データ中心のシステムとアプリケーション

エドワード・シオレ

データベースの設計と実装

第2版

Springer

データ中心のシステムとアプリケーション

More information about this series at http://www.springer.com/series/5258

Edward Sciore

データベースの設計と実装

第2版

Springer

Edward Sciore
Boston College
Chestnut Hill, MA, USA

ISSN 2197-9723 ISSN 2197-974X (electronic)
Data-Centric Systems and Applications
ISBN 978-3-030-33835-0 ISBN 978-3-030-33836-7 (eBook)
https://doi.org/10.1007/978-3-030-33836-7

この本の初版はJohn Wiley & Sons, Inc.から出版されました。

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

序文

データベース システムは、企業の世界では一般的な目に見えるツールです。従業員は、データを送信したりレポートを作成したりするために、データベース システムと直接やり取りすることがよくあります。データベース システムは、ソフトウェア システムのコンポーネントとしても一般的ですが、目に見えません。たとえば、顧客、製品、および販売情報を保持するためにサーバー側のデータベースを使用する e コマース Web サイトを考えてみましょう。または、道路地図を管理するために組み込みデータベースを使用する GPS ナビゲーション システムを考えてみましょう。これらの例の両方で、データベース システムの存在はユーザーから隠されており、アプリケーション開発者の観点から見るとデータベースを直接使用することを学ぶことはむしろ平凡なことです。なぜなら、現代のデータベース システムには、クエリやレポートを簡単に作成できる洗練されたフロント エンドが含まれているからです。一方、データベース機能をソフトウェア アプリケーションに組み込む可能性は、新しい未開拓の機会を豊富に開くため、刺激的です。

ソフトウェア開発者の観点から見るとデータベースを直接使用することを学ぶことはむしろ平凡なことです。なぜなら、現代のデータベース システムには、クエリやレポートを簡単に作成できる洗練されたフロント エンドが含まれているからです。一方、データベース機能をソフトウェア アプリケーションに組み込む可能性は、新しい未開拓の機会を豊富に開くため、刺激的です。

しかし、「データベース機能を組み込む」とはどういう意味でしょうか。データベース システムは、永続性、トランザクション サポート、クエリ処理など、多くの機能を提供します。これらの機能のうちどれが必要で、どのようにソフトウェアに統合すればよいでしょうか。たとえば、プログラマーが既存のアプリケーションを変更するように求められたとします。たとえば、状態を保存する機能を追加したり、信頼性を高めたり、ファイル アクセスの効率を改善したりします。プログラマーは、いくつかのアーキテクチャ オプションに直面します。次のようなオプションがあります

- フル機能の汎用データベースシステムを購入し、アプリケーションを変更してクライアントとしてデータベースに接続する
- 必要な機能のみを含み、そのコードをアプリケーションに直接埋め込むことができる、より専門的なシステムを取得します。
- 必要な機能を自分で書く

適切な選択を行うには、プログラマーはこれらのオプションのそれぞれが何を意味するかを理解する必要があります。データベース システムが何を行うかだけでなく、その方法と理由も知っておく必要があります。

このテキストでは、ソフトウェア開発者の観点からデータベース システムを検証します。この観点から、データベース システムがなぜそのようになっているのかを調査できます。クエリを記述できることはもちろん重要ですが、クエリがどのように処理されるかを知ることも同様に重要です。JDBC を使用するだけでなく、API にクラスとメソッドが含まれている理由を知りたいのです。ディスク キャッシュやログ機能の記述がどれだけ難しいかを知る必要があります。そもそも、データベース ドライバーとはいったい何なのでしょうか。

テキストの構成

最初の 2 章では、データベース システムとその使用法について簡単に説明します。第 1 章では、データベース システムの目的と機能について説明し、Derby および SimpleDB システムを紹介します。第 2 章では、Java を使用してデータベース アプリケーションを作成する方法について説明します。データベースと対話する Java プログラムの基本 API である JDBC の基礎を紹介します。

第 3 章から第 11 章では、一般的なデータベース エンジンの内部構造について検討します。各章では、異なるデータベース コンポーネントを取り上げます。最も低いレベルの抽象化 (ディスクおよびファイル マネージャ) から始まり、最も高いレベルの抽象化 (JDBC クライアント インターフェイス) までを取り上げます。各コンポーネントの章では、問題を説明し、考えられる設計上の決定を検討します。その結果、各コンポーネントが提供するサービスと、システム内の他のコンポーネントとのやり取りを正確に把握できます。このパートの終わりまでに、シンプルでありながら完全に機能するシステムが徐々に開発されていく様子を目の当たりにすることができます。

残りの 4 つの章では、効率的なクエリ処理に焦点を当てています。これらの章では、前述の単純な設計上の選択に代わる高度な手法とアルゴリズムについて説明します。トピックには、インデックス作成、並べ替え、インテリジェントなバッファの使用、クエリの最適化などがあります。

テキストの前提条件

このテキストは、コンピュータ サイエンスの上級学部または大学院初級コースを対象としています。読者が基本的な Java プログラミングに精通していることを前提としています。たとえば、java.util のクラス、特にコレクションとマップを頻繁に使用します。高度な Java の概念 (RMI や JDBC など) については、テキストで詳しく説明しています。

この本の内容は、通常、データベース システムの 2 番目のコースとして学習されます。ただし、データベースの経験がない学生にこの本を教えることに成功しています。そのため、この本では、SQL の知識以外にデータベースに関する事前の知識は不要であると想定しています。SQL に関するそのような知識がない学生でも、必要な内容を簡単に理解できるでしょう。

SimpleDB ソフトウェア

私の経験では、学生にとって、概念的なアイデア (同時実行制御、バッファ管理、クエリ最適化アルゴリズムなど) を理解するのは、これらのアイデアがどのように相互作用するかを理解するよりもはるかに簡単です。理想的には、学生がコンパイラ コースでコンパイラ全体を作成するのと同じように、コースワークの一部としてデータベース システム全体を作成する必要があります。ただし、データベース システムはコンパイラよりもはるかに複雑なので、このアプローチは現実的ではありません。私の解決策は、SimpleDB と呼ばれる、シンプルでありながら完全に機能するデータベース システムを作成することでした。学生は、SimpleDB コードを調べて変更することで、概念的な知識を適用できます。

SimpleDB は、機能と構造の両方において、商用データベース システムのように見えます。機能的には、SQL ステートメントを実行し、JDBC を介してクライアントと対話する、マルチユーザー、トランザクション指向のデータベース サーバーです。構造的には、商用システムと同じ基本コンポーネントと類似の API が含まれています。SimpleDB の各コンポーネントには、テキスト内の対応する章があり、コンポーネントのコードとその背後にある設計上の決定について説明しています。

SimpleDB は、コードが小さく、読みやすく、簡単に変更できるため、教育に便利なツールです。不要な機能はすべて省略され、SQL のごく一部のみが実装され、最も単純な (そして多くの場合非常に非実用的な) アルゴリズムのみが使用されています。そのため、学生が追加機能やより効率的なアルゴリズムを使用してシステムを拡張する機会が数多くあります。これらの拡張の多くは、章末の演習として提供されます。

SimpleDB は http://cs.bc.edu/~sciore/simpledb からダウンロードできます。SimpleDB のインストールと使用に関する詳細は、この Web ページと第 1 章に記載されています。コードの改善に関する提案やバグの報告を歓迎します。sciore@bc.edu まで電子メールでお問い合わせください。

章末の読み物

このテキストは、2 つの質問から生まれました。データベース システムはどのような機能を提供するのか。その機能を最もよく実装するには、どのようなアルゴリズムと設計上の決定が必要なのか。これらの質問のさまざまな側面を扱った本を書棚いっぱいに並べることもできます。1 冊のテキストですべてを網羅することは不可能なので、ここでは、関連する問題を最も明確に示しているアルゴリズムと手法のみを紹介することにしました。私の最大の目標は、手法の背後にある原理を教えることであり、そのためには、最も商業的に実現可能なバージョンの説明を省略 (または削減) する必要があります。その代わりに、各章の最後には「推奨文献」のセクションがあります。これらのセクションでは、テキストでは触れられなかった興味深いアイデアや研究の方向性について説明し、関連する Web ページ、研究記事、リファレンス マニュアル、書籍への参照を提供します。

章末の練習問題

各章の最後には、多数の演習問題が掲載されています。一部の演習問題は、その章で学んだ概念を強化するために、紙と鉛筆で書く形式になっています。その他の演習問題では、SimpleDB の興味深い変更方法が紹介されており、その多くは優れたプログラミング プロジェクトになります。ほとんどの演習問題の解答は私が作成しました。この教科書を使用するコースの講師で、解答マニュアルのコピーが必要な場合は、sciore@bc.edu までメールでお問い合わせください。

コンテンツ

著者について

Edward Sciore は、ボストン カレッジのコンピュータ サイエンス学部の准教授で、最近退職しました。彼は、データベース システムに関する理論と実践の両方にわたる多数の研究論文を執筆しています。しかし、彼の最も好きな活動は、熱心な学生にデータベース コースを教えることです。この 35 年間にわたる教育経験が、このテキストの執筆につながりました。

第1章 データベース システム

データベース システムは、コンピュータ業界で重要な役割を果たしています。一部のデータベース システム (Oracle など) は非常に複雑で、通常は大型の高性能マシンで実行されます。その他のデータベース システム (SQLite など) は小さく、合理化されており、アプリケーション固有のデータの保存を目的としています。用途は多岐にわたりますが、すべてのデータベース システムは同様の機能を備えています。この章では、データベース システムが対処しなければならない問題と、データベース システムに期待される機能について説明します。また、本書で取り上げる Derby および SimpleDB データベース システムについても紹介します。

1.1 なぜデータベースシステムなのか?

データベースとは、コンピュータに保存されているデータの集合です。データベース内のデータは通常、従業員記録、医療記録、販売記録などのレコードに整理されます。図 1.1 は、大学の学生と受講したコースに関する情報を保持するデータベースを示しています。このデータベースは、本書全体を通じて実行例として使用されます。図 1.1 のデータベースには、次の 5 種類のレコードが含まれています。

- 大学に通った学生ごとに学生記録があります。各記録には、学生の ID 番号、名前、卒業年度、および学生の専攻部門の ID が含まれています。
- 大学の各学部には DEPT レコードがあります。各レコードには、学部の ID 番号と名前が含まれています。
- 大学が提供するコースごとに COURSE レコードがあります。各レコードには、コースの ID 番号、タイトル、コースを提供する学部の ID が含まれています。
- これまでに提供されたコースの各セクションには、SECTION レコードがあります。各レコードには、セクションの ID 番号、セクションが提供された年、コースの ID、およびそのセクションを教える教授が含まれます。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_1

STUDENT	SId	SName	GradYear	MajorId
	1	joe	2021	10
	2	amy	2020	20
	3	max	2022	10
	4	sue	2022	20
	5	bob	2020	30
	6	kim	2020	20
	7	art	2021	30
	8	pat	2019	20
	9	lee	2021	10

DEPT	DId	DName
	10	compsci
	20	math
	30	drama

COURSE	CId	Title	DeptId
	12	db systems	10
	22	compilers	10
	32	calculus	20
	42	algebra	20
	52	acting	30
	62	elocution	30

ENROLL	EId	StudentId	SectionId	Grade
	14	1	13	A
	24	1	43	C
	34	2	43	B+
	44	4	33	B
	54	4	53	A
	64	6	53	A

SECTION	SectId	CourseId	Prof	YearOffered
	13	12	turing	2018
	23	12	turing	2016
	33	32	newton	2017
	43	32	einstein	2018
	53	62	brando	2017

図1.1 大学データベースのレコードの一部

- 学生が受講したコースごとに ENROLL レコードがあります。各レコードには、登録 ID 番号、学生の ID 番号、受講したコースのセクション、学生がそのコースで取得した成績が含まれます。

図 1.1 は、いくつかのレコードの概念図にすぎません。レコードの保存方法やアクセス方法については何も示していません。レコードを管理するための広範な機能を提供する、データベース システムと呼ばれるソフトウェア製品が多数あります。

レコードを「管理する」とはどういう意味でしょうか? データベース システムに必須の機能とオプションの機能はどれでしょうか? 次の 5 つの要件が基本的な要件です。

- データベースは永続的である必要があります。そうでないと、コンピューターの電源を切るとすぐに記録が消えてしまいます。
- データベースは共有できます。大学のデータベースなど、多くのデータベースは複数の同時ユーザーによる共有を想定しています。
- データベースは正確に保つ必要があります。ユーザーがデータベースの内容を信頼できない場合、データベースは役に立たなくなり、価値がなくなります。
- データベースは非常に大きくなることがあります。図 1.1 のデータベースには 29 件のレコードしか含まれておらず、これは非常に小さいです。データベースに数百万 (または数十億) 件のレコードが含まれることは珍しくありません。
- データベースは使いやすくなければなりません。ユーザーが必要なデータに簡単にアクセスできない場合、生産性が低下し、別の製品を求めることになります。

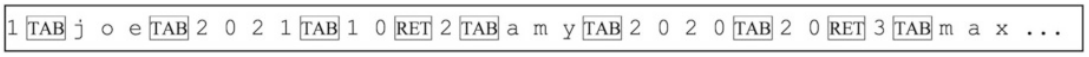


図1.2 テキストファイルでの学生記録の実装

次のサブセクションでは、これらの要件の影響について説明します。各要件により、データベース システムにさらに多くの機能が含まれるようになり、予想以上に複雑になります。

1.1.1 記録の保存

データベースを永続的にする一般的な方法は、レコードをファイルに保存することです。最も単純で直接的な方法は、データベース システムがレコードをテキスト ファイルに保存することです。レコードの種類ごとに 1 つのファイルを作成します。各レコードは 1 行のテキストで、値はタブで区切られます。図 1.2 は、STUDENT レコードのテキスト ファイルの先頭部分を示しています。

この方法の利点は、ユーザーがテキスト エディターを使用してファイルを調べたり変更したりできることです。残念ながら、この方法は 2 つの理由から非効率的すぎて役に立ちません。

最初の理由は、大きなテキスト ファイルの更新に時間がかかりすぎることです。たとえば、誰かが STUDENT ファイルから Joe のレコードを削除したとします。データベース システムは、Amy のレコードから始めて、後続の各レコードを左に移動しながらファイルを書き換えるしかありません。小さなファイルの書き換えにかかる時間は無視できるほど小さいですが、1 GB のファイルの書き換えには数分かかる可能性があり、これは許容できないほど長い時間です。データベース システムは、レコードの保存方法をより巧妙にし、ファイルの更新には小さなローカル書き換えのみが必要になるようにする必要があります。

2 つ目の理由は、大きなテキスト ファイルの読み取りに時間がかかりすぎることです。2019 年度のクラスの学生を STUDENT ファイルで検索する場合を考えてみましょう。唯一の方法は、ファイルを順番にスキャンすることです。順番にスキャンするのは非常に非効率的です。おそらく、高速な検索を可能にするツリーやハッシュ テーブルなどのメモリ内データ構造をいくつかご存知でしょう。データベース システムでは、類似のデータ構造を使用してファイルを実装する必要があります。たとえば、データベース システムは、特定の種類の検索 (学生名、卒業年度、専攻など) を容易にする構造を使用してファイル内のレコードを整理したり、それぞれが異なる種類の検索を容易にする複数の補助ファイルを作成したりすることがあります。これらの補助ファイルはインデックスと呼ばれ、第 12 章で説明します。

1.1.2 マルチユーザーアクセス

多くのユーザーがデータベースを共有する場合、データファイルに同時にアクセスする可能性が高くなります。同時実行性は、他のユーザーの処理が完了するのを待たずに各ユーザーが迅速に処理できるため、良いことです。しかし、

同時実行が多すぎると、データベースが不正確になる可能性があるため、好ましくありません。たとえば、旅行計画データベースを考えてみましょう。2 人のユーザーが、残り 40 席のフライトの座席を予約しようとしたとします。両方のユーザーが同時に同じフライト レコードを読み取ると、両方とも利用可能な座席が 40 席であることがわかります。その後、両方ともレコードを変更して、フライトの利用可能な座席が 39 席になるようにします。おっと、2 つの座席が予約されているのに、データベースには 1 つの予約しか記録されていません。この問題の解決策は、同時実行を制限することです。データベース システムは、最初のユーザーがフライト レコードを読み取って 40 席の空きがあることを確認できるようにし、最初のユーザーが完了するまで 2 番目のユーザーをブロックする必要があります。2 番目のユーザーが再開すると、39 席の空きがあることがわかり、それを 38 に修正します。一般に、データベース システムは、ユーザーが別のユーザーのアクションと競合するアクションを実行しようとしていることを検出し、最初のユーザーが完了するまでそのユーザーの実行をブロックする必要があります (競合した場合のみ)。

ユーザーは、データベースの更新を元に戻す必要がある場合もあります。たとえば、ユーザーが旅行計画データベースでマドリッドへの旅行を検索し、利用可能なフライトと空室のあるホテルの両方がある日付を見つけたとします。次に、ユーザーがフライトを予約したが、予約処理中にその日付のホテルがすべて満室になったとします。この場合、ユーザーはフライトの予約を取り消して、別の日付を試す必要がある場合があります。

元に戻せる更新は、データベースの他のユーザーには見えないようにしてください。そうしないと、他のユーザーが更新を見て、データが「本物」であると考え、それに基づいて決定を下す可能性があります。したがって、データベース システムは、ユーザーが変更を永続的にするタイミングを指定できるようにする必要があります。ユーザーは変更をコミットすることになります。ユーザーがコミットすると、変更は表示され、元に戻せなくなります。第 5 章では、これらの問題について説明します。

1.1.3 大災害への対処

すべての教授の給与を増額するプログラムを実行しているときに、データベース システムが予期せずクラッシュしたとします。システムの再起動後、一部の教授の給与は増額されているのに、他の教授の給与は増額されていないことに気付きました。どうすればよいでしょうか。プログラムを単に再実行することはできません。そうすると、一部の教授の給与が 2 倍になってしまいます。代わりに、クラッシュが発生したときに実行されていたすべてのプログラムの更新を元に戻して、データベース システムをクラッシュから正常に回復させる必要があります。これを行うメカニズムは興味深く、簡単ではありません。第 5 章で説明します。

1.1.4 メモリ管理

データベースは、ディスクドライブやフラッシュドライブなどの永続メモリに保存する必要があります。フラッシュドライブはディスクドライブよりも約100倍高速ですが、かなり高価です。通常のアクセス時間は、ディスクの場合は約6ミリ秒、フラッシュの場合は60秒です。ただし、これらの時間はどちらもメインメモリよりも桁違いに遅くなります。

(または RAM) のアクセス時間は約 60 ns です。つまり、RAM はフラッシュの約 1000 倍、ディスクの約 100,000 倍の速度です。

このパフォーマンスの違いの影響と、データベース システムが直面する結果的な問題を理解するには、次の例え話を考えてみましょう。チョコレート チップ クッキーが食べたくなったとします。クッキーを手に入れるには、キッチンで買うか、近所の食料品店で買うか、通信販売で買うかの 3 つの方法があります。この例え話では、キッチンは RAM、近所の店はフラッシュ ドライブ、通信販売会社はディスクに相当します。キッチンからクッキーを手に入れるのに 5 秒かかるとします。類似の店からクッキーを手に入れるには 5,000 秒、つまり 1 時間以上かかります。つまり、店に行って、非常に長い列に並び、クッキーを購入して戻ってくる必要があります。また、類似の通信販売会社からクッキーを手に入れるには 500,000 秒、つまり 5 日以上かかります。つまり、クッキーをオンラインで注文して、標準の配送方法で配送してもらう必要があります。この観点からすると、フラッシュ メモリとディスク メモリはひどく遅いように見えます。待ってください。状況はさらに悪化します。同時実行性と信頼性に対するデータベースのサポートにより、処理速度がさらに低下します。必要なデータを他のユーザーが使用している場合、データがリリースされるまで待たされる可能性があります。この例えで言えば、これは食料品店に到着してクッキーが売り切れていることに気づき、補充されるまで待たされる状況に相当します。

言い換えれば、データベース システムは次のような難問に直面しています。メイン メモリ システムよりも多くのデータを、より低速のデバイスを使用して管理し、複数の人がデータへのアクセスを争う中で、妥当な応答時間を維持しながら、データを完全に回復可能にする必要があります。この難問の解決策の大部分は、キャッシュを使用することです。データベース システムは、レコードを処理する必要があるときはいつでも、それを RAM にロードし、できるだけ長くそこに保持します。メイン メモリには、現在使用中のデータベースの部分が含まれます。すべての読み取りと書き込みは RAM で行われます。この戦略には、低速の永続メモリの代わりに高速のメイン メモリが使用されるという利点がありますが、データベースの永続バージョンが古くなる可能性があるという欠点があります。データベース システムは、システム クラッシュ (RAM の内容が破壊される) が発生した場合でも、データベースの永続バージョンを RAM バージョンと同期させる手法を実装する必要があります。第 4 章では、さまざまなキャッシュ戦略について説明します。

1.1.5 ユーザビリティ

ユーザーが必要なデータを簡単に抽出できない場合、データベースはあまり役に立ちません。たとえば、ユーザーが 2019 年に卒業したすべての学生の名前を知りたいとします。データベース システムがなければ、ユーザーは学生のファイルをスキャンするプログラムを書かざるを得なくなります。図 1.3 は、ファイルがテキストとして保存されていると仮定した場合の、そのようなプログラムの Java コードを示しています。Java コードの大部分は、ファイルのデコード、各レコードの読み取り、および検査する値の配列への分割を処理していることに注意してください。目的の学生の名前を決定するコード (太字) は、興味をそそらないファイル操作コード内に隠されています。

```
パブリック静的List<String> getStudents2019() { List<String> result = n
ew ArrayList<> () ; FileReader rdr = new FileReader("students.txt"); Buffe
redReader br = new BufferedReader(rdr); String line = br.readLine(); while
(line != null) { String[] vals = line.split("\t"); String gradyear = vals[2]; if (
gradyear.equals("2019")) result.add(vals[1]); line = br.readLine(); }return r
esult; }
```

図1.3 2019年度卒業予定者名の検索

その結果、ほとんどのデータベース システムはクエリ言語をサポートしており、ユーザーは必要なデータを簡単に指定できます。リレーショナル データベースの標準クエリ言語は SQL です。図 1.3 のコードは、1 つの SQL ステートメントで表現できます。

```
STUDENTからSNameを選択します。GradYearは= 2019です。
```

この SQL 文は、主にファイルから抽出する値を指定するだけで、その取得方法を指定する必要がないため、Java プログラムよりもはるかに短くて明確です。

1.2 Derbyデータベースシステム

データベースの概念を学ぶには、データベース システムを使用して対話的に学習する方がはるかに効果的です。利用できるデータベース システムは多種多様ですが、Java ベースで無料、インストールも使用も簡単な Derby データベース システムを使用することをお勧めします。最新バージョンの Derby は、URL db.apache.org/derby のダウンロード タブからダウンロードできます。ダウンロードした配布ファイルは、複数のディレクトリを含むフォルダーに解凍されます。たとえば、docs ディレクトリにはリファレンス ドキュメントが含まれ、demo ディレクトリにはサンプル データベースが含まれます。完全なシステムには、ここで説明できないほど多くの機能が含まれています。関心のある読者は、docs ディレクトリ内のさまざまなガイドやマニュアルを熟読できます。

Derby には、この本では必要のない多くの機能があります。実際、クラスパスに Derby の lib ディレクトリから derby.jar、derbynet.jar、derbyclient.jar、derbytools.jar の 4 つのファイルを追加するだけで済みます。クラスパスを変更する方法は、Java プラットフォームとオペレーティング システムによって異なります。ここでは、Eclipse 開発プラットフォームを使用して変更する方法を説明します。Eclipse に詳しくない場合は、コードとドキュメントをダウンロードできます。

eclipse.org から。別の開発プラットフォームを使用する場合は、Eclipse の指示を自分の環境に合わせて調整できるはずです。

まず、Derby 用の Eclipse プロジェクトを作成します。次に、次のようにビルド パスを構成します。プロパティ ウィンドウで、「Java ビルド パス」を選択します。「ライブラリ」タブをクリックしてから、「外部 JARS の追加」をクリックし、ファイル選択ダイアログを使用して必要な 4 つの jar ファイルを選択します。これで完了です。

Derby ディストリビューションには、Derby データベースの作成とアクセスを可能にする ij というアプリケーションが含まれています。Derby は完全に Java で記述されているため、ij は実際には org.apache.derby.tools パッケージにある Java クラスの名前です。ij を実行するには、そのクラスを実行します。Eclipse からクラスを実行するには、[実行] メニューの [構成の実行] を選択します。Derby プロジェクトに新しい構成を追加し、"Derby ij" という名前を付けます。構成のメイン クラスのフィールドに "org.apache.derby.tools.ij" と入力します。構成を実行すると、ij によって入力を求めるコンソール ウィンドウが表示されます。

ij への入力は一連のコマンドです。コマンドはセミコロンで終わる文字列です。コマンドは複数のテキスト行に分割できます。ij クライアントはセミコロンで終わる行に遭遇するまでコマンドを実行しません。すべての SQL ステートメントは有効なコマンドです。さらに、ij はデータベースへの接続と切断、およびセッションの終了を行うコマンドをサポートしています。

connect コマンドは ij が接続するデータベースを指定し、disconnect コマンドはデータベースから切断します。特定のセッションは複数回接続および切断できます。exit コマンドはセッションを終了します。図 1.4 は ij セッションの例を示しています。セッションは 2 つの部分から構成されます。最初の部分では、ユーザーは新しいデータベースに接続し、テーブルを作成し、そのテーブルにレコードを挿入して切断します。2 番目の部分では、ユーザーはそのデータベースに再接続し、挿入した値を取得して切断します。

connect コマンドの引数は接続文字列と呼ばれます。接続文字列には、コロンで区切られた 3 つの部分文字列があります。最初の 2 つの部分文字列は「jdbc」と「derby」で、JDBC プロトコルを使用して Derby データベースに接続することを示します (JDBC は第 2 章で説明します)。3 番目の部分文字列は、

```
ij> connect 'jdbc:derby:ijtest;create=true';
ij> create table T(A int, B varchar(9));
0 rows inserted/updated/deleted
ij> insert into T(A,B) values(3, 'record3');
1 row inserted/updated/deleted
ij> disconnect;
ij> connect 'jdbc:derby:ijtest';
ij> select * from T;
A          |B
----------------------
3          |record3

1 row selected
ij> disconnect;
ij> exit;
```

図1.4 ijセッションの例

データベースを識別します。文字列「ijtest」はデータベースの名前です。そのファイルは、ij プログラムが起動されたディレクトリにある「ijtest」という名前のフォルダ内にあります。たとえば、Eclipse からプログラムを実行した場合、データベース フォルダはプロジェクト ディレクトリ内にあります。文字列「create ¼ true」は、Derby に新しいデータベースを作成するように指示します。この文字列が省略されている場合 (2 番目の接続コマンドのように)、Derby は既存のデータベースが見つかるものと想定します。

1.3 データベースエンジン

ij などのデータベース アプリケーションは、ユーザー インターフェイス (UI) とデータベースにアクセスするためのコードという 2 つの独立した部分から構成されています。後者のコードはデータベース エンジンと呼ばれます。UI をデータベース エンジンから分離することは、アプリケーションの開発を簡素化するため、優れたシステム設計です。この分離のよく知られた例は、Microsoft Access データベース システムです。このシステムには、マウスをクリックして値を入力することでユーザーがデータベースを操作できるグラフィカル UI と、データ ストレージを処理するエンジンがあります。UI は、データベースからの情報が必要であると判断すると、要求を作成してエンジンに送信します。エンジンは要求を実行し、値を UI に返します。

この分離により、システムの柔軟性も向上します。アプリケーション設計者は、異なるデータベース エンジンで同じユーザー インターフェイスを使用したり、同じデータベース エンジンに異なるユーザー インターフェイスを構築したりできます。Microsoft Access では、それぞれのケースの例が提供されています。Access UI を使用して構築されたフォームは、Access エンジンまたは他の任意のデータベース エンジンに接続できます。また、Excel スプレッドシートのセルには、Access エンジンにクエリを実行する数式を含めることができます。

UI は、目的のエンジンに接続し、エンジンの API からメソッドを呼び出すことによってデータベースにアクセスします。例として、Derby ij プログラムは実際には単なる UI であることに注意してください。その connect コマンドは指定されたデータベース エンジンへの接続を確立し、各 SQL コマンドは SQL ステートメントをエンジンに送信し、結果を取得して表示します。

データベース エンジンは通常、複数の標準 API をサポートします。Java プログラムがエンジンに接続する場合、選択される API は JDBC と呼ばれます。第 2 章では、JDBC について詳しく説明し、JDBC を使用して ij のようなアプリケーションを作成する方法を示します。

UI からデータベース エンジンへの接続は、埋め込みまたはサーバー ベースにすることができます。埋め込み接続では、データベース エンジンのコードは UI のコードと同じプロセスで実行され、UI はエンジンに排他的にアクセスできます。アプリケーションは、データベースがそのアプリケーションに「属し」、アプリケーションと同じマシンに保存されている場合にのみ、埋め込み接続を使用する必要があります。その他のアプリケーションは、サーバー ベースの接続を使用する必要があります。

サーバーベースの接続では、データベース エンジンのコードは専用のサーバー プログラム内で実行されます。このサーバー プログラムは常に実行されており、クライアントの接続を待機しており、クライアントと同じマシン上にある必要はありません。クライアントがサーバーとの接続を確立すると、クライアントはサーバーに JDBC 要求を送信し、応答を受信します。

サーバーは複数のクライアントに同時に接続できます。サーバーが 1 つのクライアントのリクエストを処理している間に、他のクライアントが独自のリクエストを送信できます。サーバーにはスケジューラが含まれており、サービス待ちのリクエストをキューに入れて、いつ実行するかを決定します。各クライアントは他のクライアントを認識せず、(スケジュールによる遅延を除けば) サーバーが自分だけを処理しているのかのような錯覚に陥ります。

図 1.4 の ij セッションでは、埋め込み接続が使用されました。セッションを実行しているマシン上にデータベース「ijtest」が作成され、サーバーは使用されませんでした。類似のサーバーベースの ij セッションを実行するには、2 つの変更が必要です。Derby エンジンをサーバーとして実行し、接続コマンドを変更してサーバーを識別する必要があります。

Derby サーバーのコードは、org.apache.derby.drda パッケージの Java クラス NetworkServerControl にあります。Eclipse からサーバーを実行するには、[実行] メニューの [構成の実行] を選択します。Derby プロジェクトに新しい構成を追加し、"Derby Server" という名前を付けます。メイン クラスのフィールドに "org.apache.derby.drda.NetworkServerControl" と入力します。[引数] タブで、プログラム引数 "start -h localhost" を入力します。構成を実行するたびに、Derby サーバーが実行中であることを示すコンソール ウィンドウが表示されます。

プログラム引数「start -h localhost」の目的は何でしょうか。最初の単語はコマンド「start」で、クラスにサーバーを起動するように指示します。同じクラスを引数「shutdown」で実行することでサーバーを停止できます (または、コンソール ウィンドウからプロセスを終了することもできます)。文字列「-h localhost」は、同じマシン上のクライアントからのリクエストのみを受け入れるようにサーバーに指示します。「localhost」をドメイン名または IP アドレスに置き換えると、サーバーはそのマシンからのリクエストのみを受け入れます。IP アドレス「0.0.0.0」を使用すると、サーバーはどこからでもリクエストを受け入れるように指示されます。[1]

サーバーベースの接続の接続文字列では、サーバー マシンのネットワークまたは IP アドレスを指定する必要があります。特に、次の ij connect コマンドを検討してください。

```
ij> は 'jdbc:derby:ijtest' に接続します。ij> は 'jdbc:derby://l
ocalhost/ijtest' に接続します。ij> は 'jdbc:derby://cs.bc.edu/
ijtest' に接続します。
```

最初のコマンドは、「ijtest」データベースへの埋め込み接続を確立します。2 番目のコマンドは、マシン「localhost」、つまりローカル マシンで実行されているサーバーを使用して、「ijtest」へのサーバー ベースの接続を確立します。3 番目のコマンドは、マシン「cs.bc.edu」で実行されているサーバーを使用して、「ijtest」へのサーバー ベースの接続を確立します。

接続文字列が埋め込み接続かサーバー側接続かの決定を完全にカプセル化していることに注意してください。たとえば、図1.4をもう一度考えてみましょう。次のようにして、埋め込み接続ではなくサーバー側接続を使用するようにセッションを変更できます。

[1] Of course, if you allow clients to connect from anywhere, then you expose the database to hackers and other unscrupulous users. Typically, you would either place such a server inside of a firewall, enable Derby’s authentication mechanism, or both.

接続コマンドを変更するだけです。セッション内の他のコマンドは影響を受けません。

1.4 SimpleDB データベース システム

Derby は、洗練されたフル機能のデータベース システムです。ただし、この複雑さは、ソース コードが簡単に理解または変更できないことを意味します。私は、Derby とは正反対の SimpleDB データベース システムを作成しました。つまり、コードは小さく、読みやすく、簡単に変更できるようになっています。不要な機能はすべて省略し、SQL のごく一部だけを実装し、最も単純な (そして多くの場合非常に非実用的な) アルゴリズムのみを使用しています。その目的は、データベース エンジンの各コンポーネントと、これらのコンポーネントがどのように相互作用するかを明確に理解できるようにすることです。

SimpleDB の最新バージョンは、URL cs.bc.edu/~sciore/simpledb の Web サイトからダウンロードできます。ダウンロードしたファイルは、SimpleDB_3.x フォルダーに解凍されます。このフォルダーには、simpledb、simpleclient、derbyclient の各ディレクトリが含まれます。simpledb フォルダーには、データベース エンジンのコードが含まれます。Derby とは異なり、このコードは jar ファイルにパックされておらず、各ファイルはフォルダー内に明示的に存在します。

SimpleDB エンジンをインストールするには、クラスパスに simpledb フォルダを追加する必要があります。Eclipse を使用してこれを行うには、まず「SimpleDB Engine」という新しいプロジェクトを作成します。次に、オペレーティング システムから、SimpleDB_3.x フォルダの「simpledb」というサブフォルダをプロジェクトの src フォルダにコピーします。最後に、[ファイル] メニューの [更新] コマンドを使用して、Eclipse からプロジェクトを更新します。

derbyclient フォルダには、Derby エンジンを呼び出すサンプル プログラムが含まれています。オペレーティング システムを使用して、このフォルダの内容 (フォルダ自体ではありません) を Derby プロジェクトの src フォルダにコピーし、更新します。これらのクライアント プログラムについては、第 2 章で説明します。

simpleclient フォルダーには、SimpleDB エンジンを呼び出すサンプル プログラムが含まれています。これらのサンプル プログラム用に新しいプロジェクトを作成し、「SimpleDB Clients」という名前を付けます。サンプル プログラムが SimpleDB エンジン コードを見つけられるようにするには、SimpleDB Engine プロジェクトを SimpleDB Clients のビルド パスに追加する必要があります。次に、オペレーティング システムを使用して、simpleclient の内容を SimpleDB Clients の src ディレクトリにコピーします。

SimpleDB は、埋め込み接続とサーバーベースの接続の両方をサポートします。 simpleclient フォルダー内のプログラムの 1 つに SimpleIJ があります。これは、Derby ij プログラムの簡易版です。ij との違いは、セッションの開始時に 1 回しか接続できないことです。プログラムを実行すると、接続文字列の入力を求められます。接続文字列の構文は、ij のものと似ています。たとえば、次の SimpleDB 接続文字列を考えてみましょう。

```
jdbc:simpledb:testij
jdbc:simpledb://localhost
jdbc:simpledb://cs.bc.edu
```

最初の接続文字列は、「testij」データベースへの埋め込み接続を指定します。Derby と同様に、データベースは実行プログラムのディレクトリ (SimpleDB Clients プロジェクト) に配置されます。Derby とは異なり、SimpleDB はデータベースが存在しない場合は作成するため、明示的な「create ¼ true」フラグは必要ありません。

2 番目と 3 番目の接続文字列は、ローカル マシンまたは cs.bc.edu で実行されている SimpleDB サーバーへのサーバーベースの接続を指定します。Derby とは異なり、接続文字列ではデータベースを指定しません。これは、SimpleDB エンジンが一度に処理できるデータベースが 1 つだけであり、そのデータベースはサーバーの起動時に指定されるためです。

SimpleIJ は、SQL ステートメントを含む 1 行のテキストを入力するよう求めるプロンプトを繰り返し表示します。Derby とは異なり、行にはステートメント全体を含める必要があり、末尾にセミコロンは必要ありません。次に、プログラムがそのステートメントを実行します。ステートメントがクエリの場合は、出力テーブルが表示されます。ステートメントが更新コマンドの場合は、影響を受けるレコードの数が出力されます。ステートメントの形式が正しくない場合は、エラー メッセージが出力されます。SimpleDB は SQL の非常に限られたサブセットを理解しており、エンジンが理解できない SQL ステートメントが指定されると、SimpleIJ は例外をスローします。これらの制限については、次のセクションで説明します。

SimpleDB エンジンはサーバーとして実行できます。メイン クラスは、パッケージ simpledb.server の StartServer です。Eclipse からサーバーを実行するには、[実行] メニューの [構成の実行] に移動します。SimpleDB エンジン プロジェクトに「SimpleDB Server」という新しい構成を追加します。メイン クラスのフィールドに「simpledb.server.StartServer」と入力します。[引数] タブを使用して、必要なデータベースの名前を入力します。便宜上、引数を省略すると、サーバーは「studentdb」という名前のデータベースを使用します。構成を実行すると、SimpleDB サーバーが実行中であることを示すコンソール ウィンドウが表示されます。

SimpleDB サーバーは、Derby の「-h 0.0.0.0」コマンドライン オプションに対応して、どこからでもクライアント接続を受け入れます。サーバーをシャットダウンする唯一の方法は、コンソール ウィンドウからそのプロセスを強制終了することです。

1.5 SimpleDB バージョンの SQL

Derby は標準 SQL のほぼすべてを実装しています。一方、SimpleDB は標準 SQL のごく一部のみを実装しており、SQL 標準にはない制限を課しています。このセクションでは、これらの制限について簡単に説明します。本書の他の章では、これらの制限についてさらに詳しく説明しており、章末の演習の多くでは、省略された機能の一部を実装するように求められます。

SimpleDB のクエリは、select 句にフィールド名のリスト (AS キーワードなし) が含まれ、from 句にテーブル名のリスト (範囲変数なし) が含まれる select-from-where 句のみで構成されます。

オプションの where 句内の項は、ブール演算子 and によってのみ接続できます。項は定数とフィールド名の等価性のみを比較できます。

標準 SQL とは異なり、他の比較演算子、他のブール演算子、算術演算子、組み込み関数、括弧はありません。したがって、ネストされたクエリ、集計、計算値はサポートされていません。

範囲変数や名前の変更がないため、クエリ内のすべてのフィールド名は互いに重複していてはいけません。また、group by 句や order by 句がないため、グループ化や並べ替えはサポートされていません。その他の制限は次のとおりです。

- 選択句内の「"」省略形はサポートされていません。
- null値はありません。
- from 句には明示的な結合または外部結合はありません。
- union キーワードはサポートされていません。
- 挿入ステートメントは明示的な値のみを受け取ります。つまり、挿入はクエリによって指定できません。
- 更新ステートメントでは、set 句に割り当てを 1 つだけ含めることができます。

1.6 章の要約

- データベースは、コンピューターに保存されているデータの集合です。データベース内のデータは通常、レコードに整理されます。データベース システムは、データベース内のレコードを管理するソフトウェアです。
- データベース システムは、大規模な共有データベースを処理し、そのデータを低速の永続メモリに保存できる必要があります。また、データへの高レベル インターフェイスを提供し、ユーザーによる更新の競合やシステム クラッシュが発生した場合でもデータの正確性を確保する必要があります。データベース システムは、次の機能によってこれらの要件を満たします。 – 通常のファイル システムよりも効率的にアクセスできる形式を使用して、ファイルにレコードを保存する機能 – 高速アクセスをサポートするために、ファイル内のデータをインデックス化する複雑なアルゴリズム – 必要に応じてユーザーをブロックしながら、ネットワーク経由で複数のユーザーからの同時アクセスを処理する機能 – 変更のコミットとロールバックのサポート – メイン メモリにデータベース レコードをキャッシュし、データベースの永続バージョンとメイン メモリ バージョン間の同期を管理して、システムがクラッシュした場合にデータベースを適切な状態に復元する機能 – テーブルに対するユーザー クエリをファイル上の実行可能コードに変換する言語コンパイラ/インタープリタ – 非効率的なクエリをより効率的なクエリに変換するためのクエリ最適化戦略

- データベース エンジンは、データを管理するデータベース システムのコンポーネントです。データベース アプリケーションは、ユーザーの入力と出力を担当し、必要なデータを取得するためにデータベース エンジンを呼び出します。
- データベース エンジンへの接続は、埋め込み型またはサーバーベース型のいずれかです。埋め込み型接続を持つプログラムは、データベースへの排他的アクセス権を持ちます。

エンジン。サーバーベースの接続を持つプログラムは、他の同時実行プログラムとエンジンを共有します。

- Java ベースのデータベース システムは、Derby と SimpleDB の 2 つです。Derby は完全な SQL 標準を実装していますが、SimpleDB は SQL の限定されたサブセットのみを実装しています。SimpleDB はコードが理解しやすいため便利です。この本の第 3 章以降では、このコードを詳細に検討します。

1.7 推奨される読み物

データベース システムは、長年にわたって劇的な変化を遂げてきました。これらの変化については、National Research Council (1999) の第 6 章と Haigh (2006) で詳しく説明されています。Wikipedia の en.wikipedia.org/wiki/Database_management_system#History のエントリも興味深いものです。

クライアント サーバー パラダイムは、データベースだけでなく、コンピューティングのさまざまな分野で役立ちます。この分野の概要については、Orfali ら (1999) を参照してください。Derby サーバーのさまざまな機能と構成オプションに関するドキュメントは、URL db.apache.org/derby/manuals/index.html にあります。

Haigh, T. (2006)。「真実の事実の詰まったバケツ」。データベース管理システムの起源。ACM SIGMOD Record、35(2)、33 – 49。米国国立研究会議コンピューティングと通信の革新に関する委員会。(1999)。革命への資金提供。米国アカデミー出版。www.nap.edu/read/6323/chapter/8#159 から入手可能。Orfali, R.、Harkey, D.、Edwards, J. (1999)。クライアント/サーバー サバイバル ガイド (第 3 版)。Wiley。

1.8 演習

概念演習

1.1. 組織が比較的少数の共有レコード (たとえば 100 件程度) を管理する必要があるとします。

> (a) これらの記録を管理するために商用データベース システムを使用することは理にかなっていますか? (b) データベース システムのどの機能が必要ありませんか? (c) これらの記録を保存するためにスプレッドシートを使用することは合理的ですか? 潜在的な問題は何ですか?

1.2. 大量の個人データをデータベースに保存するとします。データベース システムのどの機能が必要ではないでしょうか。

1.3. 通常、データベース システムを使用せずに管理するデータ (買い物リスト、アドレス帳、当座預金口座情報など) について考えてみましょう。

(a) データを分解してデータベース システムに保存するには、データの大きさはどの程度になる必要がありますか? (b) データの使用方法をどのように変更すれば、データベース システムを使用する価値があるでしょうか?

1.4. バージョン管理システム (Git や Subversion など) の使い方がわかっている場合は、その機能をデータベース システムの機能と比較します。

(a) バージョン管理システムにはレコードの概念がありますか? (b) チェックイン/チェックアウトは、データベースの同時実行制御とどのように対応していますか? (c) ユーザーはどのようにしてコミットを実行しますか? ユーザーはどのようにしてコミットされていない変更を元に戻すのですか? (d) 多くのバージョン管理システムは、差分ファイルに更新を保存します。差分ファイルは、ファイルの以前のバージョンを新しいバージョンに変換する方法を記述した小さなファイルです。ユーザーがファイルの現在のバージョンを確認する必要がある場合、システムは元のファイルから開始し、すべての差分ファイルをそれに適用します。 この実装戦略は、データベース システムのニーズをどの程度満たしていますか?

プロジェクトベースの演習

1.5. 学校や会社がデータベース システムを使用しているかどうかを調べます。使用している場合は、次の操作を行います。

(a) どのような従業員が仕事でデータベース システムを明示的に使用していますか? (知らないうちにデータベースを使用する「既成」プログラムを実行する従業員とは対照的に) 彼らはデータベースを何に使用していますか? (b) ユーザーがデータを使用して何か新しいことを実行する必要がある場合、ユーザーは独自のクエリを作成しますか、それとも他の誰かが作成しますか?

1.6. Derby サーバーと SimpleDB サーバーをインストールして実行します。

(a) サーバー マシンから ij プログラムと SimpleIJ プログラムを実行します。 (b) 2 台目のマシンにアクセスできる場合は、デモ クライアントを変更し、そのマシンからもリモートで実行します。

Chapter 2
JDBC

データベース アプリケーションは、API のメソッドを呼び出すことによってデータベース エンジンと対話します。Java アプリケーションで使用される API は、JDBC (Java DataBase Connectivity) と呼ばれます。JDBC ライブラリは 5 つの Java パッケージで構成されており、そのほとんどは大規模な商用アプリケーションでのみ役立つ高度な機能を実装しています。この章では、パッケージ java.sql にあるコア JDBC 機能について説明します。このコア機能は、基本的な使用に必要なクラスとメソッドを含む基本 JDBC と、利便性と柔軟性を高めるオプション機能を含む高度な JDBC の 2 つの部分に分けられます。

2.1 基本的なJDBC

JDBC の基本機能は、Driver、Connection、Statement、ResultSet、ResultSetMetadata の 5 つのインタフェースで実現されています。また、これらのインタフェースのメソッドのうち、必須のものはごくわずかです。図 2.1 に、これらのメソッドを示します。

このセクションのサンプル プログラムは、これらのメソッドの使用方法を示しています。最初のサンプル プログラムは CreateTestDB で、プログラムが Derby エンジンに接続および切断する方法を示しています。そのコードは図 2.2 に示されており、JDBC 関連のコードは太字で強調表示されています。次のサブセクションでは、このコードを詳細に説明します。

2.1.1 データベースエンジンへの接続

各データベースエンジンは、クライアントとの接続を確立するための独自の（おそらくは独自の）メカニズムを持っています。一方、クライアントは、可能な限りサーバーに依存しないことを望んでいます。つまり、クライアントは細かい詳細を知りたくないのです。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_2

Driver
パブリック接続接続(文字列 url、プロパティ prop)
SQLException をスローします。
Connection
public Statement createStatement() は SQLException をスローします。public void close() は SQLException をスローします。

Statement
public ResultSet executeQuery(String qry) は SQLException をスローします。public int executeUpdate(String cmd) は SQLException をスローします。public void close() は SQLException をスローします。

ResultSet
public boolean next() は SQLException をスローします。public int getInt() は SQLException をスローします。public String getString() は SQLException をスローします。public void close() は SQLException をスローします。public ResultSetMetaData getMetaData() は SQLException をスローします。

ResultSetMetaData
public int getColumnCount() は SQLException をスローします。public String getColumnName(int column) は SQLException をスローします。public int getColumnType(int column) は SQLException をスローします。public int getColumnDisplaySize(int column) は SQLException をスローします。

図2.1 基本的なJDBCのAPI

org.apache.derby.jdbc.ClientDriver をインポートします。public class CreateTestDB {public static void main(String[] args) {String url = "jdbc:derby://localhost/testdb;create=true"; Driver d **=** new ClientDriver(); try {Connection conn **=** d.connect(url, null); System.out.println("Database Created"); conn.close(); }catch(SQLException e) { e.printStackTrace(); } } import java.sql.Driver; import java.sql.Connection; }

図2.2 CreateTestDBクライアントのJDBCコード

エンジンに接続する方法の規定はありません。クライアントが呼び出すクラスをエンジンが提供するだけです。このようなクラスはドライバーと呼ばれます。

JDBC ドライバ クラスは Driver インターフェイスを実装します。Derby と SimpleDB にはそれぞれ 2 つのドライバ クラスがあります。1 つはサーバー ベースの接続用、もう 1 つは埋め込み接続用です。Derby エンジンへのサーバー ベースの接続では ClientDriver クラスが使用され、埋め込み接続では EmbeddedDriver が使用されます。どちらのクラスも org.apache.derby.jdbc パッケージにあります。SimpleDB エンジンへのサーバー ベースの接続では NetworkDriver クラス (simpledb.jdbc.network パッケージ内) が使用され、埋め込み接続では EmbeddedDriver (simpledb.jdbc.embedded パッケージ内) が使用されます。

クライアントは、Driver オブジェクトの connect メソッドを呼び出してデータベース エンジンに接続します。たとえば、図 2.2 の次の 3 行は、Derby データベースへのサーバーベースの接続を確立します。

文字列 url = "jdbc:derby://localhost/testdb;create=true"; ドライバー d = new Cl
ientDriver(); 接続 conn = d.connect(url, null);

connect メソッドは 2 つの引数を取ります。メソッドの最初の引数は、ドライバー、サーバー (サーバー ベースの接続の場合)、およびデータベースを識別する URL です。この URL は接続文字列と呼ばれ、第 1 章の ij (または SimpleIJ) サーバー ベースの接続文字列と同じ構文を持ちます。図 2.2 の接続文字列は、次の 4 つの部分で構成されています。

- サブ文字列「jdbc:derby:」は、クライアントが使用するプロトコルを表します。ここで、プロトコルは、このクライアントが JDBC を使用するDerby クライアントであることを示します。
- サブ文字列「//localhost」は、サーバーが配置されているマシンを表します。localhost の代わりに、任意のドメイン名または IP アドレスを代用できます。
- サブ文字列「/testdb」は、サーバー上のデータベースへのパスを表します。Derby サーバーの場合、パスはサーバーを起動したユーザーの現在のディレクトリから始まります。パスの末尾 (ここでは「testdb」) は、このデータベースのすべてのデータ ファイルが格納されるディレクトリです。
- 接続文字列の残りの部分は、エンジンに送信されるプロパティ値で構成されます。ここでは、サブ文字列は「;create ¼ true」であり、エンジンに新しいデータベースを作成するように指示します。一般に、複数のプロパティ値を Derby エンジンに送信できます。たとえば、エンジンがユーザー認証を必要とする場合は、ユーザー名とパスワードのプロパティの値も指定します。ユーザー「einstein」の接続文字列は次のようになります。

  「jdbc:derby://localhost/testdb;create=true;user=einstein; password=emc2」

connect の 2 番目の引数は、Properties 型のオブジェクトです。このオブジェクトは、エンジンにプロパティ値を渡す別の方法を提供します。図 2.2 では、すべてのプロパティが接続文字列で指定されているため、この引数の値は null です。または、次のように、プロパティの指定を 2 番目の引数に入れることもできます。

```
文字列 url = "jdbc:derby://localhost/testdb"; プロパティ prop
= new Properties(); prop.put("create", "true"); prop.put("userna
me", "einstein"); prop.put("password", "emc2"); ドライバー d
= new ClientDriver(); 接続 conn = d.connect(url, prop);
```

各データベース エンジンには、独自の接続文字列構文があります。SimpleDB のサーバー ベースの接続文字列は、プロトコルとマシン名のみを含むという点で Derby とは異なります (データベースは SimpleDB サーバーの起動時に指定されるため、文字列にデータベース名を含めるのは意味がありません。また、SimpleDB サーバーはプロパティをサポートしていないため、接続文字列ではプロパティを指定しません)。たとえば、次の 3 行のコードは、SimpleDB サーバーへの接続を確立します。

```
文字列 url = "jdbc:simpledb://localhost"; ドライバー d
= new NetworkDriver(); conn = d.connect(url, null);
```

ドライバ クラスと接続文字列の構文はベンダー固有ですが、JDBC プログラムの残りの部分は完全にベンダーに依存しません。たとえば、図 2.2 の変数 d と conn について考えてみましょう。これらに対応する JDBC 型である Driver と Connection はインターフェイスです。コードから、変数 d が ClientDriver オブジェクトに割り当てられていることがわかります。ただし、conn はメソッド connect によって返される Connection オブジェクトに割り当てられており、その実際のクラスを知る方法はありません。この状況はすべての JDBC プログラムに当てはまります。ドライバ クラスの名前とその接続文字列を除けば、JDBC プログラムはベンダーに依存しない JDBC インターフェイスのみを認識し、それを考慮します。したがって、基本的な JDBC クライアントは、次の 2 つのパッケージからインポートします。

- ベンダーに依存しないJDBCインターフェース定義を取得するための組み込みパッケージjava.sql
- ドライバクラスを含むベンダー提供のパッケージ

2.1.2 データベースエンジンからの切断

クライアントがデータベース エンジンに接続している間、エンジンはクライアントが使用できるリソースを割り当てることがあります。たとえば、クライアントは、他のクライアントがデータベースの一部にアクセスできないように、サーバーにロックを要求することがあります。エンジンに接続する機能もリソースになることがあります。企業が商用データベース システムのサイト ライセンスを所有していて、同時接続数が制限されている場合、接続を保持すると他のクライアントが接続できなくなる可能性があります。接続には貴重なリソースが保持されるため、データベースが不要になったらすぐにクライアントはエンジンから切断することが求められます。クライアントは、

プログラムは、Connectionオブジェクトのcloseメソッドを呼び出してエンジンから切断します。このcloseの呼び出しは、図2.2に示されています。

2.1.3 SQL例外

クライアントとデータベース エンジン間のやり取りでは、さまざまな理由で例外が発生する可能性があります。次に例を示します。

- クライアントは、不正な形式の SQL ステートメント、または存在しないテーブルにアクセスする SQL クエリ、あるいは互換性のない 2 つの値を比較する SQL クエリをエンジンに実行するように要求します。
- 同時クライアントとの間でデッドロックが発生したため、クライアントを中止します。
- エンジンコードにバグがあります。
- クライアントはエンジンにアクセスできません (サーバーベースの接続の場合)。ホスト名が間違っているか、ホストにアクセスできなくなっている可能性があります。

さまざまなデータベース エンジンには、これらの例外を処理する独自の内部方法があります。たとえば、SimpleDB は、ネットワークの問題では RemoteException をスローし、SQL ステートメントの問題では BadSyntaxException をスローし、デッドロックでは BufferAbortException または LockAbortException をスローし、サーバーの問題では汎用 RuntimeException をスローします。

例外処理をベンダーに依存しないようにするために、JDBC は SQLException と呼ばれる独自の例外クラスを提供します。データベース エンジンが内部例外に遭遇すると、それを SQL 例外にラップしてクライアント プログラムに送信します。

SQL 例外に関連付けられたメッセージ文字列は、例外の原因となった内部例外を識別します。各データベース エンジンは、独自のメッセージを自由に提供できます。たとえば、Derby には約 900 のエラー メッセージがありますが、SimpleDB では、考えられるすべての問題が「ネットワークの問題」、「不正な SQL ステートメント」、「サーバー エラー」、「サポートされていない操作」、および 2 種類の「トランザクションの中止」の 6 つのメッセージにまとめられています。

ほとんどの JDBC メソッド (および図 2.1 のすべてのメソッド) は、SQL 例外をスローします。SQL 例外はチェックされるため、クライアントは例外をキャッチするか、先にスローするかのいずれかによって、明示的に処理する必要があります。図 2.2 の 2 つの JDBC メソッドは try ブロック内で実行されます。どちらかが例外を引き起こした場合、コードはスタック トレースを出力して戻ります。

図 2.2 のコードには、例外がスローされたときに接続が閉じられないという問題があることに注意してください。これはリソース リークの例です。つまり、クライアントが停止した後、エンジンは接続のリソースを簡単に再利用できません。この問題を解決する 1 つの方法は、catch ブロック内で接続を閉じることです。ただし、close メソッドは try ブロック内から呼び出す必要があるため、図 2.2 の catch ブロックは実際には次のようになります。

```
catch(SQLException e) { e.printStack
Trace(); try { conn.close(); }catch(SQ
LException ex) {} }
```

これは見苦しくなってきました。さらに、close メソッドが例外をスローした場合、クライアントは何をすべきでしょうか? 上記のコードではそれを無視していますが、これはあまり適切ではないようです。

より良い解決策は、try-with-resources 構文を使用して、Java に自動的に接続を閉じるようにすることです。これを使用するには、try キーワードの後の括弧内に Connection オブジェクトを作成します。try ブロックが終了すると (通常または例外によって)、Java は暗黙的にオブジェクトの close メソッドを呼び出します。図 2.2 の改善された try ブロックは次のようになります。

```
try (接続 conn = d.connect(url, null)) { System.out.println("デ
ータベースが作成されました"); }catch (SQLException e) {
e.printStackTrace(); }
```

このコードは、図 2.2 のシンプルさを失うことなく、すべての例外を適切に処理します。

2.1.4 SQL文の実行

接続は、データベース エンジンとの「セッション」と考えることができます。このセッションでは、エンジンがクライアントの SQL ステートメントを実行します。JDBC は、この考え方を次のようにサポートします。

Connection オブジェクトには、Statement オブジェクトを返す createStatement メソッドがあります。Statement オブジェクトには、SQL 文を実行する 2 つの方法、executeQuery メソッドと executeUpdate メソッドがあります。また、オブジェクトが保持するリソースの割り当てを解除する close メソッドもあります。

図 2.3 は、executeUpdate を呼び出して Amy の STUDENT レコードの MajorId 値を変更するクライアント プログラムを示しています。メソッドの引数は、SQL 更新ステートメントを示す文字列です。メソッドは更新されたレコードの数を返します。

Connection オブジェクトと同様に、Statement オブジェクトも閉じる必要があります。最も簡単な解決策は、try ブロックで両方のオブジェクトを自動的に閉じることです。

SQLコマンドの仕様は興味深い点を示しています。コマンドはJava文字列として保存されるため、二重引用符で囲まれます。一方、SQLの文字列は一重引用符を使用します。この区別により、

パブリック クラス ChangeMajor { パブリック スタティック void main(String[] args) { 文字列 u
rl = "jdbc:derby://localhost/studentdb"; 文字列 cmd = "update STUDENT set MajorId=30 where SNa
me='amy'"; Driver d = new ClientDriver(); try (Connection conn = d.connect(url, null); Statement st
mt = conn.createStatement()) { int howmany = stmt.executeUpdate(cmd); System.out.println(howman
y + " レコードが変更されました。"); }catch(SQLException e) { e.printStackTrace(); } } }

図2.3 ChangeMajorクライアントのJDBCコード

引用符文字が 2 つの異なる意味を持つことを心配する必要はありません。SQL 文字列では一重引用符が使用され、Java 文字列では二重引用符が使用されます。

ChangeMajor コードは、「studentdb」というデータベースが存在することを前提としています。SimpleDB ディストリビューションには、データベースを作成し、図 1.1 のテーブルをデータベースに取り込む CreateStudentDB クラスが含まれています。これは、大学のデータベースを使用するときに最初に呼び出されるプログラムです。そのコードは図 2.4 に示されています。このコードは SQL 文を実行して 5 つのテーブルを作成し、そこにレコードを挿入します。簡潔にするために、STUDENT のコードのみを示しています。

2.1.5 結果セット

ステートメントの executeQuery メソッドは、SQL クエリを実行します。このメソッドの引数は、SQL クエリを示す文字列で、ResultSet 型のオブジェクトを返します。ResultSet オブジェクトは、クエリの出力レコードを表します。クライアントは結果セットを検索して、これらのレコードを調べることができます。

結果セットの使用方法を示すプログラム例として、図 2.5 に示す StudentMajor クラスを考えてみましょう。executeQuery の呼び出しにより、各学生の名前と専攻を含む結果セットが返されます。後続の while ループにより、結果セットの各レコードが出力されます。

クライアントが結果セットを取得すると、next メソッドを呼び出して出力レコードを反復処理します。このメソッドは次のレコードに移動し、移動が成功した場合は true を返し、レコードがない場合は false を返します。通常、クライアントはループを使用してすべてのレコードを移動し、各レコードを順番に処理します。

新しい ResultSet オブジェクトは常に最初のレコードの前に配置されるため、最初のレコードを確認する前に next を呼び出す必要があります。この要件のため、レコードをループする一般的な方法は次のようになります。

```
public class CreateStudentDB {
   public static void main(String[] args) {
      String url = "jdbc:derby://localhost/studentdb;create=true";
      Driver d = new ClientDriver();
      try (Connection conn = d.connect(url, null);
            Statement stmt = conn.createStatement()) {

         String s = "create table STUDENT(SId int,
                  SName varchar(10), MajorId int, GradYear int)";
         stmt.executeUpdate(s);
         System.out.println("Table STUDENT created.");

         s = "insert into STUDENT(SId, SName,
                                  MajorId, GradYear) values ";
         String[] studvals = {"(1, 'joe', 10, 2021)",
               "(2, 'amy', 20, 2020)",
               "(3, 'max', 10, 2022)",
               "(4, 'sue', 20, 2022)",
               "(5, 'bob', 30, 2020)",
               "(6, 'kim', 20, 2020)",
               "(7, 'art', 30, 2021)",
               "(8, 'pat', 20, 2019)",
               "(9, 'lee', 10, 2021)"};
         for (int i=0; i<studvals.length; i++)
            stmt.executeUpdate(s + studvals[i]);
         System.out.println("STUDENT records inserted.");

         ...

      }
      catch(SQLException e) {
         e.printStackTrace();
      }
   }
}
```

図2.4 CreateStudentDBクライアントのJDBCコード

String qry = "select ..."; ResultSet rs = stmt.executeQ
uery(qry); while (rs.next()) { ... // レコードを処理 }

このようなループの例を図 2.5 に示します。このループを n 回実行すると、変数 rs は結果セットの n 番目のレコードに配置されます。処理するレコードがなくなるとループは終了します。

レコードを処理する際、クライアントは getInt メソッドと getString メソッドを使用してフィールドの値を取得します。各メソッドはフィールド名を引数として受け取り、そのフィールドの値を返します。図 2.5 では、コードは各レコードのフィールド SName と DName の値を取得して出力しています。

```
パブリック クラス StudentMajor { public static void main(String[] args) { String url =
"jdbc:derby://localhost/studentdb"; String qry = "select SName, DName from DEPT, ST
UDENT " + "where MajorId = DId"; Driver d = new ClientDriver(); try ( Connection co
nn = d.connect(url, null); Statement stmt = conn.createStatement(); ResultSet rs = stmt.e
xecuteQuery(qry)) { System.out.println("Name\tMajor"); while (rs.next()) { String snam
e = rs.getString("SName"); String dname = rs.getString("DName"); System.out.println(s
name + "\t" + dname); } }catch(SQLException e) { e.printStackTrace(); } } }
```

図2.5 StudentMajorクライアントのJDBCコード

結果セットはエンジン上の貴重なリソースを占有します。close メソッドはこれらのリソースを解放し、他のクライアントが使用できるようにします。したがって、クライアントは「良き市民」となるよう努め、できるだけ早く結果セットを閉じる必要があります。1 つのオプションは、通常は上記の while ループの最後に、明示的に close を呼び出すことです。図 2.5 で使用されているもう 1 つのオプションは、Java の自動クローズ メカニズムを使用することです。

2.1.6 クエリメタデータの使用

結果セットのスキーマは、各フィールドの名前、タイプ、および表示サイズとして定義されます。この情報は、ResultSetMetaData インターフェイスを通じて利用可能になります。

クライアントがクエリを実行する場合、通常は出力テーブルのスキーマが認識されます。たとえば、StudentMajor クライアントには、結果セットに 2 つの文字列フィールド SName と DName が含まれているという情報がハードコードされています。ただし、クライアントプログラムがユーザーにクエリを入力として送信することを許可しているとします。プログラムはクエリの結果セットで getMetaData メソッドを呼び出すことができ、このメソッドは ResultSetMetaData 型のオブジェクトを返します。次に、このオブジェクトのメソッドを呼び出して、出力テーブルのスキーマを決定できます。たとえば、図 2.6 のコードでは、ResultSetMetaData を使用して引数の結果セットのスキーマを出力します。

```
void printSchema(ResultSet rs) throws SQLException { ResultSetMetaData md = rs.ge
tMetaData(); for(int i=1; i<=md.getColumnCount(); i++) { String name = md.getColu
mnName(i); int size = md.getColumnDisplaySize(i); int typecode = md.getColumnTyp
e(i); String type; if (typecode == Types.INTEGER) type = "int"; else if (typecode == T
ypes.VARCHAR) type = "string"; elsetype = "other"; System.out.println(name + "\t" +
type + "\t" + size); } }
```

図 2.6 ResultSetMetaData を使用して結果セットのスキーマを出力する

このコードは、ResultSetMetaData オブジェクトの一般的な使用方法を示しています。最初に getColumnCount メソッドを呼び出して、結果セット内のフィールドの数を返します。次に、getColumnName、getColumnType、および getColumnDisplaySize メソッドを呼び出して、各列のフィールドの名前、タイプ、およびサイズを決定します。列番号は、予想される 0 ではなく、1 から始まることに注意してください。

getColumnType メソッドは、フィールド タイプをエンコードする整数を返します。これらのコードは、JDBC クラス Types で定数として定義されています。このクラスには 30 種類の異なるタイプのコードが含まれており、SQL 言語がいかに広範囲にわたるかがわかります。これらのタイプの実際の値は重要ではありません。JDBC プログラムは常に、値ではなく名前でコードを参照する必要があるためです。

メタデータの知識を必要とするクライアントの良い例は、コマンド インタープリタです。第 1 章のプログラム SimpleIJ はそのようなプログラムです。そのコードは図 2.7 に示されています。これは、重要な JDBC クライアントの最初の例であるため、そのコードを詳しく調べる必要があります。

メイン メソッドは、まずユーザーから接続文字列を読み取り、それを使用して適切なドライバーを決定します。コードは接続文字列内の文字「//」を探します。これらの文字が表示された場合、文字列はサーバーベースの接続を指定している必要があり、表示されていない場合は埋め込み接続を指定します。次に、メソッドは接続文字列を適切なドライバーの接続メソッドに渡すことで接続を確立します。

main メソッドは、while ループの各反復中に 1 行のテキストを処理します。テキストが SQL ステートメントの場合は、必要に応じて doQuery メソッドまたは doUpdate メソッドが呼び出されます。ユーザーは「exit」と入力してループを終了でき、その時点でプログラムは終了します。

```
public class SimpleIJ { public static void main(String[] args) { Scanner sc = new Scann
er(System.in); System.out.println("Connect> "); String s = sc.nextLine(); Driver d = (s.
contains("//")) ? new NetworkDriver() : new EmbeddedDriver(); try (Connection conn
= d.connect(s, null); Statement stmt = conn.createStatement()) { System.out.print("\nS
QL> "); while (sc.hasNextLine()) { // 入力の 1 行を処理します String cmd = sc.next
Line().trim(); if (cmd.startsWith("exit")) break; else if (cmd.startsWith("select")) doQu
ery(stmt, cmd); elsedoUpdate(stmt, cmd); System.out.print("\nSQL> "); } }catch (SQL
Exception e) { e.printStackTrace(); }sc.close(); }
```

```
private static void doQuery(Statement stmt, String cmd) { try (ResultSet rs = stmt.ex
ecuteQuery(cmd)) { ResultSetMetaData md = rs.getMetaData(); int numcols = md.g
etColumnCount(); int totalwidth = 0; // ヘッダーを印刷 for(int i=1; i<=numcols; i
++) { String fldname = md.getColumnName(i); int width = md.getColumnDisplaySi
ze(i); totalwidth += width; String fmt = "%" + width + "s"; System.out.format(fmt, f
ldname); }
```

図2.7 SimpleIJクライアントのJDBCコード

```
System.out.println(); for(int i=0; i<totalwidth;
i++) System.out.print("-"); System.out.println
();


// レコードを印刷します while(rs.next()) { for (int i=1; i<=numcols; i++) { Stri
ng fldname = md.getColumnName(i); int fldtype = md.getColumnType(i); String fmt = "
%" + md.getColumnDisplaySize(i); if (fldtype == Types.INTEGER) { int ival = rs.getInt(
fldname); System.out.format(fmt + "d", ival); }else { String sval = rs.getString(fldname);
System.out.format(fmt + "s", sval); } }System.out.println(); } }catch (SQLException e) {
System.out.println("SQL 例外: " + e.getMessage()); } }private static void doUpdate(State
ment stmt, String cmd) { try {int howmany = stmt.executeUpdate(cmd); System.out.printl
n(howmany + " レコードが処理されました"); }catch (SQLException e) { System.out.p
rintln("SQL 例外: " + e.getMessage()); } }

}
```

図2.7（続き）

メソッド doQuery はクエリを実行し、出力テーブルの結果セットとメタデータを取得します。メソッドの大部分は、値の適切な間隔を決定することに関係しています。getColumnDisplaySize の呼び出しは、各フィールドのスペース要件を返します。コードはこれらの数値を使用して、フィールド値を適切に整列させる書式設定文字列を作成します。このコードの複雑さは、「悪魔は細部に宿る」という格言を物語っています。つまり、概念的に難しいタスクは、ResultSet メソッドと ResultSetMetaData メソッドのおかげで簡単にコーディングできますが、データを整列させるという些細なタスクにコーディング作業のほとんどが費やされます。

doQuery メソッドと doUpdate メソッドは、エラー メッセージを出力して戻ることで例外をトラップします。このエラー処理戦略により、ユーザーが「exit」コマンドを入力するまで、メイン ループはステートメントを受け入れ続けることができます。

2.2 高度なJDBC

基本的な JDBC は比較的簡単に使用できますが、データベース エンジンと対話する方法がかなり限られています。このセクションでは、データベースへのアクセス方法をクライアントがより細かく制御できるようにする、JDBC の追加機能について説明します。

2.2.1 ドライバーの非表示

基本的な JDBC では、クライアントは Driver オブジェクトのインスタンスを取得し、その connect メソッドを呼び出すことによってデータベース エンジンに接続します。この戦略の問題点は、ベンダー固有のコードがクライアント プログラムに配置されることです。JDBC には、ドライバー情報をクライアント プログラムから切り離すための、ベンダーに依存しない 2 つのクラス (DriverManager と DataSource) が含まれています。それぞれについて順に見ていきましょう。

DriverManager の使用

DriverManager クラスは、ドライバーのコレクションを保持します。このクラスには、コレクションにドライバーを追加したり、コレクション内で特定の接続文字列を処理できるドライバーを検索したりするための静的メソッドが含まれています。これらのメソッドのうち 2 つを図 2.8 に示します。考え方としては、クライアントが registerDriver を繰り返し呼び出して、使用する可能性のある各データベースのドライバーを登録します。クライアントがデータベースに接続するときは、getConnection メソッドを呼び出して接続文字列を提供するだけです。ドライバー マネージャーは、コレクション内の各ドライバーに対して接続文字列を試行し、いずれかのドライバーが null 以外の接続を返すまで試行します。

たとえば、図 2.9 のコードについて考えてみましょう。最初の 2 行は、サーバーベースの Derby および SimpleDB ドライバーをドライバー マネージャーに登録します。最後の 2 行は、Derby サーバーへの接続を確立します。クライアントは、getConnection を呼び出すときにドライバーを指定する必要はなく、接続文字列のみを指定します。ドライバー マネージャーは、登録されているドライバーのどれを使用するかを決定します。

```
static public void registerDriver(Driver driver)
                                  throws SQLException;
static public Connection getConnection(String url, Properties p)
                                  throws SQLException;
```

図2.8 DriverManagerクラスの2つのメソッド

```
DriverManager.registerDriver(new ClientDriver());
DriverManager.registerDriver(new NetworkDriver());
String url   = "jdbc:derby://localhost/studentdb";
Connection c = DriverManager.getConnection(url);
```

図2.9 DriverManagerを使用してDerbyサーバーに接続する

図 2.9 の DriverManager の使用は、ドライバ情報が隠されておらず、registerDriver の呼び出しの中にあるため、特に満足できるものではありません。JDBC では、ドライバを Java システム プロパティ ファイルで指定できるようにすることで、この問題を解決しています。たとえば、Derby および SimpleDB ドライバは、ファイルに次の行を追加することで登録できます。

jdbc.drivers= org.apache.derby.jdbc.ClientDriver:simpledb.remote.NetworkDriver

ドライバ情報をプロパティ ファイルに配置することは、クライアント コードからドライバ仕様を削除する優れた方法です。この 1 つのファイルを変更するだけで、コードを再コンパイルすることなく、すべての JDBC クライアントで使用されるドライバ情報を変更できます。

データソースの使用

ドライバー マネージャーは、JDBC クライアントからドライバーを隠すことができますが、接続文字列を隠すことはできません。特に、上記の例の接続文字列には「jdbc:derby」が含まれているため、どのドライバーが対象であるかは明らかです。JDBC に最近追加されたのは、パッケージ javax.sql のインターフェイス DataSource です。これは現在、ドライバーを管理するために推奨される戦略です。

DataSource オブジェクトはドライバーと接続文字列の両方をカプセル化するため、クライアントは接続の詳細を知らなくてもエンジンに接続できます。Derby でデータ ソースを作成するには、Derby が提供するクラス ClientDataSource (サーバー ベースの接続用) と EmbeddedDataSource (埋め込み接続用) が必要です。どちらも DataSource を実装しています。クライアント コードは次のようになります。

ClientDataSource ds = new ClientDataSource(); ds.setServerName("localhost"); ds.setDatabaseName("studentdb"); 接続 conn = ds.getConnection();

各データベース ベンダーは、DataSource を実装する独自のクラスを提供しています。これらのクラスはベンダー固有であるため、ドライバー名や接続文字列の構文など、ドライバーの詳細をカプセル化できます。これらのクラスを使用するプログラムでは、必要な値を指定するだけで済みます。 データ ソースを使用する利点は、クライアントがドライバーの名前や接続文字列の構文を知る必要がなくなることです。ただし、クラスは依然としてベンダー固有であるため、クライアント コードはベンダーから完全に独立しているわけではありません。この問題は、さまざまな方法で対処できます。

1 つの解決策は、データベース管理者がデータソースオブジェクトをファイルに保存することです。DBAはオブジェクトを作成し、Javaシリアル化を使用してファイルに書き込むことができます。クライアントは、ファイルを読み取り、デシリアル化してデータソースを取得できます。

DataSource オブジェクトに。このソリューションは、プロパティ ファイルの使用に似ています。DataSource オブジェクトをファイルに保存すると、どの JDBC クライアントでも使用できるようになります。また、DBA は、そのファイルの内容を置き換えるだけで、データ ソースを変更できます。

2 番目の解決策は、ファイルの代わりにネーム サーバー (JNDI サーバーなど) を使用することです。DBA は DataSource オブジェクトをネーム サーバーに配置し、クライアントはサーバーからデータ ソースを要求します。ネーム サーバーは多くのコンピューティング環境で共通であるため、この解決策は簡単に実装できますが、詳細は本書の範囲外です。

2.2.2 明示的なトランザクション処理

各 JDBC クライアントは、一連のトランザクションとして実行されます。概念的には、トランザクションは「作業単位」であり、すべてのデータベース インタラクションが 1 つの単位として扱われることを意味します。たとえば、トランザクション内の 1 つの更新が失敗すると、エンジンは、そのトランザクションによって行われたすべての更新が失敗することを保証します。

トランザクションは、現在の作業単位が正常に完了するとコミットされます。データベース エンジンは、すべての変更を永続的にし、そのトランザクションに割り当てられたリソース (ロックなど) を解放することでコミットを実装します。コミットが完了すると、エンジンは新しいトランザクションを開始します。

トランザクションはコミットできない場合にロールバックします。データベース エンジンは、そのトランザクションによって行われたすべての変更を元に戻し、ロックを解除し、新しいトランザクションを開始することでロールバックを実装します。コミットまたはロールバックされたトランザクションは完了したとみなされます。

トランザクションは、基本的な JDBC では暗黙的に行われます。データベース エンジンは各トランザクションの境界を選択し、トランザクションをいつコミットするか、またロールバックするかどうかを決定します。この状況は自動コミットと呼ばれます。

自動コミット中、エンジンは各 SQL 文を独自のトランザクションで実行します。文が正常に完了するとエンジンはトランザクションをコミットし、それ以外の場合はトランザクションをロールバックします。更新コマンドは、executeUpdate メソッドが終了するとすぐに完了し、クエリの結果セットが閉じられるとクエリが完了します。

トランザクションはロックを蓄積しますが、トランザクションがコミットまたはロールバックされるまでロックは解放されません。これらのロックにより他のトランザクションが待機する可能性があるため、トランザクションが短いほど同時実行性が向上します。この原則は、自動コミット モードで実行されているクライアントは、できるだけ早く結果セットを閉じる必要があることを意味します。

自動コミットは、JDBC クライアントにとって適切なデフォルト モードです。SQL ステートメントごとに 1 つのトランザクションを持つことでトランザクションが短くなり、多くの場合はこれが適切な方法です。ただし、トランザクションが複数の SQL ステートメントで構成されている必要がある場合もあります。

自動コミットが望ましくない状況の1つは、クライアントが2つのステートメントを同時にアクティブにする必要がある場合です。たとえば、図2.10のコードフラグメントを考えてみましょう。このコードは最初にすべてのコースを取得するクエリを実行します。次にループします。

```
データソース ds = ... 接続 conn = ds.getConnection(); ステートメント stmt1 = conn.createSt
atement(); ステートメント stmt2 = conn.createStatement(); 結果セット rs = stmt1.executeQu
ery("select * from COURSE"); while (rs.next()) { String title = rs.getString("Title"); boolean go
odCourse = getUserDecision(title); if (!goodCourse) { int id = rs.getInt("CId"); stmt2.executeUp
date("delete from COURSE where CId =" + id); } }rs.close();
```

図 2.10 自動コミットモードで不正な動作をする可能性のあるコード

```
DataSource ds = ... 接続 conn = ds.getConnection(); Statement stmt = conn.createStatement(); String c
md1 = "update SECTION set Prof= 'brando' where SectId = 43"; String cmd2 = "update SECTION set
Prof= 'einstein' where SectId = 53"; stmt.executeUpdate(cmd1); // この時点でエンジンがクラッシュ
すると仮定します stmt.executeUpdate(cmd2);
```

図 2.11 自動コミットモードで不正な動作をする可能性のあるその他のコード

結果セットを調べ、各コースを削除するかどうかをユーザーに尋ねます。削除する場合は、SQL 削除ステートメントを実行して削除します。

このコードの問題は、レコード セットがまだ開いている間に削除ステートメントが実行されることです。接続では一度に 1 つのトランザクションしかサポートされないため、削除を実行するための新しいトランザクションを作成する前に、クエリのトランザクションを事前にコミットする必要があります。また、クエリのトランザクションがコミットされているため、レコード セットの残りの部分にアクセスしても意味がありません。コードは例外をスローするか、予期しない動作をします。[1]

自動コミットは、データベースへの複数の変更を同時に行う必要がある場合にも望ましくありません。図 2.11 のコード フラグメントは、その例を示しています。コードの目的は、教授の講義セクション 43 と 53 を交換することです。ただし、executeUpdate の最初の呼び出し後、2 回目の呼び出し前にエンジンがクラッシュすると、データベースは不正確になります。このコードには、SQL ステートメントと 2 つの SQL ステートメントの両方が必要です。

[1] The actual behavior of this code depends on the *holdability* of the result set, whose default value is engine dependent. If the holdability is `CLOSE_CURSORS_AT_COMMIT`, then the result set will become invalid, and an exception will be thrown. If the holdability is `HOLD_CURSORS_OVER_COMMIT`, then the result set will stay open, but its locks will be released. The behavior of such a result set is unpredictable and similar to the read-uncommitted isolation mode to be discussed in Sect. 2.2.3.

```
public void setAutoCommit(boolean ac) throws SQLException;
public void commit()                  throws SQLException;
public void rollback()                throws SQLException;
```

図2.12 明示的なトランザクション処理のための接続方法

同じトランザクション内で発生するようにし、一緒にコミットまたはロールバックされるようにします。

自動コミット モードも不便な場合があります。プログラムが、テキスト ファイルからデータをロードするなど、複数の挿入を実行しているとします。プログラムの実行中にエンジンがクラッシュすると、一部のレコードは挿入されますが、一部のレコードは挿入されません。プログラムが失敗した場所を特定し、不足しているレコードのみを挿入するようにプログラムを書き直すのは、面倒で時間のかかる作業です。よりよい代替策は、すべての挿入コマンドを同じトランザクションに配置することです。そうすれば、システム クラッシュ後にすべてのコマンドがロールバックされ、クライアントを再実行するだけで済みます。

Connection インターフェイスには、クライアントがトランザクションを明示的に処理できるようにする 3 つのメソッドが含まれています。図 2.12 にその API を示します。クライアントは、setAutoCommit(false) を呼び出して自動コミットをオフにします。クライアントは、必要に応じて commit または rollback を呼び出して現在のトランザクションを完了し、新しいトランザクションを開始します。

クライアントが自動コミットをオフにすると、失敗した SQL ステートメントをロールバックする責任がクライアント側にあります。特に、トランザクション中に例外がスローされた場合、クライアントは例外処理コード内でそのトランザクションをロールバックする必要があります。

例として、図 2.10 の誤ったコード フラグメントをもう一度考えてみましょう。修正されたバージョンは図 2.13 に示されています。このコードは、接続が確立された直後に setAutoCommit を呼び出し、ステートメントが完了した直後に commit を呼び出します。catch ブロックには、rollback の呼び出しが含まれています。この呼び出しは、例外をスローする場合に備えて、独自の try ブロック内に配置する必要があります。

一見すると、ロールバック中の例外は、図 2.11 に示すように、データベースを破損する可能性があるように見えます。幸い、データベース ロールバック アルゴリズムは、そのような可能性に対処できるように設計されています。第 5 章に注目すべき詳細が記載されています。したがって、図 2.13 のコードは、データベース エンジンが状況を正しく修正することを承知しているため、失敗したロールバックを正当に無視できます。

2.2.3 トランザクション分離レベル

通常、データベース サーバーには同時に複数のクライアントがアクティブになっており、それぞれが独自のトランザクションを実行しています。これらのトランザクションを同時に実行することで、サーバーのスループットと応答時間が向上します。したがって、同時実行は良いことです。ただし、同時実行が制御されていない場合は、問題が発生する可能性があります。これは、トランザクションが別のトランザクションによって使用されるデータを予期しない方法で変更することで、別のトランザクションに干渉する可能性があるためです。次に、発生する可能性のある問題の種類を示す 3 つの例を示します。

```
DataSource ds = ... try (Connection conn = ds.getConnection()) { conn.setAutoCommit(false); State
ment stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("select * from COURSE"); w
hile (rs.next()) { String title = rs.getString("Title"); boolean goodCourse = getUserDecision(title); if
(!goodCourse) { int id = rs.getInt("CId"); stmt.executeUpdate("delete from COURSE where CId ="
+ id); } }rs.close(); stmt.close(); conn.commit(); }catch (SQLException e) { e.printStackTrace();試
してください { if (conn != null) conn.rollback(); }catch (SQLException e2) {} }
```

図2.13 トランザクションを明示的に処理する図2.10の改訂版

例1: コミットされていないデータの読み取り
2 つのセクションの教授を交換する図 2.11 のコードをもう一度考えてみます。このコードは単一のトランザクション (つまり、自動コミットはオフ) として実行されるものとします。このトランザクションを T1 と呼びます。また、大学では、教授に、教えたセクションの数に基づいてボーナスを支給することにしたとします。そのため、各教授が教えたセクションを数えるトランザクション T2 を実行します。さらに、これら 2 つのトランザクションが同時に実行されるものとします。具体的には、T2 が T1 の最初の更新ステートメントの直後に開始され、完了するまで実行されるものとします。その結果、Brando 教授と Einstein 教授は、それぞれ本来よりも 1 つ多く、1 つ少ないコースの履修単位を取得することになり、ボーナスに影響します。

何が悪かったのでしょうか。各トランザクションは個別には正しいのですが、それらが組み合わさると、大学は間違ったボーナスを支給することになります。問題は、T2 が、読み取ったレコードが一貫している、つまり、それらが一緒に意味を成すと誤って想定したことです。ただし、コミットされていないトランザクションによって書き込まれたデータは、常に一貫しているとは限りません。T1 の場合、2 つの変更のうちの 1 つだけが行われた時点で不整合が発生しました。その時点で T2 がコミットされていない変更されたレコードを読み取ったとき、不整合が原因で誤った計算が行われました。

例2: 現行のコードへの予期しない変更
この例では、STUDENT テーブルに、学生がカフェテリアで食べ物を買うために持っているお金の額を示す MealPlanBal フィールドが含まれていると仮定します。

データソース ds = ... 接続 conn = ds.getConnection(); conn.setAutoCommit(false); ステートメント stmt = conn.createStatement(); 結果セット rs = stmt.executeQuery("select MealPlanBal from STUDENT " + "where SId = 1");

rs.next();
int 残高 = rs.getInt("MealPlanBal"); rs.close();

int newbalance = balance 10; if (newbalance < 0) throw new NoFoodAllowedException("この食事を買う余裕はありません");

```
stmt.executeUpdate("学生を更新"
                   + "set MealPlanBal = " + newbalance
                   + " where SId = 1");
```

conn.commit();

(a)

データソース ds = ... 接続 conn = ds.getConnection(); conn.setAutoCommit(false); ステートメント stmt = conn.createStatement(); stmt.executeUpdate("update STUDENT " + "set MealPlanBal = MealPlanBal + 1000 " + "where SId = 1"); conn.commit(); (b)

図 2.14 更新を「失う」可能性のある 2 つの同時トランザクション。(a) トランザクション T1 は食事プランの残高を減らし、(b) トランザクション T2 は食事プランの残高を増やします。

図 2.14 の 2 つのトランザクションについて考えてみましょう。トランザクション T1 は、ジョーが 10 ドルのランチを購入したときに実行されました。トランザクションは、ジョーの現在の残高を調べるクエリを実行し、残高が十分であることを確認し、残高を適切に減らします。トランザクション T2 は、ジョーの両親がジョーの食事プラン残高に追加するために 1000 ドルの小切手を送ったときに実行されました。このトランザクションは、単に SQL 更新ステートメントを実行してジョーの残高を増やします。

ここで、ジョーの残高が 50 ドルのときに、この 2 つのトランザクションが同時に実行されたとします。特に、T1 が rs.close を呼び出した直後に T2 が開始され、完了したとします。最初にコミットする T2 は、残高を 1050 ドルに変更します。しかし、T1 はこの変更に気付かず、残高が 50 ドルのままであると考えています。そのため、残高を 40 ドルに変更してコミットします。その結果、1000 ドルの入金はジョーの残高に入金されず、更新が「失われ」ます。

ここでの問題は、トランザクション T1 が、食事プランの残高の値は、T1 が値を読み取った時点と T1 が値を変更した時点の間で変化しないと誤って想定していることです。正式には、この想定は繰り返し読み取りと呼ばれます。これは、トランザクションがデータベースから項目を繰り返し読み取ると常に同じ値が返されると想定しているためです。

例3: レコード数の予期しない変更
大学の食堂サービスが昨年 10 万ドルの利益を上げたとします。大学は学生に過剰請求したことを後悔し、利益を学生間で均等に分配することにしました。つまり、現在 1,000 人の学生がいる場合、大学は各学生の食事プラン残高に 100 ドルを追加します。コードは図 2.15 に示されています。

このトランザクションの問題は、現在の学生の数が、払い戻し額の計算から学生レコードの更新までの間に変化しないと想定していることです。しかし、レコード セットが閉じられてから更新ステートメントが実行されるまでの間に、いくつかの新しい学生レコードがデータベースに挿入されたとします。これらの新しいレコードには、誤って計算済みの払い戻し額が加算され、大学は払い戻しに 10 万ドル以上を費やすことになります。これらの新しいレコードは、トランザクションの開始後に不可解な形で現れるため、ファントム レコードと呼ばれます。

これらの例は、2 つのトランザクションが相互作用するときに発生する可能性のある問題の種類を示しています。任意のトランザクションに問題が発生しないことを保証する唯一の方法は、他のトランザクションから完全に分離して実行することです。この形式の分離は直列化可能性と呼ばれ、第 5 章で詳細に説明されています。

残念ながら、シリアル化可能なトランザクションは、データベース エンジンが許容する同時実行性を大幅に削減する必要があるため、実行速度が非常に遅くなる可能性があります。そのため、JDBC では 4 つの分離レベルを定義し、クライアントがトランザクションに必要な分離レベルを指定できるようにしています。

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String qry = "select count(SId) as HowMany from STUDENT "
           + "where GradYear >= extract(year, current_date)"
ResultSet rs = stmt.executeQuery(qry);
rs.next();
int count = rs.getInt("HowMany");
rs.close();

int rebate = 100000 / count;
String cmd = "update STUDENT "
           + "set MealPlanBalance = MealPlanBalance + " + rebate
           + " where GradYear >= extract(year, current_date)";
stmt.executeUpdate(cmd);
conn.commit();
```

図2.15 予想よりも多くのリベートを付与する可能性のある取引

- Read-Uncommitted 分離は、分離がまったくないことを意味します。このようなトランザクションは、上記の 3 つの例のいずれかの問題に悩まされる可能性があります。
- Read-Committed 分離では、トランザクションがコミットされていない値にアクセスすることを禁止します。反復不可能な読み取りやファントムに関連する問題は依然として発生する可能性があります。
- Repeatable-Read 分離は、読み取りコミットを拡張して、読み取りが常に繰り返し可能になるようにします。発生する可能性のある唯一の問題はファントムです。
- シリアル化可能な分離です。問題が発生しないことが保証されます。

JDBC クライアントは、Connection メソッド setTransactionIsolation を呼び出して、必要な分離レベルを指定します。たとえば、次のコード フラグメントは、分離レベルを serializable に設定します。

```
データソース ds = ... 接続 conn = ds.getConnection(); conn.setAutoCommit(fals
e); conn.setTransactionIsolation( Connection.TRANSACTION_SERIALIZABLE)
;
```

これら 4 つの分離レベルは、実行速度と潜在的な問題の間でトレードオフの関係にあります。つまり、トランザクションの実行速度を速くしたいほど、トランザクションが誤って実行されるリスクが高くなります。このリスクは、クライアントを慎重に分析することで軽減できます。

たとえば、ファントム読み取りと非反復読み取りは問題にならないと確信できるかもしれません。これは、たとえば、トランザクションが挿入のみを実行する場合や、既存の特定のレコードを削除する場合 (「delete from STUDENT where SId ¼ 1」など) に当てはまります。この場合、分離レベル read-committed は高速かつ正確です。

別の例として、潜在的な問題は重要ではないと自分に言い聞かせる場合があります。トランザクションが各年について、その年に与えられた平均成績を計算するとします。トランザクションの実行中に成績が変わることはあっても、それらの変更が結果の統計に重大な影響を与える可能性は低いと判断します。この場合、分離レベルとして read-committed または read-uncommitted を選択するのが妥当です。

多くのデータベース サーバー (Derby、Oracle、Sybase など) のデフォルトの分離レベルは、読み取りコミットです。このレベルは、自動コミット モードで初心者ユーザーが実行する単純なクエリに適しています。ただし、クライアント プログラムが重要なタスクを実行する場合は、最も適切な分離レベルを慎重に決定することが同様に重要です。自動コミット モードをオフにするプログラマーは、各トランザクションの適切な分離レベルを慎重に選択する必要があります。

2.2.4 準備されたステートメント

多くの JDBC クライアント プログラムは、ユーザーから引数値を受け取り、その引数に基づいて SQL 文を実行するという意味で、パラメータ化されています。このようなクライアントの例として、デモ クライアント Find Majors があります。そのコードは図 2.16 に示されています。

```
   public class FindMajors {
      public static void main(String[] args) {
      System.out.print("Enter a department name: ");
      Scanner sc = new Scanner(System.in);
      String major = sc.next();
      sc.close();
      String qry = "select sname, gradyear from student, dept "
           + "where did = majorid and dname = '" + major + "'";

      ClientDataSource ds = new ClientDataSource();
      ds.setServerName("localhost");
      ds.setDatabaseName("studentdb");
      try ( Connection conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(qry)) {

         System.out.println("Here are the " + major + " majors");
         System.out.println("Name\tGradYear");
         while (rs.next()) {
            String sname = rs.getString("sname");
            int gradyear = rs.getInt("gradyear");
            System.out.println(sname + "\t" + gradyear);
         }
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

図 2.16 FindMajors クライアントの JDBC コード

このクライアントは、まずユーザーに部門名を尋ねます。次に、この名前を、実行する SQL クエリに組み込みます。たとえば、ユーザーが「math」という値を入力したとします。生成される SQL クエリは次のようになります。

```
STUDENT、DEPT から SName、GradYear を選択し
ます。DId = MajorId、DName = 'math'
```

コードがクエリを生成するときに、学部名を囲む単一引用符を明示的に追加する方法に注意してください。クライアントは、このように動的に SQL ステートメントを生成する代わりに、パラメータ化された SQL ステートメントを使用できます。パラメータ化されたステートメントは、欠落しているパラメータ値を「?」文字で示す SQL ステートメントです。ステートメントには複数のパラメータを含めることができ、すべて「?」で示されます。各パラメータには、文字列内の位置に対応するインデックス値があります。たとえば、次のパラメータ化されたステートメントは、卒業年度と専攻がまだ指定されていないすべての学生を削除します。GradYear の値にはインデックス 1 が割り当てられ、MajorId の値にはインデックス 2 が割り当てられます。

```
public class PreparedFindMajors { public static void main(String[] args) { System.out.print("学
部名を入力してください: "); Scanner sc = new Scanner(System.in); String major = sc.next();
sc.close(); String qry = "select sname, gradyear from student, dept " + "where did = majorid and
dname = ?"; ClientDataSource ds = new ClientDataSource(); ds.setServerName("localhost"); ds.
setDatabaseName("studentdb"); try ( Connection conn = ds.getConnection(); PreparedStatement
pstmt = conn.prepareStatement(qry)) { pstmt.setString(1, major); ResultSet rs = pstmt.executeQ
uery(); System.out.println("ここに " + 専攻 + " 専攻があります"); System.out.println("Name\
tGradYear"); while (rs.next()) { String sname = rs.getString("sname"); int gradyear = rs.getInt("
gradyear"); System.out.println(sname + "\t" + gradyear); }rs.close(); }catch(Exception e) { e.pri
ntStackTrace(); } } }
```

図 2.17 準備されたステートメントを使用するように FindMajors クライアントを修正する

GradYear = ? および MajorId = ? の STUDENT から削除します。

JDBC クラス PreparedStatement は、パラメータ化されたステートメントを処理します。クライアントは、準備されたステートメントを次の 3 つの手順で処理します。

- 指定されたパラメータ化された SQL ステートメントの PreparedStatement オブジェクトを作成します。
- パラメータに値を割り当てます。
- 準備されたステートメントを実行します。

たとえば、図 2.17 では、FindMajors クライアントが準備済みステートメントを使用するように変更されています。変更は太字で示されています。太字の最後の 3 つのステートメントは、上記の 3 つの箇条書きに対応しています。最初に、クライアントは、メソッド prepareStatement を呼び出して、パラメータ化された SQL ステートメントを引数として渡すことにより、PreparedStatement オブジェクトを作成します。次に、クライアントは setString メソッドを呼び出して、最初の (唯一の) パラメータに値を割り当てます。最後に、メソッドは executeQuery を呼び出してステートメントを実行します。

public ResultSet executeQuery() は SQLException をスローします。public int executeUpdate () は SQLException をスローします。public void setInt(int index, int val) は SQLException をスローします。public void setString(int index, String val) は SQLException をスローします。

図 2.18 PreparedStatement の API の一部

```
// クエリを準備します String qry = "select SName, GradYear from STUDENT, D
EPT " + "where DId = MajorId and DName = ?"; PreparedStatement pstmt = conn.p
repareStatement(qry); // パラメータを繰り返し取得し、クエリを実行します St
ring major = getUserInput(); while (major != null) { pstmt.setString(1, major); Resul
tSet rs = pstmt.executeQuery(); displayResultSet(rs); major = getUserInput(); }
```

図 2.19 ループ内での準備済みステートメントの使用

図 2.18 は、最も一般的な PreparedStatement メソッドの API を示しています。executeQuery メソッドと executeUpdate メソッドは、Statement の対応するメソッドに似ていますが、引数を必要としない点が異なります。setInt メソッドと setString メソッドは、パラメータに値を割り当てます。図 2.17 では、setString の呼び出しによって、部門名が最初のインデックス パラメータに割り当てられています。setString メソッドは、値の前後に一重引用符を自動的に挿入するため、クライアント側で挿入する必要がないことに注意してください。

ほとんどの人は、SQL 文を明示的に作成するよりも、準備された文を使用する方が便利だと感じています。また、図 2.19 に示すように、準備された文は、文がループ内で生成される場合にも効率的なオプションです。その理由は、データベース エンジンが、パラメータ値を知らなくても準備された文をコンパイルできるためです。文を一度コンパイルすると、その後は再コンパイルすることなく、ループ内で繰り返し実行されます。

2.2.5 スクロール可能かつ更新可能な結果セット

基本的な JDBC の結果セットは前方のみで更新できません。完全な JDBC では、結果セットをスクロールおよび更新可能にすることもできます。クライアントは、このような結果セットを任意のレコードに配置したり、現在のレコードを更新したり、新しいレコードを挿入したりできます。図 2.20 は、これらの追加メソッドの API を示しています。

beforeFirstメソッドは結果セットを最初のレコードの前に配置し、afterLastメソッドは結果セットを最後のレコードの後に配置します。absoluteメソッドは結果セットを指定されたレコードの正確な位置に配置し、falseを返します。

Methods used by scrollable result sets

public void beforeFirst() は SQLException をスローします。public void afterLast() は SQLException をスローします。public boolean previous() は SQLException をスローします。public boolean next() は SQLException をスローします。public boolean absolute(int pos) は SQLException をスローします。public boolean relative(int offset) は SQLException をスローします。

Methods used by updatable result sets

public void updateInt(String fldname, int val) は SQLException をスローします。public void updateString(String fldname, String val) public void updateRow() は SQLException をスローします。public void deleteRow() は SQLException をスローします。public void moveToInsertRow() は SQLException をスローします。public void moveToCurrentRow() は SQLException をスローします。SQLException をスローします。

図 2.20 ResultSet の API の一部

そのようなレコードがない場合。relative メソッドは、結果セットを相対的な行数で配置します。特に、relative(1) は next と同じであり、relative(-1) は previous と同じです。

updateInt メソッドと updateString メソッドは、クライアント上の現在のレコードの指定されたフィールドを変更します。ただし、変更内容は updateRow が呼び出されるまでデータベースに送信されません。updateRow を呼び出す必要があるのは少々面倒ですが、これにより、JDBC はレコードの複数のフィールドの更新をエンジンへの 1 回の呼び出しにまとめることができます。

挿入は、挿入行の概念によって処理されます。この行はテーブル内に存在しません (つまり、スクロールして移動することはできません)。その目的は、新しいレコードのステージング領域として機能することです。クライアントは、moveToInsertRow を呼び出して結果セットを挿入行に配置し、次に updateXXX メソッドを呼び出してフィールドの値を設定し、次に updateRow を呼び出してレコードをデータベースに挿入し、最後に moveToCurrentRow を呼び出してレコード セットを挿入前の位置に再配置します。

デフォルトでは、レコード セットは前方のみで更新できません。クライアントがより強力なレコード セットを必要とする場合は、Connection の createStatement メソッドでそのように指定します。基本的な JDBC の引数なしの createStatement メソッドに加えて、クライアントがスクロール可能性と更新可能性を指定する 2 つの引数のメソッドもあります。たとえば、次のステートメントを考えてみましょう。

```
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);
```

この文から生成されるすべての結果セットは、スクロール可能かつ更新可能です。定数 TYPE_FORWARD_ONLY はスクロール不可能な結果セットを指定し、CONCUR_READ_ONLY は更新不可能な結果セットを指定します。これらの定数を組み合わせて使用することで、必要なスクロール可能性と更新可能性を実現できます。

例えば、図2.10のコードを思い出してください。このコードでは、ユーザーはCOURSEテーブルを反復処理して、必要なレコードを削除できました。図2.21では、このコードを次のように修正しています。

データソース ds = ... 接続 conn = ds.getConnection(
); conn.setAutocommit(false);

ステートメント stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
CONCUR_UPDATABLE); ResultSet rs = stmt.executeQuery("select * from COURSE"); while (
rs.next()) { String title = rs.getString("Title"); boolean goodCourse = getUserDecision(title); if (!g
oodCourse) rs.deleteRow(); }rs.close(); stmt.close(); conn.commit();

図2.21 図2.10のコードを修正する

更新可能な結果セットを使用します。削除された行は、次の呼び出しまで最新の状態のままであることに注意してください。

スクロール可能な結果セットの用途は限られています。ほとんどの場合、クライアントは出力レコードで何をしたいかわかっており、それらを 2 回調べる必要がないためです。クライアントがスクロール可能な結果セットを必要とするのは、通常、ユーザーがクエリの結果を操作できる場合のみです。たとえば、クエリの出力を Swing JTable オブジェクトとして表示するクライアントを考えてみましょう。出力レコードが多すぎて画面に収まらない場合、JTable はスクロールバーを表示し、ユーザーはスクロールバーをクリックしてレコード間を前後に移動できます。この状況では、ユーザーがスクロールバックしたときに前のレコードを取得できるように、クライアントは JTable オブジェクトにスクロール可能な結果セットを提供する必要があります。

2.2.6 追加のデータ型

整数値と文字列値に加えて、JDBC には他のさまざまな型を操作するメソッドも含まれています。たとえば、ResultSet インターフェイスを考えてみましょう。getInt メソッドと getString メソッドに加えて、getFloat、getDouble、getShort、getTime、getDate などのメソッドもあります。これらのメソッドはそれぞれ、現在のレコードの指定されたフィールドから値を読み取り、(可能な場合は) 指定された Java 型に変換します。もちろん、一般的には、数値 SQL フィールドなどには数値 JDBC メソッド (getInt、getFloat など) を使用するのが最も理にかなっています。ただし、JDBC は、メソッドによって指定された Java 型に SQL 値を変換しようとします。特に、任意の SQL 値を Java 文字列に変換することは常に可能です。

2.3 Java と SQL での計算

プログラマが JDBC クライアントを作成するときは常に、計算のどの部分をデータベース エンジンで実行し、どの部分を Java クライアントで実行するかという重要な決定を行う必要があります。このセクションでは、これらの質問について説明します。

図 2.5 の StudentMajor デモ クライアントをもう一度考えてみましょう。このプログラムでは、エンジンが SQL クエリを実行して STUDENT テーブルと DEPT テーブルの結合を計算し、すべての計算を実行します。クライアントの役割は、クエリ出力を取得して印刷することだけです。

対照的に、図 2.22 に示すように、クライアントがすべての計算を実行するように記述することもできます。このコードでは、エンジンの役割は STUDENT テーブルと DEPT テーブルの結果セットを作成することだけです。クライアントは残りのすべての作業、つまり結合の計算と結果の出力を実行します。

これら 2 つのバージョンのどちらが優れているでしょうか。明らかに、元のバージョンの方が洗練されています。コードが少ないだけでなく、コードが読みやすくなっています。しかし、効率はどうでしょうか。経験則として、クライアントで行う処理はできるだけ少なくする方が常に効率的です。主な理由は 2 つあります。

- 通常、エンジンからクライアントに転送するデータは少なくなります。これは、エンジンとクライアントが異なるマシン上にある場合に特に重要です。
- エンジンには、各テーブルがどのように実装されているか、および複雑なクエリ (結合など) を計算するための可能な方法についての詳細な専門知識が含まれています。クライアントがエンジンと同じくらい効率的にクエリを計算できる可能性はほとんどありません。

たとえば、図 2.22 のコードは、2 つのネストされたループを使用して結合を計算します。外側のループは STUDENT レコードを反復処理します。内側のループは、各学生の専攻に一致する DEPT レコードを検索します。これは妥当な結合アルゴリズムですが、特に効率的というわけではありません。第 13 章と第 14 章では、より効率的な実行につながるいくつかの手法について説明します。

図 2.5 と 2.22 は、非常に優れた JDBC コードと非常に悪い JDBC コードの両極端を例示しているため、比較するのは非常に簡単です。しかし、比較が困難な場合もあります。たとえば、図 2.17 の PreparedFindMajors デモ クライアントをもう一度考えてみましょう。このクライアントは、指定された専攻部門を持つ学生を返します。このコードは、エンジンに、STUDENT と MAJOR を結合する SQL クエリを実行するように要求します。結合の実行には時間がかかる可能性があると仮定します。よく考えた結果、結合を使用せずに必要なデータを取得できることに気付きました。そのアイデアは、2 つの単一テーブル クエリを使用することです。最初のクエリは、DEPT テーブルをスキャンして、指定された専攻名を持つレコードを検索し、その DId- 値を返します。2 番目のクエリは、その値を使用して、STUDENT レコードの MajorID 値を検索します。このアルゴリズムのコードは、図 2.23 に示されています。

これは非常にシンプルで、洗練されており、効率的です。必要なのは、2 つのテーブルをそれぞれ順番にスキャンすることだけであり、結合よりもはるかに高速です。努力を誇りに思ってよいでしょう。

残念ながら、あなたの努力は無駄になりました。新しいアルゴリズムは実際には新しいものではなく、結合の巧妙な実装に過ぎません。具体的には、マルチバッファ積です。

パブリック クラス BadStudentMajor { パブリック 静的 void main(String[] args) { ClientDataSource ds = new ClientDataSource(); ds.setServerName("localhost"); ds.setDatabaseName("studentdb"); 接続 conn = null; try {conn = ds.getConnection(); conn.setAutoCommit(false); try (Statement stmt1 = conn.createStatement(); Statement stmt2 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY); ResultSet rs1 = stmt1.executeQuery(ResultSet rs2 = stmt2.executeQuery(" System.out.println("Name\tMajor"); while (rs1.next()) { // 次の学生を取得します String sname = rs1.getString("SName"); String dname = null; rs2.beforeFirst(); while (rs2.next()) // その学生の専攻部門を検索します if (rs2.getInt("DId") == rs1.getInt("MajorId")) { dname = rs2.getString("DName"); break; } System.out.println(sname + "\t" + dname); } }conn.commit(); conn.close(); }catch(SQLException e) { e.printStackTrace(); try {if (conn != null) { conn.rollback(); conn.close(); } }catch (SQLException e2) {} } } } "select * from STUDENT"); select * from DEPT")) {

図 2.22 StudentMajor クライアントをコーディングする別の方法 (ただし悪い方法)

```
public static void main(String[] args) { String major = args[0]; String qry1 = "select DId fro
m DEPT where DName = ?"; String qry2 = "select * from STUDENT where MajorId = ?"; Clie
ntDataSource ds = new ClientDataSource(); ds.setServerName("localhost"); ds.setDatabaseNam
e("studentdb"); try (Connection conn = ds.getConnection()) { PreparedStatement stmt1 = conn.p
repareStatement(qry1); stmt1.setString(1, major); ResultSet rs1 = stmt1.executeQuery(); rs1.nex
t(); rs1.close(); stmt1.close(); PreparedStatement stmt2 = conn.prepareStatement(qry2); stmt2.se
tInt(1, deptid); ResultSet rs2 = stmt2.executeQuery(); System.out.println("ここに " + major + "
専攻があります"); System.out.println("Name\tGradYear"); while (rs2.next()) { String sname =
rs2.getString("sname"); int gradyear = rs2.getInt("gradyear"); System.out.println(sname + "\t" +
gradyear); }rs2.close(); stmt2.close(); }catch(Exception e) { e.printStackTrace(); } } パブリッ
ククラス CleverFindMajors { }
```

```
int deptid = rs1.getInt("DId");  // get the major's ID
```

図2.23 FindMajorsクライアントを実装する賢い方法

第 14 章では、具体化された内部テーブルについて説明しています。適切に作成されたデータベース エンジンは、このアルゴリズム (およびその他のアルゴリズム) を認識しており、これが最も効率的であることが判明した場合には、それを使用して結合を計算します。このように、データベース エンジンによって、あなたの巧妙な処理がすべて先取りされてしまいます。教訓は StudentMajor クライアントの場合と同じです。エンジンに処理を任せるのが最も効率的な戦略である傾向があります (また、最も簡単にコーディングできる戦略でもあります)。

初心者のJDBCプログラマーが犯す間違いの1つは、クライアントで多くのことをやろうとしすぎることです。プログラマーは、Javaでクエリを実装する非常に賢い方法を知っていると考えるかもしれません。または、SQLでクエリを表現する方法がわからず、Javaでクエリをコーディングする方が快適だと感じるかもしれません。

いずれの場合も、クエリを Java でコーディングするという決定は、ほとんどの場合間違いです。プログラマーは、データベース エンジンが機能することを信頼する必要があります。[2]

2.4 章の要約

- JDBC メソッドは、Java クライアントとデータベース エンジン間のデータ転送を管理します。
- 基本的な JDBC は、Driver、Connection、Statement、ResultSet、ResultSetMetaData の 5 つのインターフェースで構成されます。
- ドライバー オブジェクトは、エンジンに接続するための低レベルの詳細をカプセル化します。クライアントがエンジンに接続する場合、適切なドライバー クラスのコピーを取得する必要があります。ドライバー クラスとその接続文字列は、JDBC プログラム内の唯一のベンダー固有のコードです。その他はすべて、ベンダーに依存しない JDBC インターフェイスを参照します。
- 結果セットと接続には、他のクライアントが必要とする可能性のあるリソースが保持されます。JDBC クライアントは、常にできるだけ早くそれらを閉じる必要があります。
- すべての JDBC メソッドは SQLException をスローする可能性があります。クライアントはこれらの例外をチェックする義務があります。
- ResultSetMetaData のメソッドは、出力テーブルのスキーマに関する情報、つまり各フィールドの名前、タイプ、および表示サイズを提供します。この情報は、SQL インタープリタのように、クライアントがユーザーから直接クエリを受け入れる場合に役立ちます。
- 基本的な JDBC クライアントは、ドライバー クラスを直接呼び出します。完全な JDBC は、接続プロセスを簡素化し、ベンダーに依存しないようにするために、クラス DriverManager とインターフェイス DataSource を提供します。
- DriverManager クラスは、ドライバーのコレクションを保持します。クライアントは、明示的に、または (推奨) システム プロパティ ファイルを介して、ドライバー マネージャーにドライバーを登録します。クライアントがデータベースに接続する場合、ドライバー マネージャーに接続文字列を提供し、ドライバー マネージャーがクライアントのために接続を行います。
- DataSource オブジェクトは、ドライバーと接続文字列の両方をカプセル化するため、ベンダーにさらに中立です。そのため、クライアントは接続の詳細を知らなくてもデータベース エンジンに接続できます。データベース管理者は、さまざまな DataSource オブジェクトを作成し、クライアントが使用できるようにサーバー上に配置できます。
- 基本的な JDBC クライアントはトランザクションの存在を無視します。データベース エンジンはこれらのクライアントを自動コミット モードで実行します。つまり、各 SQL ステートメントは独自のトランザクションになります。

[2] At least, you should start by trusting that the engine will be efficient. If you discover that your application is running slowly because the engine is not executing the join efficiently, then you can recode the program as in Fig. 2.23. But it is always best to avoid premature cleverness.

- トランザクション内のすべてのデータベース インタラクションは、1 つの単位として扱われます。トランザクションは、現在の作業単位が正常に完了するとコミットされます。トランザクションは、コミットできない場合はロールバックされます。データベース エンジンは、そのトランザクションによって行われたすべての変更を元に戻すことによってロールバックを実装します。
- 自動コミットは単純で重要でない JDBC クライアントに適したデフォルト モードです。クライアントが重要なタスクを実行する場合、プログラマはトランザクションのニーズを慎重に分析する必要があります。クライアントは、setAutoCommit(false) を呼び出して自動コミットをオフにします。この呼び出しにより、エンジンは新しいトランザクションを開始します。次に、クライアントは、現在のトランザクションを完了して新しいトランザクションを開始する必要があるときに、commit または rollback を呼び出します。クライアントが自動コミットをオフにするとき、関連するトランザクションをロールバックして取り消すことも処理する必要があります。
- クライアントは、setTransactionIsolation メソッドを使用して分離レベルを指定することもできます。JDBC では、次の 4 つの分離レベルが定義されています。 – Read-Uncommitted 分離は、分離がまったく行われないことを意味します。トランザクションでは、コミットされていないデータ、繰り返し不可能な読み取り、またはファントム レコードの読み取りが原因で問題が発生する可能性があります。 – Read-Committed 分離は、トランザクションによるコミットされていない値へのアクセスを禁止します。繰り返し不可能な読み取りとファントムに関連する問題は、依然として発生する可能性があります。 – Repeatable-Read 分離は、読み取りが常に繰り返し可能になるように、Read-Committed を拡張します。発生する可能性のある問題は、ファントムによるものだけです。 – Serializable 分離は、問題が発生しないことを保証します。

- シリアル化可能な分離が望ましいのは明らかですが、その実装によりトランザクションの実行速度が遅くなる傾向があります。プログラマーは、クライアントとの同時実行エラーのリスクを分析し、リスクが許容できる場合にのみ、より制限の少ない分離レベルを選択する必要があります。
- 準備済みステートメントには、パラメータのプレースホルダを持つことができる SQL ステートメントが関連付けられています。クライアントは後でパラメータに値を割り当て、ステートメントを実行できます。準備済みステートメントは、動的に生成された SQL ステートメントを処理するのに便利な方法です。さらに、準備済みステートメントは、パラメータが割り当てられる前にコンパイルできるため、準備済みステートメントを複数回 (ループ内など) 実行することが非常に効率的になります。
- 完全な JDBC を使用すると、結果セットをスクロールおよび更新できます。デフォルトでは、レコード セットは前方のみで更新できません。クライアントがより強力なレコード セットを必要とする場合は、Connection の createStatement メソッドでそのように指定します。
- JDBC クライアントを作成する場合の原則は、エンジンにできるだけ多くの作業をさせるということです。データベース エンジンは非常に洗練されており、通常、必要なデータを取得する最も効率的な方法を知っています。クライアントが、必要なデータを正確に取得する SQL ステートメントを決定し、それをエンジンに送信するのは、ほとんどの場合に良い考えです。つまり、プログラマーは、エンジンがその仕事をすることを信頼する必要があります。

2.5 推奨される読み物

JDBC に関する包括的でよく書かれた書籍として、Fisher 他 (2003) があります。その一部は、docs.oracle.com/javase/tutorial/jdbc にオンライン チュートリアルとして公開されています。さらに、各データベース ベンダーは、そのドライバの使用方法やベンダー固有の問題を説明するドキュメントを提供しています。特定のエンジン用のクライアントを作成する場合は、ドキュメントをよく理解しておく必要があります。
Fisher, M.、Ellis, J.、Bruce, J. (2003)。『JDBC API チュートリアルおよびリファレンス (第 3 版)』。Addison Wesley。

2.6 演習

概念演習

2.1. Derby のドキュメントでは、挿入シーケンスを実行するときに自動コミットをオフにすることを推奨しています。なぜこの推奨を行うと考えるのか説明してください。

プログラミング演習

2.2. 大学のデータベースに SQL クエリをいくつか記述します。各クエリに対して、Derby を使用してそのクエリを実行し、出力テーブルを出力するプログラムを作成します。 2.3. SimpleIJ プログラムでは、各 SQL 文が 1 行のテキストである必要があります。Derby の ij プログラムと同様に、文が複数行で構成され、セミコロンで終了するように修正します。 2.4. Derby クラス ClientDataSource と同様に動作する SimpleDB 用の NetworkDataSource クラスを作成します。このクラスを simpledb.jdbc.network パッケージに追加します。コードでは、インターフェイス javax.sql.DataSource (およびそのスーパークラス) のすべてのメソッドを実装する必要はありません。実際、実装する必要があるメソッドは、引数なしのメソッド getConnection() だけです。NetworkDataSource には、どのようなベンダー固有のメソッドが必要ですか。 2.5. SQL コマンドを含むテキスト ファイルを作成できると便利なことがよくあります。これらのコマンドは、JDBC プログラムによってバッチで実行できます。指定されたテキスト ファイルからコマンドを読み取り、それを実行する JDBC プログラムを作成します。ファイルの各行は個別のコマンドであると仮定します。2.6. 結果セットを使用して Java JTable オブジェクトにデータを入力する方法を調べます。(ヒント: クラス AbstractTableModel を拡張する必要があります。) 次に、デモ クライアント FindMajors を修正して、その出力を JTable に表示する GUI インターフェイスを作成します。2.7. 次のタスクの JDBC コードを作成します。(a) テキスト ファイルから既存のテーブルにデータをインポートします。テキスト ファイルには、1 行に 1 つのレコードがあり、各フィールドはタブで区切られている必要があります。ファイルの最初の行は、

フィールドの名前である必要があります。クライアントは、ファイル名とテーブル名を入力として受け取り、レコードをテーブルに挿入する必要があります。 (b) データをテキスト ファイルにエクスポートします。クライアントは、ファイル名とテーブル名を入力として受け取り、各レコードの内容をファイルに書き込む必要があります。ファイルの最初の行は、フィールドの名前である必要があります。

2.8. この章では、結果セットに null 値が含まれる可能性については考慮していません。null 値を確認するには、ResultSet の wasNull メソッドを使用します。getInt または getString を呼び出してフィールド値を取得するとします。その直後に wasNull を呼び出すと、取得した値が null だった場合に true が返されます。たとえば、次のループは、卒業年度を出力しますが、その一部が null である可能性があると想定しています。

```
while(rs.next()) { int gradyr = rs.getInt("gradyear")
; if (rs.wasNull()) System.out.println("null"); else S
ystem.out.println(gradyr); }
```

(a) 学生名が null になる可能性があるという想定で、StudentMajor デモ クライアントのコードを書き直します。 (b) SimpleIJ デモ クライアントを変更して、SimpleDB ではなく Derby に接続するようにします。次に、フィールド値が null になる可能性があるという想定でコードを書き直します。

第3章 ディスクとファイルの管理

データベース エンジンは、ディスクやフラッシュ ドライブなどの永続的なストレージ デバイスにデータを保存します。この章では、これらのデバイスの特性を調査し、速度と信頼性を向上させることができる技術 (RAID など) について検討します。また、オペレーティング システムがこれらのデバイスと対話するために提供する 2 つのインターフェイス (ブロック レベル インターフェイスとファイル レベル インターフェイス) についても検討し、データベース システムに最適な 2 つのインターフェイスの組み合わせを提案します。最後に、SimpleDB ファイル マネージャーを詳細に検討し、その API と実装について学習します。

3.1 永続的なデータストレージ

データベースの内容は永続的に保存する必要があります。そうしないと、データベース システムまたはコンピュータがダウンしてもデータが失われることはありません。このセクションでは、特に便利な 2 つのハードウェア テクノロジであるディスク ドライブとフラッシュ ドライブについて説明します。フラッシュ ドライブはまだディスク ドライブほど普及していませんが、テクノロジが成熟するにつれてその重要性は増すでしょう。まずはディスク ドライブから始めましょう。

3.1.1 ディスクドライブ

ディスク ドライブには、1 つ以上の回転するプラッタが含まれています。プラッタには同心円状のトラックがあり、各トラックはバイトのシーケンスで構成されています。プラッタからのバイトの読み取り (およびプラッタへのバイトの書き込み) は、読み取り/書き込みヘッドを備えた可動アームによって行われます。アームは目的のトラックに配置され、ヘッドは回転しながらバイトを読み取り (または書き込み) ます。図 3.1 は、1 つのプラッタを持つディスク ドライブを上から見た図です。もちろん、この図は縮尺どおりに描かれていません。一般的なプラッタには何十もの トラックがあるためです。

現代のディスク ドライブには通常、複数のプラッタが搭載されています。スペースの効率を高めるために、通常はプラッターのペアが背中合わせに連結され、両面プラッターのように見えるものが作成されます。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_3

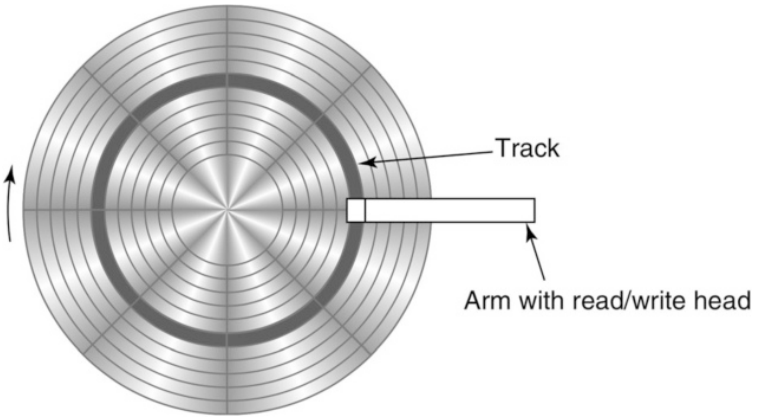


図3.1 1枚ディスクドライブの上面図

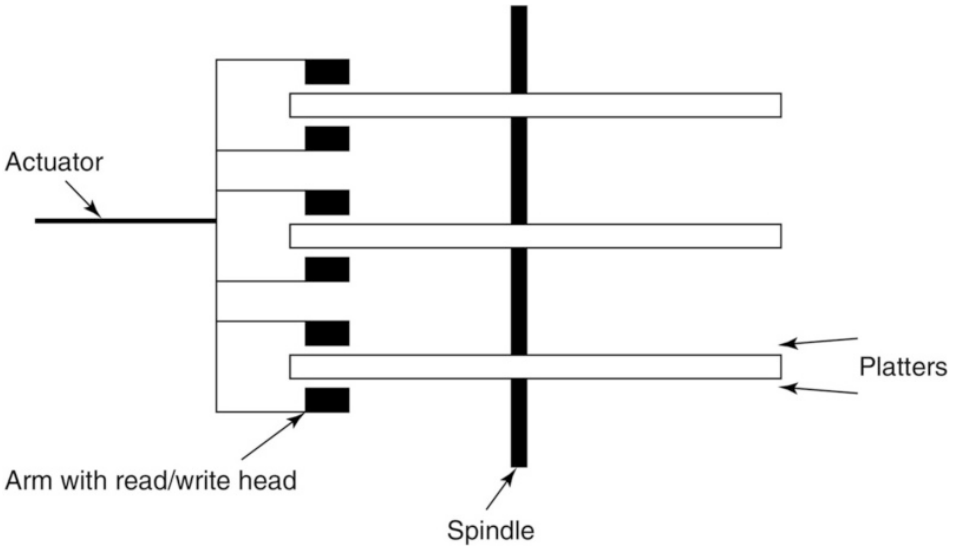


図3.2 マルチプラッタディスクドライブの側面図

しかし、概念的には、各面は依然として別々のプラッタです。各プラッタには独自の読み取り/書き込みヘッドがあります。これらのヘッドは独立して動くのではなく、すべて 1 つのアクチュエータに接続されており、各プラッタの同じトラックに同時に移動されます。さらに、コンピュータへのデータ パスは 1 つしかないため、一度にアクティブにできる読み取り/書き込みヘッドは 1 つだけです。図 3.2 は、マルチプラッタ ディスク ドライブの側面図です。

ディスク ドライブの一般的なパフォーマンスは、容量、回転速度、転送速度、シーク時間の 4 つの値で測定できます。

ドライブの容量は、保存できるバイト数です。この値は、プラッタの数、プラッタあたりのトラック数、およびトラックあたりのバイト数によって決まります。プラッタはほぼ標準サイズで提供される傾向があるため、メーカーは主にプラッタの密度を上げることによって、つまりプラッタあたりのトラック数とトラックあたりのバイト数を増やすことによって容量を増やします。現在では、40 GB を超えるプラッタ容量が一般的です。

回転速度はプラッターの回転速度で、通常は 1 分あたりの回転数で表されます。一般的な速度は 5400 rpm から 15,000 rpm の範囲です。

転送速度とは、ディスク ヘッドを通過してメモリに転送されるバイトの速度です。たとえば、プラッターが 1 回転する間に、トラック全体のバイトを転送できます。転送速度は、回転速度とトラックあたりのバイト数の両方によって決まります。100 MB/秒の速度が一般的です。

シーク時間は、アクチュエータがディスク ヘッドを現在の位置から要求されたトラックまで移動するのに要する時間です。この値は、トラバースする必要があるトラックの数によって異なります。最低 0 (宛先トラックが開始トラックと同じ場合) から最高 15〜20 ミリ秒 (宛先トラックと開始トラックがプラッターの異なる端にある場合) まであります。平均シーク時間は通常、アクチュエータ速度の適切な推定値を提供します。最近のディスクの平均シーク時間は約 5 ミリ秒です。

次の例を考えてみましょう。4 枚のプラッタを備えたディスク ドライブが 10,000 rpm で回転し、平均シーク時間が 5 ミリ秒であるとします。各プラッタには 10,000 トラックが含まれており、各トラックには 500,000 バイトが含まれています。次に計算された値をいくつか示します [1]:

ドライブの容量:
500,000 バイト/トラック x 10,000 トラック/プラッタ x 4 プラッタ/ドライブ = 2
0,000,000,000 バイト、または約 20 GB

転送速度:
500,000 バイト/回転 x 10,000 回転/60 秒 = 83,333,333 バイト/秒、または
約 83MB/秒

3.1.2 ディスクドライブへのアクセス

ディスク アクセスとは、ディスク ドライブからメモリにバイトをいくつか読み取るか、メモリからディスクにバイトをいくつか書き込む要求です。これらのバイトは、プラッタ上のトラックの連続した部分にある必要があります。ディスク ドライブは、次の 3 つの段階でディスク アクセスを実行します。

- ヘッドを指定されたトラックに移動します。この時間をシーク時間と呼びます。
- 最初の目的のバイトがディスク ヘッドの下になるまで、プラッターが回転するのを待機します。この時間を回転遅延と呼びます。
- プラッターが回転し続けると、ディスク ヘッドの下に表示される各バイトが読み取られ (または各バイトが書き込まれ)、目的の最後のバイトが表示されます。この時間を転送時間と呼びます。

ディスクアクセスの実行に必要な時間は、シーク時間、回転遅延、転送時間の合計です。これらの時間はそれぞれ、機械的な制約によって制限されます。

[1] Technically, 1 KB = 1024 bytes, 1 MB = 1,048,576 bytes, and 1 GB = 1,073,741,824 bytes. For convenience, I round them down to one thousand, one million, and one billion bytes, respectively.

ディスクの動き。機械的な動きは電気的な動きよりも大幅に遅いため、ディスク ドライブは RAM よりもずっと低速です。シーク時間と回転遅延は特に厄介です。この 2 つの時間は、すべてのディスク操作を強制的に待機させるオーバーヘッドに他なりません。

ディスク アクセスの正確なシーク時間と回転遅延を計算するのは、ディスクの以前の状態を知る必要があるため、現実的ではありません。代わりに、これらの時間を平均値を使用して見積もることができます。平均シーク時間については既にわかっています。平均回転遅延は簡単に計算できます。回転遅延は 0 (最初のバイトがたまたまヘッドの下にある場合) から 1 回転に要する時間 (最初のバイトがヘッドを通過した場合) までの範囲になります。平均すると、プラッターが目的の位置にくるまで 1/2 回転待つ必要があります。したがって、平均回転遅延は回転時間の半分になります。

転送時間も転送速度から簡単に計算できます。特に、転送速度が r バイト/秒で、b バイトを転送する場合、転送時間は b/r 秒になります。

たとえば、ディスク ドライブが 10,000 rpm で回転し、平均シーク時間が 5 ミリ秒、転送速度が 83 MB/秒であるとします。計算されたコストは次のとおりです。

平均回転遅延:
60秒/分×1分/10,000回転×½回転= 0.003秒または3ミリ秒

1バイトの転送時間:
1 バイト x 1 秒 / 83,000,000 バイト = 0.000000
012 秒または 0.000012 ミリ秒

1000バイトの転送時間:
1,000 バイト x 1 秒 / 83,000,000 バイト = 0.000012
秒または 0.012 ミリ秒

1バイトへのアクセスにかかる推定時間:
5 ms (シーク) + 3 ms (回転遅延) + 0.000012 ms (転送) = 8.000012 ms

1000 バイトへのアクセスにかかる推定時間:
5 ms (シーク) + 3 ms (回転遅延) + 0.012 ms (転送) = 8.012 ms

1000 バイトの推定アクセス時間は、1 バイトの場合と基本的に同じであることに注意してください。つまり、ディスクから数バイトだけにアクセスしても意味がありません。実際、アクセスしたくてもできません。最近のディスクは、各トラックが固定長のセクターに分割されるように構築されており、ディスクの読み取り (または書き込み) は、一度に 1 つのセクター全体に対して実行する必要があります。セクターのサイズは、ディスクの製造元によって決定される場合もあれば、ディスクのフォーマット時に選択される場合もあります。一般的なセクター サイズは 512 バイトです。

3.1.3 ディスクアクセス時間の改善

ディスク ドライブは非常に遅いため、アクセス時間を改善するためにいくつかの手法が開発されています。このセクションでは、ディスク キャッシュ、シリンダ、ディスク ストライピングの 3 つの手法について説明します。

ディスクキャッシュ
ディスク キャッシュは、ディスク ドライブにバンドルされているメモリで、通常は数千のセクターの内容を保存できるほどの大きさです。ディスク ドライブがディスクからセクターを読み取るたびに、そのセクターの内容をキャッシュに保存します。キャッシュがいっぱいになると、新しいセクターが古いセクターに置き換わります。セクターが要求されると、ディスク ドライブはキャッシュをチェックします。セクターがキャッシュ内にある場合は、実際のディスク アクセスを行わずに、すぐにコンピュータに返すことができます。

マクリン_エンジンが比較的短い期間に同じセクターを複数回要求するとします。最初の要求ではセクターがキャッシュに取り込まれ、その後の要求ではキャッシュから取り出されるため、ディスク アクセスが節約されます。ただし、この機能はデータベース エンジンにとって特に役立つものではありません。これは、エンジンが既に独自のキャッシュを実行しているためです (第 4 章で説明します)。セクターが複数回要求された場合、エンジンは独自のキャッシュでセクターを見つけ、ディスクにアクセスする必要さえありません。

ディスク キャッシュの真の価値は、セクターをプリフェッチする機能にあります。要求されたセクターだけを読み取るのではなく、ディスク ドライブは、そのセクターを含むトラック全体をキャッシュに読み込んで、トラックの他のセクターが後で要求されることを期待できます。重要なのは、トラック全体を読み取るのに、単一のセクターを読み取るのに比べてそれほど時間がかからないことです。特に、ディスクは読み取り/書き込みヘッドの下にあるセクターからトラックを読み取り、回転中ずっと読み取りを継続できるため、回転による遅延はありません。アクセス時間を比較します。

セクターの読み取り時間 = シーク時間 + ½ 回転時間 + セクターの回転時間 トラックの読み取り時間 = シーク時間 + 回転時間

つまり、単一のセクターの読み取りとセクターで満たされたトラックの読み取りの違いは、ディスクの回転時間の半分未満です。データベース エンジンがトラック上の他の 1 つのセクターのみを要求する場合、トラック全体をキャッシュに読み取ることで時間が節約されます。

シリンダー
データベース システムは、関連情報を近くのセクターに格納することで、ディスク アクセス時間を改善できます。たとえば、ファイルを格納する理想的な方法は、その内容をプラッターの同じトラックに配置することです。この戦略は、ディスクがトラック ベースのキャッシュを実行する場合に最適です。これは、ファイル全体が 1 回のディスク アクセスで読み取られるためです。ただし、この戦略はキャッシュがない場合でも有効です。これは、シーク時間がなくなるためです。別のセクターが読み取られるたびに、ディスク ヘッドが適切なトラックに配置されます。[2]

[2] A file whose contents are wildly scattered across different tracks of the disk is said to be *fragmented*. Many operating systems provide a defragmentation utility that improves access time by relocating each file so that its sectors are as contiguous as possible.

ファイルが複数のトラックを占めているとします。トラック間のシーク時間ができるだけ短くなるように、その内容をプラッタの近くのトラックに保存するのが良い方法です。しかし、さらに良い方法は、その内容を他のプラッタの同じトラックに保存することです。各プラッタの読み取り/書き込みヘッドはすべて一緒に動くので、同じトラック番号を持つすべてのトラックには、追加のシーク時間なしでアクセスできます。

同じトラック番号を持つトラックのセットは、ディスクの上部からそれらのトラックを見ると、シリンダーの外側を描写するため、シリンダーと呼ばれます。実際には、シリンダーのすべてのセクターに追加のシークなしでアクセスできるため、シリンダーは非常に大きなトラックのように扱うことができます。

ディスクストライピング

ディスク アクセス時間を改善するもう 1 つの方法は、複数のディスク ドライブを使用することです。2 つの小さなドライブは 2 つの独立したアクチュエーターを備えているため、2 つの異なるセクター要求に同時に応答でき、1 つの大きなドライブよりも高速です。たとえば、2 つの 20 GB ディスクを連続して動作させると、1 つの 40 GB ディスクの約 2 倍の速度になります。この高速化は適切にスケール化されます。一般に、N 個のディスクは 1 つのディスクの約 N 倍の速度になります。(もちろん、複数の小さなドライブは 1 つの大きなドライブよりも高価であるため、効率性の向上にはコストがかかります。)

また、複数の小さなディスクを常に使用し続けることができない場合、その効率は失われます。たとえば、1 つのディスクに頻繁に使用されるファイルが格納され、他のディスクにはあまり使用されないアーカイブ ファイルが格納されているとします。この場合、最初のディスクがすべての作業を実行し、他のディスクはほとんどの時間アイドル状態になります。この設定では、単一のディスクとほぼ同じ効率になります。

したがって、問題は、複数のディスク間でワークロードのバランスをとる方法です。データベース管理者は、各ディスクにファイルを最適に分散するためにファイルの使用状況を分析しようと試みることができますが、この方法は現実的ではありません。実行が難しく、保証が難しく、時間の経過とともに継続的に再評価および修正する必要があります。幸いなことに、ディスクストライピングと呼ばれる、はるかに優れた方法があります。

ディスクストライピング戦略では、コントローラを使用して小さなディスクをオペレーティング システムから隠し、単一の大きなディスクがあるかのように見せます。コントローラは、仮想ディスクのセクター要求を実際のディスクのセクター要求にマッピングします。マッピングは次のように機能します。それぞれ k 個のセクターを持つ N 個の小さなディスクがあるとします。仮想ディスクには N"k 個のセクターがあり、これらのセクターは実際のディスクのセクターに交互に割り当てられます。ディスク 0 には仮想セクター 0、N、2N などが含まれます。ディスク 1 には仮想セクター 1、N+1、2N+1 などが含まれます。ディスク ストライピングという用語は、次のイメージに由来しています。各小さなディスクが異なる色で塗られていると想像すると、仮想ディスクはセクターが交互に色で塗られたストライプのように見えます。[3] 図 3.3 を参照してください。

[3]Most controllers allow a user to define a stripe to be of any size, not just a sector. For example, a track makes a good stripe if the disk drives are also performing track-based disk caching. The optimal stripe size depends on many factors and is often determined by trial and error.

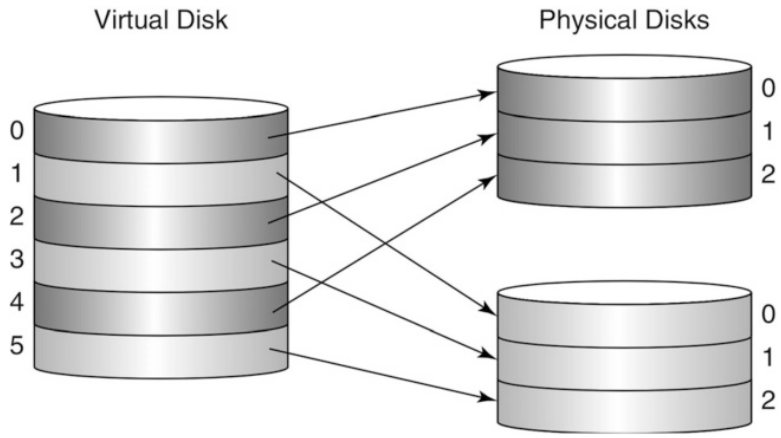


図3.3 ディスクストライピング

ディスク ストライピングは、データベースを小さなディスクに均等に分散するため効果的です。ランダムなセクターに対する要求が到着すると、その要求は均等な確率で小さなディスクの 1 つに送信されます。また、連続したセクターに対する複数の要求が到着すると、それらは異なるディスクに送信されます。したがって、ディスクは可能な限り均一に動作することが保証されます。

3.1.4 ミラーリングによるディスクの信頼性の向上

データベースのユーザーは、データがディスク上で安全に保持され、失われたり破損したりしないことを期待しています。残念ながら、ディスク ドライブは完全に信頼できるわけではありません。プラッター上の磁性材料が劣化し、セクターが読み取り不能になることがあります。また、ほこりや衝撃によって読み取り/書き込みヘッドがプラッターに擦れ、影響を受けたセクターが破損することもあります（「ヘッド クラッシュ」）。

ディスク障害に対する最も明白な対策は、ディスクの内容のコピーを保持することです。たとえば、ディスクのバックアップを毎晩作成しておき、ディスクが故障したときには、新しいディスクを購入してバックアップをコピーするだけです。この戦略の問題点は、ディスクをバックアップした時点から故障した時点までの間にディスクに加えられた変更がすべて失われることです。この問題を回避する唯一の方法は、ディスクに加えられた変更をすべて、発生した時点で複製することです。つまり、ディスクの同一バージョンを 2 つ保持する必要があります。これらのバージョンは、互いのミラーと呼ばれます。

ストライピングと同様に、2 つのミラー ディスクを管理するにはコントローラが必要です。データベース システムがディスクの読み取りを要求すると、コントローラはいずれかのディスクの指定されたセクターにアクセスできます。ディスクの書き込みが要求されると、コントローラは両方のディスクに同じ書き込みを実行します。理論上は、これらの 2 つのディスク書き込みは並列に実行でき、追加の時間はかかりません。ただし、実際には、システム クラッシュを防ぐために、ミラーに順番に書き込むことが重要です。問題は、ディスク書き込みの途中でシステムがクラッシュすると、そのセクターの内容が失われることです。したがって、両方のミラーに並列に書き込むと、セクターの両方のコピーが失われる可能性がありますが、

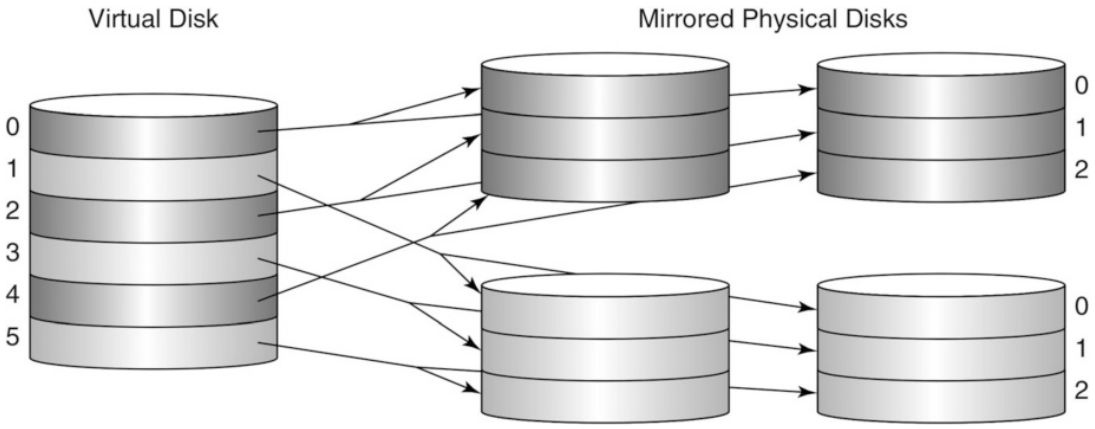


図3.4 ミラーを使用したディスクストライピング

ミラーが順番に書き込まれると、少なくとも 1 つのミラーは破損しなくなります。

ミラーリングされたペアの 1 つのディスクに障害が発生したとします。データベース管理者は、次の手順を実行してシステムを回復できます。

1. システムをシャットダウンします。 2. 故障したディスクを新しいディスクに交換します。 3. 正常なディスクから新しいディスクにデータをコピーします。 4. システムを再起動します。

残念ながら、この手順は絶対確実ではありません。正常なディスクが新しいディスクへのコピー中に故障すると、データが失われる可能性があります。両方のディスクが数時間以内に故障する可能性は低いですが (今日のディスクでは約 60,000 分の 1)、データベースが重要な場合は、この小さなリスクは受け入れられない可能性があります。2 つのミラー ディスクではなく 3 つのミラー ディスクを使用することで、リスクを軽減できます。この場合、データが失われるのは、3 つのディスクすべてが同じ数時間以内に故障した場合のみです。このような可能性はゼロではありませんが、非常に低いため、安心して無視できます。

ミラーリングはディスク ストライピングと共存できます。一般的な戦略は、ストライプ ディスクをミラーリングすることです。たとえば、4 つの 20 GB ドライブに 40 GB のデータを保存できます。ドライブのうち 2 つはストライプ化され、残りの 2 つはストライプ ドライブのミラーになります。このような構成は高速で信頼性も高くなります。図 3.4 を参照してください。

3.1.5 パリティの保存によるディスクの信頼性の向上

ミラーリングの欠点は、同じ量のデータを格納するのに2倍のディスクが必要になることです。この負担は、ディスクストライピングを使用する場合に特に顕著です。15台の20GBドライブを使用して300GBのデータを格納する場合、ミラーとしてさらに15台のドライブを購入する必要があります。大規模なデータベースのインストールでは、多数の小さなディスクをストライピングして巨大な仮想ディスクを作成することは珍しくなく、

ミラーのためだけに同じ数のディスクを使用するのは魅力的ではありません。多くのミラー ディスクを使用せずに、障害が発生したディスクから回復できれば便利です。

実際、1 つのディスクを使用して任意の数の他のディスクをバックアップする賢い方法があります。この戦略は、バックアップ ディスクにパリティ情報を格納することによって機能します。パリティは、ビットのセット S に対して次のように定義されます。

- S に奇数個の 1 が含まれている場合、S のパリティは 1 になります。
- S に 1 が偶数個含まれている場合、S のパリティは 0 になります。

つまり、パリティ ビットを S に追加すると、常に 1 が偶数個になります。

パリティには、次の興味深く重要な特性があります。パリティがわかっていれば、任意のビットの値は他のビットの値から決定できます。たとえば、S ¼ {1, 0, 1} とします。S のパリティは 0 です。1 の数が偶数だからです。最初のビットの値が失われたとします。パリティが 0 なので、{?, 0, 1 } の 1 の数は偶数でなければなりません。したがって、失われたビットは 1 であると推測できます。他の各ビット (パリティ ビットを含む) についても同様の推論が可能です。

パリティのこの使用法はディスクにも適用されます。N 個の同一サイズのディスクがあるとします。ディスクの 1 つをパリティ ディスクとして選択し、他の N 個のディスクにストライプ データを保持させます。パリティ ディスクの各ビットは、他のすべてのディスクの対応するビットのパリティを見つけることによって計算されます。いずれかのディスク (パリティ ディスクを含む) に障害が発生した場合、そのディスクの内容は、他のディスクの内容をビットごとに調べることで再構築できます。図 3.5 を参照してください。

ディスクはコントローラによって管理されます。読み取りおよび書き込み要求は、基本的にストライピングと同じように処理されます。コントローラは、要求されたセクターを保持するディスクを決定し、その読み取り/書き込み操作を実行します。違いは、書き込み要求はパリティ ディスクの対応するセクターも更新する必要があることです。コントローラは、変更されたセクターのどのビットが変更されたかを判断することで、更新されたパリティを計算できます。ルールは、ビットが変更された場合、対応するパリティ ビットも変更する必要があるというものです。したがって、コントローラはセクター書き込みを実行するために 4 回のディスク アクセスを必要とします。

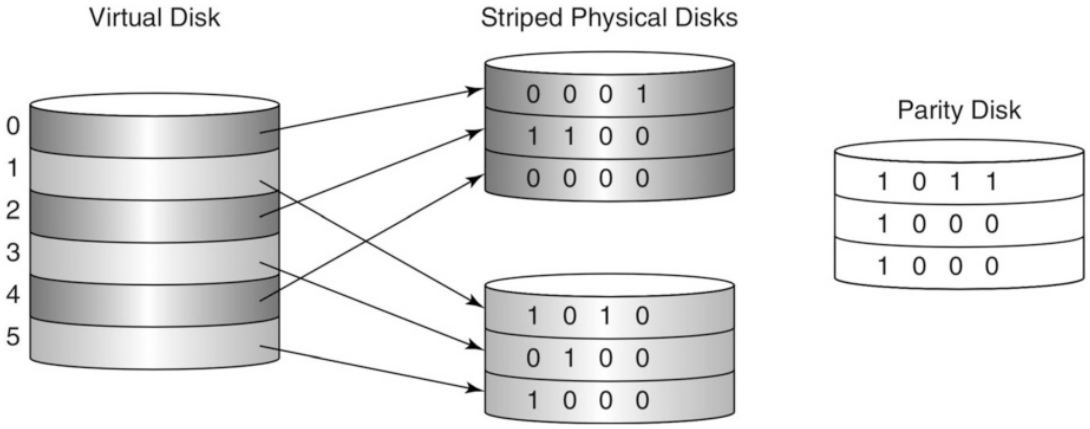


図3.5 パリティ付きディスクストライピング

操作: セクターと対応するパリティ セクターを読み取り (新しいパリティ ビットを計算するため)、両方のセクターの新しい内容を書き込む必要があります。

パリティ情報の使用は、1 つのディスクで任意の数の他のディスクを確実にバックアップできるという意味で、ある意味魔法のようです。ただし、この魔法には 2 つの欠点があります。

パリティを使用する場合の 1 つ目の欠点は、セクター書き込み操作に時間がかかることです。これは、2 つのディスクからの読み取りと書き込みの両方が必要になるためです。経験上、パリティを使用すると、ストライピングの効率が約 20% 低下することが分かっています。

パリティの 2 つ目の欠点は、データベースが回復不可能なマルチディスク障害に対してより脆弱になることです。ディスク障害が発生した場合を考えてみましょう。障害が発生したディスクを再構築するには他のすべてのディスクが必要であり、そのうちの 1 つが障害を起こすと悲惨な結果になります。データベースが多数の小さなディスク (たとえば 100 個程度) で構成されている場合、2 つ目の障害が発生する可能性が非常に高くなります。この状況をミラーリングと比較すると、障害が発生したディスクからの回復にはミラーが障害を起こさないことだけが必要であり、その可能性ははるかに低くなります。

3.1.6 RAID

前のセクションでは、複数のディスクを使用する 3 つの方法、つまりディスク アクセス時間を高速化するためのストライピングと、ディスク障害に対する保護のためのミラーリングとパリティについて説明しました。これらの戦略では、コントローラを使用して複数のディスクの存在をオペレーティング システムから隠し、単一の仮想ディスクがあるように見せかけます。コントローラは、各仮想読み取り/書き込み操作を、基礎となるディスク上の 1 つ以上の操作にマッピングします。コントローラはソフトウェアまたはハードウェアで実装できますが、ハードウェア コントローラの方が広く普及しています。これらの戦略は、RAID (Redundant Array of Inexpensive Disks) と呼ばれる大規模な戦略コレクションの一部です。RAID レベルは 7 つあります。

- RAID-0 はストライピングであり、ディスク障害に対する保護はありません。ストライピングされたディスクの 1 つに障害が発生すると、データベース全体が破損する可能性があります。
- RAID-1 はミラーリングとパリティのないストライピングです。
- RAID-2 は、セクター ストライピングの代わりにビット ストライピングを使用し、パリティの代わりにエラー訂正コードに基づく冗長性メカニズムを使用します。この戦略は実装が難しく、パフォーマンスも低いことが判明しています。現在は使用されていません。
- RAID-3 と RAID-4 は、ストライピングとパリティを使用します。違いは、RAID-3 がバイト ストライピングを使用するのに対し、RAID-4 はセクター ストライピングを使用することです。一般に、セクター ストライピングはディスク セクターの単位に対応しているため、より効率的です。
- RAID-5 は RAID-4 に似ていますが、すべてのパリティ情報を別のディスクに保存するのではなく、パリティ情報をデータ ディスクに分散します。つまり、N 個のデータ ディスクがある場合、各ディスクの N 番目のセクターごとにパリティ情報が格納されます。この戦略は、ボトルネックとなる単一のパリティ ディスクがなくなるため、RAID-4 よりも効率的です。演習 3.5 を参照してください。

- RAID-6 は、2 種類のパリティ情報を保持する点を除いて、RAID-5 と似ています。したがって、この戦略では、2 つの同時ディスク障害に対処できますが、追加のパリティ情報を保持するために別のディスクが必要になります。

最も人気のある 2 つの RAID レベルは、RAID-1 と RAID-5 です。この 2 つの選択は、実際にはミラーリングとパリティのどちらかになります。データベースのインストールでは、ミラーリングの方がより確実な選択肢になる傾向があります。その理由は、まず、その速度と堅牢性、そして、追加ディスク ドライブのコストが非常に低くなったためです。

3.1.7 フラッシュドライブ

ディスク ドライブは現在のデータベース システムでは一般的に使用されていますが、その動作は回転するプラッターと移動するアクチュエータの機械的な動作に完全に依存しているという克服できない欠点があります。この欠点により、ディスク ドライブは電子メモリに比べて本質的に低速であり、落下、振動、その他の衝撃による損傷を受けやすくなります。

フラッシュ メモリは、ディスク ドライブに代わる可能性のある最新のテクノロジです。RAM と同様に半導体テクノロジを使用しますが、無停電電源を必要としません。完全に電気的に動作するため、ディスク ドライブよりもはるかに高速にデータにアクセスでき、損傷する可能性のある可動部品がありません。

フラッシュ ドライブのシーク時間は現在約 50 マイクロ秒で、ディスク ドライブの約 100 倍の速さです。現在のフラッシュ ドライブの転送速度は、接続されているバス インターフェイスによって異なります。高速な内部バスで接続されたフラッシュ ドライブはディスク ドライブと同等ですが、外付け USB フラッシュ ドライブはディスク ドライブよりも低速です。

フラッシュ メモリは消耗します。各バイトは一定回数書き換えることができます。限界に達したバイトに書き込もうとすると、フラッシュ ドライブが故障します。現在、この最大値は数百万で、ほとんどのデータベース アプリケーションにとって十分な高さです。ハイエンド ドライブは、頻繁に書き込まれるバイトをあまり書き込まれない場所に自動的に移動する「ウェア レベリング」技術を採用しています。この技術により、ドライブ上のすべてのバイトが書き換え限界に達するまでドライブを動作させることができます。

フラッシュ ドライブはオペレーティング システムに対してセクター ベースのインターフェイスを提供するため、フラッシュ ドライブはディスク ドライブのように見えます。フラッシュ ドライブでは RAID 技術を使用できますが、フラッシュ ドライブのシーク時間が非常に短いため、ストライピングはそれほど重要ではありません。

フラッシュ ドライブの採用を阻む主な障害は、その価格です。現在、価格は同等のディスク ドライブの約 100 倍です。フラッシュ ドライブとディスク ドライブの価格は引き続き低下しますが、最終的にはフラッシュ ドライブが主流として扱われるほど安くなるでしょう。その時点で、ディスク ドライブはアーカイブ ストレージや非常に大規模なデータベースのストレージに限定される可能性があります。

フラッシュ メモリは、永続的なフロント エンドとして機能させることでディスク ドライブを強化するためにも使用できます。データベースがフラッシュ メモリに完全に収まる場合、ディスク ドライブは

決して使用されません。ただし、データベースが大きくなるにつれて、あまり使用されないセクターがディスクに移行します。

データベース エンジンに関する限り、フラッシュ ドライブはディスク ドライブと同じ特性を持ちます。つまり、永続的で、低速で、セクター単位でアクセスされます (ディスク ドライブより低速なだけです)。したがって、本書では、現在の用語に従って、永続メモリを「ディスク」と呼ぶことにします。

3.2 ディスクへのブロックレベルインターフェース

ディスクにはさまざまなハードウェア特性があります。たとえば、セクター サイズが同じである必要はなく、セクターのアドレス指定方法も異なります。オペレーティング システムは、これらの詳細 (およびその他の詳細) を隠し、アプリケーションにディスクにアクセスするためのシンプルなインターフェイスを提供します。

ブロックの概念は、このインターフェイスの中心です。ブロックは、サイズが OS によって決定される点を除けば、セクターに似ています。各ブロックは、すべてのディスクで同じ固定サイズを持ちます。OS は、ブロックとセクター間のマッピングを維持します。また、OS は、ディスクの各ブロックにブロック番号を割り当てます。ブロック番号が指定されると、OS は実際のセクター アドレスを決定します。

ブロックの内容にディスクから直接アクセスすることはできません。代わりに、ブロックを構成するセクターをまずメモリ ページに読み込み、そこからアクセスする必要があります。ブロックの内容を変更するには、クライアントはブロックをページに読み込み、ページ内のバイトを変更し、ページをディスク上のブロックに書き戻す必要があります。

OS は通常、次のようなディスク ブロックにアクセスするためのいくつかの方法を提供します。

- readblock(n,p) は、ディスクのブロック n にあるバイトをメモリのページ p に読み込みます。
- writeblock(n,p)は、メモリのページp内のバイトをディスクのブロックn に書き込みます。
- allocate(k,n) は、ディスク上の k 個の連続した未使用ブロックを見つけ、それらを使用済みとしてマークし、最初のブロックのブロック番号を返します。新しいブロックは、ブロック n にできるだけ近い位置に配置する必要があります。
- deallocate(k,n) は、ブロック n から始まる k 個の連続するブロックを未使用としてマークします。

OS は、ディスク上のどのブロックが割り当て可能で、どのブロックが割り当て不可能であるかを追跡します。採用できる基本的な戦略は、ディスク マップと空きリストの 2 つです。

ディスク マップはビット シーケンスで、ディスク上の各ブロックに 1 ビットが割り当てられます。ビット値が 1 の場合、ブロックは空いており、0 の場合、ブロックは既に割り当てられています。ディスク マップは、通常、ディスクの最初の数ブロックに格納されます。OS は、ディスク マップのビット n を 1 に変更するだけで、ブロック n の割り当てを解除できます。ディスク マップで、値が 1 である行の k ビットを検索し、それらのビットを 0 に設定することで、連続する k ブロックを割り当てることができます。

フリーリストはチャンクの連鎖であり、チャンクは連続した未割り当てブロックのシーケンスです。各チャンクの最初のブロックには、2つの値が格納されます。

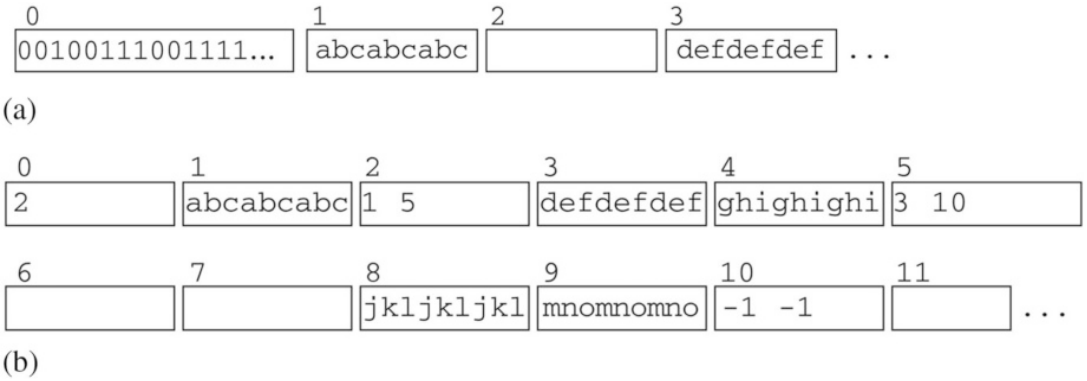


図3.6 空きブロックを追跡する2つの方法。(a) ディスクマップ、(b) 空きリスト

チャンクと、チェーン上の次のチャンクのブロック番号です。[4] ディスクの最初のブロックには、チェーン上の最初のチャンクへのポインタが含まれています。OS は、連続する k 個のブロックを割り当てるように要求されると、空きリストで十分な大きさのチャンクを検索します。次に、空きリストからチャンク全体を削除して割り当てるか、長さ k の断片に分割してそれらのブロックのみを割り当てるかを選択できます。ブロックのチャンクの割り当てを解除するように要求されると、OS はそれを空きリストに挿入するだけです。

図 3.6 は、ブロック 0、1、3、4、8、9 が割り当てられているディスクに対するこれら 2 つの手法を示しています。部分 (a) は、ディスクのブロック 0 に格納されているディスク マップを示しています。ビット値 0 は、割り当てられたブロックを示します。部分 (b) は、対応するフリー リストを示しています。ブロック 0 には値 2 が含まれており、フリー リストの最初のチャンクがブロック 2 から始まることを意味します。ブロック 2 には 2 つの値 1 と 5 が含まれており、チャンクに 1 つのブロックが含まれ、次のチャンクがブロック 5 から始まることを示します。同様に、ブロック 5 の内容は、そのチャンクが 3 ブロックの長さであり、次のチャンクがブロック 10 にあることを示しています。ヌル ポインタは -1 です。

フリー リスト手法では、追加スペースは最小限で済みます。必要なのはリストの最初のブロックを指すポインタだけです。ブロック 10 の #1 の値は、それが最後のチャンクであり、残りのすべてのブロックが含まれていることを示しています。一方、ディスク マップ手法では、マップを保持するためのスペースが必要です。図 3.6a では、マップが 1 つのブロックに収まることを前提としています。ただし、一般には、複数のブロックが必要になる場合があります。演習 3.7 を参照してください。ディスク マップの利点は、ディスクの「穴」がどこにあるかを OS がより正確に把握できることです。たとえば、OS が一度に複数のブロックの割り当てをサポートする必要がある場合、ディスク マップが選択されることがよくあります。

3.3 ディスクへのファイルレベルインターフェース

OS は、ファイル システムと呼ばれる、ディスクへの別の高レベル インターフェイスを提供します。クライアントは、ファイルを名前付きのバイト シーケンスとして見ます。このレベルではブロックの概念はありません。代わりに、クライアントはファイル内の任意の位置から任意の数のバイトを読み取る (または書き込む) ことができます。

[4] Since the block is unallocated, its contents can be used by the OS for any purpose whatsoever. In this case, it is a simple matter to use the first 8 bytes of the block to store these two integers.

Java クラス RandomAccessFile は、ファイル システムに一般的な API を提供します。各 RandomAccessFile オブジェクトは、次の読み取りまたは書き込み操作が発生するバイトを示すファイル ポインターを保持します。このファイル ポインターは、seek の呼び出しによって明示的に設定できます。readInt (または writeInt) メソッドの呼び出しによっても、ファイル ポインターは読み取った (または書き込んだ) 整数を超えて移動します。

例として、図 3.7 のコード フラグメントがあります。これは、ファイル「junk」の 7992 ～ 7995 バイト目に格納されている整数を増分します。readInt の呼び出しは、7992 バイト目の整数を読み取り、ファイル ポインタをその先の 7996 バイト目に移動します。その後の seek の呼び出しは、ファイル ポインタを 7992 バイト目に戻し、その位置の整数を上書きできるようにします。

readInt および writeInt の呼び出しは、ディスクに直接アクセスしているかのように動作し、ディスク ブロックはページを介してアクセスする必要があるという事実を隠します。通常、OS は独自の使用のためにメモリのいくつかのページを予約します。これらのページは I/O バッファと呼ばれます。ファイルが開かれると、OS はクライアントに知られずにファイルに I/O バッファを割り当てます。

ファイル レベル インターフェイスにより、ファイルをブロックのシーケンスとして考えることができます。たとえば、ブロックの長さが 4096 バイト (つまり 4K バイト) の場合、バイト 7992 はファイルのブロック 1 (つまり 2 番目のブロック) にあります。「ファイルのブロック 1」のようなブロック参照は、ブロックがファイル内のどこにあるかは示しますが、ディスク上のどこにあるかは示さないため、論理ブロック参照と呼ばれます。

特定のファイルの場所を指定すると、seek メソッドはその場所を保持する実際のディスク ブロックを決定します。特に、seek は次の 2 つの変換を実行します。

- 指定されたバイト位置を論理ブロック参照に変換します。
- 論理ブロック参照を物理ブロック参照に変換します。

最初の変換は簡単です。論理ブロック番号は、バイト位置をブロック サイズで割ったものにすぎません。たとえば、4K バイトのブロックを想定すると、7992/4096 ¼ 1 (整数除算) なので、バイト 7992 はブロック 1 にあります。

2 番目の変換はより困難で、ファイル システムの実装方法によって異なります。このセクションの残りの部分では、連続割り当て、エクステントベースの割り当て、およびインデックス割り当てという 3 つのファイル実装戦略について説明します。これら 3 つの戦略はそれぞれ、ディスク上のファイル位置に関する情報をファイル システム ディレクトリに格納します。シーク メソッドは、論理ブロック参照を物理ブロック参照に変換するときに、このディレクトリのブロックにアクセスします。これらのディスク アクセスは、ファイル システムによって課せられる隠れた「オーバーヘッド」と考えることができます。オペレーティング システムはこのオーバーヘッドを最小限に抑えようとしますが、完全になくすことはできません。

```
RandomAccessFile f = new RandomAccessFile("junk", "rws");
f.seek(7992);
int n = f.readInt();
f.seek(7992);
f.writeInt(n+1);
f.close();
```

図3.7 ディスクへのファイルシステムインターフェースの使用

継続的な割り当て
連続割り当ては最も単純な戦略で、各ファイルを連続したブロックのシーケンスとして保存します。連続割り当てを実装するために、ファイル システム ディレクトリは各ファイルの長さと最初のブロックの位置を保持します。論理ブロック参照を物理ブロック参照にマッピングするのは簡単です。ファイルがディスク ブロック b で始まる場合、ファイルのブロック N はディスク ブロック b + N にあります。図 3.8 は、2 つのファイルを含むファイル システムのディレクトリを示しています。1 つはブロック 32 で始まる 48 ブロック長の「junk」というファイルで、もう 1 つはブロック 80 で始まる 16 ブロック長の「temp」というファイルの情報です。

連続割り当てには 2 つの問題があります。最初の問題は、ファイルのすぐ後に別のファイルがある場合、ファイルを拡張できないことです。図 3.8 のファイル「junk」は、このようなファイルの例です。したがって、クライアントは、必要なブロックの最大数を使用してファイルを作成する必要があり、ファイルがいっぱいでない場合にスペースが無駄になります。この問題は、内部フラグメンテーションと呼ばれます。2 番目の問題は、ディスクがいっぱいになると、割り当てられていないブロックの小さなチャンクが多数存在する可能性がありますが、大きなチャンクは存在しないことです。したがって、ディスクに十分な空き領域があっても、大きなファイルを作成できない可能性があります。この問題は、外部フラグメンテーションと呼ばれます。言い換えると、次のようになります。

- 内部の断片化とは、ファイル内の無駄なスペースのことです。
- 外部断片化とは、すべてのファイルの外側に無駄なスペースがあることです。

エクステントベースの割り当て
エクステント ベースの割り当て戦略は、連続割り当てのバリエーションで、内部と外部の両方の断片化を軽減します。この戦略では、OS はファイルを固定長のエクステントのシーケンスとして保存します。各エクステントは連続したブロックのチャンクです。ファイルは一度に 1 つのエクステントずつ拡張されます。この戦略のファイル システム ディレクトリには、ファイルごとに、ファイルの各エクステントの最初のブロックのリストが含まれます。OS が 8 ブロックのエクステントにファイルを格納しているとします。図 3.9 は、2 つのファイル「junk」と「temp」のファイル システム ディレクトリを示しています。これらのファイルのサイズは以前と同じですが、エクステントに分割されています。ファイル「junk」には 6 つのエクステントがあり、ファイル「temp」には 2 つのエクステントがあります。ファイルのブロックNを保持するディスクブロックを見つけるために、シークメソッドはファイルシステムディレクトリでそのファイルのエクステントリストを検索し、次にエクステントリストを検索して

Name	First Block	Length
junk	32	48
temp	80	16

図3.8 連続割り当てのファイルシステムディレクトリ

Name	Extents
junk	32, 480, 696, 72, 528, 336
temp	64, 8

図3.9 エクステントベースの割り当てのためのファイルシステムディレクトリ

ブロックNを含む範囲を決定し、そこからブロックの位置を計算することができます。たとえば、図3.9のファイルディレクトリを考えてみましょう。ファイル「junk」のブロック21の位置は次のように計算できます。

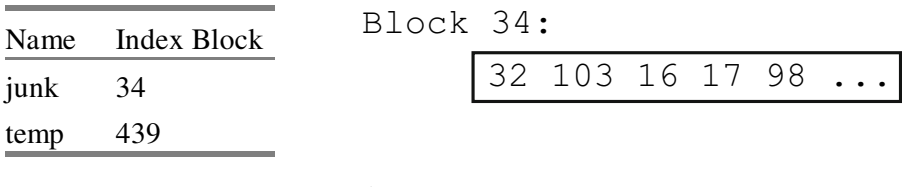
1. ブロック 21 は、21/8 ¼ 2 (整数除算) なので、ファイルのエクステント 2 にあります。2. エクステント 2 は、ファイルの論理ブロック 2-8 ¼ 16 から始まります。3. したがって、ブロック 21 はそのエクステントのブロック 21-16 ¼ 5 にあります。4. ファイルのエクステント リストによると、エクステント 2 は物理ブロック 696 から始まっています。5. したがって、ブロック 21 の位置は 696 + 5 ¼ 701 です。エクステントベースの割り当てでは、ファイルはエクステントに相当するスペースしか無駄にできないため、内部の断片化が軽減されます。また、すべてのエクステントが同じサイズであるため、外部の断片化が排除されます。

インデックス割り当て

インデックス割り当ては異なるアプローチを採用しており、ファイルを連続したチャンクで割り当てようとさえしません。代わりに、ファイルの各ブロックが個別に割り当てられます (1 ブロック長のエクステントで割り当てられます)。OS は、各ファイルに特別なインデックス ブロックを割り当てることでこの戦略を実装し、そのファイルに割り当てられたディスク ブロックを追跡します。つまり、インデックス ブロック ib は整数の配列と考えることができます。ib[N] の値は、ファイルの論理ブロック N を保持するディスク ブロックです。したがって、論理ブロックの位置を計算するのは簡単です。インデックス ブロックで調べるだけです。

図 3.10a は、2 つのファイル「junk」と「temp」のファイル システム ディレクトリを示しています。「junk」のインデックス ブロックはブロック 34 です。図 3.10b は、そのブロックの最初のいくつかの整数を示しています。この図から、ファイル「junk」のブロック 1 がディスクのブロック 103 にあることが簡単にわかります。

このアプローチの利点は、ブロックが 1 つずつ割り当てられるため、断片化が発生しないことです。主な問題は、インデックス ブロック内の値と同じ数のブロックしかファイルに格納できないため、ファイルの最大サイズが制限されることです。UNIX ファイル システムでは、複数のレベルのインデックス ブロックをサポートすることでこの問題に対処し、最大ファイル サイズを非常に大きくすることができます。演習 3.12 および 3.13 を参照してください。



Name	Index Block
junk	34
temp	439

(a)

```
Block 34:
        [ 32 103 16 17 98 ... ]
```

(b)

図3.10 インデックス割り当てのためのファイルシステムディレクトリ。(a) ディレクトリテーブル、(b) インデックスブロックの内容 34

3.4 データベースシステムとOS

OS は、ディスク アクセスに対してブロック レベルのサポートとファイル レベルのサポートという 2 つのレベルのサポートを提供します。データベース エンジンの実装者はどちらのレベルを選択すべきでしょうか。ファイル レベルのサポートを選択すると、どのディスク ブロックをどのような目的で使用するかをエンジンが完全に制御できるという利点があります。たとえば、頻繁に使用されるブロックは、シーク時間が短くなるディスクの中央に格納できます。同様に、一緒にアクセスされる傾向のあるブロックは、互いに近くに格納できます。もう 1 つの利点は、データベース エンジンがファイルに対する OS の制限に制約されないため、OS の制限よりも大きいテーブルや複数のディスク ドライブにまたがるテーブルをサポートできることです。

一方、ブロック レベル インターフェイスの使用には、次のようないくつかの欠点があります。このような戦略は実装が複雑であり、ディスクを raw ディスク (つまり、ブロックがファイル システムの一部ではないディスク) としてフォーマットしてマウントする必要があります。また、システムを微調整するには、データベース管理者がブロック アクセス パターンに関する広範な知識を持っている必要があります。

もう 1 つの極端な方法は、データベース エンジンが OS ファイル システムを可能な限り使用するというものです。たとえば、すべてのテーブルを個別のファイルに格納し、エンジンがファイル レベルの操作を使用してレコードにアクセスするという方法です。この戦略は実装がはるかに簡単で、OS がデータベース システムから実際のディスク アクセスを隠すことができます。この状況は、2 つの理由で受け入れられません。まず、データベース システムは、データを効率的に整理して取得できるように、ブロック境界がどこにあるかを知る必要があります。次に、OS の I/O バッファーの管理方法はデータベース クエリには適していないため、データベース システムは独自のページを管理する必要があります。これらの問題については、後の章で説明します。

妥協案として、データベース システムですべてのデータを 1 つ以上の OS ファイルに保存し、ファイルを生のディスクのように扱うという方法があります。つまり、データベース システムは論理ファイル ブロックを使用して「ディスク」にアクセスします。OS は、シーク メソッドを使用して、各論理ブロック参照を対応する物理ブロックにマッピングする役割を担います。シークはファイル システム ディレクトリを調べるときにディスク アクセスを引き起こす可能性があるため、データベース システムはディスクを完全に制御することはできません。ただし、これらの追加ブロックは、データベース システムがアクセスする多数のブロックと比較すると、通常は重要ではありません。したがって、データベース システムは、ディスク アクセスに対する重要な制御を維持しながら、OS への高レベルのインターフェイスを使用できます。

この妥協策は、多くのデータベース システムで使用されています。Microsoft Access はすべてを 1 つの .mdb ファイルに保存しますが、Oracle、Derby、および SimpleDB は複数のファイルを使用します。

3.5 SimpleDB ファイルマネージャ

データベース エンジンのうち、オペレーティング システムと対話する部分はファイル マネージャと呼ばれます。このセクションでは、SimpleDB のファイル マネージャについて説明します。セクション 3.5.1 では、クライアントがファイル マネージャを使用する方法について説明し、セクション 3.5.2 では、その実装について説明します。

3.5.1 ファイルマネージャの使用

SimpleDB データベースは複数のファイルに保存されます。各テーブルと各インデックスのファイル、ログ ファイル、および複数のカタログ ファイルがあります。SimpleDB ファイル マネージャーは、パッケージ simpledb.file を介してこれらのファイルへのブロック レベルのアクセスを提供します。このパッケージは、BlockId、Page、および FileMgr の 3 つのクラスを公開します。それらの API は、図 3.11 に示されています。

BlockId オブジェクトは、ファイル名と論理ブロック番号によって特定のブロックを識別します。たとえば、次の文は、

ブロックID blk = 新しいブロックID("student.tbl", 23)

BlockId

public BlockId(String filename, int blknum); public String filename(); public int number();

Page

public Page(int blocksize); public Page(byte[] b); public int getInt(int offset); public byte[] getBytes(int offset); public String getString(int offset); public void setInt(int offset, int val); public void setBytes(int offset, byte[] val); public void setString(int offset, String val); public int maxLength(int strlen);

FileMgr

public FileMgr(String dbDirectory, int blocksize); public void read(BlockId blk, Page p); public void write(BlockId blk, Page p); public BlockId append(String filename); public boolean isNew(); public int length(String filename); public int blockSize();

図3.11 SimpleDBファイルマネージャのAPI

ファイル student.tbl のブロック 23 への参照を作成します。メソッド file name と number は、そのファイル名とブロック番号を返します。

Page オブジェクトはディスク ブロックの内容を保持します。最初のコンストラクタは、オペレーティング システムの I/O バッファからメモリを取得するページを作成します。このコンストラクタはバッファ マネージャによって使用されます。2 番目のコンストラクタは、Java 配列からメモリを取得するページを作成します。このコンストラクタは主にログ マネージャによって使用されます。さまざまな get メソッドと set メソッドにより、クライアントはページの指定された場所に値を格納したり、その場所にアクセスしたりできます。ページには、int、文字列、および「blob」(任意のバイト配列) の 3 つの値タイプを保持できます。必要に応じて、追加のタイプに対応するメソッドを追加できます。演習 3.17 を参照してください。クライアントはページの任意のオフセットに値を格納できますが、どの値がどこに格納されているかを把握する必要があります。間違ったオフセットから値を取得しようとすると、予期しない結果になります。

FileMgr クラスは、OS ファイル システムの実際のやり取りを処理します。その open メソッドは、データベース名とブロック サイズを示す整数の 2 つの引数を取ります。データベース名は、データベースのファイルを含むフォルダーの名前として使用されます。このフォルダーは、エンジンの現在のディレクトリにあります。そのようなフォルダーが存在しない場合は、新しいデータベース用のフォルダーが作成されます。メソッド isNew は、この場合 true を返し、それ以外の場合は false を返します。このメソッドは、新しいデータベースを適切に初期化するために必要です。

read メソッドは、指定されたブロックの内容を指定されたページに読み込みます。write メソッドは逆の操作を実行し、ページの内容を指定されたブロックに書き込みます。length メソッドは、指定されたファイル内のブロック数を返します。

エンジンには、システムの起動時に作成される 1 つの FileMgr オブジェクトがあります。クラス SimpleDB (パッケージ simpledb.server 内) がオブジェクトを作成し、そのメソッド fileMgr が作成されたオブジェクトを返します。

図 3.12 の FileTest クラスは、これらのメソッドの使用法を示しています。このコードには 3 つのセクションがあります。最初のセクションでは、SimpleDB オブジェクトを初期化します。3 つの引数は、エンジンが 400 バイトのブロックと 8 つのバッファーのプールを使用して、「studentdb」という名前のデータベースを使用することを指定します。400 バイトのブロック サイズは、SimpleDB のデフォルトです。これは、多数のブロックを持つデモ データベースを簡単に作成できるように、意図的に小さく設定されています。商用データベース システムでは、この値はオペレーティング システムによって定義されたブロック サイズに設定されます。一般的なブロック サイズは 4K バイトです。

図 3.12 の 2 番目のセクションでは、文字列 "abcdefghijklm" を "testfile" の 2 番目のブロックの 88 番目の位置に書き込みます。次に、maxLength メソッドを呼び出して文字列の最大長を決定し、文字列に続く位置を決定します。次に、その位置に整数 345 を書き込みます。

3 番目のセクションでは、このブロックを別のページに読み込み、そこから 2 つの値を抽出します。

```
public class FileTest {
   public static void main(String[] args) throws IOException {
     SimpleDB db = new SimpleDB("filetest", 400, 8);
     FileMgr fm = db.fileMgr();

     BlockId blk = new BlockId("testfile", 2);
     Page p1 = new Page(fm.blockSize());
     int pos1 = 88;
     p1.setString(pos1, "abcdefghijklm");
     int size = Page.maxLength("abcdefghijklm".length());
     int pos2 = pos1 + size;
     p1.setInt(pos2, 345);
     fm.write(blk, p1);

     Page p2 = new Page(fm.blockSize());
     fm.read(blk, p2);
     System.out.println("offset " + pos2 +
                        " contains " + p2.getInt(pos2));
     System.out.println("offset " + pos1 +
                        " contains " + p2.getString(pos1));
   }
}
```

図3.12 SimpleDBファイルマネージャのテスト

3.5.2 ファイルマネージャの実装

このサブセクションでは、3 つのファイル マネージャー クラスの実装について説明します。

クラス BlockId
BlockId クラスのコードは図 3.13 に示されています。fileName メソッドと number メソッドの単純な実装に加えて、このクラスは equals、hashCode、t oString も実装しています。

クラスページ
Page クラスを実装するコードは、図 3.14 に示されています。各ページは 、Java ByteBuffer オブジェクトを使用して実装されています。ByteBuffer オブジェクトは、配列の任意の場所で値を読み書きするメソッドを持つバ イト配列をラップします。これらの値は、プリミティブ値 (整数など) また は小さなバイト配列にすることができます。たとえば、Page の setInt メソ ッドは、ByteBuffer の putInt メソッドを呼び出して、ページに整数を保存 します。Page の setBytes メソッドは、指定された blob のバイト数、次にバ イト自体の 2 つの値として blob を保存します。整数を書き込むために Byt eBuffer の putInt メソッドを呼び出し、バイトを書き込むためにメソッド p ut を呼び出します。

ByteBufferクラスには文字列の読み取りと書き込みを行うメソッドがないため、Pageは文字列値をBLOBとして書き込むことを選択します。JavaのStringクラスには、文字列をバイト配列に変換するgetBytesメソッドがあります。また、バイト配列を文字列に戻すコンストラクタもあります。したがって、PageのsetStringメソッドは、

パブリック クラス BlockId { private String filename; private int blknum; public BlockId(String filen ame, int blknum) { this.filename = filename; this.blknum = blknum; }public String fileName() { return filename; }public int number() { return blknum; }public boolean equals(Object obj) { BlockId blk = (BlockId) obj; return filename.equals(blk.filename) && blknum == blk.blknum; }public String toStrin g() { return "[file " + filename + ", block " + blknum + "]"; }public int hashCode() { return toString(). hashCode(); } }

図3.13 SimpleDBクラスBlockIdのコード

getBytes は文字列をバイトに変換し、それらのバイトを BLOB として書き込みます。同様に、Page の getString メソッドはバイト バッファーから BL OB を読み取り、バイトを文字列に変換します。

文字列とそのバイト表現間の変換は、文字エンコーディングによって決まります。ASCII や Unicode-16 など、いくつかの標準エンコーディングが存在します。Java Charset クラスには、これらのエンコーディングの多くを実装するオブジェクトが含まれています。String のコンストラクタとその getBytes メソッドは、Charset 引数を取ります。図 3.14 では、Page が ASCII エンコーディングを使用していることがわかりますが、CHARSET 定数を変更して、好みのエンコーディングを取得できます。

文字セットは、各文字を何バイトにエンコードするかを選択します。A SCII は 1 文字につき 1 バイトを使用するのに対し、Unicode-16 は 1 文字につき 2 バイトから 4 バイトを使用します。その結果、データベース エンジンは、特定の文字列が何バイトにエンコードされるかを正確に把握できない場合があります。Page の maxLength メソッドは、指定された文字数の文字列の BLOB の最大サイズを計算します。これは、文字数に 1 文字あたりの最大バイト数を掛け、バイトで書き込まれる整数に 4 バイトを追加することで行われます。

```
pパブリッククラス Page { プライベート ByteBuffer bb; パブリック静的最終 Charset CHA
  RSET = StandardCharsets.US_ASCII
                                                                                ;
    // データ バッファーを作成するためのコンストラクター public Page(int blocksize) {
    bb = ByteBuffer.allocateDirect(blocksize); }// ログ ページを作成するためのコンストラ
    クター public Page(byte[] b) { bb = ByteBuffer.wrap(b); }public int getInt(int offset) { retu
    rn bb.getInt(offset); }public void setInt(int offset, int n) { bb.putInt(offset, n); }public byte[]
    getBytes(int offset) { bb.position(offset); int length = bb.getInt(); byte[] b = new byte[length
    ]; bb.get(b); return b; }public void setBytes(int offset, byte[] b) { bb.position(offset); bb.putI
    nt(b.length); bb.put(b); }public String getString(int offset) { byte[] b = getBytes(offset); retu
    rn new String(b, CHARSET); }public void setString(int offset, String s) { byte[] b = s.getB
    ytes(CHARSET); setBytes(offset, b); }public static int maxLength(int strlen) { float bytesPe
    rChar = CHARSET.newEncoder().maxBytesPerChar(); return Integer.BYTES + (strlen * (i
    nt)bytesPerChar); }// パッケージのプライベートメソッド、FileMgr が必要 ByteBuffer
    contents() { bb.position(0); return bb; }

}
```

図3.14 SimpleDBクラスのコードページ

ByteBuffer オブジェクトの基になるバイト配列は、Java 配列またはオペレーティング システムの I/O バッファから取得できます。Page クラスには 2 つのコンストラクタがあり、それぞれが異なる種類の基になるバイト配列に対応しています。I/O バッファは貴重なリソースであるため、最初のコンストラクタの使用はバッファ マネージャによって慎重に制御されます。これについては次の章で説明します。データベース エンジンの他のコンポーネント (ログ マネージャなど) は、もう 1 つのコンストラクタを使用します。

FileMgr クラス

FileMgr クラスのコードは、図 3.15 に示されています。このクラスの主な役割は、ディスク ブロックにページを読み書きするメソッドを実装することです。読み取りメソッドは、指定されたファイル内の適切な位置をシークし、そのブロックの内容を指定されたページのバイト バッファーに読み取ります。書き込みメソッドも同様です。追加メソッドは、ファイルの末尾をシークし、そこに空のバイト配列を書き込みます。これにより、OS は自動的にファイルを拡張します。ファイル マネージャーが常にブロック サイズのバイト数をファイルから読み取りまたは書き込み、常にブロック境界で書き込みを行うことに注意してください。これにより、ファイル マネージャーは、読み取り、書き込み、または追加の各呼び出しで、ディスク アクセスが 1 回だけ発生するようにします。

マップ openFiles 内の各 RandomAccessFile オブジェクトは、開いているファイルに対応します。ファイルは「rws」モードで開かれることに注意してください。「rw」部分は、ファイルが読み取りと書き込み用に開かれていることを示します。「s」部分は、ディスク パフォーマンスを最適化するためにオペレーティング システムがディスク I/O を遅延させないことを指定します。代わりに、すべての書き込み操作はディスクに直ちに書き込まれる必要があります。この機能により、データベース エンジンはディスク書き込みがいつ発生するかを正確に把握できます。これは、第 5 章のデータ回復アルゴリズムを実装する際に特に重要です。

メソッド read、write、および append は同期化されています。つまり、一度に 1 つのスレッドだけがこれらのメソッドを実行できます。メソッドが RandomAccessFile オブジェクトなどの更新可能なオブジェクトを共有する場合、一貫性を維持するために同期化が必要です。たとえば、read が同期化されていない場合は、次のシナリオが発生する可能性があります。それぞれ独自のスレッドで実行されている 2 つの JDBC クライアントが、同じファイルから異なるブロックを読み取ろうとしているとします。最初にスレッド A が実行されます。read の実行を開始しますが、f.seek の呼び出し直後に中断されます。つまり、ファイルの位置は設定されていますが、まだ読み取りは実行されていません。次にスレッド B が実行され、read の実行を完了します。スレッド A が再開すると、ファイルの位置は変更されていますが、スレッド A はそれに気付かないため、間違ったブロックから誤って読み取ります。

SimpleDB には FileMgr オブジェクトが 1 つだけあります。これは、パッケージ simpledb.server の SimpleDB コンストラクタによって作成されます。FileMgr コンストラクタは、指定されたデータベース フォルダが存在するかどうかを確認し、必要に応じて作成します。また、コンストラクタは、第 13 章のマテリアライズド オペレータによって作成された可能性のある一時ファイルを削除します。

3.6 章の要約

- ディスク ドライブには、1 つ以上の回転するプラッタが含まれています。プラッタには同心円状のトラックがあり、各トラックはセクターで構成されています。セクターのサイズはディスクの製造元によって決定されます。一般的なセクター サイズは 512 バイトです。

パブリック クラス FileMgr { private File dbDirectory; private int blocksize; private boolean isNew ; private Map<String,RandomAccessFile> openFiles = new HashMap<> ( ) ;

public FileMgr(File dbDirectory, int blocksize) { this.dbDirectory = dbDirectory; this.bloc ksize = blocksize; isNew = !dbDirectory.exists(); // データベースが新規の場合はディ レクトリを作成します if (isNew) dbDirectory.mkdirs(); // 残っている一時テーブルを 削除します for (String filename : dbDirectory.list()) if (filename.startsWith("temp")) new File(dbDirectory, filename).delete(); }public synchronized void read(BlockId blk, Page p) { try {RandomAccessFile f = getFile(blk.fileName()); f.seek(blk.number() * blocksize); f. getChannel().read(p.contents()); }catch (IOException e) { throw new RuntimeException(" cannot read block " + blk); } }public synchronized void write(BlockId blk, Page p) { try { RandomAccessFile f = getFile(blk.fileName()); f.seek(blk.number() * blocksize); f.getCha nnel().write(p.contents()); }catch (IOException e) { throw new RuntimeException("ブロ ックを書き込めません" + blk); } }public synchronized BlockId append(String filename) { int newblknum = size(filename); BlockId blk = new BlockId(filename, newblknum); byt e[] b = new byte[blocksize]; try {RandomAccessFile f = getFile(blk.fileName()); f.seek(bl k.number() * blocksize); f.write(b); }

図3.15 SimpleDBクラスFileMgrのコード

```
        catch (IOException e) { throw new RuntimeException("ブロックを追加できません"
    + blk); }return blk; }public int length(String filename) { try {RandomAccessFile f = getFile
    (filename);

                return (int)(f.length() / blocksize); }catch (IOException e) { throw new RuntimeEx
    ception("cannot access " + filename); } }public boolean isNew() { return isNew; }public int
    blockSize() { return blocksize; }private RandomAccessFile getFile(String filename) throws
    IOException { RandomAccessFile f = openFiles.get(filename); if (f == null) { File dbTable
    = new File(dbDirectory, filename); f = new RandomAccessFile(dbTable, "rws"); openFiles.
    put(filename, f); }return f; }

}
```

図3.15（続き）

- 各プラッターには独自の読み取り/書き込みヘッドがあります。これらのヘッドは独立して動くのではなく、すべて 1 つのアクチュエータに接続されており、各プラッターの同じトラックに同時に移動されます。
- ディスク ドライブは、ディスク アクセスを 3 段階で実行します。 – アクチュエータがディスク ヘッドを指定されたトラックに移動します。この時間をシーク時間と呼びます。 – ドライブは、プラッターが回転して最初の目的のバイトがディスク ヘッドの下になるまで待機します。この時間を回転遅延と呼びます。 – ディスク ヘッドの下で回転するバイトが読み取られます (または書き込まれます)。この時間を転送時間と呼びます。

- ディスクドライブは機械的に動作するため、速度が遅くなります。ディスクキャッシュ、シリンダ、ディスクストライピングを使用することでアクセス時間を改善できます。ディスクキャッシュにより、

ディスクは、一度にトラック全体を読み取ることによってセクターをプリフェッチします。シリンダは、各プラッタ上の同じトラック番号を持つトラックで構成されます。同じシリンダ上のブロックには、追加のシーク時間なしでアクセスできます。ディスク ストライピングは、仮想ディスクの内容を複数の小さなディスクに分散します。小さなディスクが同時に動作できるため、速度が向上します。

- RAID 技術を使用すると、ディスクの信頼性を向上させることができます。基本的な RAID レベルは次のとおりです。

  – RAID-0 はストライピングであり、追加の信頼性はありません。ディスクに障害が発生すると、データベース全体が事実上破壊されます。
  – RAID-1 は、ストライピングされたディスクにミラーリングを追加します。各ディスクには、同一のミラー ディスクがあります。ディスクに障害が発生した場合、そのミラーを使用してディスクを再構築できます。 – RAID-4 は、追加のディスクを使用してストライピングを使用し、冗長パリティ情報を保持します。ディスクに障害が発生した場合、他のディスクの情報とパリティ ディスクを組み合わせることで、ディスクの内容を再構築できます。
- RAID 技術では、コントローラが複数のディスクの存在をオペレーティング システムから隠し、単一の仮想ディスクがあるように見せる必要があります。コントローラは、各仮想読み取り/書き込み操作を、基礎となるディスク上の 1 つ以上の操作にマッピングします。
- ディスク技術はフラッシュ メモリの脅威にさらされています。フラッシュ メモリは永続的ですが、完全に電子化されているためディスクよりも高速です。ただし、フラッシュは RAM よりも大幅に遅いため、オペレーティング システムはフラッシュ ドライブをディスク ドライブと同じように扱います。
- オペレーティング システムは、ブロック ベースのインターフェイスを提供することで、ディスク ドライブとフラッシュ ドライブの物理的な詳細をクライアントから隠します。ブロックはセクターに似ていますが、サイズが OS によって定義される点が異なります。クライアントは、ブロック番号でデバイスの内容にアクセスします。OS は、ディスク マップまたは空きリストを使用して、ディスク上のどのブロックが割り当て可能かを追跡します。
- ページ フレームは、RAM のメモリ領域です。クライアントは、ブロックの内容をページに読み込み、ページを変更し、ページをブロックに書き戻すことでブロックを変更します。
- オペレーティング システムは、ディスクへのファイル レベルのインターフェイスも提供します。クライアントは、ファイルを名前付きのバイト シーケンスとして表します。
- オペレーティング システムは、連続割り当て、エクステント ベースの割り当て、またはインデックス割り当てを使用してファイルを実装できます。連続割り当てでは、各ファイルが連続したブロックのシーケンスとして保存されます。エクステント ベースの割り当てでは、ファイルがエクステントのシーケンスとして保存されます。各エクステントは、連続したブロックのチャンクです。インデックス割り当てでは、ファイルの各ブロックが個別に割り当てられます。各ファイルには、そのファイルに割り当てられたディスク ブロックを追跡するための特別なインデックス ブロックまたはページが保持されます。
- ファイル システムでは、ブロック レベルとファイル レベルのどちらかを選択できます。妥協案としては、データをファイルに保存し、ファイルにはブロック レベルでアクセスするという方法が考えられます。

3.7 推奨される読み物

Chen ら (1994) の記事では、さまざまな RAID 戦略とそのパフォーマンス特性について詳細な調査が提供されています。UNIX ベースのファイル システムについては、von Hagen (2002) が、Windows NTFS については Nagar (1997) が優れた書籍です。さまざまなファイル システムの実装の概要は、Silberschatz ら (2004) などの多くのオペレーティング システムの教科書に記載されています。

フラッシュメモリには、既存の値を上書きすると、まったく新しい値を書き込む場合よりも大幅に遅くなるという特性があります。そのため、値を上書きしないフラッシュ ベースのファイル システムに関する研究が数多く行われてきました。このようなファイル システムでは、第 4 章のログと同様に、更新がログに保存されます。Wu と Kuo (2006) および Lee と Moon (2007) の記事では、これらの問題について検討しています。

Chen, P., Lee, E., Gibson, G., & Patterson, D. (1994) RAID: 高性能で信頼性の高いセカンダリ ストレージ。ACM Computing Surveys、26(2)、145 – 185。Lee, S.、Moon, B. (2007) フラッシュ ベース DBMS の設計: ページ内ログ記録アプローチ。ACM-SIGMOD カンファレンスの議事録、pp. 55 – 66。Nagar, R. (1997) Windows NT ファイル システムの内部。O ’ Reilly。Silberschatz, A.、Gagne, G.、Galvin, P. (2004) オペレーティング システムの概念。Addison Wesley。von Hagen, W. (2002) Linux ファイル システム。Sams Publishing。 Wu, C., & Kuo, T. (2006) フラッシュメモリ上のログベースファイルシステムの効率的な初期化とクラッシュ回復の設計。ACM Transactions on Storage、2(4)、449 – 467。

3.8 演習

概念演習

3.1. 50,000 トラックを含み、7200 rpm で回転するシングルプラッタ ディスクを考えます。各トラックには 500 セクターが含まれ、各セクターには 512 バイトが含まれます。

(a) プラッターの容量はどれくらいですか? (b) 平均回転遅延はどれくらいですか? (c) 最大転送速度はどれくらいですか?

3.2. 転送速度 100 MB/秒で 7200 rpm で回転する 80 GB のディスク ドライブを考えます。各トラックには同じ数のバイトが含まれていると仮定します。

(a) 各トラックには何バイト含まれていますか? ディスクにはトラックがいくつありますか? (b) ディスクが 10,000 rpm で回転している場合、転送速度はどれくらいになりますか?

3.3. 20 GBのディスクドライブが10台あり、各ディスクに1トラックあたり500セクターあるとします。ストライピングによって仮想200 GBドライブを作成するとします。

小さなディスクで、各ストライプのサイズは単一のセクターではなく、トラック全体になります。

(a) コントローラが仮想セクター M の要求を受信したとします。対応する実際のドライブとセクター番号を計算する式を示します。 (b) トラック サイズのストライプがセクター サイズのストライプよりも効率的である理由を示します。 (c) トラック サイズのストライプがセクター サイズのストライプよりも効率が悪い理由を示します。

3.4. この章で説明する障害回復手順では、障害が発生したディスクを交換する間、システムをシャットダウンする必要があります。多くのシステムでは、ダウンタイムを許容できず、また、データを失うことも望んでいません。

(a) 基本的なミラーリング戦略について考えます。ダウンタイムなしで障害が発生したミラーを復元するアルゴリズムを示します。このアルゴリズムによって、2 番目のディスク障害のリスクが高まりますか? このリスクを軽減するには、何をする必要がありますか? (b) 同様にダウンタイムを排除するためにパリティ戦略を変更します。2 番目のディスク障害のリスクにはどのように対処しますか?

3.5. RAID-4 パリティ戦略の結果の 1 つは、ディスク書き込み操作ごとにパリティ ディスクがアクセスされることです。提案されている改善策の 1 つは、パリティ ディスクを省略し、代わりにデータ ディスクをパリティ情報で「ストライプ化」することです。たとえば、ディスク 0 のセクター 0、N、2N などにパリティ情報が含まれ、ディスク 1 のセクター 1、N + 1、2N + 1 などにパリティ情報が含まれ、以下同様です。この改善策は RAID-5 と呼ばれます。

(a) ディスクに障害が発生したとします。どのように回復するかを説明してください。(b) この改善により、ディスクの読み取りと書き込みに必要なディスク アクセスの数は RAID-4 と同じであることを示します。(c) それでも、この改善によってディスク アクセスの効率が向上する理由を説明してください。

3.6. 図 3.5 を検討し、ストライプ ディスクの 1 つに障害が発生したと仮定します。パリティ ディスクを使用してその内容を再構築する方法を示します。

3.7. ブロックサイズが 4K バイトのファイルに保存されている 1 GB のデータベースを考えます。

(a) ファイルにはいくつのブロックが含まれますか? (b) データベース システムがディスク マップを使用して空きブロックを管理するとします。ディスク マップを保持するために必要な追加ブロックの数はいくつですか?

3.8. 図3.6を検討してください。次の操作が実行された後のディスクマップとフリーリストの図を描きます。

割り当てる(1,4); 割り当てる(4,10); 割り当てる(5,12);

3.9. 図 3.16 は、ディスクの 1 つに障害が発生した RAID-4 システムを示しています。パリティ ディスクを使用して値を再構築します。

3.10. フリー リスト割り当て戦略により、フリー リスト上に 2 つの連続したチャンクが作成されることがあります。

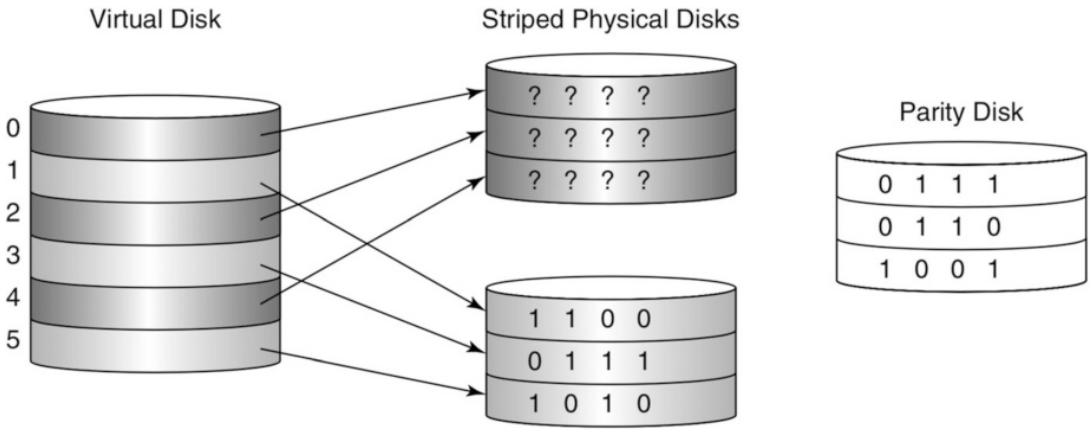


図3.16 RAID-4システムで故障した物理ディスク

(a) 連続したチャンクをマージできるようにフリーリスト手法を変更する方法を説明してください。 (b) ファイルが連続的に割り当てられている場合、割り当てられていないチャンクをマージすることがなぜ良いアイデアなのかを説明してください。 (c) エクステントベースまたはインデックス付きファイル割り当てではマージが重要でない理由を説明してください。

3.11. OSがサイズ12のエクステントを使用したエクステントベースのファイル割り当てを使用し、ファイルのエクステントリストが[240、132、60、252、12、24]であると仮定します。

(a) ファイルのサイズはどれくらいですか? (b) ファイルの論理ブロック 2、12、23、34、および 55 の物理ディスク ブロックを計算します。

3.12. インデックス付きファイル割り当てを使用するファイル実装を考えてみましょう。ブロック サイズが 4K バイトであると仮定すると、最大ファイルのサイズはいくらですか。

3.13. UNIX では、ディレクトリ エントリは、inode と呼ばれるブロックを指します。inode の 1 つの実装では、ブロックの先頭にさまざまなヘッダー情報が格納され、最後の 60 バイトに 15 個の整数が格納されます。これらの整数の最初の 12 個は、ファイル内の最初の 12 個のデータ ブロックの物理的な位置です。次の 2 つの整数は 2 つのインデックス ブロックの位置で、最後の整数はダブル インデックス ブロックの位置です。インデックス ブロックは、ファイル内の次のデータ ブロックのブロック番号のみで構成され、ダブル インデックス ブロックは、インデックス ブロック (その内容はデータ ブロックを指します) のブロック番号のみで構成されます。

(a) ブロック サイズが 4K バイトであると仮定すると、インデックス ブロックはいくつのデータ ブロックを参照しますか? (b) ダブル インデックス ブロックを無視すると、UNIX ファイルの最大サイズはどれくらいですか? (c) ダブル インデックス ブロックはいくつのデータ ブロックを参照しますか? (d) UNIX ファイルの最大サイズはどれくらいですか?

(e) 1 GB のファイルの最後のデータ ブロックを読み取るには、何回のブロック アクセスが必要ですか。 (f) UNIX ファイルのシーク機能を実装するアルゴリズムを示します。

3.14. 映画や歌のタイトル「晴れた日には永遠に見える」は、時々「透明なディスクには永遠に見えるものがある」と誤って引用されます。この語呂合わせの巧妙さと正確さについてコメントしてください。

プログラミング演習

3.15. データベース システムには、多くの場合、診断ルーチンが含まれています。

(a) FileMgr クラスを変更して、読み書きされたブロック数などの有用な統計情報を保持するようにします。これらの統計情報を返す新しいメソッドをクラスに追加します。 (b) RemoteConnectionImpl クラス (simpledb.jdbc.network パッケージ内) の commit メソッドと rollback メソッドを変更して、これらの統計情報を出力するようにします。EmbeddedConnection クラス (simpledb.jdbc.embedded パッケージ内) に対しても同じ操作を行います。その結果、エンジンは実行する各 SQL ステートメントの統計情報を出力します。

3.16. Page クラスのメソッド setInt、setBytes、setString は、新しい値がページに収まるかどうかをチェックしません。

(a) チェックを実行するようにコードを変更します。チェックが失敗した場合はどうすればよいですか? (b) チェックを実行しないのが妥当な理由を示してください。

3.17. Page クラスには、整数、BLOB、文字列を取得/設定するメソッドがあります。短整数、ブール値、日付などの他の型を処理できるようにクラスを変更します。

3.18. Page クラスは、文字列の文字から blob を作成することで文字列を実装します。文字列を実装する別の方法は、各文字を個別に記述し、最後に区切り文字を追加することです。Java で適切な区切り文字は '\0' です。クラスをそれに応じて変更します。

第4章 メモリ管理

この章では、データベース エンジンの 2 つのコンポーネント、ログ マネージャーとバッファー マネージャーについて説明します。これらの各コンポーネントは特定のファイルを担当します。ログ マネージャーはログ ファイルを担当し、バッファー マネージャーはデータ ファイルを担当します。両方のコンポーネントは、メイン メモリを使用してディスク ブロックの読み取りと書き込みを効率的に管理する方法という問題に直面しています。データベースの内容は通常、メイン メモリよりはるかに大きいため、これらのコンポーネントはメモリのブロックを出し入れする必要がある場合があります。この章では、これらのコンポーネントのメモリ要件と、使用するメモリ管理アルゴリズムについて説明します。ログ マネージャは、ログ ファイルへのシーケンシャル アクセスのみをサポートし、シンプルで最適なメモリ管理アルゴリズムを備えています。一方、バッファ マネージャは、ユーザー ファイルへの任意のアクセスをサポートする必要があり、これははるかに困難な課題です。

4.1 データベースメモリ管理の2つの原則

データベース エンジンがディスク値を読み取る唯一の方法は、その値を含むブロックをメモリのページに読み込むことであり、ディスク値を書き込む唯一の方法は、変更されたページをそのブロックに書き戻すことであることを思い出してください。データベース エンジンは、ディスクとメモリ間でデータを移動するときに、ディスク アクセスを最小限に抑え、仮想メモリに依存しないという 2 つの重要な原則に従います。

原則1: ディスクアクセスを最小限に抑える

ディスクからデータを読み取り、データを検索し、さまざまな計算を実行し、変更を加えて、データを書き戻すアプリケーションを考えてみましょう。これにかかる時間をどのように見積もることができますか? RAM操作はフラッシュの1000倍以上、ディスクの100,000倍以上高速であることを思い出してください。これは、ほとんどの実用的な状況では、ディスクからブロックを読み書きするのにかかる時間は以下のとおりであることを意味します。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_4

少なくとも、RAM 内のブロックを処理するのにかかる時間と同じだけ大きくなります。したがって、データベース エンジンが実行できる最も重要なことは、ブロック アクセスを最小限に抑えることです。

ブロック アクセスを最小限に抑える方法の 1 つは、ディスク ブロックに複数回アクセスしないようにすることです。この種の問題はコンピューティングの多くの領域で発生し、キャッシュと呼ばれる標準的な解決策があります。たとえば、CPU には、以前に実行された命令のローカル ハードウェア キャッシュがあります。次の命令がキャッシュ内にある場合、CPU はそれを RAM からロードする必要はありません。別の例として、ブラウザーは以前アクセスした Web ページのキャッシュを保持しています。ユーザーがキャッシュ内にあるページを要求した場合 (ブラウザーの [戻る] ボタンを押すなど)、ブラウザーはネットワークからそのページを取得することを回避できます。

データベース エンジンは、メモリ ページを使用してディスク ブロックをキャッシュします。どのページにどのブロックの内容が含まれているかを追跡することで、エンジンは既存のページを使用してクライアントの要求を満たすことができ、ディスクの読み取りを回避できます。同様に、エンジンは、1 回のディスク書き込みでページに対する複数の変更を行えるように、必要な場合にのみページをディスクに書き込みます。

ディスク アクセスを最小限に抑える必要性は非常に重要なので、データベース エンジンの実装全体に影響を及ぼします。たとえば、エンジンが使用する検索アルゴリズムは、ディスクへのアクセスが効率的であるために特に選択されます。また、SQL クエリに複数の検索戦略が考えられる場合、プランナーはディスク アクセスの回数が最も少なくなると思われる戦略を選択します。

原則2: 仮想メモリに頼らない

最新のオペレーティング システムは仮想メモリをサポートしています。オペレーティング システムは、各プロセスに、コードとデータを保存するための非常に大きなメモリがあるという錯覚を与えます。プロセスは仮想メモリ空間にオブジェクトを任意に割り当てることができ、オペレーティング システムは各仮想ページを物理メモリの実際のページにマップします。

オペレーティング システムでサポートされる仮想メモリ スペースは、通常、コンピュータの物理メモリよりはるかに大きいです。すべての仮想ページが物理メモリに収まるわけではないため、OS はそれらの一部をディスクに保存する必要があります。プロセスがメモリにない仮想ページにアクセスすると、ページ スワップが発生します。OS は物理ページを選択し、そのページの内容をディスクに書き込み (変更されている場合)、ディスクからそのページに仮想ページの保存された内容を読み取ります。

データベース エンジンがディスク ブロックを管理する最も簡単な方法は、各ブロックに独自の仮想ページを割り当てることです。たとえば、ファイルごとにページの配列を保持し、ファイルの各ブロックに 1 つのスロットを割り当てることができます。これらの配列は巨大になりますが、仮想メモリに収まります。データベース システムがこれらのページにアクセスすると、仮想メモリ メカニズムは必要に応じてディスクと物理メモリ間でページを交換します。これはシンプルで簡単に実装できる戦略です。ただし、深刻な問題があります。それは、ページがディスクに書き込まれるタイミングをデータベース エンジンではなくオペレーティング システムが制御することです。2 つの問題が発生します。

最初の問題は、オペレーティング システムのページ スワッピング戦略によって、システム クラッシュ後のデータベース エンジンの回復能力が損なわれる可能性があることです。その理由は、第 5 章で説明するように、変更されたページには関連するログ レコードがいくつかあり、これらのログ レコードはページより前にディスクに書き込まれなければならないためです。(そうしないと、ログ レコードはシステム クラッシュ後のデータベースの回復に使用できなくなります。) OS では、

ログについて知らない場合、ログレコードを書き込まずに変更されたページをスワップアウトし、回復メカニズムを無効にする可能性があります。[1]

2 つ目の問題は、オペレーティング システムが現在使用中のページと、データベース エンジンがもはや気にしないページを認識していないことです。OS は、最も最近アクセスされていないページをスワップするなど、知識に基づいた推測を行うことができます。しかし、OS の推測が間違っていると、再び必要になるページをスワップアウトしてしまい、不要なディスク アクセスが 2 回発生します。一方、データベース エンジンは、必要なページをより正確に認識しており、よりインテリジェントな推測を行うことができます。

したがって、データベース エンジンは独自のページを管理する必要があります。これは、物理メモリに収まることがわかっている比較的少数のページを割り当てることによって行われます。これらのページは、データベースのバッファー プールと呼ばれます。エンジンは、スワップ可能なページを追跡します。ブロックをページに読み込む必要がある場合、データベース エンジン (オペレーティング システムではありません) がバッファー プールから使用可能なページを選択し、必要に応じてその内容 (およびログ レコード) をディスクに書き込み、その後で指定されたブロックを読み取ります。

4.2 ログ情報の管理

ユーザーがデータベースを変更するたびに、データベース エンジンは、元に戻す必要がある場合に備えて、その変更を追跡する必要があります。変更を示す値はログ レコードに保持され、ログ レコードはログ ファイルに保存されます。新しいログ レコードは、ログの末尾に追加されます。

ログ マネージャは、ログ ファイルにログ レコードを書き込む役割を担うデータベース エンジン コンポーネントです。ログ マネージャはログ レコードの内容を理解しません。その役割は、第 5 章のリカバリ マネージャが担います。ログ マネージャは、ログを、増え続けるログ レコードのシーケンスとして扱います。

このセクションでは、ログ マネージャがログ ファイルにログ レコードを書き込むときにメモリを管理する方法について説明します。ログにレコードを追加する最も簡単な方法である図 4.1 のアルゴリズムを検討します。 このアルゴリズムでは、追加されたログレコードごとにディスクの読み取りと書き込みが必要になります。これは単純ですが、非常に非効率的です。図4.2は、ログレコードの処理を示しています。

1. Allocate a page in memory.
2. Read the last block of the log file into that page.
3a. If there is room, place the log record after the other records on the page,
    and write the page back to disk.
3b. If there is no room, then allocate a new, empty page, place the log record in that page,
    and append the page to a new block at the end of the log file.

図4.1 ログに新しいレコードを追加するための単純な（しかし非効率的な）アルゴリズム

[1] Actually, there do exist operating systems that address this issue, but they are not commonplace.

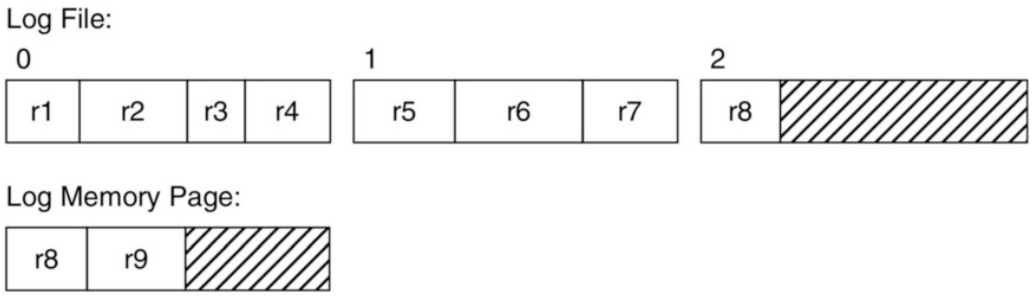


図4.2 新しいログレコードr9の追加

マネージャーはアルゴリズムのステップ 3a の途中でログ ファイルを処理し終えました。ログ ファイルには、r1 から r8 のラベルが付いた 8 つのレコードを保持する 3 つのブロックが含まれています。ログ レコードのサイズはさまざまであるため、ブロック 0 には 4 つのレコードが収まりますが、ブロック 1 には 3 つしか収まりません。ブロック 2 はまだいっぱいではなく、レコードは 1 つだけです。メモリ ページにはブロック 2 の内容が含まれています。レコード r8 に加えて、新しいログ レコード (レコード r9) がページに配置されました。

ここでログ マネージャーがページをファイルのブロック 2 に書き戻すことでアルゴリズムを完了するとします。ログ マネージャーが最終的にファイルに別のログ レコードを追加するように要求されると、アルゴリズムの手順 1 と 2 が実行され、ブロック 2 がページに読み込まれます。ただし、既存のログ ページにはすでにブロック 2 の内容が含まれているため、このディスク読み取りはまったく不要であることに注意してください。したがって、アルゴリズムの手順 1 と 2 は不要です。ログ マネージャーは、最後のログ ブロックの内容を格納するページを永続的に割り当てるだけで済みます。その結果、ディスク読み取りはすべて不要になります。

ディスクへの書き込みを減らすこともできます。上記のアルゴリズムでは、ログ マネージャは、新しいレコードがページに追加されるたびに、そのページをディスクに書き込みます。図 4.2 を見ると、レコード r9 をすぐにディスクに書き込む必要がないことがわかります。ページに空きがある限り、新しいログ レコードはそれぞれページに追加できます。ページがいっぱいになると、ログ マネージャはページをディスクに書き込み、その内容をクリアして、新たに開始できます。この新しいアルゴリズムでは、ログ ブロックごとに 1 回のディスク書き込みで済み、明らかに最適です。

このアルゴリズムには 1 つの問題があります。ログ マネージャの制御外の状況により、ログ ページがいっぱいになる前にディスクに書き込む必要がある場合があります。問題は、そのページに関連付けられたログ レコードがディスクに書き込まれるまで、バッファー マネージャは変更されたデータ ページをディスクに書き込むことができないことです。それらのログ レコードの 1 つがログ ページ内にあってもまだディスク上にない場合は、ページがいっぱいかどうかに関係なく、ログ マネージャはそのページをディスクに書き込む必要があります。この問題については、第 5 章で説明します。

図 4.3 は、結果として得られるログ管理アルゴリズムを示しています。このアルゴリズムでは、メモリ ページがディスクに書き込まれる場所が 2 つあります。ログ レコードを強制的にディスクに書き込む必要がある場合と、ページがいっぱいになった場合です。その結果、メモリ ページが同じログ ブロックに複数回書き込まれる可能性があります。ただし、これらのディスク書き込みは絶対に必要であり、回避できないため、このアルゴリズムは最適であると結論付けることができます。

1. ログ ファイルの最後のブロックの内容を保持するために、1 つのメモリ ページを永続的に割り当てます。このページを P と呼びます。2. 新しいログ レコードが送信された場合: a) P に空きがない場合: P をディスクに書き込み、その内容をクリアします。 b) 新しいログ レコードを P に追加します。 3. データベース システムが特定のログ レコードをディスクに書き込むように要求した場合: a) そのログ レコードが P にあるかどうかを判断します。 b) ある場合は、P をディスクに書き込みます。

図4.3 最適なログ管理アルゴリズム

4.3 SimpleDB ログ マネージャー

このセクションでは、SimpleDB データベース システムのログ マネージャーについて説明します。セクション 4.3.1 では、ログ マネージャーの使用方法を説明します。セクション 4.3.2 では、その実装について説明します。

4.3.1 ログマネージャのAPI

SimpleDB ログ マネージャーの実装は、パッケージ simpledb.log にあります。このパッケージは、図 4.4 に示す API を持つクラス LogMgr を公開します。データベース エンジンには、システムの起動時に作成される LogMgr オブジェクトが 1 つあります。コンストラクターへの引数は、ファイル マネージャーへの参照とログ ファイルの名前です。

メソッド append は、ログにレコードを追加し、整数を返します。ログ マネージャにとって、ログ レコードは任意のサイズのバイト配列です。ログ マネージャは、配列をログ ファイルに保存しますが、その内容が何を意味するかは知りません。唯一の制約は、配列がページ内に収まらなければならないことです。append からの戻り値は、新しいログ レコードを識別します。この識別子は、ログ シーケンス番号 (または LSN) と呼ばれます。ログにレコードを追加しても、そのレコードがディスクに書き込まれることは保証されません。その代わりに、図 4.3 のアルゴリズムのように、ログ マネージャがログ レコードをディスクに書き込むタイミングを選択します。クライアントは、メソッド flush を呼び出すことによって、特定のログ レコードを強制的にディスクに書き込むことができます。flush の引数は、ログ レコードの LSN です。このメソッドは、このログ レコード (および以前のすべてのログ レコード) がディスクに書き込まれることを保証します。

クライアントは、ログ内のレコードを読み込むためにメソッド iterator を呼び出します。このメソッドは、ログ レコードの Java iterator を返します。iterator の次のメソッドを呼び出すたびに、

LogMgr

パブリック LogMgr(FileMgr fm、String logfile); パブリック int append(byte[] rec); パブリック void flush(int lsn); パブリック Iterator<byte[]> iterator();

図4.4 SimpleDBログマネージャのAPI

ログ内の次のレコードを示すバイト配列を返します。イテレータ メソッドによって返されるレコードは逆順で、最新のレコードから始まり、ログ ファイルを逆方向に移動します。レコードがこの順序で返されるのは、リカバリ マネージャーがレコードを表示する順序だからです。

図 4.5 のクラス LogTest は、ログ マネージャ API の使用方法の例を示しています。このコードは、それぞれ文字列と整数で構成される 70 個のログ レコードを作成します。整数はレコード番号 N で、文字列は値「record N」です。このコードは、最初の 35 個のレコードが作成された後に 1 回レコードを出力し、70 個すべてが作成された後にもう一度レコードを出力します。

コードを実行すると、printLogRecords の最初の呼び出し後には 20 レコードのみが印刷されることがわかります。これは、これらのレコードが最初のログ ブロックに書き込まれ、21 番目のログ レコードの作成時にディスクにフラッシュされたためです。他の 15 のログ レコードはメモリ内のログ ページに残り、フラッシュされませんでした。createRecords の 2 番目の呼び出しでは、レコード 36 から 70 が作成されます。flush の呼び出しでは、ログ マネージャーにレコード 65 がディスク上にあることを確認するように指示します。ただし、レコード 66 から 70 はレコード 65 と同じページにあるため、これらもディスクに書き込まれます。その結果、printLogRecords の 2 番目の呼び出しでは、70 レコードすべてが逆順序で印刷されます。

createLogRecord メソッドがログ レコードとなるバイト配列を割り当てる方法に注目してください。このメソッドは、その配列をラップする Page オブジェクトを作成し、ページの setInt メソッドと setString メソッドを使用して、文字列と整数をログ レコード内の適切なオフセットに配置できるようにします。次に、コードはバイト配列を返します。同様に、printLogRecords メソッドは、ログ レコードをラップする Page オブジェクトを作成し、レコードから文字列と整数を抽出できるようにします。

4.3.2 ログマネージャの実装

LogMgr のコードは図 4.6 に示されています。コンストラクタは、指定された文字列をログ ファイルの名前として使用します。ログ ファイルが空の場合、コンストラクタは新しい空のブロックを追加します。コンストラクタは、1 つのページ (logpage と呼ばれる) を割り当て、ファイル内の最後のログ ブロックの内容を含むように初期化します。

ログ シーケンス番号 (LSN) はログ レコードを識別することを思い出してください。メソッド append は、変数 latestLSN を使用して、1 から始まる LSN を順番に割り当てます。ログ マネージャーは、次に使用可能な LSN と、ディスクに書き込まれた最新のログ レコードの LSN を追跡します。メソッド flush は、最新の LSN を指定された LSN と比較します。指定された LSN の方が小さい場合、目的のログ レコードは既にディスクに書き込まれている必要があります。それ以外の場合は、ログ ページがディスクに書き込まれ、最新の LSN が最新のものになります。

append メソッドは、ログ レコードのサイズを計算して、現在のページに収まるかどうかを判断します。収まらない場合は、現在のページをディスクに書き込み、appendNewBlock を呼び出してページをクリアし、空になったページをログ ファイルに追加します。この戦略は、図 4.3 のアルゴリズムとは少し異なります。つまり、ログ マネージャは、完全なページを追加してファイルを拡張するのではなく、空のページを追加してログ ファイルを拡張します。この戦略は、flush がブロックがすでにディスク上にあると想定できるため、実装が簡単です。

```
pパブリッククラス LogTest { プライ
  ベート静的 LogMgr lm;

   public static void main(String[] args) { SimpleDB db = new SimpleDB("logtest", 400,
   8); lm = db.logMgr(); createRecords(1, 35); printLogRecords("ログ ファイルには現
   在、次のレコードがあります:"); createRecords(36, 70); lm.flush(65); printLogRec
   ords("ログ ファイルには現在、次のレコードがあります:"); }private static void p
   rintLogRecords(String msg) { System.out.println(msg); Iterator<byte[]> iter = lm.itera
   tor(); while (iter.hasNext()) { byte[] rec = iter.next(); Page p = new Page(rec); String s
   = p.getString(0); int npos = Page.maxLength(s.length()); int val = p.getInt(npos); Syst
   em.out.println("[" + s + ", " + val + "]"); }System.out.println(); }private static void cre
   ateRecords(int start, int end) { System.out.print("レコードを作成しています: "); for
   (int i=start; i<=end; i++) { byte[] rec = createLogRecord("record"+i, i+100); int lsn =
   lm.append(rec); System.out.print(lsn + " "); }System.out.println(); }private static byte[
   ] createLogRecord(String s, int n) { int npos = Page.maxLength(s.length()); byte[] b =
   new byte[npos + Integer.BYTES]; Page p = new Page(b); p.setString(0, s); p.setInt(np
   os, n); return b; }

}
```

図4.5 ログマネージャのテスト

プblic クラス LogMgr { private FileMgr fm; private String logfile; private Page logpage; private BlockId currentblk; private int latestLSN = 0; private int lastSavedLSN = 0; public LogMgr(FileMgr fm, String logfile) { this.fm = fm; this.logfile = logfile; byte[] b = new byte[fm.blockSize()]; logpage = new Page(b); int logsize = fm.length(logfile); if (logsize == 0) currentblk = appendNewBlock(); else { currentblk = new BlockId(logfile, logsize-1); fm.read(currentblk, logpage); } }public void flush(int lsn) { if (lsn >= lastSavedLSN) flush(); }public Iterator<byte[]> iterator() { flush(); return new LogIterator(fm, currentblk); }public synchronized int append(byte[] logrec) { int boundary = logpage.getInt(0); int recsize = logrec.length; int bytesneeded = recsize + Integer.BYTES; if (boundary - bytesneeded < Integer.BYTES) { // 収まらないので、flush(); // 次のブロックに移動します。 currentblk = appendNewBlock(); boundary = logpage.getInt(0); }int recpos = boundary - bytesneeded; logpage.setBytes(recpos, logrec); logpage.setInt(0, recpos); // 新しい境界 latestLSN += 1; latestLSN を返します。}private BlockId appendNewBlock() { BlockId blk = fm.append(logfile); logpage.setInt(0, fm.blockSize()); fm.write(blk, logpage); blk を返します。 }private void flush() { fm.write(currentblk, logpage); lastSavedLSN = latestLSN; }

}

図4.6 SimpleDBクラスLogMgrのコード

append は、ログ レコードをページの右から左に配置することに注意してください。変数 boundary には、最後に追加されたレコードのオフセットが含まれます。この戦略により、ログ イテレータは左から右に読み取ることで、レコードを逆順に読み取ることができます。境界値はページの最初の 4 バイトに書き込まれるため、イテレータはレコードの開始位置を認識できます。

イテレータ メソッドはログをフラッシュし (ログ全体がディスク上にあることを確認するため)、LogIterator オブジェクトを返します。LogIterator クラスはイテレータを実装するパッケージ プライベート クラスです。そのコードは図 4.7 に示されています。LogIterator オブジェクトは、ログ ブロックの内容を保持するためのページを割り当てます。コンストラクタは、ログの最後のブロックの最初のレコード (最後のログ レコードが書き込まれた場所) にイテレータを配置します。メソッド next は、ページ内の次のレコードに移動します。レコードがなくなると、前のブロックを読み取ります。

```
クラス LogIterator は Iterator を実装します<byte[]> { private FileMgr fm; private Blo
ckId blk; private Page p; private int currentpos; private int boundary; public LogIterator(
FileMgr fm, BlockId blk) { this.fm = fm; this.blk = blk; byte[] b = new byte[fm.blockSiz
e()]; p = new Page(b); moveToBlock(blk); }public boolean hasNext() { return currentpos
<fm.blockSize() || blk.number()>0; }public byte[] next() { if (currentpos == fm.blockS
ize()) { blk = new BlockId(blk.fileName(), blk.number()-1); moveToBlock(blk); }byte[]
rec = p.getBytes(currentpos); currentpos += Integer.BYTES + rec.length; return rec; }pri
vate void moveToBlock(BlockId blk) { fm.read(blk, p); boundary = p.getInt(0); currentp
os = boundary; } }
```

図4.7 SimpleDBクラスLogIteratorのコード

ページに挿入し、最初のレコードを返します。ページにこれ以上レコードがなく、前のブロックもない場合は、hasNext メソッドは false を返します。

4.4 ユーザーデータの管理

ログ レコードは、限定された、よく理解された方法で使用されます。そのため、ログ マネージャーはメモリの使用を微調整できます。特に、1 つの専用ページでジョブを最適に実行できます。同様に、各 LogIterator オブジェクトに必要なのは 1 ページだけです。

一方、JDBC アプリケーションは、まったく予測できない方法でデータにアクセスします。アプリケーションが次にどのブロックを要求するか、また以前のブロックに再度アクセスするかどうかを知る方法はありません。また、アプリケーションがブロックの処理を完全に終了した後でも、別のアプリケーションが近い将来に同じブロックにアクセスするかどうかはわかりません。このセクションでは、このような状況でデータベース エンジンがメモリを効率的に管理する方法について説明します。

4.4.1 バッファマネージャ

バッファ マネージャは、ユーザー データを保持するページを管理するデータベース エンジンのコンポーネントです。バッファ マネージャは、バッファ プールと呼ばれる固定されたページ セットを割り当てます。この章の冒頭で述べたように、バッファ プールはコンピューターの物理メモリに収まる必要があり、これらのページはオペレーティング システムが保持する I/O バッファから取得する必要があります。

ブロックにアクセスするために、クライアントは図 4.8 に示すプロトコルに従ってバッファ マネージャと対話します。

あるページが現在何らかのクライアントによって固定されている場合、そのページは固定されているとみなされます。そうでない場合、そのページは固定解除されています。バッファ マネージャは、ページが固定されている限り、そのページをクライアントが利用できるようにしておく義務があります。逆に、ページの固定が解除されると、バッファ マネージャはそれを別のブロックに割り当てることができます。

クライアントがバッファ マネージャにページをブロックに固定するように要求すると、バッファ マネージャは次の 4 つの可能性のいずれかに遭遇します。

- ブロックの内容はバッファ内のどこかのページにあり、
  - ページはピン留めされています。
  - ページのピン留めは解除されています。

1. クライアントは、バッファ マネージャに、バッファ プールからそのブロックにページを *pin* するように要求します。2. クライアントは、必要なだけページの内容にアクセスします。3. クライアントは、ページの処理を完了すると、バッファ マネージャにそのページを *unpin* するように指示します。

図4.8 ディスクブロックにアクセスするためのプロトコル

- ブロックの内容は現在どのバッファにも存在せず、

  – バッファー プール内に固定されていないページが少なくとも 1 つ存在します。 – バッファー プール内のすべてのページが固定されています。

最初のケースは、1 つ以上のクライアントが現在ブロックの内容にアクセスしているときに発生します。ページは複数のクライアントによって固定される可能性があるため、バッファ マネージャは単にページに別のピンを追加し、ページをクライアントに返します。ページを固定している各クライアントは、その値を同時に読み取り、変更できます。バッファ マネージャは、発生する可能性のある潜在的な競合については考慮しません。その責任は、第 5 章の同時実行マネージャにあります。

2 番目のケースは、バッファを使用していたクライアントがバッファの使用を終了したが、バッファがまだ再割り当てされていない場合に発生します。ブロックの内容はまだバッファ ページ内にあるため、バッファ マネージャはページを固定してクライアントに返すだけでページを再利用できます。

3 番目のケースでは、バッファ マネージャがブロックをディスクからバッファ ページに読み込む必要があります。これにはいくつかの手順があります。バッファ マネージャは、最初に、再利用する固定されていないページを選択する必要があります (固定されたページはクライアントによってまだ使用されているため)。次に、選択されたページが変更されている場合、バッファ マネージャはページの内容をディスクに書き込む必要があります。このアクションは、ページのフラッシュと呼ばれます。最後に、要求されたブロックを選択されたページに読み込み、ページを固定できます。

4 番目のケースは、第 14 章のクエリ処理アルゴリズムのように、バッファが頻繁に使用される場合に発生します。この場合、バッファ マネージャはクライアントの要求を満たすことができません。最適な解決策は、バッファ マネージャが、固定されていないバッファ ページが使用可能になるまでクライアントを待機リストに配置することです。

4.4.2 バッファ

バッファ プール内の各ページには、固定されているかどうか、固定されている場合はどのブロックに割り当てられているかなどのステータス情報が関連付けられています。バッファは、この情報を含むオブジェクトです。バッファ プール内のすべてのページには、バッファが関連付けられています。各バッファは、そのページの変更を監視し、変更されたページをディスクに書き込む役割を担っています。ログと同様に、バッファはページの書き込みを遅らせることができれば、ディスク アクセスを減らすことができます。たとえば、ページが複数回変更された場合、すべての変更が行われた後で、そのページを 1 回書き込む方が効率的です。適切な戦略は、ページの固定が解除されるまで、バッファによるページのディスクへの書き込みを延期することです。

実際には、バッファはそれよりも長く待機できます。変更されたページの固定が解除されたが、ディスクに書き込まれていないとします。ページが同じブロックに再び固定されると (上記の 2 番目のケースのように)、クライアントは変更された内容をそのまま見ることができます。これは、ページがディスクに書き込まれてから読み戻された場合と同じ効果がありますが、ディスク アクセスはありません。ある意味では、バッファのページはディスク ブロックのメモリ内バージョンとして機能します。ブロックを使用するクライアントは、バッファ ページに誘導されるだけで、ディスクアクセスを一切行わずに、そのページを読み取ったり変更したりできます。

実際、バッファが変更したページをディスクに書き込む必要がある理由は2つしかありません。バッファが固定されているためページが置き換えられるか、

別のブロック（上記の 3 番目のケースのように）に書き込むか、リカバリ マネージャーがシステム クラッシュの可能性を防ぐためにその内容をディスクに書き込む必要があります（第 5 章で説明します）。

4.4.3 バッファ置換戦略

バッファ プール内のページは、割り当てられていない状態で始まります。ピン要求が到着すると、バッファ マネージャは、要求されたブロックを未割り当てページに割り当てて、バッファ プールを準備します。すべてのページが割り当てられると、バッファ マネージャはページの置き換えを開始します。バッファ マネージャは、バッファ プール内のピンされていないページを置き換え用に選択できます。

バッファ マネージャがページを置き換える必要があり、すべてのバッファ ページが固定されている場合、要求元のクライアントは待機する必要があります。したがって、各クライアントは「善良な市民」として、不要になったらすぐにページの固定を解除する責任があります。

複数のバッファ ページが固定解除されている場合、バッファ マネージャはどのページを置き換えるかを決定する必要があります。この選択は、データベース システムの効率に劇的な影響を与える可能性があります。たとえば、次にアクセスされるページを置き換えるのは最悪の選択です。バッファ マネージャはすぐに別のページを置き換える必要があるためです。最も長い時間使用されないページを常に置き換えるのが最善の選択であることがわかります。

バッファ マネージャは次にどのページがアクセスされるかを予測できないため、推測せざるを得ません。ここで、バッファ マネージャは、仮想メモリ内のページをスワップするときのオペレーティング システムとほぼ同じ状況にあります。ただし、大きな違いが 1 つあります。オペレーティング システムとは異なり、バッファ マネージャは、ページが現在使用されているかどうかを把握しています。これは、使用中のページがまさに固定されているページであるためです。固定されたページを置き換えることができないという負担は、ありがたいことです。クライアントは、責任を持ってページを固定することにより、バッファ マネージャが本当に間違った推測を行わないようにします。バッファ置き換え戦略では、現在不要なページから選択するだけでよく、これはそれほど重要ではありません。

固定されていないページのセットが与えられた場合、バッファ マネージャは、それらのページのうち、最も長い間必要とされないページを決定する必要があります。たとえば、データベースには通常、データベースの存続期間中ずっと使用されるページ (第 7 章のカタログ ファイルなど) がいくつかあります。バッファ マネージャは、そのようなページを置き換えないようにする必要があります。なぜなら、それらのページはすぐに再び固定されることがほぼ確実だからです。最善の推測を行う置き換え戦略はいくつかあります。このセクションでは、そのうちの 4 つ、Naïve、FIFO、LRU、および Clock について説明します。

図 4.9 は、それらの置換アルゴリズムの動作を比較できる例を示しています。パート (a) は、ファイルの 5 つのブロックを固定および固定解除する一連の操作を示し、パート (b) は、バッファー プールに 4 つのバッファーが含まれていると仮定して、バッファー プールの結果の状態を示しています。唯一のページ置換は、5 番目のブロック (つまり、ブロック 50) が固定されたときに発生しました。ただし、その時点で固定解除されたバッファーは 1 つだけであったため、バッファー マネージャーには選択の余地がありませんでした。つまり、ページ置換戦略に関係なく、バッファー プールは図 4.9b のようになります。

ピン(10); ピン(20); ピン(30); ピン(40); ピン解除(20); ピン(50); ピン解除(40); ピン解除(10); ピン解除(30); ピン解除(50);

(a)

Buffer:	0	1	2	3
block#	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

(b)

図4.9 4つのバッファプールに対するピン/アンピン操作の影響。(a) 10回のピン/アンピン操作のシーケンス、(b) バッファプールの結果の状態

図 4.9b の各バッファには、ブロック番号、バッファに読み込まれた時刻、および固定解除された時刻の 3 つの情報が保持されています。図の時刻は、図 4.9a の操作の位置に対応しています。

図 4.9b のバッファはすべてピン留めされていません。ここで、バッファ マネージャがさらに 2 つのピン留め要求を受信したとします。

ピン(60); ピン(70);

バッファ マネージャは 2 つのバッファを置き換える必要があります。すべてのバッファが使用可能ですが、どれを選択すればよいでしょうか。次の置き換えアルゴリズムはそれぞれ異なる答えを出します。

ナイーブ戦略
最も単純な置換戦略は、バッファ プールを順番に走査し、最初に見つかった固定されていないバッファを置換することです。図 4.9 の例では、ブロック 60 はバッファ 0 に割り当てられ、ブロック 70 はバッファ 1 に割り当てられます。

この戦略は実装が簡単ですが、他に推奨できる点はあまりありません。たとえば、図 4.9 のバッファーをもう一度考えてみましょう。クライアントが次のようにブロック 60 と 70 を繰り返し固定したり固定解除したりするとします。

ピン(60); ピン解除(60); ピン(70); ピン解除(70); ピン(60); ピン解除(60);...

単純な置換戦略では、両方のブロックにバッファ 0 が使用されるため、ブロックが固定されるたびにディスクから読み取る必要があります。問題は、バッファ プールが均等に使用されないことです。置換戦略でブロック 60 と 70 に 2 つの異なるバッファを選択した場合、ブロックはそれぞれ 1 回ずつディスクから読み取られることになります。これは、効率の大幅な向上です。

FIFO戦略
ナイーブな戦略では、利便性のみに基づいてバッファを選択します。FIFO 戦略は、最も最近置換されていないバッファ、つまりバッファプールに最も長く留まっているページを選択することで、よりインテリジェントにしようとします。この戦略は通常、ナイーブな戦略よりもうまく機能します。古いページは削除される可能性が低いためです。

最も古いページは、最近フェッチされたページよりも多く必要とされます。図 4.9 では、最も古いページは「読み込み時間」の値が最小のページです。したがって、ブロック 60 はバッファ 0 に割り当てられ、ブロック 70 はバッファ 2 に割り当てられます。

FIFO は合理的な戦略ですが、常に正しい選択になるとは限りません。たとえば、データベースには、第 7 章のカタログ ページなど、頻繁に使用されるページが多数あります。これらのページはほぼすべてのクライアントによって使用されるため、可能な限り置き換えない方が合理的です。ただし、これらのページは最終的にプール内で最も古いページになり、FIFO 戦略では置き換え対象として選択されます。

FIFO 置換戦略は 2 つの方法で実装できます。1 つは、図 4.9b に示すように、各バッファにそのページが最後に置換された時刻を保持させることです。置換アルゴリズムはバッファ プールをスキャンし、置換時刻が最も早い固定されていないページを選択します。2 つ目のより効率的な方法は、バッファ マネージャが置換時刻順に並べられたバッファへのポインタのリストを保持することです。置換アルゴリズムはリストを検索し、最初に見つかった固定されていないページを置換し、そのページへのポインタをリストの末尾に移動します。

LRU戦略

FIFO 戦略では、ページがバッファ プールに追加された時点に基づいて置換を決定します。同様の戦略として、ページが最後にアクセスされた時点に基づいて決定を下す方法があります。その理由は、最近使用されていないページは、近い将来も使用されないからです。この戦略は LRU (Least Recently Used) と呼ばれます。図 4.9 の例では、「time unpinned」の値はバッファが最後に使用された時点に対応します。したがって、ブロック 60 はバッファ 3 に割り当てられ、ブロック 70 はバッファ 0 に割り当てられます。

LRU 戦略は、効果的な汎用戦略になる傾向があり、よく使用されるページの置き換えを回避します。FIFO の両方の実装オプションを LRU に適応させることができます。必要な変更は、バッファー マネージャーが、ページが置き換えられるたびではなく、ページが固定解除されるたびにタイムスタンプを更新する (最初のオプションの場合) か、リストを更新する (2 番目のオプションの場合) ことだけです。

クロック戦略

この戦略は、上記の戦略の興味深い組み合わせであり、実装が簡単でわかりやすいものです。単純な戦略と同様に、クロック置換アルゴリズムはバッファー プールをスキャンし、最初に見つかった固定されていないページを選択します。違いは、アルゴリズムが常に前回の置換後のページからスキャンを開始することです。バッファー プールが円を形成すると想像すると、置換アルゴリズムはアナログ時計の針のようにプールをスキャンし、ページが置換されたときに停止し、別の置換が必要なときに開始します。

図 4.9b の例ではクロックの位置は示されていません。ただし、最後に置換されたのはバッファ 1 であり、クロックはその直後に配置されていることを意味します。したがって、ブロック 60 はバッファ 2 に割り当てられ、ブロック 70 はバッファ 3 に割り当てられます。

クロック戦略は、バッファを可能な限り均等に使用しようとします。ページが固定されている場合、クロック戦略はそれをスキップし、プール内の他のすべてのバッファを調べるまで、再度検討しません。この機能により、この戦略はLRU風になります。ページが頻繁に使用される場合、そのページが再び使用される可能性が高くなるという考えです。

交換の順番が来たときにピン留めされます。そうであれば、スキップされて「別のチャンス」が与えられます。

4.5 SimpleDB バッファマネージャ

このセクションでは、SimpleDB データベース システムのバッファー マネージャーについて説明します。セクション 4.5.1 では、バッファー マネージャーの API について説明し、その使用例を示します。セクション 4.5.2 では、これらのクラスを Java で実装する方法を示します。

4.5.1 バッファマネージャのAPI

SimpleDB バッファ マネージャは、パッケージ simpledb.buffer によって実装されます。このパッケージは、BufferMgr と Buffer の 2 つのクラスを公開します。それらの API は、図 4.10 に示されています。

各データベース システムには、システムの起動時に作成される BufferMgr オブジェクトが 1 つあります。そのコンストラクタには、バッファ プールのサイズと、ファイル マネージャおよびログ マネージャへの参照という 3 つの引数があります。

BufferMgr オブジェクトには、ページを固定および固定解除するメソッドがあります。メソッド pin は、指定されたブロックを含むページに固定された Buffer オブジェクトを返し、unpin メソッドはページの固定を解除します。available メソッドは、固定解除されたバッファ ページの数を返します。また、メソッド flushAll は、指定されたトランザクションによって変更されたすべてのページがディスクに書き込まれたことを確認します。

Buffer オブジェクトが与えられると、クライアントはそのコンテンツ メソッドを呼び出して、関連付けられたページを取得できます。クライアントがページを変更する場合、適切なログ レコードを生成し、バッファの setModified メソッドを呼び出すこともクライアントの責任となります。

BufferMgr

```
public BufferMgr(FileMgr fm, LogMgr lm, int numbuffs); public Buffer pin(Bl
ockId blk); public void unpin(Buffer buff); public int available(); public void flu
shAll(int txnum);
```

Buffer

```
public Buffer(FileMgr fm, LogMgr lm); public Page contents(); public
BlockId block(); public boolean isPinned(); public void setModified(int
txnum, int lsn); public int modificationTx();
```

図4.10 SimpleDBバッファマネージャのAPI

このメソッドには、変更トランザクションを識別する整数と、生成されたログ レコードの LSN という 2 つの引数があります。

図 4.11 のコードは Buffer クラスをテストします。最初に実行したときには「新しい値は 1 です」と出力され、その後の実行ごとに出力された値が増分されます。コードの動作は次のとおりです。3 つのバッファを持つ SimpleDB オブジェクトを作成します。ページをブロック 1 に固定し、オフセット 80 の整数を増分し、ページが変更されたことを示すために setModified を呼び出します。setModified への引数は、生成されたログ ファイルのトランザクション番号と LSN である必要があります。これら 2 つの値の詳細については第 5 章で説明します。それまでは、指定された引数は適切なプレースホルダです。

バッファ マネージャは、実際のディスク アクセスをクライアントから隠します。クライアントは、自分に代わってディスク アクセスが何回発生し、いつ発生したかを正確に把握していません。ディスク読み取りは、pin の呼び出し中のみ、特に指定されたブロックが現在バッファ内に存在しない場合に発生します。ディスク書き込みは、pin または flushAll の呼び出し中にのみ発生します。pin の呼び出しにより、置き換えられたページが変更されている場合はディスク書き込みが発生し、flushAll の呼び出しにより、指定されたトランザクションによって変更されたページごとにディスク書き込みが発生します。

例えば、図 4.11 のコードにはブロック 1 に対する 2 つの変更が含まれています。これらの変更はどちらもディスクに明示的に書き込まれません。コードを実行すると、

```
public class BufferTest {
   public static void main(String[] args) {
      SimpleDB db = new SimpleDB("buffertest", 400, 3);
      BufferMgr bm = db.bufferMgr();

      Buffer buff1 = bm.pin(new BlockId("testfile", 1));
      Page p = buff1.contents();
      int n = p.getInt(80);
      p.setInt(80, n+1);         // This modification will
      buff1.setModified(1, 0);   // get written to disk.
      System.out.println("The new value is " + (n+1));
      bm.unpin(buff1);
      // One of these pins will flush buff1 to disk:
      Buffer buff2 = bm.pin(new BlockId("testfile", 2));
      Buffer buff3 = bm.pin(new BlockId("testfile", 3));
      Buffer buff4 = bm.pin(new BlockId("testfile", 4));

      bm.unpin(buff2);
      buff2 = bm.pin(new BlockId("testfile", 1));
      Page p2 = buff2.contents();
      p2.setInt(80, 9999);       // This modification
      buff2.setModified(1, 0);   // won't get written to disk.
      bm.unpin(buff2);
   }
}
```

図4.11 Bufferクラスのテスト

最初の変更はディスクに書き込まれますが、2 番目の変更は書き込まれません。最初の変更について考えてみましょう。バッファー プールにはバッファーが 3 つしかないため、バッファー マネージャーはブロック 2、3、4 のページを固定するために、ブロック 1 のページを置き換える (つまりディスクに書き込む) 必要があります。一方、2 番目の変更後にはブロック 1 のページを置き換える必要がないため、ページはディスクに書き込まれず、その変更は失われます。失われた変更の問題については、第 5 章で説明します。

データベース エンジンに多数のクライアントがあり、そのすべてが大量のバッファーを使用しているとします。すべてのバッファー ページが固定される可能性があります。この場合、バッファー マネージャーは固定要求をすぐに満たすことができません。代わりに、クライアントを待機リストに入れます。バッファーが使用可能になると、バッファー マネージャーはクライアントを待機リストから外し、固定要求を完了できるようにします。つまり、クライアントはバッファーの競合に気付かず、エンジンの速度が低下したように見えることだけを感知します。

バッファ競合が深刻な問題を引き起こす可能性がある状況が 1 つあります。クライアント A と B がそれぞれ 2 つのバッファを必要としているが、使用できるバッファは 2 つしかないというシナリオを考えてみましょう。クライアント A が最初のバッファをピン留めするとします。これで 2 番目のバッファをめぐる競争が発生します。クライアント A がクライアント B より先に 2 番目のバッファを取得した場合、B は待機リストに追加されます。クライアント A は最終的にバッファの処理を終了してピン留めを解除し、その時点でクライアント B はバッファをピン留めできます。これは良いシナリオです。今度は、クライアント B がクライアント A より先に 2 番目のバッファを取得した場合を考えてみましょう。その場合、A と B の両方が待機リストに載ります。システム内のクライアントがこれら 2 つだけであれば、バッファのピン留めの解除イベントを待つ実際のデッドロックが永久に待機リストに載ったままになりますが、正確は悪い可能性はほとんどありません。A と B はデッドロック状態にあるため、その可能性に対処できるように準備しておく必要があります。SimpleDB が採用している解決策は、クライアントがバッファを待機している時間を追跡することです。待機時間が長すぎる場合 (たとえば、10 秒)、バッファ マネージャはクライアントがデッドロック状態にあると想定し、BufferAbortException タイプの例外をスローします。クライアントは、通常、トランザクションをロールバックし、場合によっては再起動することで、例外を処理する必要があります。

図 4.12 のコードは、バッファ マネージャをテストします。ここでも、3 つのバッファのみを持つ SimpleDB オブジェクトを作成し、バッファ マネージャを呼び出して、ファイル「testfile」のブロック 0、1、2 にページを固定します。次に、ブロック 1 の固定を解除し、ブロック 2 を再度固定し、ブロック 1 を再度固定します。これらの 3 つのアクションでは、ディスクの読み取りは発生せず、使用可能なバッファは残りません。ブロック 3 を固定しようとすると、スレッドは待機リストに入ります。ただし、スレッドはすでにすべてのバッファを保持しているため、固定が解除されることはなく、バッファ マネージャは 10 秒待機した後に例外をスローします。プログラムは例外をキャッチして続行します。ブロック 2 の固定を解除します。バッファが使用可能になったため、ブロック 3 の固定は成功します。

パブリック クラス BufferMgrTest { public static void main(String[] args) throws Exception { Simp leDB db = new SimpleDB("buffermgrtest", 400, 3); BufferMgr bm = db.bufferMgr(); Buffer[] buff = new Buffer[6]; buff[0] = bm.pin(new BlockId("testfile", 0)); buff[1] = bm.pin(new BlockId("testfile", 1)); buff[2] = bm.pin(new BlockId("testfile", 2)); bm.unpin(buff[1]); buff[1] = null; buff[3] = bm.pin(new BlockId("testfile", 0)); buff[4] = bm.pin(new BlockId("testfile", 1)); System.out.println("使用可能なバッファー: " + bm.available()); try {System.out.println("ブロック3をピン留めしようとしています..."); buff[5] = bm.pin(new BlockId("testfile", 3)); }catch(BufferAbortException e) { Syste m.out.println("例外: 使用可能なバッファーがありません\n"); }bm.unpin(buff[2]); buff[2] = null; buff[5] = bm.pin(new BlockId("testfile", 3)); // これで動作するようになりました System.out.print ln("最終バッファー割り当て:"); for (int i=0; i<buff.length; i++) { バッファー b = buff[i]; if (b != null) System.out.println("buff["+i+"] はブロック " + b.block()" に固定されています); } } }

図4.12 バッファマネージャのテスト

4.5.2 バッファマネージャの実装

図 4.13 には、Buffer クラスのコードが含まれています。Buffer オブジェクトは、そのページに関する 4 種類の情報を追跡します。

- ページに割り当てられたブロックへの参照。ブロックが割り当てられていない場合、値は null になります。
- ページがピン留めされた回数。ピン留めの回数はピン留めするたびに増加し、ピン留めを解除するたびに減少します。
- ページが変更されたかどうかを示す整数。値が !1 の場合、ページは変更されていないことを示します。それ以外の場合、この整数は変更を行ったトランザクションを識別します。
- ログ情報。ページが変更された場合、バッファには最新のログレコードのLSNが保持されます。LSNの値は負になることはありません。クライアントが

```
プblic クラス Buffer { private FileMgr fm; private LogMgr lm; privat
   e Page contents; private BlockId blk = null; private int pins = 0; priv
   ate int txnum = -1; private int lsn = -1; public Buffer(FileMgr fm, L
   ogMgr lm) { this.fm = fm; this.lm = lm; contents = new Page(fm.bl
   ockSize()); }public Page contents() { return contents; }public Block
   Id block() { return blk; }public void setModified(int txnum, int lsn)
   { this.txnum = txnum; if (lsn>=0) this.lsn = lsn; }public boolean isP
   inned() { return pins > 0; }public int modificationTx() { return txnu
   m; }void assignmentToBlock(BlockId b) { flush(); blk = b; fm.read(
   blk, contents); pins = 0; }void flush() { if (txnum >= 0) { lm.flush(ls
   n); fm.write(blk, contents); txnum = -1; } }void pin() { pins++; }voi
   d unpin() { pins--; }
```

}

図4.13 SimpleDBクラスBufferのコード

負の LSN で setModified された場合、その更新に対してログ レコードが生成されなかったことを示します。

メソッド flush は、バッファの割り当てられたディスク ブロックがページと同じ値を持つようにします。ページが変更されていない場合、メソッドは何もする必要はありません。ページが変更されている場合、メソッドは最初に LogMgr.flush を呼び出して、対応するログ レコードがディスク上にあることを確認し、次にページをディスクに書き込みます。

メソッド assignToBlockは、バッファをディスク ブロックに関連付けます。バッファは最初にフラッシュされ、前のブロックへの変更が保持されます。次に、バッファは指定されたブロックに関連付けられ、その内容をディスクから読み取ります。

BufferMgr のコードは図 4.14 に示されています。メソッド pin は、指定されたブロックにバッファを割り当てます。これは、プライベート メソッド tryToPin を呼び出すことによって行われます。このメソッドは 2 つの部分から構成されます。最初の部分である findExistingBuffer は、指定されたブロックにすでに割り当てられているバッファを見つけようとします。見つかった場合は、そのバッファが返されます。それ以外の場合は、アルゴリズムの 2 番目の部分である chooseUnpinnedBuffer が、単純な置換を使用して固定されていないバッファを選択します。選択されたバッファの assignmentToBlock メソッドが呼び出され、既存のページをディスクに書き込み (必要な場合)、新しいページをディスクから読み取ります。固定されていないバッファが見つからない場合は、メソッドは null を返します。tryToPin がnull を返す場合、pin メソッドは Java メソッド wait を呼び出します。Java では、すべてのオブジェクトに待機リストがあります。オブジェクトの wait メソッドは、呼び出しスレッドの実行を中断し、そのスレッドをそのリストに配置します。図 4.14 では、スレッドは次の 2 つの条件のいずれかが発生するまでそのリストに残ります。

- 別のスレッドが、notifyAll を呼び出します (これは、unpin の呼び出しから発生します)。
- MAX_TIME ミリ秒が経過しました。これは、スレッドの待機時間が長すぎることを意味します。

待機中のスレッドが再開すると、ループを継続し、(待機していた他のすべてのスレッドとともに) バッファを取得しようとします。スレッドは、バッファを取得するか、時間制限を超えるまで、待機リストに戻され続けます。

unpin メソッドは、指定されたバッファのピンを解除し、そのバッファがまだピン留めされているかどうかを確認します。ピン留めされていない場合は、notifyAll が呼び出され、待機リストからすべてのクライアント スレッドが削除されます。これらのスレッドはバッファをめぐって争い、最初にスケジュールされたスレッドが勝ちます。他のスレッドの 1 つがスケジュールされたときに、すべてのバッファがまだ割り当てられていることが判明する場合があります。その場合、そのスレッドは待機リストに戻されます。

4.6 章の要約

- データベース エンジンは、ディスク アクセスを最小限に抑えるよう努める必要があります。そのため、ディスク ブロックを保持するために使用するメモリ内ページを慎重に管理します。これらのページを管理するデータベース コンポーネントは、ログ マネージャーとバッファー マネージャーです。

```
プblic class BufferMgr { private Buffer[] bufferpool; private int numAvailable; private sta
  tic final long MAX_TIME = 10000; // 10 秒 public BufferMgr(FileMgr fm, LogMgr l
  m, int numbuffs) { bufferpool = new Buffer[numbuffs]; numAvailable = numbuffs; for
  (int i=0; i<numbuffs; i++) bufferpool[i] = new Buffer(fm, lm); }public synchronized in
  t available() { return numAvailable; }public synchronized void flushAll(int txnum) { fo
  r (Buffer buff : bufferpool) if (buff.modifyingTx() == txnum) buff.flush(); }public sync
  hronized void unpin(Buffer buff) { buff.unpin(); if (!buff.isPinned()) { numAvailable
  ++; notificationAll(); } }public synchronized Buffer pin(BlockId blk) { try {long timest
  amp = System.currentTimeMillis(); Buffer buff = tryToPin(blk); while (buff == null &
  & !waitingTooLong(timestamp)) { wait(MAX_TIME); buff = tryToPin(blk); }if (buff
  == null) throw new BufferAbortException(); return buff; }catch(InterruptedException e
  ) { throw new BufferAbortException(); } }
```

図4.14 SimpleDBクラスBufferMgrのコード

```
private boolean waitingTooLong(long starttime) { return System.currentTimeMillis() - st
arttime > MAX_TIME; }private Buffer tryToPin(BlockId blk) { Buffer buff = findExisti
ngBuffer(blk); if (buff == null) { buff = chooseUnpinnedBuffer(); if (buff == null) retur
n null; buff.assignToBlock(blk); }if (!buff.isPinned()) numAvailable--; buff.pin(); return
buff; }private Buffer findExistingBuffer(BlockId blk) { for (Buffer buff : bufferpool) { B
lockId b = buff.block(); if (b != null && b.equals(blk)) return buff; }return null; }private
Buffer chooseUnpinnedBuffer() { for (Buffer buff : bufferpool) if (!buff.isPinned()) retur
n buff; return null; }
```

}

図4.14（続き）

- ログ マネージャは、ログ ファイルにログ レコードを保存する役割を担います。ログ レコードは常にログ ファイルに追加され、変更されることがないため、ログ マネージャは非常に効率的です。ログ マネージャは 1 ページを割り当てるだけでよく、そのページをディスクに書き込む回数をできるだけ少なくするシンプルなアルゴリズムを備えています。
- バッファ マネージャは、バッファ プールと呼ばれる複数のページを割り当てて、ユーザー データを処理します。バッファ マネージャは、クライアントの要求に応じて、バッファ ページをディスク ブロックに固定したり、固定を解除したりします。クライアントは、バッファ ページが固定された後にそのページにアクセスし、完了したらバッファの固定を解除します。
- 変更されたバッファは、ページが置き換えられるときと、リカバリ マネージャがそれをディスク上に置くことを必要とするときの 2 つの状況でディスクに書き込まれます。

- クライアントがページをブロックに固定するように要求すると、バッファ マネージャは適切なバッファを選択します。そのブロックのページがすでにバッファ内にある場合は、そのバッファが使用されます。そうでない場合は、バッファ マネージャは既存のバッファの内容を置き換えます。
- どのバッファを置き換えるかを決定するアルゴリズムは、バッファ置き換え戦略と呼ばれます。興味深い置き換え戦略は次の 4 つです。

  – Naïve: 最初に見つかった固定されていないバッファを選択します。 – FIFO: 内容が最も最近置き換えられた固定されていないバッファを選択します。 – LRU: 内容が最も最近固定解除された固定されていないバッファを選択します。 – Clock: 最後に置き換えられたバッファからバッファを順番にスキャンし、最初に見つかった固定されていないバッファを選択します。

4.7 推奨される読み物

Effelsberg ら (1984) の論文には、この章の多くのアイデアを拡張した、バッファ管理に関するわかりやすく包括的な説明が含まれています。Gray と Reuter (1993) の第 13 章には、バッファ管理に関する詳細な説明が含まれており、典型的なバッファ マネージャの C ベースの実装を例に挙げて説明されています。

Oracle のデフォルトのバッファ置換戦略は LRU です。ただし、大きなテーブルをスキャンするときには FIFO 置換を使用します。その理由は、テーブル スキャンでは通常、固定解除されたブロックは必要ないため、LRU では間違ったブロックを保存してしまうからです。詳細については、Ashdown ら (2019) の第 14 章を参照してください。

バッファ マネージャ自体をよりインテリジェントにする方法を研究した研究者は数名います。基本的な考え方は、バッファ マネージャが各トランザクションのピン要求を追跡できるようにすることです。パターンが検出されると (たとえば、トランザクションがファイルの同じ N ブロックを繰り返し読み取る)、ピンされていなくても、それらのページを置き換えないようにします。Ng ら (1991) の記事では、この考え方についてさらに詳しく説明し、シミュレーション結果をいくつか提供しています。

Ashdown, L., et al. (2019). Oracle データベースの概念。ドキュメント E96138-01、Oracle Corporation。https://docs.oracle.com/en/database/oracle/ oracle-database/19/cncpt/database-concepts.pdf から入手可能。 Effelsberg, W., & Haerder, T. (1984). データベース バッファ管理の原則。ACM Transactions on Database Systems、9(4)、560 – 595。 Gray, J., & Reuter, A. (1993). トランザクション処理: 概念と手法。Morgan Kaufman。 Ng, R., Faloutsos, C., & Sellis, T. (1991). 限界利益に基づく柔軟なバッファ割り当て。ACM SIGMOD カンファレンスの議事録、pp. 387 – 396。

4.8 演習

概念演習

4.1. LogMgr.iterator のコードは flush を呼び出します。この呼び出しは必要ですか? 説明してください。

4.2. BufferMgr.pin メソッドが同期される理由を説明してください。同期されなかった場合、どのような問題が発生する可能性がありますか?

4.3. 同じブロックに複数のバッファを割り当てることはできますか? 説明してください。

4.4. この章のバッファ置換戦略では、使用可能なバッファを探すときに、変更されたページと変更されていないページを区別しません。改善策としては、バッファ マネージャが可能な限り常に変更されていないページを置換するようにすることが考えられます。

> (a) この提案によってバッファ マネージャによるディスク アクセスの回数が減る理由を 1 つ挙げてください。 (b) この提案によってバッファ マネージャによるディスク アクセスの回数が増加する理由を 1 つ挙げてください。 (c) この戦略は価値があると思いますか。説明してください。

4.5. バッファ置換戦略として他に考えられるのは、最近最も変更されていない戦略です。バッファ マネージャは、LSN が最も低い変更されたバッファを選択します。このような戦略がなぜ価値があるのか説明してください。 4.6. バッファ ページがディスクに書き込まれずに数回変更されたとします。バッファは最新の変更の LSN のみを保存し、ページが最終的にフラッシュされるときにこの LSN のみをログ マネージャに送信します。バッファが他の LSN をログ マネージャに送信する必要がない理由を説明してください。 4.7. 図 4.9a の pin/unpin シナリオの例と、追加の操作 pin(60); pin(70) を検討してください。テキストに示されている 4 つの置換戦略のそれぞれについて、バッファ プールに 5 つのバッファが含まれていると仮定して、バッファの状態を描画してください。 4.8.図 4.9b のバッファ状態から始めて、次のシナリオを示してください。(a) FIFO 戦略では、ディスク アクセスが最も少なくて済みます。(b) LRU 戦略では、ディスク アクセスが最も少なくて済みます。(c) クロック戦略では、ディスク アクセスが最も少なくて済みます。4.9. 2 つの異なるクライアントがそれぞれ同じブロックを固定したいが、使用できるバッファがないため待機リストに載せられているとします。SimpleDB クラス BufferMgr の実装について考えます。1 つのバッファが使用可能になると、両方のクライアントがそれを使用できるようになることを示します。

4.10. 「仮想はそれ自体の報酬である」という格言を考えてみましょう。この語呂合わせの巧妙さについてコメントし、バッファー マネージャーへの適用性について議論します。

プログラミング演習

4.11. SimpleDB ログ マネージャーは独自のページを割り当て、それを明示的にディスクに書き込みます。別の設計オプションとしては、最後のログ ブロックにバッファーを固定し、バッファー マネージャーにディスク アクセスを処理させるというものがあります。

(a) このオプションの設計を作成します。対処する必要がある問題は何ですか? これは良いアイデアですか? (b) SimpleDB を変更して設計を実装します。

4.12. 各 LogIterator オブジェクトは、アクセスするログ ブロックを保持するためのページを割り当てます。

(a) ページの代わりにバッファを使用する方がはるかに効率的である理由を説明してください。 (b) ページの代わりにバッファを使用するようにコードを変更します。 バッファの固定を解除するにはどうすればよいですか?

4.13. この演習では、JDBC プログラムがバッファ プール内のすべてのバッファを悪意を持って固定できるかどうかを調べます。

(a) SimpleDB バッファー プールのすべてのバッファーを固定する JDBC プログラムを作成します。すべてのバッファーが固定されるとどうなりますか? (b) Derby データベース システムは、SimpleDB とは異なる方法でバッファーを管理します。JDBC クライアントがバッファーを要求すると、Derby はバッファーを固定し、バッファーのコピーをクライアントに送信して、バッファーの固定を解除します。コードが他の Derby クライアントに対して悪意のあるものではない理由を説明してください。 (c) Derby は、常にエンジンからクライアントにページをコピーすることで SimpleDB の問題を回避します。このアプローチの結果を説明してください。SimpleDB のアプローチよりもこのアプローチを好みますか? (d) 不正なクライアントがすべてのバッファーを独占しないようにする別の方法は、各トランザクションがバッファー プールの特定の割合 (たとえば 10%) を超えて固定できないようにすることです。SimpleDB バッファー マネージャーにこの変更を実装してテストしてください。

4.14. この章で説明した他の各置換戦略を実装するために、BufferMgr クラスを変更します。

4.15. 演習 4.4 では、変更されたページよりも変更されていないページを選択するページ置換戦略が提案されています。この置換戦略を実装してください。 4.16. 演習 4.5 では、LSN が最も低い変更されたページを選択するページ置換戦略が提案されています。この戦略を実装してください。

4.17. SimpleDB バッファ マネージャは、バッファを検索するときにバッファ プールを順番にトラバースします。プールに何千ものバッファがある場合、この検索には時間がかかります。コードを変更して、検索時間を改善するデータ構造 (専用リストやハッシュ テーブルなど) を追加します。

4.18. 演習 3.15 では、ディスク使用量に関する統計情報を維持するコードを記述するように求められました。このコードを拡張して、バッファ使用量に関する情報も提供するようにして下さい。

第5章
トランザクション管理

バッファ マネージャを使用すると、複数のクライアントが同じバッファに同時にアクセスして、その値を任意に読み書きできます。その結果、混乱が生じる可能性があります。クライアントがページを参照するたびに、ページの値が異なり (一貫性がなくなることもあります)、クライアントがデータベースの正確な情報を取得できなくなります。または、2 つのクライアントが互いの値を無意識に上書きし、データベースが破損する可能性もあります。したがって、データベース エンジンには、同時実行マネージャと回復マネージャがあり、その役割は、順序を維持し、データベースの整合性を確保することです。各クライアント プログラムは、一連のトランザクションとして記述されます。同時実行マネージャは、これらのトランザクションの実行を規制して、一貫した動作を実現します。回復マネージャは、ログのレコードを読み書きして、コミットされていないトランザクションによって行われた変更を必要に応じて元に戻すことができます。この章では、これらのマネージャの機能と、それらを実装するために使用される手法について説明します。

5.1 取引

次のフィールドを持つ 2 つのテーブルがある航空会社の予約データベースを考えてみましょう。

SEATS(フライトID、利用可能数、価格) CUST
(顧客ID、残高支払期限)

図 5.1 には、指定されたフライトの指定された顧客のチケットを購入するための JDBC コードが含まれています。このコードにはバグはありませんが、複数のクライアントによって同時に使用されている場合やサーバーがクラッシュした場合には、さまざまな問題が発生する可能性があります。次の 3 つのシナリオは、これらの問題を示しています。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_5

```
public void reserveSeat(Connection conn, int custId,
                        int flightId) throws SQLException {
   Statement stmt = conn.createStatement();
   String s;

   // step 1: Get availability and price
   s  = "select NumAvailable, Price from SEATS " +
        "where FlightId = " + flightId;
   ResultSet rs = stmt.executeQuery(s);
   if (!rs.next()) {
      System.out.println("Flight doesn't exist");
      return;
   }
   int numAvailable = rs.getInt("NumAvailable");
   int price = rs.getInt("Price");
   rs.close();

   if (numAvailable == 0) {
      System.out.println("Flight is full");
      return;
   }

   // step 2: Update availability
   int newNumAvailable = numAvailable - 1;
   s  = "update SEATS set NumAvailable = " + newNumAvailable +
        " where FlightId = " + flightId;
   stmt.executeUpdate(s);

   // step 3: Get and update customer balance
   s = "select BalanceDue from CUST where CustID = " + custId;
   rs = stmt.executeQuery(s);
   int newBalance = rs.getInt("BalanceDue") + price;
   rs.close();

   s  =  "update CUST set BalanceDue = " + newBalance +
         " where CustId = " + custId;
   stmt.executeUpdate(s);
}
```

図5.1 フライトの座席を予約するためのJDBCコード

最初のシナリオでは、クライアント A と B の両方が、次の一連のアクションで JDBC コードを同時に実行するとします。

- クライアント A はステップ 1 をすべて実行し、その後中断されます。
- クライアント B は完了まで実行します。
- クライアントAが実行を完了します。

この場合、両方のスレッドは numAvailable に同じ値を使用します。その結果、2 つの座席が販売されますが、利用可能な座席数は 1 回だけ減少します。

2 番目のシナリオでは、クライアントがコードを実行しているときに、ステップ 2 の実行直後にサーバーがクラッシュしたとします。この場合、座席は予約されますが、顧客に料金は請求されません。

3 番目のシナリオでは、クライアントがコードを完了するまで実行したが、バッファ マネージャが変更されたページをすぐにディスクに書き込まなかったとします。サーバーがクラッシュした場合 (おそらく数日後)、どのページ (ある場合) が最終的にディスクに書き込まれたかを知る方法はありません。最初の更新が書き込まれ、2 番目の更新が書き込まれなかった場合、顧客には無料チケットが提供されます。2 番目の更新が書き込まれ、最初の更新が書き込まれなかった場合、顧客には存在しないチケットの料金が請求されます。どちらのページも書き込まれなかった場合、やり取り全体が失われます。

これらのシナリオは、クライアント プログラムが無差別に実行できる場合に、データが失われたり破損したりする可能性があることを示しています。データベース エンジンは、クライアント プログラムをトランザクションで構成するように強制することでこの問題を解決します。トランザクションとは、単一の操作として動作する一連の操作です。「単一の操作として」の意味は、原子性、一貫性、独立性、および永続性という、いわゆる ACID プロパティによって特徴付けられます。

- アトミック性とは、トランザクションが「すべてかゼロか」であることを意味します。つまり、すべての操作が成功するか (トランザクションがコミットされる)、すべてが失敗するか (トランザクションがロールバックされる) のいずれかです。
- 一貫性とは、すべてのトランザクションがデータベースを一貫した状態で残すことを意味します。これは、各トランザクションが他のトランザクションとは独立して実行できる完全な作業単位であることを意味します。
- 分離とは、トランザクションが、エンジンを使用している唯一のスレッドであるかのように動作することを意味します。複数のトランザクションが同時に実行されている場合、その結果は、すべてが何らかの順序でシリアルに実行された場合と同じになります。
- 耐久性とは、コミットされたトランザクションによって行われた変更が永続的であることが保証されることを意味します。

上記の各シナリオは、ACID プロパティの違反によって発生します。最初のシナリオは、両方のクライアントが numAvailable に同じ値を読み取ったため、分離プロパティに違反しています。一方、シリアル実行では、2 番目のクライアントは最初のクライアントによって書き込まれた値を読み取ることになります。2 番目のシナリオは原子性に違反しており、3 番目のシナリオは永続性に違反しています。

アトミック性と耐久性のプロパティは、コミットおよびロールバック操作の適切な動作を記述します。コミットされたトランザクションは耐久性がある必要があり、コミットされていないトランザクション (明示的なロールバックまたはシステム クラッシュによる) は変更が完全に元に戻されている必要があります。これらの機能はリカバリ マネージャーの責任であり、セクション 5.3 で取り上げます。

一貫性と独立性の特性は、同時実行クライアントの適切な動作を記述します。データベース エンジンは、クライアントが互いに競合しないようにする必要があります。一般的な戦略は、競合が発生しそうなときにそれを検出し、競合がなくなるまでクライアントの 1 つを待機させることです。これらの機能は同時実行マネージャの責任であり、セクション 5.4 で取り上げます。

Transaction

```
public Transaction(FileMgr fm, LogMgr lm, BufferMgr bm);
public void commit();
public void rollback();
public void recover();

public void pin(BlockId blk);
public void unpin(BlockId blk);
public int  getInt(BlockId blk, int offset);
public String getString(BlockId blk, int offset);
public void setInt(BlockId blk, int offset, int val,
                   boolean okToLog);
public void setString(BlockId blk, int offset, String val,
                      boolean okToLog);
public int  availableBuffs();

public int   size(String filename);
public Block append(String filename);
public int   blockSize();
```

図5.2 SimpleDBトランザクションのAPI

5.2 SimpleDB でのトランザクションの使用

リカバリ マネージャと同時実行マネージャがどのように機能するかについて詳しく説明する前に、クライアントがトランザクションをどのように使用するかを把握しておくと役立ちます。SimpleDB では、すべての JDBC トランザクションに独自の Transaction オブジェクトがあります。その API は図 5.2 に示されています。

Transaction のメソッドは 3 つのカテゴリに分類されます。最初のカテゴリは、トランザクションの存続期間に関連するメソッドで構成されます。コンストラクターは新しいトランザクションを開始し、commit メソッドと rollback メソッドはそれを終了し、recover メソッドはコミットされていないすべてのトランザクションをロールバックします。commit メソッドと rollback メソッドは、トランザクションの固定されたバッファーページを自動的に固定解除します。

2 番目のカテゴリは、バッファにアクセスするためのメソッドで構成されます。トランザクションは、バッファの存在をクライアントから隠します。クライアントがブロックの pin を呼び出すと、トランザクションはバッファを内部的に保存し、クライアントに返しません。クライアントが getInt などのメソッドを呼び出すと、BlockId 参照が渡されます。トランザクションは対応するバッファを見つけ、バッファのページで getInt メソッドを呼び出し、結果をクライアントに返します。

トランザクションは、クライアントからバッファを非表示にし、同時実行マネージャと回復マネージャへの必要な呼び出しを行えるようにします。たとえば、setInt のコードは、適切なロックを取得し (同時実行制御用)、バッファを変更する前に、現在バッファ内にある値をログに書き込みます (回復用)。setInt と setString の 4 番目の引数は、更新をログに記録するかどうかを示すブール値です。この値は通常 true ですが、ログに記録することが適切ではない特定のケース (新しいブロックのフォーマットやトランザクションの取り消しなど) があり、その場合は値を false にする必要があります。

```
public class TxTest { public static void main(String[] args) throws Exception { SimpleDB db = ne
w SimpleDB("txtest", 400, 8); FileMgr fm = db.fileMgr(); LogMgr lm = db.logMgr(); BufferMgr b
m = db.bufferMgr(); Transaction tx1 = new Transaction(fm, lm, bm); BlockId blk = new BlockId("
testfile", 1); tx1.pin(blk); // 初期ブロック値をログに記録しません。 tx1.setInt(blk, 80, 1, false)
; tx1.setString(blk, 40, "one", false); tx1.commit(); Transaction tx2 = new Transaction(fm, lm, bm);
tx2.pin(blk); int ival = tx2.getInt(blk, 80); String sval = tx2.getString(blk, 40); System.out.println("
位置80の初期値 = " + ival); System.out.println("位置40の初期値 = " + sval); int newival = ival
+ 1; int newsval = sval + "!"; tx2.setInt(blk, 80, newival, true); tx2.setString(blk, 40, newsval, true);
tx2.commit(); トランザクション tx3 = new Transaction(fm, lm, bm); tx3.pin(blk); System.out.pri
ntln("場所 80 の新しい値 = " + tx3.getInt(blk, 80)); System.out.println("場所 40 の新しい値 = "
+ tx3.getString(blk, 40)); tx3.setInt(blk, 80, 9999, true); System.out.println("場所 80 のロールバ
ック前の値 = " + tx3.getInt(blk, 80)); tx3.rollback(); トランザクション tx4 = new Transaction(f
m, lm, bm); tx4.pin(blk); System.out.println("場所 80 のロールバック後 = " + tx4.getInt(blk, 80)
); tx4.commit(); } }
```

図5.3 SimpleDBトランザクションクラスのテスト

3 番目のカテゴリは、ファイル マネージャーに関連する 3 つのメソッドで構成されます。size メソッドはファイル マーカーの末尾を読み取り、append メソッドはそれを変更します。これらのメソッドは、潜在的な競合を回避するために並行性マネージャーを呼び出す必要があります。blockSize メソッドは、それを必要とする可能性のあるクライアントの利便性のために存在します。

図 5.5 は、Transaction メソッドの簡単な使用方法を示しています。このコードは 4 つのトランザクションで構成されており、図 4.11 の BufferTest クラスと同様のタスクを実行します。4 つのトランザクションはすべて、ファイル「testfile」のブロック 1 にアクセスします。トランザクション tx1 は、オフセット 80 と 40 の値を初期化します。これらの更新はログに記録されません。トランザクション tx2 は、それらの値を読み取って出力し、増分します。トランザクション tx3 は、増分された値を読み取って出力します。次に、整数を 9999 に設定してロールバックします。トランザクション tx4 は、ロールバックが実際に発生したことを確認するために整数を読み取ります。

このコードを第 4 章のコードと比較して、Transaction クラスが何を行うかを確認します。Transaction クラスは、バッファーを管理し、更新ごとにログ レコードを生成してログ ファイルに書き込み、必要に応じてトランザクションをロールバックできます。ただし、コードが ACID プロパティを満たすようにするために、このクラスがバックグラウンドでどのように動作するかも同様に重要です。たとえば、プログラムの実行中にランダムにプログラムを中止したとします。その後、データベース エンジンを再起動すると、コミットされたすべてのトランザクションの変更がディスク上に保存され (永続性)、実行中だったトランザクションの変更がロールバックされます (アトミック性)。

さらに、Transaction クラスは、このプログラムが ACID 分離プロパティを満たすことも保証します。トランザクション tx2 のコードを検討してください。変数 newival と newsval (太字のコードを参照) は次のように初期化されます。

```
int newival = ival + 1;文字列 newsv
al = sval + "!";
```

このコードでは、ブロックの位置 80 と 40 の値が変更されていないと想定しています。ただし、同時実行制御がなければ、この想定は正しくない可能性があります。問題は、セクション 2.2.3 の「非反復読み取り」シナリオです。tx2 が ival と sval を初期化した直後に割り込まれ、別のプログラムがオフセット 80 と 40 の値を変更したとします。すると、ival と sval の値は古くなり、tx2 は正しい値を取得するために getInt と getString を再度呼び出す必要があります。Transaction クラスは、このような可能性が発生しないようにする責任があり、このコードが正しいことが保証されます。

5.3 リカバリ管理

リカバリ マネージャは、ログを読み取って処理するデータベース エンジンの一部です。リカバリ マネージャには、ログ レコードの書き込み、トランザクションのロールバック、システム クラッシュ後のデータベースの回復という 3 つの機能があります。このセクションでは、これらの機能について詳しく説明します。

<START、1>COMMIT、1>START、2>SETINT、2、テストファイル、1、80、1、2> <SETSTRING、2、テストファイル、1、40、1、1!> <COMMIT、2>START、3>SETINT、3、テストファイル、1、80、2、9999> <ROLLBACK、3>START、4>COMMIT、4>

図5.4 図5.3から生成されたログレコード

5.3.1 ログレコード

トランザクションをロールバックできるようにするために、リカバリ マネージャーはトランザクションのアクティビティに関する情報をログに記録します。特に、ログ可能なアクティビティが発生するたびに、ログにログ レコードを書き込みます。ログ レコードには、開始レコード、コミット レコード、ロールバック レコード、更新レコードの 4 つの基本的な種類があります。ここでは、SimpleDB に従って、整数の更新用と文字列の更新用の 2 種類の更新レコードを想定します。

ログ レコードは、次のログ可能なアクティビティによって生成されます。

- トランザクションが開始されると、開始レコードが書き込まれます。
- トランザクションが完了すると、コミットまたはロールバック レコードが書き込まれます
- トランザクションが値を変更すると、更新レコードが書き込まれます。

ログに記録できる可能性のある別のアクティビティは、ファイルの末尾にブロックを追加することです。その後、トランザクションがロールバックすると、追加によって割り当てられた新しいブロックをファイルから割り当て解除できます。簡単にするために、この可能性は無視します。演習 5.48 でこの問題を取り上げます。

例として、図 5.3 のコードを考えてみましょう。tx1 の id は 1 などとします。図 5.4 は、このコードによって生成されたログ レコードを示しています。各ログ レコードには、レコードの種類 (START、SETINT、SETSTRING、COMMIT、または ROLLBACK) の説明とトランザクションの ID が含まれます。更新レコードには、変更されたファイルの名前とブロック番号、変更が発生したオフセット、そのオフセットの古い値、およびそのオフセットの新しい値という 5 つの追加値が含まれます。

一般に、複数のトランザクションが同時にログに書き込むため、特定のトランザクションのログ レコードはログ全体に散在することになります。

5.3.2 ロールバック

ログの用途の1つは、リカバリマネージャが特定のトランザクションTをロールバックできるようにすることです。リカバリマネージャは、変更を元に戻すことでトランザクションをロールバックします。

1. 現在のレコードを最新のログ レコードに設定します。2. 現在のレコードが T の開始レコードになるまで、次の操作を実行します。a) 現在のレコードが T の更新レコードである場合は、保存されている古い値を指定された場所に書き込みます。b) ログ内の前のレコードに移動します。3. ロールバック レコードをログに追加します。

図5.5 トランザクションTをロールバックするアルゴリズム

これらの変更は更新ログレコードにリストされているので、ログをスキャンして各更新レコードを見つけ、変更された各値の元の内容を復元するのは比較的簡単です。図 5.5 にアルゴリズムを示します。

このアルゴリズムがログを先頭から順方向ではなく、末尾から逆方向に読み取る理由は 2 つあります。1 つ目の理由は、ログ ファイルの先頭には、かなり前に完了したトランザクションのレコードが含まれているためです。探しているレコードはログの末尾にある可能性が高いため、末尾から読み取る方が効率的です。2 つ目の、より重要な理由は、正確性を確保するためです。ある場所の値が複数回変更されたとします。その場合、その場所には複数のログ レコードが存在し、それぞれが異なる値を持ちます。復元する値は、これらのレコードのうち最も古いものから取得する必要があります。ログ レコードが逆順に処理されると、実際にこのようになります。

5.3.3 回復

ログのもう 1 つの用途は、データベースを回復することです。回復は、データベース エンジンが起動するたびに実行されます。その目的は、データベースを適切な状態に復元することです。「適切な状態」という用語は、次の 2 つのことを意味します。

- 未完了のトランザクションはすべてロールバックする必要があります。
- コミットされたすべてのトランザクションの変更内容がディスクに書き込まれる必要があ

通常のシャットダウン後にデータベース エンジンが起動すると、既存のトランザクションが完了するまで待機し、すべてのバッファーをフラッシュするという通常のシャットダウン手順のため、エンジンはすでに適切な状態になっているはずです。ただし、クラッシュによってエンジンが予期せずダウンした場合は、実行が失われた未完了のトランザクションが存在する可能性があります。エンジンがそれらを完了する方法がないため、それらの変更を元に戻す必要があります。また、変更がまだディスクにフラッシュされていないコミット済みのトランザクションが存在する可能性もあります。これらの変更はやり直す必要があります。

リカバリ マネージャは、ログ ファイルにトランザクションのコミットまたはロールバック レコードが含まれている場合、トランザクションが完了したものと想定します。したがって、システム クラッシュ前にトランザクションがコミットされていても、そのコミット レコードがログ ファイルに記録されていない場合、リカバリ マネージャはトランザクションが完了しなかったものとして扱います。この状況は公平ではないように思われるかもしれませんが、リカバリ マネージャが実行できる操作は他にありません。リカバリ マネージャが認識できるのは、ログ ファイルに含まれている内容だけです。トランザクションに関するその他の情報はすべてシステム クラッシュで消去されているためです。

// 元に戻す段階
1各ログレコードについて（最後から逆方向に読み
ます）：a) 現在のレコードがコミットレコードの
場合：
そのトランザクションをコミットされたトランザクションのリストに追加します。
b) 現在のレコードがロールバックレコードの場合:
そのトランザクションをロールバックされたトランザクションのリストに追加します。
c) 現在のレコードがコミットまたはロールバック リストにないトランザクションの更新レコードである場合は、指定された場所の古い値を復元します。
// やり直し段階
2. 各ログ レコードについて (先頭から順方向に読み取り): 現在のレコードが更新レコードであり、そのトランザクションがコミット済みリストにある場合は、指定された場所に新しい値を復元します。

図5.6 データベースを回復するための元に戻す・やり直しアルゴリズム

実際、コミットされたトランザクションをロールバックすることは不公平なだけでなく、耐久性の ACID プロパティに違反します。したがって、リカバリ マネージャーは、このようなシナリオが発生しないようにする必要があります。これは、コミット操作を完了する前にコミット ログ レコードをディスクにフラッシュすることによって行われます。ログ レコードをフラッシュすると、以前のログ レコードもすべてフラッシュされることを思い出してください。したがって、リカバリ マネージャーがログでコミット レコードを見つけると、そのトランザクションのすべての更新レコードもログにあることがわかります。

各更新ログ レコードには、変更の古い値と新しい値の両方が含まれます。変更を元に戻す場合は古い値を使用し、変更をやり直す場合は新しい値を使用します。図 5.6 に回復アルゴリズムを示します。

ステージ 1 では、未完了のトランザクションを元に戻します。ロールバック アルゴリズムと同様に、正確性を保証するために、ログを最後から逆方向に読み取る必要があります。ログを逆方向に読み取るということは、コミット レコードが常に更新レコードの前に見つかるということでもあります。そのため、アルゴリズムは更新レコードに遭遇すると、そのレコードを元に戻す必要があるかどうかを認識します。

ステージ 1 では、ログ全体を読み取ることが重要です。たとえば、最初のトランザクションでは、無限ループに入る前にデータベースに変更が加えられている可能性があります。ログの先頭まで読み取らない限り、その更新レコードは見つかりません。

ステージ 2 では、コミットされたトランザクションをやり直します。リカバリ マネージャーは、どのバッファーがフラッシュされ、どのバッファーがフラッシュされなかったかを判断できないため、コミットされたすべてのトランザクションによって行われたすべての変更をやり直します。

リカバリ マネージャーは、ログを先頭から読み込んでステージ 2 を実行します。リカバリ マネージャーは、ステージ 1 でコミットされたトランザクションのリストを計算しているため、どの更新レコードを再実行する必要があるかを認識しています。再実行ステージでは、ログを前向きに読み取る必要があることに注意してください。複数のコミットされたトランザクションが同じ値を変更した場合、最終的に回復される値は最新の変更によるものになります。

リカバリ アルゴリズムは、データベースの現在の状態を認識しません。ログによってデータベースの内容が正確に示されるため、その場所の現在の値を確認せずに、古い値または新しい値をデータベースに書き込みます。この機能には、次の 2 つの影響があります。

- 回復はべき等です。
- リカバリによって、必要以上にディスクへの書き込みが発生する可能性があります。

べき等性とは、リカバリ アルゴリズムを複数回実行しても、1 回実行した場合と同じ結果になることを意味します。特に、リカバリ アルゴリズムの一部だけを実行した直後に、リカバリ アルゴリズムを再実行しても、同じ結果が得られます。この特性は、アルゴリズムの正確性にとって不可欠です。たとえば、リカバリ アルゴリズムの途中でデータベース システムがクラッシュしたとします。データベース システムが再起動すると、リカバリ アルゴリズムが最初から再度実行されます。アルゴリズムがべき等性でない場合、再実行するとデータベースが破損します。

このアルゴリズムはデータベースの現在の内容を参照しないため、不必要な変更が行われる可能性があります。たとえば、コミットされたトランザクションによって行われた変更がディスクに書き込まれているとします。ステージ 2 でこれらの変更をやり直すと、変更された値が既存の内容に設定されます。アルゴリズムを修正して、これらの不必要なディスク書き込みを行わないようにすることができます。演習 5.44 を参照してください。

5.3.4 元に戻すだけのリカバリとやり直しだけのリカバリ

前のセクションの回復アルゴリズムは、元に戻す操作とやり直し操作の両方を実行します。データベース エンジンは、アルゴリズムを簡略化して元に戻す操作のみ、またはやり直し操作のみを実行するように選択できます。つまり、アルゴリズムのステージ 1 またはステージ 2 のいずれかを実行しますが、両方は実行しません。

5.3.4.1 元に戻すだけのリカバリ

リカバリ マネージャがコミットされた変更がすべてディスクに書き込まれたことを確信している場合は、ステージ 2 を省略できます。リカバリ マネージャは、コミット レコードをログに書き込む前に、バッファを強制的にディスクに書き込むことで、これを実行できます。図 5.7 は、このアプローチをアルゴリズムとして表現しています。リカバリ マネージャは、このアルゴリズムの手順を、指定された順序どおりに実行する必要があります。

元に戻すだけのリカバリと元に戻すやり直しのリカバリのどちらが優れているでしょうか。元に戻すだけのリカバリの方が、ログ ファイルのパスが 2 回ではなく 1 回だけなので高速です。更新レコードに新しい変更された値を含める必要がなくなるため、ログのサイズも少し小さくなります。一方、コミット操作は変更されたバッファをフラッシュする必要があるため、はるかに低速です。システム クラッシュが頻繁に発生しないと仮定すると、元に戻す/やり直しのリカバリの方が優れています。トランザクションのコミットが高速になるだけでなく、バッファ フラッシュが延期されるため、全体的なディスク書き込みも少なくなります。

1.2. コミット レコードをログに書き込みます。トランザクションの変更されたバッファをディスクにフラッシュします。3. コミット レコードを含むログ ページをフラッシュします

図5.7 元に戻すだけのリカバリを使用したトランザクションのコミットアルゴリズム

5.3.4.2 再実行のみのリカバリ

コミットされていないバッファがディスクに書き込まれない場合は、ステージ 1 を省略できます。リカバリ マネージャは、トランザクションが完了するまで各トランザクションのバッファを固定しておくことで、この特性を確保できます。固定されたバッファは置換対象として選択されないため、その内容はフラッシュされません。さらに、ロールバックされたトランザクションでは、変更されたバッファを「消去」する必要があります。図 5.8 は、ロールバック アルゴリズムに必要な変更を示しています。

再実行のみのリカバリは、コミットされていないトランザクションを無視できるため、元に戻す/再実行のリカバリよりも高速です。ただし、各トランザクションは変更するブロックごとにバッファを固定しておく必要があるため、システム内のバッファの競合が増加します。大規模なデータベースでは、この競合がすべてのトランザクションのパフォーマンスに重大な影響を与える可能性があるため、再実行のみのリカバリはリスクの高い選択となります。

元に戻すだけの手法とやり直しだけの手法を組み合わせて、ステージ 1 もステージ 2 も必要としない回復アルゴリズムを作成できるかどうかを考えるのは興味深いことです。演習 5.19 を参照してください。

5.3.5 先行書き込みログ

図 5.6 の回復アルゴリズムのステップ 1 は、さらに詳しく調べる必要があります。このステップでは、ログを反復処理し、未完了のトランザクションからの更新レコードごとに元に戻す操作を実行することを思い出してください。このステップの正しさを証明するために、私は次のような仮定を立てました。未完了のトランザクションのすべての更新には、ログ ファイル内に対応するログ レコードがあるはずです。そうでない場合、更新を元に戻す方法がないため、データベースが破損します。

システムはいつでもクラッシュする可能性があるため、この仮定を満たす唯一の方法は、ログ マネージャが各更新ログ レコードを書き込んだらすぐにディスクにフラッシュすることです。しかし、セクション 4.2 で示したように、この戦略は非常に非効率的です。もっと良い方法があるはずです。

問題が発生する可能性のある事柄の種類を分析してみましょう。未完了のトランザクションがページを変更し、対応する更新ログ レコードを作成したとします。サーバーがクラッシュした場合、次の 4 つの可能性があります。
(a) ページとログ レコードの両方がディスクに書き込まれました。 (b) ページのみがディスクに書き込まれました。 (c) ログ レコードのみがディスクに書き込まれました。 (d) どちらもディスクに書き込まれませんでした。

トランザクションによって変更された各バッファについて:

a) バッファを未割り当てとしてマークします。(SimpleDB では、ブロック番号を -1 に設定します) b) バッファを未変更としてマークします。 c) バッファの固定を解除します。

図5.8 再実行のみのリカバリを使用したトランザクションのロールバックアルゴリズム

それぞれの可能性を順に考えてみましょう。(a) の場合、リカバリ アルゴリズムはログ レコードを見つけ、ディスク上のデータ ブロックへの変更を元に戻します。問題ありません。(b) の場合、リカバリ アルゴリズムはログ レコードを見つけられないため、データ ブロックへの変更は元に戻されません。これは深刻な問題です。(c) の場合、リカバリ アルゴリズムはログ レコードを見つけ、ブロックへの存在しない変更を元に戻します。ブロックは実際には変更されていないため、これは時間の無駄ですが、間違いではありません。(d) の場合、リカバリ アルゴリズムはログ レコードを見つけられませんが、ブロックに変更がないため、元に戻すものは何もありません。問題ありません。

したがって、(b) が唯一の問題ケースです。データベース エンジンは、対応する変更されたバッファー ページをフラッシュする前に更新ログ レコードをディスクにフラッシュすることで、このケースを回避します。この戦略は、先行書き込みログの使用と呼ばれます。ログには、データベースへの変更が記述されることがあります (上記の可能性 (c) のように)。ただし、データベースが変更された場合、その変更のログ レコードは常にディスク上にあります。

先行書き込みログを実装する標準的な方法は、各バッファが最新の変更の LSN を追跡することです。バッファは変更されたページを置き換える前に、ログ マネージャにその LSN までログをフラッシュするように指示します。その結果、変更がディスクに保存される前に、変更に対応するログ レコードが常にディスク上に存在することになります。

5.3.6 静止チェックポイント

ログには、データベースに対するすべての変更の履歴が含まれます。時間が経つにつれて、ログ ファイルのサイズは非常に大きくなり、場合によってはデータ ファイルよりも大きくなります。リカバリ中にログ全体を読み取り、データベースに対するすべての変更を元に戻す/やり直す作業は、大変な作業になります。そのため、ログの一部のみを読み取るリカバリ戦略が考案されました。基本的な考え方は、リカバリ アルゴリズムが次の 2 つのことを認識するとすぐにログの検索を停止できるというものです。

- 以前のログ レコードはすべて、完了したトランザクションによって書き込まれました。
- これらのトランザクションのバッファはディスクにフラッシュされました。

最初の箇条書きは、リカバリ アルゴリズムの元に戻す段階に適用されます。これにより、ロールバックするコミットされていないトランザクションがなくなることが保証されます。2 番目の箇条書きはやり直し段階に適用され、以前にコミットされたすべてのトランザクションをやり直す必要がないことが保証されます。リカバリ マネージャーが元に戻すだけのリカバリを実装する場合、2 番目の箇条書きは常に当てはまることに注意してください。

リカバリ マネージャは、図 5.9 に示すように、いつでも静止チェックポイント操作を実行できます。このアルゴリズムのステップ 2 では、最初の箇条書きが満たされていることが保証され、ステップ 3 では、2 番目の箇条書きが満たされていることが保証されます。

静止チェックポイント レコードはログ マーカーとして機能します。回復アルゴリズムのステージ1がチェックポイントレコードに遭遇すると、

1. 新しいトランザクションの受け入れを停止します。 2. 既存のトランザクションが終了するまで待機します。 3. 変更されたすべてのバッファをフラッシュします。 4. 静止チェックポイントレコードをログに追加し、ディスクにフラッシュします。 5. 新しいトランザクションの受け入れを開始します。

図5.9 静止チェックポイントを実行するアルゴリズム

<START, 0>SETINT, 0, junk, 33, 8, 542, 543> <START, 1>START, 2>CO
<MMIT, 1> <SETSTRING, 2, junk, 44, 20, hello, ciao> //静止チェックポイント手順はここから始まります <SETSTRING, 0, junk, 33, 12, joe, joseph > <COMMIT, 0> //tx 3はここで開始したいのですが、待機する必要があります <SETINT, 2, junk, 66, 8, 0, 116> <COMMIT, 2> <CHECKPOINT> <START, 3> <SETINT、3、ジャンク、33、8、543、120>

図5.10 静止チェックポイントを使用したログ

ログでは、それ以前のログ レコードはすべて無視できることがわかっているため、ログのその時点からステージ 2 を開始して先に進むことができます。つまり、回復アルゴリズムは、静止チェックポイント レコードより前のログ レコードを調べる必要がありません。

静止チェックポイント レコードを書き込むのに適したタイミングは、システムの起動時、リカバリが完了した後、新しいトランザクションが開始される前です。リカバリ アルゴリズムはログの処理を完了したばかりなので、チェックポイント レコードによって、それらのログ レコードを再度調べる必要がなくなります。

例として、図 5.10 に示すログを考えてみましょう。このログの例は、次の 3 つのことを示しています。まず、チェックポイント プロセスが開始されると、新しいトランザクションを開始できません。2 番目に、最後のトランザクションが完了してバッファがフラッシュされるとすぐにチェックポイント レコードが書き込まれます。3 番目に、チェックポイント レコードが書き込まれるとすぐに他のトランザクションが開始される場合があります。

5.3.7 非静止チェックポイント

静止チェックポイントは実装が簡単で理解しやすい。しかし、リカバリマネージャが既存のトランザクションの完了を待つ間、データベースが利用できない状態になる必要がある。多くのデータベースアプリケーションでは、これは重大な欠点である。企業はデータベースが時々応答しなくなることを望んでいない。

1. T1 k を現在実行中のトランザクションとします。2. 新しいトランザクションの受け入れを停止します。3. 変更されたすべてのバッファをフラッシュします。4. レコード <NQCKPT T1 k> をログに書き込みます。5. 新しいトランザクションの受け入れを開始します。

図5.11 非静止チェックポイントレコードを追加するためのアルゴリズム

任意の期間にわたって。その結果、静止を必要としないチェックポイントアルゴリズムが開発されました。アルゴリズムは図 5.11 に示されています。 このアルゴリズムは、非静止チェックポイント レコードと呼ばれる異なる種類のチェックポイント レコードを使用します。非静止チェックポイント レコードには、現在実行中のトランザクションのリストが含まれます。 回復アルゴリズムは次のように修正されます。アルゴリズムのステージ 1 では、以前と同様にログを逆方向に読み取り、完了したトランザクションを追跡します。非静止チェックポイント レコード <NQCKPT T1,. . .,Tk> に遭遇すると、これらのトランザクションのうちどのトランザクションがまだ実行中であるかを判断します。その後、これらのトランザクションのうち最も古いものの開始レコードに遭遇するまで、ログを逆方向に読み取り続けます。この開始レコードより前のすべてのログ レコードは無視できます。

例として、図 5.10 のログをもう一度考えてみましょう。非静止チェックポイントでは、ログは図 5.12 のように表示されます。このログでは、図 5.10 でチェックポイント プロセスが開始された場所に <NQCKPT. . .> レコードが表示され、その時点でトランザクション 0 と 2 がまだ実行中であることがわかります。このログは、トランザクション 2 がコミットされないという点で、図 5.10 のログと異なります。

回復アルゴリズムがシステムの起動中にこのログを検出すると、ステージ 1 に入り、次のように進行します。

- <SETINT、3、...> ログ レコードに遭遇すると、トランザクション 3 がコミットされたトランザクションのリストに含まれているかどうかを確認します。そのリストは現在空なので、アルゴリズムは元に戻す操作を実行し、整数 543 をファイル「junk」のブロック 33 のオフセット 8 に書き込みます。

<START、0> <SETINT、0、ジャンク、33、8、542、543> <START、1> <START、2> <COMMIT、1> <SETSTRING、2、ジャンク、44、20、hello、ciao> <SETSTRING、0、ジャンク、33、12、ジョー、ジョセフ> <COMMIT、0> <START、3> <NQCKPT、0、2>SETINT、2、ジャンク、66、8、0、116> <SETINT、 3、ジャンク、33、8、543、120>

<

図5.12 非静止チェックポイントを使用したログ

- ログ レコード <SETINT、2、...> も同様に処理され、整数 0 が「junk」のブロック 66 のオフセット 8 に書き込まれます。
- <COMMIT、0> ログ レコードにより、コミットされたトランザクションのリストに 0 が追加されます。
- コミットされたトランザクション リストに 0 があるため、<SETSTRING、0、...> ログ レコードは無視されます。
- <NQCKPT 0,2> ログ レコードに遭遇すると、トランザクション 0 がコミットされたことがわかり、トランザクション 2 の開始レコードより前のすべてのログ レコードを無視できます。
- <START、2> ログ レコードに遭遇すると、ステージ 2 に入り、ログ内を前進し始めます。
- コミットされたトランザクション リストに 0 があるため、<SETSTRING、0、...> ログ レコードがやり直されます。値「joseph」は、「junk」のブロック 33 のオフセット 12 に書き込まれます。

5.3.8 データ項目の粒度

このセクションのリカバリ管理アルゴリズムでは、ログ記録の単位として値を使用します。つまり、変更された値ごとにログ レコードが作成され、ログ レコードには値の以前のバージョンと新しいバージョンが含まれます。このログ記録の単位は、リカバリ データ項目と呼ばれます。データ項目のサイズは、粒度と呼ばれます。

リカバリ マネージャーは、データ項目として値を使用する代わりに、ブロックまたはファイルの使用を選択できます。たとえば、データ項目としてブロックが選択されたとします。この場合、ブロックが変更されるたびに更新ログ レコードが作成され、ブロックの以前の値と新しい値がログ レコードに格納されます。

ブロックをログに記録する利点は、元に戻すだけのリカバリを使用する場合に必要なログ レコードが少なくなることです。トランザクションがブロックを固定し、いくつかの値を変更してから固定を解除するとします。変更された値ごとに 1 つのログ レコードを書き込む代わりに、ブロックの元の内容を 1 つのログ レコードに保存できます。もちろん、欠点は更新ログ レコードが非常に大きくなることです。つまり、実際に変更された値の数に関係なく、ブロックの内容全体が保存されます。したがって、ブロックをログに記録することは、トランザクションがブロックごとに多くの変更を行う傾向がある場合にのみ意味があります。

ここで、ファイルをデータ項目として使用するとどうなるか考えてみましょう。トランザクションは、変更したファイルごとに 1 つの更新ログ レコードを生成します。各ログ レコードには、そのファイルの元の内容全体が含まれます。トランザクションをロールバックするには、既存のファイルを元のバージョンに置き換えるだけです。この方法は、値またはブロックをデータ項目として使用する場合よりも実用的ではありません。なぜなら、変更された値がいくつあっても、各トランザクションでファイル全体のコピーを作成する必要があるためです。

ファイル粒度データ項目はデータベース システムには実用的ではありませんが、非データベース アプリケーションではよく使用されます。たとえば、コンピュータが

ファイルの編集中にクラッシュすることがあります。システムの再起動後、一部のワードプロセッサでは、ファイルの 2 つのバージョン (最後に保存したバージョンとクラッシュ時に存在していたバージョン) を表示できます。これは、これらのワードプロセッサが変更内容を元のファイルに直接書き込むのではなく、コピーに書き込むためです。保存すると、変更されたファイルが元のファイルにコピーされます。この戦略は、ファイルベースのログ記録の粗雑なバージョンです。

5.3.9 SimpleDBリカバリマネージャ

SimpleDB リカバリ マネージャーは、パッケージ simpledb.tx.recovery のクラス RecoveryMgr を介して実装されます。RecoveryMgr の API は図 5.13 に示されています。

各トランザクションには独自の RecoveryMgr オブジェクトがあり、そのメソッドはトランザクションの適切なログ レコードを書き込みます。たとえば、コンストラクタは開始ログ レコードをログに書き込みます。commit メソッドと rollback メソッドは対応するログ レコードを書き込みます。setInt メソッドと setString メソッドは指定されたバッファから古い値を抽出し、更新レコードをログに書き込みます。rollback メソッドと recovery メソッドはロールバック (またはリカバリ) アルゴリズムを実行します。

RecoveryMgr オブジェクトは、値の粒度データ項目を使用した元に戻すだけのリカバリを使用します。そのコードは、ログ レコードを実装するコードと、ロールバックおよびリカバリ アルゴリズムを実装するコードの 2 つの領域に分けられます。

5.3.9.1 ログレコード

セクション 4.2 で述べたように、ログ マネージャは各ログ レコードをバイト配列として認識します。ログ レコードの種類ごとに独自のクラスがあり、バイト配列に適切な値を埋め込む役割を担います。各配列の最初の値はレコードの演算子を示す整数です。演算子は定数 CHECKPOINT、START、COMMIT、ROLLBACK、SETINT、または SETSTRING のいずれかになります。残りの値は演算子によって異なります。静止チェックポイント レコードには他の値はありません。更新レコードには他の 5 つの値があり、その他のレコードには他の 1 つの値があります。

RecoveryMgr

public RecoveryMgr(Transaction tx、int txnum、LogMgr lm、BufferMgr bm); public void commit(); public void rollback(); public void recovery(); public int setInt(Buffer buff、int offset、int newval); public int setString(Buffer buff、int offset、String newval);

図5.13 SimpleDBリカバリマネージャのAPI

各ログ レコード クラスは、図 5.14 に示すような LogRecord インターフェイスを実装します。このインターフェイスは、ログ レコードのコンポーネントを抽出する 3 つのメソッドを定義します。メソッド op は、レコードの演算子を返します。メソッド txNumber は、ログ レコードを書き込んだトランザクションの ID を返します。このメソッドは、ダミーの ID 値を返すチェックポイント レコード以外のすべてのログ レコードに有効です。メソッド undo は、そのレコードに格納されている変更を復元します。setint および setstring ログ レコードにのみ、空でない undo メソッドがあります。これらのレコードのメソッドは、指定されたブロックにバッファーを固定し、指定されたオフセットに指定された値を書き込んで、バッファーの固定を解除します。

個々の紹介レコードのクラスはすべて同様のコードを持っています。図 5.15 に示すコードを持つクラスの 1 つ、たとえば SetStringRecord を調べるだけで十分です。

このクラスには、2 つの重要なメソッドがあります。静的メソッド writeToLog は、SETSTRING ログ レコードの 6 つの値をバイト配列にエンコードし、コンストラクタは、そのバイト配列から 6 つの値を取り出します。writeToLog の実装について考えてみましょう。まず、バイト配列のサイズと、各値の配列内のオフセットを計算します。次に、そのサイズのバイト配列を作成し、それを Page オブジェクトにラップして、ページの setInt メソッドと setString メソッドを使用します。

```
パブリック インターフェイス LogRecord { static final int CHECKPOINT = 0、START = 1、COMMIT = 2、ROLLBACK = 3、SETINT = 4、SETSTRING = 5; int op(); int txNumber(); void undo(int txnum); static LogRecord createLogRecord(byte[] bytes) { Page p = new Page(bytes); switch (p.getInt(0)) { case CHECKPOINT: return new CheckpointRecord(); case START: return new StartRecord(p); case COMMIT: return new CommitRecord(p); case ROLLBACK: return new RollbackRecord(p); case SETINT: return new SetIntRecord(p); case SETSTRING: return new SetStringRecord(p); default: return null; } } }
```

図5.14 SimpleDB LogRecordインターフェースのコード

プblic クラス SetStringRecord は LogRecord を実装します { private int txnum, offset; private S tring val; private BlockId blk; public SetStringRecord(Page p) { int tpos = Integer.BYTES; txnu m = p.getInt(tpos); int fpos = tpos + Integer.BYTES; String filename = p.getString(fpos); int bp os = fpos + Page.maxLength(filename.length()); int blknum = p.getInt(bpos); blk = new BlockId (filename, blknum); int opos = bpos + Integer.BYTES; offset = p.getInt(opos); int vpos = opos + Integer.BYTES; val = p.getString(vpos); }public int op() { return SETSTRING; }public int tx Number() { return txnum; }public String toString() { return "<SETSTRING " + txnum + " " + bl k + " " + offset + " " + val + ">"; }public void undo(Transaction tx) { tx.pin(blk); tx.setString(bl k, offset, val, false); // 元に戻す操作をログに記録しません。 tx.unpin(blk); }public static int writeToLog(LogMgr lm, int txnum, BlockId blk, int offset, String val) { int tpos = Integer.BYT ES; int fpos = tpos + Integer.BYTES; int bpos = fpos + Page.maxLength(blk.fileName().length( )); int opos = bpos + Integer.BYTES; int vpos = opos + Integer.BYTES; int reclen = vpos + Pag e.maxLength(val.length()); byte[] rec = new byte[reclen]; Page p = new Page(rec); p.setInt(0, SE TSTRING); p.setInt(tpos, txnum); p.setString(fpos, blk.fileName()); p.setInt(bpos, blk.number()); p.setInt(opos, offset); p.setString(vpos, val); return lm.append(rec); }

}

図5.15 SetStringRecordクラスのコード

適切な場所に値を書き込みます。コンストラクタも同様です。ページ内の各値のオフセットを決定し、それらを抽出します。

undo メソッドには、undo を実行するトランザクションである引数が 1 つあります。このメソッドでは、トランザクションはレコードで示されるブロックを固定し、保存された値を書き込んで、ブロックの固定を解除します。undo (ロールバックまたは回復) を呼び出すメソッドは、バッファの内容をディスクにフラッシュする役割を担います。

5.3.9.2 ロールバックとリカバリ

RecoveryMgr クラスは、元に戻すだけのリカバリ アルゴリズムを実装します。そのコードは図 5.16 に示されています。commit メソッドと rollback メソッドは、ログ レコードを書き込む前にトランザクションのバッファーをフラッシュし、doRollback メソッドと doRecover メソッドは、ログを 1 回逆方向にパスします。

doRollback メソッドはログ レコードを反復処理します。そのトランザクションのログ レコードが見つかるたびに、そのレコードの undo メソッドを呼び出します。そのトランザクションの開始レコードに遭遇すると停止します。

doRecover メソッドも同様に実装されています。ログを読み取り、静止チェックポイント レコードに達するかログの終わりに達し、コミットされたトランザクション番号のリストを保持します。ロールバックと同様に、コミットされていない更新レコードを元に戻しますが、特定のトランザクションだけでなく、すべてのコミットされていないトランザクションを処理する点が異なります。このメソッドは、すでにロールバックされたトランザクションを元に戻すため、図 5.6 の回復アルゴリズムとは少し異なります。この違いによってコードが間違っているわけではありませんが、効率は悪くなります。演習 5.50 では、コードの改善を求めています。

5.4 並行処理管理

同時実行マネージャは、同時トランザクションの正しい実行を担当するデータベース エンジンのコンポーネントです。このセクションでは、実行が「正しい」とはどういう意味かを調べ、正確性を保証するいくつかのアルゴリズムについて説明します。

5.4.1 シリアル化可能なスケジュール

トランザクションの履歴とは、データベースファイルにアクセスするメソッド、特にget/setメソッドの呼び出しのシーケンスです。[1] たとえば、図5.3の各トランザクションの履歴は、図5.17aに示すように、やや面倒ですが記述できます。トランザクションの履歴を表現する別の方法は、

[1]The `size` and `append` methods also access a database file but more subtly than do the get/set methods. Section 5.4.5 will consider the effect of `size` and `append`.

プblic クラス RecoveryMgr { private LogMgr lm; private BufferMgr bm; private Transactio
n tx; private int txnum; public RecoveryMgr(Transaction tx, int txnum, LogMgr lm, Buffe
rMgr bm) { this.tx = tx; this.txnum = txnum; this.lm = lm; this.bm = bm; StartRecord.writ
eToLog(lm, txnum); }public void commit() { bm.flushAll(txnum); int lsn = CommitRecor
d.writeToLog(lm, txnum); lm.flush(lsn); }public void rollback() { doRollback(); bm.flush
All(txnum); int lsn = RollbackRecord.writeToLog(lm, txnum); lm.flush(lsn); }public void
recovery() { doRecover(); bm.flushAll(txnum); int lsn = CheckpointRecord.writeToLog(l
m); lm.flush(lsn); }public int setInt(Buffer buff, int offset, int newval) { int oldval = buff.c
ontents().getInt(offset); BlockId blk = buff.block(); return SetIntRecord.writeToLog(lm, tx
num, blk, offset, oldval); }public int setString(Buffer buff, int offset, String newval) { Stri
ng oldval = buff.contents().getString(offset); BlockId blk = buff.block(); return SetStringR
ecord.writeToLog(lm, txnum, blk, offset, oldval); }private void doRollback() { イテレー
タ<byte[]> iter = lm.iterator(); while (iter.hasNext()) {

図5.16 RecoveryMgrクラスのコード

byte[] bytes = iter.next(); LogRecord rec = LogRecord.createLogRecord(bytes); if (rec.txNumber() == txnum) { if (rec.op() == START) return; rec.undo(tx); } } }private void doRecover() { Collection<Integer> finishedTxs = new ArrayList<Integer> (); Iterator<byte[]> iter = lm.iterator(); while (iter.hasNext()) { byte[] bytes = iter.next(); LogRecord rec = LogRecord.createLogRecord(bytes); if (rec.op() == CHECKPOINT) return; rec.op() == COMMIT || rec.op() == ROLLBACK) が終了している場合、finishedTxs.add(rec.txNumber()); それ以外の場合、finishedTxs.contains(rec.txNumber()) が終了している場合、rec.undo(tx); } }

}

図5.16（続き）

図 5.17b に示すように、影響を受けるブロックが表示されます。たとえば、tx2 の履歴には、ブロック blk から 2 回読み取り、次に blk に 2 回書き込みを行ったことが示されています。

正式には、トランザクションの履歴とは、そのトランザクションによって行われたデータベース アクションのシーケンスです。「データベース アクション」という用語は意図的に曖昧にされています。図 5.17 のパート (a) では、データベース アクションを値の変更として扱い、パート (b) では、データベース アクションをディスク ブロックの読み取り/書き込みとして扱いました。その他の粒度も可能であり、セクション 5.4.8 で説明します。それまでは、データベース アクションはディスク ブロックの読み取りまたは書き込みであると仮定します。

複数のトランザクションが同時に実行されている場合、データベース エンジンはスレッドの実行をインターリーブし、定期的に 1 つのスレッドを中断して別のスレッドを再開します (SimpleDB では、Java ランタイム環境がこれを自動的に行います)。したがって、同時実行マネージャによって実行される実際の操作のシーケンスは、トランザクションの履歴の予測できないインターリーブになります。このインターリーブはスケジュールと呼ばれます。

同時実行制御の目的は、正しいスケジュールだけが実行されるようにすることです。しかし、「正しい」とはどういう意味でしょうか。最も単純なスケジュール、つまりすべてのトランザクションがシリアルに実行されるスケジュールを考えてみましょう (図 5.17 など)。このスケジュール内の操作はインターリーブされません。つまり、スケジュールは各トランザクションの履歴が連続したものになります。このようなスケジュールはシリアル スケジュールと呼ばれます。

同時実行制御は、同時実行性がないため、シリアル スケジュールが正確でなければならないという前提に基づいています。

tx1: setInt(blk, 80, 1, false); setString(blk, 40, "one", fal
se);

tx2: getInt(blk, 80); getString(blk, 40); setInt(blk, 80, new
ival, true); setString(blk, 40, newsval, true);

tx3: getInt(blk, 80)); getString(blk, 40)); setInt(bl
k, 80, 9999, true); getInt(blk, 80));

tx4: getInt(blk, 80));

(ア)

tx1: W(blk); W(blk) tx2: R(blk); R(blk); W(blk); W
(blk) tx3: R(blk); R(blk); W(blk); R(blk) tx4: R(blk
)

（イ）

図5.17 図5.3のトランザクション履歴。(a)データアクセス履歴、(b)ブロックアクセス履歴

シリアルスケジュールに関する正確さの 1 つは、同じトランザクションの異なるシリアルスケジュールが異なる結果をもたらす可能性があることです。たとえば、次の同じ履歴を持つ 2 つのトランザクション T1 と T2 を考えてみましょう。

T1:W(b1); W(b2) T2:
W(b1); W(b2)

これらのトランザクションは同じ履歴を持ちますが (つまり、両方とも最初にブロック b1 を書き込み、次にブロック b2 を書き込みます)、トランザクションとしては必ずしも同一ではありません。たとえば、T1 は各ブロックの先頭に「X」を書き込み、T2 は「Y」を書き込む可能性があります。T1 が T2 より前に実行される場合、ブロックには T2 によって書き込まれた値が含まれますが、逆の順序で実行される場合、ブロックには T1 によって書き込まれた値が含まれます。

この例では、T1 と T2 はブロック b1 と b2 に何を含めるべきかについて異なる意見を持っています。また、データベース エンジンの視点ではすべてのトランザクションが同等であるため、ある結果が他の結果よりも正しいとは言えません。したがって、どちらかのシリアル スケジュールの結果が正しいことを認めざるを得ません。つまり、正しい結果が複数存在する可能性があります。

非シリアルスケジュールは、シリアルスケジュールと同じ結果を生成する場合、シリアル化可能であると言われます。[2]シリアルスケジュールが正しいので、シリアル化可能

[2]The term *serializable* is also used in Java—a serializable class is one whose objects can be written as a stream of bytes. Unfortunately, that use of the term has absolutely nothing to do with the database usage of it.

スケジュールも正確でなければなりません。例として、上記のトランザクションの次の非シリアル スケジュールを考えてみましょう。

W1(b1); W2(b1); W1(b2); W2(b2)

ここで、W1(b1) はトランザクション T1 がブロック b1 を書き込むことを意味します。このスケジュールは、T1 の前半、T2 の前半、T1 の後半、T2 の後半の順に実行された結果です。このスケジュールは、最初に T1 を実行してから T2 を実行するのと同じなので、シリアル化可能です。一方、次のスケジュールを考えてみましょう。

W1(b1); W2(b1); W2(b2); W1(b2)

このトランザクションは、T1 の前半、T2 のすべて、そして T1 の後半を実行します。このスケジュールの結果、ブロック b1 には T2 によって書き込まれた値が含まれますが、ブロック b2 には T1 によって書き込まれた値が含まれます。この結果は、どのシリアル スケジュールでも生成できないため、スケジュールはシリアル化不可能であると言われています。

分離の ACID プロパティを思い出してください。それは、各トランザクションはシステム内の唯一のトランザクションであるかのように実行される必要があるというものです。シリアル化できないスケジュールにはこのプロパティがありません。したがって、シリアル化できないスケジュールは間違っていると認めざるを得ません。言い換えると、スケジュールはシリアル化できる場合にのみ正しいということです。

5.4.2 ロックテーブル

データベース エンジンは、すべてのスケジュールがシリアル化可能であることを確認する責任があります。一般的な手法は、ロックを使用してトランザクションの実行を延期することです。セクション 5.4.3 では、ロックを使用してシリアル化可能性を確保する方法について説明します。このセクションでは、基本的なロック メカニズムの仕組みについて簡単に説明します。

各ブロックには、共有ロック (または slock) と排他ロック (または xlock) の 2 種類のロックがあります。トランザクションがブロックに対して排他ロックを保持している場合、他のトランザクションはそのブロックに対していかなる種類のロックも保持できません。トランザクションがブロックに対して共有ロックを保持している場合、他のトランザクションはそのブロックに対して共有ロックのみを保持できます。これらの制限は他のトランザクションにのみ適用されることに注意してください。1 つのトランザクションは、ブロックに対して共有ロックと排他ロックの両方を保持できます。

ロック テーブルは、トランザクションにロックを付与する役割を担うデータベース エンジン コンポーネントです。SimpleDB クラス LockTable はロック テーブルを実装します。その API は図 5.18 に示されています。

LockTable

パブリック void sLock(ブロック blk); パブリック void xLock(ブロック blk); パブリック void unlock(ブロック blk);

図5.18 SimpleDBクラスLockTableのAPI

メソッド sLock は、指定されたブロックの共有ロックを要求します。ブロックにすでに排他ロックが存在する場合、メソッドは排他ロックが解除されるまで待機します。メソッド xLock は、ブロックの排他ロックを要求します。このメソッドは、他のトランザクションがブロックにロックをかけなくなるまで待機します。unlock メソッドは、ブロックのロックを解除します。

図 5.19 は、ロック要求間のいくつかの相互作用を示す ConcurrencyTest クラスを示しています。

パブリック クラス ConcurrencyTest { プライベート 静的 FileMgr fm; プライベート 静的 LogMgr lm; プライベート 静的 BufferMgr bm;

public static void main(String[] args) { //データベースエンジンを初期化しますSimpleDB db = new SimpleDB("concurrencytest", 400, 8);

fm = db.fileMgr(); lm = db.logMgr(); bm = db.bufferMgr(); A a = new A(); new Thread(a).start(); B b = new B(); new Thread(b).start(); C c = new C(); new Thread(c).start(); }static class A implements Runnable { public void run() { try { Transaction txA = new Transaction(fm, lm, bm); BlockId blk1 = new BlockId("testfile", 1); BlockId blk2 = new BlockId("testfile", 2); txA.pin(blk1); txA.pin(blk2); System.out.println("Tx A: request slock 1"); txA.getInt(blk1, 0); System.out.println("Tx A: slock 1 を受信"); Thread.sleep(1000); System.out.println("Tx A: slock 2 を要求"); txA.getInt(blk2, 0); System.out.println("Tx A: slock 2 を受信"); txA.commit(); }catch(InterruptedException e) {}; } }static class B implements Runnable { public void run() { try {Transaction txB = new Transaction(fm, lm, bm);

図5.19 ロック要求間の相互作用のテスト

```
BlockId blk1 = 新しい BlockId("testfile", 1); BlockId blk2 = 新しい BlockId("testfile", 2); txB.pin(blk1); txB.pin(blk2); System.out.println("Tx B: request xlock 2"); txB.setInt(blk2, 0, 0, false); System.out.println("Tx B: received xlock 2"); Thread.sleep(1000); System.out.println("Tx B: request slock 1"); txB.getInt(blk1, 0); System.out.println("Tx B: received slock 1"); txB.commit(); }catch (InterruptedException e) {}; } }static class C implements Runnable { public void run() { try {Transaction txC = new Transaction(fm, lm, bm); BlockId blk1 = new BlockId("testfile", 1); BlockId blk2 = new BlockId("testfile", 2); txC.pin(blk1); txC.pin(blk2); System.out.println("Tx C: request xlock 1"); txC.setInt(blk1, 0, 0, false); System.out.println("Tx C: received xlock 1"); Thread.sleep(1000); System.out.println("Tx C: request slock 2"); txC.getInt(blk2, 0); System.out.println("Tx C: received slock 2"); txC.commit(); }catch(InterruptedException e) {}; } }
```

}

図5.19（続き）

main メソッドは、クラス A、B、C のそれぞれのオブジェクトに対応する 3 つの同時スレッドを実行します。これらのトランザクションは、ブロックを明示的にロックおよびロック解除しません。代わりに、Transaction の getInt メソッドは slock を取得し、setInt メソッドは xlock を取得し、commit メソッドはすべてのロックを解除します。各トランザクションのロックとロック解除のシーケンスは次のようになります。

txA: sLock(blk1); sLock(blk2); unlock(blk1); unlock(blk2) txB: xLock(blk2); sLock(blk1); unlock(blk1); unlock(blk2) txC: xLock(blk1); sLock(blk2); unlock(blk1); unlock(blk2)

スレッドには、トランザクションにロック要求を交互に実行させるスリープ ステートメントがあります。次の一連のイベントが発生します。

1. スレッド A が blk1 で slock を取得します。 2. スレッド B が blk2 で xlock を取得します。 3. スレッド C は、他の誰かが blk1 をロックしているため、blk1 で xlock を取得できません。そのため、スレッド C は待機します。 4. スレッド A は、他の誰かが blk2 を xlock しているため、blk2 で slock を取得できません。そのため、スレッド A も待機します。 5. スレッド B は続行できます。現在誰もブロック blk1 に xlock を持っていないため、スレッド B はブロック blk1 で slock を取得します (スレッド C がそのブロックで xlock を待機していることは関係ありません)。 6. スレッド B はブロック blk1 のロックを解除しますが、これはどちらの待機中のスレッドにも役立ちません。 7. スレッド B はブロック blk2 のロックを解除します。 8. スレッド A は続行できるようになり、blk2 で slock を取得します。 9. スレッド A はブロック blk1 のロックを解除します。

10. スレッド C は最終的に blk1 の xlock を取得できます。
11. スレッド A と C は、完了するまで任意の順序で続行できます。

5.4.3 ロックプロトコル

ここで、すべてのスケジュールがシリアル化可能であることを保証するためにロックをどのように使用できるかという問題に取り組みます。次の履歴を持つ 2 つのトランザクションを考えてみましょう。

T1: R(b1); W(b2) T2:
W(b1); W(b2)

シリアル スケジュールが異なる結果になる原因は何でしょうか。トランザクション T1 と T2 は両方とも同じブロック b2 に書き込みます。つまり、これらの操作の順序が重要になります。つまり、最後に書き込みを行うトランザクションが「勝者」になります。操作 {W1(b2), W2(b2)} は競合していると言われています。一般に、2 つの操作は、実行順序によって結果が異なる場合に競合します。2 つのトランザクションに競合する操作がある場合、シリアル スケジュールは異なる (ただし、どちらも正しい) 結果になる可能性があります。

この競合は、書き込み-書き込み競合の例です。2 つ目の種類の競合は、読み取り-書き込み競合です。たとえば、操作 {R1(b1), W2(b1)} は競合します。R1(b1) が最初に実行される場合、T1 はブロック b1 の 1 つのバージョンを読み取りますが、W2(b1) が最初に実行される場合、T1 はブロック b1 の別のバージョンを読み取ります。2 つの読み取り操作が競合することはなく、異なるブロックに関係する操作も競合しないことに注意してください。

競合に注意する必要がある理由は、それがスケジュールの直列化可能性に影響を与えるからです。非直列スケジュールで競合する操作が実行される順序によって、同等の直列スケジュールがどうなるかが決まります。上記の例では、W2(b1)がR1(b1)の前に実行される場合、同等の直列スケジュールは

1. ブロックを読み取る前に、そのブロックの共有ロックを取得します。 2. ブロックを変更する前に、そのブロックの排他ロックを取得します。 3. コミットまたはロールバック後にすべてのロックを解除します。

図5.20 ロックプロトコル

T2 は T1 より先に実行する必要があります。一般に、T1 で T2 と競合するすべての操作を考慮すると、それらの操作はすべて、競合する T2 操作の前に実行されるか、それらの後に実行される必要があります。競合しない操作は、任意の順序で発生する可能性があります。[3]

ロックを使用すると、書き込みと書き込み、読み取りと書き込みの競合を回避できます。特に、すべてのトランザクションが図 5.20 のプロトコルに従ってロックを使用すると仮定します。

このプロトコルから、2 つの重要な事実を推測できます。まず、トランザクションがブロックの共有ロックを取得した場合、他のアクティブなトランザクションはそのブロックに書き込むことはできません (そうでない場合は、一部のトランザクションがブロックの排他ロックを保持したままになります)。次に、トランザクションがブロックの排他ロックを取得した場合、他のアクティブなトランザクションはそのブロックに何らかの方法でアクセスすることはできません (そうでない場合は、一部のトランザクションがブロックのロックを保持したままになります)。これらの事実は、トランザクションによって実行される操作が、別のアクティブなトランザクションによる以前の操作と競合することはないことを意味します。つまり、すべてのトランザクションがロック プロトコルに従う場合、次のようになります。

- 結果のスケジュールは常にシリアル化可能 (したがって正しい) になります。
- 等価シリアルスケジュールは、トランザクションがコミットされる順序によって決定される。

トランザクションが完了するまでロックを保持するように強制することで、ロック プロトコルはシステムの同時実行性を大幅に制限します。トランザクションが不要になったときにロックを解放できれば、他のトランザクションがそれほど長く待たなくて済みます。ただし、トランザクションが完了する前にロックを解放すると、2 つの深刻な問題が発生する可能性があります。トランザクションがシリアル化できなくなる可能性があり、他のトランザクションがコミットされていない変更を読み取ることができる可能性があります。次に、この 2 つの問題について説明します。

5.4.3.1 直列化可能性の問題

トランザクションがブロックのロックを解除すると、シリアル化可能性に影響を与えずに別のブロックをロックできなくなります。理由を理解するには、ブロック y をロックする前にブロック x のロックを解除するトランザクション T1 を考えてみましょう。

T1: ... R(x); UL(x); SL(y); R(y); ...

[3] Actually, you can construct obscure examples in which certain write-write conflicts can also occur in any order; see Exercise 5.26. Such examples, however, are not practical enough to be worth considering.

T1 が x のロック解除と y のロック解除の間の時間間隔で中断されたとします。この時点では、x と y の両方がロック解除されているため、T1 は非常に脆弱です。別のトランザクション T2 が介入し、x と y の両方をロックし、それらに書き込み、コミットし、ロックを解除するとします。次の状況が発生しています。T1 は、T2 が書き込む前に存在していたブロック x のバージョンを読み取るため、シリアル順序では T2 の前に来なければなりません。一方、T1 はシリアル順序では T2 の後に来なければなりません。T1 は、T2 によって書き込まれたブロック y のバージョンを読み取るためです。したがって、結果として生じるスケジュールはシリアル化できません。

逆もまた真であることが証明されています。つまり、トランザクションがロックを解除する前にすべてのロックを取得した場合、結果のスケジュールはシリアル化可能であることが保証されます (演習 5.27 を参照)。このロック プロトコルのバリエーションは、2 フェーズ ロックと呼ばれます。この名前は、このプロトコルでは、トランザクションに 2 つのフェーズ (ロックを蓄積するフェーズとロックを解放するフェーズ) があることに由来しています。

2 フェーズ ロックは理論的にはより一般的なプロトコルですが、データベース エンジンはこれを簡単に利用できません。通常、トランザクションが最終ブロックへのアクセスを完了する (ロックが最終的に解放される) までには、コミットする準備が整っています。そのため、完全に一般的な 2 フェーズ ロック プロトコルは、実際にはほとんど効果がありません。

5.4.3.2 コミットされていないデータの読み取り

ロックを早期に解放することによるもう 1 つの問題は (2 フェーズ ロックを使用している場合でも)、トランザクションがコミットされていないデータを読み取ることができるようになることです。次の部分的なスケジュールを検討してください。

... W1(b); UL1(b); SL2(b); R2(b); ...

このスケジュールでは、T1 はブロック b に書き込み、それをロック解除します。次にトランザクション T2 がブロック b をロックして読み取ります。T1 が最終的にコミットした場合は、問題はありません。しかし、T1 がロールバックを実行するとします。その場合、T2 もロールバックする必要があります。これは、その実行がもはや存在しない変更に基づいているためです。また、T2 がロールバックすると、他のトランザクションもロールバックする可能性があります。この現象はカスケード ロールバックと呼ばれます。

データベース エンジンがトランザクションにコミットされていないデータの読み取りを許可すると、同時実行性は高まりますが、データを書き込んだトランザクションがコミットされないというリスクを負うことになります。確かに、ロールバックは頻繁に行われない傾向があり、カスケード ロールバックはさらに頻繁に行われるべきです。問題は、データベース エンジンが、トランザクションを不必要にロールバックするリスクを負うかどうかです。ほとんどの商用データベース システムでは、このリスクを負うことを望まないため、排他ロックを解放する前に、常にトランザクションが完了するまで待機します。

5.4.4 デッドロック

ロック プロトコルはスケジュールがシリアル化可能であることを保証しますが、すべてのトランザクションがコミットされることを保証するものではありません。特に、トランザクションがデッドロックになる可能性があります。セクション 4.5.1 では、2 つのクライアント スレッドがそれぞれ他方のバッファの解放を待機しているデッドロックの例を示しました。ロックについても同様の可能性があります。デッドロックは、最初のトランザクションが 2 番目のトランザクションが保持しているロックを待機し、2 番目のトランザクションが 3 番目のトランザクションが保持しているロックを待機し、最後のトランザクションが最初のトランザクションが保持しているロックを待機するまで、トランザクションのサイクルが存在する場合に発生します。このような場合、待機中のトランザクションはいずれも続行できず、すべてが潜在的に永久に待機することになります。簡単な例として、トランザクションが同じブロックに異なる順序で書き込む次の 2 つの履歴を考えてみましょう。

T1:W(b1); W(b2) T2:
W(b2); W(b1)

T1 が最初にブロック b1 のロックを取得したとします。これでブロック b2 のロックをめぐって競合が発生します。T1 が最初にロックを取得した場合、T2 は待機し、最終的に T1 はコミットしてロックを解除し、T2 は続行できます。問題ありません。しかし、T2 が最初にブロック b2 のロックを取得した場合、デッドロックが発生します。つまり、T1 は T2 がブロック b2 のロックを解除するのを待機し、T2 は T1 がブロック b1 のロックを解除するのを待機します。どちらのトランザクションも続行できません。

並行性マネージャは「待機」グラフを維持することでデッドロックを検出できます。このグラフには、トランザクションごとに 1 つのノードと、T1 が T2 が保持するロックを待機している場合は T1 から T2 へのエッジがあります。各エッジには、トランザクションが待機しているブロックのラベルが付けられます。ロックが要求または解放されるたびに、グラフが更新されます。たとえば、上記のデッドロック シナリオに対応する待機グラフは、図 5.21 に示されています。

デッドロックが存在するのは、待機グラフにサイクルがある場合に限ります。演習 5.28 を参照してください。トランザクション マネージャーがデッドロックの発生を検出すると、サイクル内のトランザクションのいずれかを即座にロールバックすることでデッドロックを解除できます。合理的な戦略は、サイクルを「引き起こした」ロック要求のトランザクションをロールバックすることですが、他の戦略も可能です。演習 5.29 を参照してください。

バッファを待機しているスレッドとロックを待機しているスレッドの両方を考慮すると、デッドロックのテストはかなり複雑になります。たとえば、バッファ プールにバッファが 2 つしか含まれていないと仮定し、次のシナリオを検討します。

```
T1: xlock(b1); pin(b4)
T2: pin(b2); pin(b3); xlock(b1)
```

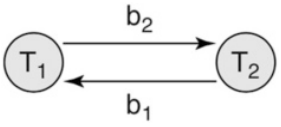


図5.21 待機グラフ

トランザクション T1 がブロック b1 のロックを取得した後に中断され、その後 T2 がブロック b2 と b3 をピン留めするとします。T2 は xlock(b1) の待機リストに載り、T1 はバッファの待機リストに載ります。待機グラフが非循環であっても、デッドロックが発生します。

このような状況でデッドロックを検出するには、ロック マネージャは待機グラフを保持するだけでなく、どのトランザクションがどのバッファを待機しているかを知る必要があります。この追加の考慮事項をデッドロック検出アルゴリズムに組み込むことは、かなり難しいことがわかります。冒険心のある読者は、演習 5.37 に挑戦することをお勧めします。

デッドロックを検出するために waits-for グラフを使用する際の問題は、グラフの保守が多少難しく、グラフ内のサイクルを検出するプロセスに時間がかかることです。その結果、デッドロック検出を近似するより単純な戦略が開発されました。これらの戦略は、常にデッドロックを検出するという意味で保守的ですが、デッドロックでない状況をデッドロックとして扱う可能性もあります。このセクションでは、そのような 2 つの戦略を検討します。演習 5.33 では、別の戦略を検討します。

最初の近似戦略は wait-die と呼ばれ、図 5.22 で定義されています。この戦略では、waits-for グラフには古いトランザクションから新しいトランザクションへのエッジのみが含まれるため、デッドロックが発生しないことが保証されます。ただし、この戦略では、すべての潜在的なデッドロックをロールバックの原因として扱います。たとえば、トランザクション T1 が T2 よりも古く、T2 が現在 T1 によって保持されているロックを要求しているとします。この要求がすぐにデッドロックを引き起こすことはないかもしれませんが、後になって T1 が T2 によって保持されているロックを要求する可能性があるため、デッドロックが発生する可能性があります。したがって wait-die 戦略では、事前に T2 をロールバックします。

2 番目の近似戦略は、時間制限を使用してデッドロックの可能性を検出することです。トランザクションが事前に設定された時間待機している場合、トランザクション マネージャーはデッドロックが発生していると想定し、ロールバックします。図 5.23 を参照してください。

デッドロック検出戦略にかかわらず、並行性マネージャはアクティブなトランザクションをロールバックしてデッドロックを解除する必要があります。

T1 が、T2 が保持しているロックと競合するロックを要求したとします。T1 が T2 より古い場合、T1 はロックを待機します。それ以外の場合、T1 はロールバックされます (つまり、「終了」します)。

図5.22 ウェイトダイデッドロック検出戦略

T1 が T2 が保持しているロックと競合するロックを要求したとします。

1. T1 はロックを待機します。2. T1 が待機リストに長く留まると、T1 はロールバックされます。

図5.23 時間制限デッドロック検出戦略

そのトランザクションのロックを解除すると、残りのトランザクションを完了できるようになります。トランザクションがロールバックされると、同時実行マネージャは例外をスローします。SimpleDB では、この例外は LockAbortException と呼ばれます。第 4 章の BufferAbortException と同様に、この例外は中止されたトランザクションの JDBC クライアントによってキャッチされ、クライアントがその例外の処理方法を決定します。たとえば、クライアントは単に終了することを選択することも、トランザクションを再度実行することもできます。

5.4.5 ファイルレベルの競合とファントム

この章ではこれまで、ブロックの読み取りと書き込みから生じる競合について検討してきました。別の種類の競合は、ファイル終了マーカーの読み取りと書き込みを行う size メソッドと append メソッドに関係しています。これら 2 つのメソッドは明らかに互いに競合しています。トランザクション T1 がトランザクション T2 が size を呼び出す前に append を呼び出すと仮定すると、T1 はどのような順序でも T2 より前に来なければなりません。この競合の結果の 1 つは、ファントム問題として知られています。T2 がファイルの内容全体を繰り返し読み取り、各反復の前に size を呼び出して、読み取るブロックの数を決定するとします。さらに、T2 が最初にファイルを読み取った後、トランザクション T1 がファイルにいくつかの追加ブロックを追加し、値を入力してコミットするとします。次にファイルを読み取ったとき、T2 はこれらの追加値を認識しますが、これは ACID の分離特性に違反しています。これらの追加値は、T2 にとって不可解に現れるため、ファントムと呼ばれます。

同時実行マネージャはどのようにしてこの競合を回避できるでしょうか。ロック プロトコルでは、T2 が読み取るブロックごとに slock を取得する必要があるため、T1 はそれらのブロックに新しい値を書き込むことができません。ただし、このアプローチはここでは機能しません。T1 が新しいブロックを作成する前に、T2 が新しいブロックを slock する必要があるためです。

解決策は、トランザクションがファイル終了マーカーをロックできるようにすることです。特に、トランザクションは、append メソッドを呼び出すためにマーカーを xlock する必要があり、size メソッドを呼び出すためにマーカーを slock する必要があります。上記のシナリオでは、T1 が最初に append を呼び出すと、T2 は T1 が完了するまでファイル サイズを判別できません。逆に、T2 がすでにファイル サイズを判別している場合は、T2 がコミットするまで T1 は追加をブロックされます。どちらの場合も、ファントムは発生しません。

5.4.6 マルチバージョンロック

多くのデータベースアプリケーションにおけるトランザクションは読み取り専用です。読み取り専用トランザクションは、ロックを共有し、お互いを待つ必要がないため、データベースエンジン内でうまく共存できます。ただし、更新トランザクションとはうまく共存できません。1 つの更新トランザクションがブロックに書き込みを行っているとします。その場合、すべての読み取り専用トランザクションは

そのブロックを読み取るトランザクションは、ブロックが書き込まれるまでだけでなく、更新トランザクションが完了するまで待機する必要があります。逆に、更新トランザクションがブロックを書き込む場合は、ブロックを読み取るすべての読み取り専用トランザクションが完了するまで待機する必要があります。

言い換えると、読み取り専用トランザクションと更新トランザクションが競合すると、どちらのトランザクションが最初にロックを取得するかに関係なく、多くの待機が発生します。この状況はよくあることなので、研究者はこの待機を減らすための戦略を開発しました。そのような戦略の 1 つが、マルチバージョン ロックと呼ばれます。

5.4.6.1 マルチバージョンロックの原則

名前が示すように、マルチバージョン ロックは各ブロックの複数のバージョンを保存することによって機能します。基本的な考え方は次のとおりです。

- ブロックの各バージョンには、それを書き込んだトランザクションのコミット時刻がタイムスタンプとして付けられます。
- 読み取り専用トランザクションがブロックから値を要求すると、同時実行マネージャーはトランザクションの開始時に最後にコミットされたブロックのバージョンを使用します。

つまり、読み取り専用トランザクションでは、トランザクションが開始された時点のコミットされたデータのスナップショットが表示されます。「コミットされたデータ」という用語に注意してください。トランザクションでは、開始前にコミットされたトランザクションによって書き込まれたデータが表示されますが、それ以降のトランザクションによって書き込まれたデータは表示されません。

マルチバージョン ロックの次の例を考えてみましょう。4 つのトランザクションに次の履歴があるとします。

T1:W(b1); W(b2) T2:
W(b1); W(b2) T3: R(b
1); R(b2) T4: W(b2)

以下のスケジュールに従って実行されます。

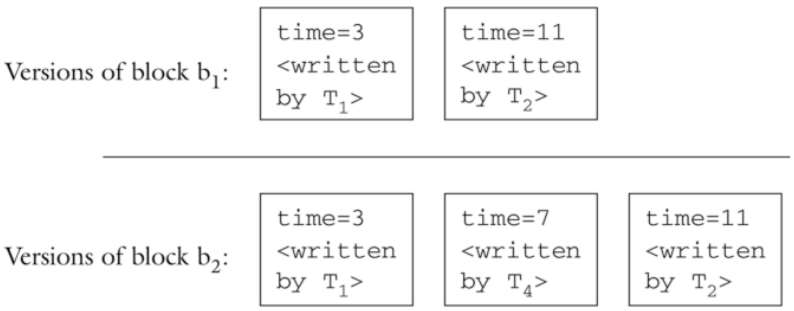


図5.24 マルチバージョン同時実行

W1(b1); W1(b2); C1; W2(b1); R3(b1); W4(b2); C4; R3(b2); C3; W2(b1); C2

このスケジュールでは、トランザクションが最初の操作で開始され、必要になる直前にロックを取得することを前提としています。操作 Ci は、トランザクション Ti がコミットするタイミングを示します。更新トランザクション T1、T2、および T4 は、スケジュールから確認できるように、ロック プロトコルに従います。トランザクション T3 は読み取り専用トランザクションであり、プロトコルに従いません。

同時実行マネージャは、ブロックに書き込む更新トランザクションごとにブロックのバージョンを保存します。したがって、図 5.24 に示すように、b1 のバージョンは 2 つ、b2 のバージョンは 3 つ存在することになります。各バージョンのタイムスタンプは、書き込みが発生した時刻ではなく、トランザクションがコミットされた時刻です。各操作に 1 時間単位かかると仮定すると、T1 は時刻 3、T4 は時刻 7、T3 は時刻 9、T2 は時刻 11 にコミットされます。

さて、読み取り専用トランザクション T3 について考えてみましょう。これは時刻 5 で開始されるため、その時点でコミットされた値、つまり T1 によって行われた変更 (T2 または T4 によって行われた変更ではない) を参照する必要があります。したがって、時刻 3 にタイムスタンプが付けられた b1 と b2 のバージョンを参照します。読み取りが発生した時点でそのバージョンがコミットされていたにもかかわらず、T3 は時刻 7 にタイムスタンプが付けられた b2 のバージョンを参照しないことに注意してください。

マルチバージョン ロックの利点は、読み取り専用トランザクションはロックを取得する必要がないため、待機する必要がないことです。同時実行マネージャは、トランザクションの開始時間に応じて、要求されたブロックの適切なバージョンを選択します。更新トランザクションは同じブロックに同時に変更を加えることができますが、読み取り専用トランザクションは、そのブロックの異なるバージョンを参照するため、気にしません。

マルチバージョン ロックは読み取り専用トランザクションにのみ適用されます。更新トランザクションはロック プロトコルに従い、必要に応じて slock と xlock の両方を取得する必要があります。これは、すべての更新トランザクションがデータの現在のバージョン (以前のバージョンではない) を読み書きするため、競合が発生する可能性があるためです。ただし、これらの競合は更新トランザクション間でのみ発生し、読み取り専用トランザクションとは発生しないことに注意してください。したがって、競合する更新トランザクションが比較的少ないと仮定すると、待機の頻度は大幅に少なくなります。

5.4.6.2 マルチバージョンロックの実装

マルチバージョン ロックがどのように機能するかを確認したので、並行性マネージャが必要な処理をどのように実行するかを調べてみましょう。基本的な問題は、各ブロックのバージョンをどのように維持するかです。単純ですがやや難しいアプローチは、各バージョンを専用の「バージョン ファイル」に明示的に保存することです。別のアプローチは、ログを使用してブロックの任意のバージョンを再構築することです。その実装は次のように機能します。

各読み取り専用トランザクションには、開始時にタイムスタンプが付与されます。各更新トランザクションには、コミット時にタイムスタンプが付与されます。更新トランザクションのコミット メソッドは、次のアクションを含むように改訂されました。

- リカバリ マネージャーは、トランザクションのタイムスタンプをコミット ログ レコードの一部として書き込みます。
- トランザクションによって保持される各 xlock について、同時実行マネージャーはブロックを固定し、ブロックの先頭にタイムスタンプを書き込み、バッファーの固定を解除します。

タイムスタンプ t を持つ読み取り専用トランザクションがブロック b を要求するとします。同時実行マネージャは、適切なバージョンを再構築するために次の手順を実行します。

- ブロック b の現在のバージョンを新しいページにコピーします。
- 次のように、ログを 3 回逆方向に読み取ります。

  – 時刻 t 以降にコミットされたトランザクションのリストを作成します。トランザクションはタイムスタンプ順にコミットされるため、並行性マネージャは、タイムスタンプが t より小さいコミット レコードを見つけると、ログの読み取りを停止できます。 – コミット レコードまたはロールバック レコードがないトランザクションによって書き込まれたログ レコードを検索して、未完了のトランザクションのリストを作成します。静止チェックポイント レコード、または非静止チェックポイント レコード内のトランザクションの最も古い開始レコードを見つけると、ログの読み取りを停止できます。 – 更新レコードを使用して、b のコピー内の値を元に戻します。上記のいずれかのリストにあるトランザクションによって書き込まれた b の更新レコードを見つけると、元に戻す操作を実行します。リストにある最も古いトランザクションの開始レコードを見つけると、ログの読み取りを停止できます。
- b の変更されたコピーがトランザクションに返されます。

言い換えると、並行性マネージャは、t より前にコミットされなかったトランザクションによって行われた変更を元に戻すことによって、時刻 t のブロックのバージョンを再構築します。このアルゴリズムでは、簡単にするためにログを 3 回通過します。演習 5.38 では、アルゴリズムを書き直してログを 1 回通過するように求められます。

最後に、同時実行マネージャは 2 種類のトランザクションを別々に扱うため、トランザクションは読み取り専用かどうかを指定する必要があります。JDBC では、この指定は Connection インターフェイスの setReadOnly メソッドによって実行されます。例:

```
接続 conn = ... // 接続を取得します conn.setReadOnly(true);
```

setReadOnly の呼び出しは、データベース システムへの「ヒント」と見なされます。システムは、マルチバージョン ロックをサポートしていない場合、呼び出しを無視することを選択できます。

ISOLATION LEVEL	PROBLEMS	LOCK USAGE	COMMENTS
serializable	none	slocks held to completion, slock on eof marker	the only level that guarantees correctness
repeatable read	phantoms	slocks held to completion, no slock on eof marker	useful for modify-based transactions
read committed	phantoms, values may change	slocks released early, no slock on eof marker	useful for conceptually separable transactions whose updates are “all or nothing”
read uncommitted	phantoms, values may change, dirty reads	no slocks at all	useful for read-only transactions that tolerate inaccurate results

図5.25 トランザクション分離レベル

5.4.7 トランザクション分離レベル

ロック プロトコルではトランザクションが完了するまでロックを保持することが必要となるため、直列化可能性を強制すると、かなりの待機時間が発生します。その結果、トランザクション T1 が、T2 が保持するロックと競合する 1 つのロックだけを必要とする場合、T2 が完了するまで T1 は他の操作を実行できません。

マルチバージョン ロックは、読み取り専用トランザクションをロックなしで実行できるため、待機する手間が省けるという点で非常に魅力的です。ただし、マルチバージョン ロックの実装はやや複雑で、バージョンを再作成するために追加のディスク アクセスが必要になります。さらに、マルチバージョン ロックは、データベースを更新するトランザクションには適用されません。

トランザクションがロックを待つ時間を短縮する別の方法があります。それは、完全な直列化可能性を必要としないことを指定することです。第 2 章では、JDBC の 4 つのトランザクション分離レベルについて説明しました。図 5.25 は、これらのレベルとその特性をまとめたものです。

第 2 章では、これらの分離レベルと、発生する可能性のあるさまざまな問題とを関連付けました。図 5.25 の新しい点は、これらのレベルと slock の使用方法も関連付けていることです。直列化可能分離では非常に制限の厳しい共有ロックが必要ですが、読み取り非コミット分離では slock さえ使用されません。明らかに、ロックの制限が少ないほど、待機時間が少なくなります。ただし、制限の少ないロックでは、クエリの結果に不正確さも生じます。トランザクションでファントムが見られたり、異なる時間に 1 つの場所で 2 つの異なる値が見られたり、コミットされていないトランザクションによって書き込まれた値が見られたりする可能性があります。

これらの分離レベルはデータの読み取りにのみ適用されることを強調しておきます。すべてのトランザクションは、分離レベルに関係なく、データの書き込みに関しては正しく動作する必要があります。トランザクションは適切な xlock (EOF マーカーの xlock を含む) を取得し、完了するまで保持する必要があります。その理由は、個々のトランザクションはクエリを実行するときに不正確さを許容することを選択できますが、不正確な更新はデータベース全体に悪影響を及ぼすため、許容できないためです。

読み取り非コミット分離とマルチバージョン ロックの違いは何でしょうか。どちらも読み取り専用トランザクションに適用され、ロックなしで動作します。ただし、読み取り非コミット分離を使用するトランザクションでは、どのトランザクションがいつ書き込んだかに関係なく、読み取った各ブロックの現在の値を参照します。これはシリアル化にはほど遠いものです。一方、マルチバージョン ロックを使用するトランザクションでは、ある時点でブロックのコミットされた内容を参照するため、シリアル化可能です。

5.4.8 データ項目の粒度

この章では、同時実行マネージャがブロックをロックすることを前提としています。ただし、他のロック粒度も可能です。同時実行マネージャは、値、ファイル、またはデータベース全体をロックすることもできます。ロックの単位は、同時実行データ項目と呼ばれます。

同時実行制御の原則は、使用されるデータ項目の粒度によって影響を受けません。この章のすべての定義、プロトコル、およびアルゴリズムは、任意のデータ項目に適用されます。したがって、粒度の選択は、効率と柔軟性のバランスをとる実用的な選択です。このセクションでは、これらのトレードオフのいくつかについて説明します。

同時実行マネージャは、各データ項目のロックを保持します。粒度サイズが小さいほど、同時実行性が高まるため便利です。たとえば、2 つのトランザクションが同じブロックの異なる部分を同時に変更するとします。これらの同時変更は、値粒度ロックでは可能ですが、ブロック粒度ロックでは不可能です。

ただし、粒度を小さくすると、より多くのロックが必要になります。値では膨大な数のロックが必要になるため、データ項目が非現実的に小さくなる傾向があります。反対に、ファイルをデータ項目として使用すると、必要なロックはごくわずかですが、同時実行性に大きく影響します。クライアントは、ファイルの一部を更新するには、ファイル全体を xlock する必要があります。ブロックをデータ項目として使用するのは、妥当な妥協策です。

余談ですが、一部のオペレーティング システム (MacOS や Windows など) では、ファイルの粒度ロックを使用して、同時実行制御の原始的な形式を実装していることに注意してください。特に、アプリケーションは、ファイルに xlock がないとファイルに書き込むことができず、そのファイルが現在別のアプリケーションによって使用されている場合は slock を取得できません。

一部の同時実行マネージャは、ブロックやファイルなどの複数の粒度でデータ項目をサポートします。ファイルのいくつかのブロックのみにアクセスする予定のトランザクションは、それらを個別にロックできますが、トランザクションがファイルのすべて (またはほとんど) にアクセスする予定の場合は、単一のファイル粒度ロックを取得します。このアプローチは、小粒度項目の柔軟性と高レベル項目の利便性を融合します。

もう 1 つの粒度は、データ レコードを並行データ項目として使用することです。データ レコードはレコード マネージャによって処理されます。レコード マネージャについては次の章で説明します。SimpleDB は並行マネージャがレコードを理解できないように構造化されているため、レコードをロックできません。ただし、一部の商用システム (Oracle など) は並行マネージャがレコード マネージャを認識し、そのメソッドを呼び出すことができるように構築されています。この場合、データ レコードは適切な並行データ項目になります。

ConcurrencyMgr
パブリック ConcurrencyMgr(int txnum); パブリック void sLock(Block blk); パブリック void xLock(Block blk); パブリック void release();

図5.26 SimpleDB同時実行マネージャのAPI

データ レコードの粒度は魅力的に見えますが、ファントムに関する追加の問題を引き起こします。新しいデータ レコードが既存のブロックに挿入される可能性があるため、ブロックからすべてのレコードを読み取るトランザクションでは、他のトランザクションがそのブロックにレコードを挿入しないようにする方法が必要です。解決策は、同時実行マネージャがブロックやファイルなどのより粗い粒度のデータ項目もサポートすることです。実際、一部の商用システムでは、挿入を実行する前にトランザクションにファイルの xlock を取得するように強制するだけでファントムを回避しています。

5.4.9 SimpleDB同時実行マネージャ

SimpleDB 同時実行マネージャは、パッケージ simpledb.tx.concurrency のクラス ConcurrencyMgr を介して実装されます。同時実行マネージャは、ブロック レベルの粒度を使用してロック プロトコルを実装します。その API は、図 5.26 に示されています。

各トランザクションには、独自の同時実行マネージャがあります。同時実行マネージャのメソッドは、ロック テーブルのメソッドと似ていますが、トランザクション固有です。各 ConcurrencyMgr オブジェクトは、そのトランザクションによって保持されているロックを追跡します。メソッド sLock および xLock は、トランザクションがまだロックを持っていない場合のみ、ロック テーブルからロックを要求します。メソッド release は、トランザクションの終了時に呼び出され、すべてのロックを解除します。

ConcurrencyMgr クラスは、SimpleDB ロック テーブルを実装する LockTable クラスを使用します。このセクションの残りの部分では、これら 2 つのクラスの実装について説明します。

5.4.9.1 クラス LockTable

LockTable クラスのコードは図 5.27 に示されています。LockTable オブジェクトは、locks と呼ばれる Map 変数を保持します。このマップには、現在ロックが割り当てられている各ブロックのエントリが含まれます。エントリの値は Integer オブジェクトになります。値 -1 は排他ロックが割り当てられていることを示し、正の値は割り当てられている共有ロックの現在の数を示します。

sLockメソッドとxLockメソッドは、BufferMgrのpinメソッドと非常によく似た動作をします。各メソッドはループ内でJavaのwaitメソッドを呼び出します。

```
cl ass LockTable { private static final long MAX_TIME = 10000; // 10 秒 private Map<Block,Inte
   ger> locks = new HashMap<Block,Integer> ( ) ; public synchronized void sLock(Block blk) { tr
   y {long timestamp = System.currentTimeMillis(); while (hasXlock(blk) && !waitingTooLong(ti
   mestamp)) wait(MAX_TIME); if (hasXlock(blk)) throw new LockAbortException(); int val = g
   etLockVal(blk); // 負の値にはなりません locks.put(blk, val+1); }catch(InterruptedException e
   ) { throw new LockAbortException(); } }public synchronized void xLock(Block blk) { try {long
   timestamp = System.currentTimeMillis(); while (hasOtherSLocks(blk) && !waitingTooLong(ti
   mestamp)) wait(MAX_TIME); if (hasOtherSLocks(blk)) throw new LockAbortException(); loc
   ks.put(blk, -1); }catch(InterruptedException e) { throw new LockAbortException(); } }public sy
   nchronized void unlock(Block blk) { int val = getLockVal(blk); if (val > 1) locks.put(blk, val-1)
   ; else { locks.remove(blk); notificationAll(); } }private boolean hasXlock(Block blk) { return get
   LockVal(blk) < 0; }
```

図5.27 SimpleDBクラスLockTableのコード

```
private boolean hasOtherSLocks(Block blk) { return getLockVal(blk) > 1; }private boole
an waitingTooLong(long starttime) { return System.currentTimeMillis() - starttime > M
AX_TIME; }private int getLockVal(Block blk) { Integer ival = locks.get(blk); return (iv
al == null) ? 0 : ival.intValue(); }
```

}

図5.27（続き）

ループ条件が成立する限り、クライアント スレッドは待機リストに継続的に配置されます。sLock のループ条件は hasXlock メソッドを呼び出します。このメソッドは、ブロックの locks に値が -1 のエントリがある場合に true を返します。xLock のループ条件は hasOtherLocks メソッドを呼び出します。このメソッドは、ブロックの locks に値が 1 より大きいエントリがある場合に true を返します。その理由は、同時実行マネージャは xlock を要求する前に常にブロックの slock を取得するため、値が 1 より大きい場合は、他のトランザクションもこのブロックをロックしていることを示すためです。

unlock メソッドは、指定されたロックをロック コレクションから削除するか (排他ロックまたは 1 つのトランザクションだけが保持する共有ロックの場合)、ロックを共有しているトランザクションの数を減らします。ロックがコレクションから削除されると、メソッドは Java の notificationAll メソッドを呼び出し、すべての待機中のスレッドをスケジュールの準備完了リストに移動します。内部の Java スレッド スケジューラは、不特定の順序で各スレッドを再開します。複数のスレッドが同じ解放されたロックを待機している場合があります。スレッドが再開されるまでに、必要なロックが使用できないことがわかり、スレッドが再び待機リストに載る場合があります。このコードは、スレッド通知の管理方法が特に効率的ではありません。notifyAll メソッドは、他のロックを待機しているスレッドを含む、すべての待機スレッドを移動します。これらのスレッドは、スケジュールされると、(当然ですが) ロックがまだ利用できないことがわかり、待機リストに戻ります。一方で、同時に実行されている競合するデータベース スレッドが比較的少ない場合、この戦略はそれほどコストがかかりません。一方、データベース エンジンはそれよりも洗練されている必要があります。演習 5.53 〜 5.54 では、待機/通知メカニズムを改善するように求められます。

5.4.9.2 クラス ConcurrencyMgr

ConcurrencyMgr クラスのコードは図 5.28 に示されています。各トランザクションには同時実行マネージャがありますが、それらはすべて同じロック テーブルを使用する必要があります。

```
public class ConcurrencyMgr {
   private static LockTable locktbl = new LockTable();
   private Map<Block,String> locks  = new HashMap<Block,String>();

   public void sLock(Block blk) {
      if (locks.get(blk) == null) {
         locktbl.sLock(blk);
         locks.put(blk, "S");
      }
   }

   public void xLock(Block blk) {
      if (!hasXLock(blk)) {
         sLock(blk);
         locktbl.xLock(blk);
         locks.put(blk, "X");
      }
   }

   public void release() {
      for (Block blk : locks.keySet())
         locktbl.unlock(blk);
      locks.clear();
   }

   private boolean hasXLock(Block blk) {
      String locktype = locks.get(blk);
      return locktype != null && locktype.equals("X");
   }
}
```

図5.28 SimpleDBクラスConcurrencyMgrのコード

この要件は、各 ConcurrencyMgr オブジェクトが静的 LockTable 変数を共有することで実装されます。トランザクションによって保持されるロックの説明は、ローカル変数 locks に保持されます。この変数は、ロックされたブロックごとにエントリを持つマップを保持します。エントリに関連付けられた値は、そのブロックに slock または xlock があるかどうかに応じて、「S」または「X」のいずれかになります。

sLock メソッドは、まずトランザクションがブロックにすでにロックをかけているかどうかを確認します。かけている場合は、ロック テーブルに移動する必要はありません。かけていない場合は、ロック テーブルの sLock メソッドを呼び出して、ロックが許可されるのを待ちます。トランザクションがブロックに xlock をかけている場合は、xLock メソッドは何もする必要はありません。かけていない場合は、まずブロックの slock を取得し、次に xlock を取得します (ロック テーブルの xLock メソッドは、トランザクションがすでに slock をかけていることを前提としていることを思い出してください)。ブロックに xlock をかけているトランザクションは、暗黙的に slock をかけているという意味で、xlock は slock よりも「強力」であることに注意してください。

5.5 SimpleDBトランザクションの実装

セクション 5.2 では、Transaction クラスの API を紹介しました。これで、その実装について説明できます。Transaction クラスは、固定したバッファを管理するために BufferList クラスを使用します。各クラスについて順に説明します。

クラス取引

Transaction クラスのコードは図 5.29 に示されています。各 Transaction オブジェクトは、独自のリカバリ マネージャと同時実行マネージャを作成します。また、現在固定されているバッファを管理するために、オブジェクト myBuffers を作成します。

コミットおよびロールバック メソッドは、次のアクティビティを実行します。

- 残っているバッファの固定を解除します。
- リカバリ マネージャーを呼び出して、トランザクションをコミット (またはロールバック
- 同時実行マネージャを呼び出してロックを解除します。

getInt メソッドと getString メソッドは、まず同時実行マネージャから指定されたブロックの slock を取得し、次にバッファから要求された値を返します。setInt メソッドと setString メソッドは、まず同時実行マネージャから xlock を取得し、次にリカバリ マネージャ内の対応するメソッドを呼び出して適切なログ レコードを作成し、その LSN を返します。この LSN は、バッファの setModi fi ed メソッドに渡すことができます。

size メソッドと append メソッドは、ファイル終了マーカーをブロック番号 -1 の「ダミー」ブロックとして扱います。size メソッドはブロックの slock を取得し、append メソッドはブロックの xlock を取得します。

クラス BufferList

BufferList クラスは、トランザクションの現在固定されているバッファのリストを管理します。図 5.30 を参照してください。BufferList オブジェクトは、指定されたブロックにどのバッファが割り当てられているか、および各ブロックが何回固定されているかという 2 つのことを認識する必要があります。コードでは、マップを使用してバッファを決定し、リストを使用して固定回数を決定します。リストには、固定されている回数と同じ数の BlockId オブジェクトが含まれます。ブロックの固定が解除されるたびに、リストからインスタンスが削除されます。

unpinAll メソッドは、トランザクションがコミットまたはロールバックするときに必要なバッファ関連のアクティビティを実行します。つまり、バッファ マネージャに、トランザクションによって変更されたすべてのバッファをフラッシュさせ、まだ固定されているバッファの固定を解除させます。

5.6 章の要約

- クライアント プログラムが無差別に実行されると、データが失われたり破損したりする可能性があります。データベース エンジンは、クライアント プログラムをトランザクションで構成するように強制します。
- トランザクションは、単一の操作として動作する一連の操作です。これは、原子性、一貫性、独立性、および永続性という ACID プロパティを満たします。
- 同時実行マネージャは、アトミック性と耐久性を保証する役割を担っています。これは、ログを読み取って処理するサーバの一部です。次の3つの機能があります。

プblic クラス Transaction { private static int nextTxNum = 0; private static final int END_OF_F ILE = -1; private RecoveryMgr recoveryMgr; private ConcurrencyMgr concurMgr; private Buf ferMgr bm; private FileMgr fm; private int txnum; private BufferList mybuffers; public Transa ction(FileMgr fm, LogMgr lm, BufferMgr bm) { this.fm = fm; this.bm = bm; txnum = nextTx Number(); recoveryMgr = new RecoveryMgr(this, txnum, lm, bm); concurMgr = new Concurr encyMgr(); mybuffers = new BufferList(bm); }public void commit() { recoveryMgr.commit(); concurMgr.release(); mybuffers.unpinAll(); System.out.println("トランザクション " + txnum + " がコミットされました"); }public void rollback() { recoveryMgr.rollback(); concurMgr.r elease(); mybuffers.unpinAll(); System.out.println("トランザクション " + txnum + " がロー ルバックされました"); }public void recovery() { bm.flushAll(txnum); recoveryMgr.recover( ); }public void pin(BlockId blk) { mybuffers.pin(blk); }public void unpin(BlockId blk) { mybu ffers.unpin(blk); }public int getInt(BlockId blk, int offset) { concurMgr.sLock(blk); バッファ ー バッファー = mybuffers.getBuffer(blk); return buff.contents().getInt(offset); }public Strin g getString(BlockId blk, int offset) { concurMgr.sLock(blk); バッファー buff = mybuffers.get Buffer(blk); return buff.contents().getString(offset);

}

図5.29 SimpleDBクラスTransactionのコード

```
public void setInt(BlockId blk, int offset, int val, boolean okToLog) { concurMgr.xLock(
blk); バッファー buff = mybuffers.getBuffer(blk); int lsn = -1; if (okToLog) lsn = recov
eryMgr.setInt(buff, offset, val); ページ p = buff.contents(); p.setInt(offset, val); buff.set
Modified(txnum, lsn); }public void setString(BlockId blk, int offset, String val, boolean o
kToLog) { concurMgr.xLock(blk); バッファー buff = mybuffers.getBuffer(blk); int lsn
= -1; if (okToLog) lsn = recoveryMgr.setString(buff, offset, val); Page p = buff.contents()
; p.setString(offset, val); buff.setModified(txnum, lsn); }public int size(String filename) {
BlockId dummyblk = new BlockId(filename, END_OF_FILE); concurMgr.sLock(dumm
yblk); return fm.length(filename); }public BlockId append(String filename) { BlockId du
mmyblk = new BlockId(filename, END_OF_FILE); concurMgr.xLock(dummyblk); retur
n fm.append(filename); }public int blockSize() { return fm.blockSize(); }public int availa
bleBuffs() { return bm.available(); }private static synchronized int nextTxNumber() { nex
tTxNum++; System.out.println("新しいトランザクション: " + nextTxNum); return nex
tTxNum; }
```

```
}
```

図5.29（続き）

```
クラス BufferList { private Map<BlockId,Buffer> buffers = new HashMap<>(); p
rivate List<BlockId> pins = new ArrayList<>(); private BufferMgr bm; public Buf
ferList(BufferMgr bm) { this.bm = bm; }Buffer getBuffer(BlockId blk) { return buff
ers.get(blk); }void pin(BlockId blk) { Buffer buff = bm.pin(blk); buffers.put(blk, buf
f); pins.add(blk); }void unpin(BlockId blk) { Buffer buff = buffers.get(blk); bm.unpi
n(buff); pins.remove(blk); if (!pins.contains(blk)) buffers.remove(blk); }void unpin
All() { for (BlockId blk : pins) { バッファバッファ = buffers.get(blk); bm.unpin(b
uff); }buffers.clear(); pins.clear(); } }
```

図5.30 SimpleDBクラスBufferListのコード

ログ レコードを書き込み、トランザクションをロールバックし、システム クラッシュ後にデータベースを回復します。

- 各トランザクションは、開始時刻を示す開始レコード、変更内容を示す更新レコード、完了時刻を示すコミットまたはロールバック レコードをログに書き込みます。さらに、リカバリ マネージャーは、さまざまなタイミングでチェックポイント レコードをログに書き込むことができます。
- リカバリ マネージャーは、ログを逆方向に読み取ってトランザクションをロールバックします。トランザクションの更新レコードを使用して変更を元に戻します。
- リカバリ マネージャーは、システム クラッシュ後にデータベースを回復します。

- 元に戻す/やり直しリカバリアルゴリズムは、コミットされていないトランザクションによって行われた変更を元に戻し、コミットされたトランザクションによって行われた変更をやり直します。
- 元に戻すだけのリカバリアルゴリズムでは、コミットされたトランザクションによって行われた変更は、トランザクションがコミットされる前にディスクにフラッシュされていると想定しています。したがって、コミットされていないトランザクションによって行われた変更のみを元に戻す必要があります。
- 再実行のみのリカバリアルゴリズムでは、トランザクションがコミットされるまで変更されたバッファはフラッシュされないものと想定しています。このアルゴリズムでは、トランザクションが完了するまで変更されたバッファを固定しておく必要がありますが、コミットされていないトランザクションを元に戻す必要はありません。
- 先行書き込みログ戦略では、更新ログ レコードを変更されたデータ ページの前にディスクに強制的に書き込む必要があります。先行書き込みログにより、データベースへの変更は常にログに記録され、常に元に戻せることが保証されます。
- チェックポイント レコードは、リカバリ アルゴリズムが考慮する必要があるログの部分を減らすためにログに追加されます。静止チェックポイント レコードは、現在トランザクションが実行されていないときに書き込むことができます。非静止チェックポイント レコードはいつでも書き込むことができます。元に戻す/やり直し (またはやり直しのみ) リカバリを使用する場合、リカバリ マネージャーは、チェックポイント レコードを書き込む前に、変更されたバッファーをディスクにフラッシュする必要があります。
- リカバリ マネージャは、値、レコード、ページ、ファイルなどをログに記録することを選択できます。ログの単位はリカバリ データ項目と呼ばれます。データ項目の選択にはトレードオフが伴います。粒度の大きいデータ項目では更新ログ レコードが少なくて済みますが、各ログ レコードのサイズは大きくなります。
- 同時実行マネージャは、同時トランザクションの正しい実行を担当するデータベース エンジンの部分です。
- エンジン内のトランザクションによって実行される一連の操作は、スケジュールと呼ばれます。スケジュールは、シリアル スケジュールと同等である場合、シリアル化可能です。シリアル化可能なスケジュールの実行が正しくなります。
- 同時実行マネージャは、スケジュールがシリアル化可能であることを保証するため、ロックを使用します。特に、すべてのトランザクションがロック プロトコルに従う必要があります。ロック プロトコルには、次の内容が記載されています。 – ブロックを読み取る前に、そのブロックの共有ロックを取得します。 – ブロックを変更する前に、そのブロックの排他ロックを取得します。 – コミットまたはロールバック後にすべてのロックを解除します。
- 各トランザクションが次のトランザクションによって保持されるロックを待機するトランザクションのサイクルが存在する場合、デッドロックが発生する可能性があります。同時実行マネージャは、待機グラフを保持しサイクルをチェックすることでデッドロックを検出できます。
- 同時実行マネージャは、デッドロック検出を近似するバックオフ アルゴリズムを使用することもできます。wait-die アルゴリズムは、古いトランザクションによって保持されているロックが必要な場合に、トランザクションを強制的にロールバックします。time-limit アルゴリズムは、予想よりも長くロックを待機している場合に、トランザクションを強制的にロールバックします。これらのアルゴリズムは両方とも、デッドロックが存在する場合はそれを削除しますが、トランザクションを不必要にロールバックする可能性もあります。

- あるトランザクションがファイルを調べている間に、別のトランザクションが新しいブロックを追加することがあります。これらのブロック内の値はファントムと呼ばれます。ファントムは直列化可能性に違反するため望ましくありません。トランザクションは、ファイル終了マーカーをロックすることでファントムを回避できます。
- シリアル化を強制するために必要なロックにより、同時実行性が大幅に低下します。マルチバージョン ロック戦略により、読み取り専用トランザクションをロックなしで (したがって待機せずに) 実行できます。同時実行マネージャは、各トランザクションにタイムスタンプを関連付け、それらのタイムスタンプを使用して、指定された時点のブロックのバージョンを再構築することで、マルチバージョン ロックを実装します
- ロックによって課せられる待機時間を減らすもう 1 つの方法は、直列化可能性の要件を削除することです。トランザクションは、直列化可能、反復可能読み取り、読み取りコミット、または読み取り非コミットの 4 つの分離レベルのいずれかに属するように指定できます。直列化不可能な各分離レベルは、ログ プロトコルによって指定された slock の制限を減らし、待機時間が短縮される一方で読み取りの問題の重大度が増します。直列化不可能な分離レベルを選択する開発者は、不正確な結果が発生する範囲と、そのような不正確さが許容されるかどうかを慎重に考慮する必要があります。
- リカバリと同様に、同時実行マネージャは値、レコード、ページ、ファイルなどをロックすることを選択できます。ロックの単位は同時実行データ項目と呼ばれます。データ項目の選択にはトレードオフが伴います。粒度の大きいデータ項目では必要なロックの数が少なくなりますが、ロックが大きいと競合が発生しやすくなり、同時実行性が低下します。

5.7 推奨される読み物

トランザクションの概念は、データベース システムだけでなく、分散コンピューティングの多くの分野で基礎となっています。研究者たちは、広範囲にわたる一連の技術とアルゴリズムを開発してきましたが、この章で紹介するアイデアは、非常に大きな氷山の一角にすぎません。この分野の概要を解説した 2 冊の優れた書籍は、Bernstein と Newcomer (1997) と Gray と Reuter (1993) です。多くの同時実行制御および回復アルゴリズムを包括的に扱った書籍は、Bernstein ら (1987) に掲載されています。広く採用されている回復アルゴリズムは ARIES と呼ばれ、Mohan ら (1992) で説明されています。

Oracle のシリアライズ可能な分離レベルの実装はスナップショット分離と呼ばれ、マルチバージョン同時実行制御を更新にまで拡張します。詳細は Ashdown ら (2019) の第 9 章に記載されています。Oracle はこの分離レベルを「シリアライズ可能」と呼んでいますが、微妙に異なります。スナップショット分離はロック プロトコルよりも効率的ですが、シリアライズ可能性を保証するものではありません。ほとんどのスケジュールはシリアライズ可能ですが、シリアライズ不可能な動作になるシナリオがいくつかあります。Fekete ら (2005) の記事では、これらのシナリオを分析し、リスクのあるアプリケーションを修正してシリアライズ可能性を保証する方法を示しています。

Oracle は、undo-redo リカバリを実装していますが、undo 情報 (つまり、古い上書きされた値) と redo 情報 (新しく書き込まれた値) を分離しています。 redo 情報は redo ログに保存され、この章の説明と同様に管理されます。ただし、undo 情報はログ ファイルではなく、特別な undo バッファに保存されます。これは、Oracle がマルチバージョン同時実行とリカバリの両方に以前の上書きされた値を使用するためです。詳細については、Ashdown ら (2019) の第 9 章を参照してください。

トランザクションは、複数のより小さな、調整されたトランザクションから構成されていると考えると、役に立つことがよくあります。たとえば、ネストされたトランザクションでは、親トランザクションは 1 つ以上の子サブトランザクションを生成できます。サブトランザクションが完了すると、その親がどうするかを決定します。サブトランザクションが中止された場合、親はすべての子を中止するか、中止されたトランザクションを置き換える別のトランザクションを生成して続行するかを選択できます。ネストされたトランザクションの基本は、Moss (1985) に記載されています。記事 Weikum (1991) では、ネストされたトランザクションに似たマルチレベル トランザクションを定義しています。違いは、マルチレベル トランザクションでは、並列実行によって効率を高める方法としてサブトランザクションを使用する点です。

Ashdown, L., et al. (2019). Oracle データベースの概念。ドキュメント E96138-01、Oracle Corporation。https://docs.oracle.com/en/database/oracle/ oracle-database/19/cncpt/database-concepts.pdf から取得 Bernstein, P., Hadzilacos, V., & Goodman, N. (1987). データベース システムにおける同時実行制御とリカバリ。Reading, MA: Addison-Wesley。Bernstein, P., & Newcomer, E. (1997). トランザクション処理の原則。San Mateo: Morgan Kaufman。Fekete, A., Liarokapis, D., O’Neil, E., O’Neil, P., & Shasha, D. (2005). スナップショット分離をシリアル化可能にする。ACM Transactions on Database Systems、30(2)、492 – 528。 Gray, J., & Reuter, A. (1993). トランザクション処理: 概念とテクニック。サンマテオ: Morgan Kaufman。Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwartz, P. (1992). ARIES: 先行書き込みログを使用した細粒度ロックと部分ロールバックをサポートするトランザクション回復方法。ACM Transactions on Database Systems, 17 (1), 94 – 162。Moss, J. (1985). ネストされたトランザクション: 信頼性の高い分散コンピューティングへのアプローチ。ケンブリッジ、MA: MIT Press。Weikum, G. (1991). マルチレベル トランザクション管理の原理と実現戦略。ACM Transactions on Database Systems, 16(1), 132 – 180。

5.8 演習

概念演習

5.1. 図 5.1 のコードが 2 人の同時ユーザーによって実行されているが、トランザクションは実行されていないと仮定します。2 つの座席が予約されているが、1 つの販売のみが記録されるシナリオを示します。

5.2. Git や Subversion などのソフトウェア構成マネージャーを使用すると、ユーザーはファイルへの一連の変更をコミットし、ファイルを以前の状態にロールバックできます。また、複数のユーザーが同時にファイルを変更することもできます。 (a) このようなシステムでのトランザクションの概念は何ですか? (b) このようなシステムはどのようにして直列化可能性を保証しますか? (c) このようなアプローチはデータベース システムで機能しますか? 説明してください。 5.3. 複数の無関係な SQL クエリを実行するが、データベースを変更しない JDBC プログラムについて考えます。プログラマーは、何も更新されないため、トランザクションの概念は重要ではないと判断し、プログラム全体を単一のトランザクションとして実行します。 (a) トランザクションの概念が読み取り専用プログラムにとって重要である理由を説明してください。 (b) プログラム全体を大きなトランザクションとして実行することの問題は何ですか? (c) 読み取り専用トランザクションのコミットには、どのくらいのオーバーヘッドが関係しますか? プログラムが SQL クエリのたびにコミットするのは理にかなっていますか? 5.4.リカバリ マネージャは、各トランザクションの開始時に、ログに開始レコードを書き込みます。 (a) ログに開始レコードを含めることの実際的な利点は何ですか? (b) データベース システムが、ログに開始レコードを書き込まないことにしたとします。リカバリ マネージャは、それでも正常に機能しますか? どのような機能が影響を受けますか? 5.5. SimpleDB のロールバック メソッドは、ロールバック ログ レコードをディスクに書き込んでから戻ります。これは必要ですか? これは良い考えですか? 5.6. リカバリ マネージャが変更され、完了時にロールバック ログ レコードを書き込まなくなったとします。問題は発生しますか? これは良い考えですか? 5.7. 図 5.7 の元に戻すのみのコミット アルゴリズムを検討します。アルゴリズムの手順 1 と 2 を入れ替えるのはなぜ間違っているのか説明してください。 5.8. ロールバックまたはリカバリ中にシステムがクラッシュした場合でも、ロールバック (またはリカバリ) をやり直すことが正しいことを示します。 5.9.ロールバックまたはリカバリ中にデータベースに加えられた変更をログに記録する理由はありますか? 説明してください。

5.10. 非静止チェックポイント アルゴリズムのバリエーションとして、チェックポイント ログ レコード内の 1 つのトランザクション、つまりその時点で最も古いアクティブなトランザクションのみに言及する方法があります。(a) 回復アルゴリズムがどのように機能するかを説明してください。(b) この戦略をテキストで示されている戦略と比較してください。どちらが実装が簡単ですか? どちらがより効率的ですか?

5.11. ロールバック メソッドは、静止チェックポイント ログ レコードに遭遇した場合、どうすればよいですか? 非静止ログ レコードに遭遇した場合はどうなりますか? 説明してください。

5.12. 非静止チェックポイントのアルゴリズムでは、チェックポイント レコードの書き込み中に新しいトランザクションを開始することはできません。この制限が正確性にとって重要である理由を説明してください。 5.13. 非静止チェックポイントを行う別の方法は、ログに 2 つのレコードを書き込むことです。最初のレコードは <BEGIN_NQCKPT> で、他には何も含まれていません。2 番目のレコードは標準の <NQCKPT ...> レコードで、アクティブなトランザクションのリストが含まれています。最初のレコードは、リカバリ マネージャがチェックポイントを実行すると決定するとすぐに書き込まれます。2 番目のレコードは、アクティブなトランザクションのリストが作成された後で書き込まれます。 (a) この戦略が演習 5.12 の問題を解決する理由を説明してください。 (b) この戦略を組み込んだ改訂版のリカバリ アルゴリズムを示してください。 5.14. リカバリ マネージャがリカバリ中に複数の静止チェックポイント レコードに遭遇することがない理由を説明してください。 5.15.リカバリ マネージャーがリカバリ中に複数の非静止チェックポイント レコードに遭遇する可能性があることを示す例を示します。最初のチェックポイント レコードの後に見つかった非静止チェックポイント レコードを処理するための最適な方法は何ですか? 5.16. リカバリ マネージャーがリカバリ中に非静止チェックポイント レコードと静止チェックポイント レコードの両方に遭遇できない理由を説明します。

5.17. 図5.6の回復アルゴリズムを検討してください。ステップ1cでは、ロールバックされたトランザクションの値は元に戻されません。

> (a) なぜこれが正しいことなのか説明してください。 (b) これらの値を元に戻した場合、アルゴリズムは正しいでしょうか? 説明してください。

5.18. ロールバック メソッドが値の元の内容を復元する必要がある場合、ページに直接書き込み、いかなる種類のロックも要求しません。これにより、別のトランザクションとの非シリアル化競合が発生する可能性がありますか?説明してください。 5.19. 元に戻すだけのリカバリとやり直しだけのリカバリの手法を組み合わせたリカバリ アルゴリズムを使用できない理由を説明してください。つまり、元に戻す情報またはやり直し情報のいずれかを保持する必要がある理由を説明してください。 5.20. システムがクラッシュ後に再起動したときに、リカバリ マネージャーがログ ファイルで次のレコードを見つけたとします。

<START、1>START、2>SETSTRING、2、ジャンク、33、<0、abc、def> <SETSTRING、1、ジャンク、44、0、abc、xyz> <START、3>COMMIT、2>SETSTRING、3、ジャンク、33、< 0、def、joe> <START、4>SETSTRING、4、ジャンク、55、< 0、abc、sue> <NQCKPT、1、3、4>SETSTRING、4、ジャンク、55、0、sue、max> <START、5>COMMIT、4>

<

<

(a) 元に戻す/やり直しリカバリを想定して、データベースにどのような変更が行われるかを示します。 (b) 元に戻すのみのリカバリを想定して、データベースにどのような変更が行われるかを示します。 (c) ログにコミット レコードがない場合でも、トランザクション T1 がコミットされる可能性はありますか? (d) トランザクション T1 がブロック 23 を含むバッファを変更する可能性はありますか? (e) トランザクション T1 がディスク上のブロック 23 を変更する可能性はありますか? (f) トランザクション T1 がブロック 44 を含むバッファを変更しない可能性はありますか?

5.21. シリアルスケジュールは常にシリアル化可能でしょうか? シリアル化可能なスケジュールは常にシリアルでしょうか? 説明してください。

5.22. この演習では、非連続的な学校の必要性を検討するよう求めています。

(a) データベースがバッファ プールのサイズよりもはるかに大きいと仮定します。トランザクションを同時に実行できる場合、データベース システムがトランザクションをより速く処理できる理由を説明します。 (b) 逆に、データベースがバッファ プールに収まる場合、同時実行がそれほど重要ではなくなる理由を説明します。

5.23. SimpleDB クラスの Transaction の get/set メソッドは、指定されたブロックのロックを取得します。完了したときにブロックのロックを解除しないのはなぜですか?

5.24. 図 5.3 を検討してください。ファイルが並行性の要素である場合のトランザクションの履歴を示します。

5.25. 次の 2 つのトランザクションとその履歴について考えてみましょう。

T1: W(b1); R(b2); W(b1); R(b3); W(b3); R(b4); W(b2) T2: R(b2); R(b3); R(b1); W(b3); R(b4); W(b4)

(a) これらのトランザクションの直列化可能な非直列スケジュールを指定します。 (b) ロック プロトコルを満たすこれらの履歴にロックおよびロック解除アクションを追加します。 (c) デッドロックが発生するこれらのロックに対応する非直列スケジュールを指定します。 (d) ロック プロトコルに従うこれらのトランザクションには、デッドロックが発生しない非直列の直列化可能なスケジュールが存在しないことを示します。

5.26. シリアル化可能だが、トランザクションのコミット順序に影響を与えない競合する書き込み-書き込み操作があるスケジュールの例を示します。(ヒント: 競合する操作の中には、対応する読み取り操作がないものもあります。)

5.27. すべてのトランザクションが 2 フェーズ ロック プロトコルに従う場合、すべてのスケジュールがシリアル化可能であることを示します。

5.28. デッドロックがある場合に限り、待機グラフにサイクルが存在することを示します。5.29. トランザクションマネージャがデッドロックを正確に検出するために待機グラフを維持していると仮定します。セクション5.4.4では、トランザクションが

マネージャーは、グラフ内のサイクルの原因となったリクエストのトランザクションをロールバックします。他の可能性としては、サイクル内の最も古いトランザクション、サイクル内の最も新しいトランザクション、最も多くのロックを保持しているトランザクション、または最も少ないロックを保持しているトランザクションをロールバックすることが考えられます。どの可能性が最も理にかなっていると思いますか、説明してください。

5.30. SimpleDB でトランザクション T が現在ブロックに共有ロックを持っており、そのブロックに対して setInt を呼び出すとします。これがデッドロックを引き起こすシナリオを示してください。

5.31. 図 5.19 の ConcurrencyTest クラスを考えます。デッドロックを引き起こすスケジュールを指定します。

5.32. 図 5.19 で説明したロック シナリオを検討します。ロックが要求され、解放されるときの待機グラフのさまざまな状態を描画します。

5.33. wait-die プロトコルのバリエーションは、wound-wait と呼ばれ、次のようになります。

- T1 の数値が T2 より低い場合、T2 は中止されます (つまり、T1 が T2 に「ダメージ」を与えます)。
- T1 の番号が T2 より大きい場合、T1 はロックを待機します。古いトランザクションが新しいトランザクションによって保持されているロックを必要とする場合、新しいトランザクションを強制終了してロックを取得するという考え方です。(a) このプロトコルがデッドロックを防ぐことを示します。(b) wait-die プロトコルと wond-wait プロトコルの相対的な利点を比較します。

5.34. wait-die デッドロック検出プロトコルでは、古いトランザクションが保持しているロックを要求した場合、トランザクションは中止されます。プロトコルを変更して、新しいトランザクションが保持しているロックを要求した場合にトランザクションが中止されるようにするとします。このプロトコルはデッドロックも検出します。この修正されたプロトコルは元のプロトコルと比べてどうですか? トランザクション マネージャーにはどちらを使用することをお勧めしますか? 説明してください。

5.35. LockTable クラスの lock/unlock メソッドが同期されている理由を説明してください。同期されていない場合、どのような問題が発生する可能性がありますか? 5.36. データベース システムが並行性要素としてファイルを使用するとします。ファントムが不可能な理由を説明してください。 5.37. バッファを待機しているトランザクションも処理するデッドロック検出アルゴリズムを提供してください。 5.38. 並行性マネージャがログ ファイルを 1 回だけ通過するように、マルチバージョン ロックのアルゴリズムを書き直してください。 5.39. 読み取りコミット トランザクション分離レベルは、slock を早期に解放することでトランザクションの待機時間を短縮することを目的としています。一見したところ、トランザクションが既に保持しているロックを解放することで待機時間が短縮される理由は明らかではありません。ロックを早期に解放することの利点を説明し、シナリオの例を示してください。 5.40. メソッド nextTransactionNumber は、Transaction で同期されている唯一のメソッドです。他のメソッドでは同期が不要である理由を説明してください。 5.41. SimpleDB クラスの Transaction について考えてみましょう。

(a) トランザクションはブロックをロックせずに固定できますか? (b) トランザクションはブロックを固定せずにロックできますか?

プログラミング演習 5.42. SimpleDB トランザクションは、getInt メソッドまたは getString メソッドが呼び出されるたびに、ブロックの slock を取得します。別の可能性として、ブロックの内容を確認するつもりがない限りブロックを固定しないという前提で、ブロックが固定されたときにトランザクションが slock を取得するという方法があります。(a) この戦略を実装してください。(b) この戦略の利点を SimpleDB の利点と比較してください。どちらが適切だと思いますか。その理由も教えてください。 5.43. リカバリ後、ログはアーカイブ目的以外では必要ありません。リカバリ後にログ ファイルが別のディレクトリに保存され、新しい空のログ ファイルが開始されるように、SimpleDB コードを修正してください。 5.44. 必要な場合にのみ更新レコードを元に戻すように、SimpleDB リカバリ マネージャを修正してください。 5.45. ブロックをリカバリの要素として使用するように、SimpleDB を修正してください。考えられる戦略としては、トランザクションが最初にブロックを変更したときに、そのブロックのコピーを保存するというものがあります。コピーは別のファイルに保存でき、更新ログ レコードにはコピーのブロック番号を保持できます。また、ファイル間でブロックをコピーできるメソッドも記述する必要があります。 5.46. Transaction クラスに、静止チェックポイントを実行する静的メソッドを実装します。メソッドの呼び出し方法 (N トランザクションごと、N 秒ごと、または手動など) を決定します。Transaction を次のように修正する必要があります。

- 静的変数を使用して、現在アクティブなすべてのトランザクションを保持します。
- Transaction のコンストラクタを修正して、チェックポイントが実行されているかどうかを確認し、実行されている場合は、チェックポイント手順が完了するまで待機リストに自身を配置します。

5.47. 本文で説明されている戦略を使用して、非静止チェックポイントを実装します。 5.48. トランザクションがファイルに多数のブロックを追加し、これらのブロックに一連の値を書き込んでからロールバックするとします。新しいブロックは初期状態に復元されますが、それら自体はファイルから削除されません。削除されるように SimpleDB を変更します。(ヒント: 一度に 1 つのトランザクションだけがファイルに追加できるという事実を利用できます。つまり、ロールバック中にファイルを切り捨てることができます。ファイル マネージャにファイルを切り捨てる機能を追加する必要があります。) 5.49. ログ レコードは、リカバリだけでなくシステムの監査にも使用できます。監査のために、レコードにはアクティビティが発生した日付とクライアントの IP アドレスを保存する必要があります。(a) SimpleDB ログ レコードを次のように修正します。

(b) ブロックが最後に変更された日時や、特定のトランザクションまたは特定の IP アドレスから発生したアクティビティを見つけるなどの一般的な監査タスクをサポートするメソッドを持つクラスを設計および実装します。

5.50. サーバーが起動するたびに、トランザクション番号は 0 から再び始まります。つまり、データベースの履歴全体にわたって、同じ番号を持つトランザクションが複数存在することになります。

(a) トランザクション番号のこの一意性がないことが重大な問題ではない理由を説明してください。 (b) トランザクション番号がサーバーが最後に実行されていたときから継続されるように SimpleDB を修正してください。

5.51. SimpleDB を修正して、undo-redo リカバリを使用するようにします。

5.52. 以下を使用して SimpleDB でデッドロック検出を実装します。

(a) テキストに記載されている待機-死亡プロトコル (b) 演習5.33に記載されている負傷-待機プロトコル

5.53. ロック テーブルを修正して、各ブロックに個別の待機リストを使用するようにします。(そのため、notifyAll は同じロックで待機しているスレッドにのみ影響します。)

5.54. ロック テーブルを修正して、独自の明示的な待機リストを保持し、ロックが使用可能になったときに通知するトランザクションを自ら選択するようにします。(つまり、notifyAll ではなく Java メソッドの notification を使用します。)

5.55. SimpleDB 同時実行マネージャを次のように修正します。

(a) ファイルは並行性の要素です。 (b) 値は並行性の要素です。 (警告: size メソッドと append メソッドが競合を引き起こさないようにする必要があります。)

5.56. テストプログラムを書く:

(a) リカバリマネージャが機能することを確認する (コミット、ロールバック、リカバリ) (b) ロックマネージャをより完全にテストする (c) トランザクションマネージャ全体をテストする

第6章 記録管理

トランザクション マネージャは、ディスク ブロックの指定された場所で値を読み書きできます。ただし、ブロック内の値やそれらの値がどこにあるのかはわかりません。この責任はレコード マネージャにあります。レコード マネージャは、ファイルをレコードのコレクションに整理し、レコードを反復処理して値を配置するメソッドを備えています。この章では、レコード マネージャが提供する機能と、その機能を実装するために使用される手法について説明します。

6.1 レコードマネージャの設計

レコード管理者は、次のようないくつかの問題に対処する必要があります。

- 各レコードは 1 つのブロック内に完全に配置する必要がありますか?
- ブロック内のすべてのレコードは同じテーブルからのものですか?
- 各フィールドは、事前に決められたバイト数を使用して表現できますか?
- 各フィールド値はレコード内のどこに配置すればよいですか?

このセクションでは、これらの問題とそのトレードオフについて説明します。

6.1.1 スパンドレコードとアンスパンドレコード

レコード マネージャが、ブロック サイズが 1000 バイトのファイルに 300 バイトのレコードを 4 つ挿入する必要があるとします。3 つのレコードは、ブロックの最初の 900 バイトにうまく収まります。しかし、レコード マネージャは 4 番目のレコードをどう処理すればよいでしょうか。図 6.1 に 2 つのオプションを示します。

図6.1aでは、レコードマネージャは、値が2つ以上のブロックにまたがるレコードを作成します。レコードの最初の100バイトを

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_6

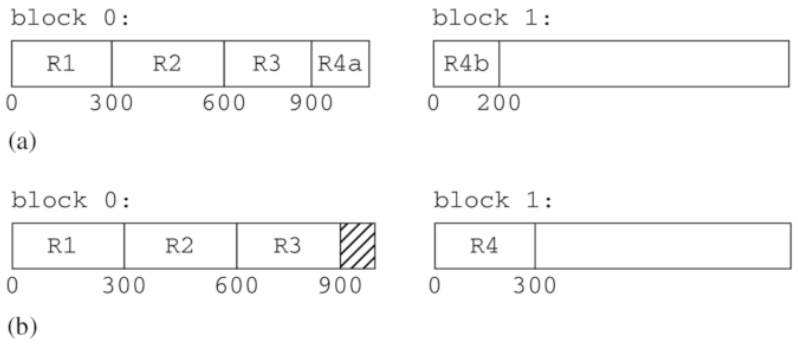


図6.1 スパンドレコードとアンスパンドレコード。(a) レコードR4はブロック0と1にまたがり、(b) レコードR4はブロック1に完全に格納される。

既存のブロックに格納され、レコードの次の 200 バイトが新しいブロックに格納されます。図 6.1b では、レコード マネージャは 4 番目のレコード全体を新しいブロックに格納します。

ほとんどのシステムでは、スパンドレコードを作成するかどうかを決定する必要があります。スパンされていないレコードの欠点は、ディスク領域を無駄にすることです。図 6.1b では、各ブロックの 100 バイト (つまり 10 %) が無駄になっています。さらに悪いケースは、各レコードが 501 バイトの場合です。この場合、ブロックには 1 つのレコードしか格納できず、その領域の約 50% が無駄になります。もう 1 つの欠点は、スパンされていないレコードのサイズがブロック サイズに制限されることです。レコードがブロックよりも大きくなる可能性がある場合は、スパンが必要です。

スパンド レコードの主な欠点は、レコード アクセスの複雑さが増すことです。スパンド レコードは複数のブロックに分割されるため、読み取りには複数のブロック アクセスが必要になります。さらに、スパンド レコードを別のメモリ領域に読み取って、これらのブロックから再構築する必要がある場合もあります。

6.1.2 同種ファイルと非同種ファイル

すべてのレコードが同じテーブルから取得される場合、ファイルは同種です。レコード マネージャは、非同種ファイルを許可するかどうかを決定する必要があります。ここでも、効率性と柔軟性のトレードオフが発生します。

たとえば、図 1.1 の STUDENT テーブルと DEPT テーブルについて考えてみましょう。同種の実装では、すべての STUDENT レコードが 1 つのファイルに、すべての DEPT レコードが別のファイルに格納されます。この配置により、単一テーブルの SQL クエリへの応答が容易になります。レコード マネージャは、1 つのファイルのブロックをスキャンするだけで済みます。ただし、複数テーブルのクエリは効率が悪くなります。これらの 2 つのテーブルを結合するクエリ (「学生の名前と専攻学部を検索する」など) について考えてみましょう。レコード マネージャは、一致するレコードを探すために、STUDENT レコードのブロックと DEPT レコードのブロックの間を行ったり来たりして検索する必要があります (第 8 章で説明します)。クエリが余分な検索なしで実行できたとしても (第 12 章のインデックス結合などによって)、ディスク ドライブは、STUDENT ブロックと DEPT ブロックの読み取りを交互に行うため、繰り返しシークする必要があります。

図6.2 クラスター化された非均質なレコード

非同種組織では、STUDENT レコードと DEPT レコードが同じファイルに保存され、各学生のレコードはその専攻学部のレコードの近くに格納されます。図 6.2 は、ブロックあたり 3 つのレコードがあると仮定した、このような組織の最初の 2 つのブロックを示しています。ファイルは DEPT レコードと、その学部を専攻とする STUDENT レコードで構成されます。この組織では、結合されたレコードが同じ (または近くの) ブロックにクラスター化されるため、結合を計算するために必要なブロック アクセスが少なくなります。

クラスタリングにより、一致するレコードが一緒に保存されるため、クラスタ化されたテーブルを結合するクエリの効率が向上します。ただし、クラスタリングにより、各テーブルのレコードがより多くのブロックに分散されるため、単一テーブル クエリの効率は低下します。同様に、他のテーブルとの結合も効率が低下します。したがって、クラスタリングは、最も頻繁に使用されるクエリがクラスタリングによってエンコードされた結合を実行する場合にのみ有効です。[1]

6.1.3 固定長フィールドと可変長フィールド

テーブル内のすべてのフィールドには、定義されたタイプがあります。レコード マネージャーは、そのタイプに基づいて、固定長表現または可変長表現のどちらを使用してフィールドを実装するかを決定します。固定長表現では、各フィールドの値を格納するためにまったく同じバイト数が使用されますが、可変長表現では、格納されるデータ値に基づいて拡張および縮小されます。

ほとんどの型は、本来固定長です。たとえば、整数と浮動小数点数は、どちらも 4 バイトのバイナリ値として格納できます。実際、すべての数値型と日付/時刻型は、本来固定長の表現を持っています。Java の String 型は、文字列が任意の長さになる可能性があるため、可変長の表現を必要とする型の代表的な例です。

可変長表現は、重大な問題を引き起こす可能性があります。たとえば、レコードが詰め込まれたブロックの中央にあるレコードを考えてみましょう。そのフィールド値の 1 つを変更するとします。フィールドが固定長の場合、レコードのサイズは変わりません。フィールドは、その場で変更できます。しかし、フィールドが可変長の場合、レコードは大きくなる可能性があります。レコードを大きくするために、レコード マネージャは、レコードの位置を再配置する必要がある場合があります。

[1] In fact, clustering is the fundamental organizational principle behind the early hierarchical database systems, such as IBM's *IMS* system. Databases that are naturally understood hierarchically can be implemented very efficiently in such systems.

ブロック内のレコードがすべて削除されます。実際、変更されたレコードが大きくなりすぎると、1つ以上のレコードをブロックから移動して別のブロックに配置する必要がある場合があります。

その結果、レコードマネージャは可能な限り固定長表現を使用するように努めます。たとえば、レコード マネージャは文字列フィールドの3つの異なる表現から選択できます。

- 可変長表現。レコードマネージャが文字列に必要なレコード内の正確なスペース量を割り当てる。
- 固定長表現。レコードマネージャは文字列をレコード外の場所に保管し、レコード内のその場所への固定長参照を保持します。
- 固定長表現。レコードマネージャは、長さに関係なく、各文字列に対してレコード内に同じ量のスペースを割り当てます。

これらの表現は図6.3に示されています。パート(a)は3つのCOURSEレコードを示しており、タイトルフィールドは可変長表現を使用して実装されています。これらのレコードはスペース効率に優れていますが、先ほど説明した問題があります。

パート(b)は同じ3つのレコードを示していますが、タイトル文字列は別の「文字列領域」に配置されています。この領域は別のファイルの場合もあれば、(文字列が非常に大きい場合は) 各文字列が独自のファイルに格納されるディレクトリの場合もあります。いずれの場合も、フィールドにはその領域内の文字列の場所への参照が含まれます。この表現により、レコードは固定長で小さくなります。レコードが小さいのは、より少ないブロックに格納できるため、ブロック アクセスが少なくて済むからです。この表現の欠点は、レコードから文字列値を取得するために追加のブロックアクセスが必要になることです。

パート(c)は、固定長のタイトルフィールドを使用して実装された2つのレコードを示しています。この実装の利点は、レコードが固定長であり、文字列がレコードに格納されることです。ただし、欠点は、一部のレコードが必要以上に大きくなることです。文字列のサイズに大きな差がある場合、

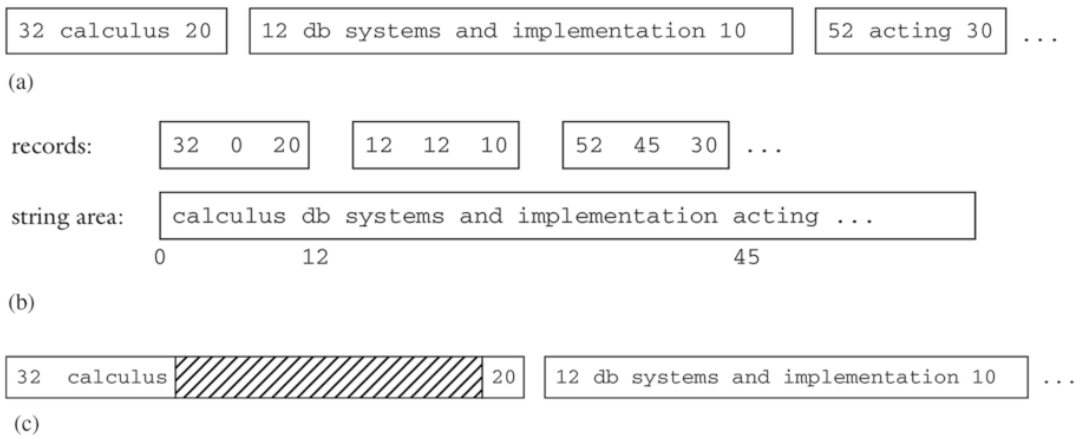


図6.3 COURSEレコードのタイトルフィールドの代替表現。(a) 各文字列に必要なだけのスペースを割り当てる、(b) 文字列を別の場所に格納する、(c) 各文字列に同じ量のスペースを割り当てる

無駄なスペースが大幅に増加し、結果としてファイルが大きくなり、それに応じてブロックアクセスも増加します。

これらの表現は、どれも明らかに他より優れているわけではありません。レコード マネージャが適切な表現を選択できるように、標準 SQL では char、varchar、clob という 3 つの異なる文字列データ型が用意されています。char(n) 型は、正確に n 文字の文字列を指定します。varchar(n) 型と clob(n) 型は、最大で n 文字の文字列を指定します。これらの違いは、n の予想されるサイズです。varchar(n) では、n は適度に小さく、4K 以下です。一方、clob(n) の n の値はギガ文字の範囲になることがあります (頭字語 CLOB は「character large object」の略です)。clob フィールドの例として、大学のデータベースが SECTION テーブルに Syllabus フィールドを追加し、このフィールドの値に各セクションのシラバスのテキストを格納するとします。シラバスが 8000 文字以下であると仮定すると、フィールドを clob(8000) として定義するのが妥当です。

char 型のフィールドは、図 6.3c に最も自然に相当します。すべての文字列は同じ長さになるため、レコード内に無駄なスペースはなく、固定長表現が最も効率的になります。

varchar(n) 型のフィールドは、図 6.3a に最も自然に相当します。n は比較的小さいため、レコード内に文字列を配置してもレコードが大きくなりすぎることはありません。さらに、文字列のサイズが変化するため、固定長表現ではスペースが無駄になります。したがって、可変長表現が最適な代替手段です。

n が小さい場合 (たとえば 20 未満)、レコード マネージャは 3 番目の表現を使用して varchar フィールドを実装することを選択する場合があります。その理由は、固定長表現の利点と比較すると、無駄になるスペースがわずかだからです。

clob 型のフィールドは図 6.3b に対応します。この表現は大きな文字列を最もうまく処理できるためです。大きな文字列をレコードの外部に格納すると、レコード自体が小さくなり、管理しやすくなります。

6.1.4 レコード内のフィールドの配置

レコード マネージャは、レコードの構造を決定します。固定長レコードの場合、レコード マネージャはレコード内の各フィールドの位置を決定します。最も単純な戦略は、フィールドを互いに隣接して保存することです。この場合、レコードのサイズはフィールドのサイズの合計になり、各フィールドのオフセットは前のフィールドの末尾になります。

フィールドをレコードに密に詰め込むこの戦略は、Java ベースのシステム (SimpleDB や Derby など) には適していますが、他のシステムでは問題を引き起こす可能性があります。問題は、値がメモリ内で適切に整列されていることを確認することです。ほとんどのコンピュータでは、整数にアクセスするためのマシン コードでは、整数が 4 の倍数のメモリ位置に格納されている必要があります。つまり、整数は 4 バイト境界に整列されていると言えます。したがって、レコード マネージャは、レコード内のすべての整数が 4 バイト境界に整列されていることを確認する必要があります。

各ページは 4 バイト境界に揃えられています。OS ページは、ある程度大きな N に対して常に 2^N バイト境界に揃えられているため、各ページの最初のバイトは適切に揃えられます。したがって、レコード マネージャは、各ページ内の各整数のオフセットが 4 の倍数であることを確認するだけで済みます。前のフィールドが 4 の倍数でない場所で終了した場合、レコード マネージャは、4 の倍数になるように十分なバイト数を埋め込む必要があります。

たとえば、3 つの整数フィールドと varchar(10) 文字列フィールドで構成される STUDENT テーブルを考えてみましょう。整数フィールドは 4 の倍数なので、パディングする必要はありません。ただし、文字列フィールドには 14 バイトが必要です (セクション 3.5.2 の SimpleDB 表現を想定)。したがって、後続のフィールドが 4 の倍数に揃えられるように、さらに 2 バイト パディングする必要があります。

一般に、異なる型には異なる量のパディングが必要です。たとえば、倍精度浮動小数点数は通常 8 バイト境界に揃えられ、小さい整数は通常 2 バイト境界に揃えられます。レコード マネージャは、これらの配置を保証する責任があります。単純な戦略は、フィールドを宣言された順序に配置し、各フィールドにパディングを入れて次のフィールドが適切に配置されるようにすることです。より賢い戦略は、必要なパディングの量が最小限になるようにフィールドを並べ替えることです。たとえば、次の SQL テーブル宣言を考えてみましょう。

テーブル T を作成します (A smallint、B double precision、C smallint、D int、E int)

フィールドが指定された順序で格納されていると仮定します。この場合、フィールド A には 6 バイト、フィールド C には 2 バイトの余分なバイトを埋め込む必要があり、レコード長は 28 バイトになります (図 6.4a を参照)。一方、フィールドが [B、D、A、C、E] の順序で格納されている場合は、パディングは必要なく、レコード長は 20 バイトのみになります (図 6.4b を参照)。

フィールドのパディングに加えて、レコード マネージャは各レコードのパディングも行う必要があります。各レコードは k バイト境界で終了する必要があり、k はサポートされている最大のアラインメントであるため、ページ内のすべてのレコードのアラインメントは最初のレコードと同じになります。レコード長が 28 バイトである図 6.4a のフィールド配置をもう一度考えてみましょう。最初のレコードがブロックのバイト 0 から始まるとします。次に、2 番目のレコードがブロックのバイト 28 から始まるとします。つまり、2 番目のレコードのフィールド B はブロックのバイト 36 から始まることになり、これは間違ったアラインメントです。各レコードが 8 バイト境界で始まることが重要です。図 6.4 の例では、部分 (a) と部分 (b) の両方のレコードに 4 バイトを追加してパディングする必要があります。

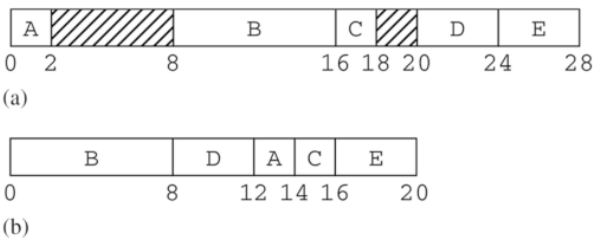


図6.4 レコード内のフィールドの配置による位置合わせ。(a) パディングが必要な配置、(b) パディングが不要な配置

Java プログラムでは、バイト配列内の数値に直接アクセスできないため、パディングを考慮する必要はありません。たとえば、ページから整数を読み取る Java メソッドは ByteBuffer.getInt です。このメソッドは、整数を取得するためにマシン コード命令を呼び出すのではなく、配列の指定された 4 バイトから整数自体を構築します。このアクティビティは、単一のマシン コード命令よりも効率は劣りますが、アライメントの問題を回避できます。

6.2 レコードファイルの実装

前のセクションでは、レコード マネージャが対処しなければならないさまざまな決定について検討しました。このセクションでは、これらの決定がどのように実装されるかを検討します。まず、最も単純な実装、つまり、均質でスパンされていない固定長のレコードを含むファイルから始めます。次に、他の設計上の決定がこの実装にどのように影響するかを検討します。

6.2.1 簡単な実装

均一で、スパンされていない、固定長のレコードのファイルを作成するとします。レコードがスパンされていないということは、ファイルをブロックのシーケンスとして扱うことができ、各ブロックに独自のレコードが含まれていることを意味します。レコードが均一で固定長であるということは、ブロック内の各レコードに同じ量のスペースを割り当てることができることを意味します。言い換えると、各ブロックをレコードの配列と考えることができます。SimpleDB では、このようなブロックをレコード ページと呼びます。

レコード マネージャーは、レコード ページを次のように実装できます。ブロックをスロットに分割します。各スロットは、レコードと 1 バイトを保持できる大きさです。このバイトの値は、スロットが空か使用中かを示すフラグです。たとえば、0 は「空」、1 は「使用中」を意味します。[2]

たとえば、ブロック サイズが 400 でレコード サイズが 26 の場合、各スロットの長さは 27 バイトで、ブロックには 14 個のスロットがあり、22 バイトの無駄なスペースがあります。図 6.5 はこの状況を示しています。この図は 14 個のスロットのうち 4 個を示しています。スロット 0 と 13 には現在レコードが含まれていますが、スロット 1 と 2 は空です。

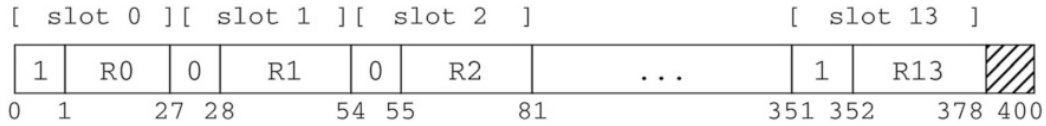


図6.5 14個の26バイトレコードを収容できるレコードページ

[2] You can improve the space usage by only using a bit to hold each empty/inuse flag. See Exercise 6.7.

レコード マネージャーは、レコード ページでレコードを挿入、削除、および変更できる必要があります。そのためには、レコードに関する次の情報を使用します。

- スロットのサイズ
- レコードの各フィールドの名前、タイプ、長さ、オフセット

これらの値はレコードのレイアウトを構成します。例として、図 2.4 で定義されているテーブル STUDENT を考えてみましょう。STUDENT レコードには、3 つの整数と 10 文字の varchar フィールドが含まれています。SimpleDB のストレージ戦略を想定すると、各整数には 4 バイト、10 文字の文字列には 14 バイトが必要です。また、パディングは不要で、varchar フィールドは最大文字列に固定スペースを割り当てることによって実装され、空/未使用フラグは各スロットの先頭で 1 バイトを占めると仮定します。図 6.6 は、このテーブルの結果のレイアウトを示しています。

レイアウトが与えられれば、レコード マネージャはページ内の各値の位置を決定できます。スロット k のレコードは、位置 RL!k から始まります。ここで、RL はレコード長です。そのレコードの空/未使用フラグは位置 RL!k にあり、そのフィールド F の値は位置 RL!k+Offset(F) にあります。レコード マネージャーは、挿入、削除、変更、および取得を非常に簡単に処理できます。

- 新しいレコードを挿入するには、レコード マネージャーは各スロットの空/未使用フラグを調べ、0 を見つけます。次に、フラグを 1 に設定し、そのスロットの場所を返します。すべてのフラグ値が 1 の場合、ブロックはいっぱいで、挿入はできません。
- レコードを削除するには、レコード マネージャーは単にその空/未使用フラグを 0 に設定
- レコードのフィールド値を変更するには (または新しいレコードのフィールドを初期化するには)、レコード マネージャーはそのフィールドの場所を決定し、その場所に値を書き込みます。
- ページ内のレコードを取得するために、レコード マネージャは各スロットの空/未使用フラグを調べます。1 が見つかるたびに、そのスロットに既存のレコードが含まれていることがわかります。

レコード マネージャには、レコード ページ内のレコードを識別する方法も必要です。レコードが固定長の場合、最も簡単なレコード識別子はスロット番号です。

スロットサイズ:
27 フィールド情報:

Name	Type	Length	Offset
SId	int	4	1
SName	varchar(10)	14	5
GradYear	int	4	19
MajorId	int	4	23

図6.6 STUDENTのレイアウト

6.2.2 可変長フィールドの実装

固定長フィールドの実装は非常に簡単です。このセクションでは、可変長フィールドの導入がその実装にどのような影響を与えるかを検討します。

1つの問題は、レコード内のフィールド オフセットが固定ではなくなったことです。特に、可変長フィールドに続くすべてのフィールドのオフセットは、レコードごとに異なります。これらのフィールドのオフセットを判別する唯一の方法は、前のフィールドを読み取って、その終了位置を確認することです。レコード内の最初のフィールドが可変長の場合、n 番目のフィールドのオフセットを判別するには、レコードの最初の n-1 フィールドを読み取る必要があります。したがって、レコード マネージャは通常、固定長フィールドを各レコードの先頭に配置して、事前に計算されたオフセットでアクセスできるようにします。可変長フィールドはレコードの末尾に配置されます。最初の可変長フィールドには固定オフセットがありますが、残りのフィールドには固定オフセットがありません。

もう1つの問題は、フィールド値を変更するとレコードの長さが変わる可能性があることです。新しい値の方が大きい場合は、変更された値の右側のブロックの内容をシフトしてスペースを確保する必要があります。極端な場合、シフトされたレコードがブロックから溢れてしまいます。この状況は、オーバーフロー ブロックを割り当てることによって処理する必要があります。

オーバーフロー ブロックは、オーバーフロー領域と呼ばれる領域から割り当てられた新しいブロックです。元のブロックからはみ出したレコードは、元のブロックから削除され、オーバーフロー ブロックに追加されます。このような変更が多数発生する場合は、複数のオーバーフロー ブロックのチェーンが必要になることがあります。各ブロックには、チェーン上の次のオーバーフロー ブロックへの参照が含まれます。概念的には、元のブロックとオーバーフロー ブロックは1つの(大きな)レコード ページを形成します。

ここでは、COURSE テーブルで、コース タイトルが可変長文字列として保存されているとします。図 6.7a は、テーブルの最初の 3 つのレコードを含むブロックを示しています。(他のフィールドは固定長であるため、タイトル フィールドはレコードの末尾に移動されています。) 図 6.7b は、タイトル「DbSys」を「Database Systems Implementation」に変更した結果を示しています。ブロック サイズが 80 バイトであると仮定すると、3 番目のレコードはブロックに収まらなくなり、オーバーフロー ブロックに配置されます。元のブロックには、そのオーバーフロー ブロックへの参照が含まれています。

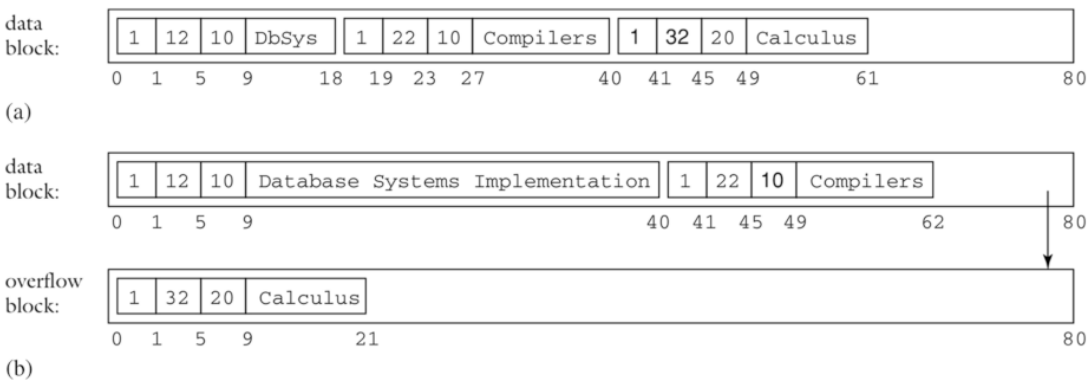


図6.7 オーバーフローブロックを使用して可変長レコードを実装する。(a) 元のブロック、(b) コース12のタイトルを変更した結果

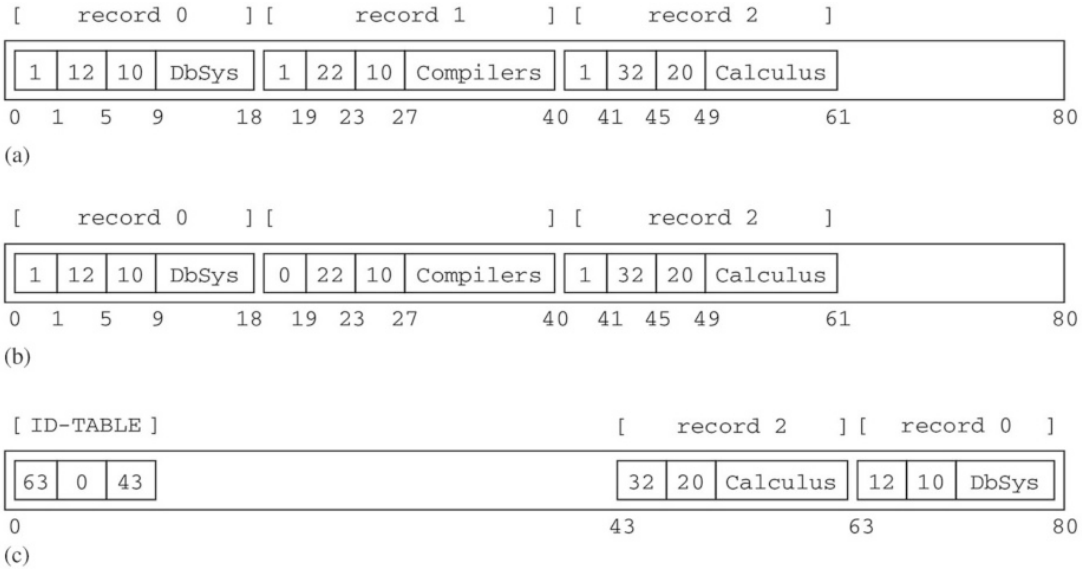


図 6.8 ID テーブルを使用して可変長レコードを実装する。(a) 元のブロック、(b) レコード 1 を削除する直接的な方法、(c) ID テーブルを使用してレコード 1 を削除する

3 番目の問題は、スロット番号をレコード識別子として使用することに関するものです。固定長レコードの場合のように、スロット番号にスロット サイズを掛けることはできなくなりました。特定の ID を持つレコードの先頭を見つける唯一の方法は、ブロックの先頭からレコードを読み取ることです。スロット番号をレコード識別子として使用すると、レコードの挿入も複雑になります。図 6.8 にこの問題を示します。

パート (a) は、図 6.7a と同様に、最初の 3 つの COURSE レコードを含むブロックを示しています。コース 22 のレコードを削除すると、フラグが 0 (「空」) に設定され、パート (b) に示すように、レコードはそのまま残ります。このスペースは、挿入に使用できるようになります。ただし、レコードをスペースに挿入できるのは、タイトル フィールドの文字数が 9 文字以下である場合のみです。一般に、小さなレコードが削除されたために多数の空きスペースが残っていても、新しいレコードがブロックに収まらない場合があります。このブロックは断片化されていると言われます。

この断片化を減らす方法は、残りのレコードをすべてブロックの一方の端にグループ化するようにシフトすることです。ただし、これを行うとシフトされたレコードのスロット番号が変わり、残念ながら ID も変わります。

この問題を解決するには、ID テーブルを使用して、レコードのスロット番号とページ内の位置を切り離します。ID テーブルは、ページの先頭に格納される整数の配列です。配列内の各スロットはレコード ID を表します。配列スロットの値は、その ID を持つレコードの位置です。値が 0 の場合、その ID を持つレコードは現在ないことを意味します。図 6.8c は、図 6.8b と同じデータですが、ID テーブルが含まれています。ID テーブルには 3 つのエントリが含まれます。そのうち 2 つはブロックのオフセット 63 と 43 にあるレコードを指し、もう 1 つは空です。位置 63 のレコードの ID は 0、位置 43 のレコードの ID は 2 です。現在、ID 1 を持つレコードはありません。

ID テーブルは、レコードマネージャがブロック内でレコードを移動できるようにする間接レベルを提供します。レコードが移動すると、ID テーブル内のエントリが調整されます。

同様に、レコードが削除された場合、そのエントリは 0 に設定されます。新しいレコードが挿入されると、レコード マネージャは配列内で使用可能なエントリを見つけ、それを新しいレコードの ID として割り当てます。このように、ID テーブルにより、各レコードに固定の識別子を提供しながら、可変長レコードをブロック内で移動できるようになります。

ID テーブルは、ブロック内のレコード数が増えるにつれて拡張されます。ブロックには可変長レコードをさまざまな数だけ保持できるため、配列のサイズは必然的にオープンエンドになります。通常、ID テーブルはブロックの一方の端に配置され、レコードはもう一方の端に配置され、互いに向かって拡張されます。この状況は、ブロックの最初のレコードが右端にある図 6.8c に示されています。

ID テーブルにより、空/未使用フラグは不要になります。ID テーブルのエントリがレコードを指している場合、そのレコードは使用中です。空のレコードの ID は 0 です (実際には存在しません)。また、ID テーブルにより、レコード マネージャーはブロック内の各レコードをすばやく見つけることができます。特定の ID を持つレコードに移動するには、レコード マネージャーは ID テーブルのそのエントリに格納されている場所を使用するだけです。次のレコードに移動するには、レコード マネージャーは ID テーブルをスキャンして、次の 0 以外のエントリを見つけます。

6.2.3 スパンドレコードの実装

このセクションでは、スパンされたレコードを実装する方法について説明します。レコードがスパンされていない場合、各ブロックの最初のレコードは常に同じ場所から始まります。スパンされたレコードでは、この状況は当てはまりません。したがって、レコード マネージャーは、最初のレコードのオフセットを保持するために、各ブロックの先頭に整数を格納する必要があります。

たとえば、図 6.9 を考えてみましょう。ブロック 0 の最初の整数は 4 で、最初のレコード R1 がオフセット 4 (つまり、整数の直後) から始まることを示しています。レコード R2 はブロック 0 と 1 にまたがっているため、ブロック 1 の最初のレコードはオフセット 60 から始まる R3 です。レコード R3 はブロック 2 を通ってブロック 3 に続きます。レコード R4 はブロック 3 の最初のレコードで、オフセット 30 から始まり、ブロック 2 の最初の整数は 0 で、そのブロックにはレコードがないことを示しています。

レコード マネージャは、スパンされたレコードを 2 つの異なる方法で分割できます。最初の方法は、ブロックを可能な限りいっぱいにして、ブロック境界で分割することです。残りのバイトは、ファイルの次のブロックに配置されます。2 番目の方法は、レコード値を値ごとに書き込むことです。ページがいっぱいになると、書き込みは新しいページに続行されます。最初の方法には、スペースをまったく無駄にしないという利点がありますが、値がブロックにまたがって分割されるという欠点があります。分割された値にアクセスするには、レコード マネージャは、2 つのブロックのバイトを連結して値を再構築する必要があります。

図6.9 スパンドレコードの実装

6.2.4 非同種レコードの実装

レコード マネージャーが非同種レコードをサポートする場合、異なるテーブルのレコードは同じサイズである必要はないため、可変長レコードもサポートする必要があります。ブロック内に非同種レコードが存在することに関連する問題が 2 つあります。

- レコード マネージャーは、ブロック内の各レコード タイプのレイアウトを把握する必要が
- レコードが与えられた場合、レコード マネージャーは、そのレコードがどのテーブルから

レコード マネージャーは、各テーブルごとに 1 つずつレイアウトの配列を保持することで、最初の問題に対処できます。レコード マネージャーは、各レコードの先頭に追加の値を追加することで、2 番目の問題に対処できます。この値は、タグ値と呼ばれることもあり、レイアウト配列のインデックスであり、レコードが属するテーブルを指定します。

たとえば、DEPT テーブルと STUDENT テーブルの非同種ブロックを示す図 6.2 をもう一度考えてみましょう。レコード マネージャーは、これら両方のテーブルのレイアウト情報を含む配列を保持します。DEPT 情報は配列のインデックス 0 にあり、STUDENT 情報はインデックス 1 にあると仮定します。この場合、DEPT の各レコードのタグ値は 0 になり、STUDENT の各レコードのタグ値は 1 になります。

レコード マネージャの動作に大きな変更は必要ありません。レコード マネージャがレコードにアクセスすると、タグ値からどのテーブル情報を使用するかを決定します。その後、同種の場合と同じように、そのテーブルを使用して任意のフィールドの読み取りまたは書き込みを行うことができます。

SimpleDB のログ レコードは、非同種レコードの例です。各ログ レコードの最初の値は、ログ レコードの種類を示す整数です。リカバリ マネージャーはその値を使用して、レコードの残りの部分を読み取る方法を決定します。

6.3 SimpleDB レコード ページ

次の 2 つのセクションでは、セクション 6.2.1 の基本的なレコード マネージャーを実装する SimpleDB レコード マネージャーについて説明します。このセクションではレコード ページの実装について説明し、次のセクションではレコード ページのファイルを実装する方法について説明します。章末の演習の一部では、他の設計上の決定を処理するためにレコード マネージャーを変更するように求められます。

6.3.1 レコード情報の管理

SimpleDB レコード マネージャーは、レコードの情報を管理するために Schema クラスと Layout クラスを使用します。それらの API は図 6.10 に示されています。

Schema

public Schema(); public void addField(String fldname, int type, int length); public void addIntField(String fldname); public void addStringField(String fldname, int length); public void add(String fldname, Schema sch); public void addAll(Schema sch);

パブリック List<String> fields(); パブリック boolean hasField(String fldname); パブリック int type(String fldname); パブリック int length(String fldname);

Layout

パブリックレイアウト(スキーマスキーマ); パブリックレイアウト(スキーマ スキーマ、マップ<文字列、整数>オフセット、
int スロットサイズ);

パブリック スキーマ schema(); パブリック int offset(String fldname); パブリック int slotSize();

図6.10 SimpleDBレコード情報のAPI

スキーマ オブジェクトは、レコードのスキーマ、つまり各フィールドの名前とタイプ、および各文字列フィールドの長さを保持します。この情報は、ユーザーがテーブルを作成するときに指定するものに対応しており、物理的な情報は含まれていません。たとえば、文字列の長さは、バイト単位のサイズではなく、許可される最大文字数です。

スキーマは、[フィールド名、タイプ、長さ] という形式の 3 つの要素のリストと考えることができます。Schema クラスには、リストに 3 つの要素を追加するための 5 つのメソッドがあります。addField メソッドは、3 つの要素を明示的に追加します。addIntField、addStringField、add、および addAll メソッドは便利なメソッドです。これらのメソッドの最初の 2 つは 3 つの要素を計算し、最後の 2 つは既存のスキーマから 3 つの要素をコピーします。このクラスには、フィールド名のコレクションを取得したり、指定されたフィールドがコレクション内にあるかどうかを確認したり、指定されたフィールドのタイプと長さを取得したりするためのアクセサ メソッドもあります。

Layout クラスには、レコードに関する物理情報も含まれています。このクラスは、フィールドとスロットのサイズ、およびスロット内のフィールド オフセットを計算します。このクラスには、Layout オブジェクトを作成する 2 つの理由に対応する 2 つのコンストラクターがあります。最初のコンストラクターは、テーブルの作成時に呼び出され、指定されたスキーマに基づいてレイアウト情報を計算します。2 番目のコンストラクターは、テーブルの作成後に呼び出され、クライアントは以前に計算された値を提供するだけです。

図6.11のコード断片は、これら2つのクラスの使用法を示しています。コードの最初の部分は、COURSEテーブルの3つのフィールドを含むスキーマを作成します。

```
スキーマ sch = new Schema(); sch.addIntField
("cid"); sch.addStringField("title", 20); sch.add
IntField("deptid"); レイアウト layout = new L
ayout(sch);


for (String fldname : layout.schema().fields()) { int offset = layout.offset(fldname);
System.out.println(fldname + " には offset " + offset); }
```

図6.11 COURSEレコードの構造の指定

そして、そこからレイアウト オブジェクトを作成します。コードの 2 番目の部分では、各フィールドの名前とオフセットを出力します。

6.3.2 スキーマとレイアウトの実装

Schema クラスのコードは単純で、図 6.12 に示されています。内部的には、クラスはフィールド名をキーとするマップにトリプルを格納します。フィールド名に関連付けられたオブジェクトは、フィールドの長さと型をカプセル化するプライベート クラス FieldInfo に属します。

タイプは、JDBC クラス Types で定義されている定数 INTEGER および VARCHAR で示されます。フィールドの長さは文字列フィールドに対してのみ意味を持ちます。メソッド addIntField は整数に長さの値 0 を与えますが、この値はアクセスされることがないため無関係です。

レイアウトのコードは図 6.13 に示されています。最初のコンストラクタは、フィールドをスキーマに現れる順序で配置します。各フィールドの長さをバイト単位で決定し、フィールド長の合計としてスロット サイズを計算し、整数サイズの空/未使用フラグ用に 4 バイトを追加します。フラグをスロットのオフセット 0 に割り当て、各フィールドのオフセットを前のフィールドが終了する位置 (つまり、パディングなし) に割り当てます。

6.3.3 ページ内のレコードの管理

RecordPage クラスはページ内のレコードを管理します。その API は図 6.14 に示されています。

nextAfter メソッドと insertAfter メソッドは、ページ内で目的のレコードを検索します。 nextAfter メソッドは、指定されたスロットの後に続く最初の使用済みスロットを返します。空のスロットはスキップされます。負の戻り値は、残りのスロットがすべて空であることを示します。 insertAfter メソッドは、指定されたスロットの後に続く最初の空スロットを検索します。空のスロットが見つかった場合、メソッドはフラグを USED に設定し、スロット番号を返します。それ以外の場合、メソッドは "1" を返します。

```
パブリック クラス Schema { private List<String> fields = new ArrayList<>(); private Map<String,FieldInfo> info = new HashMap<>(); public void addField(String fldname, int type, int length) { fields.add(fldname); info.put(fldname, new FieldInfo(type, length)); }public void addIntField(String fldname) { addField(fldname, INTEGER, 0); }public void addStringField(String fldname, int length) { addField(fldname, VARCHAR, length); }public void add(String fldname, Schema sch) { int type = sch.type(fldname); int length = sch.length(fldname); addField(fldname, type, length); }public void addAll(Schema sch) { for (String fldname : sch.fields()) add(fldname, sch); }public List<String> fields() { return fields; }public boolean hasField(String fldname) { return fields.contains(fldname); }public int type(String fldname) { return info.get(fldname).type; }public int length(String fldname) { return info.get(fldname).length; }class FieldInfo { int type, length; public FieldInfo(int type, int length) { this.type = type; this.length = length; } } }
```

図6.12 SimpleDBクラススキーマのコード

```
public class Layout { private Schema schema; private Map<String,Integer> offsets; private int
slotsize; public Layout(Schema schema) { this.schema = schema; offsets = new HashMap
<> ( ) ; int pos = Integer.BYTES; // 空/未使用フラグ用のスペース for (String fldname : sche
ma.fields()) { offsets.put(fldname, pos); pos += lengthInBytes(fldname); }slotsize = pos; }publ
ic Layout(Schema schema, Map<String,Integer> offsets, int slotsize) { this.schema = schema; t
his.offsets = offsets; this.slotsize = slotsize; }public Schema schema() { return schema; }public
int offset(String fldname) { return offsets.get(fldname); }public int slotSize() { return slotsize;
}private int lengthInBytes(String fldname) { int fldtype = schema.type(fldname); if (fldtype ==
INTEGER) return Integer.BYTES; else // fldtype == VARCHAR return Page.maxLength(sche
ma.length(fldname)); }
```

```
}
```

図6.13 SimpleDBクラスのコードレイアウト

get/set メソッドは、指定されたレコード内の指定されたフィールドの値にアクセスします。delete メソッドは、レコードのフラグを EMPTY に設定します。format メソッドは、ページ内のすべてのレコード スロットにデフォルト値を設定します。各空/未使用フラグを EMPTY に、すべての整数を 0 に、すべての文字列を "" に設定します。

RecordPage

```
public RecordPage(Transaction tx, BlockId blk, Layout layout);
public BlockId block();

public int    getInt   (int slot, String fldname);
public String getString(int slot, String fldname);
public void   setInt   (int slot, String fldname, int val);
public void   setString(int slot, String fldname, String val);
public void   format();
public void   delete(int slot);

public int    nextAfter(int slot);
public int    insertAfter(int slot);
```

図6.14 SimpleDBレコードページのAPI

RecordTest クラスは RecordPage メソッドの使用法を示しています。そのコードは図 6.15 に示されています。これは、整数フィールド A と文字列フィールド B の 2 つのフィールドを持つレコード スキーマを定義します。次に、新しいブロックの RecordPage オブジェクトを作成し、それをフォーマットします。for ループは insertAfter メソッドを使用して、ランダムな値のレコードでページを埋めます (各 A 値は 0 から 49 の間のランダムな数値で、B 値はその数値の文字列バージョンです)。2 つの while ループは nextAfter メソッドを使用してページを検索します。最初のループは選択されたレコードを削除し、2 番目のループは残りのレコードの内容を出力します。

6.3.4 レコードページの実装

SimpleDB は、図 6.5 のスロット ページ構造を実装します。唯一の違いは、空/未使用フラグが 1 バイトではなく 4 バイトの整数として実装されていることです (これは、SimpleDB がバイト サイズの値をサポートしていないためです)。RecordPage クラスのコードは、図 6.16 に示されています。

プライベート メソッド offset は、スロット サイズを使用してレコード スロットの開始位置を計算します。get/set メソッドは、フィールドのオフセットをレコードのオフセットに追加して、指定されたフィールドの位置を計算します。nextAfter メソッドと insertAfter メソッドは、プライベート メソッド searchAfter を呼び出して、それぞれ指定されたフラグ USED または EMPTY を持つスロットを検索します。searchAfter メソッドは、指定されたフラグを持つスロットが見つかるか、スロットがなくなるまで、指定されたスロットを繰り返し増分します。delete メソッドは、指定されたスロットのフラグを EMPTY に設定し、insertAfter は見つかったスロットのフラグを USED に設定します。

パブリック クラス RecordTest { public static void main(String[] args) throws Exception { SimpleDB db = new SimpleDB("recordtest", 400, 8); トランザクション tx = db.newTx(); スキーマ sch = new Schema(); sch.addIntField("A"); sch.addStringField("B", 9); レイアウト layout = new Layout(sch); for (String fldname : layout.schema().fields()) { int offset = layout.offset(fldname); System.out.println(fldname + " has offset " + offset); }BlockId blk = tx.append("testfile"); tx.pin(blk); RecordPage rp = new RecordPage(tx, blk, layout); rp.format(); System.out.println("ランダムなレコードでページを埋めています。"); int slot = rp.insertAfter(-1); while (slot >= 0) { int n = (int) Math.round(Math.random() * 50); rp.setInt(slot, "A", n); rp.setString(slot, "B", "rec"+n); System.out.println("スロット " + slot + "に挿入しています: {" + n + ", " + "rec"+n + "}"); slot = rp.insertAfter(slot); }System.out.println("A 値 < 25 のレコードを削除しました。"); int count = 0; slot = rp.nextAfter(-1); while (slot >= 0) { int a = rp.getInt(slot, "A"); String b = rp.getString(slot, "B"); if (a < 25) { count++; System.out.println("slot " + slot + ": {" + a + ", " + b + "}"); rp.delete(slot); }slot = rp.nextAfter(slot); }System.out.println(count + " 個の 25 未満の値が削除されました。\n"); System.out.println("残りのレコードは次のとおりです。"); slot = rp.nextAfter(-1); while (slot >= 0) { int a = rp.getInt(slot, "A"); String b = rp.getString(slot, "B"); System.out.println("slot " + slot + ": {" + a + ", " + b + "}"); slot = rp.nextAfter(slot); }tx.unpin(blk); tx.commit(); }

}

図6.15 RecordPageクラスのテスト

pパブリッククラス RecordPage { public static final int EMPTY = 0, USED = 1; private Transa ction tx; private BlockId blk; private Layout layout; public RecordPage(Transaction tx, BlockId blk, Layout layout) { this.tx = tx; this.blk = blk; this.layout = layout; tx.pin(blk); }public int getI nt(int slot, String fldname) { int fldpos = offset(slot) + layout.offset(fldname); return tx.getInt(bl k, fldpos); }public String getString(int slot, String fldname) { int fldpos = offset(slot) + layout.of fset(fldname); return tx.getString(blk, fldpos); }public void setInt(int slot, String fldname, int val) { int fldpos = offset(slot) + layout.offset(fldname); tx.setInt(blk, fldpos, val, true); }public void setString(int slot, String fldname, String val) { int fldpos = offset(slot) + layout.offset(fldname); tx.setString(blk, fldpos, val, true); }public void delete(int slot) { setFlag(slot, EMPTY); }public void format() { int slot = 0; while (isValidSlot(slot)) { tx.setInt(blk, offset(slot), EMPTY, false); スキーマ sch = layout.schema(); for (String fldname : sch.fields()) { int fldpos = offset(slot) + l ayout.offset(fldname); if (sch.type(fldname) == INTEGER) tx.setInt(blk, fldpos, 0, false); elsetx .setString(blk, fldpos, "", false); }slot++; } }

図6.16 SimpleDBクラスRecordPageのコード

```
public int insertAfter(int slot) { int newslot = searchAfter(slot, EMPT
Y); if (newslot >= 0) setFlag(newslot, USED); return newslot; }publi
c BlockId block() { return blk; }// プライベート補助メソッド priva
te void setFlag(int slot, int flag) { tx.setInt(blk, offset(slot), flag, true);
}private int searchAfter(int slot, int flag) { slot++; while (isValidSlot(
slot)) { if (tx.getInt(blk, offset(slot)) == flag) return slot; slot++; }ret
urn -1; }private boolean isValidSlot(int slot) { return offset(slot+1)
<= tx.blockSize(); }private int offset(int slot) { return slot * layout.slo
tSize(); }public int nextAfter(int slot) { return searchAfter(slot, USE
D); }
```

```
}
```

図6.16（続き）

6.4 SimpleDB テーブルスキャン

レコード ページはレコードのブロックを管理します。このセクションでは、ファイルの複数のブロックに任意の数のレコードを格納するテーブル スキャンについて説明します。

6.4.1 テーブルスキャン

TableScan クラスはテーブル内のレコードを管理します。その API を図 6.17 に示します。

TableScanオブジェクトは現在のレコードを追跡し、そのメソッドは現在のレコードを変更し、その内容にアクセスします。beforeFirstメソッドは、現在のレコードをファイルの最初のレコードの前に配置し、nextメソッドは、現在のレコードをファイル内の次のレコードに配置します。現在のブロックにレコードがない場合は、

TableScan

```
public TableScan(Transaction tx, String tblname,
                                 Layout layout);
public void    close();
public boolean hasField(String fldname);

// methods that establish the current record
public void    beforeFirst();
public boolean next();
public void    moveToRid(RID r);
public void    insert();

// methods that access the current record
public int     getInt(String fldname);
public String  getString(String fldname);
public void    setInt(String fldname, int val);
public void    setString(String fldname, String val);
public RID     currentRid();
public void    delete();
```

RID

```
public RID(int blknum, int slot);
public int     blockNumber();
public int     slot();
```

図6.17 SimpleDBテーブルスキャンのAPI

next は、別のレコードが見つかるまでファイル内の後続のブロックを読み取ります。レコードが見つからない場合、 next の呼び出しは false を返します。get/set メソッドと delete メソッドは、現在のレコードに適用されます。挿入メソッドは、現在のレコードのブロックから始めて、ファイル内のどこかに新しいレコードを挿入します。RecordPage の挿入メソッドとは異なり、この挿入メソッドは常に成功します。ファイルの既存のブロック内にレコードを挿入する場所が見つからない場合は、ファイルに新しいブロックを追加し、そこにレコードを挿入します。

ファイル内の各レコードは、ファイル内のブロック番号とブロック内のスロットという 2 つの値で識別できます。これらの 2 つの値は、レコード識別子 (または RID) と呼ばれます。クラス RID は、これらのレコード識別子を実装します。そのクラス コンストラクターは 2 つの値を保存し、アクセサー メソッド blockNumber と slot はそれらの値を取得します。

TableScan クラスには、rid と対話する 2 つのメソッドが含まれています。moveToRid メソッドは、現在のレコードを指定された rid に配置し、currentRid メソッドは、現在のレコードの rid を返します。

TableScan クラスは、これまで見てきた他のクラスとは大きく異なる抽象化レベルを提供します。つまり、Page、Buffer、Transaction、および RecordPage のメソッドはすべて、特定のブロックに適用されます。一方、TableScan クラスは、ブロック構造をクライアントから隠します。一般に、クライアントは、現在どのブロックにアクセスしているかを知りません (または気にしません)。

```
public class TableScanTest { public static void main(String[] args) throws Exception { SimpleDB db = new SimpleDB("tabletest", 400, 8); Transaction tx = db.newTx(); Schema sch = new Schema(); sch.addIntField("A"); sch.addStringField("B", 9); Layout layout = new Layout(sch); for (String fldname : layout.schema().fields()) { int offset = layout.offset(fldname); System.out.println(fldname + " has offset " + offset); }TableScan ts = new TableScan(tx, "T", layout); System.out.println("テーブルに 50 個のランダム レコードを入力します。"); ts.beforeFirst(); for (int i=0; i<50; i++) { ts.insert(); int n = (int) Math.round(Math.random() * 50); ts.setInt("A", n); ts.setString("B", "rec"+n); System.out.println("スロット " + ts.getRid() + "に挿入しています: {" + n + ", " + "rec"+n + "}"); }System.out.println("A値が25のレコードを削除しています 。"); int count = 0; ts.beforeFirst(); while (ts.next()) { int a = ts.getInt("A"); String b = ts.getString("B"); if (a < 25) { count++; System.out.println("slot " + ts.getRid() + ": {" + a + ", " + b + "}"); ts.delete(); } }System.out.println(count + " 個の 10 未満の値が削除されました。\n"); System.out.println("残りのレコードは次のとおりです。"); ts.beforeFirst(); while (ts.next()) { int a = ts.getInt("A"); String b = ts.getString("B"); System.out.println("スロット " + ts.getRid() + ": {" + a + ", " + b + "}"); }ts.close(); tx.commit(); }

}
```

図6.18 テーブルスキャンのテスト

図 6.18 の TableScanTest クラスは、テーブル スキャンの使用法を示しています。コードは RecordTest に似ていますが、ファイルに 50 レコードを挿入する点が異なります。ts.insert の呼び出しにより、レコードを保持するために必要な数の新しいブロックが割り当てられます。この場合、3 つのブロックが割り当てられます (ブロックあたり 18 レコード)。ただし、コードではこれが行われていることを認識していません。このコードを複数回実行すると、ファイルにさらに 50 レコードが挿入され、以前に削除されたレコードによって放棄されたスロットが埋められることがわかります。

6.4.2 テーブルスキャンの実装

TableScan クラスのコードは図 6.19 に示されています。TableScan オブジェクトは、現在のブロックのレコード ページを保持します。get/set/delete メソッドは、レコード ページの対応するメソッドを呼び出すだけです。現在のブロックが変更されると、プライベート メソッド moveToBlock が呼び出されます。このメソッドは、現在のレコード ページを閉じて、指定されたブロックの最初のスロットの前に配置された別のレコード ページを開きます。

次の方法のアルゴリズムは次のとおりです。

1. 現在のレコード ページの次のレコードに移動します。2. そのページにレコードがない場合は、ファイルの次のブロックに移動して次のレコードを取得します。3. 次のレコードが見つかるか、ファイルの終わりに達するまで続行します。

ファイルの複数のブロックが空になる可能性があるので (演習 6.2 を参照)、 next の呼び出しでは複数のブロックをループする必要がある場合があります。

挿入メソッドは、現在のレコードの後に新しいレコードを挿入しようとします。現在のブロックがいっぱいの場合は、次のブロックに移動し、空きスロットが見つかるまで続行します。すべてのブロックがいっぱいの場合は、ファイルに新しいブロックを追加し、そこにレコードを挿入します。 TableScan は、UpdateScan インターフェイス (および拡張により Scan) を実装します。これらのインターフェイスはクエリの実行の中心であり、第 8 章で説明します。getVal メソッドと setVal メソッドも第 8 章で説明します。これらのメソッドは、Constant 型のオブジェクトを取得および設定します。定数は値型 (int や String など) を抽象化したものなので、特定のフィールドの型を知らなくてもクエリを簡単に表現できます。

RID オブジェクトは、ブロック番号とスロット番号の 2 つの整数の組み合わせです。したがって、RID クラスのコードは単純で、図 6.20 に示されています。

```
プblic クラス TableScan は UpdateScan を実装します { private Transaction tx; private Layout l
    ayout; private RecordPage rp; private String filename; private int currentslot; public TableScan(
    Transaction tx, String tblname, Layout layout)

                                                                                                {
        this.tx = tx; this.layout = layout; filename = tblname + ".tbl"; if (tx.size(f
    ilename) == 0) moveToNewBlock(); else moveToBlock(0); }// Scan を実装
    するメソッド public void close() { if (rp != null) tx.unpin(rp.block()); }publi
    c void beforeFirst() { moveToBlock(0); }public boolean next() { currentslot =
    rp.nextAfter(currentslot); while (currentslot < 0) { if (atLastBlock()) return fal
    se; moveToBlock(rp.block().number()+1); currentslot = rp.nextAfter(currents
    lot); }return true; }public int getInt(String fldname) { return rp.getInt(currents
    lot, fldname); }public String getString(String fldname) { return rp.getString(c
    urrentslot, fldname); }public Constant getVal(String fldname) { if (layout.sch
    ema().type(fldname) == INTEGER) return new IntConstant(getInt(fldname));
    elsereturn new StringConstant(getString(fldname)); }public boolean hasField(
    String fldname) { return layout.schema().hasField(fldname); }// UpdateScan
    を実装するメソッド public void setInt(String fldname, int val) {
```

図6.19 SimpleDBクラスTableScanのコード

```
    rp.setInt(currentslot, fldname, val); }public void setString(String fldname, String
val) { rp.setString(currentslot, fldname, val); }public void setVal(String fldname, Con
stant val) { if (layout.schema().type(fldname) == INTEGER) setInt(fldname, (Integer)
val.asJavaVal()); elsesetString(fldname, (String)val.asJavaVal()); }public void insert()
{ currentslot = rp.insertAfter(currentslot); while (currentslot < 0) { if (atLastBlock())
moveToNewBlock(); else moveToBlock(rp.block().number()+1); currentslot = rp.inse
rtAfter(currentslot); } }public void delete() { rp.delete(currentslot); }public void move
ToRid(RID rid) { close(); BlockId blk = new BlockId(filename, rid.blockNumber()); r
p = new RecordPage(tx, blk, layout); currentslot = rid.slot(); }public RID getRid() { r
eturn new RID(rp.block().number(), currentslot); }// プライベート補助メソッド pri
vate void moveToBlock(int blknum) { close(); BlockId blk = new BlockId(filename,
blknum); rp = new RecordPage(tx, blk, layout); currentslot = -1; }private void moveT
oNewBlock() { close(); BlockId blk = tx.append(filename); rp = 新しい RecordPage(
tx, blk, layout); rp.format(); currentslot = -1; }private boolean atLastBlock() { return r
p.block().number() == tx.size(filename) - 1; }
```

```
}
```

図6.19（続き）

```
パブリック クラス RID { プライベート int blknum; プライベート int slot
; パブリック RID(int blknum, int slot) { this.blknum = blknum; this.slot = sl
ot; }パブリック int blockNumber() { return blknum; }パブリック int slot()
{ return slot; }パブリック boolean equals(Object obj) { RID r = (RID) obj; re
turn blknum == r.blknum && slot==r.slot; }パブリック String toString() { r
eturn "[" + blknum + ", " + slot + "]"; } }
```

図6.20 SimpleDBクラスRIDのコード

6.5 章の要約

- レコード マネージャは、ファイルにレコードを保存するデータベース システムの一部です。レコード マネージャには、次の 3 つの基本的な役割があります。
  – レコード内にフィールドを配置する – ブロック内にレコードを配置する – ファイル内のレコードへのアクセスを提供する

レコード マネージャーを設計する際には、対処しなければならない問題がいくつかあります

- 1 つの問題は、可変長フィールドをサポートするかどうかです。固定長レコードは、フィールドをその場で更新できるため、簡単に実装できます。可変長フィールドを更新すると、レコードがブロックからあふれ、オーバーフロー ブロックに配置される可能性があります。
- SQL には、char、varchar、clob の 3 つの異なる文字列型があります。 – char 型は、固定長表現を使用して最も自然に実装されます。 – varchar 型は、可変長表現を使用して最も自然に実装されます。 – clob 型は、補助ファイルに文字列を格納する固定長表現を使用して最も自然に実装されます。

- 可変長レコードの一般的な実装手法は、ID テーブルを使用することです。テーブル内の各エントリは、ページ内のレコードを指します。レコードは、ID テーブル内のエントリを変更するだけで、ページ内を移動できます。
- 2 番目の問題は、スパン レコードを作成するかどうかです。スパン レコードは、スペースを無駄にせず、大きなレコードを作成できるため便利ですが、実装が複雑になります。
- 3 番目の問題は、ファイル内で非同種レコードを許可するかどうかです。非同種レコードを使用すると、関連するレコードを 1 つのページにまとめることができます。クラスタリングにより結合が非常に効率的になりますが、他のクエリのコストが高くなる傾向があります。レコード マネージャーは、各レコードの先頭にタグ フィールドを保存することで非同種レコードを実装できます。タグは、レコードが属するテーブルを示します。
- 4 番目の問題は、レコード内の各フィールドのオフセットをどのように決定するかです。レコード マネージャは、フィールドが適切なバイト境界に揃えられるようにフィールドを埋め込む必要がある場合があります。固定長レコード内のフィールドは、各レコードに対して同じオフセットを持ちます。可変長レコードでは、フィールドの先頭を検索する必要がある場合があります。

6.6 推奨される読み物

この章で紹介するアイデアとテクニックは、リレーショナル データベースの誕生当初から存在しています。Stonebraker ら (1976) のセクション 3.3 では、INGRES の最初のバージョンで採用されたアプローチについて説明しています。このアプローチでは、セクション 6.2.2 で説明した ID テーブルのバリエーションを使用しています。Astrahan ら (1976) のセクション 3 では、レコードを非同種的に格納していた初期の System R データベース システム (後に IBM の DB2 製品となる) のページ構造について説明しています。どちらの記事も、実装に関する幅広いアイデアについて説明しており、全文を読む価値があります。これらのテクニックのより詳細な説明と、サンプル レコード マネージャーの C ベースの実装については、Gray と Reuter (1993) の第 14 章で取り上げています。

各レコードをページ内に連続して格納する戦略は、必ずしも最善とは限りません。Ailamaki ら (2002) の論文では、ページ上のレコードを分割し、各フィールドの値を一緒に配置することを推奨しています。このレコード編成では、レコード マネージャによるディスク アクセスの回数は変わりませんが、データ キャッシュがより効率的に使用されるため、CPU のパフォーマンスが大幅に向上します。Stonebraker ら (2005) の論文では、さらに踏み込んで、テーブルをフィールド値別に編成すること、つまり、各フィールドのすべてのレコード値を一緒に格納することを提案しています。この論文では、フィールド ベースのストレージがレコード ベースのストレージよりもコンパクトになり、クエリの効率が向上することを示しています。非常に大きなレコードの実装戦略については、Carey ら (1986) で説明されています。

Ailamaki, A., DeWitt, D., & Hill, M. (2002). 深いメモリ階層上のリレーショナルデータベースのデータページレイアウト。VLDB Journal, 11(3), 198 – 215.
Astrahan, M., Blasgen, M., Chamberlin, D., Eswaren, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., &

Watson, V. (1976). System R: データベース管理へのリレーショナル アプローチ。ACM Transactions on Database Systems、1(2)、97 – 137。Carey, M.、DeWitt, D.、Richardson, J.、および Shekita, E. (1986). EXODUS 拡張可能データベース システムにおけるオブジェクトおよびファイル管理。VLDB カンファレンスの議事録 (pp. 91 – 100)。Gray, J.、および Reuter, A. (1993). トランザクション処理: 概念とテクニック。サンマテオ、カリフォルニア州: Morgan Kaufman。 Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O’Neil, E., O’Neil, P., Rasin, A., Tran, N., & Zdonik, S. (2005). C-Store: 列指向 DBMS. VLDB カンファレンスの議事録 (pp. 553 – 564). Stonebraker, M., Kreps, P., Wong, E., & Held, G. (1976). INGRES の設計と実装. ACM Transactions on Database Systems, 1(3), 189 – 222.

6.7 演習

概念上の問題

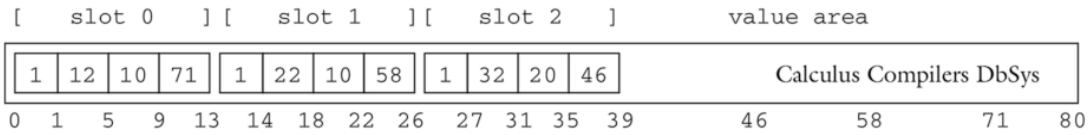
6.1. ブロック サイズが 400 バイトで、レコードがブロックにまたがることはできないと仮定します。10 バイト、20 バイト、50 バイト、100 バイトの各スロット サイズについて、SimpleDB レコード ページに収まるレコードの最大数とページ内の無駄なスペースの量を計算します。

6.2. テーブルのファイルにレコードのないブロックが含まれる場合があることを説明してく

6.3. 大学データベース内の各テーブル（STUDENT を除く）を検討します。

(a) 図 6.6 のように、そのテーブルのレイアウトを示します。(デモ クライアント ファイルの varchar 宣言を使用するか、すべての文字列フィールドが varchar(20) として定義されていると想定できます。) (b) 図 1.1 のレコードを使用して、各テーブルのレコード ページ (図 6.5 のように) の図を描きます。図 6.5 のように、空/フル フラグは 1 バイト長であると想定します。また、文字列フィールドは固定長で実装されていると想定します。 (c) パート (b) を実行しますが、文字列フィールドは可変長で実装されていると想定します。図 6.8c をモデルとして使用します。 (d) パート (b) と (c) の図を修正して、2 番目のレコードが削除された後のページの状態を示します。 6.4. 非常に大きな文字列を処理する別の方法は、データベースに格納しないことです。代わりに、文字列を OS ファイルに配置し、ファイル名をデータベースに格納することができます。この戦略では、clob 型が不要になります。この戦略があまり良くない理由をいくつか挙げてください。 6.5. 図 6.7b のように、オーバーフロー ブロックを含むブロックにレコードを挿入するとします。オーバーフロー ブロックにレコードを保存するのは良い考えでしょうか。説明してください。

6.6. 可変長レコードを実装する別の方法を紹介します。各ブロックには、固定長スロットのシーケンス (SimpleDB の場合と同じ) と可変長値が格納される場所の 2 つの領域があります。レコードはスロットに格納されます。固定長値はレコードとともに格納され、可変長値は値領域に格納されます。レコードには、値が配置されているブロック オフセットが含まれます。たとえば、図 6.8a のレコードは次のように格納できます。



(a) 可変長値が変更されるとどうなるか説明してください。オーバーフロー ブロックは必要ですか。必要な場合、どのようなものになるでしょうか。(b) このストレージ戦略を ID テーブルのストレージ戦略と比較してください。それぞれの利点を比較して説明してください。(c) どちらの実装戦略が好みですか。その理由は?

6.7. 空/未使用フラグごとにバイトを使用すると、必要なのは 1 ビットだけなので、スペースが無駄になります。別の実装方法としては、各スロットの空/未使用ビットをブロックの先頭のビット配列に格納します。このビット配列は、1 つ以上の 4 バイト整数として実装できます。

(a) このビット配列を図 6.8c の ID テーブルと比較してください。(b) ブロック サイズが 4K で、レコードが少なくとも 15 バイトであると仮定します。ビット配列を格納するために必要な整数の数はいくつですか。(c) 新しいレコードを挿入するための空きスロットを見つけるアルゴリズムを説明してください。(d) ブロック内の次の空でないレコードを見つけるアルゴリズムを説明してください。

プログラミングの問題

6.8. RecordPage クラスを修正して、そのブロックがコンストラクタによって固定されるのではなく、各 get/set メソッドの先頭で固定されるようにします。同様に、ブロックは各 get/set メソッドの終了時に固定解除されるため、close メソッドは不要になります。これは SimpleDB 実装よりも優れていると思いますか。説明してください。

6.9. varchar フィールドに可変長実装が含まれるようにレコード マネージャーを修正します。

6.10. SimpleDB は、ファイルを順方向にのみ読み取る方法を知っています。

(a) TableScan クラスと RecordPage クラスを修正して、previous メソッドと afterLast メソッドをサポートします。このメソッドは、現在のレコードをファイル (またはページ) 内の最後のレコードの後に配置します。 (b) TableScanTest プログラムを修正して、レコードを逆の順序で印刷します。

6.11. レコードがスパンされるようにレコード マネージャーを修正します。

6.12. Layout クラスを修正して、文字列フィールドのサイズが常に 4 の倍数になるようにパディングします。

6.13. SimpleDB レコード マネージャーを修正して、NULL フィールド値を処理します。特定の整数値または文字列値を使用して NULL を示すのは不合理なので、フラグを使用してどの値が NULL であるかを指定する必要があります。特に、レコードに N 個のフィールドが含まれているとします。この場合、各レコードに N 個の追加ビットを保存して、i 番目のフィールドの値が NULL の場合に限り i 番目のビットの値が 1 になるようにすることができます。N<32 と仮定すると、空/未使用の整数をこの目的に使用できます。この整数のビット 0 は、以前と同様に空/未使用を示します。ただし、他のビットには null 値情報が格納されます。コードに次の修正を加える必要があります。

- Layout を変更して、フィールドの null 情報ビットが配置されているフラグ内の位置を返すメソッド bitLocation(fl dname) を追加します。
- RecordPage と TableScan を変更して、2 つの追加のパブリック メソッド (フラグの適切なビットに 1 を格納する void メソッド setNull(fl dname)、および現在のレコードの指定されたフィールドの null ビットが 1 の場合に true を返すブール メソッド isNull(fl dname)) を追加します。
- RecordPage の format メソッドを変更して、新しいレコードのフィールドを明示的に null 以外に設定します。
- 指定されたフィールドを null 以外に設定するには、setString メソッドと setInt メソッドを変更します。

6.14. スキーマで指定された長さよりも長い文字列で setString が呼び出されたとします。

(a) どのような問題が発生する可能性があり、いつそれが検出されるかを説明します。 (b) エラーが検出され、適切に処理されるように SimpleDB コードを修正します。

第7章
メタデータ管理

前の章では、レコード マネージャーがレコードをファイルに格納する方法について説明しました。ただし、ファイルだけでは役に立たないことはおわかりでしょう。レコード マネージャーは、各ブロックの内容を「デコード」するために、レコードのレイアウトも把握する必要があります。レイアウトはメタデータの一例です。この章では、データベース エンジンでサポートされているメタデータの種類、その目的と機能、およびエンジンがメタデータをデータベースに格納する方法について説明します。

7.1 メタデータマネージャ

メタデータはデータベースを説明するデータです。データベース エンジンはさまざまなメタデータを保持します。例:

- テーブル メタデータは、各フィールドの長さ、タイプ、オフセットなど、テーブルのレコードの構造を記述します。レコード マネージャーによって使用されるレイアウトは、この種のメタデータの例です。
- ビュー メタデータは、各ビューの定義や作成者などのプロパティを記述します。このメタデータは、プランナーがビューを参照するクエリを処理するのに役立ちます。
- インデックス メタデータは、テーブルに定義されているインデックスを記述します (第 12 章で説明します)。プランナーはこのメタデータを使用して、クエリがインデックスを使用して評価できるかどうかを確認します。
- 統計メタデータは、各テーブルのサイズとフィールド値の分布を記述します。クエリ オプティマイザーはこのメタデータを使用して、クエリのコストを見積もります。

最初の 3 つのカテゴリのメタデータは、テーブル、ビュー、またはインデックスの作成時に生成されます。統計メタデータは、データベースが更新されるたびに生成されます。

メタデータ マネージャーは、メタデータを保存および取得するデータベース エンジンのコンポーネントです。SimpleDB メタデータ マネージャーは、4 つのメタデータ タイプそれぞれに対応する 4 つの個別のマネージャーで構成されています。この章の残りのセクションでは、これらのマネージャーについて詳しく説明します。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_7

7.2 テーブルメタデータ

SimpleDB クラス TableMgr はテーブル データを管理します。図 7.1 に示すその API は、コンストラクタと 2 つのメソッドで構成されています。コンストラクタは、システムの起動時に 1 回呼び出されます。メソッド createTable は、テーブルの名前とスキーマを引数として受け取り、レコードのオフセットを計算して、それをすべてカタログに保存します。メソッド getLayout はカタログにアクセスし、指定されたテーブルのメタデータを抽出して、メタデータを含むレイアウト オブジェクトを返します。

図7.2のTableMgrTestクラスはこれらのメソッドを示しています。まず、整数フィールド「A」と文字列フィールド「A」を含むスキーマを定義します。

TableMgr

public TableMgr(boolean isnew, Transaction tx); public void createTable(String tblname, Schema sch, Transaction tx); public Layout getLayout(String tblname, Transaction tx);

図7.1 SimpleDBテーブルマネージャのAPI

```
public class TableMgrTest {
   public static void main(String[] args) throws Exception {
      SimpleDB db = new SimpleDB("tblmgrtest", 400, 8);
      Transaction tx = db.newTx();
      TableMgr tm = new TableMgr(true, tx);

      Schema sch = new Schema();
      sch.addIntField("A");
      sch.addStringField("B", 9);
      tm.createTable("MyTable", sch, tx);

      Layout layout = tm.getLayout("MyTable", tx);
      int size = layout.slotSize();
      Schema sch2 = layout.schema();
      System.out.println("MyTable has slot size " + size);
      System.out.println("Its fields are:");
      for (String fldname : sch2.fields()) {
         String type;
         if (sch2.type(fldname) == INTEGER)
            type = "int";
         else {
            int strlen = sch2.length(fldname);
            type = "varchar(" + strlen + ")";
         }
         System.out.println(fldname + ": " + type);
      }
      tx.commit();
   }
}
```

図7.2 テーブルマネージャメソッドの使用

「B」。次に、createTable を呼び出して、このスキーマを持つ「MyTable」という名前のテーブルを作成します。次に、コードは getLayout を呼び出して、計算されたレイアウトを取得します。

メタデータ マネージャーは、カタログと呼ばれるデータベースの一部にメタデータを保存します。しかし、カタログはどのように実装されるのでしょうか。最も一般的な戦略は、データベース エンジンがカタログ情報をデータベース テーブルに保存することです。SimpleDB は、テーブル メタデータを保持するために 2 つのテーブルを使用します。テーブル tblcat には各テーブル固有のメタデータが格納され、テーブル fldcat には各テーブルの各フィールドに固有のメタデータが格納されます。これらのテーブルには次のフィールドがあります。

tblcat(TblName, スロットサイズ) fldcat(TblName, FldName, タイプ, 長さ, オフセット)

tblcat には各データベース テーブルに対して 1 つのレコードがあり、fldcat には各テーブルの各フィールドに対して 1 つのレコードがあります。SlotSize フィールドは、Layout によって計算されたスロットの長さをバイト単位で示します。Length フィールドは、テーブルのスキーマで指定されたフィールドの長さを文字単位で示します。例として、図 1.1 の大学データベースに対応するカタログ テーブルを図 7.3 に示します。テーブルのレイアウト情報が一連の fldcat レコードに「フラット化」されていることに注意してください。テーブル fldcat の Type 値には、値 4 と 12 が含まれています。これらの値は、JDBC クラス Types で定義されている INTEGER 型と VARCHAR 型のコードです。

カタログ テーブルには、ユーザーが作成したテーブルと同じようにアクセスできます。たとえば、図 7.4 の SQL クエリは、STUDENT テーブル内のすべてのフィールドの名前と長さを取得します。[1]

カタログ テーブルには、独自のメタデータを記述するレコードも含まれています。これらのレコードは図 7.3 には表示されていません。代わりに、演習 7.1 でそれらを決定するように求められます。図 7.5 は、各テーブルのレコード長と各フィールドのオフセットを出力する CatalogTest クラスのコードを示しています。コードを実行すると、カタログ テーブルのメタデータも出力されることがわかります。

図 7.6 は TableMgr のコードを示しています。コンストラクターはカタログ テーブル tblcat と fldcat のスキーマを作成し、それらのレイアウト オブジェクトを計算します。データベースが新規の場合は、2 つのカタログ テーブルも作成されます。

createTable メソッドは、テーブル スキャンを使用してカタログにレコードを挿入します。テーブルの tblcat に 1 つのレコードを挿入し、テーブルの各フィールドの flcat に 1 つのレコードを挿入します。

getLayout メソッドは、2 つのカタログ テーブルでテーブル スキャンを開始し、指定されたテーブル名に対応するレコードをスキャンします。次に、それらのレコードから要求されたレイアウト オブジェクトを構築します。

[1] Note that the constant “student” is in lower case, even though the table was defined in upper case. The reason is that all table and field names in SimpleDB are stored in lower case, and constants in SQL statements are case-sensitive.

tblcat	TblName	SlotSize
	student	30
	dept	20
	course	36
	section	28
	enroll	22

fldcat	TblName	FldName	Type	Length	Offset
	student	sid	4	0	4
	student	sname	12	10	8
	student	majorid	4	0	22
	student	gradyear	4	0	26
	dept	did	4	0	4
	dept	dname	12	8	8
	course	cid	4	0	4
	course	title	12	20	8
	course	deptid	4	0	32
	section	sectid	4	0	4
	section	courseid	4	0	8
	section	prof	12	8	12
	section	year	4	0	24
	enroll	eid	4	0	4
	enroll	studentid	4	0	8
	enroll	sectionid	4	0	12
	enroll	grade	12	2	16

図7.3 大学データベースのカタログテーブル

Fldcat から FldName、Length を選択します。TblName = は 'student' です。

図7.4 メタデータを取得するためのSQLクエリ

```
public class CatalogTest {
   public static void main(String[] args) throws Exception {
      SimpleDB db = new SimpleDB("catalogtest", 400, 8);
      Transaction tx = db.newTx();
      TableMgr tm = new TableMgr(true, tx);

      Schema sch = new Schema();
      sch.addIntField("A");
      sch.addStringField("B", 9);
      tm.createTable("MyTable", sch, tx);

      System.out.println("All tables and their lengths:");
      Layout layout = tm.getLayout("tblcat", tx);
      TableScan ts = new TableScan(tx, "tblcat", layout);
      while (ts.next()) {
         String tname = ts.getString("tblname");
         int size     = ts.getInt("slotsize");
         System.out.println(tname + " " + size);
      }
      ts.close();

      System.out.println("All fields and their offsets:");
      layout = tm.getLayout("fldcat", tx);
      ts = new TableScan(tx, "fldcat", layout);
      while (ts.next()) {
         String tname = ts.getString("tblname");
         String fname = ts.getString("fldname");
         int offset   = ts.getInt("offset");
         System.out.println(tname + " " + fname + " " + offset);
      }
      ts.close();
   }
}
```

図7.5 テーブルスキャンを使用してカタログテーブルを読み取る

7.3 メタデータの表示

ビューとは、クエリから動的に計算されるレコードを持つテーブルです。そのクエリはビューの定義と呼ばれ、ビューの作成時に指定されます。メタデータ マネージャーは、新しく作成された各ビューの定義を保存し、要求されたときにその定義を取得します。

SimpleDB クラス ViewMgr がこの役割を担います。このクラスは、ビュー定義をカタログ テーブル viewcat に、ビューごとに 1 つのレコードとして格納します。テーブルには次のフィールドがあります。

```
viewcat(ViewName, ViewDef)
```

プblic class TableMgr { public static final int MAX_NAME = 16; // テーブルまたはフィールド名 private Layout tcatLayout, fcatLayout; public TableMgr(boolean isNew, Transaction tx) { Schema tcatSchema = new Schema(); tcatSchema.addStringField("tblname", MAX_NAME); tcatSchema.addIntField("slotsize"); tcatLayout = new Layout(tcatSchema); Schema fcatSchema = new Schema(); fcatSchema.addStringField("tblname", MAX_NAME); fcatSchema.addStringField("fldname", MAX_NAME); fcatSchema.addIntField("type"); fcatSchema.addIntField("length"); fcatSchema.addIntField("offset"); fcatLayout = new Layout(fcatSchema); if (isNew) { createTable("tblcat", tcatSchema, tx); createTable("fldcat", fcatSchema, tx); } }public void createTable(String tblname, Schema sch, Transaction tx) { Layout layout = new Layout(sch); // tblcat に 1 つのレコードを挿入します TableScan tcat = new TableScan(tx, "tblcat", tcatLayout); tcat.insert(); tcat.setString("tblname", tblname); tcat.setInt("slotsize", layout.slotSize()); tcat.close(); // フィールドごとに fldcat にレコードを挿入します TableScan fcat = new TableScan(tx, "fldcat", fcatLayout); for (String fldname : sch.fields()) { fcat.insert(); fcat.setString("tblname", tblname); fcat.setString("fldname", fldname); fcat.setInt ("type", sch.type(fldname)); fcat.setInt ("length", sch.length(fldname)); fcat.setInt ("offset", layout.offset(fldname)); }fcat.close(); }public Layout getLayout(String tblname, Transaction tx) { int size = -1; TableScan tcat = new TableScan(tx, "tblcat", tcatLayout); while(tcat.next()) if(tcat.getString("tblname").equals(tblname)) { size = tcat.getInt("slotsize"); break; }

```
        tcat.close();
```

図7.6 SimpleDBクラスTableMgrのコード

文字列 fldname = fcat.getString("fldname"); int fldtype = fcat.getInt("type"); i
nt fldlen = fcat.getInt("length"); int offset = fcat.getInt("offset"); offsets.put(fldname, offset); sch.
addField(fldname, fldtype, fldlen); } fcat.close(); return new Layout(sch, offsets, size); } } スキ
ーマ sch = new Schema(); Map<String,Integer> offsets = new HashMap<String,Integer>(); Tab
leScan fcat = new TableScan(tx, "fldcat", fcatLayout); while(fcat.next()) if(fcat.getString("tblnam
e").equals(tblname)) {

図7.6（続き）

ViewMgr のコードは図 7.7 に示されています。そのコンストラクタはシステムの起動時に呼び出され、データベースが新規の場合は viewcat テーブルを作成します。createView メソッドと getViewDef メソッドはどちらもテーブル スキャンを使用してカタログ テーブルにアクセスします。createView はテーブルにレコードを挿入し、getViewDef は指定されたビュー名に対応するレコードを検索するためにテーブルを反復処理します。

ビュー定義は varchar 文字列として保存されるため、ビュー定義の長さには比較的小さな制限があります。現在の 100 文字の制限は、もちろん、ビュー定義が数千文字になる可能性があるため、まったく現実的ではありません。より良い選択は、ViewDef フィールドを clob(9999) などの clob 型として実装することです。

7.4 統計メタデータ

データベース システムによって管理されるメタデータのもう 1 つの形式は、データベース内の各テーブルに関する統計情報です。たとえば、レコード数やフィールド値の分布などです。これらの統計は、クエリ プランナーによってコストの見積もりに使用されます。経験上、統計を適切に設定すると、クエリの実行時間が大幅に短縮されることがわかっています。そのため、商用のメタデータ マネージャーは、各テーブルの各フィールドの値と範囲のヒストグラムや、異なるテーブル内のフィールド間の相関情報など、詳細で包括的な統計を維持する傾向があります。

簡潔にするために、このセクションでは次の 3 種類の統計情報のみを検討します。

- 各テーブルで使用されるブロックの数T
- 各テーブルTのレコード数
- 表Tの各フィールドFについて、T内の異なるF値の数

```
class ViewMgr { private static final int MAX_VIEWDEF = 100; // 最大ビュー定義文字 TableMgr tblMgr; public ViewMgr(boolean isNew, TableMgr tblMgr, Transaction tx) { this.tblMgr = tblMgr; if (isNew) { Schema sch = new Schema(); sch.addStringField("viewname", TableMgr.MAX_NAME); sch.addStringField("viewdef", MAX_VIEWDEF); tblMgr.createTable("viewcat", sch, tx); } }public void createView(String vname, String vdef, Transaction tx) { Layout layout = tblMgr.getLayout("viewcat", tx); TableScan ts = new TableScan(tx, "viewcat", layout); ts.setString("viewname", vname); ts.setString("viewdef", vdef); ts.close(); }public String getViewDef(String vname, Transaction tx) { String result = null; Layout layout = tblMgr.getLayout("viewcat", tx); TableScan ts = new TableScan(tx, "viewcat", layout); while (ts.next()) if (ts.getString("viewname").equals(vname)) { result = ts.getString("viewdef"); break; } ts.close(); return result; } }
```

図7.7 SimpleDBクラスViewMgrのコード

これらの統計はそれぞれB(T)、R(T)、V(T,F)で表されます。

図 7.8 は、大学データベースの統計例を示しています。値は、年間約 900 人の学生を受け入れ、年間約 500 のセクションを提供する大学に対応しています。大学はこの情報を過去 50 年間保存しています。図 7.8 の値は現実的なものになるように努めており、必ずしも図 1.1 から計算される値に対応するわけではありません。代わりに、図では、ブロックごとに 10 件の STUDENT レコード、ブロックごとに 20 件の DEPT レコードなどが格納されると想定しています。

STUDENT テーブルの V(T,F) 値を見てください。SId が STUDENT のキーであるということは、V(STUDENT, SId) ¼ 45,000 であることを意味します。割り当て V(STUDENT, SName) ¼ 44,960 は、45,000 人の学生のうち 40 人が重複した名前を持っていることを意味します。割り当て V(STUDENT, GradYear) ¼ 50 は、過去 50 年間で少なくとも 1 人の学生が卒業したことを意味します。そして割り当て V(STUDENT,

T	B(T)	R(T)	V(T,F)	
STUDENT	4,500	45,000	45,000	for F=SId
			44,960	for F=SName
			50	for F=GradYear
			40	for F=MajorId
DEPT	2	40	40	for F=DId, DName
COURSE	25	500	500	for F=CId, Title
			40	for F=DeptId
SECTION	2,500	25,000	25,000	for F=SectId
			500	for F=CourseId
			250	for F=Prof
			50	for F=YearOffered
ENROLL	50,000	1,500,000	1,500,000	for F=EId
			25,000	for F=SectionId
			45,000	for F=StudentId
			14	for F=Grade

図7.8 大学データベースに関する統計例

StatMgr

public StatMgr(TableMgr tm, Transaction tx); public StatInfo getStatInfo(String tblname, Layout lo, Transaction tx); *StatInfo* public int blocksAccessed(); public int recordsOutput(); public int distinctValues(String fldname);

図7.9 SimpleDBテーブル統計のAPI

MajorId) ¼ 40 は、40 の部門のそれぞれに、ある時点で少なくとも 1 つの専攻があったことを意味します。

SimpleDB の StatMgr クラスは、この統計情報を管理します。データベース エンジンは、1 つの StatMgr オブジェクトを保持します。このオブジェクトには、指定されたテーブルの StatInfo オブジェクトを返す getStatInfo メソッドがあります。StatInfo オブジェクトはそのテーブルの統計情報を保持し、blocksAccessed、recordsOutput、distinctValues メソッドを持ちます。これらのメソッドは、それぞれ統計関数 B(T)、R(T)、V(T,F) を実装します。これらのクラスの API は、図 7.9 に示されています。

```
SimpleDB db = ... トランザクション tx = db.newTx(); TableMgr tbl
mgr = ... StatMgr statmgr = new StatMgr(tblmgr, tx); レイアウト layo
ut = tblmgr.getLayout("student", tx);


StatInfo si = statmgr.getStatInfo("student", layout, tx); System.out.println(si.blocks
Accessed() + " " + si.recordsOutput() + " " + si.distinctValues("majorid")); tx.comm
it();
```

図7.10 テーブルに関する統計情報の取得と印刷

図 7.10 のコード フラグメントは、これらのメソッドの一般的な使用方法を示しています。このコードは、STUDENT テーブルの統計情報を取得し、B(STUDENT)、R(STUDENT)、および V(STUDENT、MajorId) の値を出力します。

データベース エンジンは、統計メタデータを 2 つの方法のいずれかで管理できます。1 つは、データベース カタログに情報を保存し、データベースが変更されるたびに更新する方法です。もう 1 つは、メモリに情報を保存し、エンジンが初期化されるときに計算する方法です。

最初のアプローチは、次のフィールドを持つ tblstats および fldstats という 2 つの新しいカタログ テーブルを作成することです。

```
tblstats(テーブル名、ブロック数、レコード数) fldsta
ts(テーブル名、フィールド名、値数)
```

tblstats テーブルには、テーブル T ごとに 1 つのレコードがあり、B(T) と R(T) の値が含まれます。flstats テーブルには、テーブル T ごとにフィールド F ごとに 1 つのレコードがあり、V(T,F) の値が含まれます。このアプローチの問題は、統計を最新の状態に保つためのコストです。insert、delete、setInt、setString を呼び出すたびに、これらのテーブルを更新する必要がある可能性があります。変更されたページをディスクに書き込むには、追加のディスク アクセスが必要になります。さらに、同時実行性が低下します。テーブル T を更新するたびに、T の統計レコードを含むブロックが xlock され、T の統計 (および同じページにレコードがある他のテーブルの統計) を読み取る必要があるトランザクションが待機状態になります。

この問題に対する実行可能な解決策の 1 つは、セクション 5.4.7 の read-uncommitted 分離レベルのように、トランザクションが slock を取得せずに統計情報を読み取れるようにすることです。データベース システムはこれらの統計情報を使用してクエリ プランの推定実行時間を比較するため、精度の低下は許容されます。したがって、統計情報は、生成される推定値が妥当である限り、正確である必要はありません。

2 番目の実装戦略は、カタログ テーブルを無視して、統計をメモリに直接保存することです。統計データは比較的小さく、メイン メモリに簡単に収まるはずです。唯一の問題は、サーバーを起動するたびに統計を最初から計算する必要があることです。この計算では、データベース内の各テーブルをスキャンして、表示されるレコード、ブロック、および値の数をカウントする必要があります。

データベースがそれほど大きくない場合、この計算によってシステムの起動がそれほど遅れることはありません。

このメイン メモリ戦略には、データベースの更新を処理するための 2 つのオプションがあります。最初のオプションは、これまでと同様に、データベースが更新されるたびに統計を更新することです。2 番目のオプションは、統計を更新せずに、定期的に最初から再計算することです。この 2 番目のオプションも、正確な統計情報は必要ないという事実に依存しており、統計を更新する前に統計が多少古くても許容されます。

SimpleDB は、2 番目のアプローチの 2 番目のオプションを採用しています。StatMgr クラスは、各テーブルのコスト情報を保持する tableStats という変数を保持します。このクラスには、指定されたテーブルのコスト値を返すパブリック メソッド statInfo と、コスト値を再計算するプライベート メソッド refreshStatistics および refreshTableStats があります。このクラスのコードは、図 7.11 に示されています。

StatMgr クラスは、statInfo が呼び出されるたびに増加するカウンターを保持します。カウンターが特定の値 (ここでは 100) に達すると、refreshStatistics が呼び出され、すべてのテーブルのコスト値が再計算されます。既知の値がないテーブルで statInfo が呼び出されると、refreshTableStats が呼び出され、そのテーブルの統計情報が計算されます。

refreshStatistics のコードは、tblcat テーブルをループします。ループの本体はテーブルの名前を抽出し、refreshTableStats を呼び出してそのテーブルの統計を計算します。refreshTableStats メソッドは、そのテーブルの内容をループしてレコードをカウントし、size を呼び出して使用されるブロックの数を決定します。簡単にするために、このメソッドはフィールド値をカウントしません。代わりに、StatInfo オブジェクトは、テーブル内のレコード数に基づいて、フィールドの個別の値の数を推測します。

StatInfo クラスのコードは図 7.12 に示されています。distinctValues は、渡されたフィールド値を使用しないことに注意してください。これは、フィールドの値の約 1/3 が一意であると単純に想定しているためです。言うまでもなく、この想定は非常に間違っています。演習 7.12 では、この状況を修正するように求められます。

7.5 インデックスメタデータ

インデックスのメタデータは、インデックス名、インデックス付けするテーブル名、インデックス付けするフィールドのリストで構成されます。インデックス マネージャは、このメタデータを保存および取得するシステム コンポーネントです。SimpleDB インデックス マネージャは、IndexMgr と IndexInfo の 2 つのクラスで構成されています。これらの API は、図 7.13 に示されています。

インデックスのメタデータは、インデックス名、インデックス付けされるテーブル名、インデックス付けされるフィールドで構成されます。IndexMgr メソッド createIndex は、このメタデータをカタログに格納します。getIndexInfo メソッドは、指定されたテーブルのすべてのインデックスのメタデータを取得します。特に、インデックス付けされたフィールドをキーとする Indexinfo オブジェクトのマップを返します。マップの keyset メソッドは、使用可能なインデックスを持つテーブルのフィールドを示します。IndexInfo メソッドは、StatInfo クラスと同様に、選択したインデックスに関する統計情報を提供します。メソッド

```
class StatMgr { private TableMgr tblMgr; private Map<String,StatInfo> tablestats; private int
  numcalls; public StatMgr(TableMgr tblMgr, Transaction tx) { this.tblMgr = tblMgr; refreshSt
  atistics(tx); }public synchronized StatInfo getStatInfo(String tblname, Layout layout, Transact
  ion tx) { numcalls++; if (numcalls > 100) refreshStatistics(tx); StatInfo si = tablestats.get(tbln
  ame); if (si == null) { si = calcTableStats(tblname, layout, tx); tablestats.put(tblname, si); }ret
  urn si; }private synchronized void refreshStatistics(Transaction tx) { tablestats = new HashMa
  p<String,StatInfo>(); numcalls = 0; Layout tcatlayout = tblMgr.getLayout("tblcat", tx); Tabl
  eScan tcat = new TableScan(tx, "tblcat", tcatlayout); while(tcat.next()) { String tblname = tcat
  .getString("tblname"); Layout layout = tblMgr.getLayout(tblname, tx); StatInfo si = calcTable
  Stats(tblname, layout, tx); tablestats.put(tblname, si); }tcat.close(); }private synchronized Stat
  Info calcTableStats(String tblname, Layout layout, Transaction tx) { int numRecs = 0; int nu
  mblocks = 0; TableScan ts = new TableScan(tx, tblname, layout); while (ts.next()) { numRecs
  ++; numblocks = ts.getRid().blockNumber() + 1; }ts.close(); return new StatInfo(numblocks,
  numRecs); }
}
```

図7.11 SimpleDBクラスStatMgrのコード

```
public class StatInfo { private int numBlocks; private int numRecs; public StatInfo(int num
blocks, int numrecs) { this.numBlocks = numblocks; this.numRecs = numrecs; }public int b
locksAccessed() { return numBlocks; }public int recordsOutput() { return numRecs; }publi
c int distinctValues(String fldname) { return 1 + (numRecs / 3); // これはまったく不正確
です。 } }
```

図7.12 SimpleDBクラスStatInfoのコード

IndexMgr

```
public IndexMgr(boolean isnew、TableMgr tmgr、StatMgr smgr、Transaction tx); public cr
eateIndex(String iname、String tname、String fname、Transaction tx); public Map(String、In
dexInfo> getIndexInfo(String tblname、Transaction tx); IndexInfo public IndexInfo(String inam
e、String tname、String fname、Transaction tx); public int blocksAccessed(); public int record
sOutput(); public int distinctValues(String fldname); public Index open();
```

図7.13 SimpleDBインデックスメタデータのAPI

blocksAccessed は、インデックスの検索に必要なブロック アクセスの数を返します (インデックスのサイズではありません)。メソッド recordsOutput と distinctiveValues は、インデックス内のレコードの数と、インデックス付きフィールドの個別の値の数を返します。これらの値は、インデックス付きテーブルの値と同じです。

IndexInfo オブジェクトには、インデックスの Index オブジェクトを返す open メソッドもあります。 Index クラスには、インデックスを検索するためのメソッドが含まれており、第 12 章で説明します。

```
SimpleDB db = ... トランザクション tx = db.newTx(); TableMgr tblmgr = ... StatM
gr statmgr = new StatMgr(tblmgr, tx); IndexMgr idxmgr = new IndexMgr(true, tblmg
r, statmgr, tx); idxmgr.createIndex("sidIdx", "student", "sid"); idxmgr.createIndex("sn
ameIdx", "student", "sname");

Map<String、 IndexInfo> indexes = idxmgr.getIndexInfo("student", tx); for (String fldname : inde
xes.keySet()) { IndexInfo ii = indexes.get(fldname); System.out.println(fldname + "\t" + ii.blocksA
ccessed(fldname)); }
```

図7.14 SimpleDBインデックスマネージャの使用

図 7.14 のコード フラグメントは、これらのメソッドの使用方法を示しています。このコードは、STUDENT テーブルに 2 つのインデックスを作成します。次に、そのメタデータを取得し、それぞれの名前と検索コストを出力します。

図 7.15 は IndexMgr のコードを示しています。これは、カタログ テーブル idxcat にインデックス メタデータを格納します。このテーブルには、インデックスごとに 1 つのレコードと、インデックスの名前、インデックス付けされるテーブルの名前、インデックス付けされるフィールドの名前の 3 つのフィールドがあります。

コンストラクタはシステムの起動時に呼び出され、データベースが新規の場合はカタログ テーブルを作成します。メソッド createIndex および getIndexInfo のコードは単純です。どちらのメソッドもカタログ テーブルでテーブル スキャンを開始します。メソッド createIndex はテーブルに新しいレコードを挿入します。メソッド getIndexInfo は指定されたテーブル名を持つレコードをテーブルで検索し、マップに挿入します。

IndexInfo クラスのコードは図 7.16 に示されています。コンストラクタは、インデックスの名前とインデックス フィールド、および関連するテーブルのレイアウトと統計メタデータを保持する変数を受け取ります。このメタデータにより、IndexInfo オブジェクトはインデックス レコードのスキーマを構築し、インデックス ファイルのサイズを見積もることができます。

open メソッドは、インデックス名とスキーマを HashIndex コンストラクタに渡すことによってインデックスを開きます。HashIndex クラスは静的ハッシュ インデックスを実装しており、第 12 章で説明します。代わりに B ツリー インデックスを使用するには、このコンストラクタをコメントアウトされたコンストラクタに置き換えます。blocksAccessed メソッドは、インデックスの検索コストを見積もります。最初に、インデックスのレイアウト情報を使用して各インデックス レコードの長さを決定し、インデックスのブロックあたりのレコード数 (RPB) とインデックス ファイルのサイズを見積もります。次に、インデックス固有のメソッド searchCost を呼び出して、そのインデックス タイプのブロック アクセス数を計算します。recordsOutput メソッドは、検索キーに一致するインデックス レコードの数を見積もります。また、distinctValues メソッドは、インデックス テーブルと同じ値を返します。

```
pパブリック クラス IndexMgr { private Layout layout; private TableMgr tblmgr; private StatMgr stat
  mgr; public IndexMgr(boolean isnew, TableMgr tblmgr, StatMgr statmgr, Transaction tx) { if (isnew)
  { Schema sch = new Schema(); sch.addStringField("indexname", MAX_NAME); sch.addStringField(
  "tablename", MAX_NAME); sch.addStringField("fieldname", MAX_NAME); tblmgr.createTable("id
  xcat", sch, tx); }this.tblmgr = tblmgr; this.statmgr = statmgr; layout = tblmgr.getLayout("idxcat", tx);
  }public void createIndex(String idxname, String tblname, String fldname,Transaction tx) { TableScan
  ts = new TableScan(tx, "idxcat", layout); ts.insert(); ts.setString("indexname", idxname); ts.setString("
  tablename", tblname); ts.setString("fieldname", fldname); ts.close(); }public Map<String,IndexInfo> g
  etIndexInfo(String tblname, Transaction tx) { Map<String,IndexInfo> result = new HashMap<String,I
  ndexInfo>(); TableScan ts = new TableScan(tx, "idxcat", layout); while (ts.next()) if (ts.getString("ta
  blename").equals(tblname)) { String idxname = ts.getString("indexname"); String fldname = ts.getStri
  ng("fieldname"); Layout tblLayout = tblmgr.getLayout(tblname, tx); StatInfo tblsi = statmgr.getStatIn
  fo(tblname, tbllayout, tx); IndexInfo ii = new IndexInfo(idxname, fldname, tblLayout.schema(),tx, tbls
  i); result.put(fldname, ii); }ts.close(); return result; }
```

```
}
```

図7.15 SimpleDBインデックスマネージャのコード

```
public class IndexInfo { private String idxname, fldname; private Transaction tx; private Schema tblSc
hema; private Layout idxLayout; private StatInfo si; public IndexInfo(String idxname, String fldname,
Schema tblSchema, Transaction tx, StatInfo si) { this.idxname = idxname; this.fldname = fldname; thi
s.tx = tx; this.idxLayout = createIdxLayout(); this.si = si; }public Index open() { Schema sch = schem
a(); return new HashIndex(tx, idxname, idxLayout); // return new BTreeIndex(tx, idxname, idxLayout)
; }public int blocksAccessed() { int rpb = tx.blockSize() / idxLayout.slotSize(); int numblocks = si.rec
ordsOutput() / rpb; return HashIndex.searchCost(numblocks, rpb); // return BTreeIndex.searchCost(nu
mblocks, rpb); }public int recordsOutput() { return si.recordsOutput() / si.distinctValues(fldname); }p
ublic int distinctValues(String fname) { return fldname.equals(fname) ? 1 : si.distinctValues(fldname);
}private Layout createIdxLayout() { Schema sch = new Schema(); sch.addIntField("block"); sch.addI
ntField("id"); if (layout.schema().type(fldname) == INTEGER) sch.addIntField("dataval"); else { int f
ldlen = layout.schema().length(fldname); sch.addStringField("dataval", fldlen); }return new Layout(sc
h); } }
```

図7.16 SimpleDBクラスIndexInfoのコード

MetadataMgr

public void createTable(String tblname, Schema sch, Transaction tx); public Layout getLayout(String tblname, Transaction tx); public void createView(String viewname, String viewdef, Transaction tx); public String getViewDef(String viewname, Transaction tx); public void createIndex(String idxname, String tblname, String fldname, Transaction tx); public Map<String,IndexInfo> getIndexinfo(String tblname, Transaction tx); public StatInfo getStatInfo(String tblname, Layout layout, Transaction tx);

図7.17 SimpleDBメタデータマネージャのAPI

7.6 メタデータマネージャの実装

SimpleDB は、TableMgr、ViewMgr、StatMgr、および IndexMgr という 4 つの個別のマネージャ クラスを非表示にすることで、メタデータ マネージャへのクライアント インターフェイスを簡素化します。代わりに、クライアントはメタデータを取得する単一の場所としてクラス MetadataMgr を使用します。MetadataMgr API のコードは、図 7.17 に示されています。

この API には、メタデータの種類ごとに 2 つのメソッドが含まれています。1 つのメソッドはメタデータを生成して保存し、もう 1 つのメソッドはメタデータを取得します。唯一の例外は統計メタデータで、その生成メソッドは内部的に呼び出されるため、非公開です。

図 7.18 は、これらのメソッドの使用方法を示す MetadataMgrTest クラスのコードを示しています。

パート 1 では、テーブル メタデータについて説明します。テーブル MyTable を作成し、図 7.2 のようにそのレイアウトを出力します。パート 2 では、統計マネージャについて説明します。MyTable に複数のレコードを挿入し、結果のテーブル統計を出力します。パート 3 では、ビュー マネージャについて説明します。ビューを作成し、ビュー定義を取得します。パート 4 では、インデックス マネージャについて説明します。フィールド A と B にインデックスを作成し、各インデックスの統計情報を出力します。

MetadataMgr クラスはファサード クラスとして知られています。そのコンストラクタは 4 つのマネージャ オブジェクトを作成し、それらをプライベート変数に保存します。そのメソッドは、個々のマネージャのパブリック メソッドを複製します。クライアントがメタデータ マネージャのメソッドを呼び出すと、そのメソッドは適切なローカル マネージャを呼び出して作業を実行します。そのコードは図 7.19 に示されています。

この本でこれまでに説明したすべてのテスト プログラムでは、3 つの引数を持つ SimpleDB コンストラクターを呼び出しています。このコンストラクターは、指定されたブロック サイズとバッファー プール サイズを使用して、システムの FileMgr、LogMgr、および BufferMgr オブジェクトをカスタマイズします。その目的は、システムの低レベルのデバッグを支援することであり、MetadataMgr オブジェクトは作成しません。

出版 ic クラス MetadataMgrTest {
public static void main(String[] args) throws Exception { SimpleDB db = new SimpleDB("metadatamgrtest", 400, 8); Transaction tx = db.newTx(); MetadataMgr mdm = new MetadataMgr(true, tx); Schema sch = new Schema(); sch.addIntField("A"); sch.addStringField("B", 9); // パート 1: テーブル メタデータ mdm.createTable("MyTable", sch, tx); レイアウト layout = mdm.getLayout("MyTable", tx); int size = layout.slotSize(); Schema sch2 = layout.schema(); System.out.println("MyTable のスロット サイズは " + size); System.out.println("そのフィールドは次のとおりです:"); for (String fldname : sch2.fields()) { String type; if (sch2.type(fldname) == INTEGER) type = "int"; else { int strlen = sch2.length(fldname); type = "varchar(" + strlen + ")"; }System.out.println(fldname + ": " + type); }// パート 2: 統計メタデータ TableScan ts = new TableScan(tx, "MyTable", layout); for (int i=0; i<50; i++) { ts.insert(); int n = (int) Math.round(Math.random() * 50); ts.setInt("A", n); ts.setString("B", "rec"+n); }StatInfo si = mdm.getStatInfo("MyTable", layout, tx); System.out.println("B(MyTable) = " + si.blocksAccessed()); System.out.println("R(MyTable) = " + si.recordsOutput()); System.out.println("V(MyTable,A) = " + si.distinctValues("A")); System.out.println("V(MyTable,B) = " + si.distinctValues("B")); // パート 3: ビューのメタデータ文字列 viewdef = "select B from MyTable where A = 1"; mdm.createView("viewA", viewdef, tx); String v = mdm.getViewDef("viewA", tx); System.out.println("View def = " + v); // パート 4: インデックス メタデータ mdm.createIndex("indexA", "MyTable", "A", tx); mdm.createIndex("indexB", "MyTable", "B", tx);

```
Map<String,IndexInfo> idxmap = mdm.getIndexInfo("MyTable", tx);
```

図7.18 MetadataMgrメソッドのテスト

```
        ii = idxmap.get("B"); System.out.println("B(indexB) = " + ii.blocksAccessed()); System.out
.println("R(indexB) = " + ii.recordsOutput()); System.out.println("V(indexB,A) = " + ii.distinctValu
es("A")); System.out.println("V(indexB,B) = " + ii.distinctValues("B")); tx.commit(); } } IndexInfo i
i = idxmap.get("A"); System.out.println("B(indexA) = " + ii.blocksAccessed()); System.out.println("
R(indexA) = " + ii.recordsOutput()); System.out.println("V(indexA,A) = " + ii.distinctValues("A"));
System.out.println("V(indexA,B) = " + ii.distinctValues("B"));
```

図7.18（続き）

SimpleDB には、データベース名という 1 つの引数を持つ別のコンストラクタがあります。このコンストラクタは、デバッグ以外の状況で使用されます。最初に、デフォルト値を使用してファイル、ログ、およびバッファ マネージャを作成します。次に、リカバリ マネージャを呼び出してデータベースをリカバリし (リカバリが必要な場合)、メタデータ マネージャを作成します (データベースが新規の場合は、カタログ ファイルの作成も含まれます)。2 つの SimpleDB コンストラクタのコードは、図 7.20 に示されています。1 つの引数を持つコンストラクタを使用すると、図 7.18 の MetadataMgrTest のコードは、図 7.21 に示すように、より単純に書き直すことができます。

7.7 章の要約

- メタデータは、データベースの内容とは別のデータベースに関する情報です。メタデータ マネージャーは、データベース システムの中でメタデータを保存および取得する部分です。
- SimpleDB のデータベース メタデータは、次の 4 つのカテゴリに分類されます。 – テーブル メタデータは、各フィールドの長さ、タイプ、オフセットなど、テーブルのレコードの構造を説明します。 – ビュー メタデータは、定義や作成者など、各ビューのプロパティを説明します。 – インデックス メタデータは、テーブルで定義されているインデックスを説明します。 – 統計メタデータは、各テーブルのサイズとフィールド値の分布を説明します。

- メタデータ マネージャーは、メタデータをシステム カタログに保存します。カタログは、多くの場合、カタログ テーブルと呼ばれるデータベース内のテーブルとして実装されます。カタログ テーブルは、データベース内の他のテーブルと同じようにクエリできます。

```
パブリック クラス MetadataMgr { private static TableMgr tblmgr; private static ViewMgr
viewmgr; private static StatMgr statmgr; private static IndexMgr idxmgr; パブリック Metad
ataMgr(boolean isnew, Transaction tx) { tblmgr = new TableMgr(isnew, tx); viewmgr = new
ViewMgr(isnew, tblmgr, tx); statmgr = new StatMgr(tblmgr, tx); idxmgr = new IndexMgr(is
new, tblmgr, statmgr, tx); }public void createTable(String tblname, Schema sch, Transaction t
x) { tblmgr.createTable(tblname, sch, tx); }public Layout getLayout(String tblname, Transact
ion tx) { return tblmgr.getLayout(tblname, tx); }public void createView(String viewname, Str
ing viewdef, Transaction tx) { viewmgr.createView(viewname, viewdef, tx); }public String g
etViewDef(String viewname, Transaction tx) { return viewmgr.getViewDef(viewname, tx); }
public void createIndex(String idxname, String tblname, String fldname, Transaction tx) { idx
mgr.createIndex(idxname, tblname, fldname, tx); }public Map<String,IndexInfo> getIndexIn
fo(String tblname, Transaction tx) {return idxmgr.getIndexInfo(tblname, tx); }}public StatInf
o getStatInfo(String tblname, Layout layout, Transaction tx) {return statmgr.getStatInfo(tblna
me, layout, tx);
}
```

図7.19 SimpleDBクラスMetadataMgrのコード

public SimpleDB(String dirname, int blocksize, int buffsize) { String homedir = System.get Property(HOME_DIR); File dbDirectory = new File(homedir, dirname); fm = new FileMgr (dbDirectory, blocksize); lm = new LogMgr(fm, LOG_FILE); bm = new BufferMgr(fm, lm , buffsize); }

public SimpleDB(String dirname) { this(dirname, BLOCK_SIZE, BUFFER_SIZE); Transaction tx = new Transaction(fm, lm, bm); boolean isnew = fm.isNew(); if (isne w) System.out.println("新しいデータベースを作成しています"); else { System.o ut.println("既存のデータベースを回復しています"); tx.recover(); }mdm = new MetadataMgr(isnew, tx); tx.commit();

}

図7.20 2つのSimpleDBコンストラクタ

パブリック クラス MetadataMgrTest { パブリック 静的 void main(String[] args) thro ws Exception { SimpleDB db = new SimpleDB("metadatamgrtest"); MetadataMgr mdm = db.mdMgr(); トランザクション tx = db.newTx(); ... }

図7.21 1引数のSimpleDBコンストラクタの使用

- テーブル メタデータは 2 つのカタログ テーブルに保存できます。1 つのテーブルにはテーブル情報 (スロット サイズなど) が保存され、もう 1 つのテーブルにはフィールド情報 (各フィールドの名前、長さ、タイプなど) が保存されます。
- ビュー メタデータは主にビュー定義で構成され、独自のカタログ テーブルに保存できます。ビュー定義は任意の長さの文字列になるため、可変長表現が適切です。
- 統計メタデータには、データベース内の各テーブルのサイズと値の分布に関する情報が保持されます。商用データベース システムでは、各テーブルの各フィールドの値と範囲のヒストグラムや、異なるテーブル内のフィールド間の相関情報など、詳細で包括的な統計が保持される傾向があります。
- 基本的な統計セットは、次の 3 つの関数で構成されます。 – B(T) は、テーブル T で使用されるブロックの数を返します。 – R(T) は、テーブル T 内のレコードの数を返します。 – V(T,F) は、T 内の異なる F 値の数を返します。

- 統計はカタログ テーブルに保存することも、データベースが再起動するたびに最初から計算することもできます。前者のオプションでは長い起動時間は回避されますが、トランザクションの実行が遅くなる可能性があります。
- インデックス メタデータには、各インデックスの名前、インデックスが付けられているテーブル、およびインデックスが付けられたフィールドに関する情報が保持されます。

7.8 推奨される読み物

SimpleDB で使用されるカタログ テーブルは可能な限り小さく、初期の INGRES システムで使用されていたものと似ています (Stonebraker 他、1976)。一方、Oracle は現在、非常に大規模なカタログを備えているため、それを説明する 60 ページの本が執筆されています (Kreines 2003)。

標準 SQL は、データベース メタデータへのアクセスを提供する標準的なビュー セットを定義します。これらのビューは、データベースの情報スキーマと呼ばれます。この章で説明されているメタデータを拡張する定義済みのビュー テーブルは 50 個以上あります。たとえば、トリガー、アサーション、制約、ユーザー定義型などの情報を表示するビューがあります。また、権限とロールに関する情報を保持するビューもいくつかあります。各データベース システムは、このメタデータを任意の方法で保存できますが、このメタデータへの標準インターフェイスを提供する必要があります。詳細については、Gulutzan と Pelzer (1999) の第 16 章を参照してください。

正確で詳細な統計メタデータは、適切なクエリ プランニングに不可欠です。この章で採用されているアプローチは大まかであり、商用システムではるかに高度な技術が使用されています。Gibbons ら (2002) の記事では、ヒストグラムの使用について説明し、頻繁な更新に対してヒストグラムを効率的に維持する方法を示しています。ヒストグラム情報はさまざまな方法で決定できますが、最も興味深い方法の 1 つはウェーブレット技術です (Matias ら 1998)。以前に実行したクエリの統計を収集して、関連するクエリのプランニングに使用することもできます (Bruno と Chaudhuri 2004)。

Bruno, N., & Chaudhuri, S. (2004). クエリ式の統計の条件付き選択性。ACM SIGMOD カンファレンスの議事録 (pp. 311 – 322)。Gibbons, P.、Matias, Y.、Poosala, V. (2002). 増分ヒストグラムの高速増分メンテナンス。ACM Transactions on Database Systems、27(3)、261 – 298。Gulutzan, P.、Pelzer, T. (1999). 本当に完全な SQL-99。Lawrence、KA: R&D Books。Kreines, D. (2003). Oracle データ辞書ポケットリファレンス。Sebastopol、CA: O’Reilly。Matias, Y.、Vitter, J.、Wang, M. (1998). 選択性推定のためのウェーブレットベースのヒストグラム。 ACM SIGMOD カンファレンスの議事録 (pp. 448 – 459)。Stonebraker, M.、Kreps, P.、Wong, E.、Held, G. (1976)。INGRES の設計と実装。ACM Transactions on Database Systems、1(3)、189 – 222。

7.9 演習

概念演習

7.1. SimpleDB が tblcat テーブルと fldcat テーブルに対して作成する tblcat レコードと fldcat レコードを指定します。(ヒント: TableMgr のコードを調べてください。) 7.2. トランザクション T1 が行うのはテーブル X の作成のみであり、トランザクション T2 が行うのはテーブル Y の作成のみであるとします。

(a) これらのトランザクションにはどのような同時スケジュールが考えられますか? (b) T1 と T2 がデッドロックする可能性はありますか? 説明してください。

7.3. 標準 SQL では、クライアントが既存のテーブルに新しいフィールドを追加することもできます。この機能を実装するための適切なアルゴリズムを教えてください。

プログラミング演習

7.4. 標準 SQL では、クライアントが既存のテーブルからフィールドを削除できます。この機能が TableMgr の removeField というメソッドに実装されているとします。

(a) このメソッドを実装する 1 つの方法は、fldcat のフィールドのレコードを変更して、フィールド名を空白にすることです。このメソッドのコードを書きなさい。 (b) パート (a) では、テーブルのレコードはどれも変更されません。削除されたフィールド値はどうなりますか? なぜアクセスできないのですか? (c) このメソッドを実装する別の方法は、fldcat からフィールドのレコードを削除し、テーブル内の既存のデータ レコードをすべて変更することです。これは (a) よりもかなり手間がかかります。それだけの価値があるのでしょうか? トレードオフを説明しなさい。

7.5. SimpleDB カタログ テーブルでは、tblcat のフィールド tblname がキーであり、fldcat のフィールド tblname が対応する外部キーです。これらのテーブルを実装する別の方法としては、tblcat に人工キー (たとえば、tblId) を使用し、fldcat に (たとえば、tableId という名前の) 対応する外部キーを使用する方法があります。

(a) この設計を SimpleDB に実装します。 (b) この設計は元の設計よりも優れていますか? (スペースを節約しますか? ブロック アクセスを節約しますか?)

7.6. 新しいデータベースのカタログ テーブルの作成中に SimpleDB がクラッシュしたとします。

(a) システムの再起動後にデータベースが回復されると何が起こるか説明してください。どのような問題が発生しますか? (b) この問題を修正するために SimpleDB コードを修正してください。

7.7. tblcat テーブルと fldcat テーブルを直接クエリして、次の各タスクを実行する SimpleDB クライアントを作成します。

(a) データベース内のすべてのテーブルの名前とフィールドを印刷します（たとえば、
(b) 特定のテーブルを作成するために使用されるSQL create table文のテキストを再構築して出力します（たとえば、「create table T (A integer, B varchar(7))」の形式）。

7.8. 存在しないテーブル名で getLayout メソッドが呼び出されるとどうなりますか? 代わりに null が返されるようにコードを修正してください。

7.9. クライアントがカタログにすでに存在するテーブルと同じ名前のテーブルを作成すると、どのような問題が発生する可能性がありますか? この問題が起きないようにコードを修正してください。 7.10. TableMgr を修正して、データベースからテーブルを削除するメソッド dropTable を追加します。ファイル マネージャも変更する必要がありますか?

7.11. 統計がカタログ テーブルに保存され、データベースが変更されるたびに更新されるように、SimpleDB コードを修正します。

7.12. SimpleDB コードを修正して、各テーブル T とフィールド F に対して V(T, F) が計算されるようにします。(ヒント: 各フィールドの数を追跡すると、異なる値の数が無制限になる可能性があるため、メモリを大量に消費する可能性があります。合理的なアイデアは、テーブルの一部の値をカウントして推定することです。たとえば、1000 個の異なる値を読み取るために必要なレコードの数をカウントします。)

7.13. クライアントがテーブルを作成し、そこにいくつかのレコードを挿入してから、ロールバックを実行するとします。

(a) カタログ内のテーブルのメタデータはどうなりますか? (b) データを含むファイルはどうなりますか? クライアントが後で同じ名前で異なるスキーマのテーブルを作成した場合に発生する可能性がある問題について説明します。 (c) この問題が解決されるように SimpleDB コードを修正します。 7.14. インデックス マネージャを変更して、カタログにインデックスの種類も保存します。クラス BTreeIndex と HashIndex の 2 種類のインデックスがあるとします。クラス コンストラクターと静的メソッド searchCost は、これらの各クラスで同じ引数を持ちます。 7.15. SimpleDB インデックス マネージャは、テーブル idxcat を使用してインデックス情報を保持します。別の設計方法としては、カタログ テーブル fldcat にインデックス情報を保持することが挙げられます。 (a) 2 つの方法を比較します。それぞれの利点は何ですか? (b) この代替方法を実装します。

第8章 クエリ処理

次の 3 つの章では、データベース エンジンが SQL クエリを実行する方法について説明します。問題は、SQL クエリでは返されるデータは指定されますが、その取得方法は指定されないことです。解決策は、エンジンがリレーショナル代数と呼ばれる一連のデータ取得演算子を実装することです。エンジンは SQL クエリをリレーショナル代数クエリに変換し、それを実行することができます。この章では、リレーショナル代数クエリとその実装について説明します。次の 2 つの章では、SQL からリレーショナル代数への変換について説明します。

8.1 リレーショナル代数

リレーショナル代数は、演算子のセットから構成されます。各演算子は、1 つ以上のテーブルを入力として受け取り、1 つの出力テーブルを生成するという 1 つの特殊なタスクを実行します。これらの演算子をさまざまな方法で組み合わせることで、複雑なクエリを構築できます。

SimpleDB バージョンの SQL は、次の 3 つの演算子を使用して実装できます。

- 出力テーブルは入力テーブルと同じ列を持ちますが、一部の行が削除されています。
- プロジェクトの出力テーブルには入力テーブルと同じ行が含まれますが、一部の列が削除されています。
- 製品の出力テーブルは、2つの入力テーブルからのレコードの可能なすべての組み合わせで構成されます。

これらの演算子については、次のサブセクションで説明します。

E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_8

8.1.1 選択

選択演算子は、入力テーブルと述語の 2 つの引数を取ります。出力テーブルは、述語を満たす入力レコードで構成されます。選択クエリは常に、入力テーブルと同じスキーマを持ちながらレコードのサブセットを含むテーブルを返します。

たとえば、クエリ Q1 は、2019 年に卒業した学生をリストしたテーブルを返します。

Q1 = 選択(学生、卒業年度=2019)

述語は、用語のブール値の組み合わせにすることができ、SQL の where 句に対応します。たとえば、クエリ Q2 は、2019 年に卒業し、専攻が部門 10 または 20 であった学生を検索します。

Q2 = select(STUDENT、GradYear=2019、(MajorId=10 または MajorId=20))

あるクエリの出力テーブルは、別のクエリの入力になることができます。たとえば、クエリ Q3 と Q4 はそれぞれ Q2 と同等です。

Q3 = 選択(選択 (学生、GradYear=2019)、MajorId=10 または Ma jorId=20)

Q4 = 選択(Q1、メジャーID=10 またはメジャーID=20)

Q3 では、最も外側のクエリの最初の引数は、2019 年に卒業した学生を検索する、Q1 と同じ別のクエリです。外側のクエリは、それらのレコードから、部門 10 または 20 の学生を取得します。クエリ Q4 も同様ですが、定義の代わりに Q1 の名前を使用します。

リレーショナル代数クエリは、クエリ ツリーとして図式的に表現できます。クエリ ツリーには、クエリで指定されている各テーブルと演算子のノードが含まれます。テーブル ノードはツリーのリーフであり、演算子ノードは非リーフです。演算子ノードには、入力テーブルごとに子ノードがあります。たとえば、Q3 のクエリ ツリーは図 8.1 に示されています。

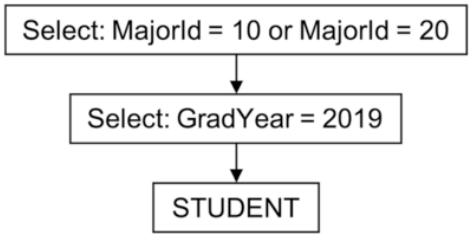


図8.1 Q3のクエリツリー

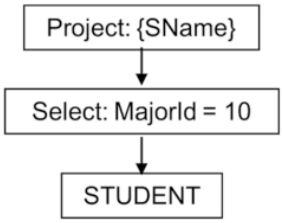


図8.2 Q6のクエリツリー

8.1.2 プロジェクト

プロジェクト演算子は、入力テーブルとフィールド名のセットの 2 つの引数を取ります。出力テーブルには入力テーブルと同じレコードが含まれますが、そのスキーマには指定されたフィールドのみが含まれます。たとえば、クエリ Q5 は、すべての学生の名前と卒業年を返します。

Q5 = プロジェクト(学生、{SName、卒業年度})

クエリは、プロジェクト演算子と選択演算子の両方で構成できます。クエリ Q6 は、学部 10 を専攻するすべての学生の名前をリストしたテーブルを返します。

Q6 = プロジェクト(選択(学生、専攻ID=10)、{SName})

Q6のクエリツリーを図8.2に示します。

プロジェクト クエリの出力テーブルには、重複したレコードが含まれる場合があります。たとえば、専攻が 10 で「pat」という名前の学生が 3 人いる場合、Q6 の出力には「pat」が 3 回含まれます。

すべての演算子の組み合わせが意味を持つわけではありません。たとえば、Q6 を反転して得られるクエリを考えてみましょう。

Q7 = select(project(STUDENT, {SName}), MajorId=10) // 違法です!

内部クエリの出力テーブルに選択する MajorId フィールドが含まれていないため、このクエリは意味がありません。

8.1.3 製品

選択演算子とプロジェクト演算子は、単一のテーブルに対して作用します。製品演算子を使用すると、複数のテーブルからの情報を結合して比較できます。この演算子は、2 つの入力テーブルを引数として受け取ります。出力テーブルは、入力テーブルのすべてのレコードの組み合わせで構成され、スキーマは、入力スキーマ内のフィールドの結合で構成されます。入力テーブルには、出力テーブルに同じ名前のフィールドが 2 つ存在しないように、フィールド名が重複していない必要があります。

Q8	SId	SName	MajorId	GradYear	DId	DName
	1	joe	10	2021	10	compsci
	1	joe	10	2021	20	math
	1	joe	10	2021	30	drama
	2	amy	20	2020	10	compsci
	2	amy	20	2020	20	math
	2	amy	20	2020	30	drama
	3	max	10	2022	10	compsci
	3	max	10	2022	20	math
	3	max	10	2022	30	drama
	4	sue	20	2022	10	compsci
	4	sue	20	2022	20	math
	4	sue	20	2022	30	drama
	5	bob	30	2020	10	compsci
	5	bob	30	2020	20	math
	5	bob	30	2020	30	drama
	6	kim	20	2020	10	compsci
	6	kim	20	2020	20	math
	6	kim	20	2020	30	drama
	7	art	30	2021	10	compsci
	7	art	30	2021	20	math
	7	art	30	2021	30	drama
	8	pat	20	2019	10	compsci
	8	pat	20	2019	20	math
	8	pat	20	2019	30	drama
	9	lee	10	2021	10	compsci
	9	lee	10	2021	20	math
	9	lee	10	2021	30	drama

図8.3 クエリQ8の出力

クエリ Q8 は、STUDENT テーブルと DEPT テーブルの積を返します。

Q8 = 製品(学生、学部)

図1.1の大学データベースには、STUDENTに9件のレコード、DEPTに3件のレコードが記録されています。図8.3は、これらの入力テーブルに対するQ8の出力を示しています。

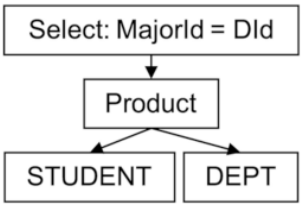


図8.4 Q9のクエリツリー

出力テーブルには、学生レコードと学部レコードのペアごとに 1 つのレコード、合計 27 のレコードが含まれます。一般に、STUDENT に N 個のレコードがあり、DEPT に M 個のレコードがある場合、出力テーブルには N !M 個のレコードが含まれます (ちなみに、これが演算子が「product」と呼ばれる理由です)。

クエリQ8は、各学生の専攻を考慮していないため、特に意味がありません。この意味は、クエリQ9と図8.4に示すように、選択述語で表現できます。

Q9 = select(product(学生、学部)、専攻ID=Did)

このクエリの出力テーブルには、述語を満たす STUDENT と DEPT のレコードの組み合わせのみが含まれます。したがって、27 通りの組み合わせのうち、残るのは学生の専攻 ID が学部の ID と同じ組み合わせだけです。つまり、結果テーブルは学生とその専攻学部で構成されます。出力テーブルには、27 レコードではなく 9 レコードが含まれます。

8.2 スキャン

スキャンは、リレーショナル代数クエリの出力を表すオブジェクトです。SimpleDB のスキャンは、インターフェイス Scan を実装します。図 8.5 を参照してください。Scan メソッドは TableScan メソッドのサブセットであり、同じ動作をします。この対応は驚くべきことではありません。クエリの出力はテーブルなので、クエリとテーブルに同じ方法でアクセスするのは当然です。

例として、図 8.6 の printNameAndGradYear メソッドを考えてみましょう。このメソッドはスキャンを反復し、各レコードのフィールド sname と gradyear の値を出力します。

パブリック インターフェイス Scan { public void beforeFirst()
; public boolean next(); public int getInt(String fldname); public S
tring getString(String fldname); public Constant getVal(String fld
name); public boolean hasField(String fldname); public void clos
e();

}

図8.5 SimpleDBスキャンインターフェース

この例のポイントは、このメソッドはスキャンがどのクエリ (またはテーブル) を表しているかをまったく知らないということです。スキャンはSTUDENT テーブルを表している可能性があり、特定の専攻の学生を選択するクエリや、アインシュタイン教授のコースを受講した学生を表す可能性もあります。唯一の要件は、スキャンの出力テーブルに学生名と卒業年が含まれていることです。

Scan オブジェクトは、クエリ ツリーのノードに対応します。SimpleDB には、各リレーショナル演算子の Scan クラスが含まれています。これらのクラスのオブジェクトはクエリ ツリーの内部ノードを構成し、TableScan オブジェクトはツリーのリーフを表します。図 8.7 は、テーブルのスキャン コンストラクターと、SimpleDB でサポートされている 3 つの基本演算子を示しています。

SelectScan コンストラクターは、基礎となるスキャンと述語の 2 つの引数を取ります。基礎となるスキャンは、選択演算子への入力です。Scan はインターフェイスであるため、SelectScan オブジェクトは、その入力が保存されたテーブルであるか、別のクエリの出力であるかを認識しません。この状況は、リレーショナル演算子への入力が任意のテーブルまたはクエリである可能性があるという事実に対応しています。

SelectScan コンストラクタに渡される選択述語は Predicate 型です。セクション 8.6 では、SimpleDB が述語を処理する方法の詳細について説明します。それまでは、この問題についてはややあいまいなままにしておきます。

クエリ ツリーは、スキャンを合成することで構築されます。ツリーの各ノードに対してスキャンが行われます。たとえば、図 8.8 は、図 8.2 のクエリ ツリーの SimpleDB コードを示しています (選択述語の詳細は省略)。スキャン変数 s1、s2、s3 はそれぞれ、クエリ ツリーのノードに対応しています。ツリーはボトムアップで構築されます。最初にテーブル スキャンが作成され、次に選択スキャン、最後にプロジェクト スキャンが作成されます。変数 s3 は最終的なクエリ ツリーを保持します。while ループは s3 を走査し、各学生の名前を出力します。

図8.9は、図8.4のクエリツリーに対応するSimpleDBコードを示しています。クエリツリーには4つのノードがあるため、コードには4つのスキャンが含まれています。変数s4は最終的なクエリツリーを保持します。whileループが前のコードとほぼ同じであることに注意してください。

```
public static void printNameAndGradyear(Scan s) { s.beforeFirst(); whil
e (s.next()) { String sname = s.getString("sname"); String gradyr = s.getI
nt("gradyear"); System.out.println(sname + "\t" + gradyr); }s.close();




}
```

図8.6 スキャンした記録の名前と卒業年度の印刷

Scan

public TableScan(トランザクション tx、文字列ファイル名、レイアウトレイアウト); pub
lic SelectScan(スキャン s、述語 pred); public ProjectScan(スキャン s、リスト<文字列> fld
list); public ProductScan(スキャン s1、スキャン s2);

図8.7 Scanを実装するSimpleDBコンストラクタのAPI

```
トランザクション tx = db.newTx(); MetadataMgr
mdm = db.MetadataMgr();

// STUDENT ノード レイアウト layout = mdm.getLayout("student
", tx); スキャン s1 = new TableScan(tx, "student", layout);


// 選択ノード Predicate pred = new Predicate(. . .); // majorid=10 Scan s2 =
new SelectScan(s1, pred);


// プロジェクト ノード List<String> c = Arrays.asList("s
name"); Scan s3 = new ProjectScan(s2, c);


s3.next() が s3.close() の間に、System.out.println(s3.getString("s
name"));
```

図8.8 図8.2をスキャンとして表す

```
トランザクション tx = db.newTx(); MetadataMgr
mdm = db.MetadataMgr();

// STUDENT ノード レイアウト layout1 = mdm.getLayout("student"
, tx); スキャン s1 = new TableScan(tx, "student", layout1);


// DEPT ノード レイアウト layout2 = mdm.getLayout("dept", tx
); スキャン s2 = new TableScan(tx, "dept", layout2);


// 製品ノード Scan s3 = new ProductScan(s1, s2)
;

// Select ノード Predicate pred = new Predicate(. . .); //majorid=did Scan s4
= new SelectScan(s3, pred);


while (s4.next()) System.out.println(s4.getString("sname") + ", " + s4.getStrin
g("gradyear") + ", " + s4.getString("dname") );


s4.close();
```

図8.9 図8.4をスキャンとして表す

スペースを節約するために、ループは各出力レコードに対して 3 つのフィールド値のみを出力しますが、6 つのフィールド値すべてを含めるように簡単に変更できます。

最後に、close メソッドはクエリ ツリーの最も外側のスキャンでのみ呼び出されることに注意してください。スキャンを閉じると、その基礎となるスキャンも自動的に閉じられます。

8.3 アップデートスキャン

クエリは仮想テーブルを定義します。Scan インターフェイスには、クライアントがこの仮想テーブルから読み取りはできるが更新はできないメソッドがあります。すべてのスキャンを意味のある形で更新できるわけではありません。スキャン内のすべての出力レコード r が、基になるデータベース テーブル内に対応するレコード r′ を持つ場合、スキャンは更新可能です。この場合、r への更新は r′ への更新として定義されます。

更新可能なスキャンは、UpdateScan インターフェイスをサポートします。図 8.10 を参照してください。インターフェイスの最初の 5 つのメソッドは、基本的な変更操作です。他の 2 つのメソッドは、スキャンの現在のレコードの基礎となる保存されたレコードの識別子を使用します。getRid メソッドはこの識別子を返し、moveToRid はスキャンを指定された保存されたレコードに配置します。

SimpleDB で UpdateScan を実装するクラスは、TableScan と SelectScan の 2 つだけです。これらの使用例として、図 8.11 を検討してください。パート (a) は、セクション 53 を受講したすべての学生の成績を変更する SQL ステートメントを示し、パート (b) はこのステートメントを実装するコードを示しています。このコードは、まずセクション 53 のすべての登録レコードの選択スキャンを作成し、次にスキャンを反復して各レコードの成績を変更します。

変数 s2 はメソッド setString を呼び出すため、更新スキャンとして宣言する必要があります。一方、SelectScan コンストラクターの最初の引数はスキャンであるため、更新スキャンとして宣言する必要はありません。代わりに、s2 の setString メソッドのコードは、その基礎となるスキャン (つまり、s1) を更新スキャンにキャストします。そのスキャンが更新可能でない場合は、ClassCastException がスローされます。

```
public interface UpdateScan extends Scan {
  public void setInt(String fldname, int val);
  public void setString(String fldname, String val);
  public void setVal(String fldname, Constant val);
  public void insert();
  public void delete();

  public RID  getRid();
  public void moveToRid(RID rid);
}
```

図8.10 SimpleDB UpdateScanインターフェース

ENROLLを更新し、Grade =を'C'に設定し、Secti
onId =を53に設定する

(a)

トランザクション tx = db.newTx(); MetadataMgr mdm = db.MetadataMgr();
レイアウト layout = mdm.getLayout("enroll", tx); スキャン s1 = new TableS
can(tx, "enroll", layout); 述語 pred = new Predicate(. . .); // SectionId=53 Updat
eScan s2 = new SelectScan(s1, pred); while (s2.next()) s2.setString("grade", "C"
); s2.close(); (b)

図8.11 SQL更新文を更新スキャンとして表現する。(a)セクション53の学生の成績を変更するSQL文、(b)その文に対応するSimpleDBコード

8.4 スキャンの実装

SimpleDB エンジンには、TableScan クラスと、演算子 select、project、product のクラスの 4 つの Scan クラスが含まれています。第 6 章では TableScan について説明しました。次のサブセクションでは、3 つの演算子クラスについて説明します。

8.4.1 スキャンの選択

SelectScan のコードは図 8.12 に示されています。コンストラクタは、その基になる入力テーブルのスキャンを保持します。スキャンの現在のレコードは、その基になるスキャンの現在のレコードと同じです。つまり、ほとんどのメソッドは、そのスキャンの対応するメソッドを呼び出すだけで実装できます。

唯一の重要なメソッドは next です。このメソッドの役割は、新しい現在のレコードを確立することです。コードは、基礎となるスキャンをループし、述語を満たすレコードを探します。そのようなレコードが見つかった場合、それが現在のレコードになり、メソッドは true を返します。そのようなレコードがない場合は、while ループが完了し、メソッドは false を返します。

選択スキャンは更新可能です。UpdateScan メソッドは、基礎となるスキャンも更新可能であることを前提としています。特に、基礎となるスキャンを、ClassCastException を発生させずに UpdateScan にキャストできることを前提としています。SimpleDB 更新プランナーによって作成されるスキャンには、テーブル スキャンと選択スキャンのみが含まれるため、このような例外は発生しないはずです。

```
プblic クラス SelectScan は UpdateScan を実装します { private Sca
    n s; private Predicate pred; public SelectScan(Scan s, Predicate pred
    ) { this.s = s; this.pred = pred; }// Scan メソッド public void before
    First() { s.beforeFirst(); }public boolean next() { while (s.next()) if (
    pred.isSatisfied(s)) return true; return false; }public int getInt(String
    fldname) { return s.getInt(fldname); }public String getString(String
    fldname) { return s.getString(fldname); }public Constant getVal(Stri
    ng fldname) { return s.getVal(fldname); }public boolean hasField(St
    ring fldname) { return s.hasField(fldname); }public void close() { s.
    close(); }// UpdateScan メソッド public void setInt(String fldname,
    int val) { UpdateScan us = (UpdateScan) s; us.setInt(fldname, val);
    }
```

図8.12 SimpleDBクラスSelectScanのコード

```
public void setString(String fldname, String val) { UpdateScan us = (Updat
eScan) s; us.setString(fldname, val); }public void setVal(String fldname, C
onstant val) { UpdateScan us = (UpdateScan) s; us.setVal(fldname, val); }

public void delete() { UpdateScan us = (UpdateSc
an) s; us.delete(); }public void insert() { UpdateSc
an us = (UpdateScan) s; us.insert(); }public RID g
etRid() { UpdateScan us = (UpdateScan) s; return
us.getRid(); }public void moveToRid(RID rid) {
UpdateScan us = (UpdateScan) s; us.moveToRid(r
id); }

}
```

図8.12（続き）

8.4.2 プロジェクトスキャン

ProjectScan のコードは図 8.13 に示されています。出力フィールドのリストはコンストラクタに渡され、hasField メソッドを実装するために使用されます。他のメソッドは、その要求を基盤となるスキャンの対応するメソッドに転送するだけです。getVal、getInt、および getString メソッドは、指定されたフィールド名がフィールド リストにあるかどうかを確認します。ない場合は、例外が生成されます。

投影は更新可能ですが、クラス ProjectScan は UpdateScan を実装していません。演習 8.12 では実装を完了するように求められます。

8.4.3 製品スキャン

ProductScanのコードは図8.14に示されています。製品スキャンは、基礎となるスキャンs1からのレコードの可能なすべての組み合わせを反復処理できる必要があります。

```
public class ProjectScan implements Scan { private Scan s; private Collection<Stri
ng> fieldlist; public ProjectScan(Scan s, List<String> fieldlist) { this.s = s; this.fiel
dlist = fieldlist; }public void beforeFirst() { s.beforeFirst(); }public boolean next()
{ return s.next(); }public int getInt(String fldname) { if (hasField(fldname)) return
s.getInt(fldname); elsethrow new RuntimeException("field not found."); }public St
ring getString(String fldname) { if (hasField(fldname)) return s.getString(fldname);
elsethrow new RuntimeException("field not found."); }public Constant getVal(Stri
ng fldname) { if (hasField(fldname)) return s.getVal(fldname); elsethrow new Runt
imeException("フィールドが見つかりません。"); }public boolean hasField(Stri
ng fldname) { return fieldlist.contains(fldname); }public void close() { s.close(); }
}
```

図8.13 SimpleDBクラスProjectScanのコード

```
pパブリッククラス ProductScan は Scan を実装します { private Scan s1, s2; publ
  ic ProductScan(Scan s1, Scan s2) { this.s1 = s1; this.s2 = s2; s1.next(); }public void
  beforeFirst() { s1.beforeFirst(); s1.next(); s2.beforeFirst(); }public boolean next() { i
  f (s2.next()) return true; else { s2.beforeFirst(); return s2.next() && s1.next(); } }pub
  lic int getInt(String fldname) { if (s1.hasField(fldname)) return s1.getInt(fldname); e
  lsereturn s2.getInt(fldname); }public String getString(String fldname) { if (s1.hasFie
  ld(fldname)) return s1.getString(fldname); elsereturn s2.getString(fldname); }public
  Constant getVal(String fldname) { if (s1.hasField(fldname)) return s1.getVal(fldnam
  e); elsereturn s2.getVal(fldname); }public boolean hasField(String fldname) { return
  s1.hasField(fldname) || s2.hasField(fldname); }public void close() { s1.close(); s2.
  close(); }
```

}

図8.14 SimpleDBクラスProductScのコード の

そして s2 です。これは、s1 の最初のレコードから開始して s2 の各レコードを反復処理し、次に s1 の 2 番目のレコードに移動して s2 を反復処理する、というように行われます。概念的には、外側のループで s1 を反復処理し、内側のループで s2 を反復処理するネストされたループのようなものです。

メソッド next は、この「ネストされたループ」の考え方を次のように実装します。 next を呼び出すたびに、s2 の次のレコードに移動します。s2 にそのようなレコードがある場合は、true を返すことができます。そうでない場合は、s2 の反復が完了しているため、メソッドは s1 の次のレコードと s2 の最初のレコードに移動します。これが可能な場合は true を返し、s1 のレコードがもうない場合は、スキャンが完了し、next は false を返します。

getVal、getInt、および getString メソッドは、適切な基礎スキャンのフィールドに単純にアクセスします。各メソッドは、指定されたフィールドがスキャン s1 内にあるかどうかを確認します。ある場合は、s1 を使用してフィールドにアクセスし、そうでない場合は、s2 を使用してフィールドにアクセスします。

8.5 パイプラインクエリ処理

これら 3 つの関係代数演算子の実装には、共通する 2 つの特性があります。

- 必要に応じて、出力レコードを 1 つずつ生成します。
- 出力レコードは保存されず、中間計算も保存されません。

このような実装はパイプライン化と呼ばれます。このセクションでは、パイプライン化実装とそのプロパティを分析します。

TableScan オブジェクトについて考えてみましょう。このオブジェクトはレコード ページを保持し、レコード ページにはバッファーが保持され、バッファーには現在のレコードを含むページが保持されます。現在のレコードは、そのページ内の単なる位置です。レコードをページから削除する必要はありません。クライアントがフィールドの値を要求した場合、レコード マネージャーは単にその値をページから抽出し、クライアントに送り返します。next を呼び出すたびに、テーブル スキャンは次のレコードに配置され、異なるレコード ページが保持される可能性があります。その next メソッドの各呼び出しは、基礎となるスキャンの現在のレコードが述語を満たすまで、基礎となるスキャンの next を繰り返し呼び出します。ただし、もちろん、実際の「現在のレコード」は存在しません。基礎となるスキャンがテーブル スキャンである場合、現在のレコードはテーブル スキャンによって保持されるページ内の単なる場所になります。また、基礎となるスキャンが別の種類のスキャン (図 8.4 および 8.9 の製品スキャンなど) である場合、現在のレコードの値は、そのノードのサブツリーにあるテーブル スキャンの現在のレコードから決定されます。

パイプライン スキャンが next への別の呼び出しを処理するたびに、中断したところから検索を開始します。その結果、スキャンは、次の出力レコードを決定するために、基礎となるスキャンから必要な数のレコードのみを要求します。

パイプライン スキャンでは、選択したレコードは追跡されません。そのため、クライアントがレコードを再度要求した場合、スキャンは検索全体を最初からやり直す必要があります。

SelectScan オブジェクトについて考えてみましょう。

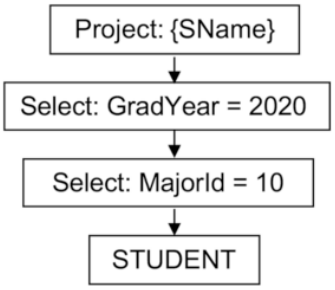


図8.15 複数の選択ノードを含むクエリツリー

「パイプライン」という用語は、クエリ ツリーを下るメソッド呼び出しのフローと、ツリーを上る結果値のフローを指します。たとえば、メソッド getInt の呼び出しについて考えてみましょう。ツリー内の各ノードは、リーフ ノードに到達するまで、その呼び出しを子ノードの 1 つに渡します。そのリーフ ノード (テーブル スキャン) は、そのページから目的の値を抽出し、その値をツリーの上に戻します。または、メソッド next の呼び出しについて考えてみましょう。各ノードは、子ノードに次のレコードの内容が含まれていることが確認されるまで、子ノードに対して 1 回以上の next (および製品ノードの場合は beforeFirst も) 呼び出しを行います。その後、親ノードに成功 (または、そのようなレコードが存在しない場合は失敗) を返します。

パイプライン実装は非常に効率的です。たとえば、図 8.15 のクエリ ツリーを考えてみましょう。これは、2020 年に専攻 10 で卒業する学生の名前を取得します。

このツリーのプロジェクト ノードと選択ノードは、テーブル スキャンに必要なブロック アクセス以外に、STUDENT テーブルへの追加のブロック アクセスを発生しません。その理由を理解するには、まずプロジェクト ノードについて考えます。そのノードでの next の各呼び出しは、その子ノードで next を呼び出し、そのノードの戻り値を返すだけです。つまり、プロジェクト ノードは、クエリの残りの部分で実行されるブロック アクセスの数を変更しません。

次に、選択ノードについて考えてみましょう。外側の選択ノードで next を呼び出すと、内側の選択ノードで next が呼び出されます。内側のノードは、現在のレコードが述語「MajorId ¼ 10」を満たすまで、子ノードで next を繰り返し呼び出します。次に、内側の選択ノードは true を返し、外側の選択ノードは現在のレコードを調べます。卒業年度が 2020 年でない場合、外側のノードは内側のノードで next を再度呼び出し、別の現在のレコードを待機します。外側の選択ノードが true を返す唯一の方法は、そのレコードが両方の述語を満たす場合です。このプロセスは、外側のノードが next を呼び出すたびに継続され、基になるテーブル スキャンは、両方の述語が満たされるまで、次のレコードに継続的に移動します。テーブル スキャンで STUDENT レコードがもう存在しないことが認識されると、next メソッドは false を返し、false の値がツリーに伝播します。つまり、STUDENT は 1 回だけスキャンされ、クエリがテーブル スキャンのみを実行した場合とまったく同じです。したがって、このクエリ内の選択ノードはコストがかかりません。

パイプライン実装はこれらのケースでは非常に効率的ですが、それほど効果的でないケースもあります。そのようなケースの 1 つは、選択ノードが製品ノードの右側にあり、複数回実行される場合です。選択を何度も実行する代わりに、出力レコードをマテリアライズして一時テーブルに格納する実装を使用する方がよい場合があります。このような実装については、第 13 章で説明します。

8.6 述語

述語は、特定のスキャンの各行に対して true または false を返す条件を指定します。条件が true を返す場合、その行は述語を満たしていると見なされます。SQL 述語は次のように構成されます。

- 述語は、用語または用語のブール組み合わせです。
- 用語は 2 つの表現を比較したものです。
- 式は、定数とフィールド名に対する演算で構成されます。
- 定数は、整数や文字列など、事前に定義された一連の型から得られる値です。

たとえば、標準 SQL の次の述語を考えてみましょう。

(GradYear>2021 または MOD(GradYear,4)=0) および MajorId=DId

この述語は 3 つの項 (太字で表示) で構成されています。最初の 2 つの項はフィールド名 GradYear (または GradYear の関数) を定数と比較し、3 番目の項は 2 つのフィールド名を比較します。各項には 2 つの式が含まれます。たとえば、2 番目の項には式 MOD(GradYear,4) と 0 が含まれます。

SimpleDB では、使用できる定数、式、項、述語が大幅に簡素化されています。SimpleDB 定数は整数または文字列のみ、式は定数またはフィールド名のみ、項は式を等価性のみで比較でき、述語は項の結合のみを作成できます。演習 8.7 〜 8.9 では、SimpleDB 述語を拡張して表現力を高めるように求められます。

次の述語を考えてみましょう。

SName = 'joe' および MajorId = DId

図 8.16 のコード フラグメントは、SimpleDB でこの述語を作成する方法を示しています。述語が、定数と式から始まり、次に項、最後に述語と、内側から外側へと作成されることに注意してください。

図 8.17 は Constant クラスのコードを示しています。各 Constant オブジェクトには、Integer 変数と String 変数が含まれています。どのコンストラクタが呼び出されたかに応じて、これらの変数のうち 1 つだけが null 以外になります。equals、compareTo、hasCode、および toString メソッドは、null 以外の変数を使用します。

Expression クラスのコードは図 8.18 に示されています。このクラスには 2 つのコンストラクタがあり、1 つは定数式用、もう 1 つはフィールド名式用です。各コンストラクタは、関連付けられた変数に値を割り当てます。isFieldName メソッドは、式がフィールド名を表すかどうかを判断する便利な方法を提供します。evaluate メソッドは、スキャンの現在の出力レコードに対する式の値を返します。式が定数の場合、スキャンは無関係であり、メソッドは単に定数を返します。式がフィールドの場合、

```
式 lhs1 = 新しい式("SName"); 定数 c = 新しい定数("joe");
式 rhs1 = 新しい式(c); 項 t1 = 新しい項(lhs1, rhs1);


式 lhs2 = 新しい式("MajorId"); 式 rhs2 = 新しい式("DId"); 項
t2 = 新しい項(lhs2, rhs2);


述語 pred1 = 新しい Predicate(t1); 述語 pred2 = 新
しい Predicate(t2); pred1.conjoinWith(pred2);
```

図8.16 述語を作成するためのSimpleDBコード

メソッドはスキャンからフィールドの値を返します。applysTo メソッドは、クエリ プランナーによって式のスコープを決定するために使用されます。 SimpleDB の用語は、図 8.19 にコードを示す Term インターフェイスによって実装されます。そのコンストラクタは、左側の式と右側の式を表す 2 つの引数を取ります。最も重要なメソッドは isSatisfied で、指定されたスキャンで両方の式が同じ値に評価された場合に true を返します。残りのメソッドは、クエリ プランナーが用語の効果と範囲を決定するのに役立ちます。たとえば、メソッド reductionFactor は、述語を満たすレコードの予想数を決定し、第 10 章で詳しく説明します。メソッド equatesWithConstant と equatesWithField は、クエリ プランナーがインデックスを使用するタイミングを決定するのに役立ちます。第 15 章で説明します。

Predicate クラスのコードは、図 8.20 に示されています。述語は項のリストとして実装され、述語は各項に対応するメソッドを呼び出すことによってメソッドに応答します。クラスには 2 つのコンストラクタがあります。1 つのコンストラクタには引数がなく、項のない述語を作成します。このような述語は常に満たされ、述語 true に対応します。もう 1 つのコンストラクタは、単一の項を持つ述語を作成します。メソッド conjoinWith は、引数の述語の項を指定された述語に追加します。

8.7 章の要約

- リレーショナル代数クエリは演算子で構成されます。各演算子は 1 つの特殊なタスクを実行します。クエリ内の演算子の構成は、クエリ ツリーとして記述できます。
- この章では、SimpleDB バージョンの SQL を理解して変換するのに役立つ 3 つの演算子について説明します。これらは次のとおりです。

  – 出力テーブルは入力テーブルと同じ列を持ちますが、一部の行が削除されています。

public クラス Constant は Comparable<Constant> を実装します { private Integer ival = null ; private String sval = null; public Constant(Integer ival) { this.ival = ival; }public Constant(Str ing sval) { this.sval = sval; }public int asInt() { return ival; }public String asString() { return sv al; }public boolean equals(Object obj) { Constant c = (Constant) obj; return (ival != null) ? ival .equals(c.ival) : sval.equals(c.sval); }public int compareTo(Constant c) { return (ival!=null) ? i val.compareTo(c.ival) : sval.compareTo(c.sval); }public int hashCode() { return (ival != null) ? ival.hashCode() : sval.hashCode(); }public String toString() { return (ival != null) ? ival.toStrin g() : sval.toString(); } }

図8.17 Constantクラス

– プロジェクト: 出力テーブルには入力テーブルと同じ行が含まれますが、一部の列は削除されています – 製品: 出力テーブルには、2 つの入力テーブルからのレコードのすべての可能な組み合わせが含まれます

- スキャンは、リレーショナル代数クエリツリーを表すオブジェクトです。各リレーショナル演算子には、スキャンインターフェースを実装する対応するクラスがあります。オブジェクト

```
public class Expression { private Constant val = null; private String fldname = null; p
ublic Expression(Constant val) { this.val = val; } public Expression(String fldname) {
this.fldname = fldname; }public boolean isFieldName() { return fldname != null; }pu
blic Constant asConstant() { return val; }public String asFieldName() { return fldnam
e; }public Constant assess(Scan s) { return (val != null) ? val : s.getVal(fldname); }pu
blic boolean appliedTo(Schema sch) { return (val != null) ? true : sch.hasField(fldnam
e); }public String toString() { return (val != null) ? val.toString() : fldname; } }
```

図8.18 クラス式

これらのクラスはクエリ ツリーの内部ノードを構成します。また、テーブル用のスキャン クラスもあり、そのオブジェクトはツリーのリーフを構成します。

- Scan インターフェースは基本的に TableScan と同じです。クライアントはスキャンを反復し、出力レコード間を移動してフィールド値を取得します。スキャンは、レコード ファイルを適切に移動して値を比較することにより、クエリの実装を管理します。
- スキャン内のすべてのレコード r が、基礎となるデータベース テーブル内に対応するレコード r′ を持つ場合、スキャンは更新可能です。この場合、仮想レコード r の更新は、格納されたレコード r′ の更新として定義されます。
- 各スキャン クラスのメソッドは、その演算子の意図を実装します。例:

プ blic クラス Term { public Term(Expression lhs, Expression rhs) { this.lhs = lhs; this.rhs = rhs; }public boolean isSatisfied(Scan s) { Constant lhsval = lhs.evaluate(s); Constant rhsval = rhs.evaluate(s); return rhsval.equals(lhsval); }public boolean appliedTo(Schema sch) { return lhs.appliesTo(sch) && rhs.appliesTo(sch); } public int reductionFactor(Plan p) { String lhsName, rhsName; if (lhs.isFieldName() && rhs.isFieldName()) { lhsName = lhs.asFieldName(); rhsName = rhs.asFieldName(); return Math.max(p.distinctValues(lhsName), p.distinctValues(rhsName)); }if (lhs.isFieldName()) { lhsName = lhs.asFieldName(); return p.distinctValues(lhsName); }if (rhs.isFieldName()) { rhsName = rhs.asFieldName(); return p.distinctValues(rhsName); }// それ以外の場合、項は定数と等しくなります if (lhs.asConstant().equals(rhs.asConstant())) return 1; elsereturn Integer.MAX_VALUE; }public Constant equatesWithConstant(String fldname) { if (lhs.isFieldName() && lhs.asFieldName().equals(fldname) && !rhs.isFieldName()) return rhs.asConstant();そうでない場合は、(rhs.isFieldName() && rhs.asFieldName().equals(fldname) && !lhs.isFieldName()) が lhs.asConstant() を返し、そうでない場合は null を返します。

プライベート式 lhs、rhs;

}

図8.19 SimpleDBクラスのコード 用語

パブリック文字列equatesWithField(文字列fldname) { if (lhs.isField Name() && lhs.asFieldName().equals(fldname) && rhs.isFieldName()) return rhs.asFieldName();

そうでない場合は、(rhs.isFieldName() && rhs.asFieldName().e quals(fldname) && lhs.isFieldName()) が lhs.asFieldName() を返す 場合、そうでない場合は null を返す場合。}public String toString( ) { lhs.toString() を返す場合 + "=" + rhs.toString(); }

}

図8.19（続き）

– 選択スキャンは、基礎となるスキャン内の各レコードをチェックし 、述語を満たすレコードのみを返します。 – 製品スキャンは、基礎と なる 2 つのスキャンからのレコードのすべての組み合わせに対してレコ ードを返します。 – テーブル スキャンは、指定されたテーブルのレコード ファイルを開き、必要に応じてバッファーを固定し、ロックを取得します。

- これらのスキャン実装は、パイプライン実装と呼ばれます。パイプライン実装では、データの先読み、キャッシュ、ソート、その他の前処理は行われません。
- パイプライン実装では、出力レコードは作成されません。クエリ ツリーの各リーフはテーブル スキャンであり、そのテーブルの現在のレコードを保持するバッファーが含まれています。操作の「現在のレコード」は、各バッファーのレコードから決定されます。フィールド値を取得する要求は、ツリーの適切なテーブル スキャンに送られ、結果はテーブル スキャンからルートまで返されます。
- パイプライン実装を使用するスキャンは、必要に応じて動作します。各スキャンは、次のレコードを決定するために必要な数のレコードのみを子から要求します。

8.8 推奨される読み物

リレーショナル代数は、ほぼすべてのデータベース入門書で定義されていますが、各テキストは独自の構文を持つ傾向があります。リレーショナル代数とその表現力の詳細な説明は、Atzeni と DeAntonellis (1992) にあります。この本では、述語論理に基づくクエリ言語であるリレーショナル計算も紹介されています。リレーショナル計算の興味深い点は、次のように拡張できることです。

プblic クラス Predicate { private List<Term> term = new ArrayList<Term > ( ) ; public Predicate() {} public Predicate(Term t) { terms.add(t); }publi c void conjoinWith(Predicate pred) { terms.addAll(pred.terms); }public b oolean isSatisfied(Scan s) { for (Term t : terms) if (!t.isSatisfied(s)) return false; return true; }public int reductionFactor(Plan p) { int factor = 1; for (Term t : terms) factor *= t.reductionFactor(p); return factor; }

パブリック述語 selectSubPred(Schema sch) { 述語結果 = 新 しい Predicate(); for (Term t : terms) if (t.appliesTo(sch)) result.t erms.add(t); if (result.terms.size() == 0) null を返す; else return result; }

パブリック述語 joinSubPred(スキーマ sch1、スキーマ sch2 ）{
述語結果 = new Predicate(); スキーマ newsch = n ew Schema(); newsch.addAll(sch1); newsch.addAll (sch2); for (Term t : terms) if (!t.appliesTo(sch1) & & !t.appliesTo(sch2) && t.appliesTo(newsch)) resu lt.terms.add(t); if (result.terms.size() == 0) return n ull;

図8.20 SimpleDBクラスPredicateのコード

```
パブリック定数equatesWithConstant(String fldname) { for (Term t : terms) {
定数c = t.equatesWithConstant(fldname); if (c != null) return c; elsereturn res
ult; }


    }return null; }public String equatesWithField(String fldname) { for
(Term t : terms) { String s = t.equatesWithField(fldname); if (s != null) r
eturn s; }return null; }public String toString() { Iterator<Term> iter = ter
ms.iterator(); if (!iter.hasNext()) return ""; String result = iter.next().toStr
ing(); while (iter.hasNext()) result += " and " + iter.next().toString(); ret
urn result; }
```

}

図8.20（続き）

再帰クエリ (つまり、クエリ定義で出力テーブルも指定されているクエリ) を許可します。再帰リレーショナル計算はデータログと呼ばれ、Prolog プログラミング言語に関連しています。データログとその表現力については、Atzeni と DeAntonellis (1992) でも説明されています。

パイプライン化されたクエリ処理のトピックは、クエリ処理パズルの小さなピースであり、後の章のトピックも含まれています。記事 Graefe (1993) には、クエリ処理テクニックに関する包括的な情報が含まれています。セクション 1 では、スキャンとパイプライン化された処理について詳しく説明されています。記事 Chaudhuri (1998) では、統計収集と最適化に加えて、クエリ ツリーについて説明しています。

Atzeni, P., & DeAntonellis, V. (1992). リレーショナル データベース理論。アッパー サドル リバー、ニュージャージー: Prentice-Hall。

Chaudhuri, S. (1998). リレーショナル システムにおけるクエリ最適化の概要。ACM Principles of Database Systems Conference の議事録 (pp. 34 – 43)。Graefe, G. (1993). 大規模データベースのクエリ評価手法。ACM Computing Surveys、25(2)、73 – 170。

8.9 演習

概念演習

8.1. いずれかの入力が空の場合、積演算の出力は何ですか? 8.2. 図 8.9 をテンプレートとして使用して、次のクエリをスキャンとして実装します。

STUDENT、DEPT、ENROLL、SECTION から sname、dname、grade を選択します。ここで、SId=StudentId、SectId=SectionId、DId=MajorId、YearOffered=2020 です。

8.3. 図8.9のコードを考えてみます。

(a) このコードを実行するためにトランザクションが取得する必要があるロックは何ですか? (b) これらのロックごとに、コードがそのロックを待機するシナリオを示します。

8.4. ProductScan のコードを検討します。

(a) 最初の基礎スキャンにレコードがない場合、どのような問題が発生する可能性がありますか? コードをどのように修正する必要がありますか? (b) 2 番目の基礎スキャンにレコードがない場合、問題が発生しない理由を説明してください。

8.5. STUDENT とそれ自身との積をとって、すべての学生のペアを検索するとします。

(a) 1 つの方法は、次のコード フラグメントのように、STUDENT でテーブル スキャンを作成し、それを製品内で 2 回使用することです。

```
レイアウト layout = mdm.getLayout("student", tx); スキャン s
1 = new TableScan(tx, "student", layout); スキャン s2 = new Pr
oductScan(s1, s1);
```

スキャンの実行時に、これが誤った（そして奇妙な）動作を引き起こす理由を説明します。

(b) より良い方法は、STUDENT に対して 2 つの異なるテーブル スキャンを作成し、それらに対して製品スキャンを作成することです。これにより、STUDENT レコードのすべての組み合わせが返されますが、問題があります。それは何でしょうか。

プログラミング演習

8.6. ProjectScan の getVal、getInt、getString メソッドは、引数フィールド名が有効かどうかをチェックします。他のスキャン クラスは、これを行いません。他のスキャン クラスごとに、(a) 無効なフィールドでこれらのメソッドが呼び出された場合に、どのような問題 (およびどのメソッド) が発生するかを教えてください。(b) 適切な例外がスローされるように SimpleDB コードを修正してください。 8.7. 現在、SimpleDB は整数定数と文字列定数のみをサポートしています。(a) SimpleDB を修正して、短整数、バイト配列、日付などの他の種類の定数を含めます。(b) 演習 3.17 では、Page クラスを変更して、短整数、日付などの型の get/set メソッドを含めるように指示しました。この演習を完了している場合は、Scan と UpdateScan (およびそれらのさまざまな実装クラス)、レコード マネージャー、トランザクション マネージャー、バッファー マネージャーに同様の get/set メソッドを追加してください。次に、getVal メソッドと setVal メソッドを適切に変更してください。 8.8.整数の算術演算子を処理できるように式を修正します。8.9. 比較演算子 < と > を処理できるようにクラス Term を修正します。8.10. ブール接続子 and、or、not の任意の組み合わせを処理できるようにクラス Predicate を修正します。8.11. 演習 6.13 では、SimpleDB レコード マネージャーを拡張して、データベースの null 値を処理できるようにしました。今度は、クエリ プロセッサを拡張して null を処理できるようにします。具体的には、次のようになります。

- クラス Constant を適切に変更します。
- TableScan の getVal メソッドと setVal メソッドを変更して、null 値を認識して適切に処理できるようにします。
- さまざまな Expression、Term、および Predicate クラスのうち、null 定数を処理するために変更する必要があるクラスを特定します。 8.12. クラス ProjectScan を更新スキャンになるように修正します。 8.13. 演習 6.10 では、クラス TableScan の previous メソッドと afterLast メソッドを作成するように指示されました。 (a) すべてのスキャンにこれらのメソッドが含まれるように SimpleDB を変更します。 (b) コードをテストするプログラムを作成します。変更内容を SimpleDB エンジンでテストするには、その JDBC 実装も拡張する必要があります。演習 11.5 を参照してください。 8.14. 名前変更演算子には、入力テーブル、テーブルからのフィールド名、および新しいフィールド名の 3 つの引数が必要です。出力テーブルは、指定されたフィールドの名前が変更されていることを除いて、入力テーブルと同一です。たとえば、次のクエリはフィールド SName の名前を StudentName に変更します。

rename(学生, SName, StudentName)

この演算子を実装するクラス RenameScan を作成します。このクラスは演習 10.13 で必要になります。

8.15. 拡張演算子は、入力テーブル、式、および新しいフィールド名の 3 つの引数を取ります。出力テーブルは、式によって値が決定される新しいフィールドも含まれる点を除いて、入力テーブルと同じです。たとえば、次のクエリは、学生が 3 年生だった年を計算する新しいフィールド (JuniorYear という名前) で STUDENT を拡張します。

extend(学生、大学院1年生、3年生)

この演算子を実装するクラス ExtendScan を作成します。このクラスは演習 10.14 で必要になります。

8.16. ユニオン関係演算子は 2 つの引数を取ります。どちらもテーブルです。出力テーブルには、入力テーブルのどこかに現れるレコードが含まれます。ユニオン クエリでは、基になるテーブルの両方が同じスキーマを持っている必要があります。出力テーブルにもそのスキーマが存在します。この演算子を実装するクラス UnionScan を記述します。このクラスは演習 10.15 で必要になります。

8.17. セミ結合演算子は、2 つのテーブルと述語の 3 つの引数を取ります。この演算子は、2 番目のテーブルに「一致する」レコードがある最初のテーブルのレコードを返します。たとえば、次のクエリは、少なくとも 1 人の学生の専攻がある学部を返します。

セミジョイン(学部、学生、専攻ID)

同様に、antijoin 演算子は、一致するレコードがない最初のテーブルのレコードを返します。たとえば、次のクエリは、学生の専攻がない学部を返します。

反結合(学部、学生、専攻ID=)

これらの演算子を実装するには、クラス SemijoinScan と AntijoinScan を記述します。これらのクラスは演習 10.16 で必要になります。

第9章 解析

JDBC クライアントは、SQL ステートメントを文字列としてデータベースエンジンに送信します。エンジンは、この文字列からクエリ ツリーを作成するために必要な情報を抽出する必要があります。この抽出プロセスには、構文ベースの段階 (解析) とセマンティクス ベースの段階 (計画) の 2 つの段階があります。この章では、解析について説明します。計画については、第 10 章で説明します。

9.1 構文と意味論

言語の構文は、意味のある文になり得る文字列を記述する一連の規則です。たとえば、次の文字列を考えてみましょう。

```
テーブルT1とT2から選択する。ただし、b - 3
```

この文字列が構文的に正しくない理由はいくつかあります。

- 選択句には何かが含まれている必要があります。
- 識別子テーブルはキーワードではなく、テーブル名として扱われます。
- テーブル名はキーワード and ではなく、カンマで区切る必要があります。
- 文字列「b – 3」は述語を表すものではありません。

これらの問題のそれぞれにより、この文字列は SQL ステートメントとして完全に無意味になります。識別子テーブル T1、T2、および b が何を示しているかに関係なく、エンジンがそれを実行する方法を見つけることは不可能です。

言語のセマンティクスは、構文的に正しい文字列の実際の意味を指定します。次の構文的に正しい文字列を考えてみましょう。

```
select a from x, z where b = 3
```


E. Sciore, *Database Design and Implementation*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-33836-7_9

この文は、2 つのテーブル (x と z) から 1 つのフィールド (a) を要求し、述語 b ¼ 3 を持つクエリであると推測できます。したがって、この文は意味がある可能性があります。

文が実際に意味を持つかどうかは、x、z、a、b の意味情報によって決まります。特に、x と z は、a という名前のフィールドと b という名前の数値フィールドを含むテーブルの名前でなければなりません。この意味情報は、データベースのメタデータから判断できます。パーサーはメタデータについて何も知らないため、SQL 文の意味を評価できません。代わりに、メタデータを調べる責任はプランナーにあります。プランナーについては、第 10 章で説明します。

9.2 語彙解析

パーサーの最初のタスクは、入力文字列をトークンと呼ばれる「チャンク」に分割することです。このタスクを実行するパーサーの部分は、字句解析器と呼ばれます。各トークンはタイプと値があります。SimpleDB 字句解析ツールは 5 つのトークン タイプをサポートしています。

- カンマなどの1文字の区切り文字
- 123などの整数定数
- 'joe' などの文字列定数
- select、from、whereなどのキーワード
- STUDENT、x、glop34aなどの識別子

空白文字 (スペース、タブ、改行) は通常トークンの一部ではありません。唯一の例外は文字列定数内です。空白の目的は、読みやすさを向上させ、トークンを互いに分離することです。

前の SQL ステートメントをもう一度考えてみましょう。

x、zからaを選択し、bを選択します= 3

字句解析器は、図 9.1 に示すように、10 個のトークンを作成します。

概念的には、字句解析器の動作は単純です。つまり、入力文字列を 1 文字ずつ読み取り、次のトークンが読み取られたと判断すると停止します。字句解析器の複雑さは、トークン タイプのセットに正比例します。つまり、検索するトークン タイプが多いほど、実装は複雑になります。

Java には、2 つの異なる組み込みトークナイザー (Java の語彙解析器の用語) が用意されています。1 つは StringTokenizer クラス、もう 1 つは StreamTokenizer クラスです。文字列トークナイザーの方が使い方は簡単ですが、サポートされるトークンは区切り記号と単語 (区切り記号間の部分文字列) の 2 種類だけです。これは SQL には適していません。特に、文字列トークナイザーは数字や引用符で囲まれた文字列を理解しないからです。一方、ストリーム トークナイザーには、SimpleDB で使用される 5 種類すべてをサポートするなど、広範なトークン タイプ セットがあります。

TYPE	VALUE
keyword	select
identifier	a
keyword	from
identifier	x
delimiter	,
identifier	z
keyword	where
identifier	b
delimiter	=
intconstant	3

図9.1 字句解析器によって生成されたトークン

図 9.2 は、StreamTokenizer の使用方法を示す TokenizerTest クラスのコードを示しています。このコードは、指定された入力行をトークン化し、各トークンの型と値を出力します。

tok.ordinaryChar('.') の呼び出しは、トークナイザーにピリオドを区切り文字として解釈するように指示します。(ピリオドは SimpleDB では使用されませんが、識別子の一部として受け入れられないようにするためには、区切り文字として識別することが重要です。) 逆に、tok.wordChars('_', '_') の呼び出しは、トークナイザーにアンダースコアを識別子の一部として解釈するように指示します。tok.lowerCaseMode(true) の呼び出しは、トークナイザーにすべての文字列トークン (引用符で囲まれた文字列を除く) を小文字に変換するように指示します。これにより、SQL はキーワードと識別子の大文字と小文字を区別しなくなります。

メソッド nextToken は、トークンをストリーム内の次のトークンに置きます。戻り値 TT_EOF は、トークンがもう存在しないことを示します。トークナイザーのパブリック変数 ttype には、現在のトークンの型が保持されます。値 TT_NUMBER は数値定数、TT_WORD は識別子またはキーワード、一重引用符の整数表現は文字列定数を示します。単一文字の区切りトークンの型は、その文字の整数表現です。

9.3 SimpleDB 字句解析器

SteamTokenizer クラスは汎用の字句解析器ですが、使いにくい場合があります。SimpleDB の Lexer クラスは、パーサーがトークン ストリームにアクセスするより簡単な方法を提供します。パーサーが呼び出すことができるメソッドには、現在のトークンについて問い合わせるメソッドと、字句解析器に現在のトークンを「食べる」ように指示してその値を返し、次のトークンに移動するように指示するメソッドの 2 種類があります。各トークン タイプには、対応するメソッドのペアがあります。これらの 10 個のメソッドの API は、図 9.3 に示されています。

最初の5つのメソッドは、現在のトークンに関する情報を返します。matchDelimメソッドは、現在のトークンが指定された区切り文字である場合にtrueを返します。

```
public class TokenizerTest { private static Collection<String> keywords = Arrays.asList("select", "fro
m", "where", "and", "insert", "into", "values", "delete", "update", "set", "create", "table","int", "varchar
", "view", "as", "index", "on"); public static void main(String[] args) throws IOException { String s =
getStringFromUser(); StreamTokenizer tok = new StreamTokenizer(new StringReader(s)); tok.ordinar
yChar('.'); tok.wordChars('_', '_'); tok.lowerCaseMode(true); // ID とキーワードを小文字に変換し
ます while (tok.nextToken() != TT_EOF) printCurrentToken(tok); } private static String getStringFro
mUser() { System.out.println("トークンを入力してください:"); Scanner sc = new Scanner(System.
in); String s = sc.nextLine(); sc.close(); return s; }private static void printCurrentToken(StreamTokeni
zer tok) throws IOException { if (tok.ttype == TT_NUMBER) System.out.println("IntConstant " + (in
t)tok.nval); else if (tok.ttype == TT_WORD) { String word = tok.sval; if (keywords.contains(word)) S
ystem.out.println("Keyword " + word); elseSystem.out.println("Id " + word); }else if (tok.ttype == '\'')
System.out.println("StringConstant " + tok.sval); elseSystem.out.println("区切り文字 " + (char)tok.tt
ype); }
```

```
}
```

図9.2 TokenizerTestクラス

値。同様に、matchKeyword は、現在のトークンが指定された値を持つキーワードである場合に true を返します。他の 3 つの matchXXX メソッドは、現在のトークンが適切なタイプである場合に true を返します。

最後の 5 つのメソッドは、現在のトークンを「消費」します。各メソッドは、対応する matchXXX メソッドを呼び出します。そのメソッドが false を返す場合は例外がスローされ、それ以外の場合は次のトークンが現在のトークンになります。さらに、eatIntConstant、eatStringConstant、eatId メソッドは、現在のトークンの値を返します。

Lexer

```
public boolean matchDelim(char d);
public boolean matchIntConstant();
public boolean matchStringConstant();
public boolean matchKeyword(String w);
public boolean matchId();

public void    eatDelim(char d);
public int     eatIntConstant();
public String  eatStringConstant();
public void    eatKeyword(String w);
public String  eatId();
```

図9.3 SimpleDB字句解析器のAPI

図 9.4 の LexerTest クラスは、これらのメソッドの使用法を示しています。コードは入力行を読み取ります。各行は「A ¼ c」または「c ¼ A」の形式であることが想定されています。ここで、A は識別子、c は int 定数です。その他の形式の入力行は例外を生成します。

Lexer のコードは図 9.5 に示されています。そのコンストラクタはストリーム トークナイザーを設定します。eatIntConstant、eatStringConstant、eatId メソッドは現在のトークンの値を返します。initKeywords メソッドは、SimpleDB の SQL バージョンで使用されるキーワードのコレクションを構築します。

パブリック クラス LexerTest { パブリック スタティック void main(String[] args) { 文字列 x = ""; int y = 0; スキャナ sc = new Scanner(System.in); while (sc.hasNext()) { 文字列 s = sc.nextLine(); レキサー lex = new Lexer(s); if (lex.matchId()) { x = lex.eatId(); lex.eatDelim('='); y = lex.eatIntConstant(); }else { y = lex.eatIntConstant(); lex.eatDelim('='); x = lex.eatId(); }System.out.println(x + " equals " + y); }sc.close(); } }

図9.4 LexerTestクラス

```
public class Lexer { private Collection<String> keywords; private StreamToken
izer tok; public Lexer(String s) { initKeywords(); tok = new StreamTokenizer(n
ew StringReader(s)); tok.ordinaryChar('.'); tok.wordChars('_', '_'); tok.lowerCas
eMode(true); nextToken(); } //現在のトークンのステータスを確認するメ
ソッド public boolean matchDelim(char d) { return d == (char)tok.ttype; }publ
ic boolean matchIntConstant() { return tok.ttype == StreamTokenizer.TT_NUM
BER; }public boolean matchStringConstant() { return '\'' == (char)tok.ttype; }p
ublic boolean matchKeyword(String w) { return tok.ttype == StreamTokenizer.
TT_WORD && tok.sval.equals(w); }public boolean matchId() { return tok.ttyp
e == StreamTokenizer.TT_WORD && !keywords.contains(tok.sval); } //現在
のトークンを「食べる」ためのメソッド public void eatDelim(char d) { if (
!matchDelim(d)) throw new BadSyntaxException(); nextToken(); }public int ea
tIntConstant() { if (!matchIntConstant()) throw new BadSyntaxException(); int i
= (int) tok.nval; nextToken(); return i; }
```

図9.5 SimpleDBクラスLexerのコード

```
public String eatStringConstant() { if (!matchStringConstant()) throw new BadSyntaxException(
); String s = tok.sval; nextToken(); return s; }public void eatKeyword(String w) { if (!matchKey
word(w)) throw new BadSyntaxException(); nextToken(); }public String eatId() { if (!matchId()
) throw new BadSyntaxException(); String s = tok.sval; nextToken(); return s; }private void next
Token() { try {tok.nextToken(); }catch(IOException e) { throw new BadSyntaxException(); } }p
rivate void initKeywords() { キーワード = Arrays.asList("select", "from", "where", "and", "inse
rt", "into", "values", "delete", "update", "set", "create", "table", "varchar", "int", "view", "as", "in
dex", "on"); }
```

```
}
```

図9.5（続き）

StreamTokenizer メソッド nextToken は IOException をスローします。Lex er メソッド nextToken はこの例外を BadSyntaxException に変換し、クライアントに返します (第 11 章で説明するように、SQLException に変換されます)。

9.4 文法

文法とは、トークンを合法的に組み合わせる方法を記述した一連の規則です。以下は文法規則の例です。

<フィールド>:= IdTok

文法規則の左側は、構文カテゴリを指定します。構文カテゴリは、言語の特定の概念を表します。上記の規則では、<Field> はフィールド名の概念を表します。文法規則の右側は、構文カテゴリに属する文字列のセットを指定するパターンです。上記の規則では、パターンは単純に IdTok であり、任意の識別子トークンと一致します。したがって、<Field> には、識別子に対応する文字列のセットが含まれます。

各構文カテゴリは、独自のミニ言語と考えることができます。たとえば、「SName」と「Glop」は <Field> のメンバーです。識別子は意味を持つ必要はなく、識別子であるだけでよいことに注意してください。したがって、「Glop」は、SimpleDB 大学データベースでも、<Field> の完全なメンバーです。ただし、「select」はキーワード トークンであり、識別子トークンではないため、<Field> のメンバーではありません。

文法規則の右側のパターンには、トークンと構文カテゴリの両方への参照を含めることができます。既知の値 (キーワードや区切り文字など) を持つトークンは明示的に表示されます。その他のトークン (識別子、整数定数、文字列定数) は、それぞれ IdTok、IntTok、StrTok と記述されます。3 つのメタ文字 (‘ [’、‘] ’、‘ | ’) は句読点として使用されます。これらの文字は言語の区切り文字ではないため、パターンを表現するために使用できます。説明のために、次の 4 つの追加の文法規則を考えてみましょう。

<定数> := StrTok | IntTok <式> := <フィールド> | <定数> <用語> := <式> = <式> <述語> := <用語> [AND <述語>]

最初のルールは、文字列または整数の任意の定数を表すカテゴリ <Constant> を定義します。メタ文字‘ | ’は「または」を意味します。したがって、カテゴリ <Constant> は文字列トークンまたは整数トークンのいずれかに一致し、その内容 (言語として) にはすべての文字列定数とすべての整数定数が含まれます。

2 番目の規則は、演算子のない式を表すカテゴリ <Expression> を定義します。この規則では、式がフィールドまたは定数のいずれかであることを指定します。

3 番目のルールは、式間の単純な等価項を表すカテゴリ <Term> を定義します (SimpleDB クラス Term と同様)。たとえば、次の文字列は <Term> に属します。

```
DeptId = DId
'math' = DName
```

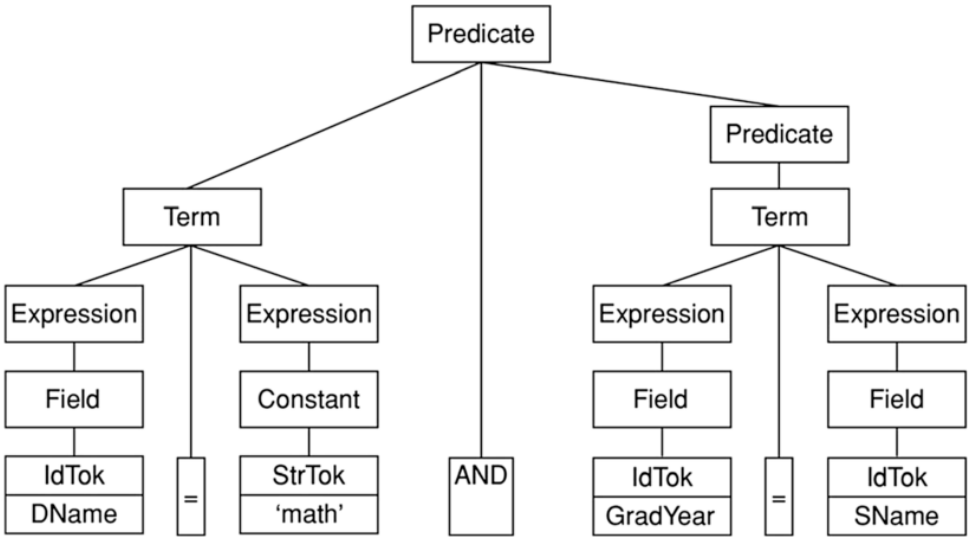


図9.6 文字列DName ¼ 'math' AND GradYear ¼ SNameの解析木

SName = 123 65 =
'abc'

パーサーは型の一貫性をチェックしないことに注意してください。したがって、最後の 2 つの文字列は意味的には正しくなくても、構文的には正しいです。

4 番目のルールは、SimpleDB クラス Predicate に似た、用語のブール値の組み合わせを表すカテゴリ <Predicate> を定義します。メタ文字 ‘ [’ と ‘] ’ はオプションであることを示します。したがって、ルールの右側は、<Term>、または <Term> の後に AND キーワード トークンが続き、さらに別の <Predicate> が (再帰的に) 続くトークンのシーケンスと一致します。たとえば、次の文字列は <Predicate> に属します。

DName = 'math' Id = 3 AND DName = 'math' MajorId
= DId AND Id = 3 AND DName = 'math'

最初の文字列の形式は <Term> です。次の 2 つの文字列の形式は <Term> AND <Predicate> です。

文字列が特定の構文カテゴリに属する場合、その理由を表すために構文ツリーを描くことができます。構文ツリーには、構文カテゴリが内部ノードとして、トークンがリーフ ノードとして存在します。カテゴリ ノードの子は、文法規則の適用に対応します。たとえば、図 9.6 には、次の文字列の構文ツリーが含まれています。

DName = 'math' AND GradYear = SName

この図では、入力文字列を読みやすくするために、ツリーのリーフ ノードがツリーの下部に沿って表示されています。ルート ノードから始めて、このツリーは、“ DName¼'math' ” が <Term> であり、“ GradYear¼SName ” が <Predicate> であるため、文字列全体が <Predicate> であると主張します。各サブツリーも同様に展開できます。たとえば、“ DName¼'math' ” は <Term> です。“ DName ” と “ 'math' ” はどちらも <Expression> に属しているためです。

<フィールド> := IdTok <定数> := StrTok | IntTok <式> := <フィールド> | <定数> <用語> := <式> = <式> <述語> := <用語> [AND <述語>] <クエリ> := SELECT <SelectList> FROM <TableList> [WHERE <述語>] <SelectList> := <フィールド> [、 <SelectList>] <TableList> := IdTok [、 <TableList>] <UpdateCmd> := <挿入> | <削除> | <変更> | <作成> <作成> := <テーブルの作成> | <ビューの作成> | <CreateIndex> <Insert> := INSERT INTO IdTok (<FieldList>) VALUES (<ConstList>) <FieldList> := <Field> [、 <FieldList>] <ConstList> := <Constant> [、 <ConstList>] <Delete> := DELETE FROM IdTok [WHERE <Predicate>] <変更> := UPDATE IdTok SET <フィールド> = <式> [WHERE <述語>] <テーブルの作成> := CREATE TABLE IdTok (<フィールド定義>) <フィールド定義> := <フィールド定義> [, <フィールド定義>] <フィールド定義> := IdTok <型定義> <型定義> := INT | VARCHAR (IntTok) <CreateView> := CREATE VIEW IdTok AS <Query> <CreateIndex> := CREATE INDEX IdTok ON IdTok (<Field>)

図9.7 SQLのSimpleDBサブセットの文法

図 9.7 は、SimpleDB でサポートされている SQL のサブセットの文法全体を示しています。文法規則は 9 つのセクションに分かれています。1 つは述語、式、フィールドなどの一般的な構成要素、もう 1 つはクエリ、そして 7 つはさまざまな種類の更新ステートメントです。

項目のリストは SQL で頻繁に使用されます。たとえば、クエリでは、select 句にはフィールドのカンマ区切りリストが含まれ、from 句には識別子のカンマ区切りリストが含まれ、where 句には用語の AND 区切りリストが含まれます。各リストは、<Predicate> で見たのと同じ再帰技法を使用して文法で指定されます。また、<Query>、<Delete>、および <Modify> のルールで「オプション ブラケット」表記が使用され、オプションの where 句を使用できるようにしていることにも注意してください。

パーサーは、識別子の型を認識できないため、型の互換性を強制できないと述べました。また、パーサーは互換性のあるリストサイズを強制することもできません。たとえば、SQL挿入ステートメントでは、フィールド名と同じ数の値を記述する必要がありますが、<Insert>の文法規則では、

文字列には <FieldList> と <ConstList> があります。プランナーは、これらのリストが同じサイズであること (および型に互換性があること) を検証する必要があります。[1]

9.5 再帰下降パーサ

解析ツリーは、特定の文字列が構文的に正しいことの証明と考えることができます。しかし、解析ツリーをどのように判断するのでしょうか。データベース エンジンは、文字列が構文的に正しいかどうかをどのように判断するのでしょうか。

コンパイラ言語の研究者は、この目的のために数多くの解析アルゴリズムを開発してきました。解析アルゴリズムの複雑さは、通常、サポートできる文法の複雑さに比例します。幸いなことに、SQL 文法は可能な限り単純なので、再帰降下法と呼ばれる、可能な限り単純な解析アルゴリズムを使用できます。

基本的な再帰下降パーサーでは、各構文カテゴリは void メソッドによって実装されます。このメソッドを呼び出すと、そのカテゴリの解析ツリーを構成するトークンが「取り込まれ」、戻ります。トークンがそのカテゴリの解析ツリーに対応していない場合、メソッドは例外をスローします。図 9.7 の最初の 5 つの文法規則を検討します。これは述語に対応する SQL のサブセットを形成します。この文法に対応する Java クラスは図 9.8 に示されています。

メソッド field について考えてみましょう。このメソッドは、字句解析器を 1 回呼び出します (戻り値は無視されます)。次のトークンが識別子の場合、呼び出しは正常に戻り、そのトークンは消費されます。そうでない場合、メソッドは呼び出し元に例外を返します。同様に、メソッド term について考えてみましょう。expression の最初の呼び出しは、1 つの SQL 式に対応するトークンを消費し、eatDelim の呼び出しは等号トークンを消費し、expression の 2 番目の呼び出しは、別の SQL 式に対応するトークンを消費します。これらのメソッド呼び出しのいずれかが、予期したトークンを見つけられなかった場合、例外がスローされ、term メソッドはそれを呼び出し元に渡します。

選択肢を含む文法規則は、if 文を使用して実装されます。if 文の条件は、現在のトークンを見て、何をするかを決定します。簡単な例として、メソッド定数を考えてみましょう。現在のトークンが文字列定数である場合、メソッドはそれを消費します。そうでない場合は、メソッドは整数定数を消費しようとします。現在のトークンが文字列定数でも整数定数でもない場合は、lex.eatIntConstant の呼び出しによって例外が生成されます。それほど簡単ではない例として、メソッド式を考えてみましょう。ここで、メソッドは、

[1] This situation is certainly not desirable; in fact, it would be really nice to have a grammar in which equal list sizes are enforced. However, one can use automata theory to prove that no such grammar is possible.

```
public class PredParser {

  private Lexer lex;

  public PredParser(String s) {
     lex = new Lexer(s);
  }

  public void field() {
     lex.eatId();
  }

  public void constant() {
     if (lex.matchStringConstant())
        lex.eatStringConstant();
     else
        lex.eatIntConstant();
  }

  public void expression() {
     if (lex.matchId())
        field();
     else
        constant();
  }

  public void term() {
     expression();
     lex.eatDelim('=');
     expression();
  }

  public void predicate() {
     term();
     if (lex.matchKeyword("and")) {
        lex.keyword("and");
        predicate();
     }
  }
}
```

図9.8 述語のための簡略化された再帰下降パーサのコード

現在のトークンが識別子である場合はフィールドを探す必要があります。それ以外の場合は定数を探す必要があります。[2]

メソッド predicate は、再帰ルールがどのように実装されるかを示しています。最初にメソッド term を呼び出し、現在のトークンがキーワード AND であるかどうかを確認します。そうである場合、AND トークンを消費し、自分自身を再帰的に呼び出します。現在のトークンが AND でない場合は、リストの最後の項を見たと認識して戻ります。その結果、predicate の呼び出しは、トークン ストリームから可能な限り多くのトークンを消費します。AND トークンを見ると、有効な述語をすでに見ていたとしても、呼び出しは続行されます。

再帰下降解析の興味深い点は、メソッド呼び出しのシーケンスによって入力文字列の解析ツリーが決まることです。演習 9.4 では、各メソッドのコードを変更して、メソッド名を適切にインデントして出力するように求められます。結果は横向きの解析ツリーのようになります。

9.6 パーサーへのアクションの追加

基本的な再帰下降構文解析アルゴリズムは、入力文字列が構文的に正しい場合、通常通り戻ります。この動作は興味深いものですが、特に有用ではありません。そのため、基本パーサーは、

[2]This example also demonstrates the limitations of recursive-descent parsing. If a grammar rule has two alternatives that both require the same first token, then there would be no way to know which alternative to take and recursive descent will not work. In fact, you may have noticed that the grammar of Fig. 9.7 has this very problem. Exercise 9.3 addresses the issue.

プランナーが必要とする情報。この変更は、パーサーへのアクションの追加と呼ばれます。

一般的に、SQL パーサーは、SQL ステートメントからテーブル名、フィールド名、述語、定数などの情報を抽出する必要があります。抽出される情報は、SQL ステートメントの種類によって異なります。

- クエリの場合: フィールド名のリスト (select 句から)、テーブル名のコレクション (from 句から)、述語 (where 句から)
- 挿入の場合: テーブル名、フィールド名のリスト、値のリスト
- 削除の場合: テーブル名と述語
- 変更の場合: テーブル名、変更するフィールド名、新しいフィールド値を示す式、述語
- テーブル作成の場合: テーブル名とそのスキーマ
- ビュー作成の場合: テーブル名とその定義
- インデックス作成の場合: インデックス名、テーブル名、インデックスフィールドの名前

この情報は、Lexer メソッドの戻り値を介してトークン ストリームから抽出できます。したがって、各パーサー メソッドを変更する戦略は簡単です。eatId、eatStringConstant、eatIntConstant の呼び出しから戻り値を取得し、それらを適切なオブジェクトに組み立てて、そのオブジェクトをメソッドの呼び出し元に返します。

図 9.9 は、図 9.7 の文法を実装するメソッドを持つ Parser クラスのコードを示しています。次のサブセクションでは、このコードを詳細に調べます。

9.6.1 述語と式の解析

パーサーの核心は、述語と式を定義する 5 つの文法規則を扱います。これらは、さまざまな種類の SQL 文を解析するために使用されるためです。Parser のこれらのメソッドは、アクションと戻り値が含まれるようになったことを除いて、PredParser (図 9.8) と同じです。特に、メソッド field は、現在のトークンからフィールド名を取得して返します。メソッド constant、expression、term、および predicate は類似しており、Constant オブジェクト、Expression オブジェクト、Term オブジェクト、および Predicate オブジェクトを返します。

9.6.2 クエリの解析

メソッド query は、構文カテゴリ <Query> を実装します。パーサがクエリを解析すると、プランナに必要な 3 つの項目 (フィールド名、テーブル名、述語) が取得され、QueryData オブジェクトに保存されます。クラス QueryData は、メソッド fields、tables、pred を介してこれらの値を提供します (図 9.10 を参照)。このクラスには、クエリ文字列を再作成するメソッド toString もあります。このメソッドは、ビュー定義を処理するときに必要になります。

パブリッククラス Parser { プライベー
ト Lexer lex; パブリック Parser(String s)
{ lex = 新しい Lexer(s); }

// 述語とそのコンポーネントを解析するためのメソッド public String fiel
d() { return lex.eatId(); }public Constant constant() { if (lex.matchStringConsta
nt()) return new Constant(lex.eatStringConstant()); elsereturn new Constant(lex.
eatIntConstant()); }public Expression expression() { if (lex.matchId()) return ne
w Expression(field()); elsereturn new Expression(constant()); }public Term ter
m() { Expression lhs = expression(); lex.eatDelim('='); Expression rhs = express
ion(); return new Term(lhs, rhs); }public Predicate predicate() { Predicate pred
= new Predicate(term()); if (lex.matchKeyword("and")) { lex.eatKeyword("and"
); pred.conjoinWith(predicate()); }return pred; }

// クエリを解析するためのメソッド public QueryData query() { lex.e
atKeyword("select"); List<String> fields = selectList(); lex.eatKeyword(
"from"); Collection<String> tables = tableList(); Predicate pred = new P
redicate(); if (lex.matchKeyword("where")) { lex.eatKeyword("where");
pred = predicate(); }return new QueryData(fields, tables, pred);

図9.9 SimpleDBクラスParserのコード

```
}private List<String> selectList() { List<String> L = new ArrayList<Stri
ng>(); L.add(field()); if (lex.matchDelim(',')) { lex.eatDelim(','); L.addAll(sel
ectList()); }return L; }private Collection<String> tableList() { Collection<Stri
ng> L = new ArrayList<String>(); L.add(lex.eatId()); if (lex.matchDelim(','))
{ lex.eatDelim(','); L.addAll(tableList()); }return L; } // さまざまな更新コマ
ンドを解析するためのメソッド public Object updateCmd() { if (lex.match
Keyword("insert")) return insert();そうでない場合は、lex.matchKeyword("
delete") で delete() を返します。そうでない場合は、lex.matchKeyword("
update") で modify() を返します。そうでない場合は、create() を返しま
す。}private Object create() { lex.eatKeyword("create"); if (lex.matchKeywor
d("table")) で createTable() を返します。そうでない場合は、lex.matchKe
yword("view")) で createView() を返します。そうでない場合は、createIn
dex() を返します。}

// 削除コマンドを解析するためのメソッド public DeleteDa
ta delete() { lex.eatKeyword("delete"); lex.eatKeyword("from")
; String tblname = lex.eatId(); Predicate pred = new Predicate();
if (lex.matchKeyword("where")) { lex.eatKeyword("where"); pr
ed = predicate(); }return new DeleteData(tblname, pred);

}
```

Fig. 9.9 (continued)

```
// 挿入コマンドを解析するためのメソッド public InsertData insert() { lex.eatKeywo
rd("insert"); lex.eatKeyword("into"); String tblname = lex.eatId(); lex.eatDelim('('); List
<String> flds = fieldList(); lex.eatDelim(')'); lex.eatKeyword("values"); lex.eatDelim('(');
List<Constant> vals = constList(); lex.eatDelim(')'); return new InsertData(tblname, flds,
vals); }private List<String> fieldList() { List<String> L = new ArrayList<String>(); L.
add(field()); if (lex.matchDelim(',')) { lex.eatDelim(','); L.addAll(fieldList()); }return L;
}private List<Constant> constList() { List<Constant> L = new ArrayList<Constant>();
L.add(constant()); if (lex.matchDelim(',')) { lex.eatDelim(','); L.addAll(constList()); }retu
rn L; } // 変更コマンドを解析するためのメソッド public ModifyData modify() { lex
.eatKeyword("update"); String tblname = lex.eatId(); lex.eatKeyword("set"); String fldna
me = field(); lex.eatDelim('=');式 newval = expression(); 述語 pred = new Predicate(); if
(lex.matchKeyword("where")) { lex.eatKeyword("where"); pred = predicate(); }return ne
w ModifyData(tblname, fldname, newval, pred); }
```

```
// テーブル作成コマンドを解析するメソッド
    public CreateTableData createTable() { lex.eatKeyword
    ("table"); 文字列 tblname = lex.eatId(); lex.eatDelim('(')
    ; スキーマ sch = fieldDefs();
```

Fig. 9.9 (continued)

```
        lex.eatDelim(')'); return new CreateTableData(tblname, sch); }
    private Schema fieldDefs() { Schema schema = fieldDef(); if (lex.
    matchDelim(',')) { lex.eatDelim(','); Schema schema2 = fieldDefs()
    ; schema.addAll(schema2); }return schema; }private Schema field
    Def() { String fldname = field(); return fieldType(fldname); }priva
    te Schema fieldType(String fldname) { Schema schema = new Sch
    ema(); if (lex.matchKeyword("int")) { lex.eatKeyword("int"); sche
    ma.addIntField(fldname); }else { lex.eatKeyword("varchar"); lex.e
    atDelim('('); int strLen = lex.eatIntConstant(); lex.eatDelim(')'); sc
    hema.addStringField(fldname, strLen); }return schema; }

// ビュー作成コマンドを解析するメソッド public CreateViewD
ata createView() { lex.eatKeyword("view"); String viewname = lex.
eatId(); lex.eatKeyword("as"); QueryData qd = query(); return new
CreateViewData(viewname, qd); }

// インデックス作成コマンドを解析するメソッド public CreateIndexData creat
eIndex() { lex.eatKeyword("index"); String idxname = lex.eatId(); lex.eatKeyword("
on"); String tblname = lex.eatId(); lex.eatDelim('('); String fldname = field(); lex.eat
Delim(')'); return new CreateIndexData(idxname, tblname, fldnam

                                                                    e);
    }
}
```

Fig. 9.9 (continued)

パブリック クラス QueryData { private List<String> fields; private Collection<String> tables; private Predicate pred; public QueryData(List<String> fields, Collection<String> tables, Predicate pred) { this.fields = fields; this.tables = tables; this.pred = pred; }public List<String> fields() { return fields; }public Collection<String> tables() { return tables; }public Predicate pred() { return pred; }public String toString() { String result = "select "; for (String fldname : fields) result += fldname + ", "; result = result.substring(0, result.length()-2); // 最後のカンマを zap します result += " from "; for (String tblname : tables) result += tblname + ", "; result = result.substring(0, result.length()-2); // 最後のカンマを zap します String predstring = pred.toString(); if (!predstring.equals("")) result += " where " + predstring; return result; } }

図9.10 SimpleDBクラスQueryDataのコード

9.6.3 更新の解析

パーサー メソッド updateCmd は、さまざまな SQL 更新ステートメントの結合を表す構文カテゴリ <UpdateCmd> を実装します。このメソッドは、JDBC メソッド executeUpdate の実行中に呼び出され、コマンドが表す更新の種類を判断します。このメソッドは、文字列の最初のトークンを使用してコマンドを識別し、そのコマンドの特定のパーサー メソッドにディスパッチします。各更新メソッドは、コマンド文字列から異なる情報を抽出するため、戻り値の型が異なります。したがって、メソッド updateCmd は Object 型の値を返します。

```
public class InsertData {
   private String tblname;
   private List<String> flds;
   private List<Constant> vals;

   public InsertData(String tblname, List<String> flds,
                     List<Constant> vals) {
      this.tblname = tblname;
      this.flds = flds;
      this.vals = vals;
   }

   public String tableName() {
      return tblname;
   }

   public List<String> fields() {
      return flds;
   }

   public List<Constant> vals() {
      return vals;
   }
}
```

図9.11 SimpleDBクラスInsertDataのコード

9.6.4 挿入の解析

パーサーメソッド insert は、構文カテゴリ <Insert> を実装します。このメソッドは、テーブル名、フィールドリスト、値リストの 3 つの項目を抽出します。図 9.11 に示すクラス InsertData は、これらの値を保持し、アクセサメソッドを介して利用できるようにします。

9.6.5 削除の解析

削除ステートメントは、delete メソッドによって処理されます。このメソッドは、DeleteData クラスのオブジェクトを返します (図 9.12 を参照)。クラス コンストラクタは、指定された削除ステートメントからテーブル名と述語を格納し、それらにアクセスするためのメソッド tableName と pred を提供します。

9.6.6 解析の変更

変更ステートメントは、メソッドmodifyによって処理されます。このメソッドは、図9.13に示すように、ModifyDataクラスのオブジェクトを返します。このクラスは、

パブリック クラス DeleteData { プライベート String tblname; プライベート Predicate pred; パブリック DeleteData(String tblname, Predicate pred) { this.tblname = tblname; this.pred = pred; }パブリック String tableName() { 戻り値 tblname; }パブリック Predicate pred() { 戻り値 pred; } }

図9.12 SimpleDBクラスDeleteDataのコード

パブリック クラス ModifyData { private String tblname; private String fldname; private Expression newval; private Predicate pred; public ModifyData(String tblname, String fldname, Expression newval, Predicate pred) { this.tblname = tblname; this.fldname = fldname; this.newval = newval; this.pred = pred; }public String tableName() { return tblname; }public String targetField() { return fldname; }public Expression newValue() { return newval; }public Predicate pred() { return pred; } }

図9.13 SimpleDBクラスModifyDataのコード

DeleteData のクラスとの違いは、このクラスが代入情報 (代入の左側のフィールド名と代入の右側の式) も保持していることです。追加のメソッド targetField と newValue がこの情報を返します。

9.6.7 テーブル、ビュー、インデックスの作成の解析

構文カテゴリ <Create> は、SimpleDB でサポートされている 3 つの SQL 作成ステートメントを指定します。テーブル作成ステートメントは、構文カテゴリ <CreateTable> とそのメソッド createTable によって処理されます。メソッド fieldDef と fieldType は、1 つのフィールドの情報を抽出し、それを独自の Schema オブジェクトに保存します。メソッド fieldDefs は、このスキーマをテーブルのスキーマに追加します。テーブル名とスキーマは、CreateTableData オブジェクト内に返されます。そのコードは図 9.14 に示されています。

ビュー作成ステートメントは、createView メソッドによって処理されます。このメソッドは、ビューの名前と定義を抽出し、CreateViewData 型のオブジェクトでそれらを返します。図 9.15 を参照してください。ビュー定義の処理は一般的ではありません。不正な形式のビュー定義を検出するには、ビュー定義を <Query> として解析する必要があります。ただし、メタデータ マネージャーは定義の解析された表現を保存したくありません。必要なのは実際のクエリ文字列です。その結果、CreateViewData コンストラクターは、返された QueryData オブジェクトに対して toString を呼び出すことによって、ビュー定義を再作成します。実際には、toString メソッドはクエリを「解析解除」します。

インデックスは、データベース システムがクエリの効率を向上させるために使用するデータ構造です。インデックスについては、第 12 章で説明します。createIndex パーサー メソッドは、インデックス名、テーブル名、フィールド名を抽出し、CreateIndexData オブジェクトに保存します。図 9.16 を参照してください。

パブリック クラス CreateTableData { プライベート String tblname; プライベート Schema sch; パブリック CreateTableData(String tblname, Schema sch) { this.tblname = tblname; this.sch = sch; }パブリック String tableName() { return tblname; }パブリック Schema newSchema() { return sch; } }

図9.14 SimpleDBクラスCreateTableDataのコード

パブリック クラス CreateViewData { プライベート String viewname; プライベート Query Data qrydata; パブリック CreateViewData(String viewname, QueryData qrydata) { this.viewna me = viewname; this.qrydata = qrydata; }パブリック String viewName() { return viewname; } パブリック String viewDef() { return qrydata.toString(); } }

図9.15 SimpleDBクラスCreateViewDataのコード

パブリック クラス CreateIndexData { private String idxname, tblname, fldname; pu blic CreateIndexData(String idxname, String tblname, String fldname) { this.idxname = idxname; this.tblname = tblname; this.fldname = fldname; }public String indexNam e() { return idxname; }public String tableName() { return tblname; }public String fiel dName() { return fldname; } }

図9.16 SimpleDBクラスCreateIndexのコード データ

9.7 章の要約

- 言語の構文は、意味のある文になる可能性のある文字列を記述する一連の規則です。

- パーサーは、入力文字列が構文的に正しいことを確認する責任があります。
- 字句解析器は、入力文字列を一連のトークンに分割するパーサーの部分です。
- 各トークンにはタイプと値があります。SimpleDB 字句解析ツールは 5 つのトークン タイプをサポートしています。

  – カンマなどの 1 文字の区切り文字 – 123 などの整数定数 – 'joe' などの文字列定数 – select、from、where などのキーワード – STUDENT、x、glop34a などの識別子

- 各トークン タイプには、現在のトークンについて問い合わせるメソッドと、字句解析プログラムに現在のトークンを「食べる」ように指示し、その値を返して次のトークンに移動するように指示するメソッドの 2 つのメソッドがあります。
- 文法とは、トークンを合法的に組み合わせる方法を記述する一連の規則です。

  – 文法規則の左側は、その構文カテゴリを指定します。構文カテゴリは、言語の特定の概念を表します。 – 文法規則の右側は、そのカテゴリの内容、つまり規則を満たす文字列のセットを指定します。

- 構文解析ツリーには、内部ノードとして構文カテゴリがあり、リーフノードとしてトークンがあります。カテゴリノードの子は、文法規則の適用に対応します。文字列が構文カテゴリ内にあるのは、そのカテゴリをルートとする構文解析ツリーがある場合のみです。
- 解析アルゴリズムは、構文的に正しい文字列から解析ツリーを構築します。解析アルゴリズムの複雑さは、通常、サポートできる文法の複雑さに比例します。単純な解析アルゴリズムは再帰降下法として知られています。
- 再帰下降パーサーには、各文法規則のメソッドがあります。各メソッドは、規則の右側の項目に対応するメソッドを呼び出します。
- 再帰下降パーサーの各メソッドは、読み取ったトークンの値を抽出して返します。SQL パーサーは、SQL ステートメントからテーブル名、フィールド名、述語、定数などの情報を抽出する必要があります。抽出される情報は、SQL ステートメントの種類によって異なります。

  – クエリの場合: フィールド名のコレクション (select 句から)、テーブル名のコレクション (from 句から)、述語 (where 句から) – 挿入の場合: テーブル名、フィールド名のリスト、値のリスト – 削除の場合: テーブル名と述語 – 変更の場合: テーブル名、変更するフィールドの名前、新しいフィールド値を示す式、述語 – テーブル作成の場合: テーブル名とそのスキーマ – ビュー作成の場合: テーブル名とその定義 – インデックス作成の場合: インデックス名、テーブル名、インデックス付きフィールドの名前

9.8 推奨される読み物

語彙分析と構文解析の分野は、60 年以上前から多大な注目を集めています。この本 (Scott、2000) は、現在使用されているさまざまなアルゴリズムの優れた入門書です。Zql (zql.sourceforge.net) など、Web 上では多数の SQL パーサーが利用可能です。SQL 文法は Date と Darwen (2004) の付録に記載されています。SQL とその文法を説明した SQL-92 標準のコピーは、URL www.contrib.andrew.cmu.edu/ ~shadow/sql/sql1992.txt にあります。標準ドキュメントを一度も見たことがない場合は、経験のためにもこれをチェックしてください。

Date, C., Darwen, H. (2004)。SQL 標準ガイド (第 4 版)。ボストン、マサチューセッツ州: Addison Wesley。
スコット、M. (2000)。プログラミング言語語用論。サンフランシスコ、カリフォルニア州：モーガン・カウフマン。

9.9 演習

概念上の問題

9.1. 次の SQL 文の解析ツリーを描画します。

(a) x から a を選択し、 b = 3 とする (b) x、y、z から a、b を選択する (c) x から a =、b、c = 0 を削除する (d) x を更新し、 a =、b、c = 3 とする (e) x に (a、b、c) の値 (3、'glop'、4) を挿入する (f) テーブル x (a varchar(3)、 b int、 c varchar(2)) を作成する

9.2. 次の各文字列について、解析時に例外が生成される場所とその理由を述べます。次に、JDBC クライアントから各クエリを実行し、何が起こるかを確認します。

a) xから選択
b) xからx xを選択
c) y zからxを選択
d) b=3からaを選択する
e) yからaを選択する。ただしbは-=3である。
f) yからaを選択する。

9.3. パーサーメソッドcreateは、図9.7のSQL文法に対応していません。

(a) <Create>の文法規則が再帰下降構文解析に使用するには曖昧すぎる理由を説明してください。

(b) create メソッドが実際にどのように動作するかに対応するように文法を修正します。

プログラミングの問題

9.4. 再帰ルールに対応する各パーサーメソッドを修正して、再帰ではなく while ループを使用するようにします。

9.5. クラス PredParser (図 9.8 参照) を修正して、メソッド呼び出しのシーケンスから得られる解析ツリーを出力します。

9.6. 演習 8.8 では、算術演算を処理するために式を変更するように求められました。

(a) 同様に SQL 文法を修正します。(b) SimpleDB パーサーを修正して文法の変更を実装します。(c) サーバーをテストするための JDBC クライアントを作成します。たとえば、専攻が 30 であるすべての学生の卒業年を増やす SQL クエリを実行するプログラムを作成します。

9.7. 演習 8.9 では用語を変更するように求められました。

(a) 同様に SQL 文法を修正します。(b) SimpleDB パーサーを修正して文法の変更を実装します。(c) サーバーをテストするための JDBC クライアントを作成します。たとえば、2010 年以前に卒業したすべての学生の名前を取得する SQL クエリを実行するプログラムを作成します。

9.8. 演習 8.10 では述語を修飾するように求められました。

(a) 同様に SQL 文法を修正します。(b) SimpleDB パーサーを修正して文法の変更を実装します。(c) サーバーをテストするための JDBC クライアントを作成します。たとえば、専攻が 10 または 20 であるすべての学生の名前を取得する SQL クエリを実行するプログラムを作成します。

9.9. シムpleDBは前置詞に括弧も許可しません icates します。

(a) SQL 文法を適切に修正します (演習 9.8 を実行しても実行しなくてもかまいません)。 (b) 文法の変更を実装するために SimpleDB パーサーを修正します。 (c) 変更をテストするために JDBC クライアントを作成します。

9.10. 結合述語は、from 句の JOIN キーワードを使用して標準 SQL で指定できます。たとえば、次の 2 つのクエリは同等です。

STUDENT、DEPT から SName、DName を選択します。MajorId = Did、GradYear = 2020 です。

```
STUDENT から SName、DName を選択し、Maj
orId = で DEPT に参加し、GradYear = 2020 で実
行します。
```

(a) SQL 字句解析器を修正して、キーワード「join」と「on」を含めます。(b) 明示的な結合を処理できるように SQL 文法を修正します。(c) SimpleDB パーサーを修正して、文法の変更を実装します。where 句から取得した述語に結合述語を追加します。(d) 変更をテストする JDBC プログラムを作成します。

9.11. 標準 SQL では、テーブルに範囲変数を関連付けることができます。そのテーブルからのフィールド参照には、その範囲変数がプレフィックスとして付けられます。たとえば、次のクエリは演習 9.10 のクエリのどちらとも同等です。

```
STUDENT s、DEPT d から s.SName、d.DName を選択
します。ここで、s.MajorId = d.Did および s.GradYear =
2020
```

(a) この機能を使用できるように SimpleDB 文法を修正します。 (b) 文法の変更を実装するために SimpleDB パーサーを修正します。パーサーによって返される情報も変更する必要があります。プランナーも拡張しない限り、SimpleDB サーバーで変更をテストすることはできないことに注意してください。演習 10.13 を参照してください。

9.12. 標準 SQL でキーワード AS を使用して、計算された値で出力テーブルを拡張できます。例:

```
STUDENTからSName、GradYear-1をJuniorYearとして選択
```

(a) SimpleDB 文法を修正して、選択句の任意のフィールドの後にオプションの AS 式を許可します。 (b) SimpleDB 字句解析器とパーサーを修正して、文法の変更を実装します。パーサーはこの追加情報をどのように利用できるようにする必要がありますか? プランナーも拡張しない限り、SimpleDB サーバーで変更をテストすることはできないことに注意してください。演習 10.14 を参照してください。

9.13. 標準 SQL では、キーワード UNION を使用して 2 つのクエリの出力テーブルを結合できます。例:

```
SName を STUDENT から選択します。MajorId は = 10
です。SName を STUDENT から選択します。MajorId
は = 20 です。
```

(a) SimpleDB の文法を修正して、クエリが他の 2 つのクエリの結合となるようにします。

(b) SimpleDBの字句解析器とパーサーを修正して文法の変更を実装します。変更をテストすることはできません。

プランナーも拡張しない限り、SimpleDB サーバーは拡張されません。演習 10.15 を参照してください。

9.14. 標準SQ L は where 句でネストされたクエリをサポー または例えば、

STUDENTからSNameを選択
MajorId が DEPT から Did を選択し、DName = が 'math' であるところ

(a) SimpleDB 文法を修正して、用語が「fieldname op query」の形式になるようにします。ここで、op は「in」または「not in」のいずれかです。 (b) SimpleDB 字句解析器とパーサーを修正して、文法の変更を実装します。プランナーも拡張しない限り、変更内容を SimpleDB サーバーでテストすることはできないことに注意してください。演習 10.16 を参照してください。

9.15. 標準 SQL では、SELECT 句で「"」文字を使用して、テーブルのすべてのフィールドを表すことができます。SQL が範囲変数をサポートしている場合 (演習 9.11 のように)、「"」の前に範囲変数を付けることもできます。

(a) SimpleDB 文法を修正して、クエリに「"」が表示されるようにします。 (b) SimpleDB パーサーを修正して、文法の変更を実装します。プランナーも拡張しない限り、SimpleDB サーバーで変更をテストすることはできないことに注意してください。演習 10.17 を参照してください。

9.16. 標準 SQL では、次の挿入ステートメントのバリエーションを使用してテーブルにレコードを挿入できます。

MATHSTUDENT(SId, SName) に挿入します。
STUDENT、DEPT から SId、SName を選択します。ここで、MajorId = DId、DName = は 'math' です。

つまり、SELECT ステートメントによって返されたレコードが指定されたテーブルに挿入されます。(上記のステートメントでは、空の MATHSTUDENT テーブルがすでに作成されていることを前提としています。)

(a) この形式の挿入を処理できるように SimpleDB SQL 文法を修正します。 (b) 文法を実装するように SimpleDB パーサー コードを修正します。プランナーも変更しないと、JDBC クエリを実行できないことに注意してください。演習 10.18 を参照してください。

9.17. 演習 8.7 では、新しいタイプの定数を作成するように求められました。

(a) SimpleDB SQL 文法を変更して、これらの型を create table 文で使用できるようにします。(b) 新しい定数リテラルを導入する必要がありますか? その場合は、<Constant> 構文カテゴリを変更します。(c) SimpleDB パーサー コードを修正して、文法を実装します。

9.18. 演習 8.11 では、null 値を実装するように求められました。この演習では、SQL を修正して null を理解するように求められます。

(a) SimpleDB 文法を修正して、キーワード null を定数として受け入れます。(b) SimpleDB パーサーを修正して、パート (a) の文法変更を実装します。(c) 標準 SQL では、用語は GradYear is null という形式にすることができ、式 GradYear が null 値の場合に true を返します。2 つのキーワード is null は、1 つの引数を持つ単一の演算子として扱われます。SimpleDB 文法を修正して、この新しい演算子を追加します。(d) SimpleDB パーサーとクラス Term を修正して、パート (c) の文法変更を実装します。(e) JDBC でプログラムを記述してコードをテストします。プログラムで値を null に設定し (または新しく挿入されたレコードの未割り当て値を使用し)、is null を含むクエリを実行します。SimpleDB の JDBC 実装を変更するまで、プログラムは null 値を出力できないことに注意してください。演習 11.6 を参照してください。

9.19. オープンソース ソフトウェア パッケージ javacc (URL javacc.github.io/javacc を参照) は、文法仕様からパーサーを構築します。javacc を使用して、SimpleDB 文法のパーサーを作成します。次に、既存のパーサーを新しいパーサーに置き換えます。

9.20. Parser クラスには、文法内の各構文カテゴリのメソッドが含まれています。簡略化された SQL 文法は小さいため、クラスは管理しやすいです。ただし、フル機能の文法では、クラスが大幅に大きくなります。別の実装戦略は、各構文カテゴリを独自のクラスに配置することです。クラスのコンストラクタは、そのカテゴリの解析を実行します。クラスには、解析されたトークンから抽出された値を返すメソッドもあります。この戦略では、比較的小さい多数のクラスが作成されます。この戦略を使用して、SimpleDB パーサーを書き直してください。