e-ビジネスを支える統合データベース・システムの実現

Realization of integrated database systems for supporting e-business

日本アイ・ビー・エム システムズ・エンジニアリング株式会社
第二システム・センター
主任アーキテクト
IBM Certified DB2 Database Administrator
ORACLE MASTER GOLD
土井 智成

Tomonari Doi
IBM Certified DB2 Database Administrator
ORACLE MASTER GOLD
IT Architect
IT System Center No.2
IBM Japan Systems Engineering Co., Ltd.

e-ビジネスおけるエンタープライズ・システムは、多様なコンポーネントが有機的に結合したアーキテクチャー構造を持ちます。現実のSIが中心の設計・開発は、短納期で高品質なシステム構築が求められます。筆者は、ある金融機関のe-ビジネス業務開発に対応して、勘定系と情報系のデータベース・システムを同時に開発しました。この統合システムは、勘定系システムであるB to Bのインターネット・バンキング・システムと、情報系であるCRM（Customer Relationship Management）の観点から法人顧客の利用実績を分析する汎用検索システムで構成されます。また、359日24時間のオンライン運用を目標とした統合データベース・システムであり、UDB/EEE（DB2 Universal Database™ / Extended Enterprise Edition）の新機能とESS（Enterprise Storage Server™）のフラッシュ・コピーを使用した日本初のプロダクション・システム事例でもあります。本論文では、その経験を基に、データベースの観点から、e-ビジネス統合データベース・システムの業務開発と基盤開発で考慮すべき技術的なポイントを整理します。

Enterprise systems in e-business possess architecture structures with diverse organically linked components. Design and development centering on real SI require high-quality system structures with short delivery deadlines. In my own case, I worked simultaneously on the development of accounting and information database systems in response to the development of e-business operations for a financial institution. This integrated system consists of an Internet banking system of the "B to B" accounting system type and an all-purpose search system which analyzes use by corporate clients from the standpoint of the CRM (Customer Relationship Management) information system. It is an integrated database system whose aim is to achieve on-line operation 24 hours a day for 359 days a year. In addition, it is Japan's first ever production system with new functions such as UDB/EEE (DB2 Universal Database™, Extended Enterprise Edition) and using the flash copy function of ESS (Enterprise Storage Server™). In this paper, on the basis of the experience I gained through work in this area and from the standpoint of databases, I have summarized the main technical points that need to be taken into consideration in connection with business development and infrastructural development of e-business integrated database systems.

1. はじめに

Webを中心としたエンタープライズ・システム構築では、旧来はソフトウェア開発手法としてDFD（Data Flow Diagram）やHIPO（Hierarchy, Plus Input, Process, Output：階層的入力処理出力記述手法）の構造化分析手法を使っていました。しかし、これでは「短納期」と「高品質」を満足するのは不可能です。そこで、最近ではパターンの利用と部品の再利用による短期開発を目的として、UML（Unified Modeling Language：統一モデリング言語）ベースのオブジェクト手法開発が提案されています。

これは、UMLがe-ビジネスの標準開発言語であり、オブジェクト指向言語であるJava™との親和性が高いことと、IBMの開発統合環境であるWSAD（WebSphere® Studio Application Developer）およびRatinal XDEなどがサポートされ、上流工程から下流工程まで連続的かつ効率的に開発・保守できる環境が整備されたことが一番の理由でしょう。

筆者は、ある金融機関のe-ビジネス業務開発に対応して、勘定系と情報系システムのデータベースを統合運用型で同時に開発しました。このシステムは、勘定系システムであるB to Bを対象にしたインターネット・バンキング・システムと、情報系であるCRM（Customer Relationship Management）の観点から、法人顧客の利用実績を分析する汎用検索システムで構成されます。

データベース開発は、データを直接利用する業務開発のみならず、業務システムのパフォーマンスや可用性などの基盤設計についても大きく影響します。このため、データベースの運用を含めて基本設計段階で業務チームへのSQLガイドの実施、システム設計や運用設計のポイントのレビュー、検証スケジュールをプロジェクト計画として立案することが重要であると考えています。

2. e-ビジネス・データベース業務開発の設計のポイント

2.1. コネクション・プール機能

データベースへの接続は、データベース・サーバーへの負荷が非常に高い処理として知られています。通常のSELECT文などのSQL処理時間は数ミリ秒ですが、接続処理時間は約1秒です。これは、各接続要求に対して1エージェント（UNIX®ではプロセス、PCではスレッド）を起動しているためです。

こうした負荷を抑えるために、WAS（WebSphere Application Server）ではコネクション・プールの機能を実装しています。ただし、データベース接続は、接続先と認証が一致する場合に再利用され、一致しなければ新規に作成されます。つまり、データベース名が同一でも接続ユーザー名が異なるとユーザー間で接続の再利用は行われません。また、データ・ソースを使用して取得したコネクション・オブジェクトは、使用後にclose( )メソッドによってコネクション・プールに返還され、ほかの要求から再利用が可能になります。

接続が終了すると、アプリケーション内で必ずclose( )を行う必要があります。特に、WASの稼働中はデータベース・サーバーに障害が発生すると、データベース接続が回復しても、コネクション・プール内の接続は使用できないので注意が必要です。このため、データベースの回復後、最初のユーザーの初回アクセスから正常に稼働させるには、図1のように、StaleConnectionExceptionをキャッチして、コネクションをclose( )した後、コネクション取得をリトライするようなロジックが必要となります。

WASは、データベース・アクセスを高速化するために、一度に実行されたSQL文をキャッシュするプリペアード・ステートメント・キャッシュ機能をサポートしています［参考文献1］。この機能は、作成されたステートメント・ハンドルを、プログラムから解放要求があっても実際には解放せずに保持します。そして、次回も同じSQL文が作成された場合は、保持していたステートメント・ハンドルを再利用します。

図2に、コーディング例を示します。

パラメーター・マーカーを使用したSQL文を使用すれば、OLTP業務ような、複数のユーザーが単純な同一SQL文を繰り

```
try {
    for (int i=2; i>0; i--) {
        try {
            conn = dataSource.getConnection(user, password);
                ・・・
                (connを使用したSQL処理)
                ・・・
            i = 0;                          // リトライのためのカウンターを0にする
        } catch (StaleConnectionException scex) {
                                            // Stale Connection を catch
            if (conn != null) {
                try {
                    conn.close(); // Stale Connection を close
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

図1. コネクション・プール対応のコーディング例

```
PreparedStatement psmt = con.preparedStatement("INSERT INTO
sample VALUE( ?,  ?, ? ) ;
```

図2. プリペアード・ステートメント対応のコーディング例

返し実行する環境では、有効な機能です。つまり、データベース・サーバーの観点から見ると、データベース・サーバーのSQL解析機能をまったく使用せず、あたかも同一プログラム中の以前実行したSQLを再実行するイメージとなります。実行時間を平均で30%前後にまで短縮できたという報告もあります[参考文献2]。

しかしながら、プリペアード・ステートメント・キャッシュ機能は、十分にテストし、注意して使用する必要があります。クライアント/サーバー型のように通常のタイプの場合には、一つの接続で一つのアプリケーションの実行が完結します。ところが、WASのコネクション・プーリングとプリペアード・ステートメント・キャッシュを利用すると、一つの接続を複数のアプリケーションが再利用するため、WAS側でデータ・ソース単位で指定するパラメーターであるStatementCasheSizeの大きさまで、保持されるSQL文が増加する可能性があります。

DB2 Universal Database™ (以下、UDB)では、このステートメント・ハンドルをアプリケーション・ヒープに格納します。この領域は、UDBエージェント用のメモリーとして用意され、アクセス・パスの実行領域として使用されます。この領域がこれ以上割り当て不能になると、CLI0129Eのエラーが発生します。

例えば、WASチューニング・ガイドで一般的に紹介されている「StatementCasheSize=1000」の場合を考えると、一つの接続に対して、業務で使用するSQL文の全体のエントリー数、または1,000エントリーのSQL文に対応できるアプリケーション・ヒープ・サイズを用意する必要があります。なお、参考までに、IBMテクニカル・フラッシュでは、「StatementCasheSize=1000」に対して4,096のアプリケーション・ヒープ・サイズが必要といった報告もあります。

もちろん、この機能を利用しなくても、アクセス・パスを共有メモリーに保管して、同一のSQL文の場合にはこれを再利用します。この領域を、UDBではパッケージ・キャッシュと呼び、その大きさを指定するパラメーターが用意されています。

私たちは、性能要件として応答時間に問題がないことを確認した上で、この方式を採用しました。

2.2. 排他制御

JDBC(Java Database Connectivity)では、SQLの実行結果を1レコードずつ処理するため、ステートメント・オブジェクトのnextメソッドでアンサー・セットを処理します。この場合、SQLを処理するために、内部的には自動的にカーソルがオープンされます。ところが、SQL文のSelect For Updateを使ってUロックでレコードを読み、ほかのユーザーによるデータ変更を禁止し、Update Where Current Of c1にて結果を更新するような排他制御を行う場合には、カーソルを明示的に指定する必要があります。ところが、JDBCの場合は、組み込みSQLと違ってDeclare Cursorを使用してカーソルを宣言しないため、カーソルが指定可能であることはあまり知られていません。このため、多くは排他制御を独自に工夫しています。

例えば、レコードに更新フラグなどを用意し、1トランザクション内でまずUpdate処理を行い、レコードに対してXロックをかけて排他制御を行い、次に読み取りを実施します。そして、最終的には業務内容データのUpdate処理を行います。

または、接続ごとにsetTransactionIsolationメソッドにより、分離レベルとしてはUDBのRS(Read Stability)と等価なTRANSACTION_REPETABLE_READを指定します。これにより、読み取りレコードに対してほかのユーザーの更新を防ぎます。また、文単位に分離レベルを指定できる場合は、Select With RSを使用する方法もあります。

いずれも、対象レコード全体がXロックによって読み取り不可となるため、トランザクションの並行性が失われ、パフォーマンスに影響するとともに、デッドロックが発生しやすくなります(図3)。

なお、WASによってUDBに対して複数SQL文を1トランザクションで制御するには、デフォルトでは1文単位に自動的にコミットする設定のため、必ずsetAutoCommitメソッドで自動コミットを解除します。また、接続を再利用すると、トランザクション範囲が次のプログラムとまたがる可能性があるため、SQL処理として必ずCommitまたはRollbackの明示的発行とcloseメソッ

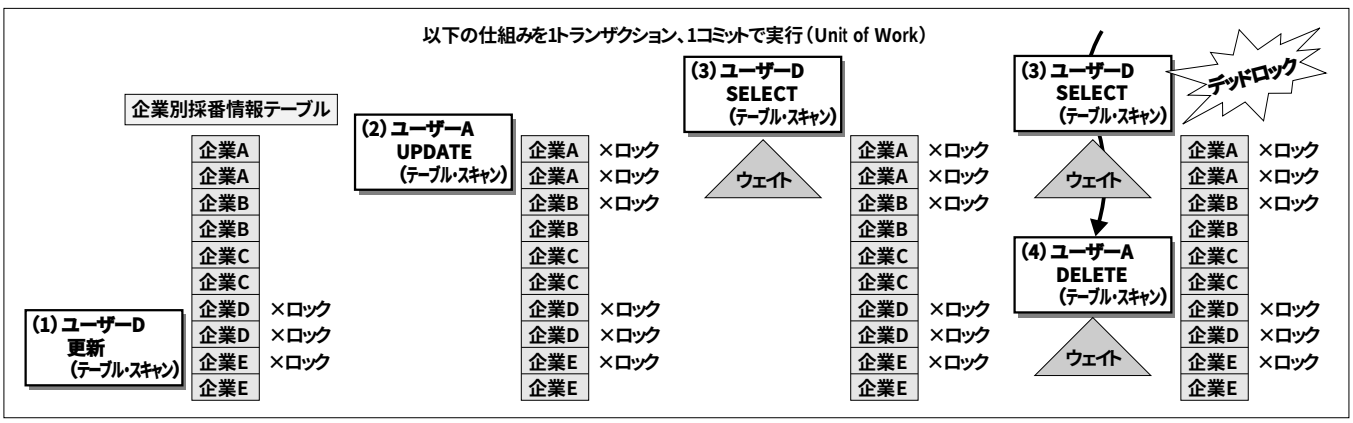

図3. デッドロックの発生例

ドの指定を行います。

2.3. バッチ

オンライン処理は比較的少量のデータを対象に応答性を重視しますが、バッチ処理は大量のデータを対象に集約化処理や洗い替え処理を行います。C言語による静的SQLを使用した場合では、JDBCの約10倍のパフォーマンス結果が報告されています[参考文献3]。

また、e-ビジネス環境におけるシステム構成では、拡張性や機能分割の観点から、アプリケーション・サーバーとデータベース・サーバーのコンポーネントを、異なる物理ノードに配置することが一般的です。このため、仮にアプリケーション・サーバー上にバッチ処理のプログラム本体が存在すると、アプリケーション・サーバーとデータベース・サーバー間で大量のデータが通信されるため、処理効率が悪くなります。

よって、バッチ処理については、データベース・サーバーにその機能を配置し、C言語による静的SQLを使用することが望まれます。また、大量のデータを対象に複数のテーブルをJoin（結合）し、その結果を複数の段階にわたってさらに別表とJoinするようなネスト処理やソート処理を行い、最終的に表データを丸ごと洗替するような処理の場合には、SQLだけで実行するよりも、各テーブルのexportファイルをプログラムによって照合し、集約処理や洗替処理を実施する方がパフォーマンスが良いでしょう。

オンライン・バッチ機能のシステム設計例を、図5に示します。

2.4. SQL一覧ファイル

業務システム全体のSQLについては、一覧ファイル形式で外出しで管理する仕組みをJ2EE上でデータベース・アクセス部品として用意すると便利です。また、SQL文ごとにSQLID付与し、テーブル名やDMLの機能である更新・削除・挿入についての略称を付けて管理するとよいでしょう。トランザクションの実行時に、タイムスタンプ、SQLID、実行結果を同時にデータベースのトランザクション・ログとして出力する仕組みにすれば、SQLトランザクションをモニターすることが可能です。トランザクション実行状況の確認、テーブルへのアクセス監査、ロックの問題発生時の調査などに利用できます。

```
con.setAutoCommit(false);
stmt1.setCursorName("c1")
result=stmt1.executeQuery(Select For Update)
rs.next()
row stmt2.executeUpdate(Update …… Where Current Of c1)
```

図4. 排他制御対応のコーディング例

```
#!/bin/ksh
SQL_FILE=$1
SQL_ID=`echo $SQL_FILE ¦ awk -F . '{print $1}`

db2 Connect To b2bdb
cd /udbinst1/sqllib/misc
db2 -tf EXPLAIN.DDL
db2 Commit
cd -
db2 Set Current Explain Mode Explain
db2 -tvf $SQL_FILE
db2 Commit
db2exfmt -d b2bdb -g TIC -w -1 -n % -s % -# 0 -o
db2exfmt.${SQL_ID}.txt
db2 Drop Table EXPLAIN_INSTANCE
db2 Drop Table EXPLAIN_STATEMENT
db2 Drop Table EXPLAIN_ARGUMENT
db2 Drop Table EXPLAIN_OBJECT
db2 Drop Table EXPLAIN_OPERATOR
db2 Drop Table EXPLAIN_PREDICATE
db2 Drop Table EXPLAIN_STREAM
db2 Drop Table ADVISE_INDEX
db2 Drop Table ADVISE_WORKLOAD
db2 Terminate
```

図6. アクセス・パス取得シェル例

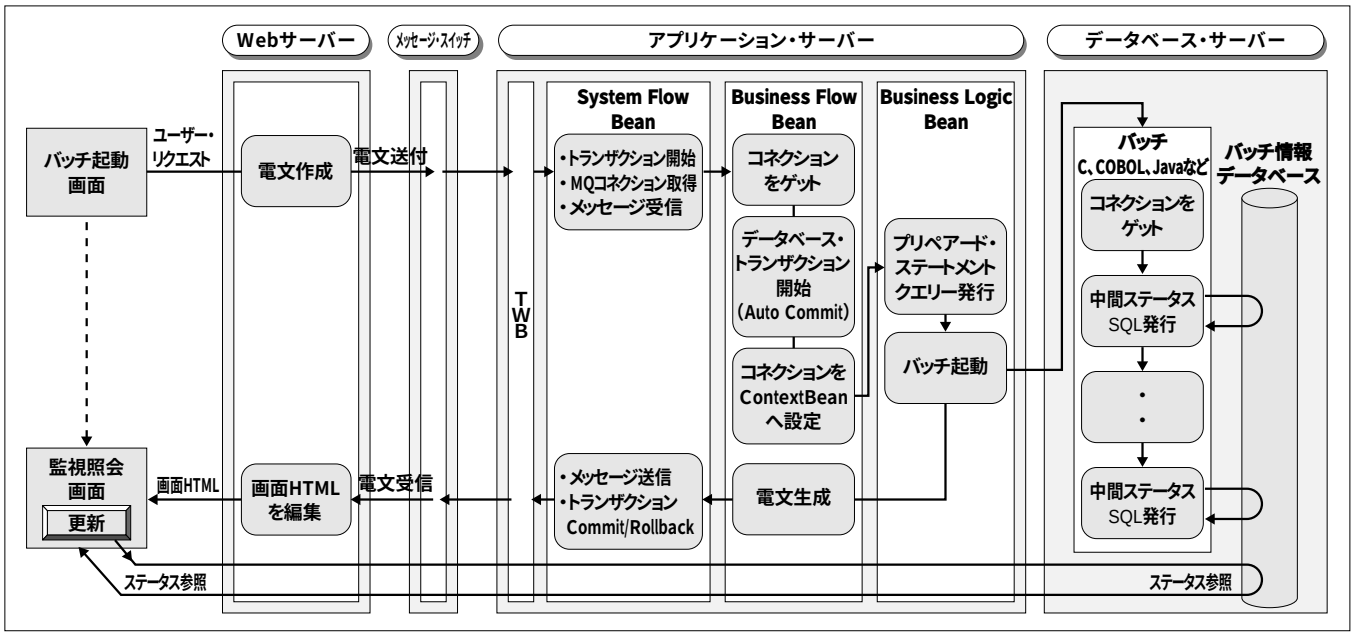

図5. オンライン・バッチ機能のシステム設計例

表1. SQLアクセス・パス検証例　　タイプの説明　S：Select　I：Insert　D：Delete　U：Update　J：Join　LOJ：Left Outer Join　ROJ：Right Outer Join　Sub：Sub Query

項番	SQL ID	タイプ	テーブル	インデックス使用	検証結果	備考
12	COBNKMST002S	S	BNK_MST	BNK_MST_PK	OK	UNION条件使用
		Sub	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
		S	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
		Sub	BNK_MST	BNK_MST_PK	OK	
13	COPTS019S	S	PAY_TRN_STS	PAY_TRN_STS_IX5	OK	
14	COSAM008S	S	SVC_ACNT_MST	SVC_ACNT_MST_PK	OK	
		S	BNK_MST	BNK_MST_PK	OK	
		S	ACNT_TYP_MST	×	OK	テーブル・スキャンであるがテーブルの件数が科目コードの12件のため、問題なし。
		Sub	BNK_MST	BNK_MST_PK	OK	
15	COSTM002S	S	SVC_TIM_MST	SVC_TIM_MST_PK	OK	
16	COTPI000S	S	TRN_PRC_INF	TRN_PRC_INF_IX3	OK	
		S	TRN_INF_MST	TRN_INF_MST_IX1	OK	
17	KOBSI001U	U	BNK_SEC_INF	BNK_SEC_INF_PK	OK	
18	KOBUM001U	U	BNK_USR_MST	BNK_USR_MST_PK	OK	
19	KOJZN103S	S	ADV_PAY_MST	ADV_PAY_MST_IX1	OK	
20	KOKNM305U	U	CRP_INF	CRP_INF_PK	OK	
21	KOSHO402S	S	TRN_PRC_INF	TRN_PRC_INF_IX3	OK	
		S	TRN_INF_MST	TRN_INF_MST_IX1	OK	

検証結果
凡例
OK　：よしと判断できるもの。
追加：インデックスを追加した方がよいと思われるもの。
SQL改善：SQLを変更していただく。

[db2cli.iniの例]

```
; Comment lines start with a semi-colon.

[COMMON]
trace=1
tracepathname=/LOG
traceflush=1
tracetimestamp=1
jdbctrace=1
jdbctracepathname=/LOG
jdbctraceflush=1
```

図7. JDBCトレース取得方法

業務プログラムで使用しているSQLが論理的に確定した段階で、SQLの最適化と同時にインデックス設計の妥当性を検証するために、必ずアクセス・パスの検証し、結果を報告書としてまとめます。特にSQL一覧ファイルで管理した場合は、SQL全体のアクセス・パス情報を図6に示すコマンドを使えば容易に取得できます。

2.5. db2cli.ini

UDBは、プログラム・インターフェースとして、「EXEC SQL～」で始まる組み込みSQLとWindows® ODBC（Open Database Connectivity）アプリケーションで使用されるCLI（Call Level Interface）を用意しています。JDBCでUDBにアクセスする場合は、ODBCの場合と同様に、内部的にはCLIに変換されます。従って、UDBから見るとWASのJDBCクライアントはCLIクライアントと等価なので、WAS側のdb2cli.iniにパラメーターを設定することにより、JDBCクライアントのカスタマイズが可能です。WASからのデータベース・アクセスに問題が生じた場合、WAS－データベース間の問題解析を行うには、図7の例のように設定してJDBCトレースを取得します。

3. 勘定系と情報系統合データベース・システム基盤の設計のポイント

3.1. 拡張性

このシステムは、法人が対象の高額資金移動／振込／振替をインターネット上で実現し、事務処理を効率化することを目的としたB to Bのバンキング・システムです。基本設計書作成時のシステム化要件のポイントは、正月とゴールデンウィークを除いた24時間連続サービスが運用可能であり、高信頼性と拡張性を兼ね備えることです。特に、社外の不特定のユーザーが使用するため、インターネット・システム経由でのシステムへの侵入を許さないセキュリティー対策が重要です。また、それと同時に急激なアクセスの増加や集中に対応する必要があります。システム化の要件をまとめると、次のようになります。

- 24時間359日のための運用・縮退・障害対策への十分な配慮
- トランザクション増加に対する容易な拡張
- オンラインでの保守・拡張を可能な限り実現
- 高セキュリティーと可監視性の実現
- 論理3層アーキテクチャーによる機能独立性の維持
- 大量オンライン、バッチに対応できる物理的機能分担
- キャパシティー計画（CP）用システム・データの監視と取得

今回のB to Bシステム全体の構成を、図8に示します。

図からも分かる通り、拡張性については、同じ機能を持つ複数サーバーによるクラスター構成で対応します。しかし、トランザクション・データは、対象となるレコードの整合性が求められる

ため、物理的に1カ所で管理することが望まれます。このため、データベース・サーバーの構成はレコードを水平分割し、複数データベース・サーバーに分散配置する方法で対応します。

例えば、「顧客A」のレコードについては「データベース・サーバー1」、「顧客B」のレコードについては「データベース・サーバー2」で対応します。しかし、全体のデータを管理・分析するには、「データベース・サーバー1」に格納される「データベース1」と「データベース・サーバー2」に格納される「データベース2」を別々に対象にする必要があり、効率が悪くなります。

また、データベース・サーバーを追加するには、既にキャパシティー・オーバー寸前である場合は、処理能力を均等にするために、既存の大量のデータを分割して既存データベース・サーバーと新規データベース・サーバー間で均等に再配分する必要があります。

今回のシステムでは、UDB/EEE（Extend Enterprise Edision）を採用し、分割キーとして「契約者番号」と「企業内利用者ID」を指定し、自動的に各ノードに分散配置しながら、一つのデータベース・システムとして管理する方式を採用しました。また、将来の最大物理ノード数の規模を想定し、複数の物理ノード構成にすると同時に、物理ノード内に事前に複数論理ノードを用意し、将来の拡張性に対応することにしました。これにより、サーバーの追加に生じるデータの再配分を防止し、短期間でサーバーの追加に対応が可能になります。

基本的には、1台の物理ノード当たり三つの論理ノードで構成し、全体は4台の物理ノードの構成としました。それぞれの物理ノードは、最大で24基のCPUをサポートするpSeries™のS85を使用し、当初は6基のCPUとしました。これにより、キャパシティー不足が発生した場合に、物理ノードにCPUを追加するだけの容易な対応で、最大で約3倍の拡張が可能です。また、物理サーバーの追加でも、論理ノードごとにボリューム・グループの割り当てなどでディスク設計を工夫すれば、比較的容易に対応できます（図9）[参考文献4]。

3.2. 高可用性システム設計

(1)物理ノード障害対策

ミッション・クリティカルなシステムの設計のポイントは、障害を防止するために設計を精緻化することはもちろん、障害発生した場合のリカバリー対策も重要です。

高可用性対策として、pSeriesのプラットフォームでは主にHACMP（High Availability Cluster Multi-Processing）を使用

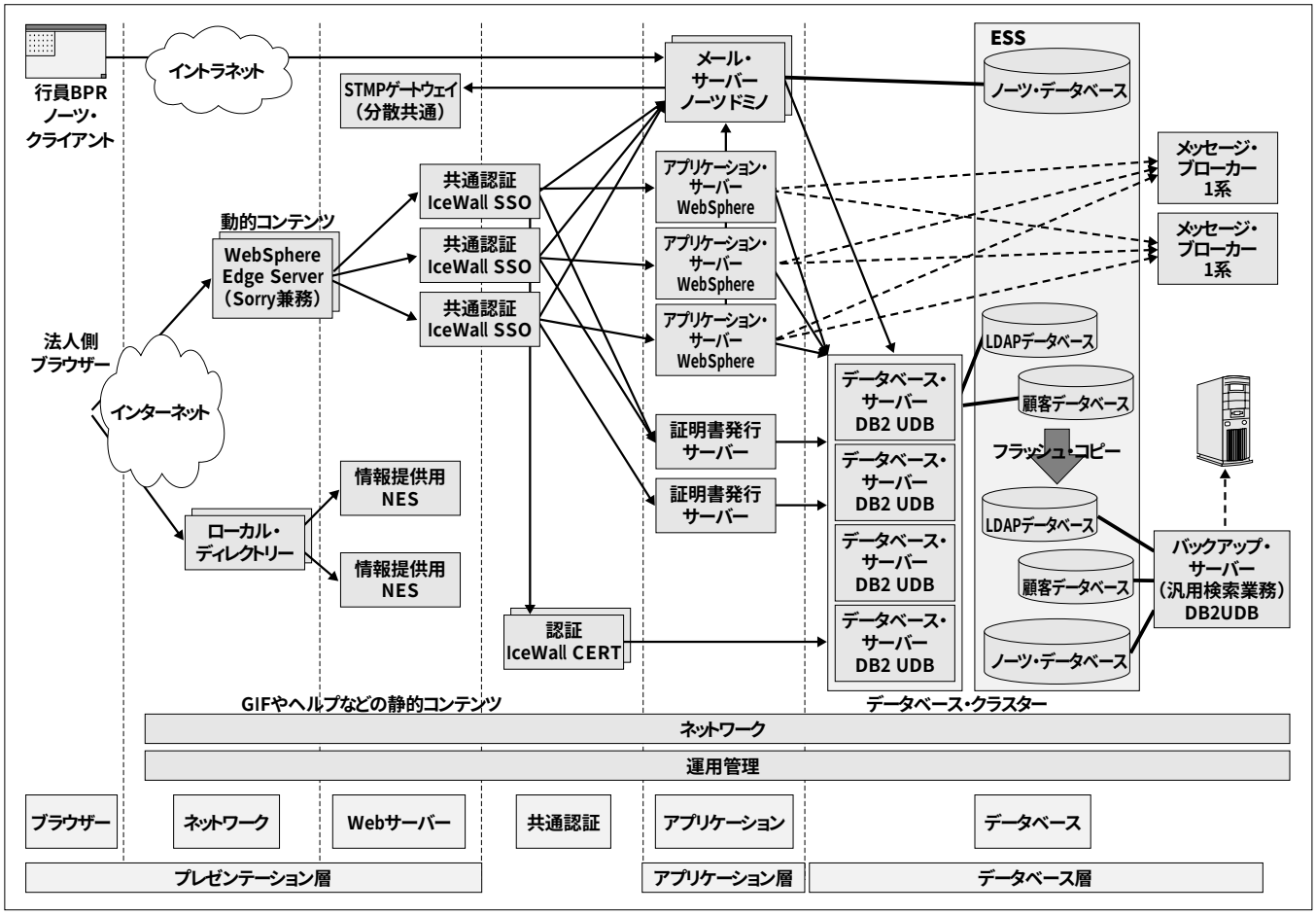

図8. システム構成図

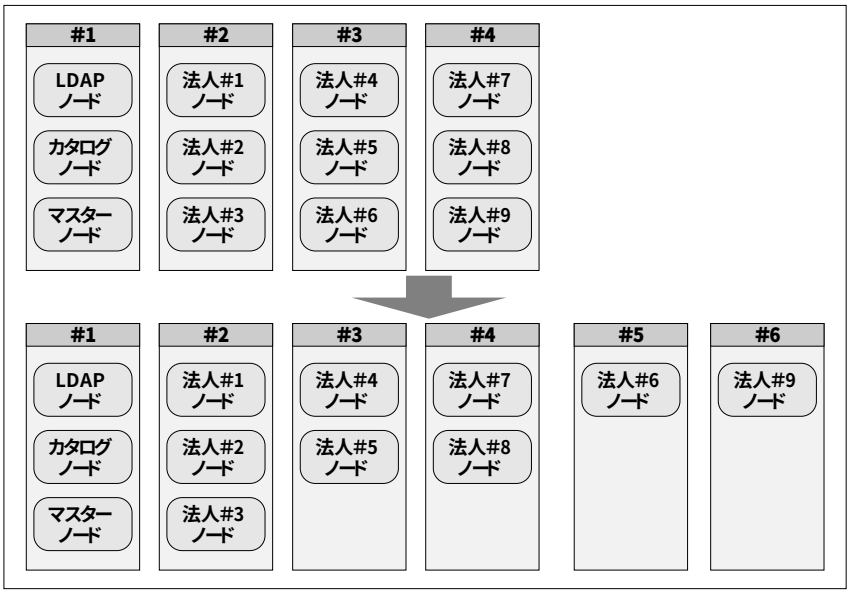

図9. 拡張性に対応したデータベース・システム設計

します。これにより、クラスター内の障害を検知し、IPアドレスやディスク資源、アプリケーション・サブシステムの引き継ぎを行います。今回は、オンライン復旧時間の目標値を3分としました。従来は、pSeriesプラットフォームのリソース引き継ぎ時間は、一般に30分をめどに考えられています。このうち、ボリューム・グループやファイル・システムを順番にチェックするディスクの引き継ぎが大部分を占めます。

筆者は、別プロジェクトでこれを短縮するために、HACMPをカスタマイズして並列にリソースの引き継ぎを設定しました。結果的に、1ノード当たりESS（Enterprise Storage Server™）200Gバイト規模で5分に短縮できました。この場合、ボリューム・グループやファイル・システムのマウントが並列に実行されるため、マウントの順番に依存しない独立のファイル・システムを定義します。今回の場合は、クラスター内でロジカル・ボリュームの共有を可能にするHACMP CRM（Concurrent Resource Manager）を使用して、さらに引き継ぎ時間を短縮しました。1ノード当たりESS100Gバイト規模の物理ノードで、引き継ぎ時間は1分20秒です。これに、UDBのリスタートおよびデータベースのクラッシュ・リカバリーを行うデータベースの回復処理時間が1分30秒なので、合計で2分50秒で3分以内にオンライン復旧を達成することができました。

図10にその概念図を示します。

このため、設定ファイル、問題判別ログ、アーカイブ・ログを除くすべてのUDBの物理資源について、ロジカル・ボリュームを使用しました。カタログ、一時表、ユーザーのすべてのテーブル・スペースは、ロジカル・ボリュームを使用したDMS（Database Managed Space）で定義し、トランザクション・ログ（newlogpath）もロジカル・ボリュームで定義しました。また、クラッシュ・リカバリー時間の短縮のために、ログ・ファイルのサイズ（logfilsz）は16Mバイト、チェックポイントの実施タイミング（softmax）を、ファイル・サイズの10%に設定しました。

なお、メディア障害回復時に使用するRollforwardコマンドについては注意が必要です。ログをロジカル・ボリューム定義で使用する場合には、User Exitが対応しないため、Rollforwardコマンドで OverflowPathでアーカイブ・パスを明示的に指定し実行しなければなりません。

また、WASからUDBの論理ノードを指定した接続は、縮退時には無効のため使用しません。

(2) プロセス障害対策

障害が発生した物理ノード内のすべての論理ノードが影響します。このため、プロセス監視機能をサポートするTivoli® DM（Distributed Monitor）で1分間隔にプロセスの状況を監視し、障害時には再起動を含む回復処理を実行する設計にしました。

ただし、検知間隔が短いため、回復処理の2重起動防止機能を組み込みました。また、このプロセス監視は、クラスターの運用を念頭に設計する必要があります。統合テスト段階での障害テストや計画停止時に、プロセス監視の誤動作によるゾンビ・プロセスの発生のため、HACMPからのUDB起動に失敗また遅延などの問題が生じる場合があります。本プロジェクトでは、運用の観点から保守や障害復旧時のHACMPのコマンドでのリソース引き継ぎや切り戻しを実行する場合は、プロセス監視を停止するよう運用ガイドに明記しました。

3.3. 24時間オンライン運用システムの実現

(1) フラッシュ・コピー機能

従来、銀行などの金融機関のシステムは勘定系と情報系が独立して運用されていますが、今回は勘定系の24時間オンライン運用を目標に、統合システムとして開発しました。UDB/EEE V7.2

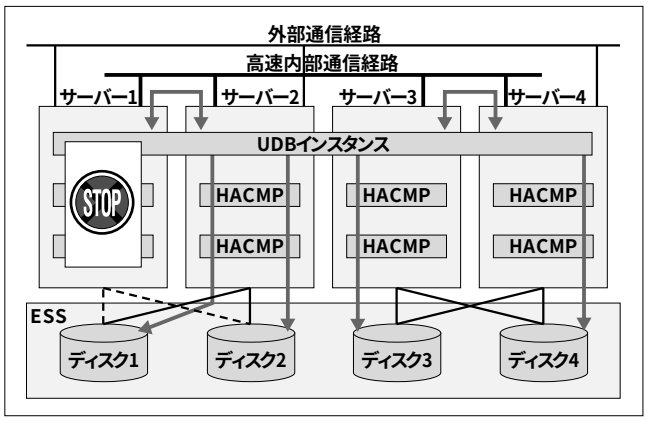

図10. HACMP CRMを使用した高可用性システム設計

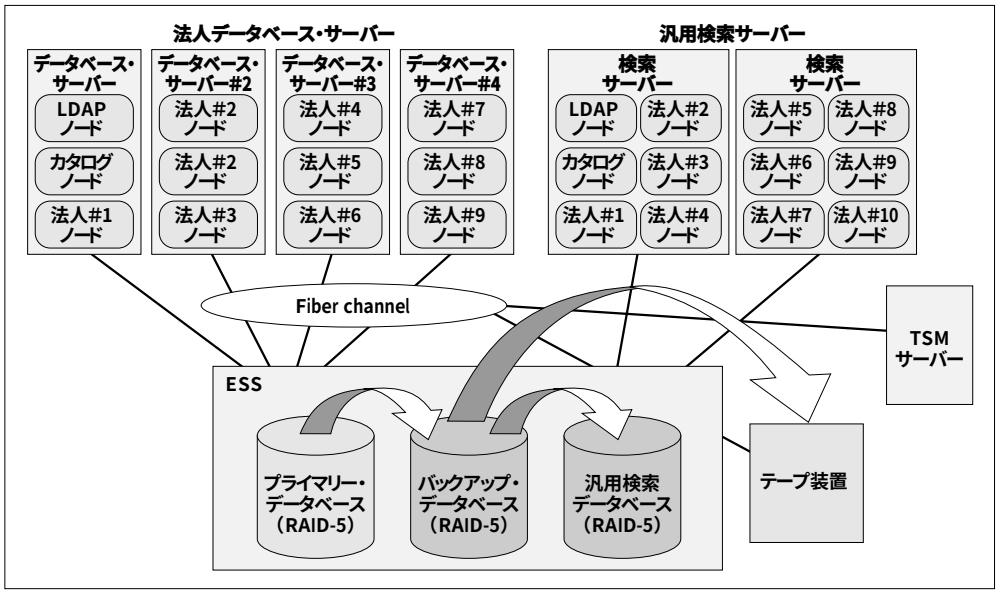

図11. 統合データベース・システムの実現

の新機能と、ESSのフラッシュ・コピー機能を利用しています。

勘定系および情報系の、それぞれのESSのLUNを含めたディスク設計を精緻化しておけば、次のような簡易な手順で実現可能です。

- UDBよりWrite Suspendコマンドを発行する。これにより、データベースの整合性が維持される（読み取りは可能ですが、書き込み要求はwait状態になる）。
- ESSのFlash Copyコマンドを発行する。
- ESSのFlash Copyコマンド終了後、UDBよりデータベースのWrite Susupendを解除する。
- TSMでバックアップ・データベースからテープ装置へバックアップを取得する。
- 汎用検索サーバー側で利用するために、relocateを行う。

relocate終了後、汎用検索サーバー側で、検索・分析に必要なインデックスを付加します。

100Gバイト規模の物理ノード当たりのESS フラッシュ・コピーによる書き込み中断時間は21秒でした。ただし、この実行は、UDBの観点からロック・タイムアウトへの考慮が必要です。また、オンライン処理の負荷が高い場合には注意が必要です。

書き込みの中断が開始されると、それ以前に開始されたトランザクションはCommit発行時に実行が保留になります。このため、書き込み中断時にトランザクション開始済みのレコードに対してほかのユーザーがアクセスした場合には、データベース・システム上はlock wait状態と認識されます。ロック・タイムアウトの設定は、書き込み中断時間より、少なくとも長く設定しなければなりません。

私たちは、ピーク時の60%OLTP負荷に対するフラッシュ・コピー・テストの結果から、ロック・タイムアウト時間を30秒に設定することにしました。また、Write Suspendコマンド実行時には、データベース接続が必要です。OLTP負荷特性が高い場合には、num_poolagentなどのパラメーターで十分なエージェント数を用意する必要があります。エージェント数が少ない場合には、接続処理に手間取ります。write resumeの実行が遅れて、結果的にデータベース・システム上の書き込み中断時間が増加する可能性があるので注意が必要です。

なお、汎用検索サーバーでは、フラッシュ・コピー実施時のバッファー・プール上のデータを回復するために、内部的にクラッシュ・リカバリーを実施し、ログからディスクにデータベース・データを展開します。

(2)連合データベース機能

今回のシステムでは、業務そのものの内容である法人データベースとユーザー認証のためのLDAPデータベースの二つが存在し、論理的に別データベース・システムとして配置しました。ただし、汎用検索サーバーで分析する際に、それぞれのデータベースに存在するテーブルをJoinして分析する必要がありました。UDBの連合データベース機能を使用すると、LDAPのテーブルがあたかも法人データベースにあるかのように定義できます。図12に定義例を示します。注意点としては、ユーザー定義の

```
ノード定義
    CATALOG LOCAL NODE ldapinst instance ldapinst
    ;
ライブラリー定義
    CREATE WRAPPER DRDA
    ;
データ・ソースの定義
    CREATE SERVER LDAPDB TYPE DB2/EEE VERSION 7
    WRAPPER DRDA
    AUTHORIZATION ldapinst PASSWORD ldapinst
    OPTIONS (NODE  'LDAPINST', DBNAME 'LDAPDB')
    ;
ユーザー定義
    CREATE USER MAPPING FOR b2binst Server LDAPDB
    OPTIONS ( REMOTE_AUTHID 'ldapinst', REMOTE_PASSWORD 'ldapinst')
    ;
    CREATE USER MAPPING FOR bnkclnt Server LDAPDB
    OPTIONS ( REMOTE_AUTHID 'bnkclnt', REMOTE_PASSWORD 'security')
    ;
ニックネームの定義
    CREATE NICKNAME ldap_view01  for LDAPDB.LDAPINST.LDAP_VIEW01
    ;
    CREATE NICKNAME ldap_view02  for LDAPDB.LDAPINST.LDAP_VIEW02
    ;
```

図12. 連合データベースの定義例

表2. CP検証テスト結果

●平均SQL処理実行数

オンライン・ピーク 正常系	オンライン・ピーク 障害系（＃3縮退） テスト時のデータベース・サーバーのリソースについて確認する。	オンライン・ピーク 障害系（＃1縮退）	フラッシュ・コピー ロック・タイムアウト20秒	フラッシュ・コピー ロック・タイムアウト30秒
処理件数/秒（合計）	処理件数/秒（合計）	処理件数/秒（合計）	処理件数/秒（合計）	
347.9	350.6	348.3	195.8	194.9

●CPU使用率

テスト・ケース	オンライン・ピーク 正常系	オンライン・ピーク 障害系（＃3縮退）	オンライン・ピーク 障害系（＃1縮退）	フラッシュ・コピー ロック・タイムアウト20秒	フラッシュ・コピー ロック・タイムアウト30秒
ホスト名	平均 CPU使用率	平均 CPU使用率	平均 CPU使用率	平均 CPU使用率	平均 CPU使用率
tmd2db1x	27.20%	26.80%	45.30%	17.70%	17.20%
tmd2db2x	20.70%	19.80%	—	15.80%	13.20%
tmd2db3x	19.60%	43.80%	19.80%	12.90%	13.00%
tmd2db4x	15.80%	—	15.90%	—	—

●メモリー使用量

テスト・ケース	オンライン・ピーク 正常系	オンライン・ピーク 障害系（＃3縮退）	オンライン・ピーク 障害系（＃1縮退）	フラッシュ・コピー ロック・タイムアウト20秒	フラッシュ・コピー ロック・タイムアウト30秒
ホスト名	MEM使用率	MEM使用率	MEM使用率	MEM使用率	MEM使用率
tmd2db1x	39.00%	37.00%	84.80%	35.90%	36.20%
tmd2db2x	59.20%	47.00%	—	53.60%	48.30%
tmd2db3x	59.60%	82.20%	54.60%	49.20%	48.70%
tmd2db4x	57.40%	—	50.10%	—	—

●エージェント数

テスト・ケース	オンライン・ピーク 正常系	オンライン・ピーク 障害系（＃3縮退）	オンライン・ピーク 障害系（＃3縮退）	オンライン・ピーク 障害系（＃1縮退）	フラッシュ・コピー ロック・タイムアウト20秒	フラッシュ・コピー ロック・タイムアウト30秒
ホスト名	Max_agents_with_appl	Max_agents_with_appl	agents_with_appl（平均）	Max_agents_with_appl	Max_agents_with_appl	Max_agents_with_appl
tmd2db1x	283	268	246.9	306	150.9	137.8
tmd2db2x	285	265	248.4	—	107.4	116.5
tmd2db3x	275	307	286.2	324	61	136.7

●バッファー・ヒット率

テスト・ケース	オンライン・ピーク 正常系	オンライン・ピーク 障害系（＃3縮退）	オンライン・ピーク 障害系（＃1縮退）	フラッシュ・コピー ロック・タイムアウト20秒	フラッシュ・コピー ロック・タイムアウト30秒
ホスト名	Buffer Hit Ratio	Buffer Hit Ratio	Buffer Hit Ratio	Buffer Hit Ratio	Buffer Hit Ratio
tmd2db1x	99.90%	99.50%	100.00%	99.70%	99.80%
tmd2db2x	99.10%	100.00%	—	86.30%	93.90%
tmd2db3x	99.50%	96.70%	95.80%	85.00%	93.70%

●ロック関連

テスト・ケース	Dead Lock Detected	Lock Timeouts	Lock Escalation	Lock liist memory in use	Lock liist utilization
オンライン・ピーク　正常系	0	0	0	1539596	7.28%
オンライン・ピーク　障害系（＃3縮退）	0	0	0	1588896	7.51%
オンライン・ピーク　障害系（＃1縮退）	0	0	0	1976976	9.34%
フラッシュ・コピー　FC-20秒設定	0	4	0	591444	2.80%
フラッシュ・コピー　FC-30秒設定	0	0	0	565668	2.67%

●DB2®関連メモリー

実施日	Catalog Cache Hit Ratio	Pckcache Hit Ratio	Application Cache Hit Ratio
オンライン・ピーク　正常系	100.00%	90.00%	78.00%
オンライン・ピーク　障害系（＃3縮退）	100.00%	90.00%	79.00%
オンライン・ピーク　障害系（＃1縮退）	100.00%	90.00%	78.00%
フラッシュ・コピー　オンライン・ピーク	103353	0.02%	0.91%
フラッシュ・コピー　オンライン・ピーク	254637.7	0.04%	2.32%

表3. 性能要件

	ピーク時操作（グランド・デザイン）	ピーク時操作量（分）	ピーク時操作量（秒）
ユーザー・ログイン数	52,000	866.7	14.4
Webページ・アクセス数	196,758	3,279.3	54.7
明細行総数	61,453	1,024.2	17.1
オンライン取引数	47,013	783.6	13.1
残高照会	7,494	124.9	2.1
明細照会	14,988	249.8	4.2
振込振替	24,321	405.4	6.8
マスター更新	210	3.5	0.1

ユーザーIDおよびパスワードは、実運用段階で変更の可能性があるため、外出しのパラメーター・ファイルからの読み込みとしました。また、Microsoft® Accessを使用する場合には、クライアントのdb2cli.iniにCURRENTSCHEMA=<スキーマ名>を記述する必要があります。

3.4. CP検証テスト

ITbの後半からSTの局面において、顧客の代表的な取引をモデル・シナリオとしたCP検証テストを実施しました。表3にシステム化性能要件を示します。

CP検証テストは、正常系の確認をITbの後半で、また縮退状況を前提とした障害系を含めた最終確認テストをSTの局面で実施しました。結果を表2に示します。

オンライン・ピーク・テストの結果、オンライン取引が1秒当たり14件、SQL処理は1秒当たり350件となり、性能要件が満たされたことを確認しました。また、1台当たりの物理ノード障害時の縮退状況においても、オンライン取引は1秒当たり14件、SQL処理件数は1秒当たり350件達成しました。さらに、CPU使用率が45%、メモリー使用率が85%であり、問題がないことが確認できました。

エージェント・プール数については、3台のアプリケーション・サーバーのコネクション・プールの合計が270に対して、バッチや監視系を含めて300の設定で問題がないことを確認しました。OLTP環境におけるデータベース・システムとしての最大のチューニング・ポイントは、バッファー・ヒット率で示されるバッファー・プール上にデータが確保できる割合を高くすることです。このテスト結果は、90%の目標値に対して95%でした。

SQLをアクセス・パスの保存領域であるパッケージ・キャッシュ、カタログ情報を保管するカタログ・キャッシュ、SQL実行領域であるアプリケーション・キャッシュは、いずれも目標値の80%をほぼ達成することができました。また、ロック・タイムアウトを30秒に設定することにより、フラッシュ・コピー実施時のロック待機によるタイムアウトの発生は起きないことを証明しました。

表4. CP検証テスト結果

基本設計・詳細設計（開発）	評価	統合テスト	評価
SQL標準化ガイド	7点	CP検証テスト	9点
運用設計書	4点	運用テスト	4点
領域設計書	3点	障害テスト	2点
障害対策設計書	2点	SQLアクセス・パス検証	2点
データベース・リソース監視設計書	2点		
データベース・パラメーター設計書	2点		

4. データベース・プロジェクト開発のポイント

まとめとして、業務開発および基盤開発を含めた総合的な観点から、プロジェクト開発のポイントを整理します。今回のプロジェクト開発では、お客様より過去のシステム開発の課題点を指摘いただき、基本設計・詳細設計・統合テストの局面ごとに、設計と検証項目の確認を目的としたお客様との検討会を実施しました。

過去のプロジェクトの課題点は20項目ありますが、このうちの18項目について、データベース・システムのプロジェクト開発のシステム設計や対応、SQLの標準化ガイドの周知、統合テストの確認によって問題の発生を防止できることが分かりました。

効果の上がった課題項目を1点とし、基本設計・詳細設計（開発）局面の各設計書ごとや統合テスト局面の各イベントごとに集計し整理した結果を表4に示します。

設計（開発）局面で最も重要度が高いのは、SQL標準化ガイドです。ネーミング・ルール、ディクショナリー、コード定義を管理したデータベース論理設計や処理効率の高いSQLコーディング方法を業務チームに展開すると、品質の高い設計・開発成果物のために有効です。また、開発段階でテーブル設計の変更は必ず発生するために、データの履歴保存期間の最終確認とともにこの開発局面の終了時に、当初の領域設計書の見直しを必ず行います。また、パラメーター設計やデータベース・リソース監視設計については、他の論文を参考にしてください［参考文献5］。

（ページ数および表記上の観点から、著者の了解を得て編集部にて手を入れてあります）

［参考文献］

［1］堀野 健司「WebsphereとDB2 UDBを組み合わせた場合の考慮点」2001年度IBM経験事例報告

［2］岡田 大輔「JDBCプログラミング徹底入門」『DBマガジン 3月号』翔泳社、2002年

［3］吉澤 剛士・室住 正晴「e-businessにおけるデータ・アクセスインターフェースの検証」2001年度IBM経験事例報告

［4］土井 智成「UDB/EEEを使用した今後のデータベース・システム構成」2000年度IBMプロフェッショナル論文

［5］土井 智成「システム設計/構築比較 DB2 UDBとOracle」1999年度IBM経験事例報告