Vol.2010-SE-170 No.14
2010/11/12

クラウド時代のデータ自動最適配置技術の開発・評価

溝渕裕司[†]　大嶽智裕[†]　山本晃治[†]　野村佳秀[†]
小高敏裕[†]

分散 KVS（Key Value Store）等の NoSQL（Not only SQL）と呼ばれる新しいデータストアや RDB 等の従来型のデータストアを同時に利用するケースが増えてくると考えられる．このような状況が広まることを想定し，設計時にデータや処理の特性に基づき最適な配置先データストア決定する技術と，この決定に基づき実行時にデータストアアクセスを切り替える技術とを開発した．評価用設計書を用いて配置技術の有無による適用評価を実施し，その結果 2 割以上作業時間を短縮できることを示した．

Automatic Optimized Use of Multiple Datastores in Cloud Computing Era

Yuji Mizobuchi[†]　Tomohiro Ohtake[†]　Koji Yamamoto[†]
Yoshihide Nomura[†]　and Toshihiro Kodaka[†]

It is expected that more and more applications will use multiple datastores, for example not only traditional RDBs but also NoSQL (“Not only” SQL) datastores such as key value stores (KVS). We developed a method that can allocate data to the most appropriate datastore, based on the data features and process features. We also evaluated the new method by applying it to specification documents prepared beforehand. The result of the evaluation shows the development time is reduced by 20%.

1. はじめに

近年，クラウドコンピューティングの登場に伴い，分散 KVS（Key Value Store）等の NoSQL（Not only SQL）[1]と呼ばれる新しいデータストアが開発されている．例えば，Google が開発する BigTable[2]や Amazon が自社のサービス向けに開発した Dynamo[3]などが有名である．分散 KVS は，複数のサーバ群でデータを分散管理するデータストアで，サーバ間でのデータの一貫性保証を即時に行わない代わりに，複数のサーバでのデータ管理によるスケーラビリティや一部のサーバがダウンしてもサービスを継続できる高可用性を備え，かつ一般に RDB よりも低コストで実現できる特徴がある．具体的には，オープンソースで提供されている Cassandra[4]や CouchDB[5]などがある．この背景には，桁違いな量のトラフィックを有する処理など，従来とは異なる特徴のアプリケーションが登場してきていることが挙げられ，アプリケーションは場合に応じて従来のデータストアと NoSQL データストアの使い分けが必要であると考えられる．このように NoSQL には，RDB だけでなく KVS 等他のデータストアを組合せて使用する，という意味も込められており，今後この NoSQL を実現するための開発手法が望まれている．

本稿では，アプリケーションが利用するデータや処理の特性に応じて，データごとに複数の候補の中から最適な配置先データストアを決定する手法を提案する．また，ソースコードのプログラムロジック部分に修正を加えることなく配置先データストアを変更できる技術を開発し，複数データストアを抽象化した．これにより顧客要件の変更や想定以上のアクセス数になどの非機能要件の変更があっても素早く修正できるメリットがある．さらに，本稿の提案技術であるデータストア自動選択技術の有無による作業工数の比較を行い，本手法の有用性を検証した．

なお，本稿で提案する技術は，Java における Web アプリケーション開発で実績のある Spring Framework[6]と JPA（Java Persistence API）[7]を利用した開発モデルに，機能追加を行うことで実現する．

Spring Framework は，エンタープライズ分野で広く使われているオープンソースのアプリケーションフレームワークで，MVC モデルに基づいて Web アプリケーションを構築するだけでなくバッチ等汎用的なアプリケーションも構築可能である．また，DI（Dependency Injection）機能によりオブジェクト間の依存関係を動的に解決する点が支持されている．

JPA は JSR 220 で標準化された EJB 3.0（Enterprise JavaBeans, Version 3.0）の一部として導入された，Java オブジェクト永続化の API で Java オブジェクトを RDB の形式に変換する O/R マッピング（Object/Relational Mapping）を実現するための仕様である．

評価では，実際にアプリケーションを開発してその効果を測定したが，その際にデータモデル設計には ER 図（Entity Relationship Diagram）を用いた．

2. 従来手法

本節では，Java を利用して，従来の複数のデータストアを用いる Web アプリケーションの，一般的な開発手法を説明する．

[†] 株式会社富士通研究所
Fujitsu Laboratories LTD.

2.1 従来の開発フロー

富士通では，Web アプリケーションの開発手法として，ComponentAA[8]を提案している．ComponentAA におけるデータの配置は，システム要件定義工程で概念 ER 図等を用いてシステム方式設計を行い，それを業務仕様工程でテーブル関連図，テーブル一覧にブレークダウンしていくこととして規定している．図 1 は ComponentAA を参考に作成した ER 図に基づき配置先データストアを決定する開発フローである．

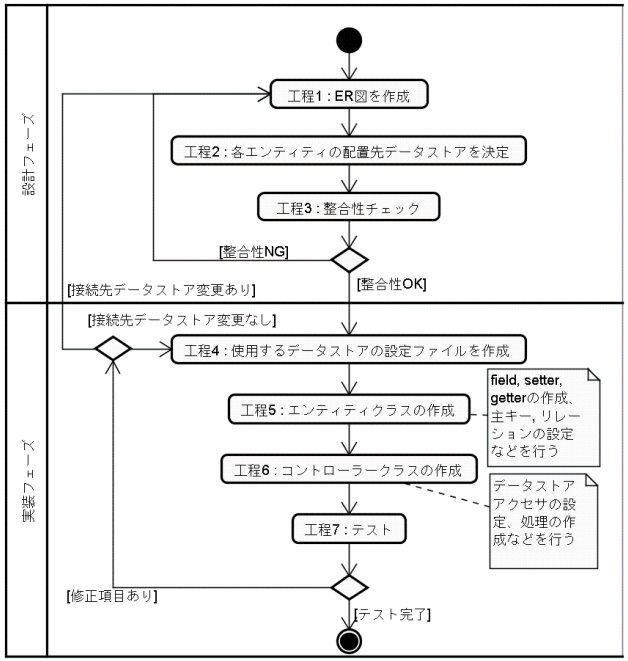


図 1 従来手法の開発フロー

設計フェーズでは，開発者はまず ER 図を作成し（図 1 の工程 1），データの特性を考慮し各エンティティの配置先を決定する（工程 2）．次に配置先のデータストアの制約などに照らし合わせ，エンティティと配置先の整合性を確認する（工程 3）．整合性チェックでは，例えばリレーションを有するエンティティが NoSQL データストアに配置されていないかを確認する．整合性が取れるまで始めから繰り返し行われる．

実装フェーズでは，利用するデータストアごとに JPA 向けの設定ファイル persistence.xml を用意するのが一般的である（工程 4）．次にエンティティクラス（本稿では JPA で規定されているエンティティクラス）と処理を行うクラス（本稿では Spring Framework で規定されているコントローラクラス）を実装する（工程 5, 6）．エンティティクラスには主キーやリレーションを設定し，コントローラクラスでは各メソッドの実装とそこで利用するデータストアアクセサを指定する．なお，JPA ではデータストアアクセサとして EntityManager クラスが利用される．最後に実装したアプリケーションのテストを行う（工程 7）．テスト工程では，例えば，業務要件でロールバックが必要な処理が，実際のデータストアで正しく実施されるか確認する．テストの結果，接続先データストアに変更を要する場合は設計フェーズから，ソースコードの修正が必要な物は実装フェーズから修正を行う．

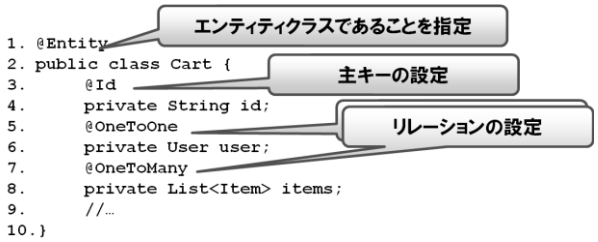
2.2 従来手法を利用した場合のソースコード例

ソースコード 1 は従来手法で作成されたコントローラクラスの例である．このクラスがコントローラクラスであることは一行目の@Controller アノテーションで指定されている．showCartItems()メソッドは，RDB へ接続しユーザのカート内にある商品一覧を取得する．接続先が RDB であることは，EntityManagerFactory を生成する際の引数に紐付く persistence.xml 設定ファイルに記述される URL で指定されている．

```
1. @Controller                                   コントローラクラスであることを指定
2. @RequestMapping(...)
3. public class CartController {
4.
5.   @Autowired(required=true)                   接続先DBに参照用RDBと更新
6.   @Qualifier("Reference_RDB");                用RDBを明示的に指定
7.   EntityManagerFactory emfReferenceRDB;
8.   @Autowired(required=true)
9.   @Qualifier("Update_RDB");
10.  EntityManagerFactory emfUpdateRDB;
11.
12.  @RequestMapping(...)
13.  public ModelAndView showCartItems() {
14.    EntityManager emReferenceRDB = this.emfReferenceRDB.createEntityManager();
15.
16.    String username = (String)session.getAttribute("username");
17.    User user = emReferenceRDB.find(User.class, username);
18.    List<Item> itemsInCart = user.getCart().getItems();
19.    // ...                                     参照用RDB用のデータアクセサで
20.  }                                            処理を実行
21.
22.  @RequestMapping(...)
23.  public ModelAndView addCartItem(@RequestParam("itemId") String itemId) {
24.    EntityManager emUpdateRDB = this.emfUpdateRDB.createEntityManager();
25.
26.    String username = (String)session.getAttribute("username");
27.    User user = emUpdateRDB.find(User.class, username);
28.    List<Item> itemsInCart = user.getCart().getItems();
29.    Item newItem = emUpdateRDB.find(Item.class, itemId);   更新用RDB用のデータアクセサで
30.    itemsInCart.add(newItem);                              処理を実行
31.    emUpdateRDB.mearge(user);
32.    // ...
33.  }
34.}
```

ソースコード 1 従来手法で作成されたコントローラクラスの例

ソースコード 2 は従来手法で作成されたエンティティクラスの例である．このクラスがエンティティクラスであることは一行目の@Entity アノテーションで表現され，データストア上では，変数 id が主キー，User クラスと Item クラスとはリレーションを持つことが示されている．なお，このデータの配置先データストアはこのデータを永続化する処理で使われる EntityManager の接続先データストアである．



ソースコード 2 従来手法で作成されたエンティティクラスの例

3. 課題

3.1 課題 1：エンティティの配置先データストアの決定が困難

2.1 節で説明したように，データの配置先は，データや処理の特性，データストアの特性との整合性の結果決定される．この作業量はエンティティ数や整合性でチェックする内容に応じて多くなり，アプリケーションの規模が大きくなったり，選択対象とするデータストアの数が増えると正しいデータストアにデータを配置できない可能性が高くなる．

3.2 課題 2：配置先データベースの変更に伴うソースコードの修正が煩雑

2.2 節のソースコード 1 では 2 つのデータストア（参照用 RDB と更新用 RDB）へ参照処理と更新処理を行う例を提示した．この例は，ApplicationContext 上に参照用 RDB と更新用 RDB の EntityManagerFactory がインスタンス化されている例で，EntityManagerFactory のインスタンス化は@Qualifier でインスタンスを指定して行われている．そのため，接続先のデータストアが変更されると，ソースコードに修正が伴う．例えば，ソースコード 1 の addCartItem()の接続先データストアを KVS に変更すれば showCartItems()の接続先データストアも KVS へ修正する必要がある．例ではメソッド数が 2 つであり，メソッド内でアクセスするエンティティクラスの数も少ないため影響を受ける範囲は限定的である．しかしながら，メソッド数が多く，個々のメソッドでアクセスするエンティティクラスの数が多くと修正箇所は増大すると考えられる．この他にも影響を受ける要因としてリレーションの有無などもあり，接続先データストアの変更に伴う修正及びテストの工数が増大する課題がある．なお，ApplicationContext は Spring Framework が提供するインスタンスを管理するための機構でインスタンス間の依存関係を動的に解決する機能を有する．

4. 開発技術

本節では，3 節で提示した 2 つの課題を解決する技術を説明する．開発した技術は次の通りで，①で課題 1 を，②で課題 2 を解決する．

① データストア自動選択技術
② データストアアクセス切換技術

図 2 はアプリケーションから複数のデータストアを利用する例である．図中には本技術の適用箇所を示している．例えば，“同期性=即時”（実際の記述形式は異なる）であるデータは，ビルド時に技術①により RDB へ配置が決定され，実行時に技術②によりこのデータを使う処理を更新用 RDB や参照用 RDB へ振り分ける．これにより，厳密な一貫性は必要ではないが大量の参照を必要とするデータは，更新用 RDB の複製である参照用 RDB から取得し，厳密な一貫性を必要とするものだけを更新用 RDB から取得するようにすることで，更新用 RDB への負荷を低減することが可能となる．一方，“同期性=緩やか”で“大量更新=あり”であるデータは，技術①により KVS へ配置が決定され，技術②によりこのデータを使う処理は KVS へ振り分ける．これにより大量の更新・参照が必要となるデータを分散 KVS に配置することができる．

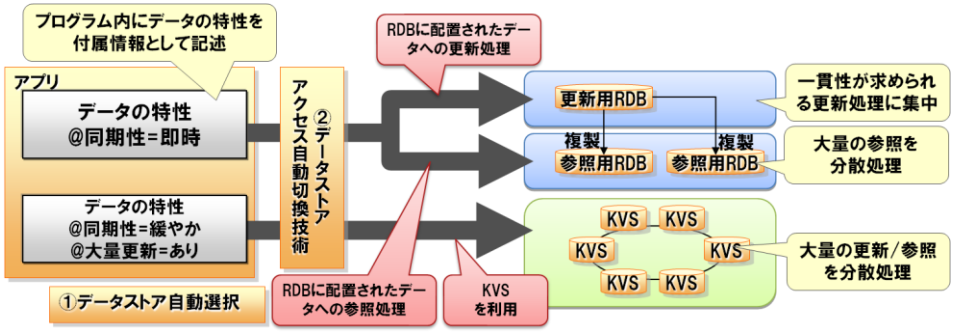


図 2 開発技術の概要

本技術により，アプリケーションのデータや処理の特性から，各データストアの特徴を活かしたデータ管理や処理をソースコードに修正を加えることなく行うことができるようになり，設計ミスを減らすことも実現できる．

図 3 は本手法の開発フローになる．従来手法とは異なり設計フェーズで配置先を決める作業がなくなる．設計+実装フェーズは従来手法の工程 5, 6 とほぼ同じであるが，

データと処理の特性を記述する分工数は多くなる．しかしこの作業によりデータストア選択フェーズでデータの配置先データストアが自動で決定され，従来手法の整合性チェックやテストを削減でき，開発効率化ができる．

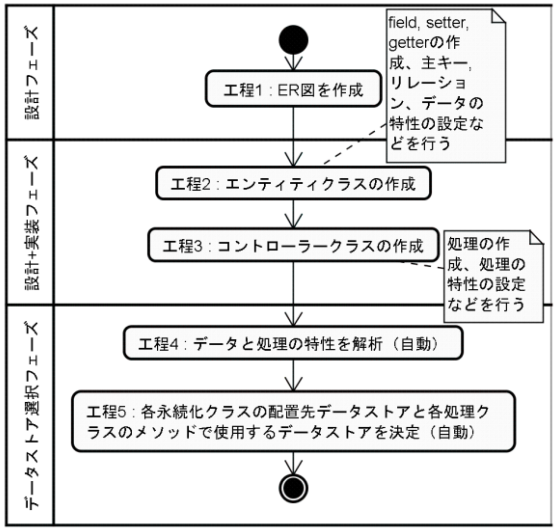


図 3 本手法の開発フロー

現状は選択対象のデータストアを MySQL[9]（RDB）と富士通研究所で開発中の分散 KVS とし，分散 KVS 用の JPA を実装した．ただし，解釈可能な JPQL はサブセットに限定しており，KVS 向けの JPA 実装では JOIN などは行えない．

各技術の詳細は 4.1 節，4.2 節で説明する．

4.1 データストア自動選択技術

データのデータストアへの配置決定には，アプリケーションのデータや処理の特性と利用するデータストアの特性やそれが提供可能な機能との整合性によって決定される．本手法では，まず開発者は設計時にアプリケーション内に Java アノテーションを用いてデータや処理の特性を記述する．データストア自動選択技術はこれらの特性を解釈し，同期性やデータ間のリレーション，読み取り専用等の複合的な条件に基づき最適なデータの配置先を決定する．表 1，

表 2 は特性（アノテーション）と取りうる値の一覧である．図 4 はエンティティクラスに付けられたアノテーションから配置先のデータストアの決定木である．例えば図 2 の①のようにデータ特性“consistency=REALTIME”と記述されたデータの場合，データの同期性を保証しうる RDB に配置されることになる．この他，エンティティクラスにリレーションがあれば RDB へ配置や，同一のコントローラクラスのメソッド内でアクセスされる複数のエンティティクラスは同じデータストアへ配置されるなどのルールがある．

表 1 データの特性(エンティティクラスへのアノテーション)一覧

アノテーション	意味	メンバ	説明	取りうる値	説明
ScaleEntityNature	データの特性を示す	consistency	データの一貫性を指定する	Eventually	更新されたデータが直ぐに参照される必要はない
				Realtime	更新されたデータが直ぐに参照される必要がある
		read	参照処理の特性を記述	Parallel	大規模な並列参照が行われる
		write	更新処理の特性を記述	SeldomUpdate	滅多に更新されない，マスタのように利用される
				FrequentlyUpdate	ロックが必要なデータの更新（書き換え）が頻繁に発生する
				FrequentlyInsert	ロックが不要なデータの追加が主な更新処理

表 2 処理の特性(コントローラクラスへのアノテーション)一覧

アノテーション	意味	メンバ	説明	取りうる値	説明
ScaleEntityAccessPolicy	処理の特性を示す	forReadOnly	処理内で参照するエンティティクラスを指定	エンティティクラスの配列	-
		forUpdate	処理内で更新するエンティティクラスを指定	エンティティクラスの配列	-
		requireRollbackSupport	Atomic 性が求められる処理か指定	boolean	-

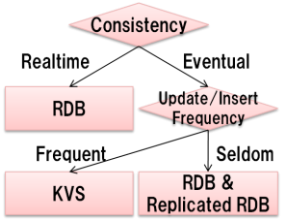


図 4 配置先データストア決定図(一部)

4.2 データストア自動切換技術

本技術は，コントローラクラスの各メソッドでソースコードを変更せずに接続するデータストアの変更を可能にする．本技術を実現するために Spring Framework が提供する DI 機能と AOP（Aspect Oriented Programming）機能を使用した．

図 5 は本技術が適用した例で，本稿が想定している一般的な動作を表している．以下に図中の吹き出しの番号について説明する．

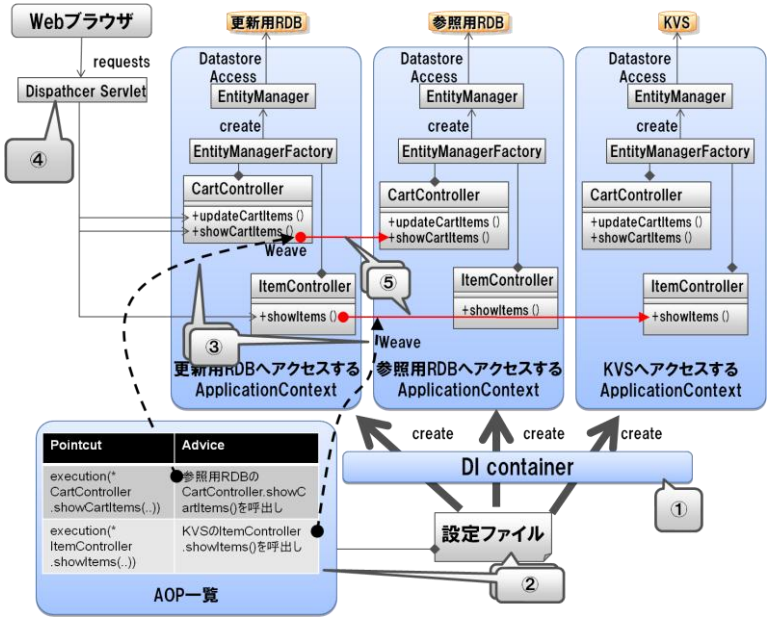


図 5 接続先データストアを切換える仕組み

① DI コンテナはデータストアごとに ApplicationContext を作成し，各 ApplicationContext 内にコントローラクラスや EntityManagerFactory をインスタンス化，依存関係を注入する．この例では更新用 RDB，参照用 RDB および KVS ごとに個別に接続する ApplicationContext が用意される．

② 設定ファイルはアプリケーション中のデータと処理の特性を解析した結果生成され，AOP のポイントカットとアドバイス一覧はこの中に含められる．この例では，CartController クラスの showCartItems()メソッドは参照用 RDB の ApplicationContext 内の同一メソッドがアドバイスとして指定されている．

③ AOP のポイントカットに基づきアドバイスをジョイン・ポイントに織り込む．

④ DispatcherServlet は各リクエストの内容に応じ対応するメソッドを呼び出す．

⑤ 参照用 RDB や KVS に接続するメソッドが呼ばれると既に織り込まれたメソッドを呼び出し，接続先データストアが切り替えられる．

なお，DispatcherServlet は Spring Framework が提供する Servlet クラスであり，リクエストに応じて適切なコントローラのメソッドを呼び出す機能を提供する．

図 6 はアプリケーションのビルドから配備までの処理フローである．開発者がソースコードをリポジトリに登録すると，ビルドツールはソースコード内に記述されているデータや処理の特性から設定ファイルを生成する．作成された war ファイルは Servlet コンテナにデプロイされ，DI コンテナを起動する．DI コンテナは設定情報から必要となる ApplicationContext やコントローラクラスなどを生成する．これでアプリケーションが実行可能な状態になる．

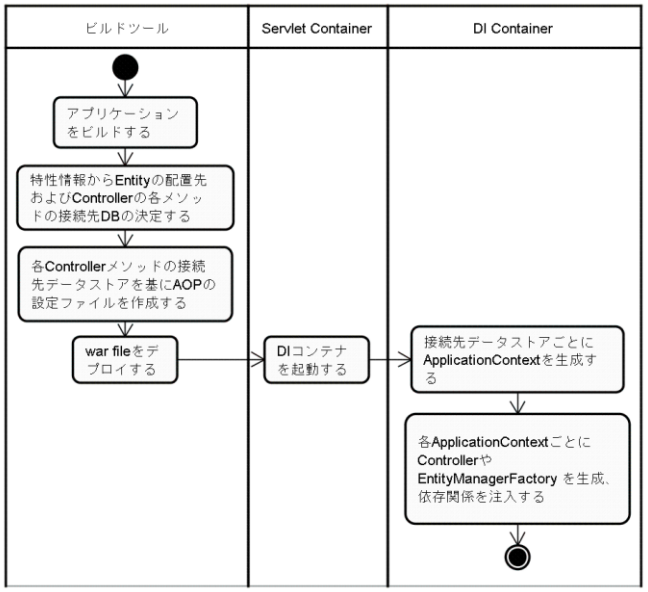


図 6 アプリケーションのビルドから配備されるまでの処理フロー

図 7 は実行時にリクエストを受信後，接続先データストアの切換までに行われる処理の流れを示したものである．リクエストを受け取った DispacherServlet は一旦更新用 RDB 内のコントローラクラスのメソッドを呼び出すが，接続先が更新用 RDB ではない場合，そこに織り込まれたアドバイスを呼び出し，対応する ApplicationContext 内の同一メソッドを呼び出す．

4.3 本手法を利用した場合のソースコード例

ソースコード 3 は本手法で作成するエンティティクラスの例である．この例ではデ

ータの特性に, “consistency=REALTIME”が指定され, 一貫性が厳密であると意味する.

ソースコード 4 は本手法で作成するコントローラクラスの例である. showCartItems()メソッドでは forReadOnly に Cart クラスのみ指定されている. EntityManagerFactory には, データストア自動選択技術で選択されたデータストアに対応した EntityManagerFactory のインスタンスが実行時に DI コンテナにより注入される.

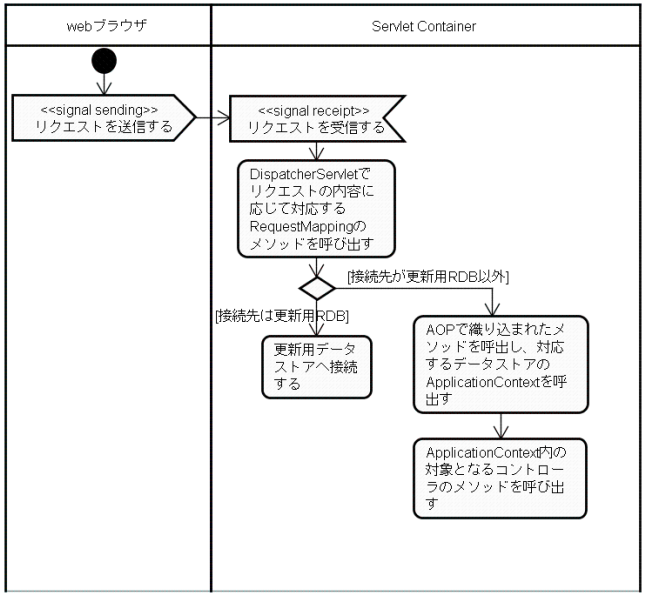


図 7 リクエスト受信からデータストア切換までの処理フロー

データの特性を記述

```
1. @Entity
2. @ScaleEntityNature(
3.      consistency=ScaleEntityConsistencyType.REALTIME,
4.      read=ScaleEntityReadType.PARALLEL,
5.      write={ScaleEntityWriteType.FREQUENTLY_UPDATE,
6.      ScaleEntityWriteType.FREQUENTLY_INSERT})
7. public class Cart {
8.      @Id
9.      private String id;
10.     @OneToOne
11.     private User user;
12.     @OneToMany
13.     private List<Item> items;
14.     //…
15.}
```

ソースコード 3 本手法で作成されたエンティティクラスの例

EntityManagerFactoryを注入

処理の特性を記述

処理の特性を記述

```
1. @Controller
2. @RequestMapping(...)
3. public class CartController {
4.   @Autowired
5.   private EntityManagerFactory emf;
6.
7.   @RequestMapping(...)
8.   @ScaleEntityAccessPolicy(
9.                    forReadOnly={Cart.class},
10.                   forUpdate={},
11.                   requireRollbackSupport=false)
12.  public ModelAndView showCartItems() {
13.    EntityManager em = this.emf.createEntityManager();
14.    String username = (String)session.getAttribute("username");
15.    User user = emReferenceRDB.find(User.class, username);
16.    List<Item> itemsInCart = user.getCart().getItems();
17.    // ...
18.  }
19.
20.  @RequestMapping(...)
21.  @ScaleEntityAccessPolicy(
22.                   forReadOnly={Cart.class},
23.                   forUpdate={Cart.class},
24.                   requireRollbackSupport=false)
25.  public ModelAndView addCartItem(@RequestParam("itemId") String itemId) {
26.    EntityManager em = this.emf.createEntityManager();
27.    String username = (String)session.getAttribute("username");
28.    User user = emUpdateRDB.find(User.class, username);
29.    List<Item> itemsInCart = user.getCart().getItems();
30.    Item newItem = emUpdateRDB.find(Item.class, itemId);
31.    itemsInCart.add(newItem);
32.    emUpdateRDB.mearge(user);
33.    // ...
34.  }
35.}
```

ソースコード 4 本手法で作成されたコントローラクラスの例

5. 計測・評価

本手法のデータストア自動選択技術の有無によって作業工数がどう変わるか計測し, 本手法の有用性を評価した. また, データストア選択作業が本節では, 計測内容, 計測結果および評価について述べる.

5.1 計測内容

データストア自動選択技術は,図 3 におけるデータストア選択フェーズに相当する．実際の開発では，設計+実装フェーズでソースコードの作成およびデータと処理の特性を設計するが，今回はデータストア自動選択技術の評価であるため，ソースコード作成は省略した．代わりにエンティティの一覧と処理の一覧を用意し，それぞれに特性を記述したものを作成した．被験者にはこの一覧表に基づいて配置先データストアを選択して頂いた．

計測では次の作業フローを実施し，それぞれの時間を計測した．このうち被験者に行っていただいた作業は作業 1，2 である．あわせてミスによる修正箇所も計測した．

1. データ特性設計作業：ER 図作成，データの特性の設計，処理の作成，処理の特性の設計
2. 作業 1：各データの配置先と各メソッドの接続先データストアの決定
3. 作業 2：整合性のチェックおよびその結果に伴う修正

今回のデータ特性設計作業では，エンティティ数 20 個，リレーション数 1 個の ER 図，15 個のメソッドを用意した．今回 2 名の方に被験者になっていただいた．両人とも Java のアプリケーション開発の経験はあるが，Spring Framework による開発経験はなかった．

5.2 計測結果

表 3 は計測結果である．

表 3 計測結果

	本手法	被験者 A	被験者 B
データ特性設計作業	115min	115min	115min
作業 1	0min	24min	21min
作業 2	0min	24min	18min
データ配置設計作業	115min	163min	154min
本手法による改善率	-	29.4%	25.3%
修正箇所	0 箇所	2 箇所	0 箇所

本手法による改善率=（作業 1+作業 2）/データ配置設計作業
データ配置設計作業=データ特性設計作業+作業 1+作業 2

5.3 評価

今回の計測結果から 25%から 30%の開発効率化が見込めることを示した．被験者 A では配置先に 2 箇所ミスがあり，原因はリレーションを見逃した結果生じたものである．原因は初歩的なものであり，当人も後で見直した時に即座に修正を行った．設定する特性の数，配置先データストアの数，また ER 図のエンティティ数やリレーション数，さらにメソッド数が増えるとさらに複雑さが増し，今回のようなミスは増えると予想される．本手法を適用することでこのようなミスを低減する効果もある．

6. 関連技術動向

本節では関連技術動向について説明する．

6.1 DataNucleus

DataNucleus[10]は，JDO （Java Data Objects）[11] および JPA の統一した API の実装で，MySQL や HBase[12]など主要なデータストア管理システムに対して永続化を行える．本稿における JPA によるデータアクセスに対応する機能と言える．対応しているデータストアには，ソースコードを変更することなく接続先を変更できる．対象とするデータストアが広範で，11 種のデータストアに対応している．導入事例としては，Google App Engine for Java[13]の Bigtable への永続化機能が挙げられる．本稿も今後様々なデータストアへの対応を考えており，一つの手段として注目している．

6.2 Yahoo! Cloud Serving Benchmark （YCSB）

Cooper ら[14]によると，近年の様々なデータストアの登場により，利用者が適切なデータストアを選択することが困難になっている．比較方法には，使用するデータ項目などの機能的側面やパフォーマンスのような非機能的側面があるが，何れの場合もあてはまるという．その理由として,（1）比較の評価基準が各々で異なること,（2）得られるベンチマーク結果などが他への適用がしづらいなどが挙げられる．これらを踏まえ，異なるクラウドシステムを評価するための標準的なベンチマークとその評価フレームワークを提案している．本稿でもデータストアを選択するときに特性を記述しているが, YSBS と同様に特性を選択するための明確な基準が今のところない. 様々なデータストアの特性を記述できるようにするための一つの技術になると思われ注目している．

6.3 AppScale

AppScale[15]は GAE（Google App Engine）のオープンソース版として知られており，UC Santa Barbara の RACELab で開発が進められている．GAE との API 互換があるため，GAE 上のアプリケーションが動作する．既に 10 種類のデータストアに対応済みで，それぞれのデータストアのベンチマークを計測した結果が発表されている[16]. API は統一されているため切換えは容易だが，アプリケーションやデータの特性に応じた自動的なデータストア決定機能はない．

7. 今後

今後，以下を検討予定である．

7.1 対応データストアの拡張

現状対応しているデータストアは RDB と KVS のみであるが，HBase や Cassandra など他のデータストアへの拡張を考えている．このために各データストアの特性付けが必要になるが，6.2 節の YCSB のような評価基準およびベンチマーク手法のみならず，使用するアプリケーションの特性を考慮した評価手法の確立を目指す．

7.2 データの配置先の変更技術

本稿のデータ自動配置技術を補完する技術として，アプリケーションの運用時にデータの配置先に変更がある場合に，データを移行する技術が考えられる．データの移行期には同時に複数データストアに跨って同一のテーブルのデータが存在することが考えられ，このような状況でも整合性を持ちつつアプリケーションを実行する技術について検討する．

7.3 異なるデータストア間で参照関係を保持する技術

本稿のデータ自動配置技術ではデータストア間を跨って参照関係を保持することができない．この制約により，データや処理の特性の書き方をある程度工夫しないとデータの配置先が RDB に偏る傾向がみられる．当面は RDB と KVS の間で参照関係を保持できる技術を実現し，最終的には汎用性のある仕組みへ展開していきたい．

8. まとめ

本稿では，アプリケーションが利用するデータや処理の特性に応じて，データごとに配置先データストアを決定する手法を提案した．また，ソースコードのプログラムロジック部分に修正を加えることなく配置先データストアを変更できる技術を開発し，複数データストアを抽象化する技術を開発することで，以下の課題を解決した．

- エンティティの配置先データストアの決定が困難
- 配置先データベースの変更に伴うソースコードの修正が煩雑

さらに，本稿の提案技術であるデータストア自動選択技術の有無による作業工数の比較を行い，本手法の有用性を示した．

謝辞 本稿の執筆にあたり，協力頂いた皆様，特に被験者になっていただいた河場さん，堀田さん，高橋さんに，謹んで感謝の意を表する．

参考文献

[1] NoSQL の世界，松下雅和 情報処理学会誌 Vol.51 No.10 Oct.2010 P1327-1331
[2]Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. A Distributed Storage System for Structured Data. In OSDI, pages 205–218, 2006.
[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In SOSP, pages 205–220, 2007.
[4] Cassandra, http://cassandra.apache.org/
[5] CouchDB, http://couchdb.apache.org/
[6] Spring framework, http://www.springsource.org/
[7]Java Persistence API Javadoc, http://download.oracle.com/javaee/5/api/javax/persistence/package-summary.html
[8] ComponentAA 開発標準, http://jp.fujitsu.com/solutions/sdas/technology/develop-guide/1-caa.html
[9]MySQL, http://www.mysql.com/
[10] DataNucleus community. http://www.datanucleus.org/
[11]Java Data Object(JDO), http://www.oracle.com/technetwork/java/index-jsp-135919.html
[12]Hbase, http://hbase.apache.org/
[13]Google App Engine, http://code.google.com/intl/en/appengine/
[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In SoCC '10: Proceedings of the 1st ACM Symposium on Cloud Computing, pages 143--154, New York, NY, USA, June 2010. ACM.
[15] AppScale, http://appscale.cs.ucsb.edu/
[16]C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y.Nomura. Key-Value Datastores Comparison in AppScale. Technical Report 2010-03, UC Santa Barbara, 2010.

正誤表

誤	P4 4.1 節 表 1，*改行* 表 2
正	P4 4.1 節 表 1，表 2