，**プライマリーキーには，特に理由がない限り，自動 UUID を選択する**ことが勧められています。

もっとも，自動インクリメント型のプライマリーキーのほうが望ましい場合もあります。前述したように，コンピューターの観点からすれば，UUID と 64 ビット整数の処理速度にさほどおおきな差はありませんが，人間にとって UUID は容易に記憶・メモ・伝達ができる識別子ではありません。ですから，プライマリーキーの値を直接ユーザーが扱うようなシステムでは，自動インクリメントを選択したほうが実際的かもしれません。

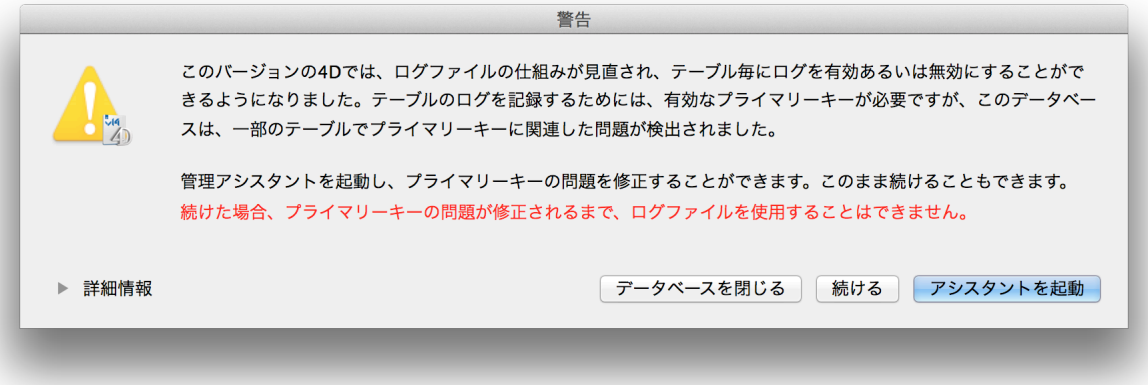
ジャーナルとプライマリーキー

プライマリーキーは，SQL REPLICATE/4DSYNC（複製と同期），REST（4D Mobile），さらにはジャーナル（ログファイル）とバックアップからの復元を支える重要な基盤です。テー

ブルにプライマリーキーが設定されていなければ，ジャーナルを作成することも，ジャーナルからデータベースを復元することもできません。たとえテーブルにプライマリーキーが設定されていたとしても，データに NULL や重複する値が含まれていれば，それを使用して作成されたジャーナルでは，論理的な整合性の保たれた，完全なデータベースを復元することができないことになります。そこで，v14 のストラクチャで v13 のデータファイルを初めて開いたときには，プライマリーキーのチェックが実行されます。チェックの内容は，次のとおりです。

- テーブルにプライマリーキーは設定されているか
- プライマリーキーに NULL は含まれていないか
- プライマリーキーに重複する値は登録されていないか

エラーが検出された場合，**インタープリター版のストラクチャであれば**，プライマリーキー管理ウィザード（アシスタント）を起動して問題を是正することができます。具体的には，そのテーブルをジャーナルの対象から除外するか，そのテーブルに適正なプライマリーキーを設定するか，どちらかを選択することになります。そのまま続けた場合，あるいはコンパイル版のストラクチャでデータファイルを開いたのであれば，それ以降，プライマリーキーに問題のないテーブルも含め，どのテーブルもジャーナルが記録されないことになります。



プライマリーキーは，信頼できるものでなければならないため，NULL が検出された段階でチェックは中止され，続けて重複のチェックは実行されません。コンパイルされたストラクチャの場合，この時点でどのテーブルもジャーナルが記録されないことになります。また，NULL の問題が解決できたとしても，重複の問題がみつかれば，やはりジャーナルは作成されません。ですから，プライマリーキーの NULL や重複は，現場でストラクチャを v14 にアップグレードするときにではなく，もっと早く解決しておかなければなりません。

準備はアップグレードの前に

プライマリーキーがほんとうに必要になるのは，v14 になってからですが，プライマリーキーそのものは，v11 以降であれば，いつでも作成することができます。v11 であれば，**SQL コマンド**，v12 以降であれば，**ストラクチャエディター**でプライマリーキーを作成することができます。ひとつ以上のフィールドで構成される一意識別子であれば，どんな値でもプライマリーキーにすることができますが，前述したように，おすすめなのは，自動 UUID または自動インクリメントです。いずれにしても，追加されたプライマリーキーは，v14 のジャーナルを見越したものであるので，既存のリレーションやレコード番号に基づいたアクセスを書き換える必要はありません。それらの「概念的なプライマリー」は，引き続き使用することができます。

運用データが別にあり，アップグレードの実施はしばらく後であるとしても，前もって自動タイプのプライマリーキーの追加に取りかかることが勧められています。そのようにしておけば，将来，v14 のストラクチャで既存のデータファイルを開いたときには，すべてのレコードに有効なプライマリーキーが作成されるようになっているからです。

4D では，いつでもデータファイルを差し替えることができますから，ストラクチャで「重複不可」「NULL 値を拒否」といったプロパティが有効にされていたとしても，あるいはプライマリーキーが設定されていたとしても，ほんとうにそれが一意識別子であるかどうかは，実際の値を調べなければ分かりません。v14 以降，**プライマリーキーは絶対的な存在**です。そのようなわけで，初めてデータファイルを v14 で開いたときには，すべてのプライマリーキーがチ

ェックされるようになっています。このとき，もしプライマリーキーが自動インクリメントまたは自動 UUID タイプだった場合，4D は自動的にプライマリーキーの問題を是正します。つまり，NULL や重複する値の代わりに有効な新しいプライマリーキーを保存します。しかし，これが行なわれるのは，プライマリーキーが自動インクリメントまたは自動 UUID タイプの場合だけです。自動タイプではないプライマリーキーは，自動的に是正されることはありませんので注意してください。その場合，チェックは実行されますが，問題のある値がひとつでも検出された時点で，全テーブルのジャーナルが記録されないことになります。

プライマリーキー管理ウィザード（アシスタント）は，v14 に組み込まれていますが，**v13 用のコンポーネント**としても用意されています。インストーラーには含まれていないので，下記の URL からダウンロードしてください。最低動作バージョンは 13.4 です。

（公式） ftp://ftp-public.4d.fr/Components/4Dv13/4Dv13_PK_Manager_Component

（日本サイト） http://ftp.4d-japan.com/PRODUCTS/4D/Current/4D_v13/pk_wizard/

結論

アップグレード対策は，いまから始めることができます。

- アップグレードを待たずに，現バージョンでプライマリーキーを追加する
- 既存のフィールドを使用するのではなく，新しい自動 UUID フィールドを追加する
- あとは，アップグレードのときにストラクチャを入れ替えるだけで良い

プライマリーキーに NULL や重複が含まれることは許されません。v14 では，これまでのバージョンにはなかった厳しいチェックが実行されるようになりました。つまり，データファイルを開いたとき，下記の条件に合致するフィールドがあれば，そのフィールドに NULL や重複が登録されていないかどうか，**すべてのレコードをチェック**するようになっています。

- データファイルに存在しないフィールドがストラクチャファイルに追加されている
- そのフィールドはプライマリーキーである

憶えておきたい点ですが，ストラクチャエディターや SQL でフィールドを追加したのであれば，こうしたことははじめから起こりません。NULL や重複が存在するフィールドをプライマリーキーにすることはできないからです。しかし，プライマリーキーの追加された後のストラクチャと追加される前のデータを開けば，そうしたことは起こり得ます。以前のバージョンでは，そのような既存レコードの該当フィールドは，実際に参照されるまで NULL 扱い（あるいはフィールドのタイプが変更された場合，型変換された以前の値）となるからです。新しく追加されたプライマリーキーの既存レコードで問題のある値が検出がされた場合，フィールドのタイプのよって次のようになります。

- **自動インクリメント・自動 UUID であれば**，自動的に値を訂正して保存する
- そうでなければ，プライマリーキー使用不可と判断し，**ジャーナリングを停止**する

したがって，事前に自動 UUID タイプのプライマリーキーが設定されていれば，たとえ現場のレコードに NULL や重複があったとしても，エラーが表示されたり，ジャーナリングが止められたりすることはありません。そのようなわけで，データベースにプライマリーキーのようなもの（重複不可フィールド・リレーションの 1 フィールド・その他，アプリケーションで定めた ID フィールド）がたとえ存在するとしても，新しい自動 UUID タイプのプライマリーキーを追加したほうが無難であるということができます。

補足: 複製とミラーリング

v12 では，SQL REPLICATE/4DSYNC（同期と複製）ができるようになりました。これは，テーブルに加えられたアクション（レコードの追加・更新・削除）を相手側でも再現することにより，データベース間でふたつのテーブルがまったく同じ内容になるようにするというものです。アクションを再現するという点で，複製とミラーリングは似ていますが，複製とミラーリングを混同してはなりません。**複製は，テーブル単位のコピーに過ぎない**からです。複製は，ジャーナルを統合するのとは違い，トランザクションレベルの論理的整合性を再現することができません。

それは，次のような例で説明することができます。注文書は，注文書テーブルおよび明細テーブルに保存されています。つまり，1 対 N のリレーショナルデータベースです。はじめに，注文書データをコピーし，続けて明細データをコピーしたとしましょう。注文書をコピーしている間に注文書が追加された場合，後から登録された注文書はコピーされませんが，明細だけはコピーされます。最初に明細をコピーしたとしても同じことです。今度は，明細のない注文書が作成されるかもしれません。もちろん，次回の同期では，不足分のデータもコピーされるので，いずれは完全なデータがコピーされる，と考えるかもしれません。しかし，ミラーリング（あるいはバックアップ）は，不測の事態，すなわちクラッシュに備えるものであるはずです。クラッシュは，発生のタイミングを選びません。いつ起きるか分からないのです。データベースがクラッシュした場合，明細のない注文書，注文書のない明細といった断片的なデータしか残されていないのでは困ります。複製（同期）は，**トランザクションレベルの不可分なオペレーションが再現されることを保証するものではない**ので，リレーショナルデータベースのコピーには勧められません。もちろん，まったくオペレーションのない時間帯であれば，問題ありませんが，ビジネスアプリケーションでそれができる状況は，大抵，とても限られています。

補足: BLOB とジャーナル

BLOB は，レコードの中，レコードの外（しかしデータファイルの中），データファイルの外（パスは 4D が管理，データフォルダー/ExternalData フォルダーの中）データファイルの外（パスはデベロッパーが管理）と，さまざまな保存オプションが用意されています。いずれの場合も，BLOB（ピクチャ・テキスト）が追加・更新されたのでない限り，ジャーナルにはデータが記録されません。しかし，保存オプションにより，若干の違いもあります。

レコードの外に保存される BLOB には，プライマリーキーおよびフィールド番号の参照が記録されています。したがって，ジャーナルにも，その BLOB が追加・更新されたのであれば，プライマリーキーおよびフィールド番号の参照がデータとともに記録されます。データファイルの外に保存される BLOB で，パスは 4D が管理しているのであれば，代わりにパスとデータがジャーナルに記録されます。ですから，どちらの場合でも，ジャーナルから BLOB データを復元することができます。データファイルの外に保存される BLOB で，パスはデベロッパーが管理しているのであれば，フィールド同様，ジャーナルにはパスが保存されています。そのため，ジャーナルからデータを復元することができませんが，本来の場所にファイルが残っていれば，レコードは元どおりになります。

補足: セレクションとプライマリーキー

v11 以降，4D のセレクションは，内部的にいくつかの形態を取り得る，ダイナミックなオブジェクトです。ALL RECORD や QUERY などのコマンドでセレクションを作成した場合，そのセレクションは並び替えられていないので，内部的には**ビットテーブル**が使用されます。1 レコードに 1 ビットが対応していますが，まとまった範囲がすべて同じ値であれば，テーブルが圧縮されるため，実際のサイズは，レコード数×ビットよりもずっと小さなものになるかもしれません。その後，ORDER BY などでセレクションを並び替えると，今度は**レコード番号の配列**が作られます。いずれにしても，4D のセレクションは，レコード番号に基づいているということです。

レコード番号に基づいたセレクションは，アクセス速度が優れています。しかし，問題がないわけではありません。**レコード番号は再利用される**ため，もし，セレクションを作成した後，誰か（別プロセス）がそのセレクション内のレコードを削除し，同じテーブルに新しいレコードを追加するようなことがあれば，そのレコードには削除されたものと同じ番号が振られ，セレクション内のレコードが当初のものとは違うものになるかもしれません。こうした事態を避けるため，多くのデベロッパーは，レコードを削除する代わりに「削除フラグ」を立てたり，一度削除されたレコードが一定の期間は再利用されないように，対策を立てています。しかし，別の方法もあります。それは，レコード番号の代わりに，**プライマリーキーでセレクションを作成**する，というやり方です。実際，Wakanda では，レコード（エンティティと呼びます）をプライマリーキーで特定しています。レコード番号は，ある瞬間のレコードを特定することができますが，時間を超えてレコードが同一であることを保証するものではありません。一方，プライマリーキーが一致するエンティティは，時間の経過と関係なく，同じエンティティであるとみなすことができます。

プライマリーキーでセレクションを作成した場合，今度はスピードが犠牲となります。セレクション内のレコードを参照するためには，その都度，プライマリーキーのインデックス検索を実行しなければならないからです。とはいえ，データベースの堅牢性という観点から，そうしたアイデアを 4D で使用できるようにすることも検討されています。

補足: ジャーナルに記録されなかったトランザクション

確定またはキャンセルされなかったトランザクションの途中経過が記録されないことに不安を感じる人もいるかもしれません。たとえば，原因不明なトランザクション未確定の理由を究明するためにその情報が欲しいようなケースです。これは，新しいジャーナルファイルのデメリットのひとつです。そうした問題に対応するため，また他のデバッグ困難な問題に対応するため，現在，特別なデバッグ情報（ジャーナルではない）をオブジェクト型を返す新しいコマンドが検討されています。そのオブジェクトには，データベースのキャッシュやインデックスに対するあらゆるアクセスが詳細に記録されています。（14R2: Get database measures）