保存本

05－開－21

データベース構築促進及び技術開発に関する報告書

グループワーク支援のための分散型トランザクション管理方式の調査研究

平成6年3月

財団法人　データベース振興センター
委託先　株式会社新世代システムセンター

KEIRIN

この事業は、競輪の補助金を受けて実施したものです。

序

データベースは、わが国の情報化の進展上、重要な役割を果たすものと期待されている。今後、データベースの普及により、わが国において健全な高度情報化社会の形成が期待される。さらに海外に対して提供可能なデータベースの整備は、国際的な情報化への貢献および自由な情報流通の確保の観点からも必要である。しかしながら、現在わが国で流通しているデータベースの中でわが国独自のものは１／３にすぎないのが現状であり、わが国データベースサービスひいてはバランスある情報産業の健全な発展を図るためには、わが国独自のデータベースの構築およびデータベース関連技術の研究開発を強力に促進し、データベースの拡充を図る必要がある。

このような要請に応えるため、㈶データベース振興センターでは日本自転車振興会から機械工業振興資金の交付を受けて、データベースの構築および技術開発について民間企業、団体等に対して委託事業を実施している。委託事業の内容は、社会的、経済的、国際的に重要で、また地域および産業の発展の促進に寄与すると考えられているデータベースの構築とデータベース作成の効率化、流通の促進、利用の円滑化・容易化などに関係したソフトウェア技術・ハードウェア技術である。

本事業の推進に当って、当財団に学識経験者の方々で構成されるデータベース構築・技術開発促進委員会（委員長　山梨学院大学教授　蓼沼良一氏）を設置している。

この「グループワーク支援のための分散型トランザクション管理方式の調査研究」は平成５年度のデータベースの構築促進および技術開発促進事業として、当財団が株式会社新世代システムセンターに対して委託実施した課題の一つである。この成果が、データベースに興味をお持ちの方々や諸分野の皆様方のお役に立てば幸いである。

なお、平成５年度データベースの構築促進および技術開発促進事業で実施した課題は次表のとおりである。

平成６年３月

財団法人　データベース振興センター

平成5年度　データベース構築・技術開発促進委託課題一覧

分　野	課　題　名	委　託　先
社　会	1　CD－ROMによるテレビ視聴率データベースの構築	㈱ビデオ・リサーチ
	2　輸入畜肉貨物の規格・重量等の検証用データベースの構築	五十嵐冷蔵㈱
	3　知的資源型データベースの調査研究	㈱ジャパンコミュニケーションズインスティテュート
	4　ビジネス雑誌記事データベースの共同構築とその利用に関する調査研究	経済文献研究会
	5　新しい電子情報サービスに関する調査研究	㈱日本経済新聞社／㈱日経データ社
	6　研修用教材データベースのプロトタイプ作成	㈱大阪科学技術センター
	7　マイクロマシン技術情報データベースの構築調査	㈱マイクロマシンセンター
中小企業振興地域活性化	8　異分野研究のための知的オリエンテーション・データベースシステムの構築	㈱けいはんな
	9　関西広域データベースセンター設立のための調査研究	関西データベース協議会
	10　地域活性化のための産・学交流支援データベースのプロトタイプ作成	東北インテリジェント・コスモス学術機構
	11　中小企業技術情報データベースの構築	㈱オーネット
	12　地域情報を対象にした分散協調型データベースシステムの開発	㈱エマーズ
海　外	13　電子デバイス情報の海外提供サービスに関する調査	電子デバイス情報サービス㈱
	14　英日キーワード変換機能をもつデータベース検索システムの開発	カテナ㈱
	15　CD－ROMによる5カ国対訳特許用語辞典及び関連諸制度一覧の構築	丸善㈱
技　術	16　人体形状画像データ合成のための技術開発	㈳人間生活工学研究センター
	17　OCRを利用したキーワード自動抽出に関する調査研究	㈱エレクトロニック・ライブラリー
	18　既存画像データのフォーマット変換システムのプロトタイプ作成	㈱ジー・サーチ
	19　安全研究における多重シソーラス・システム構築のための基本安全用語データベースの開発	㈱紀伊國屋書店
	20　データベース検索サポートシステムのプロトタイプ作成	セントラル開発㈱情報図書館 RUKIT
	21　グループワーク支援のための分散型トランザクション管理方式の調査研究	㈱新世代システムセンター

目　　次

Ⅰ．課題性の研究

1. 目　的

これまでの情報システムは、個人の作業の支援が中心であったが、今後は、複数の個人によるグループ作業の支援、いわゆるグループウェアが情報システムの役割を担うこととなる。グループワークでは、グループの情報を共有するためのデータベースシステムの構築が重要となる。従来のデータベースでは、主として小量のデータを短時間操作するものであったが、グループウェアでは、大量の複雑な構造のデータが、長時間操作される。このため、グループワークで動的に更新されるデータの管理方式について調査・研究を実施し、データベース構築技術の向上を図るとともにわが国におけるデータベースの発展に寄与することを目的とする。

1. 実施内容

本調査研究では、以下の項目を主たる内容としている。

1）　グループワークでのデータ共有方式の検討

共同ウィンドウ、共同執筆等を例として、データの共有形態を調査する。

2）トランザクションのモデルの検討

グループワークでのトランザクションのモデルを検討する。

3）　同時実行制御方式の検討

従来のreadと write演算に基づいたロック方式に対して、応用の意味に基づいた意味的なロック方式等の方式を検討する。

1. 実施体制

㈱新世代システムセンター内に当該事業遂行のための「特別調査・研究委員会」を設置し、事業の実施に当った。

調査研究委員会

滝沢　誠（東京電気大学理工学部・経営工学科助教授）

山鳥雄嗣（日本情報処理開発協会・調査部長）

市川　隆（ＡＩ・ファジー振興センター所長）

道田國夫（新世代システムセンター企画部長）

近藤伸一（新世代システムセンター調査部長）

Ⅱ．グループウェア支援のための分散型トランザクション管理方式

1章　グループウェアとデータベース

1.1　グループワークとグループウェア

グループワーク支援システム（グループウェア）成功の鍵は支援対象であるグループそのものの理解にある。グループは個人、組織の中間に位置する。グループの多くは個人よりは社会的であるが、組織ほどフォーマルなものではない。このことから、グループは個人ともまた組織とも異なる特徴をもつ実体であると言える。したがって、個人に関する知識や組織に関する知識だけではグループワーク支援は成功しがたいと考えるのが自然である。グループとその内部のグループプロセスを明らかにし、そのプロセスと整合性をもつ支援システムが望まれる。グループとグループメンバーの間のコミニケーション、グループメンバー間のコミュニケーション、リーダーシップ、コンフリクト管理などを支援する機能が求められる。

近年グループワークを支援するソフトウェアとしてグループウェアが一部で注目を集めている。グループウェアの定義は現状では必ずしも一定していないが、グループウェアはグループ間の協調を支援するソフトウェアではなく、グループ内の協調を支援するソフトウェアであると考えられる。

グループウェアが注目されるようになった背景には、コンピュータ技術の発展、通信技術とコンピュータネットワークの発展（分散システムの発展）、ユーザインタフェース技術の発展、パーソナルなコンピュータ利用形態の進展、などがあることは否定できない。しかし、これらの技術は、グループウェア実現の手段を提供するにすぎなく、グループウェア技術の本質の問題ではない。

グループウェアに限らず、何かを支援するソフトウェアは支援対象の深い理解に基づくものでなければならない。会計システムは会計についての理解に基づくものであるべきことは当然であろう。応用分野知識の重要性が認識されつつある。グループウェアでの応用分

野知識とはすなわち、グループに関する知識ということになる。また、システム開発の成否はそのシステムを導入する組織に関する知識の量にかかっている。その意味でもグループウェアではグループに関する知識が重要である。

ソフトウェア工学の世界では1970年代後半から1980年代にかけ、種々のソフトウェア開発方法論や支援ツールが提案されてきた。1980年代後半からはＣＡＳＥ（Computer-Aided Sosfware Engineering）ツールが登場してきている。しかしながら、こういうソフトウェア開発方法論、ソフトウェア開発支援ツール、ＣＡＳＥツールは期待ほど効果をあげていない。このことを真摯にとらえている一群の人たちは、我々がそもそもソフト開発自体をよく理解していないことに気付き、ソフトウェア・プロセス理解の重要性を説いている。

グループウェア技術の研究開発や発展にはグループの理解が中心課題になる。

グループワーク支援システム研究中心課題はグループの理解である。有効なグループワーク支援システムを開発するために、明らかにすべきは次のような事柄である。

＊グループというものは一体何か？

＊グループとはどういう特徴を持っているか？

＊グループ・プロセスはどうなっているか？

＊グループ作業では何が問題になっているか？

＊グループワーク支援システムが支援できるのはどの様な部分か？

＊グループワーク支援システムを使うことでどの様な問題が解決できると期待できるか？

＊グループワーク支援システムを導入するとグループ・プロセスはどう変化するか？

＊グループワーク支援システムを導入したとき、その効果をどのように測定すべきか？

グループは複数の個人が、最終的には統合された一つのあるいは複数の成果めざして協調して作業する実体であり、組織ほどフォーマルなものではない。むしろ、組織レベルでは決められることだけが決めてあり、他の重要な部分はグループ・レベルに任されている、と考える方がよい。グループは単なる個人の集合ではないし、組織でもない。

(1) グループはソフトウェアシステムである。

システム理論では、物理現象を支配ルールとするようなシステムはハードウェアシステムと呼ばれる。ハードウェアシステムではシステムの目的や評価基準は決めやすい。一方、人間を要素に含むシステムでは多くの人が同意できる明確な目的や評価基準を決めることが非常に難しい。このようなシステムはソフトウェアシステムと呼ばれる。グループは人

間中心のシステムであり、ソフトウェアシステムである。グループの目的や評価基準を決めることは難しい。

(2) グループはダイナミックなシステムである。

① グループの課題は変化する。

② グループ・メンバーの役割はダイナミックに定める。

③ グループ・メンバーの動機は時間とともに変化する。

④ グループはその外部環境の変化に影響を受ける。

(3) グループ・メンバーは多様である。

① グループ・メンバーの動機はメンバー毎に異なる。

② グループ・メンバーの能力や得意分野は異なる。

③ グループ・メンバーの経験は異なる。

(4) グループは成長する。

グループは徐々に成長する。グループ・メンバーも習熟し成長するが、グループも成長する。

(5) グループの運営はインフォーマルである。

組織レベルではフォーマルなプロセスが支配的であるが、逆にグループ・レベルではプロセスはインフォーマルである。メンバー間の合意を基本に柔軟に運営される。

1.2 グループウェアにおけるデータベース利用の拡大

データベースシステムは、これまで事務処理を中心として、種々の応用により広く利用されてきている。1960年初頭に、組織体（企業）の運用で利用されるデータを、データベースとして集中化することによりデータベースシステムの概念が確立された。これ以来、在庫管理、顧客管理、座席予約等のオンラインシステム等で、広くデータベースシステムが利用されてきている。1980年代からは、リレーショナルモデルがデータベースシステムの中心となり、従来の大型汎用コンピュータからワークステーション、さらにパソコンにいたるまで、様々な情報システムにリレーショナルモデルに基づいたデータベースシステムが実装され利用されてきている。

このようなデータベースシステム利用の普及と発展につれて、従来の定型的な事務処理か

ら、最近ではグループウェア、設計等の新しい分野でデータベースシステムが利用されつつある。こうした応用は、データベースシステムに対して、複雑なデータ構造の表現、データの意味の取り扱い、データと手続きの一体化等を要求している。しかし、リレーショナルモデルでは、こうした応用からの要求に対する十分な機能を持たないとの指摘がある。このため、オブジェクト指向モデルがリレーショナルモデルにかわるものとして注目されている。

データベースシステムでは、利用者の必要とするデータをデータベースから有効かつ迅速に検索する一方、データベース内のデータの更新も行われる。例えば、グループウェアでの個人のスケジュールを考えると、各個人の予定が決定したり変更になった場合、データベース内のスケジュールのデータを更新しなければならない。また、設計では、例えば設計の段階での設計図の変更にともない、設計図データが更新される。データベースシステムでは、こうしたテーブルの更新に対して、以下の目標を満足する必要がある。

(1) データベースを正しい状態とする。

(2) データベースシステムの性能を向上させる。

利用者がグループウェア等の仕事を行うとき、一つの仕事はデータベースに対する一つ以上の操作演算から構成される。例えば、各個人のスケジュールの決定では、個人のスケジュールのデータベースと同時に、他人のスケジュールデータベース、また、会議室等のデータベースが更新される。この中のすべてのデータベースの更新が完了せずに、一部のデータベースに対する更新に失敗すると矛盾した結果となる。例えば、会議の参加者のスケジュールの更新を行ったが、会議室の予約が出来ない場合である。この場合には、会議を実施できないので、スケジュールのデータベースの更新を無効とする必要がある。複数の演算の全てが正しく実行されたときのみこれらの演算の実行が完了するが、一つの演算実行でも正しく完了しなかった場合に全ての演算の実行を無効となるとき、これらの演算の実行は「原子的」であるとする。このように、利用者から見た仕事は、データベースに対する一つ以上の更新演算から構成されるが、これらの実行は、「原子的(atomic)」でなければならない。これを、「トランザクション(transaction) 」という。

データベースシステムは、複数の利用者からのトランザクションを実行する必要がある。このとき、データベースシステム全体のスループットを向上させる必要がある。スループットとは、単位時間内に処理されるトランザクションの数(TPS: transaction per second)

である。スループットを向上するためには、各データベースシステムでは、複数のトランザクションからの演算を「インタリーブ実行」の形式で実行する。即ち、データベースに対する操作演算は、ディスク等の二次記憶に対する操作演算（IO演算）であることから、実行時間が大きい。このため、あるトランザクションの演算を実行している間に、他のトランザクションの演算が実行される。こうした実行方法を、「インタリーブ実行」という。各データベースシステムが、インタリーブ実行した演算の履歴をログ(log）とする。

現在の情報システムは、通常複数のコンピュータから構成される。データベースシステムも、利用者のインタフェース機能をクライアントとし、データベースに対する基本的な操作と管理を行う機能をサーバとして、これらを異なったコンピュータ（ワークステーション）に配置する形態となっている。いわゆるクライアントサーバモデルに基づいたシステムである。こうしたシステムでは、クライアントからデータベースサーバに演算が発行された後、演算が実行され、結果がクライアントに返される。トランザクションは、クライアントから発行される演算の系列である。ここで、クライアントから複数のデータベースサーバに演算を発行できる。この場合には、各データベースサーバで複数のトランザクションからの演算が「インタリーブ実行」されると同時に、各トランザクションは複数のデータベースサーバで実行されることになる。このように、分散型システムでは、複数のトランザクションが複数のデータベースサーバにより並列実行され、一方各データベースサーバではインタリーブ実行される。

しかし、複数のトランザクションを複数のデータベースサーバで実行することの正しさが問題となる。この場合、正しさの根拠として、複数のトランザクションを一つ一つ直列に実行した結果と同じであるかどうかがある。ある直列実行と同じであるとき、この並列実行を直列可能(serializable)である[BERN87]とする。複数のトランザクションの実行が同じかどうかでは、演算の実行順序が問題となる。例えば、あるデータオブジェクトxに対してあるトランザクションがreadを行った後に、他のトランザクションが writeを行ったときと、この逆に実行されたときでは、read演算により読みだされる xの値は異なっている。このように、演算の実行順序により、得られる結果が異なる演算は、互いに競合(conflict)するという。従来の直列可能性は、ある実行での競合演算の実行順序が直列に実行した場合と同じであるとき、この実行は直列可能であると定義されてきた。複数のトランザクションの実行を直列可能とするために、これまで二相ロック（2PL : two-phase

locking) 方式[ESWA**]が用いられてきた。

二相ロック方式では、トランザクションが利用するオブジェクトを操作する前にロックを行い、どれかのオブジェクトのロックを解放した後にはロックが行われない。トランザクションの完了前にロックの解放を行うと、復旧を行えない場合にあるトランザクションの異常終了（アボート）に対して、連鎖的に複数のトランザクションがアボートする問題がある。このために、現在、トランザクションの終了時に、全トランザクションの解放を行う厳格(strict)な形式がとられている。この厳格な二相ロック方式の問題点は、トランザクションがオブジェクトをロックしている時間が長くなり、データベースシステムのスループットの低下をもたらすことである。座席予約、銀行の口座の入出金等の応用では、各トランザクションは少ないオブジェクト（例えば、一つのレコード）を短時間操作する。このために、二相ロック方式でも性能面での問題は大きくなかった。これに対して、グループウェアでのトランザクションは、複雑なデータ構造を持つ大量のオブジェクトを長時間操作する。このために、readと write演算についての二相ロック形式ではない同時実行制御方式が求められている。本研究では、応用の意味に基づいた同時実行制御方式について検討する。

本研究では、データベースは複数のオブジェクトから構成されると考える。各オブジェクトは、異なったデータベースサーバに分散されている。各オブジェクトは、データ構造とこれに対する操作演算を提供する。利用者は、オブジェクトのデータ構造をオブジェクトが提供する演算を用いてのみ操作できる。即ち、オブジェクトは、抽象データ構造（ADT: abstract data type）である。各オブジェクトに対して、オブジェクトの操作演算間での競合関係が意味的に与えられる。例えば、オブジェクトo が演算op_1 とop_2 が提供されるとする。このとき、op_1 とop_2 を、どのような順序で実行されても、結果が同じであるとき、op_1 とop_2 は互いに交換可能であるとする。op_1 とop_2 が交換可能でないとき、即ち、op_1 とop_2 の実行順序により結果が異なるとき、op_1 とop_2 は互いに競合するとする。この競合関係は、オブジェクトの意味により与えられる。例えば、銀行オブジェクトを例として考えると、銀行の入金演算と出金演算は交換可能である。どのような順序で実行されても、銀行口座の値は同じであるからである。

2章　データベースシステムの進展

データベースシステムは、組織体の必要とするデータを統合化し、集中管理したシステムとして利用されてきた。ワークステーションの高性能化、低価格化と、通信網の発展により、データベースシステムの構成が変化してきている。一つは、データベースシステムの利用者インタフェースの機能と、データベースに対する基本操作機能を分離し、おのおの機能をクライアントとサーバとして、異なったワークステーションに分散するものである。さらに、独立に設計され運用されてきた複数のデータベースサーバを利用することも可能となってきている。ここでは、こうした複数のデータベースサーバから構成される分散型データベースシステムについて述べる。まず、データベースシステム[TAKI92]とは何かを簡単に触れる。

2.1　統合化

学校の例を考えてみる。学校では、先生は、学生の成績を管理するために学生の成績のデータを利用し、事務は学生のクラブ活動を管理するために学生のクラブ活動のデータを利用する。この場合のデータベースシステムは、学校での成績管理、クラブ活動管理等の学校の業務に必要な学生についてのデータを記憶するシステムとなる。データの集まりをデータベース、成績管理等の業務を応用とし、業務を行うためにデータベースを利用するプログラムを応用プログラムとする。

データの利用方法の歴史を少し考えてみる。コンピュータが登場してから1960年代までは、応用プログラム毎に必要とされるデータがファイルとして管理されてきた。ファイルとは、プログラムが必要とするデータをプログラムの操作方法に合う形式で集めたものである。例えば、成績管理では、学生の成績順に学生データを利用するとき、学生の学籍番号、名前、学年、履修科目、成績についてのデータを成績順に整列させて記憶したものが成績管理のためのファイルである。また、クラブ管理の応用では、学生の学籍番号、名前、所属しているクラブ、クラブ内での役割といったデータからなるファイルを用いて処理が行われる。このようなシステムをファイルシステムという。こうしたファイルシステムでは、1950年代後半頃から以下の点が問題となってきた。

(1) ファイル間でのデータの一致性の保持が困難である。例えば、同じ学生のデータが複数のファイルに存在すると、本人のデータが変化したとき、全ファイル内のデータを更新する必要がある。ある学生が、入学した場合に、成績ファイルとクラブファイルの両方のファイルにこの学生の情報を追加する必要がある。

(2) 複数ファイル内に同一データが冗長に存在するので、格納負荷が増大する。(1)の例で、各学生の学籍番号、名前等についてのデータが成績管理ファイルとクラブファイルの両方に存在する。

(3) ファイル毎にデータの表現形式が異なることにより、ファイルシステム間でのデータの共有が困難である。例えば、あるファイルでは学生の名前をローマ字で示し、他のファイルではカタカナで示している場合である。

以上の問題点を解決するために、各応用のファイル内に記憶されていたデータを一箇所に集めることが1960年代初頭に行われた。例えば、複数の応用で利用されていた学籍番号、名前、住所、履修科目、クラブ等の種々の学生データを各学生毎に一つのデータを作ることである［図2.1］。これを統合化(integration) という。データを集めたものをデータベースとする。ファイルシステムでは、各ファイルごとにデータが分散して管理されたが、統合化により、データベース内のデータはデータベース管理者(DBA: database administrator)により集中的に管理される。ファイルシステムに対して、データベースシステムは以下の利点を持つ。

(1) データの冗長性を除去できる。

(2) データの一致性を保つことが容易である。

(3) データの共有性を向上できる。組織体としての標準のデータ形式、データ操作方法を与えることができる。

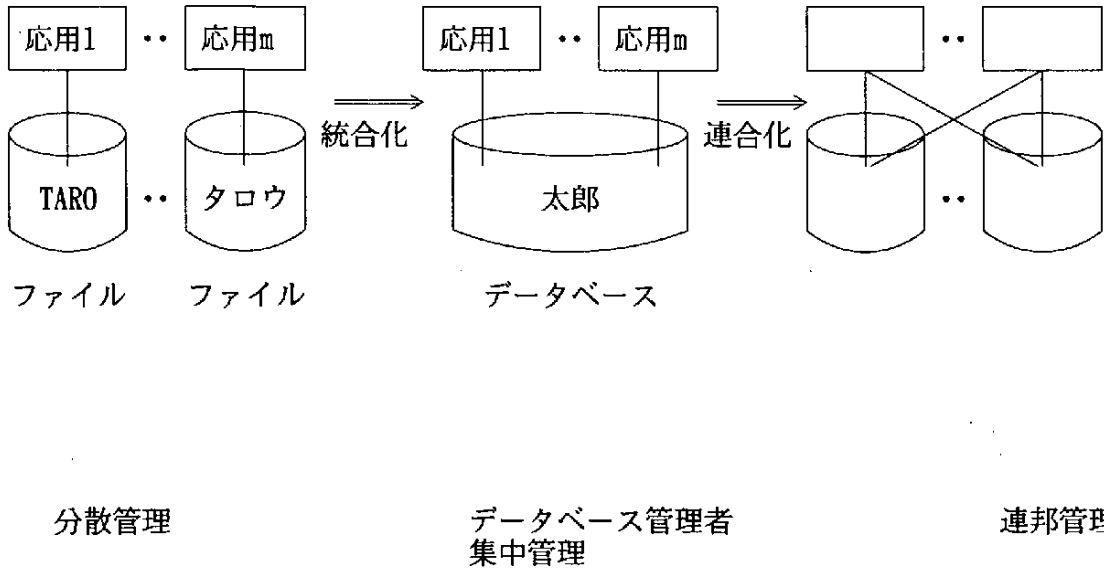


図2.1　データベースシステムの歴史

1960年代から70年代の情報システムでは、大型汎用コンピュータを中心として、利用者は端末を通じてデータベースシステムを利用していた。1980年代から、高性能、低価格なワークステーションが普及し、これらはローカルエリアネットワーク(LAN) といった通信ネットワークにより相互結合されてきた。こうした情報システムでは、データベースシステムはワークステーションにも構築されてきている。また、データベースシステムの利用者インタフェースの機能をクライアント、データベースの基本操作機能をサーバとして、各々を異なったワークステーションに分散したクライアントサーバモデル[***] のシステムが利用されてきている。これまで、各応用は一つのデータベースを利用していたが、クライアントから複数のデータベースサーバの利用が可能となった。このように、データベースシステムは、80年代から再度分散化されてきている。ファイルシステムのように、各応用毎に完全に分散した形態ではなく、各応用単位に管理されてきたデータベースシステムが、情報システム全体としての緩やかな管理に従っていく形態である。これは、ファイルシステムの完全分散化と、データベースシステムの完全集中化に対して、この中間にあるもので、連合化(federation)といわれる。

2.2 リレーショナルモデル

利用者からデータベースシステムがどのようにみえるかを示したものがデータモデル(data model) である。現在の主要なデータベースシステムは、リレーショナルモデルに基づいているので、ここでリレーショナルモデルについて説明する。リレーショナルモデル[CODD70]は、データ構造、データ構造に対する操作演算、データ構造が保つべき制約としてのインテグリティ制約により与えられる。

2.2.1 データ構造

リレーショナルモデルのデータ構造について考える。データ構造は、テーブル(table) から構成される。テーブルを例により説明する。テーブルの例を図2.2に示す。従業員の情報を示すテーブルEmp は、各従業員の番号(eno) 、名前(ename) 、給料(sal) 、所属している部の番号(dno) に関する情報を持っている。これらを属性(attribute) またはカラム(column)とする。テーブル名と属性の構成をテーブルのスキーマとする。

Emp内の各属性に値を与えたもの、例えば、<1, A, 150, 1>を行(row) とする。各行は、一人の従業員を示す。行の集合をテーブルのインスタンスという。インスタンスは、更新演算により変化する。しかし、スキーマは変化しない。

テーブルDeptは、部についての番号(dno) 、名前(dname) の情報を持っている。

Dept内の各行は、一つの部を示す。

Emp	eno	ename	sal	dno
	1	A	150	1
	2	B	200	1
	3	C	300	3
	4	A	100	2
	5	D	280	3
	6	E	450	1
	7	F	130	2
	8	D	310	2
	9	G	200	1
	10	H	280	1

Dept	dno	dname
	1	DDD
	2	EEE
	3	FFF
	4	GGG

図2.2 テーブルの例

2.2.2 代数演算

リレーショナルモデルの基本操作演算をリレーショナル代数演算という。テーブルを行の集合として考え、集合に対する代数演算がある。この中で、制限、射影、結合演算について説明する。

(1) 制限(restriction)

テーブルから、ある条件を充足する行の集合を求める演算である。例として、Emp[sal ≧ 200 ∧ eno < 5]は、テーブルEmpから給料(sal) が 200以上で、従業員番号(eno) が 5より小さいものを導出する演算である。この結果、図2.3に示すテーブルが得られる。

Emp	eno	ename	sal	dno
	2	B	200	1
	3	C	300	3

図2.3 制 限

(2) 射影(projection)

テーブルから、一部のカラムを取り出す演算である。例として、Emp[dno]は、Empからdnoの値だけを導出する演算である。この結果を図2.4に以下に示す。ここで、冗長な組が含まれていないことに注意されたい。

dno
1
2
3

図2.4 射 影

(3) 自然結合(natural join)

二つのテーブルを結合する演算である。例として、Emp とDept間の自然結合Emp * Deptの結果として導出されるテーブルを図2.5に示す。Emp とDept内で、共通の属性dno の値が同じ行が結合される。

eno	ename	sal	dno	dname
1	A	150	1	DDD
2	B	200	1	DDD
3	C	300	3	FFF
4	A	100	2	EEE
5	D	280	3	FFF
6	E	450	1	DDD
7	F	130	2	EEE
8	D	310	2	EEE
9	G	200	1	DDD
10	H	280	1	DDD

図2.5 自然結合

2.2.3 SQL

代数演算は、リレーショナルモデルの基本演算であり、コンピュータのアセンブラに対応した演算である。これに対して、高級言語に対応したものとしてSQL (structured qury language)[SQL]がある。SQL は、第一階論理に基づいた非手続き的な高水準言語であり、利用者の必要とするデータを示す条件が第一階論理により記述される。また、SQLは、ISOにより国際標準化された言語である。この意味で、リレーショナルデータベースとは、SQLを提供するシステムと定義できる。

ここでは、SQLについて概説する。SQLは、スキーマの定義、消去を行うデータ定義機能と、テーブル内の行の操作を行うデータ操作機能を持つ。データ定義機能とデータ操作機能を持つ言語をデータ言語という。SQLは、データ言語である。

A. データ定義

まず、図2.4に示したテーブルEmp(eno, ename, sal, dno) のスキーマは、以下のcreate文により定義できる。属性eno 、sal 、dno のデータ型は整数(int) であり、ename は文字列(char)である。また、eno は、空値を取れない。

```
create  table Emp
    ( eno  int NOT NULL,  /*属性eno のデータ型は整数で、空値が許されない*/
      ename      char(36),    sal    int,    dno    int )
```

B．データ操作

次に、テーブルEmp とDeptに対するSQL によるデータ操作（検索と更新）の例を説明する。

(1) DDD部員のなかで、給料が 200以上の者とその給料を求めよ。

```
select   ename, sal   from  Emp, Dept
where    dname = "DDD"  and  Dept.dno = Emp.dno  and
         sal >= 200
```

Dept.dnoは、dno がテーブルDeptの属性であることを示す。また、SQL では以下のようにも書ける。inの右側にはselect... という検索式が書かれている。このような検索式を入れ子形式という。このinは英語の関係代名詞を示し、dno を修飾している。

```
select   ename, sal  from  Emp
where    sal >= 200  and
         dno  in  ( select  dno  from Dept
                    where  dname = "DDD")
```

(2) DDD部員のなかで、給料が200以上の者とその給料を、給料の昇順に求めよ。

```
select   ename, sal  from  Emp
where    sal >= 200  and
         dno  in  ( select  dno  from Dept  where  dname = "DDD")
order  by  sal
```

(3) 部毎の平均給料を求めよ。

```
select  dno, avg(sal)  from    Emp
group   by  dno
```

(4) DDD部員の給料を1.5倍にする。

```
update  Emp   set  sal = sal * 1.5
where dno in  ( select  dno  from Dept where   dname = "DDD" )
```

(5) DDD部員を消去する。

```
delete  from   Emp
where  dno  in ( slelect  dno from  Dept  where  dname = "DDD");
```

(6) 組<11, K, 210, 1>をテーブルEmpに追加する。

```
insert into   Emp ( eno, ename, sal, dno)
values      ( 11,  K, 210, 1)
```

(7) DDD部員として、Kを追加する。

```
insert into  Emp ( eno, ename, sal, dno)
select  11, "k", 210, dno   from  Dept
where dname = "DDD"
```

C. 視　野

リレーショナルモデルでは、データベース内のテーブルを基本テーブルとする。利用者は、これらの基本テーブル内の一部を利用する。利用者が利用の度に、テーブルの結合等の操作を行うことは、容易でない。このために、利用者が必要とするデータを基本テーブルから集めた仮想的なテーブルを定義する機能がある。この仮想的テーブルを視野(view)とする。例えば、EmpとDeptの上に、従業員名と彼の属している部名を与える視野は以下のように定義できる［図2.6］。

```
create  view  ED ( name, dname)
as  select  ename, dname  from  Emp, Dept
     where  Emp.dno = Dept.dno
```

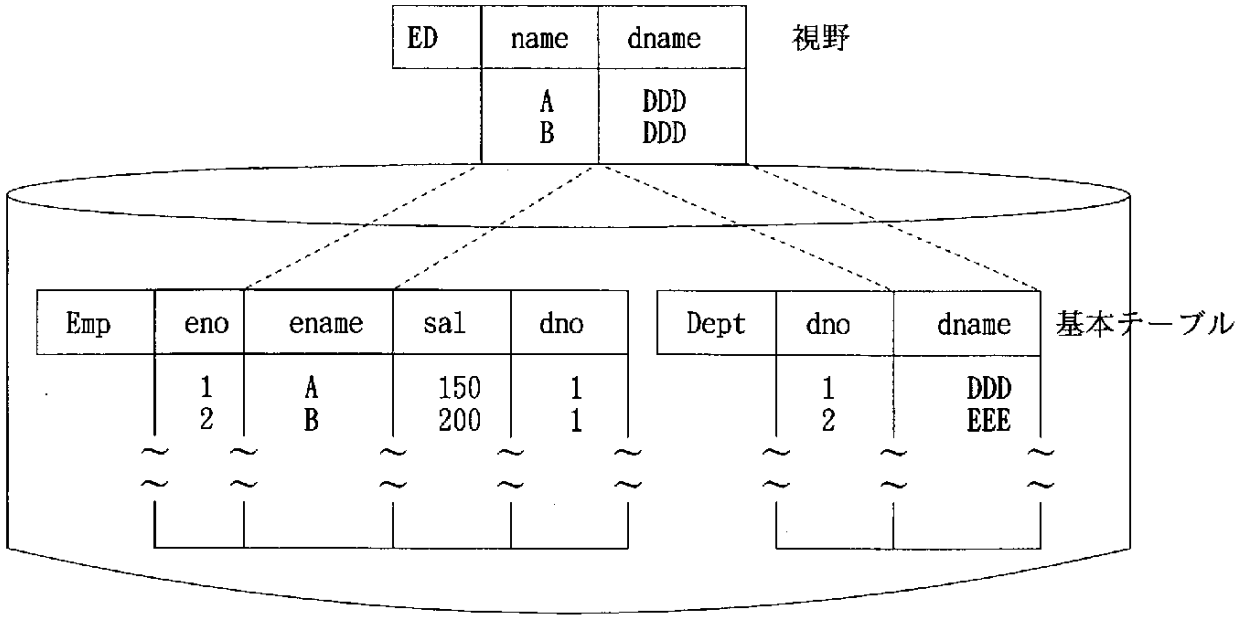


図2.6 視 野

検索演算「DDD部の部員を求めよ」は、次のように書ける。

```
select  name  from  ED  where  dname = "DDD"
```

ここで、利用者はEmpとDeptの結合を行う必要がない。データベースシステムは、このSQLから視野の定義文を用いて、以下のSQLを作成し実行する。

```
select  ename  from  Emp, Dept
    where  Emp.dno = Dept.dno and dname = "DDD"
```

以上のように、視野は以下の特徴を持つ。

(1) データの論理独立性を提供する。基本テーブルのスキーマが変化しても、視野の定義を変更することにより、応用または利用者にこの変化を隠蔽できる。

(2) 安全性の提供を容易とする。利用者は、視野をとおして見ることができるデータだけがデータベースに存在すると考える。このために、利用者に利用させたくないデータの存在を知らせずにすむ。

2.2.4 インテグリティ制約

次に、リレーショナルモデルのデータ構造が保つべきインテグリティ制約について考える。まず、キー(key) の概念がある。キーとは、各テーブル内の各行を一意に識別できる極小の属性集合である。例えば、図2.4に示したテーブルEmp で、属性eno がキーとなる。属性集合 {eno, sal} の値も、Emp 内の各行で一意であるが、これからsal を除いてもenoの値は各行で一意である。このために、 {eno, sal} はキーではない。このように、キーは、テーブルの属性の部分集合であり、一意性と極小性を満足するものである。キーについての以下の制約を実体インテグリティ制約(entity integrity constraint) という。

[実体インテグリティ制約] キーの値は、空値であってはならない。

次に、テーブル間のインテグリティ制約について考える。Deptのdno もキーである。Empもdno を持つ。Emp のdno は、キーではないがDeptのdno はキーである。このようなEmpのdno を外来キーとする。このとき、外来キーのEmp のdno の値は、Deptのdno の値として存在せねばならない。これは、Emp とDept間のインテグリティ制約であり、参照インテグリティ制約(reference integrity constraint)という。

[参照インテグリティ制約] RとS を二つのテーブルとする。a をR のキーとし、S の外来キーとする。このとき、S のa の値は、R のa の値として存在せねばならない。

Dept内のdno = 1である行が消去された場合を考える。Emp 内でdno = 1の部に属する部員の行のdno の値は1である。これは、参照インテグリティ制約を満足しない。参照インテグリティを充足するためには、以下実行しなければならない。

(1) Emp から、dno = 1である行を消去する。

(2) Emp 内のdno = 1である行のdno の値を空値とする。

(1)では、Deptの行の更新がEmp に波及していくものである。これを、更新伝播または連鎖的更新という。

２２５　特　徴

以上から、リレーショナルモデルは、以下の特徴を持っている。

(1)　簡単なテーブル形式をしたデータ独立なデータ構造を持つ。

(2)　非手続き的データ操作言語SQLを持つ。

(3)　閉包性がある。データベース内のテーブルら導出された結果はテーブルであり、再びデータ操作言語で操作できる。この性質は、データ構造を柔軟に変更できることを意味している。

(4)　リレーショナルデータベースシステムは、中程度の性能をしている。リレーショナルデータベースシステムが、非手続き的なで操作演算から物理的な操作演算の手続きを生成する。

(5)　CAD、個人用といった非定形的な業務に適する。

3章　システムモデル

本研究で検討するシステムのモデルについて考える。システムは、複数のデータベースサーバDBS1,...,DBSn (n ≧ 1)とクライアントC1,...,Cm (m ≧ 1)から構成される［図3.1］。各データベースサーバDBSiは、リレーショナルモデルのデータベースDBi を持ち、これに対するに基本的なデータ操作を行う。データベースサーバは、クランイアントからSQL を受け付け、これを実行する。クライアントは、利用者と、システム間のインタフェースである。複数のデータベースサーバと複数のクライアントから構成されるシステムを、ここでは分散型データベースシステムとする。ここで、通信ネットワークは、複数のサーバ、クランイアント間での高信頼の通信、即ち、送信されたメッセージは宛先に紛失なく、送信順に届けられる。

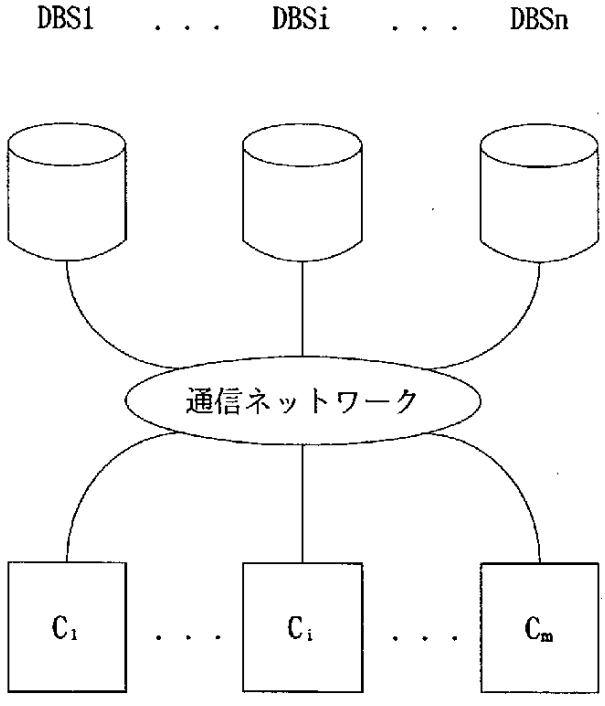


図3.1　システムモデル

クライアントとデータベースサーバ間の関係について考える。クランイアントとサーバ間のプロトコルを以下に示す。

(1) クライアントCjは、利用者からのデータ操作要求を受け取る。
(2) Cjは、操作要求されるデータを持つデータベースサーバDBSiにデータ操作要求opjiを送信する。
(3) データベースサーバDBSiは、クライアントCjからのデータ操作要求opjiを受け取り、これを実行した後、結果をCjに返す。

以上の関係を、図3.2に示す。トランザクションは、クライアントからデータベースサーバに発行されるデータ操作演算の系列である。このような計算モデルは、遠隔手続き呼び出し(RPC: remote procedure call)と呼ばれる。クライアントのプログラムから、遠隔のコンピュータ内の手続き（サーバ機能）を呼び出していることと考えることができるからである［図3.3］。

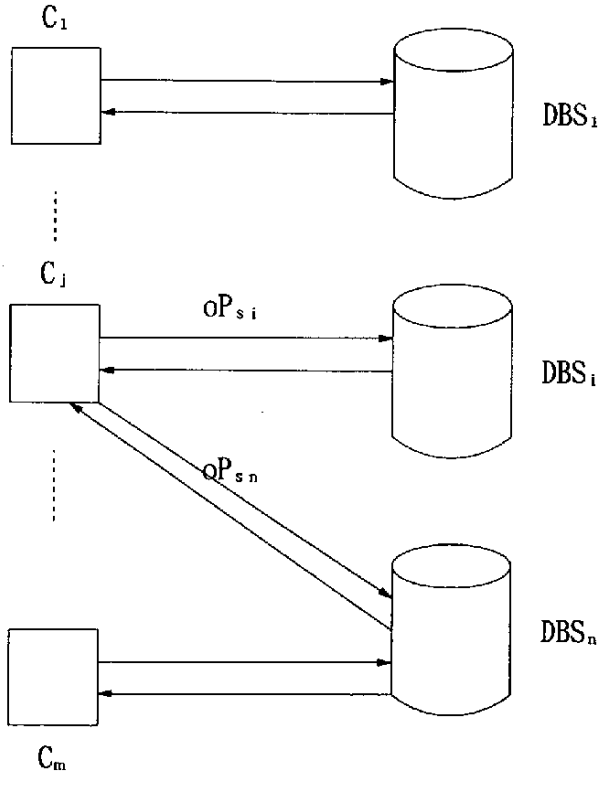


図3.2 クライアントとデータベースサーバ

各データベースDBi は、テーブルから構成される。これを物理的テーブルとする。分散型データベースシステム内の物理的テーブルの集合を、物理的データベースPDB とする。利用者が物理的データベースを操作するためには、各テーブルがどのデータベースサーバに存在するか、また同じデータが複数のテーブルに存在する場合どのテーブルを操作してよいかといったことを意識する必要がある。これらのことと独立に分散型データベースシステムを利用できることが望ましい。これを分散の独立性という。

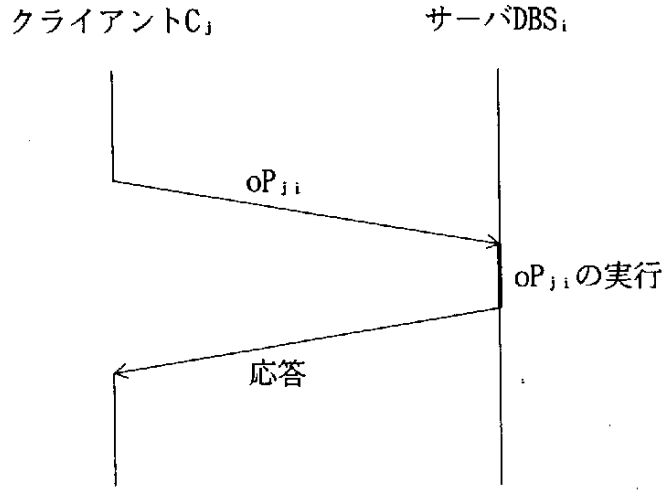


図3.3 RPC

4章　データの分散

分散型データベースシステムの目的の一つは、利用者にデータの分散の独立性を提供することである。利用者から見える分散型データベースシステムのデータベースは、テーブルの集合である。これを論理的データベースとする。論理的テーブル内の値は、実際には、あるデータベースサーバ内に格納される。各データベースサーバ内のテーブルを物理的テーブルとする。ここでは、論理的テーブルと、各データベースサーバ内の物理的テーブルとの対応関係について考える。

論理的テーブルを物理的テーブルとして格納する問題を考える。格納方法としては、断片化と冗長化がある。論理的テーブルを、SQLにより分割した各要素を断片(fragment)とする。各断片を一つの物理的テーブルとして、あるデータベースサーバに格納する。断片化には、大別して、垂直断片化と水平断片化とがある。

4.1　垂直断片化

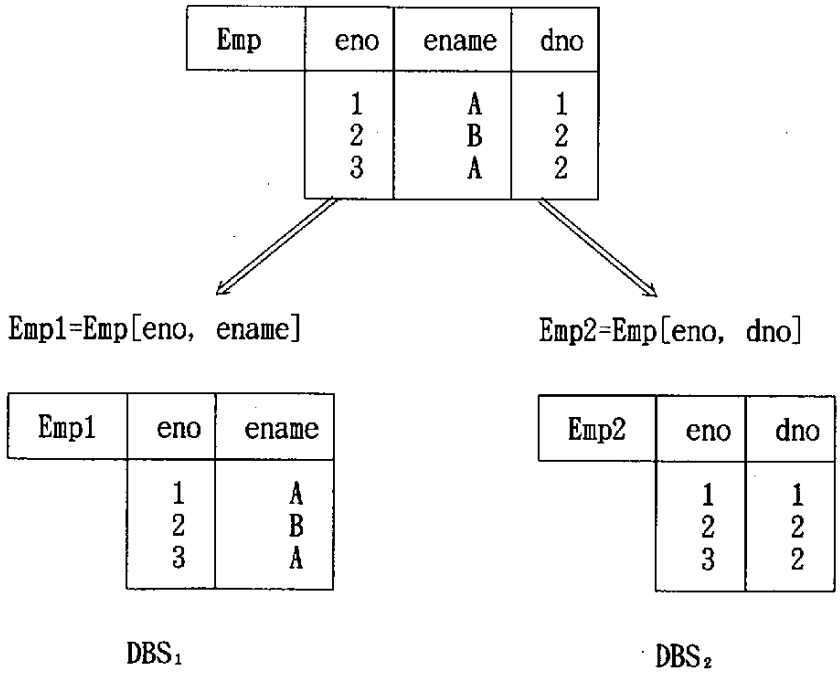


図4.1　垂直断片化

図4.1に垂直断片化の例を示す。論理的テーブルEmpを、射影断片Emp1 = Emp[eno, ename]とEmp2[eno, dno]に分割し、各々をデータベースサーバDBS1とDBS2に記憶した例である。テーブルEmp 内の属性ename についての以下のSQL 検索(1)は、断片Emp1に対する検索(2)としてDBS$_1$に発行される。

```
select ename from Emp where eno > 2    --- (1)
select ename from Emp1 where eno > 2   --- (2)
```

一方、dno についての検索は、断片Emp2に対する検索としてDBS$_2$に発行される。このように、Emp に対する検索を二つのデータベースサーバに分散でき負荷を分散できる。一方、ename とdno についての以下のSQL 検索(3)に対しては、二つの断片Emp1とEmp2に対する検索(4)が実行される。ここでは、Emp1とEmp2についての自然結合が行われる。このためには、二つのデータベースサーバDBS$_1$とDBS$_2$間での通信が必要となる。

```
select ename, dno from Emp where dno > 2    --- (3)
select Emp1.ename, Emp2.dno from Emp1, Emp2
where Emp1.eno = Emp2.eno and Emp2.dno > 2 --- (4)
```

図4.1で、二つ断片Emp1とEmp2を自然結合した結果はEmpとならねばならない（即ち、Emp = Emp1 * Emp2)。このために、各断片は、Emp の主キーeno を含む必要がある。例えば、Emp3 = Emp[ename, dno]を考え、Emp1とEmp3の二つの断片に分割するとする［図4.2］。図4.2に示すように、Emp1 * Emp3 はEmp と異なったテーブルとなってしまう［図4.2］。

Emp1=Emp[eno, ename]

Emp1	eno	ename
	1	A
	2	B
	3	A

DBS_1

Emp3=Emp[ename, dno]

Emp2	ename	dno
	A	1
	B	2
	A	2

DBS_2

Emp1 * Emp3 (≠ Emp)

Emp	eno	ename	dno
	1	A	1
	1	A	2
	2	B	2
	3	A	1
	3	A	2

図4.2 正しくない垂直断片化

ここで、どのような属性ごとに断片とするかが問題となる。例えば、図4.1で、ename のみを利用する検索については、Emp1を検索すればよくなる。dno のみを利用する検索はEmp2に対して行われるので、負荷を分散できる利点がある。しかし、ename とdno を利用する検索に対しては、Emp1とEmp2の両方が必要であり、例えば、DBS_2からEmp2をDBS_1に送信する必要がある。通信時間がかかることから、断片化せずにEmpを検索した方が検索時間が短くなる。このように、垂直断片化するためには、各応用が複数の断片を利用しなくてよいように断片を定める必要がある。

4.2 水平断面化

論理的テーブルの射影を断片としていたが、行の集合である制約を断片とすることもできる。これを水平断片化とする。例えば、図4.1のテーブルEmp を、dno = 1の行の集合Emp1 = Emp[dno = 1] とdno = 2のEmp2 = Emp[dno = 2] に分割し、各々を異なったデータベースサーバDBS_1とDBS_2に記憶する［図4.3］。dno = 1についてのデータ操作はDBS_1

に対して行えて、dno = 2についてはDBS₂に対して行える。このように、負荷を二つのサーバに分散できる。さらに、Emp 全体を探索する場合に、Emp1とEmp2を並列に探索できる。

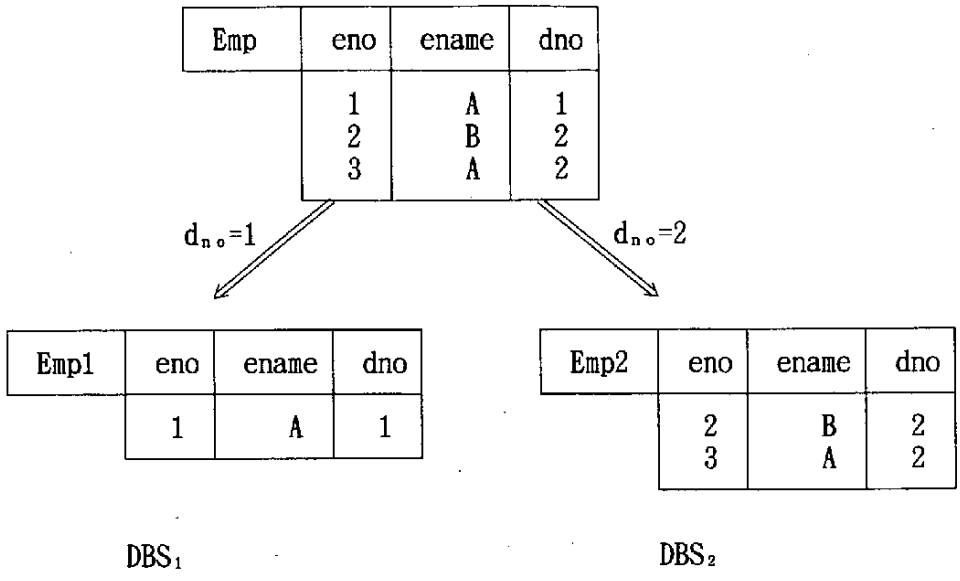


図4.3 水平断片化

実際には、垂直と水平の断片化の組み合わせを断片として用いることができる。

4.3 冗長化

断片を、複数のデータベースシステムに記憶することを、冗長化とする。冗長化により、断片が自分のデータベースシステムに存在するので、通信コストを低減できる。また、障害に対しても信頼性と可用性を向上できる。しかし、更新演算に対して、複数の断片間の一致性を保つための制御が必要となる。

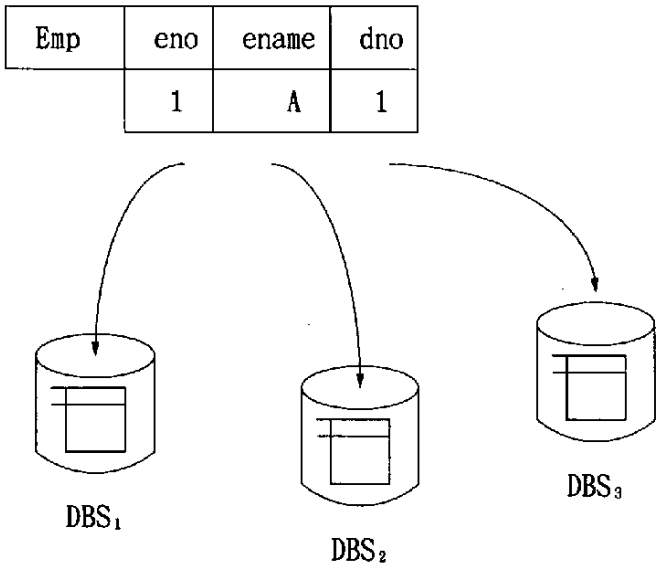


図4.4　冗長化

4.4　視　野

断片化と冗長化は、論理的テーブルをどのように複数のデータベースサーバに格納するかの方法であった。即ち、トップダウンの設計である。これに対して、各データベースシサーバ内にあるテーブルから、どのように全体層の論理的テーブルを与えるかを考える。このとき、データベースサーバ内の物理的テーブルに対する視野として、論理的テーブルを与える。

例えば、データベースサーバDBS_1がテーブルEmp を持ち、DBS_2がDeptを持つ場合を考える。これに対して、図2.6に示すような視野DE(name, dname) を定義する。利用者は、Emp とDeptの所在と独立に検索を行える。Emp とDeptに対する検索をどのように実行するか、例えば、DeptをDBS_2からDBS_1に送信してDBS_2でEmp と結合を行うかはシステムが行う。

DEを通じて、更新を行うことは、問題がある。たとえば、DEに対してname = Aでdname = DDD である行の更新を行うとする。このとき、Emp からename = A の行を消去し、Deptかかdname = DDD である行を消去すると、DEからdname = DDD のすべての行が消えてしまう。こうように、DEにename = Aでdname = HHH の行を追加すると、Deptに新しくdbame = HHH なる行を追加せねばならない。しかし、Deptの主キーdno の値が未定であるために、実態インテグリティ制約を乱すことからDeptに追加を行えない。

4.5　オブジェクト指向モデルでの分散

オブジェクト指向モデルは、クラスとインスタンスから構成される。クラスは、リレーショナルモデルのテーブルのスキーマに対応する。さらに、このスキーマに対する操作演算または手続きが一体化したものがクラスである。手続きをメソッドともいう。インスタンスは、あるクラスに対して与えられるもので、クラスの属性に具体的な値を与えた物である。リレーショナルモデルでは、テーブルの各行がインスタンスである。

オブジェクト指向モデルのデータベースシステムを分散させる問題では、以下の二点を検討する必要がある。

(1)　クラスの分散。

(2)　インスタンスの分散。

クラスの分散（分割）は、リレーショナルモデルにおける垂直分割に対応できる。クラスC の属性集合を {A_1, ..., An } とし、メソッドの集合を {M_1, ..., Mm } とする。各データベースサーバDBSiに、属性の一部と、メソッドの一部を格納する。

これに対して、インスタンスの分散では、あるクラスのインスタンスを複数のデータベースサーバに配置することである。これは、リレーショナルモデルでの水平分割に対応する。

5章　トランザクション

データベースシステムの処理単位であるトランザクション(transsaction)とは何かについて考える。

5.1　トランザクションの定義

まず、トランザクションの定義を行う。利用者が、複数のデータベースサーバDBS_1, ..., DBS_m内のデータを操作する問題を考える。各DBS_i内のデータベースDB_i をオブジェクトの集合とする。オブジェクトとは、各DBS_iの基本的な操作単位である。リレーショナルデータベースシステムでは、テーブル、断片、行がオブジェクトである。何をオブジェクトとするかは、各サーバにより決まる。オブジェクトをテーブル、行、属性値の何とするかを、粒度(granularity) という。

クライアントから各DBS_iに、SQL が発行されるが、ここでは、DBS_iの演算を抽象化して考える。各DBS_iのオブジェクトx に対する基本操作としては、以下の演算がある。

[基本操作演算]

(1) ri[x]　オブジェクトxの値の読み出す(read)演算。

(2) wi[x]　オブジェクトxに値を書き込む(write)演算。

ri[x] は、x に対するSQL のselect文に対応し、wi[x] はupdate、insert、delete文に対応する。

クライアントの利用者は、これらの操作演算をサーバに対して発行する。例として、銀行A からB に送金することを考える。送金を行う場合には、A の口座とB の口座の二つの更新を行う必要がある。両方の更新を正しく行えたときのみ、送金を完了できる。いづれか一方の更新を行えないときには、送金を行えない。このような複数の操作演算の実行系列をトランザクションという。以上の基本操作演算に加えて、トランザクションの終了を行うための演算として以下のものがある。

(1) c(commit) トランザクションを正常終了させる演算である。

(2) a(abort) トランザクションを異常終了させる演算である。

アボートa は、実行する前までに実行された更新演算を無効とする演算である。これに対して、コミットc はこれまでに実行された更新演算を完了する演算である。例えば、A の口座の更新後に、B の口座の更新を行えないことがわかったとき、a を実行することによりA の口座の更新を無効とできる。コミットメントc とアボートa は、おのおのSQL の commit_workと abort_work文の実行を示す。A とB の両方の口座の更新を完了したときに、トランザクションはコミットする。

以上をまとめると、トランザクションT とは、分散型データベースシステム内のオブジェクトに対する一つ以上の基本演算の系列の実行状態である。但し、以下の性質を充足するものである。

[トランザクションの性質(ACID)]

(1) 原子性(atomicity): T内の全操作演算が実行されるか、全くされないかのいづれかである。

(2) 一貫性(consistency): Tを単独で実行させたときに、データベースのインテグリティ制約を保つ。

(3) 独立性(isolation): Tの中間結果を他のトランザクションは見れない。

(4) 耐久性(durability): Tが正常終了した時、更新結果はデータベースから消失しない。

トランザクションの例として、銀行での送金を考える。

[例] トランザクションの例を考える。3つのデータベースサーバDBA_1, DBS_2, DBS_3があるとする。各データベースサーバは、各個人の名前(name)とその人の口座の残金(account)を示す次のテーブルBankを持つ。

Bank(name, account)

ここで、ある個人A が、DBS_1とDBS_2内の自分の口座から、おのおのx とy 円を引き出して、その合計をDBS_3内のB 氏の口座に振り込むとする。これは、SQL により以下のように記述できる。

```
select account into t1 from DBS1.Bank
   where name = "A";
if t1 - x < 0, then abort_work;                    (1)
update account = account -x from DBS1.Bank
   where name = "A";
select account into t2 from DBS2.Bank
   where name = "A";
if t2 - y < 0, then abort_work;                    (2)
update account = account -y from DBS2.Bank
   where name = "A";
update account = account +x +y from DBS3.Bank
   where name = "B";
commit_work;                                       (3)
```

ここで、abort_workは、トランザクションの異常終了を行わせる演算であり、これまでの更新結果を無効とする。例えば、この例では、(2)で、DBS_1のBankを更新した後に、DBS_2のBankが残高不足のために更新出来なかった場合に、DBS_1の更新を無効としている。commit_workはトランザクションの正常終了を示す。正常終了すると更新が完了する。このプログラムが実行する基本演算の系列がトランザクションT である。正常終了した場合のトランザクションT を以下に示す。ここで、a1とa2は、おのおのDBS_1とDBS_2内のA についてのオブジェクト（行）を示す。また、b はDBS_3内のB についてのオブジェクトを示す。

```
r1[a1]; w1[a1]; r2[a2]; w2[a2]; w3[b]; c;
```

すべての操作演算を正しく行えるときにだけ、T は正常終了する。これが原子性である。

T が正常に終了したときには、DBS_1、DBS_2、DBS_3のBank内の値は更新される。これを変更するためには、変更を行う他のトランザクションを実行する必要がある。これが耐久性である。さらに、DBS1とDBS2の更新の後でB の更新を行う前に、他のトランザクションがAの口座を操作することができないことが、独立性である。T を単独で実行した場合、A の口座の値をB に移しているので、データベースのインテグリティ制約を保つ。これが、一貫性である。

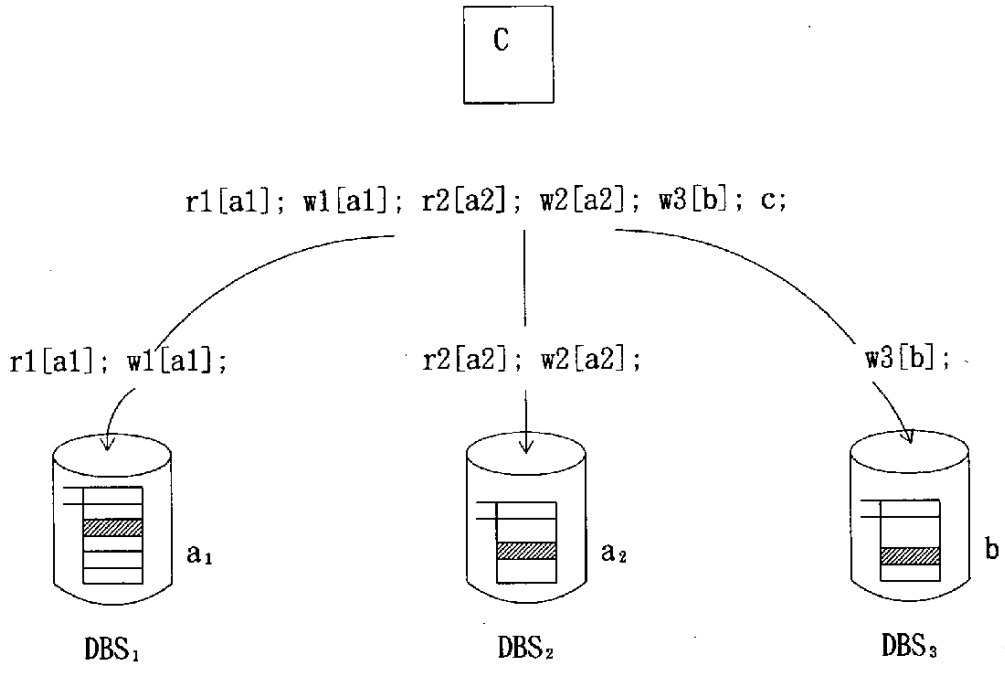


図5.1　トランザクション

トランザクションT 内の演算は、クライアントCからサーバDBS_1、DBS_2、DBS_3に送信され、各サーバで実行される。各サーバDBS_iで実行されるT の演算の系列を副トランザクションT_iとする。図5.1で、T の副トランザクション$T_1 = r_1[a1]; w_1[a1]$、$T_2 = r_2[a2]; w_2[a2]$、$T_3 = w_3[b]$である。

5.2 ロ グ

複数のトランザクション$T_1,\ldots,T_m$が、複数のデータベースサーバ$DBS_1,\ldots,DBS_n$ 内のオブジェクトを操作する場合について考える。各サーバDBS_jは、複数のトランザクションからの演算が一つづつ直列に実行される。この実行履歴を、DBS_jのローカルログL_jとする。ローカルログ$L_1,\ldots,L_n$の集合を$T_1,\ldots,T_m$のログL とする。ログL の性質について考える。

［**正しいログ**］インテグリティ制約を充足しているデータベースを正しいとする。正しいデータベースに対して、ログL を実行させた結果、正しいデータベースをもたらすならば、ログL は正しい。

まず、トランザクション間のコミット／アボートの関係について考える。ここで、x をDBS_j内のオブジェクトとする。また、$r_j[x]$ 、$w_j[x]$ 、a_j、c_jを各々トランザクションT_jのオブジェクトx に対するread、write 、abort 、commit演算とする。

［**トランザクション間の読みだし関係**］L をトランザクション$T_1,\ldots,T_m$のログとする。トランザクションT_jがT_kからオブジェクトx を読むとは、以下の条件を満足することである［図5.2］。

(1) L_j内で、$w_k[x]$ が$r_j[x]$ より先行する。即ち、$w_k[x]$ が$r_j[x]$ より先にDBS_jで実行される。

(2) $r_j[x]$ の前に、a_kは実行されていない。

(3) $w_k[x]$ の後で、$r_j[x]$ に先行して$w_h[x]$ が存在するならば、$a_h < r_j[x]$ の実行前にT_hはアボートしている。

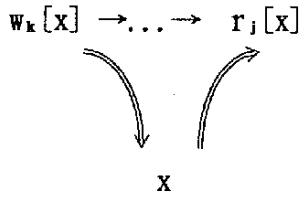


図5.2 読みだし関係

即ち、トランザクションT_kが書いたx の値を、T_jが読んでいるとき、T_jはT_kからx を読むという。定義の(3)は、T_kがx を書いた後に、他のトランザクションT_hがx を書いていても、T_jがreadする前に、T_hがアボートしているならば、T_kがx に書いた値をTjは読む事になることを述べている。

次にログの性質について考える。

[復旧可能ログ] L をログとする。このとき、L 内で、トランザクションT_jがT_kからデータオブジェクトx を読み、c_kをL が含んでいるならば、$c_k < c_j$ であるとき、L は復旧可能(recoverable) という。

L_1: ... $w_k[x]$ ... c_k ... $r_j[x]$ ... c_j ...

L_2: ... $w_k[x]$ ... $r_j[x]$ ... c_k ... c_j ...

L_3: ... $w_k[x]$ ... $r_j[x]$ ... c_j ... a_k ...

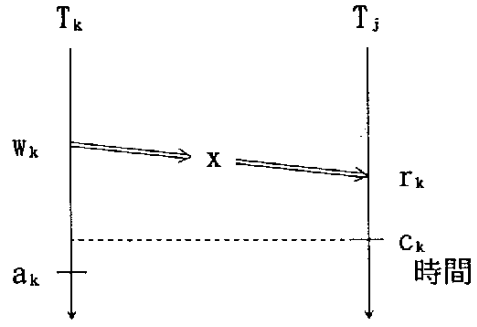


図5.3 復旧不可能

上記のログを考える。L_1では、トランザクションT_jは、T_kが書いた値をコミットした後に読んでいるので、復旧可能なログである。これに対して、L_2とL_3では、T_jは、T_kがコミットする前にT_kが書いた値を読んでいる。L_3では、T_jがコミットした後に、T_k がアボートしている。結果として、T_jは存在しない値を読んだことになってしまう。従って、このようなログでは、トランザクションのアボート等に対して、復旧を行えないので、復旧不可能なログという［図5.3］。ここで、トランザクション集合T に対して復旧可能なログの集合をRCとする。

[連鎖的なアボートを起こさないログ] トランザクションT_jがT_iからログL 内でx を読むならば、$c_k < r_j[x]$ であるとき、L は連鎖的なアボート(cascading abort)を起こさないという。

L_4: ... $w_k[x]$ ... c_k ... $r_j[x]$ ... c_j ...

L_5: ... $w_k[x]$ ... $r_j[x]$ ... c_k ... c_j ...

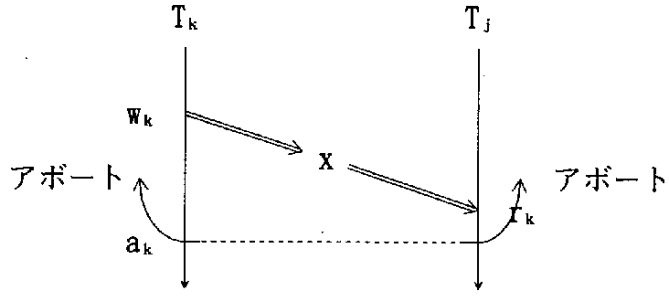


図5.4 連鎖的アボート

上記のログL_4は、連鎖的なアボートを起こさないログである。しかし、L_5では、$r_j[x]$ の後に、T_kがアボートしたならば、T_kもアボートしなければならない。従って、連鎖的なアボートを起こすログである［図5.4］。例からわかるように、連鎖的なアボートを起こさないログは、復旧可能である。ここで、トランザクション集合T に対して、連鎖的なアボートを起こさないログの集合をACA とする。

[厳格なログ] $w_k[x] < op_j[x]$ ならば、$a_k < op_j[x]$ 又は$c_k < op_j[x]$（ここで、$op_j[x]$ は$r_j[x]$ 又は$w_j[x]$ である）であるとき、ログL は厳格(strict: ST)であるという。

L_6: ... $w_k[x]$ ... a_k ... $op_j[x]$ ...

L_7: ... $w_k[x]$ ... c_k ... $op_j[x]$ ...

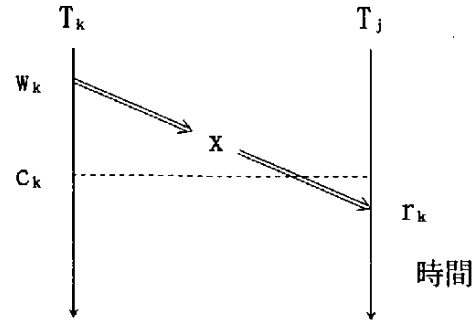


図5.5　厳格なログ

L_6とL_7は、厳格なログである。例からわかるように、厳格なログは、復旧可能であり、かつ連鎖的なアボートを起こさない。ここで、トランザクション集合T に対して、厳格であるログの集合をSTとする。以上より、ST、ACA 、RAの間には、以下の関係が成り立つ。

[定理]　ST ⊆ ACA ⊆ RC。

即ち、厳格なログは、連鎖的アボートを生じない。また、連鎖的アボートを起こさないログは、復旧可能である。ただし、この逆は、成り立たない。ログが厳格でないと、あるトランザクションのアボートにより、他のトランザクションが連鎖的にアボートされる可能性がある。このために、現在のデータベースシステムでは、トランザクションは、厳格なログとなるように、トランザクションの終了時、すなわち、コミットまたはアボート時に、すべてのロックを解放している。ロックについては、次の章で述べる。

6章　同時実行制御(concurrency control)

複数のトランザクションが、複数のデータベースサーバ内のオブジェクトを操作する問題について考える。このとき、各データベースサーバ内のデータベースがインテグリティ制約を保った状態に保つとともに、システム全体のスループットを向上させる必要がある。このための制御が、同時実行制御である。

6.1　直列可能性

分散型データベースシステムは、データベースサーバ$DBS_1,\ldots,DBS_n$ を含むとする。複数のトランザクション$T_1,\ldots,T_m$ が、これらのサーバ内のオブジェクトを更新する。各サーバDBS_jは、複数のトランザクションの演算をインタリーブ実行する。このように、複数のトランザクションが複数のサーバにより並列実行されることの正しさの根拠として、直列可能性(serializability)[BERN87, LYNC93] がある。

$op_{ij}[x]$ をT_iの演算で、DBS_j内のオブジェクトx に対する演算とする。opはr(read) またはw(write)である。同じオブジェクトx に対する$op_{ij}[x]$ と$op_{kj}[x]$ を考える。

[競合演算] 二つの演算の実行順序により演算結果が異なるとき、$op_{ij}[x]$ と$op_{kl}[x]$ は競合するという。

例えば、r[x]とw[x]、w[x]とw[x]は、互いに競合する。r[x]の次にw[x]を実行した場合と、w[x]の後にr[x]を実行した場合では、x の値は同じでも、r[x]する値が異なったものとなる。従って、競合する演算の実行順序を、各サーバでどのような順序とするかが重要な問題となる。

トランザクション集合$T_1, \ldots, T_m$ のログL は、各DBS_j毎のローカルログL_jの集合である。L_jは、DBS_jに対する操作演算の実行系列である。あるトランザクションが完了してから次のトランザクションが一つづつ実行されることを直列実行とする。各トランザクションが完了したとき、分散型データベースシステム内のデータベースを正しい状態に保つ。このことから、複数のトランザクションが直列に実行された場合も、データベースを正しい状

態に保つ。

ログL に対するトランザクション間の順序関係$<<_l$を定義する。

［**トランザクション間の順序関係**］二つのトランザクションT_jとT_kに対して、以下が成り立つとき、$T_j <<_l T_k$ とする。即ち、あるL_hで、1) T_jの演算op_{jh}とT_kの演算op_{kh}を含み、op_{jh}とop_{kh}が競合し、2) op_{jh}がop_{kh}がDBS_hで先行して実行される。

このとき、$T_1,\ldots,T_n$の実行を示すログL が直列可能であることを定義する。

［**直列可能性**］ログL について、$<<_l$ が全順序関係ならば、L は直列可能である。

例について考える。

［例］データベースシステムサーバDBS_1がオブジェクトx とy を持ち、DBS_2がz を持つとする［図6.1］。次の二つのトランザクションT_1とT_2がDBS_1とDBS_2に対して実行されるとする。

$$T_1: r_{11}[x];\ w_{12}[z];\ r_{11}[y];\ c$$
$$T_2: r_{22}[z];\ w_{21}[x];\ r_{21}[y];\ c$$

DBS_1とDBS_2に対するローカルログL_1とL_2の例を以下に示す。

$$L_1: r_{11}[x];\ w_{21}[x];\ r_{11}[y];\ r_{21}[y];$$
$$L_2: r_{22}[z];\ w_{12}[z];$$

L_1とL_2から構成されるログL は、直列可能ではない。何故ならば、DBS_1で、x の競合演算は、T_1の$r_{11}[x]$がT_2の$w_{21}[x]$より前に実行されているが、DBS_2では逆にT_2の$r_{22}[z]$がT_1の$w_{12}[z]$に先行しているからである。しかし、以下に示すローカルログM_1とM_2のログM は直列可能である。DBS_1のx については、T_1の$r_{11}[x]$がT_2の$w_{21}[x]$に先行している。DBS_2のz

については、T_1の$w_{12}[z]$がT_2の$r_{22}[z]$に先行している。y については、T_2の$r_{21}[y]$がT_1の$r_{11}[y]$に先行しているが、これらは競合しない。従って、DBS_1とDBS_2における競合演算は、T_1がT_2よりも先行している(T_1 <<M T_2)。このことは、T_1を実行した後に、T_2を直列に実行した場合と同じ結果が得られる。

M_1: $r_{11}[x]$; $w_{21}[x]$; $r_{21}[y]$; $r_{11}[y]$;

M_2: $w_{12}[z]$; $r_{22}[z]$;

同時実行制御の問題は、複数のトランザクションを、複数のデータベースシステムで実行するとき、ログを直列可能とすることが必要となる。このための方式として、二相ロックと時刻印順序化がある。

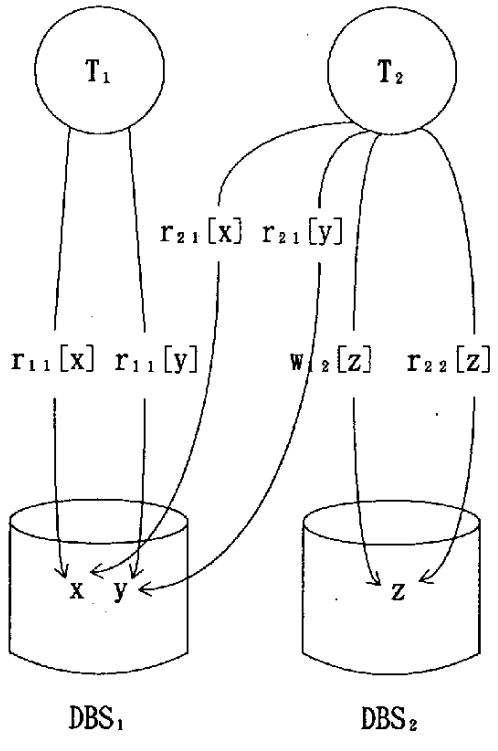


図6.1 直列可能性

各D_iが自律性を持つ場合には、演算の実行順序は、全体の順序に基づくとは限らない。このために、擬直列可能性[DU89]がある。

6.2 二相ロック

ログを直列可能とする代表的方法としては、二相ロック方式[ESWA]がある。オブジェクトx に対して、新しくロックl[x]とロックの解放u[x]を導入する。トランザクションは、以下の規則に従って、オブジェクトx に対して、ロック演算l[x]とロック解放演算u[x]を適用する。

[ロックプロトコル]

(1) 各トランザクションT_iは、オブジェクトx を操作する前に、l[x]によりロックを行う。他のトランザクションT_jがx をロックしている場合は、T_iはT_jのロックの解放u[x]を待つ。

(2) T_jは、終了前に、T_jが獲得した全ロックの解放を行う。

二相ロック方式を以下に示す。まず、冗長なオブジェクトが存在しない場合を考える。T_iをトランザクションとする。

[二相ロック方式]

(1) T_i内の各演算$op_{ij}[x]$ に対して、DBS_j内のx をロック($l_{ij}[x]$)する。

(2) T_iがコミットまたはアボートするとき、T_iのロックをすべて解放($u_{ij}[x]$)する。

二相ロックトランザクションの例を以下に示す。

[例] 図6.1に示す分散型データベースシステムに対する以下の二つのトランザクションを考える。これらは、二相ロック形式である。

T_1: $l_{11}[x]$; $r_{11}[x]$; $l_{12}[z]$; $w_{12}[z]$; $l_{11}[y]$; $u_{12}[z]$; $r_{11}[y]$; $u_{11}[y]$; $u_{11}[x]$; c;

T_2: $l_{22}[z]$; $r_{22}[z]$; $l_{21}[x]$; $w_{21}[x]$; $l_{21}[y]$; $r_{21}[y]$; $u_{21}[y]$; $u_{21}[x]$; $u_{22}[z]$; c;

T_1とT_2では、ロックの解放演算が実行された後にロック演算が実行されていない。したがって、T_1とT_2は、二相ロックトランザクションである。次のトランザクションT_3は二相ロック形式ではない。x のロックを $u_{11}[x]$ により解放した後に、z とy のロックが行われ

ているからである。

T_3: $l_{11}[x]$; $r_{11}[x]$; $l_{12}[z]$; $u_{11}[x]$; $w_{12}[z]$; $l_{11}[y]$; $u_{12}[y]$; $r_{11}[y]$; $u_{11}[y]$; c;

DBS_jでのT_iの操作演算の系列T_{ij}を、T_iの副トランザクションとする。各T_{ij}が二相ロック形式であることは明らかである。T_1とT_2の副トランザクションは、以下のようになる。

T_{11}: $l_{11}[x]$; $r_{11}[x]$; $l_{11}[y]$; $r_{11}[y]$; $u_{11}[x]$; $u_{11}[y]$; c;
T_{12}: $l_{12}[z]$; $w_{12}[z]$; $u_{12}[z]$; c;
T_{21}: $l_{21}[x]$; $w_{21}[x]$; $l_{21}[y]$; $r_{21}[y]$; $u_{21}[y]$; $u_{21}[x]$; c;
T_{22}: $l_{22}[z]$; $r_{22}[z]$; $u_{22}[z]$; c;

これらは、二相ロック形式である。T_{11} とT_{21} のDBS_1でのインタリーブ実行を示すローカルログN_1と、T_{12} とT_{22} のDBS_2でのインタリーブ実行を示すローカルログN_2を以下に示す。

N_1: $l_{11}[x]$; $r_{11}[x]$; $l_{11}[y]$; $u_{11}[x]$; $l_{21}[x]$; $u_{11}[y]$; $w_{21}[x]$; $l_{21}[y]$; $r_{21}[y]$;
$u_{21}[x]$; $u_{21}[y]$;
N_2: $l_{12}[z]$; $w_{12}[z]$; $u_{12}[z]$; $l_{22}[z]$; $r_{22}[z]$; $u_{22}[z]$;

副ログN_1とN_2から成るログでは、どの競合演算も、T_1のものがT_2よりも先に実行されている。即ち、T_1 $<<_N$ T_2 である。従って、T_1の実行後に、T_2を実行した場合を示す以下の直列ログと同値である。

S_1: $l_{11}[x]$; $r_{11}[x]$; $l_{11}[y]$; $r_{11}[y]$; $u_{11}[x]$; $u_{11}[y]$; $l_{21}[x]$; $w_{21}[x]$; $l_{21}[y]$;
$r_{21}[y]$; $u_{21}[y]$; $u_{21}[x]$;
S_2: $l_{12}[z]$; $w_{12}[z]$; $u_{12}[z]$; $l_{22}[z]$; $r_{22}[z]$; $u_{22}[z]$;

二相ロック形式のトランザクションについて、以下の結果が得られている。

［**定理**］どの副トランザクションも二相ロック形式ならば、得られるログは直列可能である。

即ち、どのトランザクションも二相ロック形式ならば、これらのトランザクションを複数のデータベースサーバで実行した結果は正しい。

6.3　ロックのモード

オブジェクトx を、あるトランザクションT_iがreadをする場合を考える。他のトランザクションT_jもx をreadするとき、T_iがx をロックしているために、Tjがx を利用できない。しかし、read演算はどのような順序で実行されても結果は同じである。このために、各演算opに対して、ロックモードmode(op)を考える。一般的に、x に対する二つの演算op_1 とop_2 をどのように実行しても結果が同じであるとき、op_1 とop_2 は、交換可能とする。交換可能な演算op_1 とop_2 のロックモードmode(op_1)とmode(op_2)は伴立である(compatible)とする。演算op_1 と伴立なモードでオブジェクトがたの演算op_2 ロックされているときには、op_1 はx を操作できる。そうでないときには、op_2 によるロックの解放を待たねばならない。例えば、read演算のロックモードRlock は互いに伴立であるが、write 演算のロックモードWlock は、Wlock ともRlock とも伴立ではない。これらのロックモード間の伴立関係を図6.2 に示す。

	Rlock	Wlock
Rlock	○	×
Wlock	×	×

○　互換
×　非互換

図6.2　ロックモードの伴立性

6.4　冗長なオブジェクトが存在する場合の二相ロック

次に、x が冗長な物理的オブジェクトx_1, ..., x_mを持つ場合を考える。例えば、テーブルEmp の断片Empl = Emp[eno, ename] が、二つのデータベースサーバDBS_1とDBS_2に存在する場合である。一つの方法は、r[x]とw[x]を行う前に、x_1, ..., xmを全てロックすることであるが、ロックする負荷が大きくなる。このために以下の方法がある。

(1)　x のreadに対しては、あるxjをRlock モードでロックしてから、$r_{ij}[x_j]$を実行する。
(2)　x のwrite に対しては、全てのx_1, ..., x_mをWlock モードでロックし、x_1, ..., x_mをwrite する。

この方法では、自分のまたは近くのD_j内のx_jをreadできる。このために、検索中心の応用に適した方法である［図6.3(1)］。
もう一つの方法は、物理的オブジェクトの中の一つ、ここではx_1を主として、ロックは主オブジェクトに対して行う方法もある［図6.3(2)］。

(1)　x のreadに対しては、x_1をRlock モードでロックしてから、あるx_jを$r_{ij}[x_j]$する。
(2)　x のwrite に対しては、x_1をWlock でロックし、すべてのx_1, ..., x_mをwrite する。

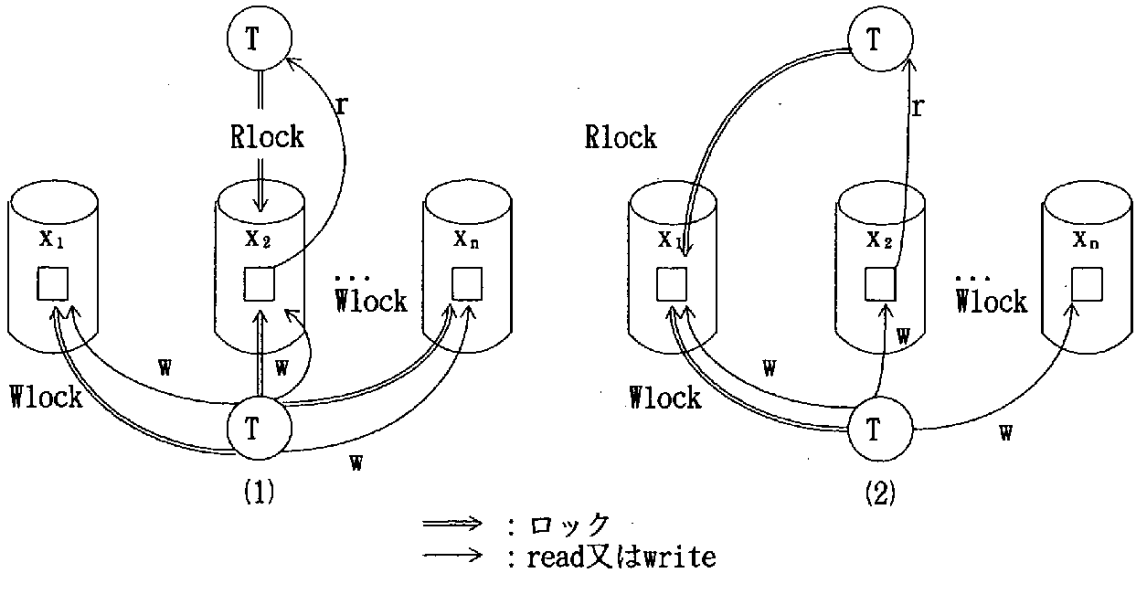


図6.3　二相ロック方式

6.5　デッドロック

デッドロック(deadlock)とは、複数のトランザクションが他のトランザクションのロックの解放を待ちあっている状態である。

6.5.1　デッドロックの定義

トランザクション間のデッドロックは、以下に示す待ちグラフ(wait-for graph)により検出できる。

[待ちグラフ]

各トランザクションT_iに対して、節点Tiを設ける。T_iがT_jによりロックされているオブジェクトの解放を待っているならば、T_iからT_jに有向辺を設ける($T_i \rightarrow T_j$)。

三つのトランザクションT_1、T_2、T_3があり、おのおのがオブジェクトx、y、z をロックしているとする。このとき、T_1がy を、T_2がz を、T_3がx のロックを行おうとしているとする。ここで、T_1はT_2のy の解放を待ち、T_2はT_3のz の解放を待ち、T_3はT_1のx の解放を待つ。この関係を示す待ちグラフを、図6.4に示す。

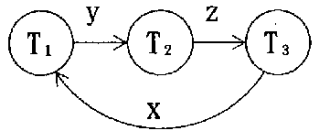


図6.4 デッドロック

図6.4に示す状態は、T_1、T_2、T_3が互いに待ちあっていることからデッドロック状態である。図6.4をみると、巡回有向閉路が存在していることがわかる。

[デッドロック]

待ちグラフが巡回閉路を含むならば、かつこのときに限りシステムはデッドロックしている。

次に、図6.1に示すように、DBS_1がオブジェクトx とy を持ち、DBS_2がオブジェクトz を持つ場合を考える。図6.5に示したように、三つのトランザクションT_1、T_2、T_3があり、おのおのがオブジェクトx 、y 、z をロックしているとする。このとき、T_1がy を、T_2がz を、T_3がx のロックを行おうとしているとする。DBS_1を見てみると、x とy を利用しているトランザクションT_1、T_2、T_3間には、巡回有向閉路は存在しない。DBS_2も同様である。しかし、分散型データベースシステムとしては、図6.4に示すようにデッドロックとなっている。このように、分散型データベースシステムでは、各データベースサーバ内でデッドロックとなっていなくても、システム全体としてみたときに、デッドロックとなっていることがある。分散型データベースシステムでデッドロックを検出するためには、システム全体の待ち関係を把握するために、データベースサーバ間、トランザクション間での通信が必要となる。

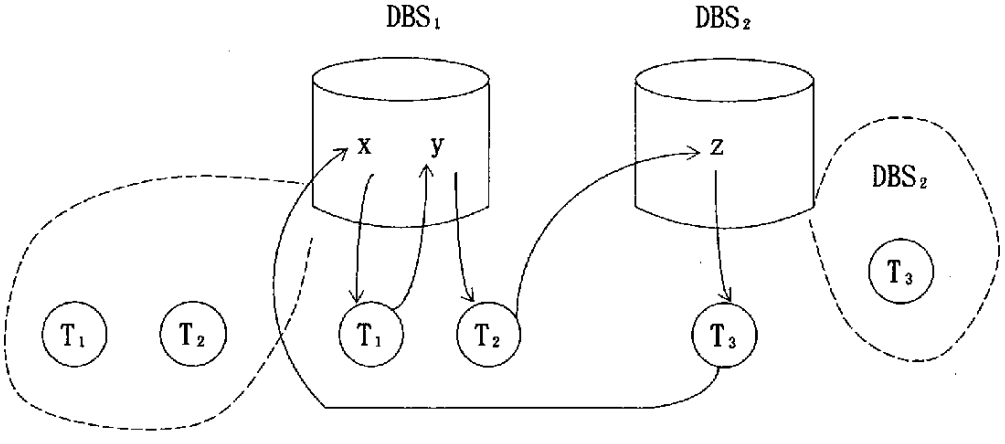


図6.5　分散型データベースシステムのデッドロック

6.5.2　デッドロックの解決方法

デッドロックの解決方法としては、以下がある。

(1)　検出方法

(2)　防止方法

検出方法では、デッドロックを検出して、デッドロック状態にあるトランザクションをアボートしてデッドロックを解消する。防止方法では、デッドロックが欠っして生じないようにする方法である。

A．検出方法

検出方法では、デッドロックを検出して、デッドロック状態を解消する方法である。この方法は、商用のシステムで広く用いられているものである。

［デッドロック検出・解除方法］

(1) 現在の状態に対する待ちグラフGを作り、巡回閉路を探す。

(2) 見付かれば、この閉路内のあるトランザクションT を（生贄として）選択し、T をアボートさせる。

例えば、図6.4で、あるトランザクション、例えば、T1が選ばれ、これがアボートされる。T_1のアボートにより、T_1が獲得していたオブジェクトx が解放される。これにより、トランザクションT_3は、x を獲得できるので、待ち状態から実行可能状態となり、デッドロックが解消される。

デッドロックが生じていなくても、デッドロック検出手順がデッドロックが生じていると見なしてしまう状態を偽デッドロック(false deadlock 又はphantom deadlock）という。これは、分散型のシステムでは、システム内の全ての要素の状態を一時に把握することができないことによっている。状態を、順々に、各データベースサーバに問合せを行っていくと、はじめに調べたデータベースサーバの状態が現在は異なっている場合がある。例えば、あるトランザクションが別のオブジェクトを待っていたが、タイムアウトによりアボートしてしまう場合がある。ただし、以下の性質が示されている。

［定理］ 各トランザクションが二相ロック形式であり、頻繁にアボートしないならば、偽デッドロックは生じない。

デッドロックの検出・解除方法では、以下が問題となる。

(1) 検出の間隔

周期的にデッドロック検出を行う方法が用いられる場合が多い。このとき、周期をどのくらいにするかが問題となる。周期を短くすると、デッドロック検出のための負荷が増大し

てしまう。一方、周期を長くすると、最悪の場合、この期間の間デッドロックが生じてしまう。

(2) **強制終了されるトランザクションの選択**

デッドロックが検出されたとき、どのトランザクションを強制終了させるかが問題となる。一番実行されている時間の短いもの、長いもの、ロックしているオブジェクト数が最少のもの等を選択する方法がある。

6.5.3 デッドロックの検出

デッドロックを検出するために、データベースサーバ間での通信が必要となる。各データベースサーバの状態を集め、トランザクション間の待ちグラフを作り、巡回閉路を見つけることにより、デッドロックを検出する。デッドロックが検出されたならば、巡回閉路内のあるトランザクションを選択し、これをアボートする。

この他に、以下の方法がある。

(1) 待ち状態でタイムアウトしたトランザクションT_jは、トークンを待っているオブジェクトx を持つサーバDBS_iに送信する。

(2) DBS_iは、x をロックしているトランザクションT_kにこのトークンを送信する。

(3) T_kは、トークンを受信したならば、(1)を行う。

(4) T_jが、自分が送信したトークンを受信したとき、自分がデッドロック状態であることがわかる。

6.5.4 防止方法

防止方法では、デッドロックが決して生じないようにする方法である。このための方法として以下がある。

(1) オブジェクトを順序付けて、トランザクションに、この順にロックさせる。

(2) トランザクションを全順序付ける。

(1)は、オブジェクトをロックする順番をあらかじめ決めておく方法である。例えば、オブ

ジェクトとして、a 、b、cがあったとき、各トランザクションは、この順にロックを行わせる方法である。この方法の問題点は、順位の高いオブジェクトにロック要求が集中することになり、トランザクション実行の同時実行性を減少させてしまう。

(2)方法について述べる。各トランザクションT に対して、T が起動された時刻をT の時刻印(time stamp) ts(T)として与える。ここで、xをオブジェクト、T_1とT_2をトランザクションとする。いま、x はT_2によりロックされているとし、新たにトランザクションT_1がxをロックしようとしている場合を考える［図6.6］。

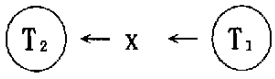


図6.6 待ち関係

防止方法として、以下の二つの方法がある。

［横取り(preemption)方法］

(1) $ts(T_1) < ts(T_2)$ ならば（T_1がT_2より旧い）、T_2 をアボートさせ、再実行させる。

(2) $ts(T_1) > ts(T_2)$ ならば、T_1を待たせる。

［非横取り(nonpreemption) 方法］

(1) $ts(T_1) < ts(T_2)$ ならば（T_1がT_2より旧い）、T_1をアボートさせ、T_1に新しい時刻印を与えて再実行させる。

(2) $ts(T_1) > ts(T_2)$ ならば、T_1を待たせる。

このデッドロック防止方法では、デッドロックは生じないが、あるトランザクションがロックしているオブジェクトをロックしよとしたときに、トランザクションのアボートと再実行が生じてしまう。トランザクションがどの位再実行されるかが問題となる。横取り方法では、トランザクションは時刻印の小さい（即ち、古い）ものが優先して実行される。これに対して、非横取り方法では、新しいトランザクションが優先して実行される。従って、オブジェクトx を操作しようとする古いトランザクションがあるとき、これらより新

しいトランザクションがx を獲得してしまうと、古いものはアボートされてしまう。このために、横取り方法の方が、一般的に、非横取り方法よりも、アボートされるトランザクションは少ない。

7章　コミットメント制御

トランザクションが複数のデータベースを更新する場合には、更新の原子性、即ち、全てが更新されたか、または全く更新されなかったかのいづれかであることを保障する必要がある［図7.1］。このための制御がコミットメント制御である。

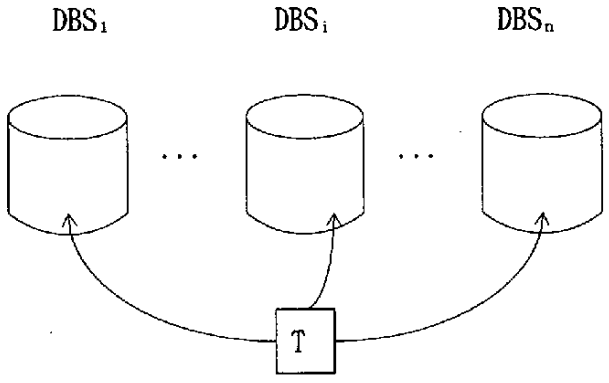


図7.1　コミットメント制御

7.1　障　害

分散型データベースシステムは、通信ネットワークで結合されたプロセスから構成されるとする。ここで、各プロセスは、データベースサーバまたはクランイアントを示す。分散型データベースシステムは、プロセスと通信ネットワークから構成されることから、障害として、プロセス障害とネットワーク障害がある。

A．プロセス障害

まず、プロセスの障害について考える。

［プロセスの障害］

(1)　停止(stop-by-failure)：プロセスが停止する障害である。

(2)　オミッション(omission)障害：プロセスが、要求に対して、応答したり、しなかったりする障害である。

(3) コミッション(comission) 障害: プロセスが要求に対してプロトコルに従わない応答を行う障害である。

プロセスの停止障害とは、プロセスが一定時間以上停止する障害である。停止した後復旧しない障害を永久停止とする。一定時間とは、後述するコミットメントプロトコルの処理時間以上の時間である。停止して直ちに復旧する障害（間欠障害）は停止障害ではなく、オミッション障害である。(2)と(3)は、プロセスが誤動作する障害であり、ビザンチン障害[LAMP**]という。

[障害についての仮定] コミットメント制御では、プロセス停止のみを障害とする。

障害していないプロセスを、動作中であるとする。プロセスがすべて障害（即ち、停止）することを全体障害とする。少なくとも一つのプロセスが動作している障害を部分障害とする。

B. ネットワーク障害

通信ネットワークは、OSI プロトコル、TCP/IPプロトコルといった通信プロトコルにより、データ単位を宛先に届けることができる。通信ネットワークが処理できない障害として、ネットワークの分割がある。ネットワーク分割とは、プロセスが複数のグループに分割され、各グループ内のプロセスは互いに通信を行えるが、グループ間では通信を行えない状態である［図7.2］。各プロセスは、他のグループは障害していると考えて独自にコミットメントの処理を行うと、グループ間で異なった決定を行ってしまう可能性がある。

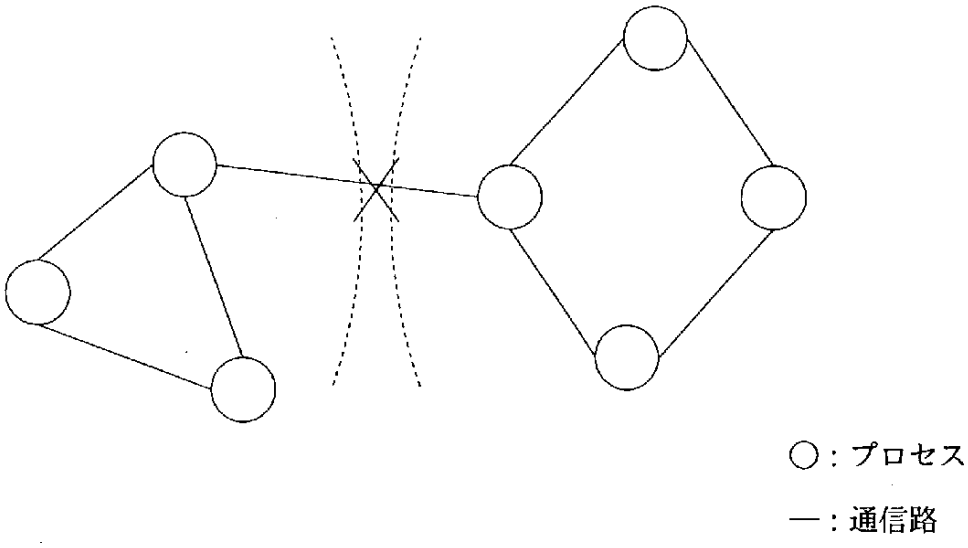


図7.2　ネットワーク分割

7.2　二相コミットメント

コミットメント制御として、二相コミットメント(two-phase commitment)[SKEE81]がある。例として、あるプロジェクトのメンバでテニスにいくことを考える。ここで、プロジェクトの全員が出席するときだけテニスに行くとする。プロジェクトのリーダは、全員に電話をし、出欠を確認する。ここで、一人でも欠席者いた場合には、出席といったメンバにテニスに行かないことを電話する。全員が出席ならば、リーダは全メンバに再度電話を行い、テニスに行くことを通知する。このように、全員出席するときのみテニスに出掛けることの決定を行うためには、出欠の意志表示を求めることと、テニスに出掛けるかどうかの決定を通知することの二つの手順が必要となる。さらに、リーダから、電話を受け、テニスに出席するとしたメンバはリーダからの電話を待つ。この状態が不確定状態である。このとき、例えば、他からゴルフに誘われてもゴルフに行くことはできない。ただ、リーダからの電話を待つことだけができる。

7.2.1　基本プロトコル

以上から、コミットメントを行うためのプロトコルを考える。プロセスがトランザクションT のコミットについて、諾又は否を他のプロセスに示すことを意志表示とする。全プロ

セスがコミットかアボートかの合意に達することを決定とする。諾表示を行った後に、全体の決定を待っているとき、プロセスは不確定状態にあるとする。プロセスが処理を続行するために、障害プロセスの復旧を待たねばならないとき、このプロセスはブロック(block) しているとする。ある動作中プロセスが不確定状態にあるならば、動作中か障害中かに関わらずコミットを決定しているプロセスはないとき、コミットメント手順は非ブロック(non-block) である。

二相コミットメント(2PC: two-phase commitment) 制御では、各トランザクションT に対して、一つの指揮プロセス (クライアント)Cと、関係プロセス (サーバ)$P_1,\ldots,P_n$がある。二相コミットメント(2PC) プロトコルの基本手順を以下に示す。

[二相コミットメント(2PC) プロトコル]

(1) クランイアントC は、全関係プロセスにVOTE-REQ(VR)を放送する。

(2) 各サーバP_jは、VOTE-REQを受信したとき、コミットできるならばYES を、クランイアントC に送信する。コミットできないならば、NOを送信してアボートする。

(3) クランイアントC は、全データベースサーバの会計プロセスからYES を受信したならば、COMMITを送信し、トランザクションT をコミットする。あるサーバP_jからNOを受信したならば、YES を受信した全サーバ (関係プロセス) にABORT を送信し、T をアボートする。

(4) 各サーバP_jは、COMMITを受信したならば、T をコミットする。ABORT をクランイアトc から受信したならばアボートする。

図7.3に、トランザクションがコミットする場合を示す。図7.4に、トランザクションがアボートする場合を示す。c からのABORT はYES を送ったデータベースサーバに対してのみ送行されている。

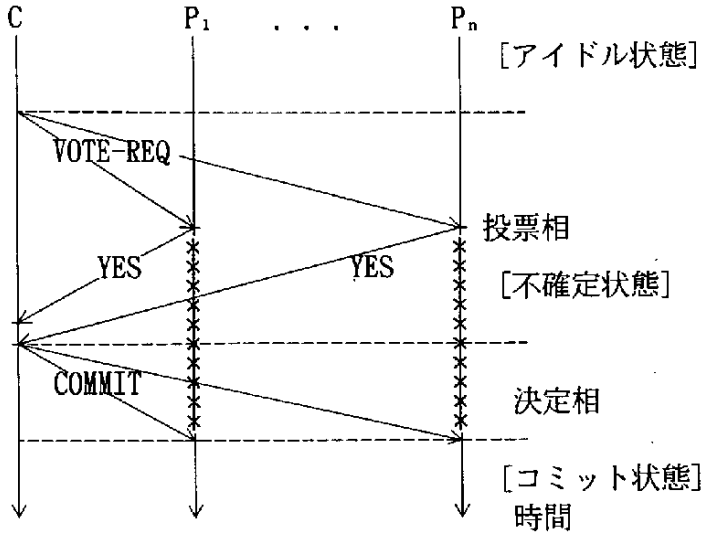


図7.3　二相コミットメントプロトコル（コミット）

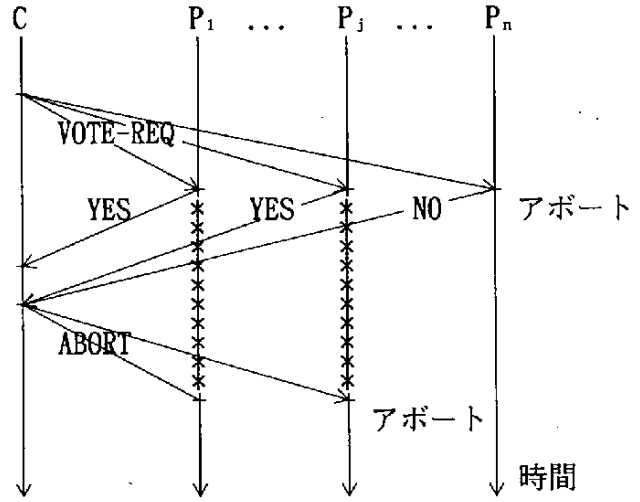


図7.4　二相コミットメントプロトコル（アボート）

7.2.2　障害に対するプロトコル

各データベースサーバ（関連プロセス）P_iが、YES を送信してから、クランイアントC からCOMMIT又はABORT を受信するまでの状態るあるときを考える。P_iがこの状態にあるとき、コミットの意志表示を他に通知しているので、意志表示に反するアボートを行えない。即ち、P_iは、C からのCOMMIT又はABORT の通知を待つだけができる。この状態を不確定状態とする。

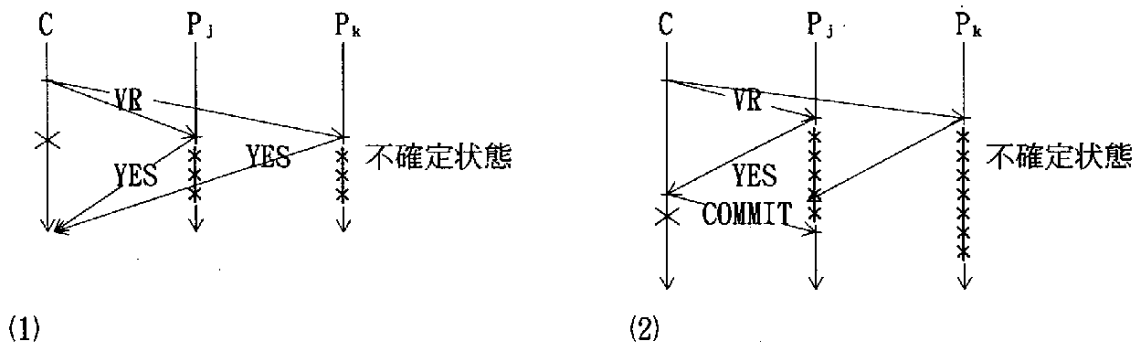


図7.5 障 害

次に、クランイアントCが障害を起こした場合を考える。VOTE-REQを送信した後に、C が障害を起こした場合には、データベースサーバ（関係プロセス）は待ち続けてしまう（ブロックする）。この問題を解決するために、データベースサーバだけにより、コミットメント制御の終了を行う手順が必要になる。これを終結プロトコル(termination protocol)という。クランイアントC がどのデータベースサーバに対しても、COMMIT又はABORT を送信する前に障害したならば、動作中のどのデータベースサーバもブロックして、C の復旧を待つ必要がある［図7.5(1)］。しかし、図7.5(2)に示すように一部のデータベースサーバに意志決定を送信した後に障害を起こした場合には、不確定状態のサーバは、確定状態にあるサーバに問い合わせを行うことにより意志決定を行うことができる。このための終結プロトコルについて考える。

複数のデータベースサーバが協調して、コミットまたはアボートの決定を行おうとする協調的終結プロトコルを以下に示す。

7.2.3 協調的終結プロトコル

不確定状態のデータベースサーバP_jが、タイムアウトしたとき以下の手順を行う［図7.6］。

(1) P_jは、すべてのデータベースサーバにSTATE-REQ(SR) を送信する。

(2) STATE-REQ(SR) を受信した動作中のデータベースサーバは、自分の状態（コミット、アボート、不確定）を送信する。

(3) P_jは、状態についての応答を受信したならば、以下の終結規則に従う。

［終結規則］

TR1: すべての動作中のデータベースサーバが不確定状態ならば、P_jは不確定状態のままで待つ。

TR2: あるデータベースサーバP_kがコミット(Committed) 状態ならば、P_j はコミットする。

TR3: あるデータベースサーバP_kがアボート(Aborted) 状態ならば、P_jはアボートする。

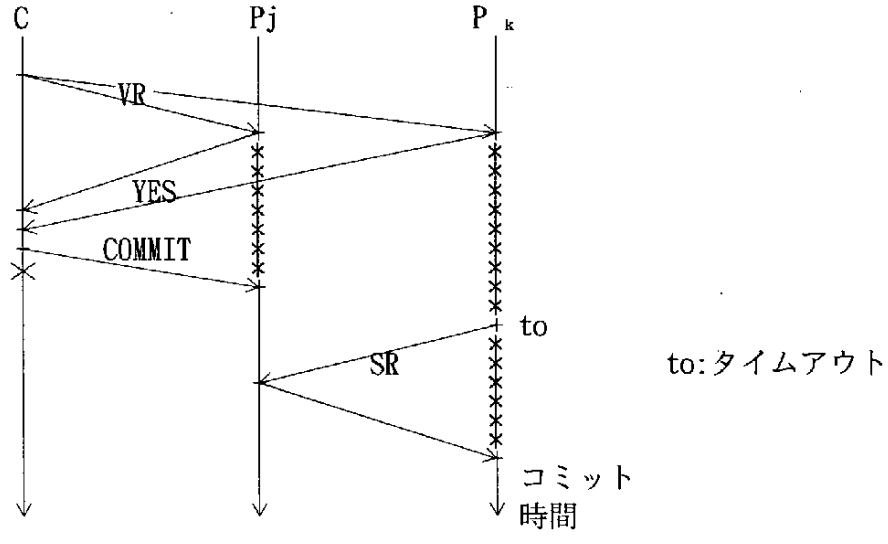
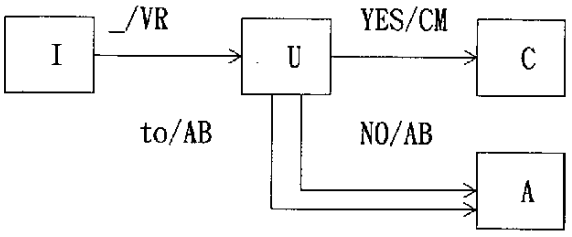


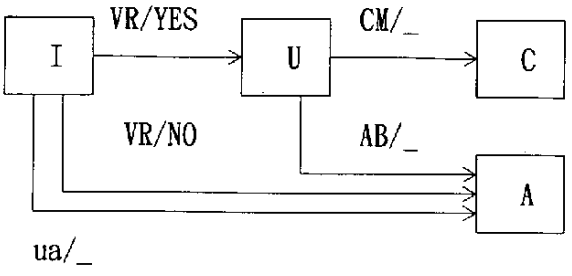
図7.6 終結プロトコル

二相コミットメントプロトコルの状態遷移図を図7.7に示す。

(1) クランイアントC



(2) データベースサーバP_i



状態	PDU	ローカル条件
I: アイドル	VR: VOTE-REQ	ua: 一方的アボート
U: 不確定	AB: ABORT	to: タイムアウト
C: コミット	CM: COMMIT	
A: アボート		

図7.7 二相コミットメントプロトコルの状態遷移図

7.2.4 障害復旧プロトコル

各データベースサーバPjは、停止障害から復旧したとき、ログにより停止したときの状態に復旧する。このとき、P_jが不確定状態であるならば、P_jだけでは、コミットまたはアボートの決定を行えない。即ち、独立復旧を行えない。Pjが復旧したとき、次の復旧プロトコルを用いる。

［復旧プロトコル］

(1) データベースサーバPjは不確定状態であれば、終結プロトコルにより決定を行う。

(2) P_jがYES 又はNOを送信する前の状態であれば、アボートする。

ネットワーク障害に対しても、同様である。

7.2.5 評 価

二相コミットメントプロトコルの性質として、以下がある。

(1) プロセス障害とネットワーク障害に対して頑強である。すなわち、プロセスの障害とネットワークの障害に対して、全プロセスで同一の決定に導ける。

(2) しかし、障害が生じたとき、ブロックする場合がある。

次に、性能について考える。ラウンドを、最遠隔のプロセスにデータ単位が届くまでの遅延時間とする。このとき、二相コミットメントプロトコルの実行時間は、以下となる［図7.8.］。

(1) 障害のない場合　　3ラウンド

(2) 障害のある場合　　5ラウンド

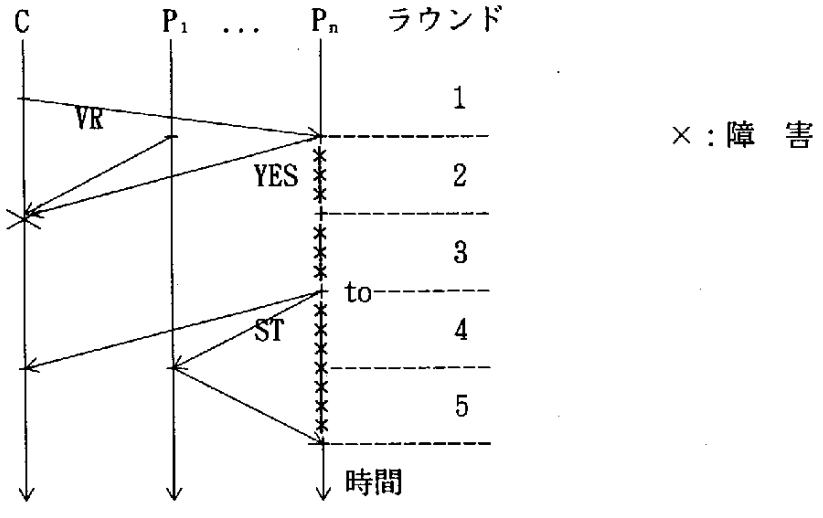


図7.8 ラウンド数

次に、送信されるデータ単位数について考える。データベースサーバの数をnとする。また、プロセス障害がおきる場合に、終結プロトコルを起動するプロセス数をmとする。このとき、データ単位数は以下となる。

(1) 障害のない場合　　3n

(2) 障害のある場合　　$nm + \Sigma_{j=1,\ldots,m}(n - m + j) = 2nm - m^2/2 + m/2$

二相コミットメントを実現するためには、各データベースサーバで、コミットに加えて、VOTE-REQを受信した時に、副トランザクションの更新状態をログに退避する機能(pre-commit) が必要となる。

7.3 三相コミットメント

二相コミットメント(2PC)プロトコルのブロック問題を解決するために、三相のコミットメントプロトコル[SKEE83]が示されている。三相のコミットメント(3PC (three-phase commitment))プロトコルの基本手順は以下のようである。

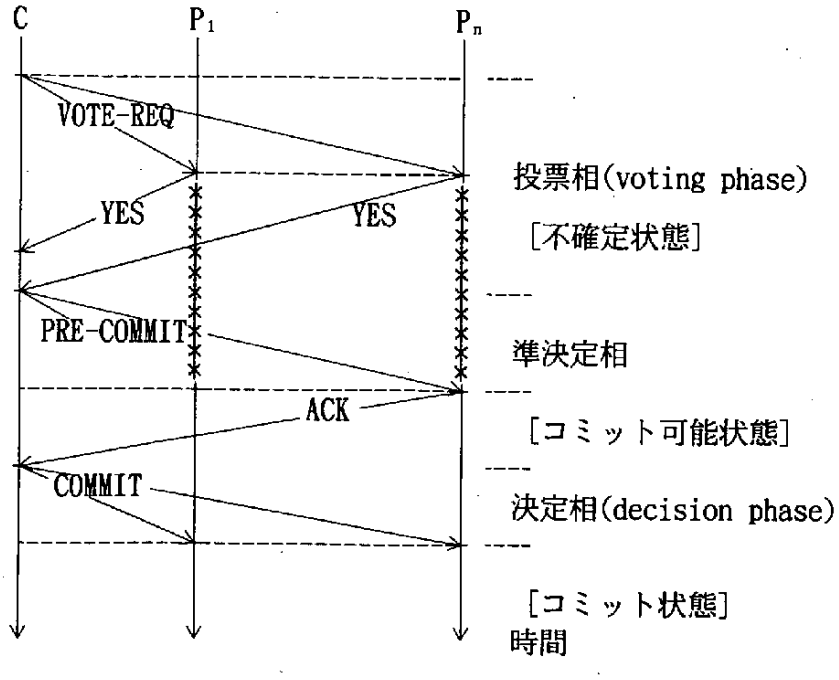


図7.9 三相コミットメントプロトコルの基本手順

［三相コミットメント(3PC) プロトコル］ ［図7.9］

(1) クランイアントC は、VOTE-REQを送信する。

(2) データベースサーバ（関係プロセス）がVOTE-REQを受信したならば、YES 又はNOをクランイアントC に送信する。NOを送信したときは、アボートする。

(3) クランイアントC が、全データベースサーバからYESを受信すれば、PRE-COMMITを全データベースサーバに送信する。一つでもNOを受信したならば、ABORT を全データベースサーバに送信して、アボートする。

(4) データベースサーバがPRE-COMMITを受信したならば、ACK(acknowledgment) をクランイアントC に送信する。ABORT を受信したならば、アボートする。

(5) クランイアントC は、全データベースサーバからACK を受信したならば、COMMITを送信して、停止する。

(6) 各データベースサーバは、COMMITを受信したならば、コミットして、停止する。

二相コミットメントプロトコルでは、不確定状態のプロセスが、指揮プロセスからのCOMMIT またはABORT を待っていてタイムアウトしたときに、コミットともアボートもできないためにブロックしてしまう。これに対して、三相コミットメントプロトコルは、不確定状態のプロセスがタイムアウトした場合には、アボートできるものである。
三相コミットメントプロセスは、ブロック問題を生じない利点があるが、処理時間が5ラウンド、送信されるデータ単位数が5nとなることから、性能面で問題がある。

7.4 意味的なコミットメント制御

以上述べてきたコミットメント制御プロトコルでは、全データベースサーバDBS_1, ..., DBS_nで更新を行えるかどうかという更新の原子性を保証するためのものであった。これに対して、旅行の例を考える。旅行では、ホテルの予約と、飛行機の予約が必要となる。この場合には、両方の予約を行えないと旅行に出掛けられない。これは、ホテルの予約データベースシステムと、飛行機の予約データベースシステムの両方を原子的に更新する必要があり、本章で述べたコミットメント制御プロトコルを利用できる。これに対して、ホテルの予約を考える。いくつかのホテルに予約を依頼し、この中のどれかのホテルの予約を

1えればよい。これは、複数のデータベースシステムサーバDBS_1，...，DBS_n の中の少なくとも一つだけ更新できればよい。このように、一般に、 n個のデータベースサーバの中の幾つ(r個) を更新できればよいかを(n,r) と書くことにする。このとき、従来のコミットメント制御プロトコルは、(n,n) となる。

ホテルの予約の例は、(n,1) となる。どのような合意をとるかは、応用の意味による。

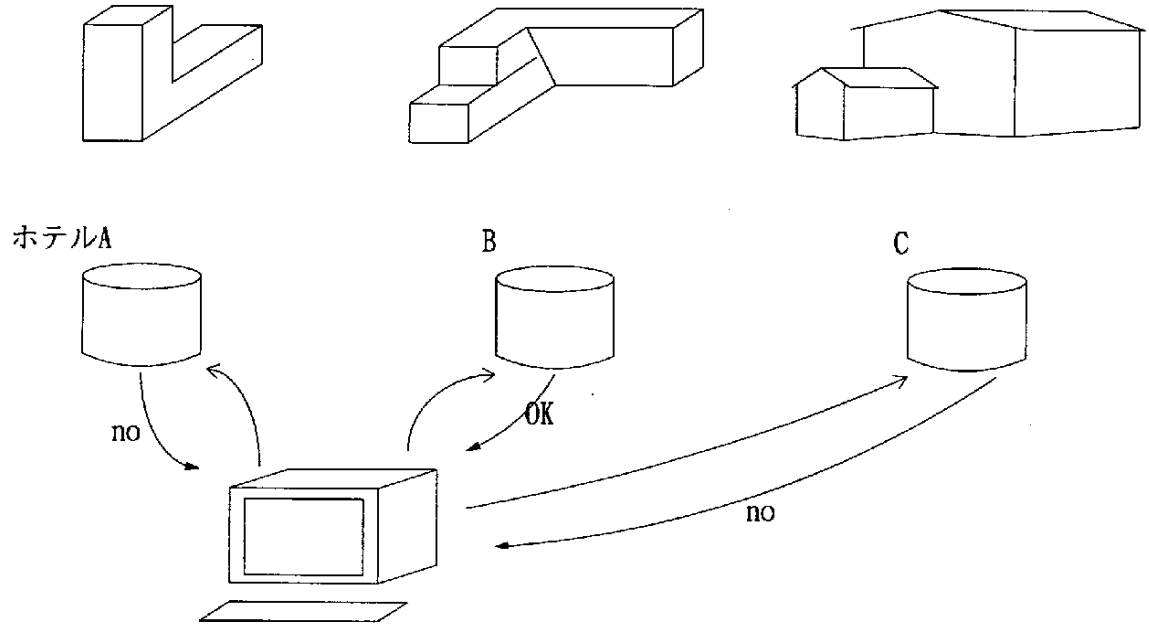


図7.10 (n,1) のコミットメント

8章　ビザンチン合意プロトコル

コミットメント制御では、プロセスの停止障害のみを考えた。ここでは、プロセスが、コミッションとオミッション障害を起こす場合について考える。

8.1　ビザンチン合意問題

障害しているプロセスを障害プロセス、障害していないプロセスを信頼プロセスとする。このとき、合意問題を以下に定義する。

［**ビザンチン合意**］ビザンチン合意(Byzantine Agreement）とは、プロセス障害について何も仮定できない場合に、全信頼プロセスが合意に達することをいう。ただし、同時に障害し得るプロセスの最大数はわかっているとする。

分散型システムは n個のプロセスから構成されているとする。プロセスは、コミットメント制御プロトコルの場合と同じく、クランイアントまたはデータベースサーバを意味する。障害するプロセスの最大数をtとする。ある１つのプロセスが送信したデータ単位の中身（値）について、全信頼プロセスが同じ値に合意させることが問題となる。ここでは、ビザンチン合意に到達するための手順は、データ単位の送信をしなかったり、データ単位の中身について嘘をつくプロセスが存在に対して、頑強でなければならない。

［**ビザンチン合意条件**］あるプロセスがある値を持つデータ単位を送信したとき、以下の制約が満たされればビザンチン合意は達成される。

C1: 全ての信頼プロセスは、同一の値に合意する。

C2: 送信プロセスが信頼プロセスならば、全信頼プロセスはこの値に合意する。

送信プロセスが信頼プロセスならば、C2よりC1は明かである。しかし、受信プロセスは送信プロセスが信頼プロセスかどうかわからない。この問題の難しい点は、発生する障害の種類と、どのプロセスが障害しているかがわからないことである。このようにプロセスの

障害について何も仮定できない状況で、制約C1とC2を満たすプロトコルを設計することが必要となる。

送信プロセスが障害している可能性があるので、プロセスがデータ単位を受信したとき、他の受信プロセスがどのような内容のデータ単位を受信しているかを調べる必要がある。このために、受信プロセス同士が、受信した値を交換し合う必要がある。しかし、プロセスの信頼性が保障されていないと、これらの値の交換においても誤動作が発生する可能性がある。この受信プロセス間で受信したデータ単位の交換で、送信プロセスと他のプロセスから異なる値を受信したら、障害しているのが送信プロセスか他のプロセスなのか区別できない。よって、データ単位を受信プロセス間で一回交換するだけでは、ビザンチン合意の決定を行えない。

8.2 合意プロトコル

まず、以下の仮定を設ける。

[仮定] (1)通信ネットワークは信頼性がある。つまり、動作中のプロセスは他のプロセスにいつでもメッセージを送信できる。また、メッセージは送信された順に、変更なしに受信される。

(1) n個のプロセスの中で、障害プロセスの最大数をtとする。

(3) 通信網のトポロジーは、完全ネットワークである。

(4) メッセージに署名は用いない。

(5) 受信プロセスは送信プロセスを識別できる。

(6) メッセージが到着しないことを発見できる。メッセージを生成して送信するのに必要な最大時間を uとし、送信、受信サイトのクロックの差（最初に同期が取られているとしたときの差）をe とする。このとき、送信プロセスの時刻T に生成され送信されたメッセージは、受信プロセスのクロックでT+u+e以内に受信される。

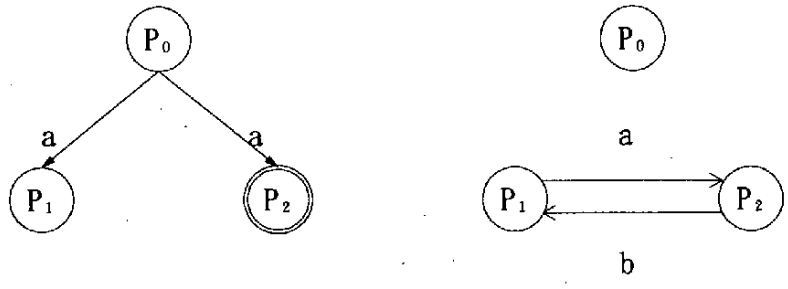
このとき、以下の結果が得られている。即ち、以下の条件が充足されないと、ビザンテン合意は得られない。

[ビザンチン合意条件] n ≧ 3t + 1。

[例] 例として、n = 3でt = 1の場合を考える。P_0、P_1、P_2をプロセスとする。プロセスP_0は、最初に値を送信するプロセスとする。このとき以下を行う。

第1相：P_0が値をP_1とP_2に送信する。

第2相：P_1とP_2は、おのおの受信した値を交換する。



◎：障害プロセス

図8.1

ここで、P_2を障害プロセスとする。P_1は、P_0からa、P_2からb を受信している。ここで、P_1はネットワーク内に障害プロセスがあることがわかるが、どのプロセスか障害しているか、いくつのプロセスが障害しているかはわからない。

第1相：　　第2相：

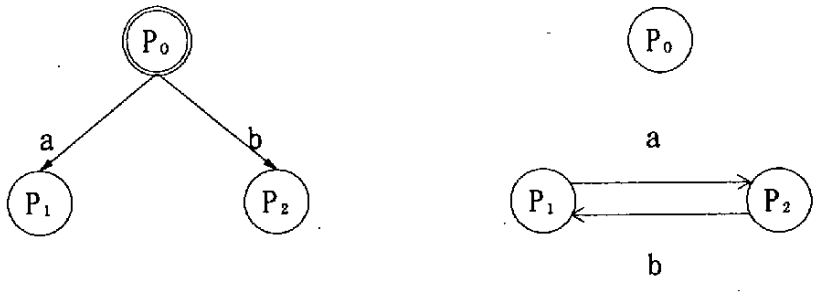


図8.2

次に、P_0が障害している場合を考える。P_1は前と同様の状況になる。よって、信頼プロセスP_1とP_2は合意に達せず、C1は満たされない。n = 3、t = 1では結論が出せない。この場合、これ以上の交換を行っても無意味である。

[相数] 最大 t個のプロセスが障害するならば、t+1相の値の交換が必要である。
第1相では、送信プロセスP_0が、他の受信プロセスに値を送信する。続く相では、受信プロセス同士が、受信した値を交換し合う。各プロセスが送信するメッセージには、自分の識別子を付ける。

(v : P0, pi1, pi2,..., pik)

プロセスP_iがこのメッセージを受信したら、以下の様に解釈する。第j相において、プロセスP_{ij}はメッセージ$(v : P_0, p_{i1}, p_{i2}..., p_{i,j-1})$を受信し、自分自身の識別子$P_{ij}$を付け、$P_{i,j+1}$を含む他のプロセスへ送信した。ここで、majorityを、値の集合を引数とし、過半数値を返す関数とする。また、過半数を超える値がなければvdefを返す。

第1相：送信プロセスP_0が、値v に自分の識別子を付けた$(v : P_0)$を、他のn-1個のプロセス$P_1, P_2, ..., P_{n-1}$に送信する。
第2相：第1相の各受信プロセスは、自分の識別子を vに付け、他のn-2個の受信プロセスに送信する。例えば、P_iは、$(v : P_0)$を受信し、$(v : P_0, P_i)$を$P_1, P_2, ..., P_{i-1}, P_{i+1}, ..., P_{n-1}$に送信する。
第3相：各プロセスは第2相で$(v : P_0, P_i)$なるn-1個のメッセージを受信しており、これに自分の識別子を付け、残りのn-3個のプロセスに送信する。以下同様に値の交換を行う。

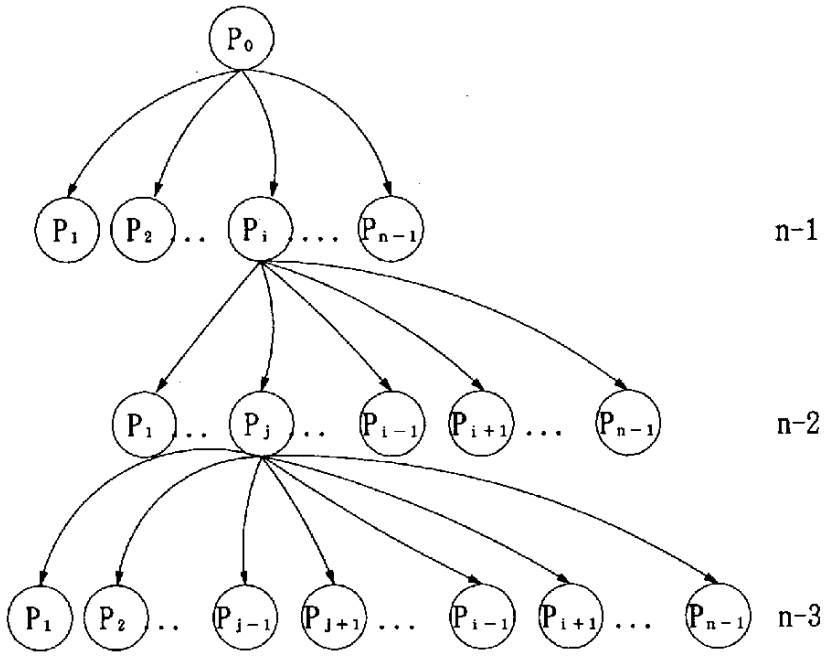


図8.3 交 換

以下に合意手順を示す。

[$UM_n(t)$: Unsigned message maximum of t faulty processes]

begin 1 : 送信プロセスP_0が、値を他のn-1個のプロセス$P_1, P_2, \ldots, P_{n-1}$に送信する;

2 : 各受信プロセスは、P_0から値を受信すればその値を、そうでなければvdefを記録する

;

3 : if t > 0 then

begin for 各受信プロセスP_iについて

3.1 : v_i := P_i が2で記録した値;

P_iはv_iを他のn-2個のプロセス

$P_1, P_2, \ldots, P_{i-1}, P_{i+1}, \ldots, P_{n-1}$ に$UM_{n-1}(t-1)$ を用いて送信する;

3.2 : v_j := P_i が3.2でP_jから受信した値($j \neq i$)、受信していなければv_{def};

P_iはmajority($v_1, \ldots, v_{n-1}$)を記録する;

end

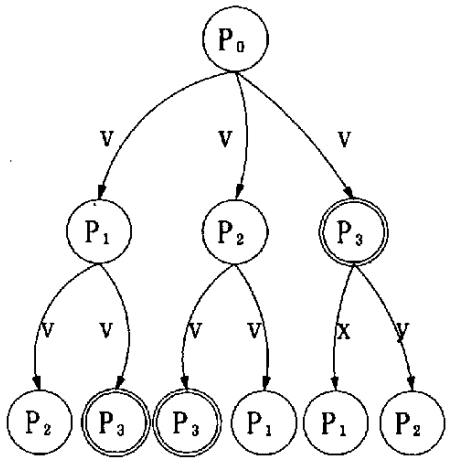
endif

end

［例］例を示す。

４つのプロセスP_0、P_1、P_2、P_3(n ＝4）に対して、t ＝ 1の場合を考える。

(1) P_3が障害している場合。図8.4に示すように、値の交換を行なう。第２相の終了時点で、信頼プロセスP_0、P_1、P_2は、majorityが vとなり、値v に合意する。よって、C1は充足される。P_0によって送信された値がmajorityの結果になるので、C2も充足される。



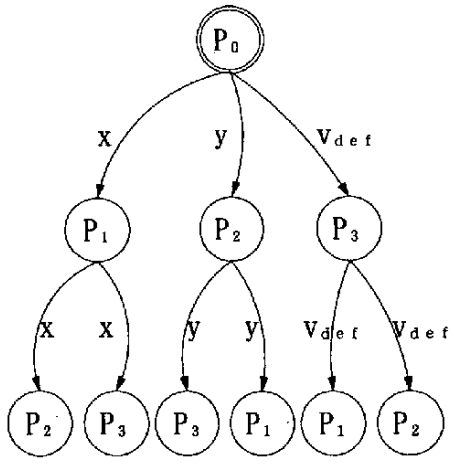
第１相：$v_1 = v$

$v_2 = v$, $v_3 = v$
第２相：P_1 : $v_1 = v$, $v_2 = v$, $v_3 = x$

P_2 : $v_1 = v$, $v_2 = v$, $v_3 = y$
P_3 : $v_1 = v$, $v_2 = v$, $v_3 = v$

図8.4

(2) 次に、P_0が障害している場合を考える。図8.5に示すように、プロセス間で値を交換する。３つの信頼プロセスは同一の値の集合を受信し、同一の値majority(x, y,vdef）を得る。よって、C1、C2は満たされる。



第１相：$v_1 = v$

$v_2 = y$, $v_3 = vdef$
第２相：P_1 ： $v_1 = x$, $v_2 = y$, $v_3 = vdef$

P_2 ： $v_1 = x$, $v_2 = y$, $v_3 = vdef$
P_3 ： $v_1 = x$, $v_2 = y$, $v_3 = vdef$

図8.5

第k 相が終了したとき、つまり深さk の木において、各データ単位は、異なるk 個の識別子の系列を含んでいる。また、第k 相ではこのようなデータ単位が (n-1)(n-2) ...(n-k)個送信される。以下の条件を全て満たすとき、そのデータ単位は正しいものとして受け入れられる。そうでなければ、廃棄される。

[受信条件]

(1) 高々k 個の識別子を含む。

(2) 識別子は全て異なる。

(3) 最初の識別子は送信プロセスPoのものである。

(4) 最後の識別子は、最後にデータ単位を中継したプロセスのものである。

(5) 自分自身の識別子は含まれていない。

アルゴリズム$UM_n(t)$ 正しさを示す。

[命題] 全てのt とp について、プロセスの合計数をn 、障害プロセスの最大数をp とすると、$n > 2p+t$ならばアルゴリズム$UM_n(t)$は制約C2を満たす。

[証明] tについての帰納法で証明する。

t = 0 のとき、全ての受信プロセスは、送信プロセスが送信した値を受信し、この値に合

意する。よって、$UM_n(t)$はC2を満たす。C2が満たされるのでC1も満たされる。

$UM_{n-1}(t-1)$について命題が成り立つとしたとき、$UM_n(t)$が成り立つことを証明する。C2について考えているので、送信プロセスが信頼プロセスであり、第１相でn－１個の受信プロセスに値v を送信する。n > 2p+tより、n-1 > 2p+t-1である。この相で必要となるn－１個のアルゴリズムUM(t-1) はC2を満たす。この相の終了した時点で、各プロセスは、各信頼プロセスPjから値v を受信しているのでv」= vである。最大p の障害プロセスが存在し、n-1 > 2p+t-1 > 2p より、これらのn－１個のプロセスの過半数は信頼プロセスである（p < n/２）。よって、各信頼プロセスは相の終了時にmajorityの値としてv を得る。よってt の場合についても成り立ち、命題は成り立つ。

［定理］同時に障害し得るプロセスの最大数をt としたとき、全プロセスの合計数n が3tよりも大きければ（n > 3t）、アルゴリズムUM(t)はC1、C2を満たす。

［証明］tについての帰納法で証明する。

t = ０のときは明かである。

$UM_{n-1}(t-1)$ について成り立つとしたとき、$UM_n(t)$ が成り立つことを証明する。送信プロセスが障害しているかしていないかで、２つの場合を考える。

(1) P_0が信頼プロセスのとき：t = p ならば、命題よりn > 3tならば$UM_n(t)$はC2を満たす。また、P_0が信頼プロセスであることから、C1も満たされる。よって、定理は成り立つ。

(2) P_0が障害しているとき：P_0を含んだt 個のプロセスが障害する可能性があり、受信プロセスの中には、高々t－１個の障害プロセスが存在する。仮定n > 3tより、P_0を別としたプロセスの合計は3t-1より大きい。これは3(t-1)より大きい。よって、$UM_{n-1}(t-1)$ についてC1、C2が成り立つ。$UM_{n-1}(t-1)$の結果、全信頼プロセスは同一の値に合意する。よって$UM_n(t)$についてのmajorityの結果、同一の値を得る。つまり、C1は満たされる。よって定理は成り立つ。

［性能］

性能について考える。合意プロトコルの性能評価の尺度には以下の２つがある。

(1) 相数：障害プロセスの最大数をt としたとき、t+１相必要である。これはビザンチン合意に至るための最小相数である。

(2) メッセージ数：(n-1)(n-2)...(n-(t+1)) = O(nt+1)。

署名を用いることにより、プロセスの障害は停止のみで、誤動作については考える必要がなくなる。つまり、メッセージの中身が変更されても、これを受信プロセスが発見できる。

[The Lamport, Shostak and Pease algorithm]

送信プロセスP_0は、送信する値v を含むメッセージに署名して送信する。P_0が障害していれば、異なる任意の値を送信するかもしれない。各受信プロセスは、受信したメッセージに自分の署名をして、まだそのメッセージに署名していないプロセスに送信する。以下これを繰り返す。(t+1) 相でメッセージの送受信は終了し、合意する値を決定するために、受信した値の集合に対して関数choiceを適用する。第k 相のメッセージはk 個の署名を含んでいる。障害プロセスがメッセージを変更しても、受信プロセスはこれを発見できる。

V_i = P_i が受信した値の集合。最初は空。

[Algorithm SMn(t) : Signed message maximum of t faulty processes]

(1) 第1相：送信プロセスP_0が、送信する値を含むメッセージに署名し、これを他のn-1個の受信プロセスに送信する;

(2) 第k 相($1 \leq k \leq t+1$)：

when Pi ($i \in \{1,\ldots,n-1\}$) がk 個の署名と値v を含むメッセージm を受信した

do Vi := Vi ∪ v;

if k < t+1 then

begin メッセージm に署名;

mに署名していない全プロセスにm を送信;

end

endif

enddo

(3) 第(t+1)相の終了： $\forall i \in \{1,\ldots,n-1\}$, P_iはchoice(Vi)の値を記録する ;

この手順では、t+1相で合意に達し、$(n-1)(n-2)\ldots(n-(t+1)) = O(nt+1)$のメッセージの交換が必要である。

9章　意味的な同時実行制御

銀行業務等の従来のトランザクションは、データベース内の小量のデータを短時間操作するものであった。これに対して、グループウェア、 CADといった新しい応用では、複雑な構造のデータが、大量にかつ長時間操作される。従来のトランザクションに対して、こうした応用のトランザクションでは、より多くのデータが長時間操作されることから、デッドロックの発生する確率が増大する。デッドロックを解決する方法として、デッドロック状態のトランザクションの一つを選択し、アボートさせることがある。トランザクションが大量データを長時間操作することから、トランザクションをアボートし、再開するために多くの計算資源が費やされる。このために、まず、トランザクション内で、デッドロックの解決のために必要な部分のみをアボートすることを考える。トランザクション全体をアボートするのに対して、その一部がアボートされることから、アボートと再開のために操作されるデータ量と時間を減少できる。

次の問題は、トランザクションをアボートするために必要となる情報量である。トランザクションが操作するデータ量が増加するにつれて、ログに記憶するデータ量も増大する。したがって、大量データを操作するトランザクションでは、ログの大きさをどのように小さくするかが問題となる。本論文では、更新されたデータではなく、実行された演算をログに記録する方式を用いる。一般に、更新されるページ等の状態情報よりも更新演算を記録する方が、ログの記憶量を減少できる。このため、ここでは実行された演算をログに記録する方式を用いる。トランザクションをアボートするためには、実行された演算に対して、その補償演算［TAKI91］を実行する。補償演算は、演算の実行前の状態に戻すトランザクションである。

補償演算がトランザクションであることから、データのロックを要求することにより、デッドロックが生じる可能性がある。本論文では、この問題について議論し、ある種の補償演算の実行によっては解決できないデッドロックが存在することを示す。これを補償不可能デッドロックとする。さらに、補償不可能デッドロックの起きないような補償演算の定義方法を示す。

9.1　システムモデル

システムM は、通信網によって結合された複数のオブジェクトから構成される。各オブジェクトo は、データ構造と操作演算により与えられる抽象データ型 [LISK77] である。オブジェクトo の属性a をo.a と示す。操作演算には、基本演算と公開演算がある。ロックによる排他制御なしに、o を直接操作し、他の演算を呼び出さない演算を基本演算とする。これに対して、公開演算は、 oの基本演算と他のオブジェクトの公開演算を呼び出すことにより実現される演算である。利用者は、 oを公開演算を通してのみ操作できる。また、公開演算は、 oのロックを要求する。ここで、 Mの各状態s に対して、op(s) は、演算opを状態s に実行して得られた状態を示す。

公開演算opは、オブジェクトo をモードmode (op) でロッする。o_1とop_2 をオブジェクトo の二つの公開演算とする [図9.1] 。 op_1が操作しているo をop_2 が操作できるとき、mode (op_2)はmode(op_1) と伴立 (compatible) [KORT90]である。例えば、銀行オブジェクトBankに対する入金deposit と出金withdrawalは、どちらの演算を先に実行しても、銀行の口座の残高は同じである。このために、両者の演算は、伴立である。

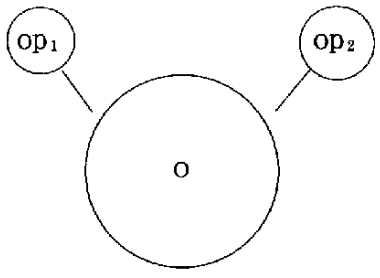


図9.1　オブジェクト

[例9.1] 車の設計を行なうシステムC を考える。 Cはcar, body, chassis, seatの 4種類のオブジェクトから構成される。オブジェクトcar は属性として、車高height、車幅width を持つ。また、 carを拡大する公開演算Enlarge が提供されている。オブジェクトbodyは属性として、車高heightと車幅width を持ち、公開演算Extend(Eと書く) 、Shorten(S), Up(U), Down(D)を提供する。これらの演算は、基本演算Write(wr) により実現される。wrはオブジェクトの属性値の更新を行なう基本演算である。bodyの公開演算

Extend(E) とShorten(S)は、基本演算wrを用いて属性width の拡大あるいは縮小を行う。オブジェクトbodyのup(U) とDown(D) は、属性heightを増加あるいは減少させる。オブジェクトchassis は、属性としてシャーシ幅width を持ち、公開演算Extend(E) と Shorten (S) を提供する。公開演算E とS は、基本演算wrによりwidth を操作する。車幅の拡大（縮小）と車高の増加（減少）は二つの属性に対する操作であり、車幅を拡大（縮小）しているときに車高を増加（減少）することができるので、公開演算E とS のモードは、D と Uに対して伴立である。しかし、同じ属性を操作する 2つのE 、 2つのS 、及び、E と S は伴立でない。

9.2 トランザクション

トランザクションは、これまでに説明したように、オブジェクトに対する公開演算の実行系列であり、原子的な実行単位である。トランザクションT 内の各公開演算opはあるオブジェクトo の演算であり、他のオブジェクトの演算op_1, ..., op_n を順に実行する。これを以下のように書く。

$$< [op, op_1, \ldots, op_n, op] >$$

ここで、[op とop] は、オブジェクトo の公開演算opの開始(begin) とコミット(commit) を示す。このとき、opは、各演算op_i を呼びだすとし、opをop_i の親とする(i=1,...,n) 。さらに、各演算op_i が公開演算ならば、他の演算op_{i1}, ..., op_{ini}を呼び出すことになる。さらに、演算op_{ij}が他の演算を呼び出すといったように、節点を演算とし、各節点の子の順序が実行順序を示す順序木として、トランザクションT を示せる。これをトランザクション木とし、 Tを階層型トランザクション（または、入れ子型トランザクション(nested transaction)) [BEER89, GARZ88, LYNC86, MOSS86, TAKI91, WEIH88, WEIH89, WEIK86, YASU91, YASU92] とする。

[定義] トランザクションT に対するトランザクション木を、以下のように定義される順序木とする。

(1) 根は、トランザクション全体を示す。これをT とする。

(2) 葉節点は、基本演算を示す。

(3) 根と葉以外の各節点opは、公開演算を示す。opが、他の演算op_1, ..., op_n を、この順序で呼び出し実行するとき、opを親、op_1, ..., op_n を子として、この順に並べる。

ここで、トランザクションT 内の二つの演算op_1 とop_2 に対して、lca(op_1, op_2) を、 Tのトランザクション木内のop_1 と op_2の最小共通祖先とする。

[例9.2] 例9.1で、演算En(=Enlarge)は、オブジェクトcar を拡大するトランザクションである [図9.2] 。ここで、 bはオブジェクトbody, cはオブジェクトchassis, wは属性width を示す。Enは、トランザクションの根を示す。葉のwr(o.w) はオブジェクトo の属性width を操作する基本演算を示す。以下は、トランザクションEnの演算系列を示す。

< [En, [E(c), wr(c.w), E(c)], [U(b), wr(b.h), U(b)], [E(b), wr(b.w), E(b)], En] >

次に、オブジェクトcar の車幅width を広げるトランザクションW$は、オブジェクトbodyとchassis の演算Extendを起動する [図9.3]。以下は、トランザクションWdの演算系列を示す。

< [Wd, [E(b), wr(b.w), E(b)], [E(c), wr(c.w), E(c)], Wd] >

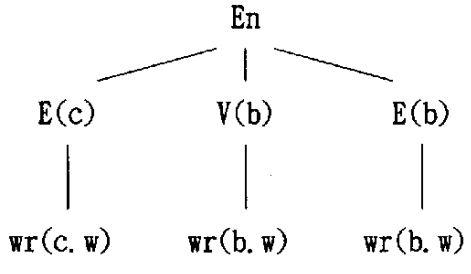


図9.2 トランザクション木En

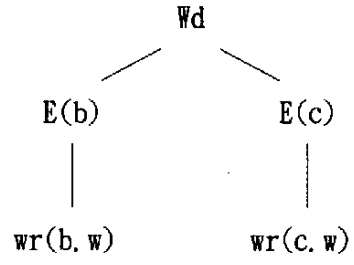


図9.3　トランザクション木Wd

9.3　同期方法

トランザクションT 内の各演算opは、実行前にopのオブジェクトo をロックする。 Tによって得られたすべてのロックは、 Tがコミットしたときに、解放されるので、 Tは2相ロック形式である。記法op_T は、トランザクションT の演算を示す。 [opは、オブジェクトo のロック演算である。ここで、通常の関数呼び出しの機構と同じように、opのローカル変数は、スタックST_T に割り当てられる。op] は、opの終了を示し、ST_T からローカル変数を解放する。しかし、ここではロックは解放されない。トランザクションT が終了したときに、獲得されている全ロックが解放される。即ち、T]は、トランザクションT が獲得した全ロックを解放する演算である。

[**定義**] オブジェクトo に対するロックのモードm_1とm_2に対して、m_2がm_1より排他的である（$m_1 \subseteq m_2$）とは、任意のモードm_3に対して、以下が成り立つことである[KORT90]。

(1)　モードm_1がm_3と伴立ならば、m_2とm_3と伴立である。

(2)　m_3がm_2と伴立ならば、m_3はm_1と伴立である。

オブジェクトo のモード集合M は、⊆について半順序集合となり、∪を、⊆上の最小上界（lub (least upper bound)）とする。トランザクションT の複数の演算が、オブジェクトo を操作するとき、 oのロックモードを転換(convert) する必要がある。例えば、トランザクションが、あるオブジェクトx をreadした後に、 writeする場合を考える。readから

write にモードを転換するときに、モードがより排他的となるためにデッドロックが起きる可能性がある。readするときにwrite モードでロックすることも一つの方法であるが、同時実行性が減少してしまう。このために、文献 [KORT90] には、readよりも排他的であるが、 writeよりも排他的でないモードU^{write}_{read}定め、非対称なモード転換が論じられている。

9.4 デッドロック

まず、演算間の依存関係を定義する。

[定義] 演算op_1 がop_2 に依存する ($op_1 \Rightarrow op_2$) ことを、以下のいづれかが成り立つと定義する。

(1) op_2 が獲得しているオブジェクトo を、他の演算op_1 が、op_2 と非伴立なモードで待っている。

(2) あるトランザクションで、 op_1は op_2に先行する。

(3) ある演算op_3 に対して、$op_1 \Rightarrow op_3 \Rightarrow op_2$である。

節点を演算、 op_1から op_2の有向辺が、$op_1 \Rightarrow op_2$示す有向グラフを、拡張待ち(EWF (extended wait-for graph))グラフとする。演算opが、拡張待ちグラフ内の有向サイクルに含まれるとき、opはデッドロックしている。本論文では、ある機構 [KNAP87] により、各システム状態に対する拡張待ちグラフが構成されるとして、デッドロックの解除方法を検討する。

[例9.3] 図9.2と図9.3に示す2つのトランザクションEnとWdを考える。[E_{wd}(b) が既に獲得しているオブジェクトb(=body)を、[E_{En}(b) がロックを要求したとする。このとき、[E_{En}(b) は[E_{wd}(b) を待ち、[E_{En}(b) →[E_{wd}(b) である。同様に、[E_{wd}(c) → [EE_{En}(c) である。このときの拡張待ちグラフを図9.4に示す。ここで、以下が成り立つ。

$$[E_{En}(b) \rightarrow [E_{wd}(b) \rightarrow [E_{wd}(c) \rightarrow [E_{En}(c) \rightarrow [E_{En}(b)$$

従って、巡回有向閉路があり、トランザクションEnとWdはデッドロックしている。

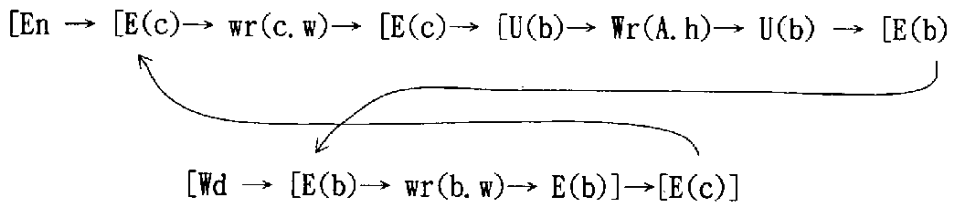


図9.4　拡張待ちグラフとデッドロック

ここで、各トランザクションT 内で、Current(T)は現在実行中の演算を示し、FirstD(T)はデッドロックしている最先頭の演算を示すとする。図9.4で、Current(En) は[E(b) , FirstD(En)は[E(c) である。

9.5　トランザクションの補償

デッドロックが検出されたとき、デッドロックしているあるトランザクションT をアボートすることにより、デッドロック解除できる。ここでは、トランザクションT をどのようにアボートするかについて考える。

9.5.1　補償演算

opを演算、 sをシステムの状態とする。$op^{-}(op(s)) = s$となる演算 op^{-}をopの補償演算とする。補償演算の性質は、[KORT90, TAKI91]で論じられている。本論文では、各演算opに対して、少なくとも１つの補償演算op^{-}が応用の意味に基づいて定義されているとする。例えば、例10.1で、E(= Extend) の補償演算E はS(=Shorten}) である。また、以下が成り立つ。

$$< op_1, \ldots, op_m >^{-} = < op_m^{-}, \ldots, op_1^{-} >$$

補償演算が、公開演算ならば、オブジェクトのロックを要求する。あるトランザクションで、opの後に$\overline{op}$が実行されたときは、opと$\overline{op}$によりロックされたオブジェクトは解放される。 このとき、opは$\overline{op}$ によりアボートされたことになる。補償演算によるアボートを補償とする。基本演算opの$\overline{op}$は、opと同じオブジェクトの基本演算とする。

9.5.2 補償演算によるアボート

トランザクションT の各演算opが呼び出されると、これのローカル変数の領域が、スタックST_t に割り当てられる。演算opが演算$op_1, \ldots, op_m$ を呼び出すとする。op_j の実行により、状態は $< S_{j-1}, L_{j-1} >$ から $< S_j, L_j >$ に変更される $(j = 1, \ldots, m)$。ここで、 S_j と L_j は、各々オブジェクトの状態とopのローカル変数の状態である。op_j のコミット後の状態は、 $< S_j, L_j >$ である。ここで、障害が起きたとする。op_j により、オブジェクト状態は S_{j-1} に復旧されるが、ローカル状態は L_j のままである。ここで、op_j が $< S_{j-1}, L_j >$ で再実行されると、 $< S_{j-1}, L_{j-1} >$ でop_j を実行した結果と異なる場合がある。しかし、$op_1, \ldots, opj$ が $\$op_j^{-}, \ldots, op_1^{-}$ によりアボートされ、 T_t からopのローカル変数の解放後に、すなわち、$([op)^{-}$ の実行後に、新ためて、opを呼び出すとする。opのローカル変数は、新しく割り当てられるので、ここで述べた問題は起きない。このように、トランザクションを部分的にアボートし、再実行するためには、オブジェクト状態とともにローカル状態の復旧が必要となる。

[例9.4] オブジェクトcar を拡大するトランザクションEnlarge は、図9.5のように書ける。ここで、 constはある定数とする。 Enlargeの演算(4)が終了した後に、(2)までがアボートされるとする。補償演算 $< E^{-}(c), U^{-}(b) >$ により、オブジェクトchassis の属性width とオブジェクトbodyの属性heightに対する更新結果が除去される。しかし、変数 u の値は復旧されず、(3)で得られた値である。ここで、(2)から再実行されると、(3)が再度更新されるために、(4)では、以前の更新と異った値によりオブジェクトchassis のwidth が更新される。したがって、再実行するために、 uの値も復旧されねばならない。全演算を補償し、スタックからu の領域を解放した後、再びEnlarge を呼び出すなら、スタック内にu が新しく割り当てられ、再実行できる。

Enlarge

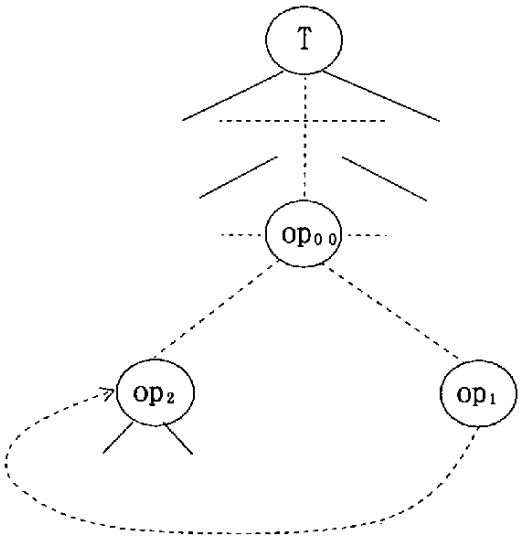
(1) 変数t とu に、オブジェクトcar の属性width とheightをreadする。

(2) オブジェクトbodyのheightを、 t内の値で変更(Up(b)) 。

(3) u = u + const 。

(4) オブジェクトchassis の属性width を uに変更(Extend(c)) 。

図9.5 トランザクションEnlarge



Current(T) = op_1, First D(T) = op_2
op_0 = lca (op_1, op_2)

図9.6 トランザクション木

デッドロックしたトランザクションT の全体をアボートするかわりに、デッドロック閉路内の演算をアボートすることにより、デッドロックを解除することを試みる。このようにトランザクションの一部の演算をアボートすることを部分アボートとする。図9.6で、演算op_1 をCurrent(T)とし、op_2 をFirstD(t), op_0をop_1 とop_2 の最小共通祖先lca (op_1, op_2)とする。また, op_{00}をop_0 の親とする。op_2 がop_1 まで補償しても、例9.4に示したように再実行できない。このために、op_0 をアボートする。即ち、op_{00}が呼び出している演算を補償し、op_{00}のローカル変数を解放する。この後に、 op_0を再実行する。

[例9.5] 例9.3で、Enの[E(b) から[E(c) までの演算は、デッドロックしている。この場合、lca(E(b), E(c)) は根Enである。[En から[E(b) までの全演算<[En, [E(c), wr(w), E(c)], [U(b), wr(h), U(b)], [E(b) > を、補償演算系列 < ([E(b))$^{-}$, (U(b)])$^{-}$, wr(w), ([U(b))$^{-}$, (E(c)])$^{-}$, wr^{-}(h), ([E(b))$^{-}$, ([En)$^{-}$ >により補償することが1つの方法である。ここで、演算opに対して、([op)$^{-}$は、opによって得られたロックの解放を行なう。(op])$^{-}$は、空演算である。一方、各演算は応用の意味にしたがって定義された補償演算を持ち、E(c), U(b)はコミットしている。したがって，以下によっても、トランザクションEnを補償できる。

< [En, E(c), U(b), [E(b) >$^{-}$ = < ([E(b))$^{-}$, U^{-}(b), E^{-}(c), ([En)$^{-}$>

これは、[E(b) のロックを解放した後、U^{-}(b) , E^{-}(c) , ([En)$^{-}$が実行されることを示す。ここでは、トランザクション木内のコミットした上位の演算の補償演算が実行される。

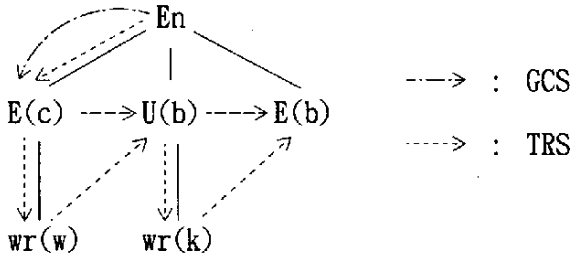


図9.7 最大コミット系列

トランザクションT で、lca(FirstD(T), Current(T))からCurrent(T)までの演算系列をトランザクション系列TRS(T)とする。TRS(T)は、ロック演算と基本演算の系列である。 TRS(T) が、各演算opについて、部分系列 < [op, ..., op] > を含めば、即ち、opがコミットしていれば、これをopで置き換えていき、これ以上置き換えできなくなったときに得られる系列を最大コミット系列 GCS(T) とする。GCS(T)は、コミットした演算を含み、コミットした演算が呼び出す演算を含まない。図9.7で、最大コミット系列は、以下となる。

GCS(En) = < [En, E(c), U(b), [E(b) >

TRS(En) = < [En, [E(c), wr(w), E(c)], [U(b), wr(h), E(b)], [E(b) >

トランザクション系列TRS(T)は、最下位のレベルで実行された演算の系列である。これに対し、最大コミット系列GCS(T)は、コミットした演算の系列であり、上位のオブジェクトの演算から構成されている。したがって、 GCS(T)$^{-}$により、より意味的なレベルにより、トランザクションT の補償を行なえる。さらに、GCS(T)は、TRS(T)よりも少ない演算を含むので、ログのサイズを小さくできる。

9.6 補償不可能デッドロック

補償演算は、トランザクションであり、オブジェクトのロックを要求するので、次の例に示すデッドロックが生じる場合がある。

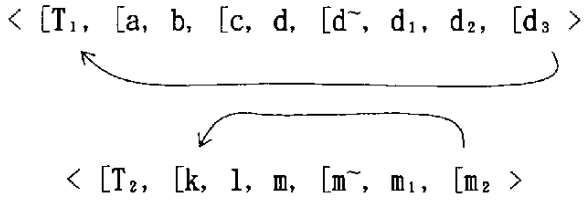


図9.8 補償不可能デッドロック

[例9.6] トランザクションT_1 = < [T_1, [a, b, [c, d, [e > において、[cから[eにデッドロック閉路が存在し、T_1をアボートするとする。T_1で、< [c, d, [e > は、< ([e)$^{-}$, d^{-}, ([c)$^{-}$> により補償できる。ここで、$\bar{d}$ は、< d_1, d_2, d_3, ..., > を呼び出すとする。この実行により、図9.8に示すように、T_1とT_2がデッドロックするとする。最大コミット系列GCS(T_1) は、< [T_1, [a, b, [c, d, ([d)$^{-}$, d_1, d_2, [d_3 > である。[m_2 は、[aが獲得しているオブジェクトx のロックを要求し、[d_3 は、[kが獲得しているオブジェクトy のロックを要求している。このデッドロックを解除するために、T_1を補償し、< ([d)$^{-}$, d_1, d_2, [d_3 >$^{-}$が実行される。その後、d をアボートするために$\bar{d}$ が再度実行される。この結果、図9.8と同じデッドロック状態になる。T_2を補償しても同様である。これを補償不可能デッドロックとする。

補償不可能デッドロックは、トランザクションT の最大コミット系列の補償GCS(T)を実行することによって解除できないデッドロックである。補償不可能デッドロックを定義する。まず、以下を定義する。

I(T) = T内の補償演算で最後に実行されたもの。

L(T) = TにおいてI(T)に先行する演算の集合。

[定義] トランザクションT がトランザクションU に非安全に依存する(T ----> U)とは、以下のいずれかが成り立つことである。

(1) Current(T)が依存する演算がL(U)内の演算に存在する。

(2) T ----> V ----> U なるトランザクションV が存在する。

次に、補償不可能デッドロックを以下に定義する。

[定義] トランザクションT が補償不可能にデッドロックしているとは、T が、T に非安全に依存する(T ----> T)ことである。

[例9.7] 例9.1で、$I(T_1)$ = [d_3 , $L(T_1)$ = { [a, b, [c, d } である。また、$I(T_2)$ = [m_2 , $L(T_2)$ = { [k, l, m } である。[d_3 は[kを待っているので, T_1 ----> T_2である。また、[m_2 は[aを待っているので、T_2 ----> T_1である。したがって、T_1とT_2は、補償不可能にデッドロックしている。例9.1で示したように、このデッドロックは、最大コミット系列の補償によって解除できない。

補償不可能デッドロックと最大コミット系列の間には、以下の関係がある[YASU93]。

[定理] 補償不可能デッドロックしているトランザクションT を最大コミット系列によって補償できない。

したがって、最大コミット系列によりトランザクションを補償すると解除できないデッドロックが存在する。

9.7 安全システム

トランザクションT が補償不可能にデッドロックしているとき、Current(T)は、ある補償演算により呼び出されていて、他のトランザクションの演算を待っている。したがって、補償演算内の演算が他の演算を待たずに実行できれば、補償不可能デッドロックは生じない。この問題について考える。

まず、演算opとその補償演算op^{-}について考える。opがオブジェクトを獲得しているもとで、op^{-}が実行される。このとき、mode(op) $\subseteq$ mode(op^{-})でないならば、ロックモードの転換が必要になる。[KORT83]において、モードx から他のモードy への転換U^{y}_{x} が論じられている。これをもとに、補償演算に対する転換C^{y}_{x} を以下に定める。

[転換モード C^{y}_{x}]

(1) モードz がx に対して伴立で、y がz に対して伴立なら、z はC^{y}_{x} に伴立である。

(2) モードy がz に対して伴立なら、C^{y}_{x} はz に対して伴立である。

(3) x がv に対して伴立であり、y とw が互いに伴立なら、C^{y}_{x} とC^{w}_{x} は伴立である。

ここで、(2)と(3)はU^{y}_{x}と同じである。T の演算opと補償演算op^{-}のモードをおのおのx とy とする。opがC^{y}_{x}でオブジェクトo をロックした後、他のトランザクションS が、以上の規則(1)に従い、モードz でo をロックしたとする。op^{-}がoのロックを要求したとき、y は、x とz に対して伴立なので、op^{-}はo をロックできる。このことより、opとop^{-}のロックモードの転換を以下のように定める。

[転換規則]

(1) y $\subseteq$ xならば、opはモードx でオブジェクトo をロックし、op^{-}はモードx でoを用いることができる。

(2) x $\subseteq$ yならば、opはC^{y}_{x}でo を獲得し、op^{-}はy にモードを転換する。

(3) x $\subseteq$ yとy $\subseteq$ xのいづれでもない場合、opは$C^{x}U^{y}_{x}$でo を獲得し、op^{-}はx $\cup$ yにモードを転換する。

各演算opが転換規則に従えば、op^{-}は、待たずにオブジェクトoを操作できる。次に、op^{-}が呼び出す演算op_1について考える。トランザクションT 内の演算によりモードm_2でロックされているオブジェクトo_1を、op_1がm_1でロックするとする。$m_1 \subseteq m_2$ ならば、op_は待たずにo_1を利用できる。$m_2 \subseteq m_1$ならば、op_1 はm_2よりもさらに排他的なロックを要求せねばならないので、op_1 はo_1を待つ場合がある。これが、補償演算により、デッドロックが起きる原因である。

[安全条件] op^{-}が呼び出す任意の演算op_1 に対して、mode(op_)$\subseteq$ mode(op_2)であるop_2が、opにより呼び出される。

[定義] (1) 補償演算op^{-}が転換規則を満たし、(2)op^{-}が呼び出す各演算op_iが安全条件を充足とき、op^{-}を安全とする。全補償演算が安全であるとき、システムを安全とする。

安全システムにおいて、op^{-}は待つことなくロックを行なえる。

[定理] システムが安全ならば、補償不可能デッドロックは起こらない。

安全システムでは、補償不可能デッドロックを起こさずに、最大コミット系列により、トランザクションをアボートできる。したがって、ログ内に、トランザクションT の全演算を記録せずに、最大コミット系列GCS(T)だけを記録するばよくなり、ログの大きさを縮小できる。即ち、各演算opがコミットしたとき、opが呼び出した演算をログから除き、opだけをログに記憶できる。
安全システムを構築するための転換規則と安全条件は、各補償演算op^{-}を定義するときの指針となる。例えば、op^{-}の定義方法がいくつかあるときに、opが用いるオブジェクトのみを用いて定義することが一つの方法となる。このように定義できない場合には、システムは安全なものとは限らない。

9.8　非安全システム

次に安全でないシステムにおいて、補償不可能デッドロックの解除法を考える。
トランザクションT の最大コミット系列GCS(T)の補償は、新たなロックを要求す場合がある。一方、トランザクション系列TRS(T)はロックと基本演算からなり、ロックは解放されない。したがって、TRS(T)は、基本演算の補償を行ない、新たなロックを要求しないので、デッドロックは生じない。

[定理] トランザクションT のトランザクション系列 TRS(T) の補償は、デッドロックを起こさない。

[例9.8] 図9.9で、a^{-}が呼び出す[fは[sを待ち、v^{-}が呼び出す[xは[bを待つ。したがって、T_1とT_2は、補償不可能にデッドロックしている。T_1をアボートする。lca(b, f) = kより、トランザクション系列の以下の補償を実行することにより、[kから[fまでがアボートされる。

$$< [k,\ [a,\ [b,\ c,\ b],\ a],\ [a^{-},\ [d^{-},\ e,\ [f >^{-} = < [f^{-},\ e^{-},\ ([d^{-})^{-},\ ([a^{-})^{-},$$
$$a]^{-},\ b]^{-},\ c^{-},\ [b^{-},\ [a^{-},\ [k^{-} >$$

この系列には、ロック演算が存在しないので、デッドロックは生じない。

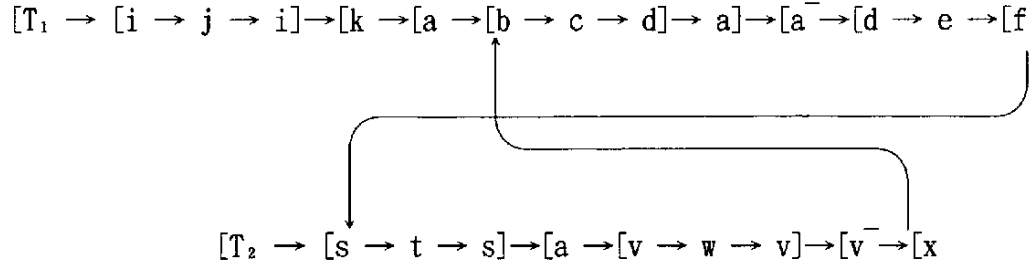


図9.9　補償不可能デッドロック

以上から、どのようなデッドロックも、トランザクションTのTRS(T)を補償することによ

り解決できることがわかる。トランザクション系列TRS(T)$^{-}$は、Tが実行した演算を基本演算のレベルで実行と逆の順序で補償していくものである。これに対して、最大コミット系列は、コミットした演算opに対して、これのop^{-}が実行される。op^{-}は、opにより実行された演算をそのまま補償するとは限らず、意味的に決まるものである。さらに、最大コミット系列は、コミットした演算を含み、これが呼び出した演算を含まないのでトランザクション系列よりも演算が少ない。したがって、ログの記憶量を減少できる利点がある。

9.9 まとめ

本章では、階層型トランザクションのデッドロック解除法について議論した。ここで述べた方法では、従来のデータベースシステムのように、トランザクション全体のアボートではなく、その一部をアボートする。また、演算をアボートするために補償演算を用いる。システムの状態に関係なく演算をログに記憶するために、ログの大きさを減少できる。演算と同様に、補償演算はオブジェクトのロックを要求する。これは、補償演算が実行されたとき、デッドロックが発生する場合があることを意味する。安全システムにおいては、補償演算の実行による補償不可能デッドロックが起こらないことを示した。また、非安全システムにおいて、基本演算の補償演算を実行することによって、補償不可能デッドロックを起こさずにトランザクションを補償できることを示した。

10章　意味的同時実行制御の応用

これまで、データベースシステムは、事務処理等の分野を中心として、伝票処理などの定型的な応用のために用いられてきたが、設計、グループウェア等の新しい分野でも利用されてきている。その中でも、 CADシステムは、表現する対象が複雑であり、データ構造やデータ操作が複雑なものとなる。ここでは、オブジェクト指向データベースシステムのUniSQLを用いた、ソリッドモデルの CADシステムを考える。

CAD システムでは、共有データの更新を行なうことから、トランザクションの概念が重要である。事務処理等の従来のトランザクションは，小量のデータの単純な操作を行なうものであった。トランザクションはオブジェクトのメソッドから構成され、各メソッドは他のオブジェクトを操作するメソッドを実行できる。このために、トランザクションの中にトランザクションが含まれる構造、つまり階層型トランザクションとなる。また、デッドロック等のトランザクション障害、あるいはCPU 障害等のシステム障害が生じた場合に、実行中のトランザクションをアボートする必要がある。 CADのトランザクションは大量のオブジェクトを操作し、実行時間の長いものとなる。このため、更新した状態をログに退避しておく従来の方法では、ログの記憶量が増大することと、復旧時間が長くなる問題があった。このようなトランザクションに対するアボートの方法として、トランザクションのメソッドの中で、障害から復旧するのに必要な部分のみを補償メソッドを用いてアボートする方法（部分的復旧方法）がある。ここでは、 CADシステムで広く用いられているソリッド（幾何形状）モデルを例として考え、補償メソッドを用いたCAD トランザクションのアボート方法を示す。ここでは、データベースを過去の同一の形状に戻すのではなく、意味的に同値な状態に戻す方法を示す。

10.1　システムモデル

10.1.1　ソリッドモデル

ソリッドモデルとは、3次元の形状や線や面で表現するのではなく、中身の詰まった固体として物体を表現する方法である。ソリッドモデルの表現方法として、CSG(Constructive

Solid Geometry) 表現と境界表現(Boundary Representation) の二つがある。CSG 表現では、基本立体形状をプリミティブとし、プリミティブを組み合わせて目的とする幾何形状が表現される。 CSG表現は、人間の思考に合うため、幾何形状を理解し易い。一方、境界表現では、図10.1に示すように、立方体は6つの面f_1～f_6から成り、面f_1は4つの稜線e_1～e_4から成り、稜線e_1は2つの端点v_1とv_2から成るものとして表現される。現在、多くのCADシステムでは、利用者インタフェースとして、 CSG表現を用い、システムの内部処理には環境表現が用いられている。したがって、ここでは利用者が操作する外部モデルとして CSG表現を適用し、これを実装するための内部モデルとして環境表現を用いることにする。

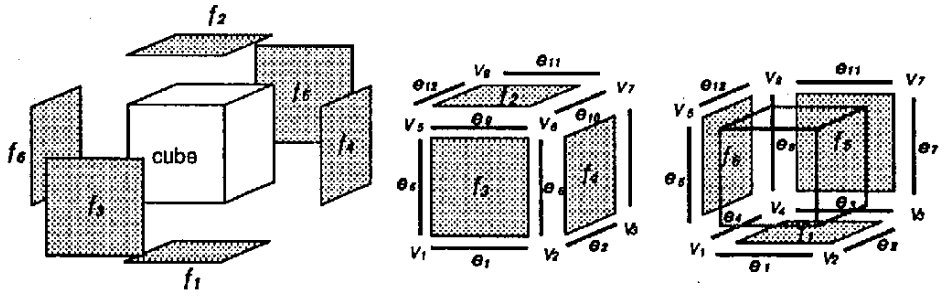


図10.1　境界表現の例

10.1.2　空間モデル

以下に、システム例を示す。

[例1] 3次元(xyz) 空間を考える。3次元(xzy) 空間に物体を置くことにより、目的とする物体を構成するソリッドモデリングシステムを考える。ここでは、プリミティブとして1辺の長さを基本体位とする立方体を考える。プリミティブは面の組、面は稜線の組、稜線は端点の組から構成される。また、空間を基本単位で区切り、各区切りを基本フレームとする。基本フレームは、立方体と同一の大きさであり、1つの立方体を置くことができる [図10.2] 。空間内に置かれる物体をオブシェクトとする。オブジェクトにはプリミティブと、プリミティブを結合した複合オブジェクトがある。

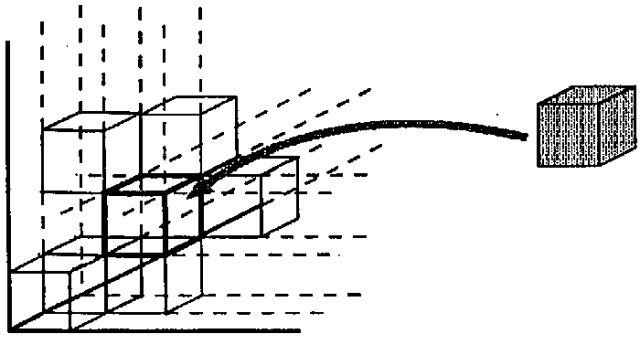


図10.2　基本フレームとプリミティブ

オブジェクトのクラス階層を図10.3に示す。space （有限空間）はunit_space（基本フーム）の組から構成される。composed_obj（複合物体）はobject（物体）の集合から構成される。cubeはfaceから、faceはedgeから、edgeはvertexから構成される。

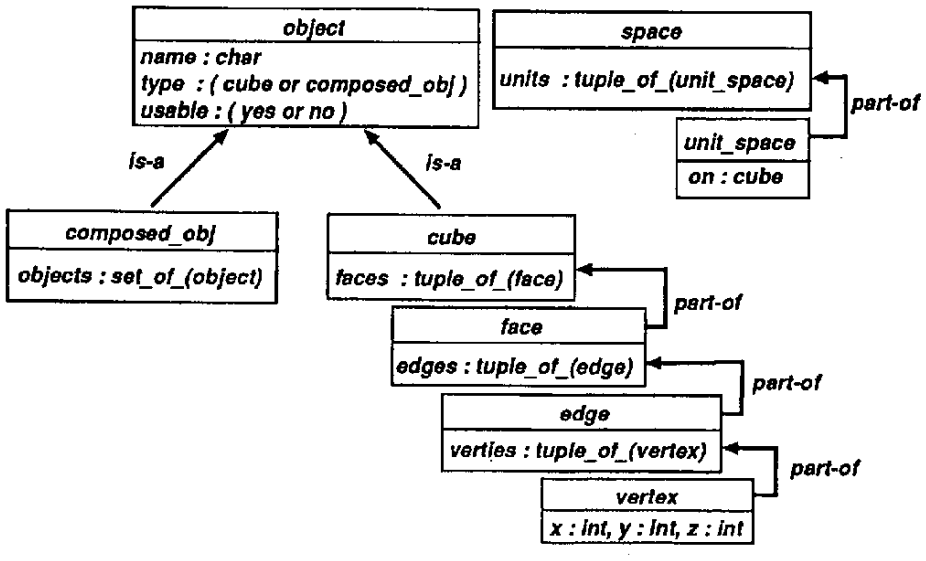


図10.3　クラス階層

cube, face, edge, vertexのオブジェクト識別子(oid) を考える。cubeの oidを iとすると、faceの oidは {i. 1, i. 2, ..., i. 6}, edgeの oidは {i. 1, i. 2, ..., i. 12}, vertexの oidは {i. 1, i. 2, ..., i. 8} である。

[例2] オブジェクトo を操作するためのメッセージ式をo. method(par) と書く。ここで、methodは oのメソッドであり、 parは引数である。 AとBをオブジェクトとする。A. put( ) は、 Aを空間に置く操作を示す。A. remove( ) は、 Aを空間から取り除く。A. concatenate(B)は、 Aを Bと複合して1つのオブジェクトとする。A. separte( )は、 Aを切り離して複合のオブジェクトとする。 A=cube. create( )と A. drop( )は、クラスA に対するメソッドであり、おのおの Aをcubeとして生成し、 Aを削除する。createは、各クラスcomposed_obj, cube, face, edge, vertexのインスタンスを生成する。図10. 4に複合オブジェクトの表現を示し、 concatenateとseparateの関係を示す。 Eは、 Cと Dを複合したオブジェクトで、 Cは Aと Bを複合したオブジェクトである。

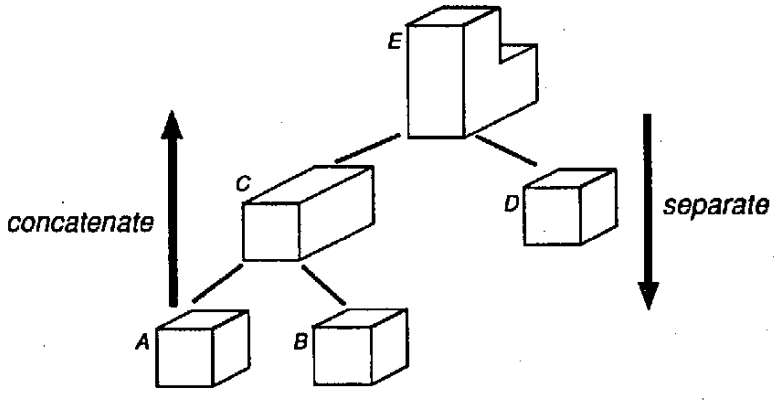


図10.4 複合オブジェクトの表

10.1.3 ロックモード

利用者は、メソッドを用いてオブジェクトを操作する。そのメソッドは、さらに他のオブジェクトを操作するメソッドを実行できる。オブジェクトは複数の利用者によって利用されるので、オブジェクトのインテグリティ制約を保つために、トランザクションの実行を直列化 [1] する必要がある。メソッドを実行する前にオブジェクトをロックする。 oをオブジェクト、 op_1と op_2を oのメソッドとする。このとき、 op_1と op_2をどのような順

序で実行してもシステムの状態が同一であるとき、op_1とop_2は可換である。 op_1と op_2が可換であるとき、 op_1のロックモードmode(op_1) は、mode(op_2) に対して伴立である[13]。 op_1と op_2の可換関係は、オブジェクトo の意味により与えられる。ここでのシステムにおけるロックモードとそのモード間の伴立関係の例を以下に示す。ロック獲得したメソッドが終了した時に、ロックは解放される。

[例3] オブジェクトをロックするモードには、排他ロックXL(eXclusive Lock)と共有ロックSL(Shared Lock) がある。XLでロックされたオブジェクトを操作できるのは、ロックを獲得したトランザクションのみである。SLでロックされたオブジェクトを操作できるのは、ロックを獲得したトランザクションと、SLでロックを要求したトランザクションである。各メソッドのモードを、表10.1に示す。また、伴立関係を示す例を図10.5に示す。図10.5は、あるトランザクションによって作られたオブジェクトA とB に対して、(1) C.put( ),(2) A.remove( ), (3) D=A.concatenate(B)が実行されたときの、ロックモードとその伴位関係を示す。object, space, unit_space は、(1)と(2)と(3)によって、図10.5のようなロックモードでロックされる。(1)と(2)を実行するとき、 spaceオブジェクト xyzに対して、SLでロックを要求する。システムの状態は、どのような順序で実行しても同一である。また、(2)と(3)を実行するとき、 Aに対して、XLでロックを要求する。(3)によって Aと Bは複合されて Dとなる。よって、(2)と(3)の実行順序によって、システムの状態は異なる。従って、removeと concatenateのロックモードは伴立でない。

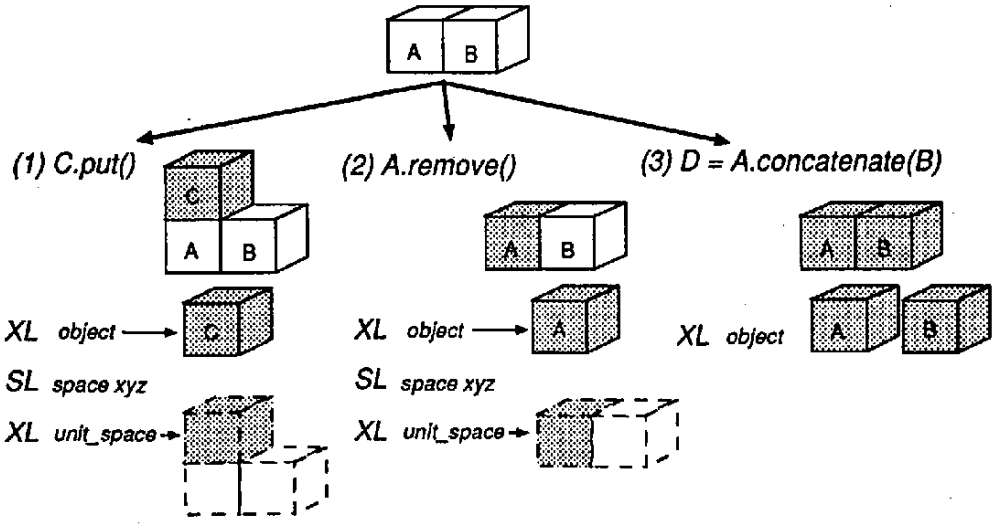


図10.5 伴立関係の例

表10.1 ロックモード

Table 1: Lock mode

メソッド	ロック	モード
put()	置くオブジェクト *object*をロックする.	*XL*
	*object*を構成するオブジェクト (*composed_obj*, *cube*, *face*, *edge*, *vertex*) をロックする.	*XL*
	空間 *space*をロックする.	*SL*
	*space*のメソッド *put*() が実行したメソッド *state*() により求められた *unit_space*をロックする.	*XL*
remove()	取り除かれるオブジェクト *object*をロックする.	*XL*
	*object*を構成するオブジェクト (*composed_obj*, *cube*, *face*, *edge*, *vertex*) をロックする.	*XL*
	put() と同様に *space*をロックする.	*SL*
	put() と同様に *unit_space*をロックする.	*XL*
concatenate()	複合されるオブジェクト *object*をロックする.	*XL*
	*object*を構成するオブジェクトをロックする.	*XL*
separate()	切り離されるオブジェクト *object*をロックする.	*XL*
	*object*を構成するオブジェクトをロックする.	*XL*
create()	クラス *object*をロックする.	*SL*
drop()	クラス *object*をロックする.	*SL*

10.1.4 補償メソッド

トランザクションが障害から復旧するためには、更新した状態をログとして記録する必要がある。変化する以前の状態をログに記録する方法がこれまで用いられている。トランザクションが大量のオブジェクトを操作する場合には、ログに記録するデータ量が増大する問題がある。この問題を解決するために、メソッドをログに記録する方法を用いる。この方法では、ログ内に記録されたメソッドを用いて復旧を行なう。そこで、各メソッドopに対して、システムの状態を元に戻せるメソッドを補償メソッドop^{-}とする。 システムの状態 sに、メソッドopを適用した結果を op(s)とする。補償メソッドは、各メソッドに対して、意味的に与えられるものである。

[定義] op^{-}がopの補償メソッドであるとは、両者は同一のオブジェクトoのメソッドであり、システムの各状態 Dに対して$op^{-}(op(D))$ =D であることである。

[例 4] 本システムのメソッドとその補償メソッドを表10.2に示す。例えば、複数のオブ

ジェクトを１つのオブジェクトにする concatenateの補償メソッドは、１つのオブジェクトを複数に分割するseparateである。図10.6で、B.put( )¯は B.remove( )であり、B.remove( )¯はB.put( )である。 {C =A.concatenate(B)} は C.separate( )であり、C.separate( )¯は C=A.concatenate(B)である。

表10.2 補償メソッド

Table 2: Compensating methods

メソッド	補償メソッド	メソッド	補償メソッド
put()	*remove*()	*remove*()	*put*()
concatenate()	*separate*()	*separate*()	*concatenate*()
create()	*drop*()	*drop*()	*create*()
insert()	*delete*()	*delete*()	*insert*()

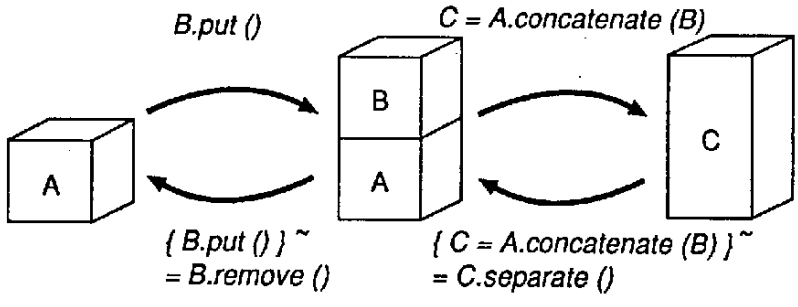


図10.6 補償メソッドの例

10.2 トランザクション

ここでは、10.1システムモデルでのトランザクショクについて述べる。

10.2.1 階層型トランザクション

CAD トランザクションは、オブジェクとのメソッドから構成される。さらに、各メソッドは他のオブジェクトを操作するメソッドから構成される。トランザクションは、メソッドが他のメソッドを含むような階層構造を持ち、いわゆる、階層型のトランザクションとな

る。

[例5] 図10.7に示すトランザクションT_1は、立方体A とB を複合して、複合オブジェクトC を作り、さらに、Cと立方体D を複合してEを作る。objectのメソッドA =cub.create( ) は、A をobjectのインスタンスcubeとして生成する。これは、cubeのcreateを起動する。cubeのcreateは、vertex, edge, faceのcreateを起動し、各クラスのインスタンスを生成し、cube, face, edgeのwrite を起動する。A.put( )は、空間にオブジェクトを置くことができるかを調べるため、space のput を起動する。また、A がcubeであればcubeのput, composed_obj であればcomposed_objのput を起動する。また、space のput は、space のstate を起動して空間の物体の関係を求める。cubeのput は、vertexのwrite を起動し座標を書き込む。AとBをobjectのオブジェクトとする。C =B.concatenate(A)は、objectのC=composed。bj.create( )を起動し、複合オブジェクトC を生成する。C.insert(A,B) により、C を、A とB を複合したオブジェクトとする。このように、ここでのシステムでのトランザクションは、階層型となる。親子関係がメソッドとそれに呼び出されるメソッドの関係を表す木構造により、T_1を書ける。これをトランザクション木とする。

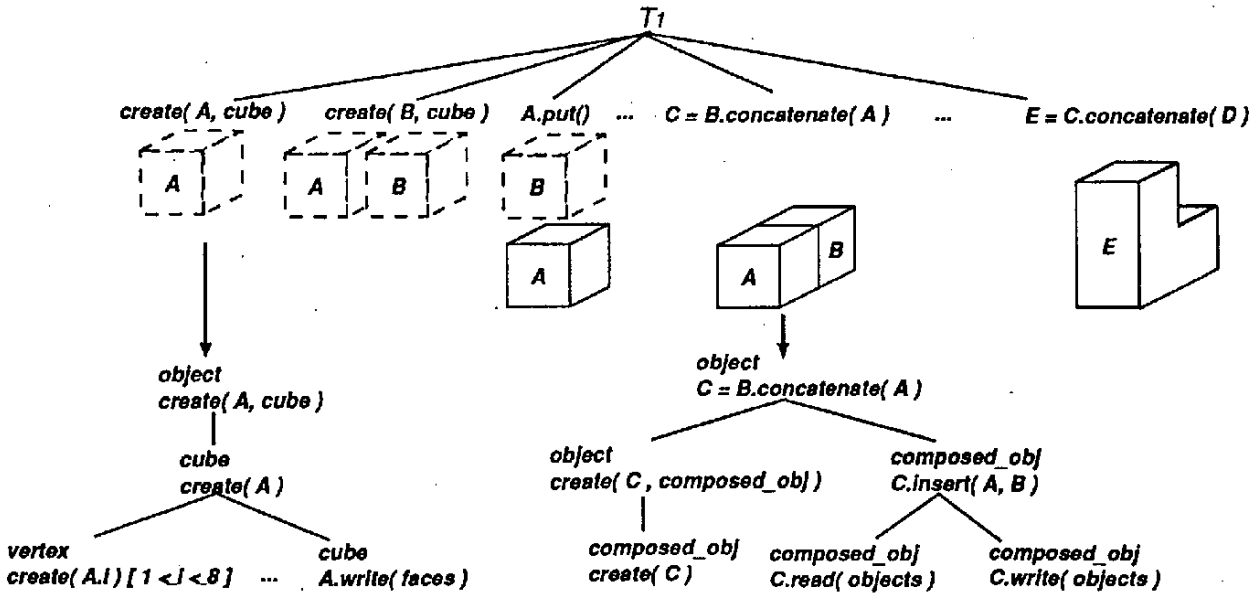


図10.7 階層型トランザクションT_1

10.2.2 半開方式

opをオブジェクトo のメソッドとする。opの実行が完了したとき、opはコミットしたとする。$op_1, \cdots, op_n$ を、各々オブジェクト$o_1, \cdots, o_n$ のメソッドとする。opが、$op_1, \cdots, op_n$ を起動するとする。ここでは、以下の実行方式を用いる。これを半開方式とする。

［半開方式］

(1)opの実行前に、o をロックする。

(2)op_i （$1 \leq i \leq n$）の実行前に、o_i （$1 < i < n$）をロックする。

(3)$op_1, \cdots, op_n$ が終了した時、$op_1, \cdots, op_n$ によって獲得された$o_1, \cdots, o_n$ のロックを解放する。

半開方式では、o のコミット時に、$o_1, \cdots, o_n$ のロックは解放されるが、o は解放されない。一方、2相ロック方式［5］では、$o_1, \cdots, o_n$ とともに、o の解放も行なわない。また、トランザクション全体がコミットしたときに、獲得した全オブジェクトが解放される。このため、半開方式は、2相ロック方式よりもオブジェクトをロックしている時間が短くなる。

［例6］図10.7のトランザクションT_1によって作られたオブジェクトE を取り除くトランザクションT_2を図10.8に示す。T_2は、E.remove( ) を実行する。 Eは、 Cと Dの複合オブジェクトであるので、C.remove( ) とD.remove( ) を実行する。同様に、C は、A とB の複合オブジェクトであるので、A.remove( ) とB.remove( ) を実行する。従って、図10.8に示すメソッドの関係がある。また、図10.8のT_2のメソッドの実行順序と半開方式の例を図10.9に示す。図10.9において、ロックの獲得を［で示し、ロックの解放を］で示す。例えば、［E.remove( ) は、E をモードXLでロックすることを意味する。E は、E.remove( ) を実行する前にロックされる。E.remove( ) の実行、即ち、composed_obj E の E.remove ( ) の実行が終了したときに、ロックは解放される。また、cube Aは、A.remove( ) を実行する前にロックされる。A のロックの解放は、A.remove( ) の実行が終了したときである。

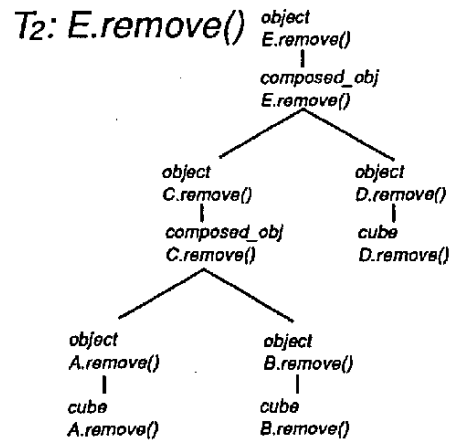


図10.8　トランザクションT_2

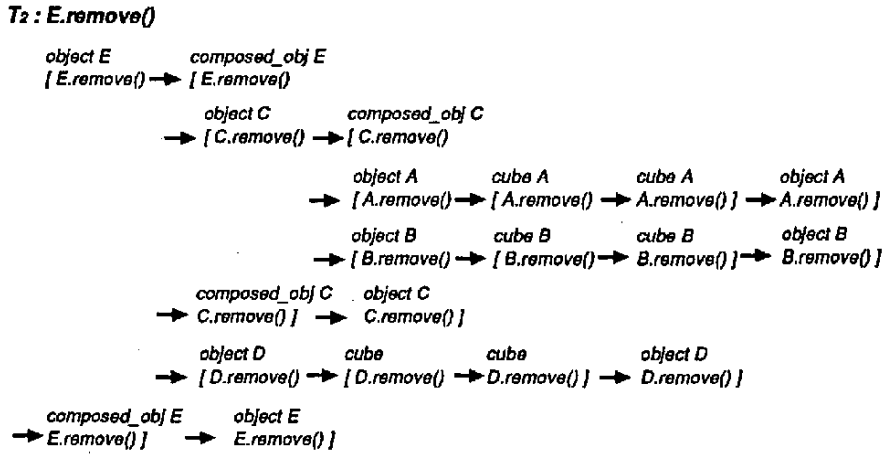


図10.9　半開方式

10.3　補償

ここでは、トランザクションが障害を起こした場合のアボート方法について述べる。その方法として補償メソッドを用いてアボートする方法を考える。

10.3.1　部分復旧

CAD 等の応用におけるデータベースシステムでは、従来のトランザクションと比較してトランザクションの実行時間は長く、より多くのオブジェクトが操作される。このため、実行された全メソッドをアボートするのではなく、障害から復旧するのに必要な部分のみを補償メソッドを用いてアボート（部分的復旧）する。利用者によるトランザクションの停止，デッドロック等によるトランザクションのアボート等のトランザクション障害を考える。部分的復旧方法の例を以下に示す。

［**例 7**］図10.7のT_1が、D.put( )を終了した時に、B.put( )が終了した時点まで復旧する場合を考える。図10.10 に、従来のアボート方法と部分的アボート方法を示す。アボート方法では、T_1が実行した全メソッドがアボートされる。即ち、create(A, cube) →create (B, cube) →A.put( )→B.put( )を実行する。これに対して、部分的アボート方法では、最も高いレベルでコミットしたメソッドの補償メソッドにより、トランザクションをアボートする。実行したメソッドopが、起動しているメソッド$op_1 \cdots op_n$ を全てコミットしたたとき、ログから$op_1 \cdots op_n$ を除くことができる。このために、ログ内に記録するデータ量を減少できる。このとき、実行する補償メソッドは、$D.put( )^{-}$=D.remove( ) → $create(D, cube)^{-}$=drop(D) → $\{C = B.concatenate(A)\}^{-}$=C.separate( ) である。

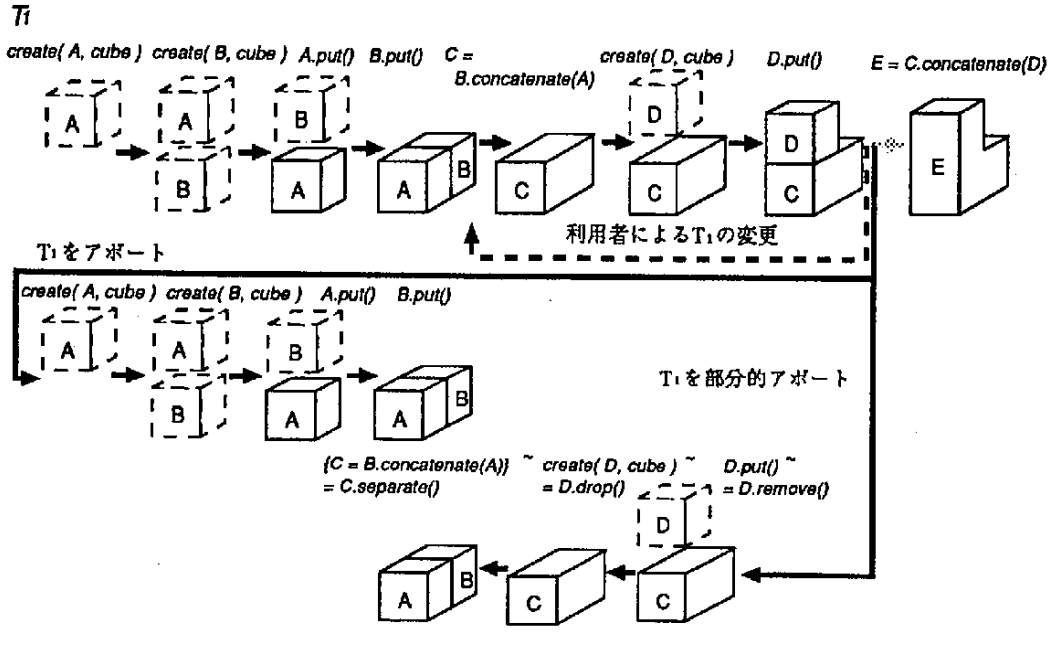


図10.10　アボート方法

10.3.2　意味的に同値な状態への復旧

構造は異なるが、形状が同じであるオブジェクトを意味的に同値なオブジェクトとする。構造とは、構成しているオブジェクトとオブジェクト間の関係である。図10.11 に、意味的な同値なオブジェクトの例を示す。オブジェクトE とS は、構造は異なるが形状が同じてある。このとき、E とS は、意味的に同値なオブジェクトである。システムの状態D とD' に対して、この中のすべてのオブジェクトが意味的に同値であるとき、D とD' は意味的に同値とする。トランザクションをアボートするとき、過去と同じ状態に復旧するのではなく、意味的に同値な状態に復旧する方法を考える。以下に意味的に同値な状態への復旧方法について例を用いて述べる。

[例8] 図10.8に示すように、E を取り除くトランザクションはE.remove( ) である。E.remove( ) の補償メソッドは、E.put( )である。ここで、E とS は、意味的に同値な状態であるので、E.remove( )の補償メソッドとして、S.put( )を用いることができる。E.put( ) を実行したときのシステムの状態と、S.put( )を実行したときのシステムの状態は、意味的に同値な状態となる。また、S.remove( ) の補償メソッドとして、E.put( )を用いることができる。どちらの系列を用いるかは、最適化の問題となる。

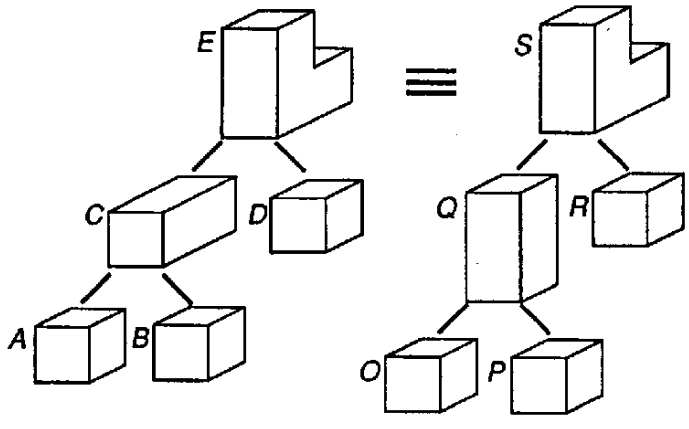


図10.11　意味的同値関係

10.3.3 最適な復旧

復旧のために実行される補償メソッドを減少させることにより、復旧時間の短縮を試みる。復旧するための補償メソッドの系列の中には、省略できるメソッドの組が存在することがある。例えば、復旧に必要なメソッドの系列がA.put( ), A.remove( ) であるとき、復旧前後の状態は同じとなる。このような、省略可能なメソッドを実行しないことにより、補償メソッドを減少できる。省略できるメソッドの条件は以下の通りである。

[補償縮退条件] L をログする。op_1 とop_2 を L内のメソッドで、同じオブジェクトについてのメソッドとする。 Lは、系列$\alpha op_1\ \beta op_2\ \gamma$とする。ここで、$\alpha$、$\beta$、$\gamma$は、メソッドの系列とし、$op_2 = op_1^{-}$とする。このとき、以下の条件を満足するとき、L を、補償縮退系列$\alpha\ \beta'\ \gamma$に縮退できる。

(1)β内に、o についてのメソッドがない。このとき、$\beta = \beta'$ である。又は、

(2)β' が、βの補償縮退系列である。

例えば、・・・A.put( ) B.put( ) C =A.concatenate(B) D.put( ) E.put( ) C.separate( ) F.put( )・・・ は、・・・ A.put( ) B.put( ) D.put( ) E.put( ) F.put( )・・・ に、縮退可能である。

次に、オブジェクト$o_1, \cdots, o_n$ から複合オブジェクトo が構成される場合を考える。トランザクションが、$o_1, \cdots, o_n$ からo を生成しているときには、o を除くことにより、この操作を補償できる。逆に、o を$o_1, \cdots, o_n$ に分割した後に、$o_1, \cdots, o_n$ を消去することは、o をput することにより補償できる。

[構成縮退条件]

(1) Lが、$\alpha_1\ o_1.put( )\ \alpha_2 \cdots \alpha_n\ o_n.put( )\ \beta o = o_1.concatenate\ (o_2.concatenate\ (\cdots o_n)\cdots)\ \gamma$のとき、L を$\alpha_1 \cdots \alpha_n\ \beta\ \gamma$に、o.remove( ) により構成縮退できる。

(2) Lが、αo.separate( ) $\beta_1\ o_1.remove( )\ \beta_2 \cdots \beta_n\ o_n.remove( )\ \gamma$のとき、L を、$\alpha\ \beta_1 \cdots \beta_n\ \gamma$に、o.put( )により構成縮退できる。

ここで、ログL から、補償縮退、又は、構成縮退により得られるログをL' とする。L から L' を得ることを縮退とする。L を補償メソッドでアボートするよりも、L' は少ないメソッドを含むので、アボート時間を減少できる。また、ログの記憶量も減少できる。

10.4 まとめ

以上、 CADシステムで広く用いられているソリッドモデルを例として、オブジェクト指向データベースシステムでのトランザクションの補償メソッドを用いた復旧方法を考えた。以上で述べた方法では、従来のデータベースシステムのようにトランザクション全体をアボートするのではなく、部分的にアボートする。ログにはシステムの状態ではなく、メソッドを記録するためにログの大きさを減少できる。本システムは、UniSQLを用いてSUN 上に実装されている。さらに、復旧方法として、過去の同一の状態に復旧するのではなく、意味的に同値を状態に復旧する方法を考えた。これによって、過去の同一の状態に復旧できない場合の復旧として、意味的に同値な状態への復旧が可能となる。また、最適な復旧を考えることで、補償のためのメソッドの実行の数を減少できる。
今後の課題として、複数の利用者がシステムを扱うときに生じる問題（同時実行制御の問題、デッドロックの問題）を考える必要がある。

10.5 今後の課題

グループウェアが実現することによって、在宅勤務がより身近なものとなり、今後、作業形態も変革していこう。しかし、それぞれの利用者が持つ個性や立場の違いから生じる問題も少なくない。特に、数多くの個性的な利用者がシステムを扱う場合、これまで以上に同時実行制御やデッドロックに関して注意を払う必要がある。また、分散型トランザクション管理方式におけるセキュリティの問題がある。つまり、ネットワークや情報の利用が極めて「オープン」であるだけに、ＣＡＤ等企業の戦略的な情報についてのセキュリティの保護について十分配慮しなければならない。

参考文献

[1] Bernstein, P. A., Hadzilacos, V., and Goodman, N.: *Concurrency Control and Recovery in Database Systems, Addisson Wesley,* (1987).

[2] CODASYL: *Data Description Language Journal of Development, Canadian Government Publishing Center,* (1973).

[3] Codd, E. F.: *A Relational Model for Large Data Banks, CACM,* Vol.13, No.6, pp.377-387 (1970).

[4] Deen, S, M., Hamada, S., and Takizawa, M. : *Broad Path Decision in Vehicle System, Proc. of International Conference on Database and Expert Systems Applications,* pp.8-13 (1992).

[5] Eswaren, K. P., Gray, J., Lorie, R. A., and Traiger, I. L., *The Notion of Consistency and Predicate Locks in Database Systems, CACM,* Vol.19, No.11, pp.624-637 (1976).

[6] Garcia-Molina, H. and Salem, K.: *Sagas, Proc. of the ACM SIGMOD,* pp.249-259 (1987).

[7] Gray, J.: *The Transaction Concept : Virtues and Limitations, Proc. of the 7th VLDB,* (1981).

[8] Holt, R. C.: *Some Deadlock Properties on Computer Systems, ACM Computing Surveys,* Vol.14, No.3 pp.179-196 (1972).

[9] Kemper, A. and Wallrath, M.: *An Analysis of Geometric Modeling in Database System, ACM Computing Surveys,* Vol.19, No.1, pp.47-91 (1987).

[10] Kim W., et al.: *Composite Object Support in an Object-Oriented Database System, Proc. of the Data Engineering Conf.,* (1987).

[11] Kim W., et al.: *Features of the ORION Object-Oriented Database System, Object-Oriented Concepts, Databases, and Applications, acm press,* pp.251-281 (1989).

[12] Knapp, E.: *Deadlock Detection in Distributed Databases, ACM Computing Surveys,* Vol.19, No.4, pp.303-328 (1987).

[13] Korth, H. ,F.: *Locking Primitives in a Database System, JACM,* Vol.30, No.1, pp.55-79 (1983).

[14] Lynch, N. and Merritt, M. *Introduction to the Theory of Nested Transactions, MIT/LCS/TR 367,* (1986).

[15] 松下温: 図解 グループウェア入門, オーム社, (1991).

[16] Moss, J. E.: *Nested Transactions : An Approach to Reliable Distributed Computing, The MIT Press Series in Information Systems,* (1985).

[17] 滝沢誠: データベース入門技術解説, ソフト・リサーチ・センター, (1991).

[18] Takizawa, M. and Deen, S, M.: *Lock Mode Based Resolution of Uncompensatable Deadlock in Compensating Nested Transactions, Proc. of the 2nd Far-East Workshop on Future Database Systems,* pp.168-175 (1992).

[19] 遠山茂樹: インテリジェントソリッドモデル入門, オーム社, (1988).

[20] UniSQL, Inc.: *UniSQL/X Release 1.0 Installation and Administration Guide, UniSQL/X User's Manual, UniSQL, Inc.,* (1991).

[21] 安沢伸二, 滝沢誠, Deen, S, M.: 補償不可能デッドロックの解除法, 情報処理学会論文誌, (1993).

平成6年3月発行

発　行　財団法人　データベース振興センター
東京都港区浜松町二丁目4番1号
世界貿易センタービル7階
TEL　03（3459）8581

委託先　株式会社　新世代システムセンター
東京都新宿区富久町11-5-402
TEL　03（3355）5184

印　刷　株式会社　エーヴィスシステムズ
東京都文京区本郷2-39-5
TEL　03（3816）4111