平成18年度 修士論文

アプリケーションに応じた AOP による高速化が可能な 永続システム

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 05M37027

青木 康博

指導教員

千葉 滋 助教授

平成19年1月19日

概要

Java 言語におけるリレーショナルデータベースを利用したアプリケーション開発には、EJB2 に代表される永続システムが利用されてきた。ここで言う永続システムとは、オブジェクト指向システムとデータベースとのインピーダンス・ミスマッチを自動的に吸収してくれるシステムのことを指す。オブジェクト指向とデータベースとの連結には、SQL の発行や永続オブジェクト生成といった煩雑な作業が必要である。一方で、永続システムでは、初めに永続オブジェクト (ルート・オブジェクト) を取得する作業以外は、永続オブジェクトのリファレンスを辿っていく際に自動的に煩雑な SQL を発行し必要なオブジェクトを取得してくれるため、開発者はデータベースを意識する必要が無く開発効率を高めることが出来る。

アプリケーションの高速化を図るためにはデータベースからの効率のよいデータ取得が重要である。効率のよいデータ取得とは、少ないデータベース・アクセス回数でアプリケーションに必要最小限のデータのみを取得することを指す。例えば、今後アプリケーションで利用される可能性が高いデータを先読みして一括取得することでデータベースへのアクセス回数を大幅に削減出来る。しかし、既存の永続システムのように、オブジェクト指向を用いてクラス毎にデータ取得をカスタマイズするだけでは、効率のよいデータ取得を実現できない。効率のよいデータ取得を実現するためには、永続オブジェクトのリファレンスが辿られていく過程でアプリケーションの文脈に合わせてデータベースから適切にデータを取得するようにカスタマイズする必要がある。

そこで我々は、アスペクト指向による高速化を支援するための永続システム AspectualStore を提案する。AspectualStore を利用することで開発者は、クラス毎のカスタマイズだけではなく、どの永続オブジェクトからの参照か、どのメソッド処理かといったアプリケーションの文脈に応じた、クラス横断的な最適化を指示することができる。また、永続システムへの最適化の指示がアプリケーション内に散在することなく、アスペクトとして分離して記述出来るようになる。アスペクト指向による高速化を実現するにあたって AspectualStore では、アスペクト指向を用いて容易に最適化出来る構造を提供している。まず、外部キー等のデータベースに関わる作業を意識することなく永続オブジェクトまたはそのコレクションに対し

ての最適化の指示をだす API を用意した。これらの API を利用することで永続システムに対して直感的かつ容易に最適化の指示を出せるようになる。また、最適化の指示を出すためには、対象となる永続オブジェクトがどのようなリファレンス辿ってきたのか、どのようなデータベース・アクセスを引き起こしたのかといったアプリケーションの様々な文脈が必要となる。AspectualStore では、アスペクトによって容易に最適化を実現出来るよう、アスペクト内でこれらの文脈情報を格納するための機構や、アスペクトのウィーブ/アンウィーブにかかる手間を削減するためのロードタイム・ウィービング等の機構を用意している。

謝辞

本研究を支えていただいた多くの方々への感謝の意をここでは表したいと思います。東京工業大学　千葉滋助教授には、私が千葉研究室に所属した 3 年間の間、様々な面でご指導いただきました。本研究を進めるにあたり、研究の方向づけから論文の組立て方、発表資料に至るまで多岐にわたってのご指導して頂きました。深く感謝いたします。

日立製作所の佐藤芳樹氏には、本研究の当初から研究に必要な基礎的な知識や関連研究について様々なご意見をいただきました。心より感謝いたします。千葉研究室の西沢無我氏には、多くの疑問を聞いて頂き、指導して頂きました。柳澤佳里氏にはデータベースや実験環境に関する様々な助言を頂きました。また、東京工業大学　光来健一助手には、研究内容や研究発表について様々な意見をいただきました。最後に、本研究に関して貴重な意見をくださった千葉研究室の方々に心から感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景

J2EE サーバなどの大規模な Java システムにおいてデータベース・アクセスは必須の機能となっている。データベースには多くの管理方法があるが、その中でデータの蓄積量やパフォーマンス、コスト面で最も優れているとされているのがリレーショナルデータベース (以下データベース) である。Java 言語におけるデータベースを利用したアプリケーション開発には、EJB2 に代表される永続システムが利用されてきた。ここで言う永続システムとは、オブジェクト指向システムとデータベースとのインピーダンス・ミスマッチを自動的に吸収してくれるシステムのことを指す。オブジェクト指向システムとデータベースの間には、両者の設計の違いからデータ間の関係やデータ構造の表わし方・継承やポリモノフィズムのありなしといった様々な違いが存在する。このような両者の違いは、O/R 間のインピーダンス・ミスマッチと呼ばれる。例えば、Java 言語でデータベースのテーブル・レコードをオブジェクト (永続オブジェクト) として扱うには、SQL の文字列を記述してデータを取得し、取得したデータからオブジェクトを生成するといった煩雑な作業が必要である。一方で、永続システムでは初めに永続オブジェクト (ルート・オブジェクト) を取得する作業以外は、永続オブジェクトのリファレンスを辿っていく際に自動的に SQL を発行し必要なオブジェクトを生成してくれる。そのため、永続システムを利用することで開発者はインピーダンス・ミスマッチを意識する必要が無く、アプリケーションの開発効率を高めることが出来る。

1.2　研究の目的

このように永続システムを利用することでアプリケーションの開発効率を格段に高めることが出来るにも関わらず、今日、永続システムは今ひとつ利用されていないという現状がある。これはオブジェクトのリファレンスが参照される度毎にいちいち SQL を発行していたのでは、効率が悪すぎるためである。アプリケーションの高速化を図るためには、効率よくデータベースからデータを取得することが重要である。効率のよいデータ

取得とは、アプリケーションに必要なデータのみを少ないデータベース・アクセスで取得することを指す。

1.2.1　クラス毎のデータ取得

既存の永続システムには、オブジェクト指向によってクラス毎にデーターベースから取得するデータをカスタマイズする方法が提供されている。しかし、取得データをクラス毎にカスタマイズするだけでは、アプリケーションの文脈に応じた効率のよいデータ取得を実現することが出来ず、アプリケーションの高速化を図ることが難しい。永続オブジェクトから実際にどのようなリファレンスが辿られていくかはアプリケーションの文脈によって異なってくるためである。例えば、どのメソッド内でそのオブジェクトが実行されているかといったアプリケーションの文脈に応じて、次にアプリケーションで利用される可能性が高いデータを先読みすることで効率のよいデータ取得を実現出来る。

しかし、クラス毎のデータ取得の指示だけではこのようなカスタマイズは出来ない。あるクラス X のオブジェクトを取得する際にはその X に関連する Y クラスのオブジェクトも必ず取得するといったクラス毎の指示だけでは、不十分である。例えば、X のオブジェクトから Z クラスのオブジェクトが繰り返し参照された際には、過剰なデータベース・アクセスが発生してしまう。また、このような過剰なデータベース・アクセスを防ぐために X に関連する全てのオブジェクトを取得する指示を出してしまうと、今度は不必要なデータを大量にデータベースから取得してしまう危険性がある。このような大量なデータ取得は、データベースとの通信のオーバーヘッドやアプリケーションのメモリ容量への圧迫を引き起こしパフォーマンスを低下させてしまう。

1.2.2　既存の永続システムへのアスペクト指向言語の適用

既存の永続システムには、開発者が直接 SQL を記述して必要なデータをデータベースから取得出来るものがある。しかし、既存の永続システムとアスペクト指向言語を組み合わせるだけでは、最適化の指示に手間がかかりすぎてしまい現実的ではない。

データベースから取得された永続オブジェクトに対してデータ取得の指示を直接 SQL で記述しようとすると、そのオブジェクトに対応する外部キーの名前や値といったデータベース・テーブルに関わる情報を基に煩雑な SQL を記述しなければならない。本来の永続システムでは、このようなデータベースに関わる情報 (*shadow information*) は永続システムに

よって管理されており、容易に取得することは出来ない。また、永続オブジェクトに対する SQL はそのオブジェクトがどのようにリファレンスを辿ってきたかによって記述内容が異なってくる。リファレンスの辿られ方によっていちいち SQL を記述するのは現実的ではない。さらに、既存の AOP ではリファレンスの辿られ方といった時間的に素なデータ・アクセスを取得することは想定されておらず、取得には煩雑な記述が必要となってしまう。

1.2.3　O/R mapper によるデータ取得のカスタマイズ

近年、永続システムに代わってよく利用されている O/R mapper では、ルート・オブジェクトの取得をカスタマイズする API が用意されている。原理的には、この API を用いてアプリケーションで必要なオブジェクトを全てルート・オブジェクトに含めることで効率のよいデータ取得を実現することが出来る。

しかし、この方法は現実的ではない。ルート・オブジェクトを取得する際に、そのルート・オブジェクトからどのような永続オブジェクトのリファレンスが辿られていくかを予測するのは困難であるためである。オブジェクト指向によるモジュール化やユーザ・インタラクションがこの予測をより難しくする。効率よいデータ取得を実現するためには、永続オブジェクトのリファレンスが辿られていく過程で、アプリケーションの文脈に応じてデータベースから適切にデータを取得するようにカスタマイズする必要がある。例えば、ルート・オブジェクトからのリファレンスの辿られ方に応じて、次のデータ取得を予測し先読みすることで、効率よいデータ取得を実現できる。

1.3　研究成果

そこで我々は、アスペクト指向を用いた永続システムの高速化を提案する。また、アスペクト指向による高速化を支援するために、Java 向けの永続システム AspectualStore の開発を提案する。AspectualStore を利用したアプリケーションの高速化には、アスペクト指向を用いてアプリケーションの文脈に応じた最適化 (データ取得) を永続オブジェクトに指示する。AspectualStore を利用することで開発者は、クラス毎のカスタマイズだけではなく、どの永続オブジェクトからの参照か、どのメソッド処理かといったアプリケーションの文脈に応じた、クラス横断的な最適化を指示することができる。また、永続システムへの最適化の指示がアプリケーション内に散在することなく、アスペクトとして分離して記述出来るよう

になる。AspectualStore では、アスペクト指向を用いて容易に最適化出来る構造を提供している。

まず、永続オブジェクトまたはそのコレクションに対しての最適化の指示をだす API を用意した。開発者が発行する SQL を直接記述しようとすると、外部キーの名前やその値といった永続クラスとデータベースとのマッピング定義を元に煩雑な記述を行わなければならない。また、このような記述は、永続オブジェクトがどのようにリファレンスを辿ってきたかによって異なってくる。AspectualStore では、内部で自動的にオブジェクトがどのようにリファレンスを辿ってきたかを管理している。そのため、開発者は AspectualStore が提供する API を利用することで永続オブジェクトに対して直接的かつ容易に最適化の指示を出せるようになる。

また、最適化のアルゴリズムを決定する際には、指示を出す永続オブジェクトがどのようなリファレンス参照によって取得されたのか、どのようなデータベース・アクセスを引き起こしたのかといったアプリケーションの様々な文脈が重要となる。AspectualStore では、最適化に必要な文脈情報を容易にアスペクト内から利用出来るよう、アプリケーションの文脈情報を格納するための機構を用意している。開発者は、AspectualStore を拡張することによって必要な文脈を容易に利用出来る。AspectualStore 自身もこの機構を利用して永続オブジェクトがどのようにリファレンスを辿ってきたかを内部で管理している。

加えて、アスペクトのウィーブ/アンウィーブのかかる手間を削減するためのロードタイム・ウィービング等の機構を用意している。開発者は、AspectualStore を拡張することでアプリケーションの高速化を図る。ロードタイム・ウィービングをサポートすることで、いちいち AsepctualStore の JAR ファイルを作り直すことなく AspectualStore を拡張することが出来る。AspectualStore では、Tomcat と連結して利用することを想定して Tomcat 用のクラスローダを提供している。

最後に、予備的な実験から AspectualStore を利用してアスペクト指向によるクラス横断的な最適化を施すことによって、オブジェクト指向によってクラス毎の最適化を行う既存の永続システムに比べて実行速度、メモリ消費量およびデータベースへの通信回数を改善できることを確認した。ただし、SQL を直書きする方法程の高速化は実現出来なかった。AspectualStore では、あくまでデータベースから効率よくデータを取得することによって高速化を図ることを目標としているためである。そのため、SQL を直書きしてアプリケーションの挙動を変更するような高速化を図ることは出来ない。その一方で、最適化の記述によってアプリケーションのバグが発生する危険性はない。

1.4　本論文の構成

本稿の残りは、次のような構成からなっている。第 2 章は、研究の背景として、Java 言語における永続手法と永続システムによる既存の高速化技術の問題点について述べる。第 3 章では、アスペクト指向によるアプリケーションの文脈に応じた高速化支援するための永続システム AspectualStore の提案を行う。第 4 章では、AspectualStore の実装について述べ、第 5 章では、AspectualStore と既存技術との比較および予備的な実験、関連研究について述べる。最後に第 6 章でまとめを述べる。

第2章　Java 言語における永続化と最適化にかかるコスト

本章ではまず、研究の背景として Java 言語における永続化について述べる。Java 言語での永続化処理には、EJB2 [21] に代表される永続システムが利用されてきた。永続システムでは、永続オブジェクト (永続化されたオブジェクト) へのリファレンスを参照する際に自動的に SQL を発行し参照されたオブジェクトを生成してくれるためアプリケーションの開発効率を格段に高めることが出来る。また、本章では永続システムについて述べると共に、永続システムを利用したアプリケーションの高速化の重要性とオブジェクト指向によってクラス毎にデータベースから取得するデータをカスタマイズする既存手法だけでは、効率のよい高速化が困難であることを述べる。

2.1　Java 言語における永続化

この節では、 Java 言語における永続化技術について述べる。Java 言語におけるリレーショナルデータベース (以下データベース) を利用したアプリケーション開発には、EJB2 [21] に代表される永続システムが利用されてきた。ここで言う永続システムとは、オブジェクト指向システムとデータベースとの間の違い (O/R 間のインピーダンス・ミスマッチ) を自動的に吸収してくれるシステムのことを指す。永続システムでは、永続オブジェクトのフィールド・アクセスに合わせて煩雑な SQL を自動的に発行してくれるため、開発者はデータベースを意識する必要が無く開発効率を高めることが出来る。以下、インピーダンス・ミスマッチおよび永続システムについての説明と永続システムの有用性について述べる。

2.1.1　O/R 間のインピーダンス・ミスマッチ

オブジェクト指向とデータベースとの間には、両者の設計の違いから様々な違いが存在する。オブジェクト指向では、処理やデータをオブジェクトして扱い、アプリケーションを構築していく。一方、データベースで

は全てのデータをデータベース・テーブルと呼ばれる表に格納し、テーブル同士の結びつきによって大量のデータを管理する。このような両者の設計の違いを O/R 間のインピーダンス・ミスマッチと呼ぶ。例えば、SQL にはオブジェクト指向の継承やカプセル化といった概念は存在しない。オブジェクト指向システムからデータベースを操作するためには JDBC (Java Database Connectivity) [22] を利用して SQL の文字列を記述する必要がある。

インピーダンス・ミスマッチを解消するために、オブジェクト指向システムとデータベースとの連結には煩雑な作業が必要となる。例えば、図 2.1 は、ある文献検索システム内で利用されるデータベース・テーブルとそのテーブルに対応する永続クラス (データベース・テーブルと対応づけられたクラス) の例である。データベースには、論文情報が格納された `paper` テーブルおよび著者情報が格納された `author` テーブルが定義されており、両テーブルは外部キー `paper.author_id` と主キー `author.id` によって関連付けられている。また、アプリケーション側にはデータベースの `paper / author` テーブルに対応する `Paper` クラス / `Author` クラスが定義されており、両クラスは、`Paper` クラスの `author` フィールドによって関連付けられている。

オブジェクト指向では、オブジェクトのフィールドを参照することで関連する他のオブジェクトを取得出来る。例えば、今 図 2.1 の `paper` テーブルの黒塗りのレコード (id = 2) に対応する `Paper` オブジェクトがアプリケーションで操作されているとする。このとき、オブジェクト指向ではこの `Paper` オブジェクトに関連する `Author` オブジェクトを取得するには、`Paper` オブジェクトの フィールドである `author` フィールドを参照すればよい。つまり、`Paper.getAuthor()` メソッドを呼び出せばよい。それに対して、データベースではテーブルの外部キーを元に SQL を発行することによって関連するテーブルのレコード群を取得する。つまり、`paper` テーブルと `Paper` クラスをマッピングさせるためには、`author` フィールドにアクセスがあった際に、そのオブジェクトに対応するテーブル・レコードの外部キーを基に

```
SELECT id FROM author t0 WHERE t0.id = 98
```

といった SQL を発行する作業が必要となる。ここで 98 は、`paper` テーブルの黒塗りのレコード (id = 2) に格納されている外部キー `author_id` の値である。図 2.1 は、もっとも単純なマッピング例であるが、`Author` クラスが `Document` クラスを継承しており、`Document` クラスに対応する `document` テーブルがある場合等には、より複雑な SQL の発行や `Author` オブジェクトの生成が必要となる。

また、SQL の作成以外にも様々な作業が必要である。データベースを利用したアプリケーションでは、オブジェクトのリファレンスが参照される度に、データベースとのコネクションの取得、SQL を発行、SQL の結果からオブジェクトを生成するといった煩雑な作業が必要となる。それ以外にもトランザクションの管理やデーベース・コネクションの管理といった作業がある。このように、オブジェクト指向システムとデータベースとの連結には、O/R 間のインピーダンス・ミスマッチを解決するために煩雑な作業が必要となり、開発効率を低下させる原因となっていた。

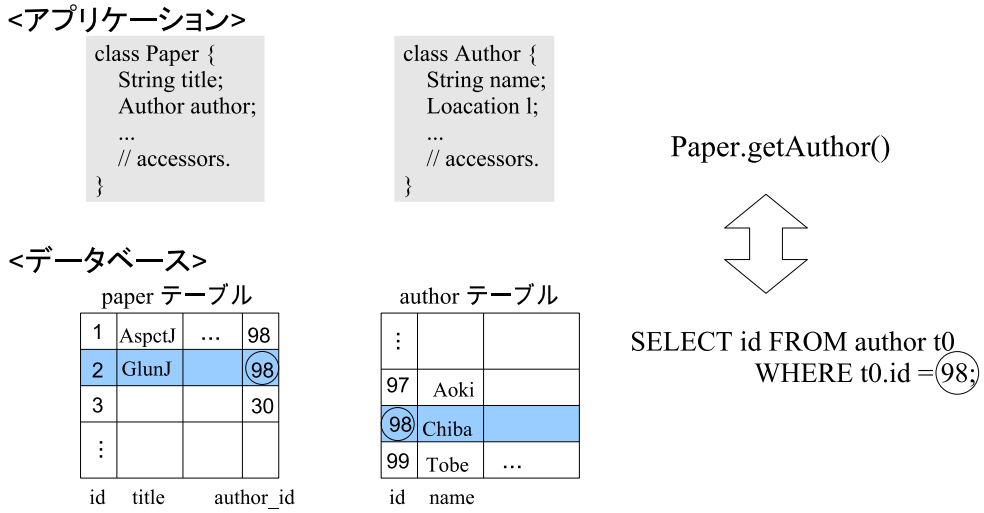

図 2.1: クラスとテーブルのマッピング

2.1.2　永続システム

永続システムとは、前節で述べた O/R 間のインピーダンス・ミスマッチを吸収し、データベースに関わる煩雑な作業を自動化してくれるシステムのことを指す。オブジェクト指向システムとデータベースのマッピング作業でもっとも手間がかかるのは、SQL の発行および発行した SQL の結果からのオブジェクトの生成である。永続システムでは、外部のマッピング・ファイルもしくはアノテーション等を利用して予め永続クラス (永続化されるクラス) とデータベース・テーブルとを対応づける定義を与えておくことで、永続オブジェクト (永続クラスのオブジェクト) のリファレンスが参照された際に自動的に SQL を発行し、オブジェクトの生成を行ってくれる。

永続システムでのデータベース・アクセスは、永続システムによって最初に取得された永続オブジェクトから、他の永続オブジェクトへのリファレンスをたどる際に透過的に行われる。永続システムから最初に取得される永続オブジェクトのことをルート・オブジェクト という。ルート・オブジェクトを取得する際には何らかの指示が必要である。しかし、ルート・オブジェクトを取得さえしてしまえば、その後は永続オブジェクトへのリファレンスをたどる際に永続システムが自動的にデータベースからのデータ取得を行ってくれるため、アプリケーションの開発者はデータベースを意識することなく開発を行うことが出来る。

例えば、図2.2 は、永続システムを利用した場合のアプリケーション・コードの一部である。showPaperList() メソッドでは、引数として与えられた Paper オブジェクトの一覧を表示するためのメソッドである。このメソッド内部で Paper.getAuthor() や Paper.getTitle()、Author.getName() メソッドが呼ばれた際には、このメソッド呼び出しに対応する SQL をデータベースに発行して必要なデータを取得してくる必要がある。

```
01   void showPaperList(List<Paper> papers) {
02       Iterator it = papers.iterator();
03       while(it.hasNext()) {
04           Paper p = (Paper)it.next();
05           show(p);
06           ...
07       }
08   }

10   void show(Paper p) {
11       Author a = p.getAuthor();
12       String title = p.getTitle();
13       String name = a.getName();
14       /* p.title と a.name を表示 */
     }
```

図 2.2: 永続システムのコード

しかし、見てわかるように、このコードにはデータベースに関わる処理は一切登場しない。これは、永続システムが内部で自動的に Paper.getAuthor() メソッドが呼ばれた際に適切な SQL を発行し Paper オブジェクトを生

成してくれるためである。このように永続システムを利用すると、開発者はデータベースを意識する必要がないためアプリケーションの開発効率を格段に高めることが可能である。

また、オブジェクトのフィールド・アクセスに合わせて適切な SQL を発行するためには、そのオブジェクトに対応するテーブル・レコードの主キー (primary key) や外部キー (foreign key) の名前はその値を管理しておく必要がある。しかし、主キーや外部キーは本来クラスには定義するべきでないものである。つまり、データベース・テーブルには、他のテーブルとの関連付けのために外部キーや主キーが必要となってくるが、クラスとしては外部キーや主キーといった情報は必要ない。代わりに対応する永続クラスのフィールドさえ定義してあればよい。このようにクラスに定義する必要の無い情報は *shadow information* と呼ばれる。そのため、永続システムでは永続クラスの shadow information を隠蔽してくれる。つまり、主キーや外部キーといった情報は、永続システムが内部で自動的に管理してくれるために永続クラスにはこれらをフィールドとして定義する必要は無く、POJO クラスとして扱うことが出来る。永続クラスを POJO として定義できるために、クラスは完全にデータベースから切り離して定義できアプリケーションの再利用性を高めることが出来る。

2.1.3　文献検索システム

ここでは、本稿全体を通してのアプリケーション例として登場する文献検索システムについて説明する。文献検索システムは、DBLP サーバのような論文やジャーナルの情報が格納されており、これらの文献情報を追加・検索できるアプリケーションを想定している。

図 2.3 に文献検索システムで管理する情報を格納しているデータベースのテーブル群を示す。関連するテーブル同士は外部キー (もしくは主キー) によって関連づけられている。主キーとは、テーブル・レコード (ロー) を特定するための値が格納されているカラムのことを指し、外部キーとは、他のテーブルとの関連づけを行うために用意されているカラムのことを指す。例えば、`bibliography` テーブルと `proceeding` テーブルは、`bibliography` テーブルの主キー `id` と `proceeding` テーブルの外部キー `bibliography_id` によって関連づけられている。また、２つのテーブル同士の関係は 1 対多の関係となっている。`bibliography` テーブルと `proceeding` テーブルが 1 対多の関係であるとは、`bibliography` テーブルの１つのテーブル・レコード (ロー) に対して `proceeding` テーブルの複数のテーブル・レコードが対応しているという意味である。テーブル同士の関係には、1 対 1 、1 対多、多対多の関係がある。それぞれのテーブル同士の関係は図 2.3 に記述した通りである。

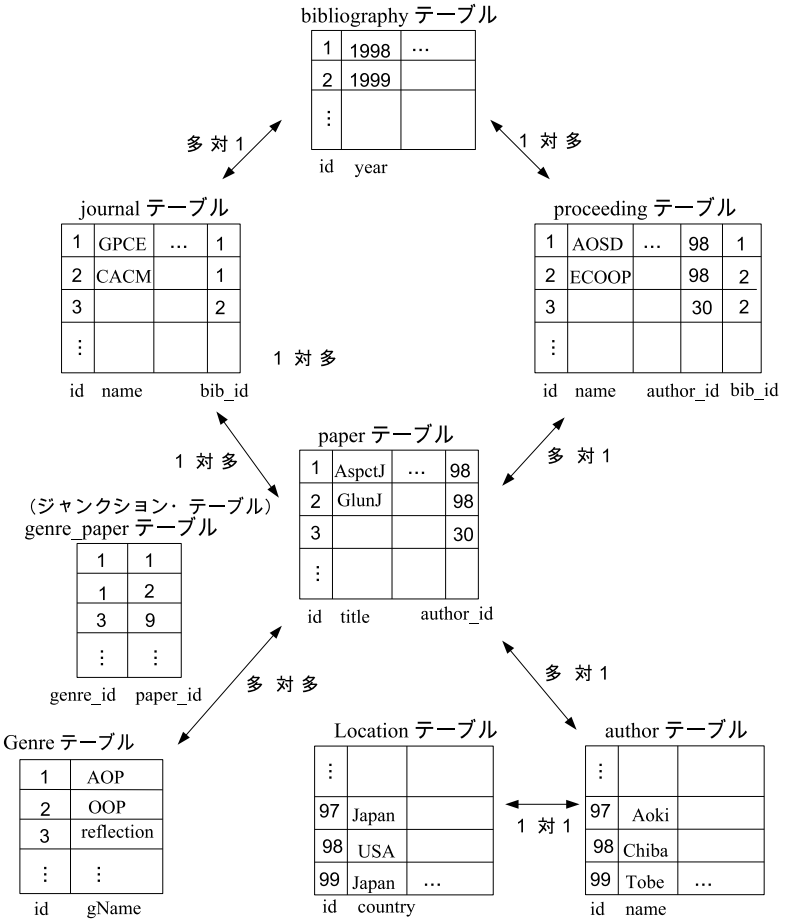

図 2.3: 文献検索システムのテーブル群

なお、このテーブル設計の例では、各テーブルは主キー `id` を持ち、テーブル同士に 1 対 1 の関係は、2 つのテーブルの主キーによって関連づけられているものとする。また、1 対多の関係は主キー (id) と外部キー (相手のテーブル名_id) によって関連づけられ、多対多の関係はジャンクション・テーブルを介して関連づけられるものとする。例えば、多対多の関係を持つ `paper` テーブルと `genre` テーブルは、ジャンクション・テーブルである `paper_author` テーブルを介して関連づけられている。ジャンクション・テーブルとは 2 つのテーブル間の関係を定義するために作成されたテーブルのことを指す。

次に、図 2.3 のテーブル群に対応する永続クラスの UML を図 2.4 に示す。各クラスは、データベース・テーブルと対応づけられている。例えば、図 2.4 の `Bibliography` クラスは、図 2.3 の `bibliography` テーブルに

対応しているものとする。テーブル同士の関連は、永続クラスのフィールドとして表される。対応するテーブルが他のテーブルと 1 対 多もしくは多対多の関係を持っている場合には、フィールドの型はコレクション型となる。１つのオブジェクト (テーブル・レコード) に対応するオブジェクト (関連するテーブル・レコード) が複数行あるためである。一方、テーブルが 1 対 1 もしくは 多対 1 の関係を持つ場合には、フィールドの型は対応するテーブルを表す永続クラス型となる。

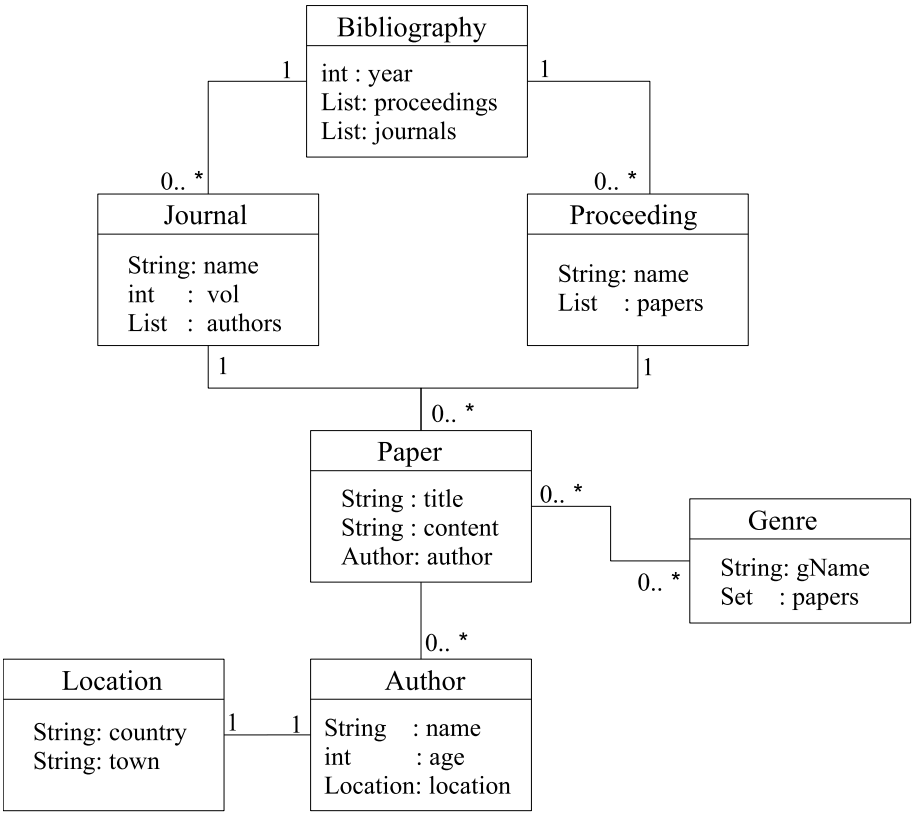

図 2.4: 文献検索システムの永続クラス群

2.2　アプリケーション高速化の困難さ

この節では、既存の Java 向け永続システムによるオブジェクト指向を用いたクラス毎の最適化の指示だけでは、アプリケーションのパフォーマンスを向上させることが難しいことを述べる。次に、アプリケーションの高速化を図るための永続システムの代替システムである O/R mapper について述べる。また、O/R mapper による最適化の方法では、十分な最

適化の指示を出すことが困難であることを述べ、既存の O/R mapper とアスペクト指向システムを連携させるだけでは、問題が解消されないことを述べる。

2.2.1　永続システムの問題点

前節で述べたように永続システムを利用するとアプリケーションの開発効率を格段に高めることが出来る。それにも関わらず、EJB2 等の永続システムは今日、今いちはやっていないという現状がある。これは、永続システムを利用した開発ではアプリケーションのパフォーマンスが悪すぎることが原因である。永続システムのような抽象度の高いシステム (つまり、アプリケーションからデータベースが全く見えないシステム) では、効率的な SQL を自動的に発行することは難しい。永続システムでは、永続クラスへのリファレンスが参照された際にクラス毎に静的に決定されたデータ取得を行うことで透過的なデータ取得を実現する。しかし、このようなオブジェクト指向によるクラス毎の最適化の指示だけでは、次のアクセスを予測して先読みするといった効率のよいデータ取得を実現することが出来ず、過剰なデータベース・アクセスや不必要なデータ取得によってアプリケーションのパフォーマンスが低下してしまう。

データベースを利用したアプリケーションの高速化を図るためには、データベースとの通信回数を出来るだけ抑えながら、アプリケーションに必要となるデータを効率よく取得することが重要である。前節の図 2.2 で説明した永続システムのコード `showPaperList()` メソッドについてもう一度考える。この `showPaperList()` メソッド内の `while` 文内では、`show()` メソッドが呼び出されている。また、`show()` メソッド内では `Paper.getAuthor()` や `Author.getName()` といったデータベース・アクセスを引き起こすメソッドが何度も呼ばれることがわかっている。つまり、このままでは `while` 文が回るたびごとに SQL が発行されることになってしまい、過剰なデータベース・アクセスによってパフォーマンスが著しく低下してしまう (図 2.5)。

このような場合、`showPaperList()` メソッド内で利用されるデータをデータベースから一括して先読み取得しておくことによって SQL の発行回数を大幅に削減することが可能である。また、JDBC [22] 等によって自分で SQL を発行するのであれば、そのような SQL を発行するのが当然である。しかし、オブジェクト指向によるクラス毎に決定されたデータ取得では、このようなクラスを跨った文脈を理解することは出来ない。つまり、今発生している SQL 発行が `showPaperList()` メソッド内で引き起こされたものであるといった情報や Paper オブジェクトが `papers` リストの一要素であるといった情報を理解することが出来ない。その結果として

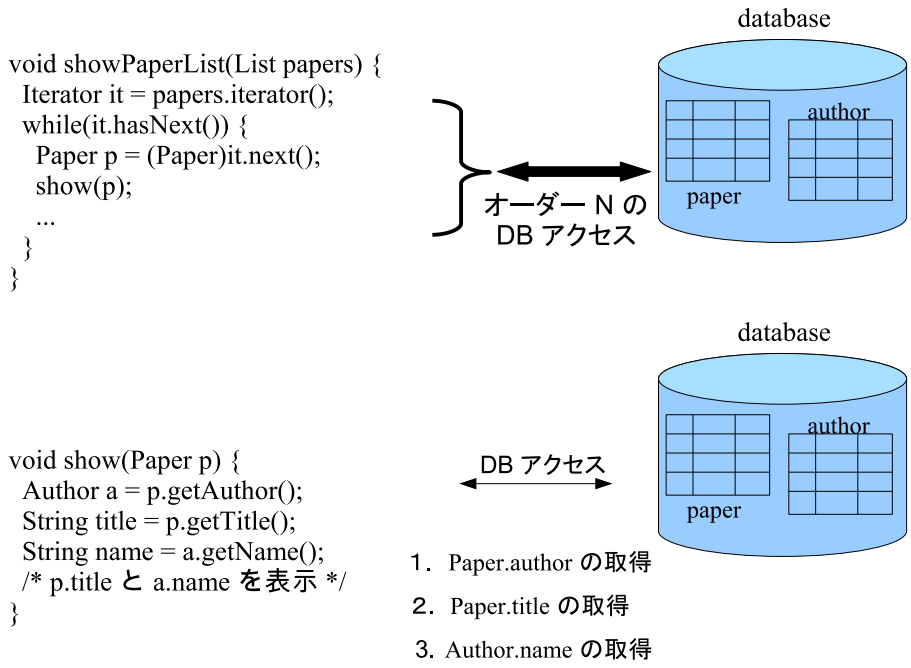

図 2.5: 過剰なデータベース・アクセス

N+1 問題などの過剰な SQL 発行を引き起こしてしまいパフォーマンスの低下を引き起こしてしまう。

一方、オブジェクト指向によるクラス毎の静的な先読みの指示では、不必要なデータを大量に取得することでアプリケーションのメモリ容量を圧迫してしまう危険性がある。例えば、`Paper` オブジェクトをデータベースから取得する際には、関連する `Author` オブジェクトも一括して取得するといったクラス毎の指示を出すことで、過剰な SQL 発行を防ぐことが出来る場合もある。しかし、このような指示では、Paper オブジェクトの `author` フィールドにアクセスが無い場合でも必ず `Author` オブジェクトが取得されることになってしまう。データベースには大量なデータが格納されている。不用意に必要のないデータ取得を指示してしまうと、すぐにアプリケーションのメモリ容量を圧迫してしまう。このような、不必要な永続オブジェクトを取得は、大量のメモリ使用によって逆にパフォーマンスが低下してしまう危険性もある。

2.2.2　O/R mapper による高速化

アプリケーションの高速化を図りたいが為に、近年永続システムに代わってよく利用されているシステムが O/R mapper である。O/R mapper の代表的なものとしては、 Hibernate [11], Cayenne [3], Torque 等がある。永続システムでは、データベースからのデータ取得は永続オブジェクト (永続化されたオブジェクト) のリファレンスをたどる際に透過的に行われた。しかし、クラス毎に静的に決定されたデータ取得だけでは不必要なデータ読み込みや冗長なデータベース・アクセスによってアプリケーションのパフォーマンスが低下してしまう。そこで、O/R mapper には、永続システムのような透過的なデータベース・アクセスに加えて、ルート・オブジェクトまたはルート・オブジェクトのコレクションの取得を明示的にカスタマイズ出来る API や任意の SQL を発行するためにローレベルの API が用意されている。例えば、Hibernate には、HQL と呼ばれる独自のクエリ言語が提供されている。開発者がアプリケーションに明示的に HQL の文字列を記述してやることで任意の永続オブジェクトをルート・オブジェクトとして出得することが出来る。

図 2.6 のコードは、図 2.2 を Hibernate が提供している API によって最適化したコードである。O/R mapper によって提供されたローレベルの API (HQL) を使って明示的に必要なデータをルート・オブジェクトに含めてやることによってアプリケーションの高速化を図ることが出来る。つまり、ルート・オブジェクトを取得する際に明示的にアプリケーションで必要となるデータをデータベースから一括して取得する先読み記述を与えることで過剰な SQL 発行や余分なデータ取得を防ぎ高速化を実現する。図 2.6 の 02-06 行目に記述されている文字列は、ルート・オブジェクトとして `Proceeding` オブジェクトを取得するための記述である。また、`Proceeding` オブジェクトを取得する際に、この `Proceeding` オブジェクトの `papers` フィールドに格納される `Paper` オブジェクト (の List) とその `Paper` オブジェクトの `author` フィールドに格納される `Author` オブジェクトを同時に取得しろという記述が 03・04 行目に記述されている。以上のような HQL を記述することによって、09 行目で Hibernate からルート・オブジェクトとして `Proceeding` オブジェクトを取得する際は、関連する `Paper` オブジェクトおよび `Author` オブジェクトが先読みされているために、12 行目で showPaperList メソッドが呼び出された際にも図 2.5 のような過剰な SQL 発行が起きる危険性はない。

原理的には、ルート・オブジェクトを取得するためのクエリに先読みの指示を追加してやることで、アプリケーションの高速化を図ることが出来る。しかし、この方法は現実的ではない。というのは、ルート・オブジェクトを取得する際に、どのような先読みを指示すればよいかを決定するの

```
01  // HQL の記述
02  String query = "FROM Proceeding p
03      LEFT JOIN FETCH p.papers
04      LEFT JOIN FETCH p.papers.author
05      WHERE p.name = 'AOSD'
06      and p.bibliography_id = 2";
07  Query q = session.createQuery(query);
08  // ルート・オブジェクトの取得
09  Proceeding p = q.load();
    ...
10  List papers = p.getPapers();
11  // 論文リストの表示
12  showPaperList(papers);
    ...
```

図 2.6: Hibernate による最適化コード

は困難であるためである。先読みを決定するためには、ルート・オブジェクトからどのような永続オブジェクトのリファレンスが辿られていくかを正確に予測する必要がある。しかし、モジュール化されているオブジェクト指向アプリケーションにおいてこのような予測は大変難しい。例えば、showPaperList() メソッド内で呼び出されている show() メソッドによってどのような永続オブジェクトが呼び出されるかを予め把握することは難しい。

また、アプリケーション内にユーザのアクションが含まれる場合には、予めアプリケーションの挙動を完全に予測することはさらに難しくなる。例えば、文献検索システム (2.1.3 節) の例では、図 2.6 で、showPaperList() メソッドの呼び出しによって論文のタイトルと著者名の一覧が表示された後の挙動は、アプリケーションのユーザがどの論文もしくは著者に興味を持つかによって異なってくる。

2.3　アプリケーションの文脈を利用した高速化

この節では、アプリケーションの文脈に応じて永続オブジェクトに最適化の指示を出すことによるアプリケーションの高速化の方法について述べる。また、このような指示を出すためには、データベースに関わる指示が

アプリケーションに散在してしまい、アプリケーションの可読性・保守性に問題があること、これらの指示を既存のアスペクト指向言語で分離しようとしても、先読みを行うために重要な対象となる永続オブジェクトがどのオブジェクトのリファレンス参照によって取得されたのか、どのようなデータベース・アクセスを引き起こしたのかといった情報を取得するために煩雑な記述が必要となってしまい現実的ではないことについて述べる。

データベースを利用したアプリケーションの高速化を図るには、アプリケーションの文脈情報に応じて最適化の指示を出してやることが重要である。2.2.2 節で述べたように、先読等の最適化の指示をルート・オブジェクトが取得される際に記述することは難しい。アプリケーションの最適化の指示は、アプリケーション内で永続オブジェクトのリファレンスが辿られていく過程で、アプリケーションの文脈に応じて永続オブジェクトもしくは永続オブジェクトのコレクションに対して最適化のヒント (どのようなデータを取得するか) を出していく必要がある。アプリケーションの文脈とは、どのメソッドによる処理が行われているのか、永続オブジェクトがどのようなリファレンスを辿って取得されたのか、どのようなデータベース・アクセスを引き起こしたのかといった情報のことを指す。

例えば、 2.1.3 節の図 2.4 で登場する `Author` クラスについて考えてみる。今、`Author` オブジェクトの `name` フィールドがアプリケーションから参照されたとする。このとき、発行すべき最適な SQL は `Author` オブジェクトの状態によって異なってくる。もし `Author` オブジェクトがコレクションの一要素である場合 (図 2.7 の CASE:1) には、`Author` オブジェクトの `name` フィールドへのアクセスがコレクションの全ての要素に対して同様に行われる可能性がある。このような場合には、コレクション内の全ての `Author` オブジェクトの `name` フィールドの値を取得するような SQL を発行すべきである。ただし、実際にコレクション内の多くの要素に同様のアクセスがあるかどうかは、どのようなメソッドから呼び出されているかといった他の文脈に依存する。一方 図 2.7 の CASE:2 のような場合には今のような考慮は必要ない。以上のようなアプリケーションの文脈情報をうまく利用しながら、リファレンス参照によって取得された永続オブジェクトに最適化の指示を与えることによってアプリケーションの高速化を実現することが出来る。

2.3.1　ローレベルの API を利用した高速化

この節では、任意の SQL を発行できる ローレベルの API を提供するだけでは、永続オブジェクトに対する最適化の指示を出すために膨大な手間がかかることについて述べる。データベースを利用したアプリケーションの高速化を図るためには、アプリケーションの文脈に応じて永続オブ

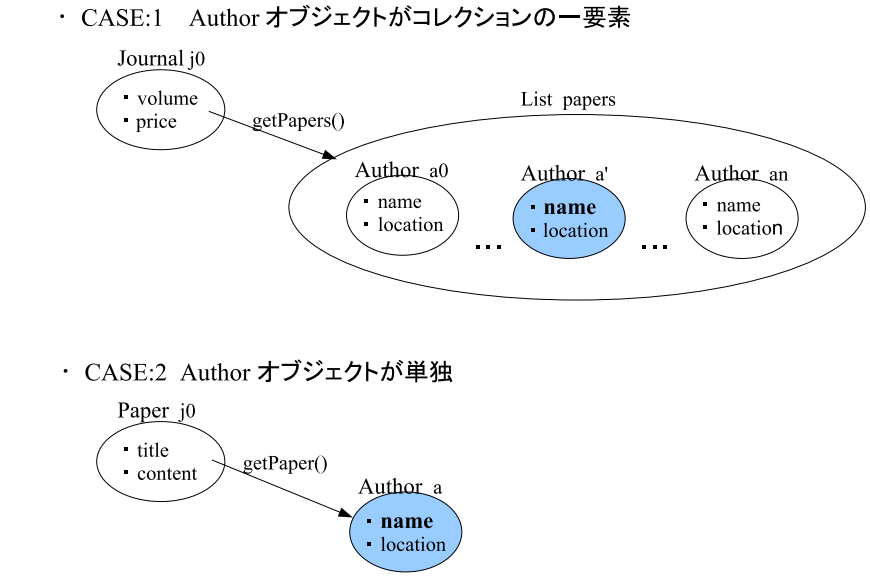

図 2.7: データベース・アクセスパターンの例

ジェクトや永続オブジェクトのコレクションに対して最適化の指示を出すことが重要である。O/R mapper には、任意の SQL を発行するためのローレベルの API が提供されている場合がある。この API を利用すれば文脈に応じた SQL を発行することが可能である。しかし、SQL の発行には外部キーの名前や値といったデータベースに関わる情報が必要となり、大変煩雑な SQL の記述を行わなければならない。また、このようなデータベースに関わる指示が散在し、アプリケーションのモジュール化を破壊してしまう危険性がある。

永続オブジェクトおよびそのコレクションに対しての指示がどのように煩雑であるかについて述べる。 2.1.3 で登場する図 2.4 の文献検索システム内の論文を表す永続クラスである `Paper` クラスについて考えてみる。今、下図のように学会を表す `Proceeding` オブジェクト `prc` の getter によって `Paper` クラスのリファレンスが参照され `Paper` オブジェクトのリスト `papers` がデータベースから取得されたとする。本稿では、このとき、`prc` を `papers` の 親オブジェクト と呼ぶこととする。

```
Proceeding prc = ...;
List<Paper> papers = prc.getPapers();
...
```

リファレンス参照によって取得された永続オブジェクトや永続オブジェクトのコレクションに対して最適化のヒントを出すとは、例えば `Proceeding`

オブジェクトの getPapers() メソッドが呼ばれた時点で、透過的にデータベースから取得されるデータをを変更して「Paper オブジェクトをデータベースから取得する際に Author オブジェクトも join によって一括して取得しろ」といった先読みの指示を出したり、getPapers() によって取得されたリスト papers の要素である Paper オブジェクトの title フィールドが参照された時点で「papers に含まれる全ての要素に対して title フィールドを取得しろ」といった指示を出したりすることである。このような永続オブジェクトに対する指示は、永続クラスに対する指示と考えれば簡単な指示のように思える。しかし、ルート・オブジェクトからのリファレンスを辿って取得された永続オブジェクトに対して、実際にこのような指示に対応する SQL (もしくは SQL に類似したクエリ) を開発者が直接記述しようとするとデータベース・テーブルに依存する煩雑な記述を行わなければならない。永続オブジェクトに対して最適化の指示を出すためには、以下の作業が必要となる。これらの手間について以下で詳しく説明する。

- 発行したい SQL を記述するためにデータベース・テーブルの情報が必要である
- 発行する SQL を個別のケースに応じて書かなければならない
- SQL の結果をオブジェクトに割り当ててやる手間がかかる

例えば、再び図 2.4 の文献検索システム内の永続クラスである Paper クラスについて考えてみる。今、ある Paper オブジェクトのリスト papers があるとする。この papers に対して「papers に含まれる全ての Paper オブジェクトの author フィールドの値を取得する」といった先読みを実現する方法を考える。この指示を出すためにどのような作業が必要かについて述べる。

データベース・テーブル情報の取得

永続オブジェクトやそのコレクションに対して最適化の指示を出すためには、データベース・テーブルの情報が必要となる。今、papers の親オブジェクトが Proceeding オブジェクト prc であるとすると、発行する SQL は以下のようになる。
図 2.8 を見てわかるとおり、発行する SQL には author_id や proceeding_id といった paper テーブルや proceeding テーブルの主キーや外部キーといったデータベース・テーブルに関する情報 (shadow information) が必要となる。これらの値はそもそもオブジェクトしてはユーザから見える必要ない値であるため、O/R mapper が内部で管理していることが多い。例え

```
SELECT t0.id, t1.id, t1.title FROM paper
  INNER LEFT JOIN author t0 on t0.author_id = t1.id
  WHERE t0.proceeding_id = 10                          --- (  )
```

図 2.8: 最適化の指示に対応する SQL (1)

ば、図 2.4 の Proceeding クラスや Paper クラスには、主キー `id` や 外部キー `proceeding_id` といったフィールドは定義されていない。これらの値を取得するには、O/R mapper の内部構造を理解しておく必要がある。

個別のケースに応じた SQL 記述

次に、図 2.8 の SQL を発行するためには papers リストの親オブジェクトの情報が必要となる。つまり、papers リストに対して最適化のヒントを出すための SQL は、papers リストの親オブジェクトに応じて 発行する SQL を変更しなければならないことについて述べる。

「`papers` リストに含まれる全ての Paper オブジェクトの `author` フィールドを取得する」という指示に対応する SQL は、`papers` の親オブジェクトがどのクラスであるのかによって異なってくる。例えば、papers の親オブジェクトが Proceeding オブジェクトである場合に発行する SQL は、(I) のようになった。しかし、図 2.4 を見てわかるように Paper オブジェクトのリストは、Proceeding オブジェクト以外にも Journal オブジェクトや Genre オブジェクト、Author オブジェクト等の多くのオブジェクトから取得される可能性がある。

下図のように親オブジェクトが Genre オブジェクトである場合について考える。

```
Genre gne = ...;
List papers = gne.getPapers();
...
```

このとき、発行すべき SQL は、図 2.9 のようになる。
このように、親オブジェクトが異なるケース毎にいちいち SQL を記述しなければならないのは、大変手間のかかる作業である。例えば、図 2.10 は、図 2.2 の引数 `papers` の親オブジェクトが Proceeding オブジェクトである場合の最適化コードである。02 -05 行目の文字列が最適化のための指示であるが、このクエリは親オブジェクトごとにことなるため、このようなメソッドを 親オブジェクトの数だけ作成する必要がある。

SQL 発行後の処理にかかる手間

最後に、永続オブジェクトに対する最適化の指示を出す際には、SQL

```
SELECT t0.id, t1.id t1.title FROM paper
  INNER LEFT JOIN author t0 on t0.author_id = t1.id
  WHERE t0.id IN
  (SELECT t3.paper_id FROM genre_paper t3
  WHERE t3.genre_id = 98)                --- (  )
```

(ただし 98 は　Author オブジェクトが保持する主キー id の値)

図 2.9: 最適化の指示に対応する SQL (2)

```
01    void showPaperListFromProceeding(int fKey) {
02        String query = "SELECT paper.title, author.name
03            FROM Paper as paper
04            INNER JOIN paper.author Author as author
05            WHERE paper.proceeding_id = " + fKey;
06        Query q = session.createQuery(query);
          ...
   }
```

図 2.10: Hibernate による文脈ごとの最適化

の結果を適切なオブジェクトに割り当ててやる作業が必要であることを示す。多くの O/R mapper では、任意のルート・オブジェクトを取得する API が提供されていた。例えば、これらの API を利用して、下図のように papers リストに含まれる Paper オブジェクトの author フィールドの値を取得するためのクエリを記述することが出来たとする。

```
List papers = ...;
String query = "FROM Author author WHERE ...";
Query q = session.createQuery(query);
List authors = q.list();
```

しかし、これだけでは papers リストの要素である Paper オブジェクトの author フィールドは初期化されない。つまり、Paper オブジェクトの author フィールドがアプリケーションから参照された際には、新たに author フィールドの値を取得するための SQL が発行されてしまう。papers に対して最適化を施すためには、発行した SQL によって得られた結果 (authors リスト) を papers の要素である Paper オブジェクトに割り当ててやる作業が必要となる。このような作業は大変煩雑なものである。

また、これらの作業以外にも、低レベルの API ではクラスの変更に対応出来ないという問題点がある。多くの O/R mapper では、より細かな SQL の指示が出せるように外部キーや主キーといったデータベースに基づいた情報を利用して SQL の指示を出す。しかし、このような記述方法では、永続クラスとデータベース・テーブルとのマッピングが変更された場合に対応出来ない。例えば、永続クラスにフィールドを追加した (つまり、データベース・テーブルにカラムと追加した)　場合には、アスペクト内の最適化の指示を探しだし、SQL の文字列を書き直す必要が生じる。

2.3.2　最適化コードを分離する困難さ

この節では、既存の永続システムにアスペクト指向言語を適用しただけでは、2.3.1 節で述べたようなアプリケーションの文脈に応じた最適化のためのコードをアプリケーションからうまく分離できないことについて述べる。永続システムを利用したアプリケーションの高速化には、永続オブジェクトのリファレンスが辿られていく過程で、どのメソッドによる処理が行われているのか、永続オブジェクトがどのようなリファレンスを辿って取得されたのか、どのようなデータベース・アクセスを引き起こしたのかといったアプリケーションの文脈に応じて最適化のヒントを出すことが重要であった。そのためには、クラスを横断するような最適化の指示が必要である。しかし、このような指示はアプリケーションの文脈ごとに永続オブジェクトに指示を出さなければならないため、アプリケーション内に最適化の指示が散在してしまい、アプリケーションの保守性・可読性を損ねる危険性がある。また、既存の永続システムに AspectJ [9, 4] のようなアスペクト指向言語を適用しただけでは、アプリケーションの文脈情報を取得することは難しい。

アスペクト指向プログラミング (AOP) とは、横断的関心事 (crosscutting-concern) をモジュール化するための技術である。横断的関心事とは、オブジェクト指向など従来のプログラミング技法でモジュール化することの難しい処理群のことを指す。オブジェクト指向では、プログラムをオブジェクトとしてモジュール化する。それぞれのオブジェクトはある機能を表している。オブジェクト指向は様々な機能をカプセル化する能力に優れており、保守性、拡張性、可読性の高いプログラムを記述できる。しかし、ある種の機能のための処理群は複数オブジェクトに散らばってしまうことがある。これらの処理群が横断的関心事である。一般的な横断的関心事の例として、ログ処理や例外処理、プログラムの認証処理などが知られている。本稿では、アプリケーションの文脈に応じた最適化の指示が横断的関心事に当たる。横断的関心事は、プログラム中の様々な箇所に散らばってしまうため修正を行う際には、プログラムの様々な個所を変更する

する必要が出てしう。その結果一つの機能が一つのプログラムで表現されるという簡潔性が損なわれ、プログラムの保守性・可読性が損なわれてしまう。アスペクト指向プログラミングは、このようにオブジェクト指向だけでは、ログラム中に散らばってしまう処理を 1 つのモジュールにまとめることを目的としている。アスペクト指向プログラミングを用いてプログラムを設計することで、横断的関心事をアスペクトと呼ばれる一つのモジュールにまとめて記述することが出来る。代表的なアスペクト指向言語に AspectJ がある。AspectJ では、ポイントカットとアドバイスによってアスペクトを記述する。ポイントカットとは、プログラム実行中のある時点を指定するための記述である。また、アドバイスとは、ポイントカットによって指定された実行箇所で実行する処理を指す。ポイントカットとアドバイスを組み合わせることによって横断的関心事をアスペクトとしてモジュール化することが出来る。

しかし、既存の永続システムに AspectJ のようなアスペクト指向言語を適用しただけでは、アプリケーションの文脈情報を取得することは難しい。既存の永続システムとアスペクト指向を組み合わせただけでは、時間的に異なった時点でのデータアクセス情報を取得することが困難であるためである [5]。例えば、図 2.2 の `showPaperList()` を最適化する方法を考える。`showPaperList()` メソッドの最適化を行うためには、メソッドの引数である `Paper` オブジェクトのリスト `papers` に対して指示を出す必要がある。最適化の指示を出すためには、この `papers` リストの親オブジェクトの情報が必要であった。つまり、`papers` リストがどのようなリファレンスを辿って取得されたのかを知る必要がある。また、`papers` に対してどのような最適化の指示を出すかの決定するためには、`showPaperList()` メソッド内でどのようなデータが必要であるかといったアプリケーションの文脈が必要となってくる。しかし、多くのアスペクト指向言語では、このような時間的に異なった複数のアクセス・ポイントを取得することは難しい。つまり、`showPaperList()` 時点で、メソッド引数の `papers` がどのオブジェクトから取得されたかを知ることは出来ない。AspectJ のインタータイプ宣言や `percflow` 等を駆使すれば取得することも可能であるが、煩雑な記述が必要となってしまう。

`papers` の親オブジェクトを取得ためのアスペクト記述は図 2.3.2 のようになる。このコード例のように、既存の永続システムにアスペクト指向システムを適用するだけでは、対象となる永続オブジェクトがどのようなリファレンスを辿ってきたかを判断するのには、大変な労力がかかる。この例では、`papers` が取得される可能性のある全ての親オブジェクトに対していちいち処理を記述する必要がある。このような煩雑な記述は現実的ではない。

```
01  // 親オブジェクトが Proceeding の場合の処理
02  public aspect OptimizationByProceeding
03          percflow (execute(..ProceedingCotroller())){
04      Proceeding prc;
05      /* 親オブジェクトを格納 */
06      Proceeding around() :
07              call(Proceeding.getPapers()) {
08          Proceeding p = proceed();
09          this.prc = p;
10          return p;
11      }
12
13      /* showPaperList() の最適化処理 */
14      before(List<Paper> papers) :
15              call(..showPaperList(List))
16              && args(papers) {
17          // 最適化処理
18      }
19  }
20
21  // 親オブジェクトが Author の場合の処理
22  public aspect OptimizationbyAuthor {...}
23
24  // 親オブジェクトが Journal の場合の処理
25  public aspect OptimizationbyJournal {...}
26  ....
```

図 2.11: AspectJ による親オブジェクトの取得

2.4　問題点のまとめ

本章では、研究の背景について述べた。Java 言語の永続化には、昔から永続システムが利用されてきた。永続システムは、自動的に SQL を発行してくれるために、アプリケーションの開発効率を高めるが、パフォーマンスの面で問題があった。オブジェクト指向によってクラス毎にデータベースから取得するデータをカスタマイズするだけでは、効率の良いデータ取得は難しい。O/R mapper は、ルート・オブジェクトの取得をカスタマイズ出来る。原理的には、アプリケーションで必要なオブジェクトを全てルート・オブジェクトに含めることで正確な先読みを実現することが出来る。しかし、この方法は現実的ではない。ルート・オブジェクトを取得する際に、そのルート・オブジェクトからどのような永続オブジェクトのリファレンスが辿られていくかを予測するのは、困難であるためである。

アプリケーションの高速化を図るためには、永続オブジェクトのリファレンスが参照される際にデータベースから取得されるデータをアプリケーションの文脈に合わせて適切にカスタマイズする必要がある。しかし、このような指示を出そうとすると、データベースに関わる指示がアプリケーションに散在してしまい、アプリケアプリケーションの高速化を図るためにーションの可読性・保守性に問題がある。また、既存の永続システムとアスペクト指向言語を連結して最適化の指示を分離しようとしても、先読みを行うために重要なデータベースやフィールドへのアクセス履歴といったアプリケーションの文脈をを取得するために煩雑な記述が必要となってしまう。

第3章　アスペクト指向を利用した永続システムの高速化

本章では、アスペクト指向を用いた永続システムの高速化を提案する。また、アスペクト指向による高速化を支援するために、 Java 向けの永続システム AspectualStore を提案する。AspectualStore を利用したアプリケーションの高速化には、アスペクト指向を用いてアプリケーションの文脈に応じた最適化を永続オブジェクトに指示する。永続システムの高速化には、オブジェクト指向によるクラス毎のデータ取得の指示だけではなく、アプリケーションの文脈に応じた先読み等の最適化を永続オブジェクトに指示することが重要である。アプリケーションの文脈には、どの永続オブジェクトからの参照か、どのメソッド処理、どのようなデータベース・アクセスを引き起こしたのかといったものが考えられる。開発者は AspectualStore を利用することで、このようなアプリケーションの文脈に応じた、クラス横断的な最適化を指示することが出来る。また、永続システムへの最適化の指示がアプリケーション内に散在することなく、かつ柔軟にアスペクトとしてプログラミングできるようになる。アスペクトとして指示を与えるので、アプリケーション本体を変更せずに、性能テストの結果に合わせて、必要なところに最適化のヒント を徐々に与えてゆける。

アスペクト指向による高速化を実現するにあたって AspectualStore では、アスペクト指向を用いて容易に最適化出来る構造を提供している。まず、外部キーの取得といったデータベースに関わる作業を意識することなく永続オブジェクトまたはそのコレクションに対しての最適化の指示をだす API を用意した。永続オブジェクトやそのコレクションに対する指示は、対応する外部キーの名前や値といったデータベース・テーブルに関わる情報が必要であった。O/R mapper のように任意のクエリが記述できたとしても、リファレンスを辿りながら取得されている永続オブジェクトに対する指示を出すためには、記述が煩雑になり過ぎてしまう。AspectualStore が提供する API を利用することで、開発者は永続オブジェクトに対してテーブル情報やどのオブジェクトからのリファレンスかを意識することなく、直接的かつ容易に最適化の指示を出せるようになる。

また、AspectualStore では、永続オブジェクトおよび永続コレクションに対して、アプリケーションの文脈を格納するためのオブジェクト (コンテキスト・オブジェクト) を付加している。オブジェクト毎に最適化の指示を出すためには、対象となる永続オブジェクトの参照履歴やデータベース・アクセス履歴といったアプリケーションの文脈が必要となる。また、どのような文脈が必要となるかは、最適化のアルゴリズムによって異なってくる。開発者は、AspectualStore を拡張することで、最適化に必要な対象となるアプリケーションの文脈をコンテキスト・オブジェクトに格納し、アプリケーションの文脈に応じた最適化の指示を出すことが出来る。加えて、アスペクトのウィーブ/アンウィーブにかかる手間を削減するためにロードタイム・ウィービング等の機構を用意している。

3.1　アスペクト指向による最適化の例

永続システム AspectualStore によって構築されたアプリケーションをアスペクト指向によって最適化した例を示す。最適化の例としては、2.1.2 節の図 2.2 で示した永続システムのコードである `showPaperList()` メソッドを最適化することとする。

3.1.1　最適化のアルゴリズム

図 2.2 で `showPaperList()` メソッドのパフォーマンスが悪い原因は、メソッド内の `while` 文中でデータベース・アクセスが発生することだった。その結果としてループが回るたびごとにデータベースとの通信が必要となり、過剰な通信によってパフォーマンが低下してしまう。よって、`showPaperList()` メソッドを高速化するための要求は、「`showPaperList()` メソッド内で必要となるデータを出来るだけ少ないデータベース・アクセス回数で取得したい」ということになる。

そこで最適化のアルゴリズムとしては、`showPaperList()` 内の while 文で一回目のループが回る際に、発生するデータベース・アクセスを記録し、このメソッド内でどのようなデータが必要となるかを予測する。次に、メソッドの引数 `papers` に対して予測したデータを一括して先読みする指示を出すことで、過剰なデータベース・アクセスを防ぐこととする。つまり、一回目の `while` ループが回る際には、`while` 文内の `Paper` オブジェクト `p` もしくは `p` からのリファレンス参照によって取得された永続オブジェクトへのフィールド・アクセスによってどのようなデータベース・アクセスが発生するかを収集する。つまり、一回目の `while` ループが回る際には、`while` 文内の `Paper` オブジェクト `p` から永続オブジェクト

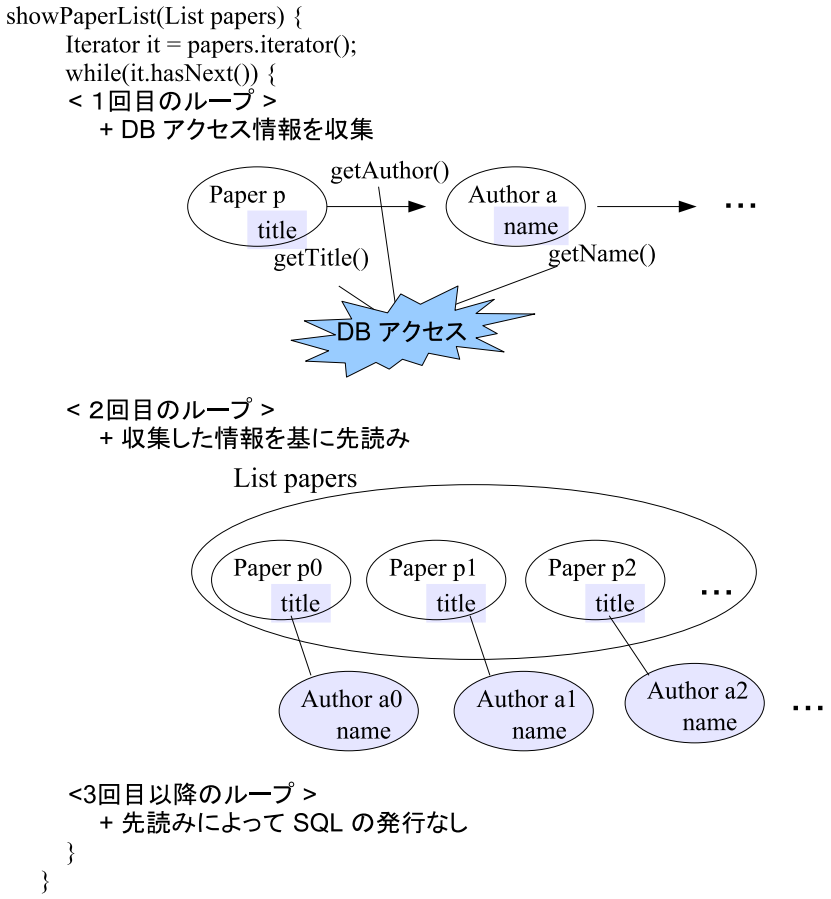

図 3.1: 最適化のアルゴリズム

のリファレンスが参照されていく過程でどのようなデータをデータベースから取得する必要があるかを記録する。例えば、while 内で `getAuthor()` メソッドによって `Paper` オブジェクトから `Author` オブジェクトへのリファレンスが参照された際にデータベース・アクセスが発生した場合には、そのリファレンス (フィールド名) を記録していく。

次に、一回目のループが回る間に記録したデータベース・アクセスを引き起こしたリファレンスを基に メソッドの引数 `papers` に対して先読みの指示を出す。この `while` 文は、全ての `papers` の全ての要素に対して同様の処理を行っている。そのため、一回目のループで辿られたリファレンスは、二回目以降のループでも辿られる可能性が高い。例えば、一回目のループで `Paper` オブジェクト `p` のフィールド `author` のリファレンスを参照した際にデータベース・アクセスが発生したのであれば、同様のデータベース・アクセスが二回目以降のループでも発生する可能性が高

い。そのため、papers　に含まれる全ての Paper オブジェクトの author フィールドの値を取得するような先読みを行う。このような先読みを行うことで二回目以降のループでデータベース・アクセスが発生する危険性がなくなり、パフォーマンスを高速化できる。最適化のアルゴリズム例を図 3.1 に示す。

3.1.2　アスペクト記述

3.1.1 節での showPaperList() メソッドへの最適化のアスペクト記述を図 3.2 に示す。AspectualStore では、最適化を記述するためのアスペクト機構として GluonJ [20, 17] を採用している。よって図 3.2 のアスペクト記述は、GluonJ の文法に基づいている。

図 3.2 の PrefetchDefine アスペクトは、最適化のための 3 つの処理が記述されている。以下の 3 つの処理である。

- データベース・アクセス情報を格納する場所の定義
- データベース・アクセスが発生した際にフィールド名を追加する処理の定義
- 先読みを実行する処理

これらの処理について詳しく説明する。

1 つ目の処理は、showPaperList() メソッド内でのデータベース・アクセス情報を格納する場所の定義である。PrefetchDefine アスペクトの 02 行目 から 11 行目では、Context クラスにデータベース・アクセスを引き起こしたフィールド名を格納するためのリスト loadedFields を追加している。また、同時に loadedFields リストを外部のクラスから利用するためのメソッド群の定義を行っている。これらのメソッドの詳細は割愛した。Context クラスは、AspectualStore がアプリケーションの文脈を格納するために提供しているクラスである。開発者は、この Context クラスを拡張することによって最適化に必要な文脈を管理することが出来る。Context クラスについては 3.2.2 節にて詳しく述べる。

2 つ目の処理は、showPaperList() メソッドの実行中に永続オブジェクトのフィールドが参照されることによってデータベース・アクセスが発生した場合に、そのフィールド名を 1 つ目の処理で追加した Context クラスのフィールドに追加する処理の定義である。GluonJ では、ポイントカットとアドバイスを Pointcut 型のフィールドとアノテーション (注釈) で定義する。つまり、16 行目の @Before アノテーションはアドバイスを表わす。GluonJ には、@Before 以外にも @After や @Around といったア

```
01  @Glue class PrefetchDefine {
02      /* DB アクセス情報を格納するフィールドを定義 */
03      @Refine
04      static class ContextDefine extends Context{
05        Set loadedFields = new HashSet();
06        int depth = 0;
07        void addFields(String f) {...}
08        Set getLoadedFields() {...}
09        void clear() {...}
10          boolean hasLoadedFields() {...}
11      }
12
13      /* showPaperList() 内でリファレンスを辿る過程で発生した
14       * データベース・アクセス情報 (フィールド名) を格納
15       */
16    @Before("{Context c = $2;
17              while(c.depth > 0) {...}
18              c.addField(ref);}")
19     Pointcut pc =
20         Pcd.call("...PersistentEntity#load*(..)").
21         and.cflow("showPaperList(..)");
22
23     /* showPaperList() の先読み定義 */
24     @Refine
25     static class IteratorDefine
26                      extends PersistentListIterator {
27         public PersistentEntity next(Context cxt) {
28             if(ctx.hasLoadedFields()) {
29                 Loader loader = getLoader(cxt);
30                 Collection fs = cxt.getLoadedFiels();
31                     // SQL の作成と先読みの実行
32                     loader.addFetchProperties(fs);
33                     loader.load();
34                 context.clear();
35             }
36             super.next(); }
37         }}}
```

図 3.2: 最適化を行うアスペクト記述

ドバイスを定義するためのアノテーションが提供されている。19 行目から 21 行目の Pointcut 型のフィールド pcd は、ポイントカットを表す。$Pcd.call(\text{"}...PersistentEntity\#load*(..)\text{"})$ は、PersistentEntity クラスの名前が load から始めるメソッド呼び出しを表す。PersistentEntity クラスは、AspectualStore 内のキャッシュ部分を表すクラスである。$Pcd.call(\text{"}...PersistentEntity\#load*(..)\text{"})$ は、リファレンスが参照された際に、データベースからデータを取得する時点を表す。また、$cflow(\text{"}showPaperList(..)\text{"})$ は、showPaperList() メソッドが実行されている間を表す。 $Pcd.call$ と $Pcd.cflow$ を *.and* で結合することによって 2 つのポイントカットの積をとっている。@Before アノテーション内で記述されている $1 および $2 は、ポイントカットされたメソッド (load*()) メソッドの第一引数および第二引数を表わす。$Pcd.call()$ や $Pcd.cflow()$ は、GluonJ が提供している Pcd クラスのメソッドである。アスペクトの記述者は、これらのメソッドを利用してポイントカットの定義を行う。GluonJ には、$Pcd.call()$ や $Pcd.cflow()$ 以外にも、指定されたフィールド値の読み込みおよび書き込み時を表す $Pcd.get()$ や $Pcd.set()$ メソッド等が提供されている。

最後に、PersistentIterator クラスを拡張して showPaperList() メソッド内での先読みを実行するコードを追加している。PersistentIterator は、AspectualStore が内部で利用しているクラスである。AspectualStore では、透過的なデータベース・アクセスを実現するために永続オブジェクトの getter が返す java.util.Set や java.util.List といったコレクションを内部で自動的に独自の実装のものに置き換えている。PersistentIterator は、java.util.Iterator の機能を代用するクラスである。つまり、図 2.2 で Iterator クラスの next() メソッドが実行された際には AspectualStore 内部で PersistentIterator の next(Context) メソッド が実行される。23 行目から 37 行目では、この next(Context) メソッドを拡張している。まず、next(Context) メソッドが呼ばれた際には、そのコンテキスト内でデータベース・アクセスが発生下かどうかを調べ (25 行目)、発生していた場合には、コレクション内の全てのオブジェクトに対してデータベース・アクセスを引き起こしたフィールドを初期化する処理を実行する。

3.1.3　より細かな最適化

アスペクト指向を利用した最適化の利点は、最適化のための指示をアスペクトとしてモジュール化出来ることである。最適化の指示はアプリケーション内に散在することは無く、またアスペクトとして局所化されているために、アプリケーションの最適化の必要性に合わせて容易にパフォーマンス・チューニングすることが可能である。例えば、3.1.1 節のアルゴリ

ズムによる最適化で、showPaperList() メソッドの最適化を実現できた。しかし、性能テストの結果、アプリケーションの開発者がさらにパフォーマンスを向上する必要があると判断した場合には、より細かな最適化の指示を出す必要がある。このような場合にもアスペクト記述を変更することで容易に最適化のアルゴリズムを追加できる。

showPaperList() 内で発生するデータベース･アクセスは、引数 papers の状態によって異なってくる。 2.1.3 節の図 2.4 から Paper オブジェクトのリストは、Proceeding オブジェクトや Journal オブジェクト、Author オブジェクトと様々なオブジェクトからのリファレンス参照によって取得されることがわかる。いま、簡単のため showPaperList() メソッド内の show() メソッド内では、Paper オブジェクトの title フィールドと author フィールドへの参照によってデータベース・アクセスが発生したとする。このとき、papers リストの 親オブジェクトが Author オブジェクトである場合と、Author オブジェクト以外の場合では、効率的な最適化の方法が異なってくる。

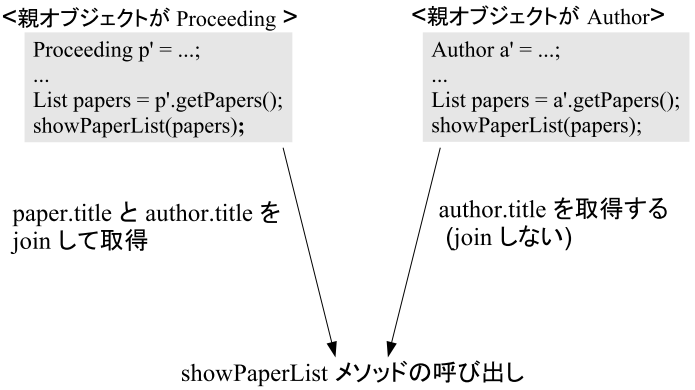

図 3.3: リファレンスの辿り方による最適化アルゴリズム

引数の papers リストが Proceeding オブジェクトから取得された場合 (Proceeding オブジェクトを papers の 親オブジェクトと呼ぶ) を考える (図 3.3 の左上)。このとき、showPaperList() メソッド以外の処理によって papers へのアクセスがなかったと仮定すると、データベースから取得するデータは図 3.4 のようになる。そのため、join 句によって二つのテーブル paper と author ををを結合して paper.title と author.name を一括して先読み取得するような SQL を発行してやることで showPaperList() メソッドを高速化出来る。

一方、papers の親オブジェクトが Author であった場合を考える (図 3.3 の右上)。親オブジェクトが Author である時は、その他の場合と事情が

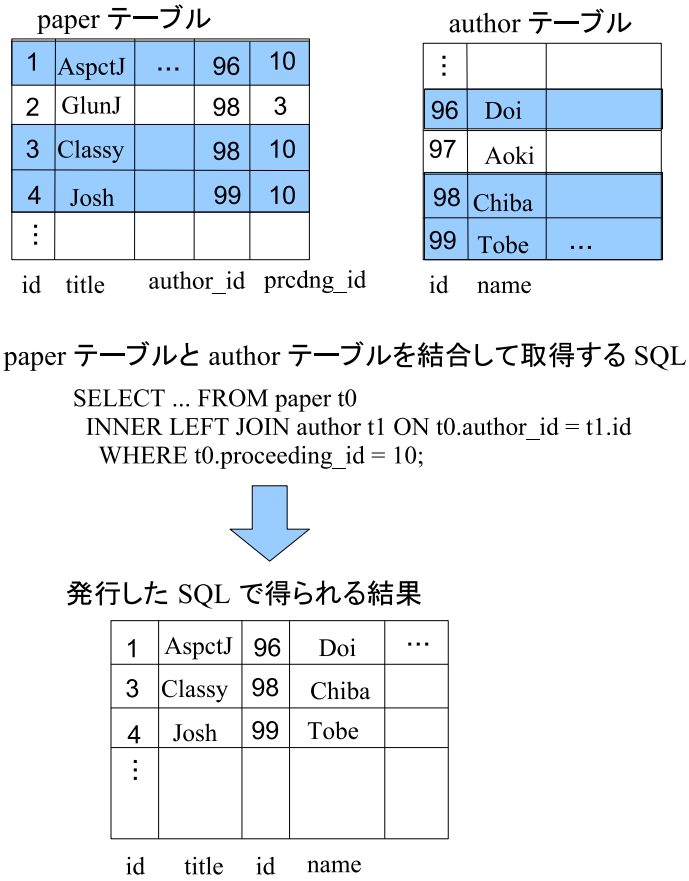

図 3.4: 親オブジェクトが Proceeding である場合の SQL 発行

異なる。`showPaperList()` メソッドの `while` 文内で生成される `Author` オブジェクトは全て同一のオブジェクトを表すためである。データベースから取得するデータは図 3.5 のようになる。
他の場合 (親オブジェクトが `Proceeding`) のように `paper` テーブルと `author` テーブルを join するといったコストのかかる SQL を発行するのではなく、`paper` テーブルのデータのみを取得することでより高速化を図ることが出来る。

以上のように、`showPaperList()` メソッドの最適化アルゴリズムとしては、メソッドの引数 `papers` リストの親オブジェクトに関するデータベース・アクセスは二回目の `while` 文で先読みする候補から外したほうが効率的なデータアクセスが出来ることがわかる。修正したアルゴリズムのアスペクト記述は、図 3.2 のアスペクト記述を図 3.6 のように変更することで実現できる。

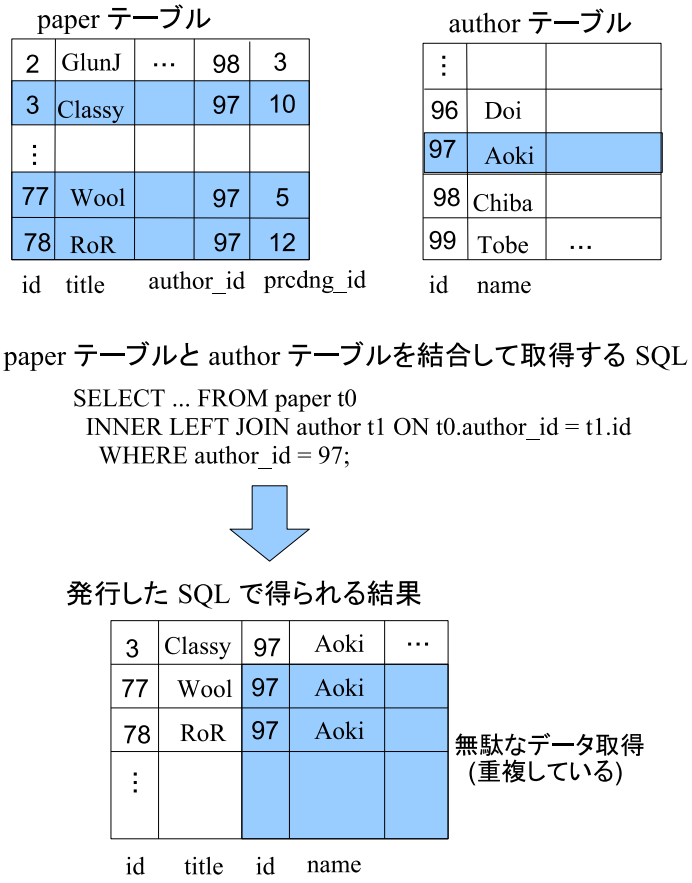

図 3.5: 親オブジェクトが Author である場合の SQL 発行

3.2　AspectualStore の提供する機能

本節では、AspectualStore がアスペクト指向による最適化を容易に実現出来るように提供する機能について述べる。AspectualStore では、アスペクト指向を用いて容易に最適化出来る構造を提供している。アプリケーションを最適化するためには、ルート・オブジェクトから永続オブジェクトのリファレンスが辿られていく過程で、そのオブジェクトの状況に合わせてどのようなデータをデータベースから取得すべきかの指示を出すことが重要であった。AspectualStore は、アスペクトとしてアプリケーションの文脈に応じた最適化の指示を容易に記述できるように、永続オブジェクトおよび永続オブジェクトのコレクションに対して最適化の指示を出すための API やアスペクト内から最適化のアルゴリズムを決定する際に必要な文脈を収集するための機構、容易なアスペクトの抜き差しを実現するた

```
01     /* showPaperList() の先読み定義 */
02     @Refine
03     static class IteratorDefine
04                     extends PersistentListIterator {
05         public PersistentEntity next(Context cxt) {
06             if(ctx.hasLoadedFields()) {
07                 Loader loader = getLoader(cxt);
08                 Persistable p = context.getBody();
09                 List fs = cxt.getLoadedFiels();
10                 // 親オブジェクト p のデータを除く
11                 removeParentProperty(fs, p);
12                 loader.addFetchPropertyes(fs);
13                 // SQL の生成および発行
14                 loader.load();
15                 context.clear();
16             }
17             super.next(); }
18         }}}
```

図 3.6: より細かな最適化のためのアスペクト記述

めのロードタイム・ウィービングを提供している。以下、AspectualStore が提供する機能について述べる。

3.2.1　永続オブジェクトに対する最適化の指示

この節では、AspectualStore が提供する、永続オブジェクトに対する最適化の指示を出すための API について述べる。アプリケーションの文脈に応じた最適化を実現するためには、ルート・オブジェクトからのリファレンスが辿られることによって取得される永続オブジェクトおよび永続オブジェクトのコレクションに対して最適化の指示を出す必要があった。AspectualStore では、永続オブジェクトおよびそのコレクションに対して容易に最適化の指示を出すための API を提供している。例えば、図 3.1 で先読みを行うためには、`showPaperList()` メソッドの引数である papers に対して最適化の指示を出した。`papers` は、図 2.4 で登場する永続クラス `Paper` のコレクションである。以下、永続オブジェクトを要素にもつコレクションのことを永続コレクションと呼ぶことにする。

例として、`Paper` オブジェクトのリスト `papers` に対して「papers に含まれる全ての Paper オブジェクトの author フィールドを取得しろ」という指示を出す場合を考える。2.3.1 で述べたように、O/R mapper のように任意のクエリを発行できるだけでは、永続オブジェクトに対して最

適化の指示を出すために大変煩雑な作業が必要であった。指示を出すため必要な作業としては以下のものがあった。

- SQL を記述するために親オブジェクトの情報が必要である
- 発行する SQL を個別のケースに応じて書かなければならない
- SQL の結果をオブジェクトに割り当てる作業が必要である

これらの作業にかかる負担を軽減するために AspectualStore では、予め「`papers` に含まれる全ての `Paper` オブジェクトの `author` フィールドを取得しろ 」という指示に対応する先読みを実現するための API を用意している。図 3.2 のアスペクト記述をみてわかるとおり、最適化の指示には O/R mapper が提供するような実際に発行する SQL に対応するクエリが記述されていない。最適化を実行する際には、クエリを記述する代わりに `Loader` クラスに対してどのリファレンスを取得するかの指示を出す。`Loader` クラスは、AspectualStore が永続オブジェクトに対して最適化の指示を出すために提供しているクラスである。

永続オブジェクトもしく永続コレクションの `Loader` オブジェクトを取得するためには、永続オブジェクトもしくは永続コレクションのキャッシュ部分に定義されている `getLoader(Context)` メソッドを呼び出せばよい。`Loader` がどのフィールドの値を取得するかの指示は `addFetchProperty()` および `addFetchProperties()` メソッドにフィールド名を渡すことで指示する。`Loader` オブジェクトは、永続クラスとデータベース・テーブルとのマッピング定義を元に自動的に SQL を作成してくれる。そのため、アスペクトを記述する開発者は、マッピング定義や外部キー等のデータベース・テーブルに関わる要素を意識することなく簡潔な指示を出せる。

最適化の記述を行うために必要となる作業の 1 つに、SQL を記述するためには親オブジェクトの情報を取得しなければならないという手間があった。AspectualStore では、親オブジェクトの情報を容易に取得出来るよう永続クラスおよび永続クラスを格納するコレクションクラスにその親オブジェクトへの参照を持たせている。ただし、永続オブジェクトおよび永続コレクションはアプリケーション内でキャッシュとして何度も使い回されるため、単純に永続クラスにフィールドを持たせるだけではうまくいかない。親オブジェクトへの参照の管理は 3.2.2 節で述べる。`Loader` クラスは、親オブジェクトへの参照を基にフィールド名から自動的に発行すべき SQL を生成する。また、発行した SQL の結果を元に指示を出した永続オブジェクトまたは永続コレクションにデータベースから取得したデータを自動的に割り当てる。そのため、O/R mapper を用いたときのような煩雑な作業は一切必要なくなる。

永続オブジェクトや永続コレクションに対する指示は、上の例のように全てのオブジェクトに対してフィールドの値を取得しろといった単純な指示だけとは限らない。ある学会で発表された論文のうちジャンルが AOP に関するものだけがほしいとする。つまり、「ある Proceeding オブジェクトに対して、Paper クラスの List 型のフィールド papers のデータを取得する際に Paper オブジェクトのフィールド genre の値が ”AOP” であるものだけを所得しろ」という指示について考える。このとき、Loader クラスへの指示は以下のようになる。

```
Loader loader = ...;
Expression exp = ExpressionFactory.eq("genre", "AOP");
loader.addExpression(exp);
loader.load();
```

Expression クラスや ExpressionFactory クラスは、AspectualStore が条件を指示するために用意しているクラスである。取得したいデータを Expression として指示してやることで煩雑な条件を加えることが出来る。なお、このとき Loader は、SQL 文を二度発行する。一度目の SQL は、Proceeding の papers フィールド の初期化を行うための SQL で、二度目の SQL は、指示された AOP に関するデータを取得するためである。本来は、AOP に関する Paper オブジェクトだけを取得すればよいのがが、AspectualStore では、最適化のヒントによってアプリケーションの挙動を変えてしまわないようにするため、指示のなかった Paper オブジェクトに対しても主キーだけは、取得する。

3.2.2　文脈毎の情報を格納する機構

AspectualStore では、アプリケーションの文脈を格納するための機構を提供している。最適化のアルゴリズムを決定する際には、対象となる永続オブジェクトがどのようなリファレンスを辿ってきたのか、どのようなフィールド・アクセスやデータベース・アクセスを引き起こしたかといったアプリケーションの文脈に応じて最適化の指示を変更したいことがある。また、どのようなアプリケーションの文脈を必要とするかは、最適化のアルゴリズムによって異なってくる。例えば、図 3.1 の例では、while 文内でどのようなデータベース・アクセスが発生したのかという情報を基に最適化を実行した。また、図 3.3 では、showPaperList() メソッドの引数 papers がどのようなリファレンスを辿ってきたかによって先読みの指示を変更した。

永続オブジェクトがどのようなリファレンスを辿ってきたかや、フィールドやデータベースへのアクセス履歴といった時間的に異なった時点での

データアクセス情報を取得することは、既存の永続システムにアスペクト言語を適用するだけでは難しい。例えば、代表的なアスペクト指向言語である AspectJ で提供されている `cflow` や `execution` ポイントカット等では、これらの情報を取得することが出来ない。インタータイプ宣言等を用いて永続クラスにフィールドを追加すれば、アプリケーションの文脈毎に異なるリファレンスの参照情報やアクセス履歴を管理することが出来るかもしれない。しかし、永続システムの場合には、一般のアプリケーションと異なり、単純に永続クラスにフィールドを追加するだけでは、文脈ごとの情報を管理することが難しい。というのは、多くの永続システムでは、生成された永続オブジェクトはキャッシュに格納されて何度も使い回されるためである。つまり、異なる文脈から同一のデータベース・レコードを表わすオブジェクトにアクセスがあった場合には、キャッシュに積まれているオブジェクトが何度も利用される。キャッシュを利用することで一度発行した SQL を再び発行する手間を省くことが出来るためである。このようにキャッシュとして利用されるため、永続オブジェクトに文脈情報を格納してしまうと異なる文脈ごとの情報が衝突してしまう危険性がある。

そこで、AspectualStore では、予め永続クラスおよび永続クラスを格納するコレクションクラスに対して利用したいアプリケーションの文脈情報を格納するためのクラス (`Context` クラス) を付加している。`Context` オブジェクトは、キャッシュ部分と異なり何度も利用されることはなく、リファレンス参照毎に新たに生成さる。そのため、開発者は、アスペクトによってコンテキスト・クラスを拡張することでデータベースへのアクセス履歴やフィールドへのアクセス履歴といった最適化に利用したいアプリケーションの文脈を Context オブジェクトへ格納し、管理することが出来る。また、このような文脈を利用して、アプリケーションの最適化アルゴリズムを決定することが可能となっている。例えば、図 3.2 では、`Context` クラスにデータベース・アクセスを引き起こしたリファレンスを格納し、その情報を基に先読みするデータを決定した。

アプリケーションの文脈の１つとして、その永続オブジェクトの親オブジェクトへの参照がある。永続オブジェクトは複数の異なる文脈で呼び出される可能性がある。例えば、以下の例を考える。図 3.7 は、あるアプリケーションでの永続オブジェクトのリファレンスが参照される流れである。この例ではある学会 N で発表された論文 X の著者 A のオブジェクトを取得し、著者 A の論文一覧である 論文 P, Q, X からなる List を取得している。このとき、このアプリケーション内では論文 X オブジェクトが二度登場する。一つは学会 N から取得されたオブジェクトであり、もう一方は、著者 A の論文一覧の中のオブジェクトである。この二つの論

文 X オブジェクトは共に同一のデータベース・レコードを表わすが、それぞれの親オブジェクトは異なる (学会 N と 著者 A)。このような文脈ごとに異なる親オブジェクトの情報を管理するように AspectualStore では、コンテキスト部分に親オブジェクトへの参照を持たせている。前節で述べた永続オブジェクトや永続コレクションに対して最適化の指示を出す Loader クラスは、このコンテキスト情報を利用して発行する SQL の作成を行う。

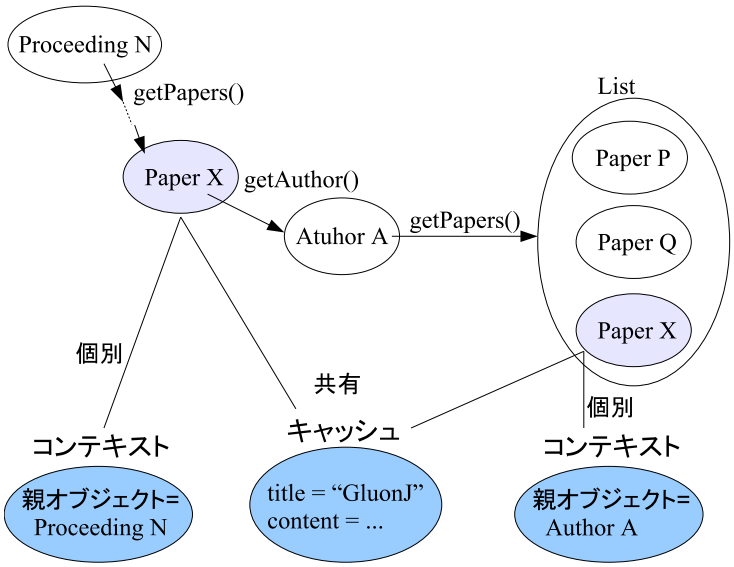

図 3.7: アプリケーションの文脈を格納する仕組み

3.2.3　ロードタイム・ウィービング

AspectualStore では、アスペクトのウィーブおよびモデルクラスの永続化を行うタイミングとしてロードタイムを選択している。また、AspectualStore を Tomcat [2] と連結させた場合にもロードタイムにバイトコード変換を行うため、独自のクラスローダを提供している。ロードタイムにウィーブすることによって、開発者はアスペクト記述を書き換えるだけでアプリケーションの最適化を実現することが出来る。

ロードタイム・ウィービングとは、あるクラスのバイナリ表現 (クラスファイル) がクラスローダによってロードされるタイミングに、そのクラスに対するアスペクト処理をバイトコード変換することを指す。AspectualStore を利用したアプリケーションでは、開発者は AspectualStore 自体をを拡張することによってアプリケーションの高速化を図る。例えば、図 3.2 で

は、AspectualStore 内部の `Context` クラスや `PersistentEntity` クラスのバイトコードを変更している。バイトコード変換をロードタイム以外のタイミング (デプロイタイムやコンパイルタイム) で行おうとした場合、アスペクトのウィーブ/アンウィーブに手間がかかる。つまり、拡張を反映させた AspectualStore の JAR ファイルを新たに作成し、拡張前の JAR ファイルと取り替える必要がある。AspectualStore では、このような煩わしい作業を削減するためロードタイム・ウィービングをサポートしている。ロードタイムにバイトコード変換を施すことで、透過的に AspectualStore の内部クラスを拡張することが可能となっている。

AspectualStore では、ロードタイム・ウィービングを実行する方法を 2 通り用意した。1 つ目は、AspectualStore を単独で利用する場合である。2 つ目は、AspectualStore を tomcat と連結させた場合である。AspectualStore を単独で利用する場合、AspectualStore は java 5 の機能である javaagent オプションを利用してバイトコード変換を行う。java 5 では、クラスファイルのバイトコードをロードタイムに書き換える枠組みが提供されている。アプリケーションの実行時に以下のような javaagent オプションを追加することでこの仕組みが利用できる。

```
java -javaagent:aspectualstore.jar <アプリケーション>
```

javaagent オプションを使用すると、アプリケーションの main メソッドが実行される前に、指定した JAR ファイル (aspectualstore.jar) 内のエージェントクラスの `static` メソッド `premain (String agentArgs, Instrumentation inst)` が呼び出される。`premain (String agentArgs, Instrumentation inst)` エージェントクラスとは、JAR ファイルのマニフェスト・ファイルに記述されている属性 Premain-Class で指定されたクラスを指す。AspectualStore は、`premain` メソッドで、永続クラスに対する永続化処理を追加するバイトコード変換を行う。また、`premain` メソッドの引数である `Instrumentation` オブジェクトに `ClassFileTransformer` オブジェクトを追加している。`Instrumentation`、`ClassFileTransformer` は `java.lang.instrumet` パッケージに定義されているクラスおよびインターフェースであり、`ClassFileTransformer` インターフェースは、クラスファイルがクラスローダにロードされる過程で呼び出されるクラスである。この `ClassFileTransformer` インターフェースを実装することでアスペクト記述に対するロードタイムのバイトコード変換を実現している。なお、アスペクト記述の織り込みには、GluonJ のウィーバーを利用している。

次に、AspectualStore は Tomcat を連結させた場合も容易にロードタイム・ウィービングを実現出来るように独自の Tomcat 用クラスローダ

を用意した。先ほど述べたように java 5 から追加された javaagent オプションとバイトコード・トランスレータ javassist を用いればクラスローダを差し替えなくとも、バイトコード変換は可能である。しかし、クラスローダを差し替えなければ対処できない場合も存在する。例えば、データベースを利用したアプリケーションの多くは、Web アプリケーションである。Tomcat は、Java 向けの Web アプリケーションサーバとして広く利用されている。AspectualStore を Tomcat と連結させて利用する際に、ロードタイムにバイトコード変換を行う方法として javaagent オプションだけでは不十分である。Tomcat 等の Web サーバでは一般にいくつもの Web アプリケーションが動作する。もし javaagent オプションを指定して Tomcat を起動したとすると全ての アプリケーションに対して同一のバイトコード変換とアスペクト記述が適用されてしまうことになってしまう。これでは、個別のアプリケーションに対応したアスペクトを記述することが難しくなってしまう。

Tomcat では、Web アプリケーション毎にクラスローダを指定出来る。AspectualStore と Tomcat を連結する際には、以下のような設定を記述することでロード・タイムに永続クラスに対する永続化処理とアスペクトのウィーブを透過的に行ってくる。`ASLoader` および `ASClassLoader` は、AspectualStore が用意した Tomcat 用のクラスローダである。

```
<Context privileged="true">
    <Loader className="aspectualstore.classloader.ASLoader"
        loaderClass="aspectualstore.classloader.ASClassLoader"
        useSystemClassLoaderAsParent="false"
    />
</Context>
```

図 3.8: Tomcat のクラスローダの差し替え

3.2.4　その他の機能

アスペクト指向を利用して効率よくアプリケーションの最適化が出来るよう AspectualStore が提供しているその他の機能について述べる。

プロパティ単位のデータ取得

AspectualStore では、プロパティ単位のデータ取得をサポートしている。プロパティ単位のデータ取得とは、つまりフィールド単位でデータベースから取得するデータを指示することである。フィールド単位での

データベースから取得するデータを指示を出すことで、アプリケーションのメモリ容量を抑えることが出来、アプリケーションの高速化を図ることが可能となる。

多くの 永続システムや O/R mapper では、プロパティ単位のデータ取得はサポートしていない。プロパティ単位でデータを取得することにすると永続オブジェクトのフィールドにアクセスがある度にデータベースからデータを取得しなければならないため、データベースへの過剰なアクセスを引き起こしてしまう危険性があるためである。多くの場合、永続システムではリファレンス参照によって永続オブジェクトを生成する際には、そのオブジェクトの全てのフィールドの値をデータベースから取得するように設計されている。

しかし、アプリケーションの文脈によってアクセスのあるフィールドが定まる場合には、そのフィールドの値だけをデータベースから取得することでアプリケーションの高速化を図ることが出来るはずである。そこで AspectualStore では、開発者が細かな最適化を行いたい場合に備えてプロパティ単位データ取得をサポートしている。永続オブジェクトが生成される際に、取得するデータはアスペクトとして記述する。

```
01  @Glue
02  class PropertyLoadDefine {
03      @Before("{$0.setFetchLevel(FetchLevel.PRIMITIVE)}")
04      Pointcut pcd = Pcd.call("Loader.beforeLoad()");
05  }
```

例えば、以上のような記述で永続オブジェクトが生成される際には、プリミティブ型のフィールドの値のみをデータベースから取得することになる。FetchLevel クラス、Loader がどのようなフィールドを取得するかを指示するためのクラスである。Loader の振る舞いに対する指示には表 3.1 のようなものがある。アプリケーションの文脈毎に決まるデータ取得は別途アスペクトとして記述することとなる。

表 3.1: Loader クラスのデフォルトの挙動

	挙動
DEFAULT	主キーと外部キーを取得する
NONE	主キーのみを取得する
ALL	主キーと全てのプロパティを取得する
PRIMITIVE	主キーと外部キー、プリミティブ型のプロパティを取得する

内部ドキュメントの公開

AspectualStore では、開発者が AspectualStore 内部のクラスを拡張して、最適化を実現しやすいように、内部ドキュメントを公開している。アプリケーションの高速化を図るためには、どうしても AspectualStore 内部の構造を拡張する必要がある。例えば、図 3.1 の例でも AspectualStore 内部のクラスである Context クラスや PersistentCollection クラスを拡張した。このような内部クラスを容易に拡張するために、AspectualStore では内部実装を内部クラスのドキュメントにどのような拡張を施せばよいかのドキュメントを用意している。開発者は、アプリケーションの高速化を図る際には、このドキュメントを元にどのクラスを拡張すべきかを判断することになる。

第4章　実装

この章では、AspectualStore の実装について述べる。AspectualStore では、開発者がアスペクト指向を利用して最適化の指示を出しやすい構造に設計されている。AspectualStore と他の永続システムとの一番の設計の違いは、データベース・アクセスが必要となった時点で SQL を作成する点と最適化の際に利用したい文脈情報を管理するために、永続クラスをキャッシュ部分とコンテキスト部分に分離している点である。

これらの機能は、永続システムを利用するだけなら全く必要ない機能である。永続オブジェクトのフィールドにアクセスがあった際に発行する SQL は、予め永続クラスとデータベース・テーブルとの対応付けの定義から決定することができるため、予め作成しておいた方が SQL を作成する手間が省ける。また、永続オブジェクトがどのオブジェとからのリファレンスを辿ってきたかなどのアプリケーションの文脈情報は本来データを取得する際には、必要ない情報である。しかし、アプリケーションの文脈に応じて、永続オブジェクトのフィールドへのアクセスによって透過的に発行される SQL を変更することによって最適化を実現しようとした際にはこれらの構造が必要となる。以下、AspectualStore の構造について述べる。

4.1　SQL 発行

AspectualStore では、永続オブジェクトまたは、永続オブジェクトのリファレンスを辿っていく際に取得される永続オブジェクトや永続オブジェクトのコレクションに対して最適化の指示を出すための API を用意している。多くの永続システムでは、永続オブジェクトへのリファレンスが参照された際に発行する SQL のひな形を予め作成している。例えば、図 2.4 に登場する `Paper` クラスには、`author` フィールドへのアクセスがあった場合に備えて、

```
"SELECT t0.id, t0.name FROM author t0 WHERE t0.id = ??"
```

といった SQL の文字列を予め用意しておく。実際に `Paper` オブジェクトの `author` フィールドへのアクセスがあった際には、この SQL 文字列の

?? の部分にオブジェクト毎の値を代入することで SQL を一から作成することなくデータベースから `author` フィールドの値を取得出来る。

しかし、このように予め SQL 文字列を用意していたのでは、永続オブジェクト毎に最適化の指示を変更することが出来ない。そこで、AspectualStore では、フィールド・アクセスがあった場合に永続オブジェクトが保持しているデーベース・レコードの値とクラスとデータベース・テーブルとの対応付けの定義を元に SQL を作成する。なお、AspectualStore では、バイトコード変換によって永続クラスに永続化の処理を追加する。また、永続オブジェクトのコレクションに対して内部で用意した永続化用のコレクション (`PersistentList` や `PersistentSet`) を割り当てる。AspectualStore 内の SQL 発行の仕組みは、図 4.1 のようになっている。

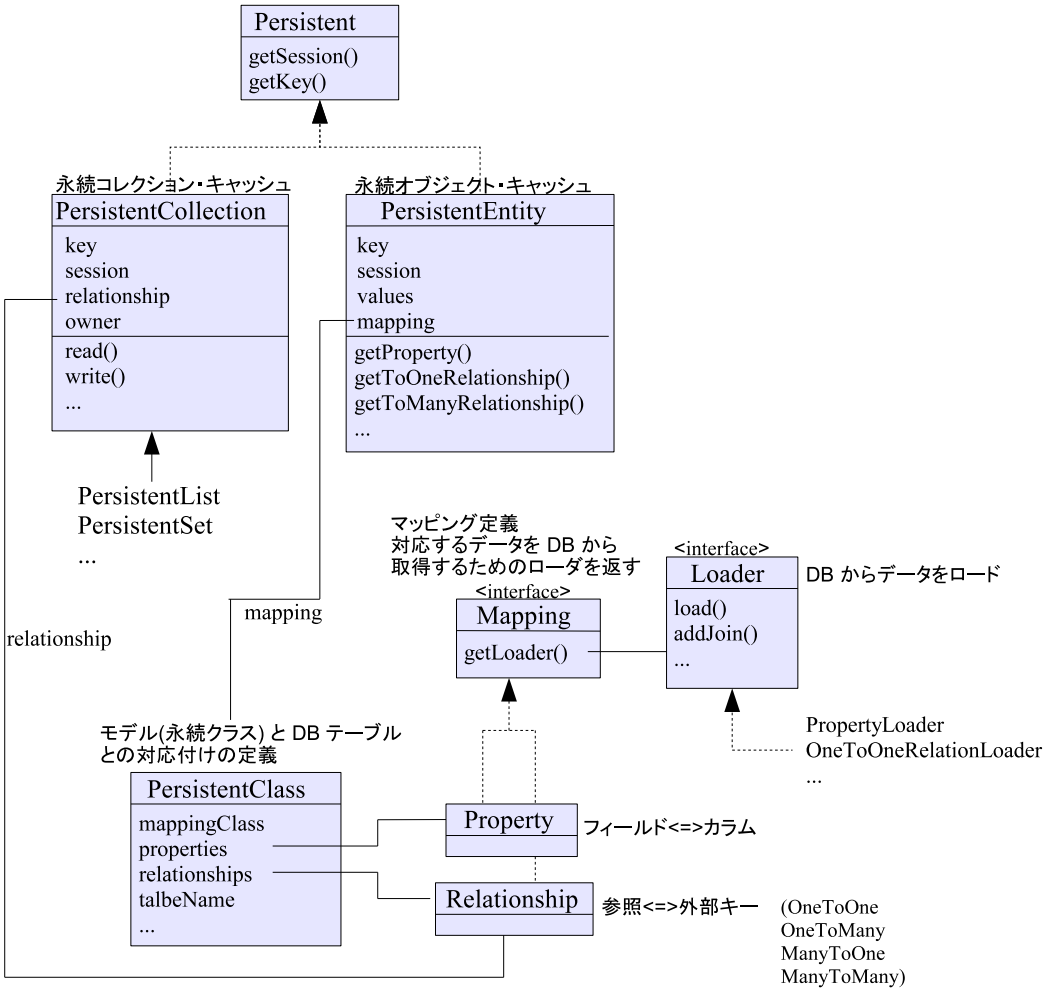

図 4.1: SQL 発行の仕組み

SQL の発行および発行した SQL の結果を永続オブジェクトに割り当てる作業は、`Loader` クラスが行う。`Loader` クラスは、アクセスのあったフィールドとマッピング定義から SQL 文字列を作成し、データベースとのコネクションを取得し、SQL を発行する。その後、SQL の発行によって得られた結果 (`ResultSet`) からオブジェクトへの割り当て・生成を行う。なお、開発者が永続オブジェクトや永続オブジェクトのコレクションに対する最適化の指示を出す際にも、永続オブジェクトから `Loader` オブジェクトを取得して指示を与える。Loader クラスが提供している API は、表 4.1 のようになる。

表 4.1: Loader クラスが提供する API

メソッド名	機能
void addProperty(String)	データベースから取得するフィールド群の追加
void addProperties(Collection)	データベースから取得するフィールドの追加
void addExpression(Expression)	データベースからデータへの条件の追加
void addJoin(JoinLoader)	join 句によって一括して取得するデータの追加
Persist load()	データベースから指定されたデータの取得と取得したデータのオブジェクトへの割り当て

4.1.1　バイトコード変換

AspectualStore では、バイトコード変換によって永続化処理の追加とアプリケーションの文脈情報を格納するためのコンテキスト・クラスの付加を行う。永続化処理とは、アプリケーションによって永続オブジェクトのフィールドが参照された際に、そのフィールドの値をデータベースから取得する等の永続化のために必要な処理を指す。また、コンテキスト・クラスは、データベースへのアクセス履歴などのアプリケーションの文脈毎に異なる情報を格納するために AspectualStore が提供しているクラスである。開発者は、アスペクトによってコンテキスト・クラスを拡張することでデータベースへのアクセス・履歴やフィールドへのアクセス履歴といった文脈毎に異なるオブジェクトの情報を管理出来る。また、このよう

な文脈情報を利用して、アプリケーションの最適化アルゴリズムを決定することが可能となっている。

永続クラスのバイトコード変換の例を図 4.2 に示す。永続クラスは、バイトコード変換によってキャッシュ部分 (`PersistentEntity`) と文脈情報を格納するコンテキスト部分 (`Context`) に分離される。バイトコード変換には javassist [19, 18] を利用している。

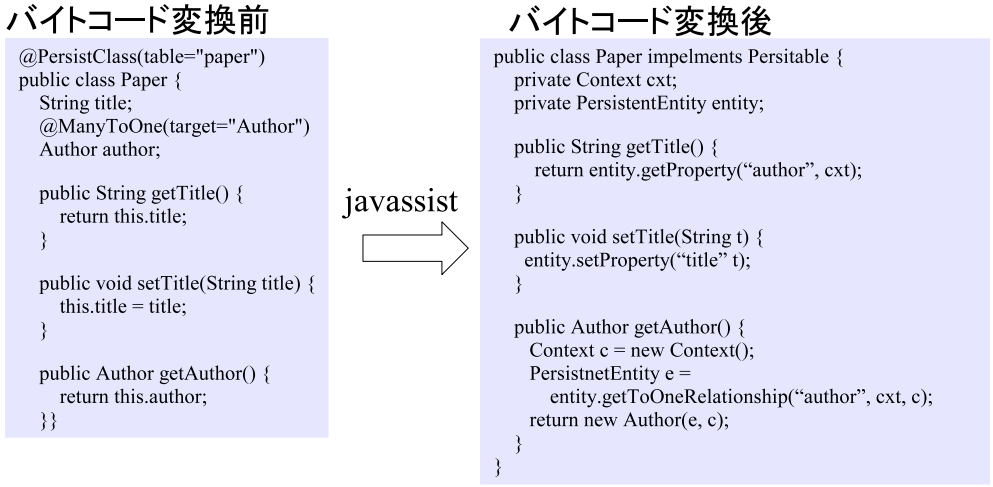

図 4.2: 永続クラスに対するバイトコード変換

キャッシュ部分は、データベース・レコードに関連する情報を管理し、アプリケーション内で何度も使いまわされる。具体的には、永続オブジェクトのフィールドに対応するデータベース・レコードの値の管理やデータベースとの一貫性を保つためにオブジェクトの状態 (レコードへの書き込みやレコードの削除が必要かどうか) を管理している。一方、コンテキストは、どの親オブジェクトに属しているか等のオブジェクトごとに異なる情報を管理する。異なった文脈でキャッシュが再利用される際にも、コンテキスト部分には新たなオブジェクトが割り当てられる。そのため、開発者は、このコンテキストクラスを拡張することによって、オブジェクトの文脈ごとに異なる情報を収集・利用することが出来る。

4.2　クラスローダ

AspectualStore では、アスペクトのロードタイム・ウィーブを実現するために、独自のクラスローダを実装した。ロードタイムにウィーブすることによって、開発者はアスペクト記述を書き換えるだけでアプリケーショ

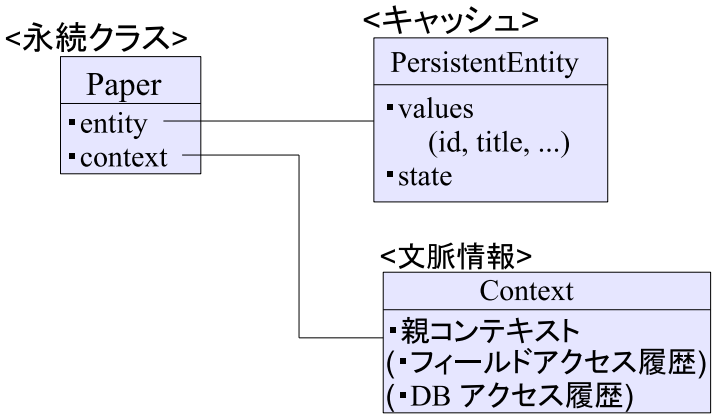

図 4.3: 文脈情報を格納する仕組み

ンの最適化を実現することが出来る。アプリケーションの高速化には、アプリケーションのコードだけではなく、AspectualStore の内部クラスを拡張しなければならない場合も多い。また、AspectualStore も Context クラス等の開発者が拡張することを前提とした内部クラスを提供している。ロードタイムにアスペクトをウィーブすることで、開発者は JAR ファイルの置き換え等の作業なしにアスペクト記述を書き換えるだけで最適化のアルゴリズムを変更することが可能となる。

ロードタイム・ウィービングを実現する主な方法としては、以下の 2 通りの方法がある。

- javaagent オプションの利用
- クラスローダの差し替え

java 5 から導入された javaagent オプションを利用すれば、クラスローダを置き換えずともロードタイムにバイトコードを変更することは可能である。しかし、AspectualStore を tomcat と連結させて利用する場合には、javaagent オプションだけでは対応しきれない。というのは、Tomcat 等の Web サーバでは一般にいくつもの Web アプリケーションが動作するためである。もし javaagent オプションを指定して Tomcat を起動したとすると全ての アプリケーションに対して同一のバイトコード変換とアスペクト記述が適用されてしまうことになってしまう。これでは、個別のアプリケーションに対応したアスペクトを記述することが難しくなってしまう。

そこで、AspectualStore では、Tomcat と連結した場合にもロードタイム・ウィービングを実現できるよう独自のクラスローダを用意している。

実装は、Tomcat 5.5 を基に行った。Tomcat が各 Web アプリケーションをロードするために用意しているクラスローダを AspectualStore が提供しているクラスローダに差し替えることによってロードタイム・ウィービングを実行する。AspectualStore のクラスローダは、クラスローダが生成された際に、まず AspectualStore の設定ファイルを基に永続化されるクラスに対するバイトコード変換を行う。また、Web アプリケーション内のクラスおよび aspectualstore.jar のクラスがロードされる際には、アスペクト記述を基に必要に応じてバイトコード変換を施すように実装されている。

Tomcat は、コンテナやコンテナ上で動作する Web アプリケーションが利用可能なクラスやリソースのリポジトリにアクセスできるよう様々なクラスローダを利用している。Tomcat のクラスローダの構成は、図 4.4 のようになっている。`Bootstrap` クラスローダおよび `System` クラスローダは汎用的なものである。`Bootstrap` クラスローダは、JVM によって提供されているクラス群をロードし、`System` クラスローダは CLASSPATH に追加されている内容をロードする。一方、`Common` クラスローダや `Shared` クラスローダは、全ての Web アプリケーションで利用されるクラス群をロードする。また、`Catalina` クラスローダは、Tomcat の内部で利用するクラス群をロードする。WebappX (X = 1, 2, ...) クラスローダは、各

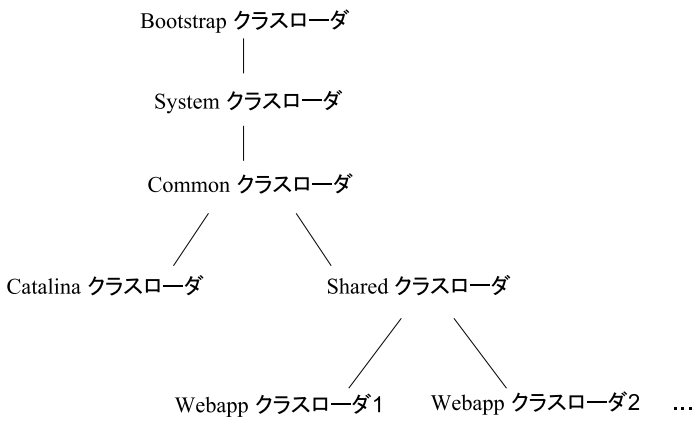

図 4.4: Tomcat のクラスローダ群

Web アプリケーション毎にクラスローダである。AspectualStore は、この WebappX クラスローダを独自のクラスローダに置き換えることで、Web アプリケーション毎の永続化処理の追加およびアスペクト記述の読み取りを行っている。つまり、WebappX クラスローダによって Web アプリケーションのクラスファイルがロードされる際に、そのクラスファイ

ルにアスペクト記述を反映させる必要があるかどうかを判断し、もしアスペクトを織り込む必要があるならば、Javassit によってバイトコード変換を行う。

第5章　考察

本章では、提案したアスペクト指向による永続システムの高速化についての考察を行う。まず、アスペクト指向によるアプリケーションの最適化での利点と欠点について述べる。また、その他の永続システムの高速化手法との比較や関連研究について述べる。

5.1　AspectualStore による最適化の限界

この節では、AspectualStore を利用して永続オブジェクトに対して最適化の指示を出す方法と方法、JDBC や Hibernate の HQL を用いて開発者が、直接クエリを記述する方法について比較する。AspectualStore では、アスペクトとして永続オブジェクト対して最適化の指示を出す。アスペクト指向を利用することで、アプリケーションの文脈に応じた永続オブジェクトに対する最適化の指示が、アプリケーション内に散在することなくアプリケーションを高速化することが出来るようになる。

しかし、AspectualStore でアスペクトから出す指示は、あくまで最適化のためのヒントでありアプリケーションの挙動を変更すべきではない。例えば、以下のコードを考えてみる。

```
01  Proceeding prc = ...;
02  List<Paper> papers = prc.getPapers();
03  Iterator<Paper> it = papers.iterator();
04  while(it.next()) {
05    Paper p = it.next();
06    if(p.getGenre().equals("AOP") {
07      // show Paper details
08      ...
09    }
10  }
```

このコードは、Paper オブジェクトのリスト papers のうち、ジャンルが AOP であるものに対して処理を行うコードとなっている。このとき、AspectualStore では、以下のようなアスペクトを定義することによって、

Proceeding オブジェクト prc に対して papers フィールドのリファレンスが参照される際には、ジャンル が AOP であるのみに対して全てのフィールドを初期化するような最適化を発行することとする。

```
01  @Before("{Loader l = $0.getValue().getLoaer();
02            Expression exp =
03                ExpressionFacotry.eq("genre", "AOP");
04            l.addExpression(exp);
05            l.setFetchLevel(ALL);
06            l.load();
07           }")
08  Pointcut pcd = Pcd.call("Paper#getPapers()")
```

高速化のことだけをデータベースから取得するデータは、図 5.1 のようになる。つまり、ジャンルが ”AOP” である Paper オブジェクトのみをデータベースから取得することで高速化を図る方法がもっとも効率的である。

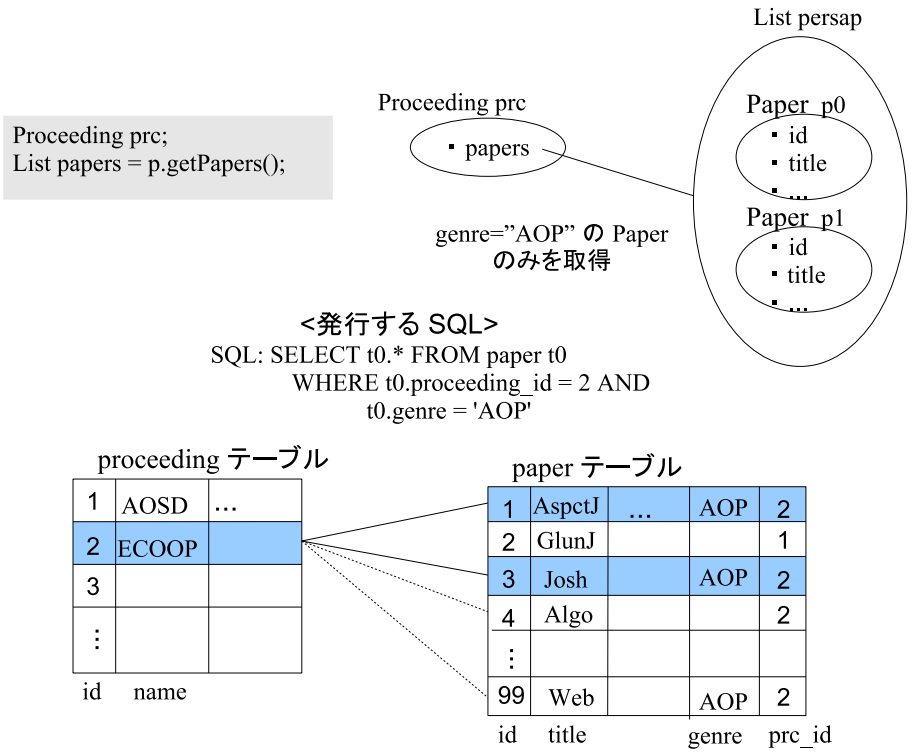

図 5.1: jdbc によるデータ取得

一方、AspectualStore を利用して `Proceeding` オブジェクト `prc` に対する最適化の指示によって `getPapers()` メソッドが呼ばれた際にデータベースから取得されるデータは、図 5.2 のようになる。

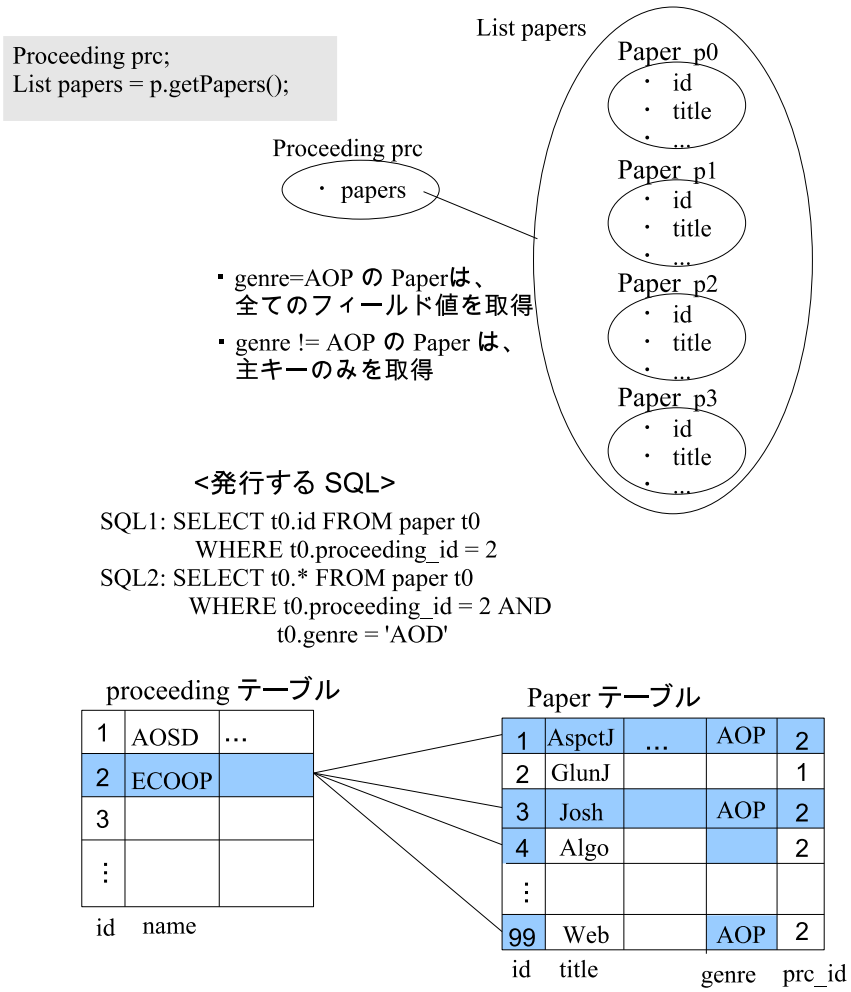

図 5.2: AspectualStore によるデータ取得

AspectualStore では、`Proceeding` オブジェクト `prc` にジャンルが AOP であるものだけを取得する指示を出したとしても、ジャンルが AOP でないオブジェクトに対してもその 主キーだけは、データベースから取得する仕様になっている。つまり、このような指示に対して AspectualStore では、SQL を 2 回発行する。一つ目は、`Proceeding` オブジェクト `prc` のフィールド `papers` に対応する全ての `Paper` オブジェクトの主キー id を取得する SQL である。その後、ジャンルが AOP である `Paper` オブジェクトに対して全てのプロパティを取得する SQL を発行する。2 回の SQL

発行によってジャンルが AOP でない `Paper` オブジェクトは、主キー `id` のみが取得され、ジャンルが AOP である Paper オブジェクトは、全てのフィールドの値が取得される。

このような仕様にすることによって、クエリによって直接指示を出す方法と比べると最適化の精度は下がる。つまり、クエリによって直接指示を出す方法程の高速化は実現できていない。それにも関わらず、AspectualStore 不必要なデータ (ジャンルが AOP ではない `Paper` オブジェクトの主キー) まで取得しているのは、最適化の指示によってアプリケーションの挙動を変化させることが無いようにするためである。AspectualStore では、リファレンスを辿る際に取得されたオブジェクトやコレクションをキャッシュとして利用され何度も再利用される。そのため、取得されるオブジェクト自体を変更してしまうと、別の文脈からこのオブジェクトが呼び出された場合にも影響が出てしまう。例えば、もし、アプリケーションから `papers` の `size()` メソッドを読んだ場合を考える。図 5.1 のような最適化では、`size()` メソッドが返す値が最適化する前と最適化した後で変わってきてしまう。このような最適化の挙動の変化は、アプリケーションに不必要なバグを生む危険性がある。一方、AspectualStore の最適化の指示は、永続オブジェクトに対するデータ取得の指示を出すが、最適化の指示によってアプリケーションの挙動が変わってしまうことはない。AspectualStore では、あくまで永続オブジェクトにデーベースから取得するデータの指示を出すことによってアプリケーションを高速化することを目的としている。表 5.1 に AspectualStore によって永続アプリケーションを構築した場合と JDBC や Hibernate の HQL を用いて逐次データベースから取得するクエリを記述した場合の比較を示す。

表 5.1: AspectualStore とローレベルな API による高速化の比較

	AspectualStore	ローレベル API
アプリケーション構築の容易さ	○	×
チューニングの容易さ	○	×
高速化	○	◎

5.1.1　実験

AspectualStore による高速化の有効性を検証するために、予備的な実験を行った。実験内容としては、2.1.3 節の文献検索システムを用いて、ルート・オブジェクトである `Bibliography` オブジェクトからのリファレンスを辿りながら、任意の論文およびその著者の情報を取得するアプリ

ケーションに対し、実行速度およびデータベースへの通信回数、データベースから取得したテーブル・プロパティ (セル) の個数を以下のシステムを実行し比較した。

- 既存の永続システムによってクラス単位の静的な指示を出す場合
- AspectualStore によってアスペクトから指示を出す場合
- JDBC によって必要なデータを適宜取得する場合

既存の永続システムとしては、AspectualStore にクラス単位の静的な指示だけを出すことで代用した。また、マッピングのポリシーとしては、永続オブジェクトをデータベースから取得する際には、そのオブジェクトの全てのプロパティ(フィールド) を取得するものとした。また、JDBC によるデータ取得を行う場合には、AspectualStore を改造し実際にアプリケーションで必要なデータのみを SQL によって取得することとした。SQL を記述する際には、アプリケーションのモジュール性は全て無視してアプリケーションの性能のみを求めるように設計した。

実験環境　データベース用のサーバとしては　Sun Fire V60x Server (CPU Intel(R) Xeon(TM) CPU 3.06Hz × 2, 2GB, Linux 2.6.7) を用い、データベースには PostgreSQL 7.4.2 を用いた。アプリケーション用のサーバとしては、Xserve G5 (CPU PowerPC G5 2.3GHz x 2, 3GB, Mac OS X) を用いた。LAN は　1000 BaseTX である。

表 5.2: 速度および DB アクセス、データ取得量の比較

	実行時間 (秒)	DB 通信 (回)	オブジェクト生成 (個)	取得データ (個)
既存の永続システム	3.48	43	381	2585(1522)
AspectualStore	2.46	8	381	756(289)
SQL 直書き	2.16	6	181	471(287)

実験結果　実験結果は、表 5.2 の通りである。取得データ数の () 内の値は、取得したプロパティ (セル) のうち主キー・外部キーを除いた数である。この予備実験では、AspectualStore を利用したアプリケーションに対してアスペクトによって最適化を行うことで、約 30 パーセントの速度向上を図れたことを確認した。また、SQL を直接記述することによって必要なデータの指示を出した場合には、約 38 パーセントの速度向上が確認され

た。AspectualStore による高速化が SQL を直書きする方法ほどの高速化を実現できていない理由は、前節に述べたとおりである。AspectualStore では、アプリケーションの挙動を変えることなくデータベースから取得するデータを最適化する。そのため、表 5.2 の取得データ数は、既存の永続システムに比べて大幅に少なくなっているが、生成したオブジェクト数は既存の永続システムと変わらない。また、JDBC 直書きの方法と比べて取得データ数が増加している原因は、アプリケーションの挙動を変えないために主キーや外部キーを必ず取得するように設計されているためである。そのため、主キーと外部キーを除いたデータ取得するは、SQL を直書きした場合とさして変わらない。

5.2　関連研究

この節では、永続システムの高速化やアスペクト指向の永続システムへの適用といった本論文と関連する研究について述べる。

5.2.1　永続システムの高速化

永続システムや O/R mapper の高速化については、数多くの研究がなされている。以下に述べる関連研究と本研究の大きな違いは、アプリケーションの最適化を誰が行うのかという点である。以下に述べる研究では、アクセス・ログやプロファイリングによってオブジェクトやページのアクセスパターンを解析し、システムが自動的に最適化を行ってくれる。

一方、我々の提案するアスペクト指向による高速化方法では、開発者がアスペクトを記述することで最適化を行う。そのため、システムが自動的に最適化を行う方法と比べてアプリケーションの高速化を図るためにアスペクトを記述しなければならないという手間が増える。その一方で、AspectualStore では、個別のアプリケーションにおいて開発者が任意の最適化を施すことが可能となっている。また、必要性に応じてアクセス・パターン等の永続オブジェクトに関する文脈だけではなく、どのメソッドから実行されているかなどのアプリケーション固有の情報を基に高速化することが出来る。

最適化のアルゴリズムは、以下に述べるように数多くあるが、それぞれに利点や欠点がある。自動的な高速化では、開発者がさらなる高速化を望んだとしても対応出来ない。AspectualStore では、最適化の程度はアプリケーションの高速化の必要性と開発者にかかるコストとを見比べて選択できる。つまり、それほど高速化が必要でない場合には、単純なアルゴリズムによる高速をアスペクトとして記述すればよいし、高度な高速化が必

要である場合には、細かな文脈に応じた最適化の指示をアスペクトで記述してやればよい。また、AspectualStore では出来るだけ最適化のアルゴリズムが容易に記述出来るような API を提供している。

context-controlled prefetch [15] は、アプリケーションからのデータベースへのアクセスパターンによって動的に先読みするデータを決定する最適化方法である。例えば、コレクションの一要素である永続オブジェクトのフィールドにアクセスがあった際には、このコレクションに含まれる全ての永続オブジェクトの対してこのフィールドの先読みを行う。同じコレクション内の要素に対しては、同様のフィールド・アクセスが行われる可能性が高いためである。このような動的な先読みによってアプリケーションの高速化を図る。しかし、コレクションの一要素にアクセスがあったからといって必ず、同様のアクセスが全ての要素にあるとは限らない。コレクションの多くの要素にアクセスがなかった場合には、逆にパフォーマンスを低下さえる原因となってしまう。このように [15] は、深さ優先探索 (DFS) 等のアクセス・パターンに対して弱いという欠点がある。

AUTOFETCH [1] は、トラバーサル・プロファイリングによって自動的な先読みを行うことで高速化を図る研究である。トラバーサル・プロファイリングとは、クエリによって取得されたルートオブジェクトからどの永続オブジェクトへのリファレンスを参照しながらアプリケーションが実行されるかを記録していくことを指す。AUTOFETCH では、アプリケーションを最適化なしで一度実行することによって得られた情報を基にして、ある一定以上の確立で参照のあった永続オブジェクトをルートオブジェクトが取得される際に一括して取得する。始めにアプリケーションを動作させる際には、パフォーマンスが悪いという欠点はあるが、一度情報を収集した後はプロファイリングで得られた情報を基にした先読みを行うことでアプリケーションを自動的に高速化することが可能である。自動的な先読みを行ってくれるために、O/R mapper のようにアプリケーションのコードが変更される度に先読みの記述を変更するような作業は必要ない。また、PrefetchGuide[23] は、永続オブジェクトのアクセス・ログを基に動的な先読みを行う。これらのアルゴリズムの欠点は、ユーザ・インタラクションを含むアプリケーションに対応出来ないという点とアプリケーションの環境を考慮出来ない点である。例えば、AUTOFETCH [1] では、ルートオブジェクトを取得する際に発行する SQL に先読みするデータを含めることでアプリケーションの高速化を図るが、アプリケーションにとって最適な SQL の発行アルゴリズムは、利用するデータベースの性能やアプリケーションのメモリ容量、データベースとの通信速度等によって異なってくる。例えば、二つのテーブルからデータを取得する場合、join によって SQL を一回にまとめる方が SQL を二回にまとめる方法より高

速になるとは限らないことがわかっている。SQL を一回にまとめることによって通信のオーバーヘッドは削減できるが、その分 join を利用することによって取得するデータ容量が増加してしまう可能性がある。

このほかにも数多くの高速化アルゴリズムが提案されている。ObjectStore [7, 16] では、ページ単位の先読みを行う。あるオブジェクトをロードする際には、同一のページ内の全てのオブジェクトをロードする。同一のページ内のアクセスが連続して生じる場合には有効であるが、そうでない場合には逆にパフォーマンスを低下させてしまう危険性がある。また、Curewitz らは、データ圧縮のアルゴリズムを先読みに応用することを提案している [12]。[12] では、オブジェクトやページのデータベースに対するアクセスをモニタリングしながら、データ圧縮のアルゴリズムを基にアクセス・パターンを解析し、先読みを行う。しかし、この方法は、同一のオブジェクトに複数回アクセスがあった場合にのみ有効である。

5.2.2　システムの永続化

永続システムの開発には、オブジェクトの透過的な永続化を実現するためにオブジェクト指向に特化したデータベースシステムが利用されることがある。ObjectStore [7, 16] や caché [10] に代表されるオブジェクト指向データベース (OODB) がこれにあたる。OODB は、予めオブジェクト指向的システムによるアクセスを前提としているため、リレーショナル・データベース (RDB) を利用する場合と比べてインピーダンス・ミスマッチを意識する必要が無いという利点がある。一方、RDB を用いる利点は、2 つ考えられる。1 つ目はもともと存在している RDB を永続オブジェクトとして扱うことが出来ること、2 つ目は OODB のような新たなデータベースエンジンを導入する必要が無いことである。そのため、RDB と OODB を相互運用しなければならいといった問題が生じることは無い。本研究では、RDB を利用した永続システムの高速化を目的としている。

Java に柔軟性の高い永続記憶を導入することを目的とした Atkinson らによる PJama プロジェクトも行われている [13]。PJama は Java 言語を拡張し、オブジェクトに対する柔軟性の高い永続化を実現したシステムである。PJama では、永続化は型に関係なく、オブジェクトして定義されているすべてのデータは永続化される。また、永続化のために特別なコードを記述する必要がない。また、任意のオブジェクト指向言語に対して、透過的な永続化を実現する研究もある。nitr0 [8] は、リフレクションを利用して、任意のオブジェクト指向言語に対して透過的な永続化処理を実現する。一方、本研究はリレーショナルデータベースのデータを効率よく取得することによって永続システムの高速化を図るための研究である。

5.2.3　アスペクト指向の適用

永続システムにアスペクト指向を利用した研究も数多く存在する。Awais Rashid らは、永続システムをアスペクト指向を用いて設計することによって、永続化を必要とするシステムのモジュラリティを高めることが可能であることを、典型的なデータベースアプリケーションである文献検索システムを実際に構築した経験を元に示している [6]。また、アスペクト指向を用いて永続システムを構築する際に考慮すべき点やアスペクトの再利用性についても述べられている。本研究では、永続システムを用いたアプリケーションの高速化のためのチューニングにアスペクト指向を用いている。

また、Awais Rashid らは、データ指向のアスペクトをサポートするためのジョイポイントの提案を行っている [5]。従来のアスペクト指向では、与えられたアプリケーションの実行コンテキスト内でのデータに基づいたジョインポイントを扱うことが出来るが、時間的に異なった時点でのデータアクセスには対応していない。そこで、Awais らは、アクセスのあった時点のデータとタイムスタンプをコンテキスト情報として利用できるジョインポイントを提案した。しかし、Awais らのジョインポイントは永続システムには対応していない。また、アスペクトから取得出来る情報は、データの更新のあったタイムスタンプと回数だけである。我々は、永続オブジェクトに対するフィールド・アクセスやデータベース・アクセスへの履歴といった最適化に必要なアプリケーションの文脈を柔軟に利用できるような構造をもつ永続システム AspectualStore の開発を行った。

本研究では、アスペクト指向を用いて、永続システムの高速化を実現したが、他の分野での、アスペクト指向を利用したシステムの高速化技術として、μ-Dyner [14] がある。μ-Dyner は C 言語用の動的アスペクト指向システムの一つである。μ-Dyner は、動的アスペクト指向システムであるため、先読みに関する記述がキャッシュプログラム中に散在せず、動的に先読みを行うことが出来る。状況に応じて適切に先読みのポリシーを変更することによって効率のよい Web キャッシュの実現が可能となっている。

第6章　まとめ

本稿では、アスペクト指向によるリレーショナルデータベースを用いた永続システムの高速化について提案した。また、アスペクト指向による高速化を支援するために、Java 向けの永続システム AspectualStore を提案した。AspectualStore では、アスペクト指向を利用してルート・オブジェクトからリファレンスが参照されていく過程で取得される永続オブジェクトや永続オブジェクトのコレクションに対して最適化の指示を出す。永続システムの高速化には、オブジェクト指向によってクラス毎にデータベースからのデータ取得をカスタマイズするだけではなく、どの永続オブジェクトからの参照か、どのメソッド処理かといったアプリケーションの文脈に応じて永続オブジェクトごとに最適化を指示することが重要であった。AspectualStore を利用することで、このようなクラス横断的な最適化の指示がアプリケーション内に散在することなく、かつ柔軟にアスペクトとしてプログラミングできるようになった。また、最適化の指示がアスペクトしてモジュール化されるために、アプリケーションの高速化の必要性に合わせて容易にチューニングを行うことが可能となった。

永続システムを高速化するためには、出来る限りデータベースとの通信回数を減らし、かつアプリケーションで利用される必要最小限のデータのみを取得することが重要である。過剰なデータベース・アクセスは、データベースとの通信のオーバーヘッドによってパフォーマンスの低下をもたらし、またデータベースからの不必要なデータ取得はアプリケーションのメモリ容量に負荷をかけるためである。必要なデータのみをデータベースから一括して取得するためには、アプリケーションの文脈に応じてデータベースから取得するデータをカスタマイズする必要がある。例えば、アプリケーションの文脈を基に次に必要となるデータを予測し一括して先読み取得することでアプリケーションのパフォーマンスを大幅に向上することが出来る。

しかし、EJB2 のような既存の永続システムでは、クラス毎の静的な指示しか出せないため、アプリケーションの文脈に応じた最適化が出来ないという問題点があった。近年永続システムに代わって良く利用されている O/R mapper では、アプリケーションの高速化を図るためにルート・オブジェクトの取得をカスタマイズする API が用意されている。原理的には、

この API を用いてアプリケーションで必要なオブジェクトを全てルート・オブジェクトに含めることで最低限のデータベース通信で必要なデータのみを取得することが出来る。しかし、この方法は現実的ではなかった。ルート・オブジェクトを取得する際に、そのルート・オブジェクトからどのような永続オブジェクトのリファレンスが辿られていくかを予測するのは、困難であるためである。AspectualStore では、ルート・オブジェクトからのリファレンスを辿っていくことで取得された永続オブジェクトに対して、アプリケーションの文脈に応じてデータベースから取得するデータの指示を出すことでアプリケーションの高速化を図る。そのため、O/R mapper のようにルート・オブジェクトが取得される際に全ての先読みを行う必要はなく、ルート・オブジェクトからのリファレンスの辿られ方に応じて最適化のアルゴリズムを変更することが出来るようになった。

AspectualStore では、アスペクト指向による高速化を実現するにあたって、アスペクトから永続オブジェクトに対して容易に最適化の指示を出すための構造を用意した。まず、SQL を直接記述することなく永続オブジェクトまたはそのコレクションに対しての最適化の指示をだす API を用意した。開発者が永続オブジェクトに対する指示を直接 SQL で指定しようとすると、そのクラスに対応する外部キーの名前や値といったデータベース・テーブルの属性が必要であった。また、発行する SQL は、そのオブジェクトの参照を持つ親オブジェクトによって異なった。AspectualStore が提供する API を利用することで、開発者は永続オブジェクトに対して直接的かつ容易に最適化の指示を出せるようになった。また、アプリケーションの文脈に応じた指示を出すためには、対象となる永続オブジェクトがどのようなリファレンス参照によって取得されたのか、データベース・アクセス履歴やフィールド・アクセス履歴が重要となる。AspectualStore では、アスペクトによって容易に最適化を実現出来るよう、アスペクト内で必要なアプリケーションの文脈を利用しながら最適化の指示を出すための機構や、アスペクトのウィーブ/アンウィーブにかかる手間を削減するためのロードタイム・ウィービング等の機構を用意した。

参考文献

[1] Ali Ibrahim, William R. Cook: Automatic Prefetching by Traversal Profiling in Object Persistence Architectures, *European Conference on Object-Oriented Programming (ECOOP)* (2006).

[2] Apache Software Foundation: Apache Tomcat, `http://tomcat.apache.org/`.

[3] Apache Software Foundation: Cayenne, `http://incubator.apache.org/cayenne/`.

[4] AspectJ Project: Aspectj, `http://eclipse.org/aspectj/`.

[5] Awais Rashid, Ruzanna Chichyan: Data-Oriented Aspect, *Asian Workshop on Aspect-Oriented Software Development (AOAsia)* (2006).

[6] Awais Rashid, Ruzanna Chitchyan: Persistence as an aspect, *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD)*, pp. 120–129 (2003).

[7] Charles Lamb, Gordon Landis, Jack Orenstein, Dan Weinreb: The ObjectStore database system, *Communications of the ACM*, Vol. 34, pp. 50–63 (1991).

[8] Francisco Ortin, Benjamin Lopez, J. Baltasar Carcia Perez-Schofield: Separating adaptable persistence attributes through computational reflection, *IEEE Software*, Vol. 21, pp. 41–49 (2004).

[9] Gregor Kiczales: An Overview of AspectJ, *European Conference on Object-Oriented Programming (ECOOP)*, pp. 327–353 (2001).

[10] InterSystems Corporation: Caché, `http://www.intersystems.com/cache/`.

[11] JBoss, Inc: HIBERNATE, `http://www.hibernate.org/`.

[12] Kenneth M. Curewitz, P. Krishnan, Jeffrey Scott Vitter: Practical prefetching via data compression, *Proceedings of International Conference on Management of Data*, pp. 257–266 (1993).

[13] Malcom Atkinson, Mick Jordan, Laurent Daynes, Susan Spence: Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, *The Proceedings fo the 7th International Workshop on Persistent Object Systems*, pp. 33–47 (1996).

[14] Marc Segura-Devillechaise, Jean-Marc Menaud: Web cache prefetching as an aspect: towards a dynamic-weaving based solution, *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD)*, pp. 110–119 (2003).

[15] Philip A. Bernstein, Shankar Pal, David Shutt: Context-Based Prefetch for Implementing Objects on Relations, *Proceedings of the 25th International Conference on Very Large Data Bases* (1999).

[16] Progress Software Corporation: ObjectStore, `http://www.progress.com/realtime/products/objectstore/`.

[17] Shigeru Chiba: GluonJ, `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

[18] Shigeru Chiba: Javassist Home Page, `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[19] Shigeru Chiba: Load-time Structural Reflection in Java, *Proceedings of the European Conference on Object-Oriented Programming*, pp. 313–336 (2000).

[20] Shigeru Chiba, Rei Ishikawa: Aspect-Oriented Programming beyond Dependency Injection (ECOOP), *European Conference on Object-Oriented Programming*, pp. 121–143 (2005).

[21] Sun Microsystems, Inc: EJB, `http://java.sun.com/products/ejb/`.

[22] Sun Microsystems, Inc: JDBC, `http://java.sun.com/javase/technologies/database/`.

[23] Wook-Shin Han, Yang-Sae Moon, Kyu-Young Whang: PrefetchGuide: capturing navigational access patterns for prefetch-

ing in client/server object-oriented/object-relational DBMSs, *Information Sciences*, Vol. 152, pp. 47–61 (2003).

付 録A　AspectualStore の仕様とプログラム例

付録として AspectualStore を利用した場合のアプリケーションの構築方法とそのプログラム例を示す。

A.1　永続クラスの定義

リレーショナルデータベースを利用した永続システムを利用する際には、永続クラスを定義し、永続クラスとデータベース・テーブルとの対応付けを行う必要がある。どのクラスとどのテーブルが対応してどのフィールドとどのカラムが対応するのかがわからなければデータベースからデータを取得しようがないためである。

クラスとテーブルとの対応付けには、XML ファイルやアノテーションによる定義がが一般的である。AspectualStore では、アノテーションによる定義を用いている。永続化されるクラスには、`@PersistClass` アノテーションを注釈することで永続クラスと他のクラスを識別する。AspectualStore が永続クラスとデータベース・テーブルとの対応付けのために用意しているアノテーションは表 A.1 の通りである。

A.1.1　永続クラスの自動生成

永続化クラスとデータベース・テーブルとのマッピング定義は、形式的な定義が多く大変面倒な作業である。AspectualStore のマッピング定義には、A.1 節のようにアノテーションを用いて出来る限りユーザの記述を少なくしているが、それでもいちいちアノテーションを注釈していくのは面倒である。本来、永続クラスとデータベース・テーブルは一対一に対応するものである。そのため、例えば、データベース・テーブルを定義すれば永続クラスを定義したと同値である。AspectualStore では、データベース・テーブルから永続クラスを自動生成するユーティリティを提供している。永続クラスを自動生成するに当たっては、表 A.1 の定義の default 値を用いる。例えば、テーブルに `proceeding_id` というカラムがあった場

表 A.1: 永続クラスの定義に用いるアノテーション

PersistClass	永続化させるクラスに注釈
String table	対応するテーブル名
String pk	主キーを表わすカラム名 (default は id)
Persist	永続化させるフィールドに注釈
String column	対応するカラム名 (default はフィールド名)
boolean readOnly	書き込みを禁止 (default は false)
Transient	永続化させないフィールドに注釈
OneToOne	他の永続クラスと１対１関連をもつフィールドに注釈
String target	相手のテーブル名 (default はフィールド名)
String sourceColumn	自テーブルの外部キー (default は id)
String targetColumn	相手テーブルの外部キー (default は id)
boolean cascade	カスケード (default は false)
OneToMany	他の永続クラスと一対多関連をもつコレクション型フィールドに注釈
String target	相手のテーブル名 (default はフィールド名)
String sourceColumn	自テーブルの外部キー (default は id)
String targetColumn	相手テーブルの外部キー (default は 自テーブル名_id)
boolean cascade	カスケード (default は false)
ManyToOne	他の永続クラスと関連をもつフィールドに注釈
String target	相手のテーブル名 (default はフィールド名)
String sourceColumn	自テーブルの外部キー (default は 自テーブル名_id)
String targetColumn	相手テーブルの外部キー (default は id)
boolean cascade	カスケード (default は false)
ManyToMany	他の永続クラスと関連をもつフィールドに注釈
String target	相手のテーブル名 (default はフィールド名)
String juntionTable	ジャンクション・テーブル名 (default はフィールド名)
String sourceColumn	自テーブルの外部キー (default は id)
String targetColumn	相手テーブルの外部キー (default は id)
String cascade	カスケード (default は false)

合には、永続クラスには、List proceedings というフィールドが生成されることになる。また、自動生成された永続クラスには、テーブルのカラムや他のテーブルとの関連を表わすフィールドの定義と各フィールドの getter、setter が定義される。それ以外に必要なフィールドやメソッドは、開発者が生成された永続クラスに新たに追加することになる。

A.2　サンプルアプリケーション

AspectualStore のサンプルコードを示す。次のコードは、ある学会で発表された論文とその著者の一覧を表示するといった簡単なものになっている。なお、データベースの URL やドライバといった設定は、aspectualstore.cfg に記述することになっている。設定ファイルの記述は割愛する。

```
01  public class Sample {
02      /* サンプルプログラムの実行 */
03      public void run(int year, String confernce) {
04          Session session = SessionFactory.getFactory().openSession();
05          Transaction t = session.beginTransaction();
06          try {
07              Bibliography bib = getBib(session, year);
08              List<Proceeding> prcs = bib.getProceedings();
09              for(Iterator it = prcs.iterator(); it.hasNext();) {
10                  Proceeding proc = it.next();
11                 if(proc.getName().equals(conference)) {
12                      List<Paper> papers = proc.getPpaers();
13                      showPaperList(papers);
14                  }
15              }
16              t.commit();
17          }
18          catch(Exception e) {
19              t.rollback();
20              showError(e);
21          }
22      }

30      /* ルート・オブジェクトの取得 */
31      Bibiliography getBib(Session session, int year) {
32          Expression exp = ExpressionFactory.eq("year", new Integer(year));
33         Criteria c = new SelectCriteria(Bibliography.class, exp);
34          return session.load(c);
35      }

40      /* 論文一覧の表示 */
41      void showPaperList(Collection<Paper> papers) {
```

```
42          Iterator<Paper> it = papers.iterator();
43          while(it.hasNext()) {
44              Paper p = it.next();
45              showPaper(p);
46          }
47      }

50      /* 論文の表示 */
51      void showPaper(Paper p) {
52          String title = p.getTitle();
53          Author a = p.getAuthor();
54          String name = a.getName();
55          System.out.println(title + " : " + name);
56      }

60      /* main */
61      public static void main(String[] args) {
62          int year = 2006;
63          String conference_name = "AOSD";
64          new Sample().run(year, conference_name);
65      }
66  }
```

AspectualStore の提供するクラスのインターフェースは、Hibernate のクラスを参考にしている。Hibernate と同様に永続オブジェクトを取得するためには、SessionFactory クラスから Session クラスを取得する必要がある。30-35 行目の getBibliography() メソッドではルート・オブジェクトの取得を行っている。AspectualStore では、Expression クラスをルート・オブジェクトの取得をカスタマイズする。Expression は、ExpressionFactory クラスから生成される。利用出来る Expression には、以下のものがある。開発者は、各 Expression を結合してルート・オブジェクトを取得する際に任意の条件を指定出来る。

ルート・オブジェクトを取得した後は、データベースを全く意識することなくアプリケーションを開発することが出来る。例えば、showPaperList() や showPaper() メソッドにはデータベースに関わる処理は一切登場しない。AspectualStore が内部で透過的に永続オブジェクトのリファレンスへの参照があった際にデータベースからデータを取得するためである。もしアプリケーションを構築した後でそのパフォーマンスに不満がある場合には、アスペクトによって永続オブジェクトに対して最適化の指示を出すことで容易にパフォーマンス・チューニングを行うことが出来る。

表 A.2: ルート・オブジェクトの取得をカスタマイズする API

Expression	制約
EqualExpression(String *col*, Object *val*)	*col* の値が *val* に等しい
GeExpression(String *col*, Number *val*)	*col* の値が *val* 以上
GtExpression(String *col*, Number *val*)	*col* の値が val より大
LeExpression(String *col*, Number val)	*col* の値が *val* 以下
LtExpression(String *col*, Number val)	*col* の値が *val* より小
BetweenExpression(String *col*, Number *lo, Number ho)*	*col* の値が *lo* 以上 *ho* 以下
LikeExpression(String *col*, String *val*)	*col* の値が *val* の表す表現と一致
NotLikeExpression(String *col*, String *val*)	*col* の値が *val* の表す表現と不一致
InExpression(String *col*, List *vals)*	*col* の値が *vals* の要素
NotExpression(Expression *exp)*	*exp* の否定