Google App Engineを用いたWebアプリケーションの開発支援に関する研究

2007MI163 中澤祐斗　2008MI091 片岡久幸　2009SE044 服部圭太
指導教員：蜂巣吉成

1　はじめに

現在，多くのクラウドコンピューティングサービスが提供されている．クラウドコンピューティングサービスは，ユーザーがネットワークを利用する環境を用意するだけで，サービスを提供している企業のコンピューティング・リソースの利用が可能になる．クラウドコンピューティングサービスを利用することで，サーバーの新規設置などの設備投資やメンテナンスなどの管理コストをサービス提供側に全て任せることができ，コストを削減することができる．これまで自前でサーバーを用意し，その上でWebアプリケーションを開発・運用していたユーザーもクラウド上に移行することでクラウドコンピューティングサービスの利点を受けることができる．

クラウドコンピューティングサービスでは，一般的にKeyValue型のデータストアが提供されている．KeyValue型のデータストアは，これまで一般的に利用されていたRDBと比べて負荷分散や可用性に優れており，拡張を行うことが容易である．データの増加による機能の低下が少なく，データの登録が多いWebアプリケーションの運用にも向いている．しかし，KeyValue型のデータストアは，テーブルへのJOINコマンドが使えないこと，複雑な条件検索を行うことができないなどの違いがある．データストアの構成自体が異なるので，RDBの設計の考え方をそのまま用いることは相応しくない．RDBから移行を行う場合は，互いのデータベースの違いを理解し，利用するクラウドコンピューティングサービスに適した設計に再構成をする必要がある．クラウドコンピューティングサービスを利用したことがないユーザーがWebアプリケーションを移行をするには敷居が高い．

本研究は，RDBを利用しているWebアプリケーションをKeyValue型のデータストアを利用するWebアプリケーションへの書き換えを支援することを目的とする．Webアプリケーションの機能を維持したままクラウドコンピューティングへ移行するための書き換え方法を提案する．同じ機能を持つWebアプリケーションをクラウドコンピューティング上に再度構築する手間がなくなり，Webアプリケーションのクラウドコンピューティングへの移行を低コストで実現することができる．利用するクラウドコンピューティングサービスには，Googleが提供しているGoogle App Engine for Java[1][2](以下，GAEという)を用いる．書き換えを行う対象として，Javaフレームワーク Strutsで作成されたWebアプリケーションを対象とする．Strutsは基礎的なCRUD機能を提供するフレームワークであり，Webアプリケーションの開発においてクラウドコンピューティングの普及前から現在に至るまで多く利用されている．

2　関連研究・関連技術

2.1　関連研究

[4]はStrutsフレームワークで構成されたWebアプリケーションをSlim3フレームワークを利用したWebアプリケーションに変更を行い，GAEに移行を行うものである．フレームワーク変更に伴う設定ファイルやJavaコードの書き換え方法を示している．データアクセス部については，Like句などの書き換えが必要であることを述べているが，その方法については示していない．本研究ではデータアクセス部をRDBからKeyValue型に書き換える方法を提案し，GAEへの移行方法を示す．

2.2　データストア API

GAEではデータストアを操作するAPIとしてJDOとLow Level APIが提供されている．GAEのLow Level APIは，GAE専用のAPIである．JDOは，データベースへアクセスを行う一般的なAPIであり，GAEにも利用が可能である．GAEのJDOは，内部ではLow Level APIの呼び出しに変換して処理を行なっている．

JDOはオブジェクトの永続化を管理するPersistenceManagerFactoryの初期化に時間がかかり，一般にLow Level APIよりもスピンアップ(GAEにおけるWebアプリケーションの起動)が遅い．GAEでは数分間アクセスがない状態が続くとWebアプリケーションが終了してしまう(スピンダウンという)．これにより，GAEのWebアプリケーションはスピンアップが頻繁に行われることになる．

JDOとLow Level APIを用いて家計簿のアプリケーションをサンプルとして実現し，スピンアップと登録，削除，更新，検索の実行時間を計測した(表1)．

表1　JDOとLow Level APIの実行時間(ms)

処理	JDO	Low Level API
スピンアップ	15246.0ms(674.1)	10151.7(253.1)
登録(1件)	138.5(9.6)	127.3(10.3)
削除(1件)	228.0(11.3)	187.8(15.5)
更新(1件)	202.5(19.6)	114.3(1.7)
検索(1500件)	67.0(19.4)	52.5(24.1)

括弧内は標準偏差

登録，削除，更新，検索はJDOとLow Level APIではほとんど差ないが，スピンアップではJDOがLow Level APIに比べて，5秒ほど遅いことが確認できた．

本研究では，上記の結果を考慮して，Webアプリケーションの書き換えにLow Level APIを用いる．JDOを用いた書き換えについては，6.3節の考察で詳細を示す．

3　移行における課題

3.1　データストアによる制約

RDB を利用している Web アプリケーションを GAE に移行する際に，データストアの違いによるクエリの機能の違いを考慮する必要がある．また，データストアの違いとクエリの機能の違いから作成するテーブルの構成を変更する必要がある．GAE の Bigtable では，データベースへのクエリは簡単な検索機能しか使うことができず，RDB で使用していた SQL クエリの一部の機能が利用できない．一般的な Web アプリケーションで使用されているもので，使用できない SQL クエリの機能には次のものがある [2][3].

1. JOIN による複数テーブルの参照
2. 集約関数 (SUM，MAX，MIN，AVG，COUNT)
3. LIKE を用いた部分一致検索 (前方一致を除く)
4. 複数プロパティへの不等号記号を用いた検索
5. OR による論理結合
6. GROUP BY 句を用いたグループ化

3.2　問題解決の必要性

GAE に Web アプリケーションを移行する場合，データストアへアクセスを行う API の変更だけでは不十分である．Web アプリケーションの機能を変更することなく移行を行うには，GAE で扱えないクエリを Web アプリケーション側での処理や GAE で扱えるクエリを組み合わせて代替する必要がある．しかし，Java コードの追加や変更が必要な箇所が多く，実行時間なども考慮した場合は，データストアの違いを理解する必要がありコストがかかる．そこで，移行を行う際のテーブルの設計やクエリの代替方法を提案することで，Web アプリケーション移行の際の問題点を短時間で解決できる．

4　書き換え方法の提案

3.1 節で述べた Bigtable の制約は，テーブルの再設計とクエリの書き換えの 2 通りの対応方法がある．制約 1, 2 は RDB として定義されたテーブルを Bigtable 用に再設計することで対応できる．制約 2，3，4，5 は SQL クエリを Low Level API によるデータ取得処理と取得後のリストに対する処理 (以下，Web アプリ処理) に書き換えることで対応できる．制約 2 の集約関数は，テーブルの再設計とクエリの書き換えの 2 つの方法で対応が可能なので，アプリケーションの処理内容に応じて使い分けが必要となる．

4.1　テーブルの再設計

- 制約 1：JOIN による複数テーブルの参照
  表 2 は，サンプルに利用した家計簿 Web アプリケーションの元のテーブル構成である．Bigtable では，JOIN による複数テーブルからのデータ取得が出来ないので，表 2 のように外部キーを使用する場合は，category_key の値からカテゴリテーブルを取得する．
  しかし，データストアへのアクセス回数に比例して実行時間やリソースの消費量が増加するので，クエリを何度も実行するのは望ましくない．対象プロパティの更新頻度が低ければ，表 3 のようにテーブルを非正規化し，一つのテーブルに情報をまとめておくことでクエリの回数を減らすことが可能である．

表 2　Key による外部テーブルの参照

買い物テーブル

key	id	description	category_key	amount	year	month	day
xxx	1	商品 A	y	500	2013	1	10

カテゴリテーブル

key	id	name
y	1	食費

表 3　テーブルの非正規化

key	id	description	category_name	amount	year	month	day
xxx	1	商品 A	食費	500	2013	1	10

- 制約 2：集約関数 (SUM)
  集約関数は，専用のテーブル (以下，集約エンティティという) を新たに定義し，計算結果を格納しておくことで対応が可能である．
  図 1 は，集約関数 SUM を集約エンティティを用いて対応した例である．あらかじめ 1 件登録されているテーブルに 1 件追加した場合の集約エンティティの処理を示している．

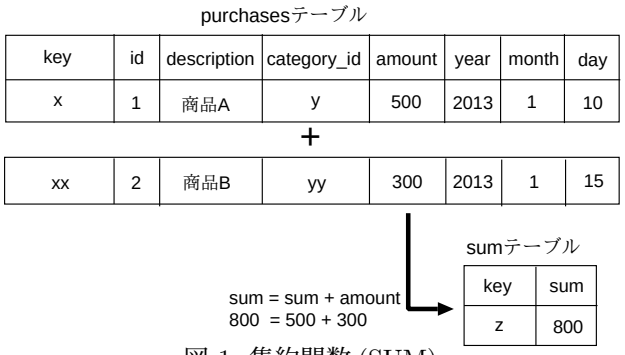

図 1　集約関数 (SUM)

4.2　SQL クエリの書き換え

本研究では，Low Level API を用いて SQL クエリの書き換えを行う．SQL クエリの書き換えに必要な情報として次のものがある．

- 書き換えを行う SQL クエリ
- 動的 SQL の変数または PreparedStatement の set メソッド
- テーブルの再設計によって得られた対応情報

SQL クエリを分割し，Low Level API が提供する機能に置き換えることで単純な CRUD 機能は書き換えることが可能である．また，テーブルの再設計を行ったことでプロパティ名の変更や検索，更新処理の追加を行う必要がある場合もある．書き換えにはテーブルの再設計時に得られた情報も利用しなければならない．

- 制約 2：集約関数 (SUM)

  全件取得後，合計したいプロパティを抜き出し，for 文による繰り返しで合計値を求める (図 2).

```
select sum(プロパティ) from テーブル where 条件
```

```
Query query = new Query("テーブル");
query.addFilter(条件);

List<Entity> list = ds.prepare(query).asList(OPT);
int amount = 0;
for(Entity entity: list){
  int value = Integer.valueOf(entity.getProperty(プロパティ)
                              .toString()).intValue();
  amount = amount + value;
}
```

図 2 SUM 関数

- 制約 3：Like 演算子

  一度全件取得した後，Pattern クラス及び Matcher クラスを用いてキーワードを含むエンティティのみを抜き出すことで部分一致検索を実現する (図 3).

```
select * from テーブル where プロパティ like 検索ワード
```

```
//全件取得
Query query = new Query("テーブル");
List<Entity> rs = ds.prepare(query).asList(OPT);

 //キーワードを含むもののみを抜き出す
for(Entity entity: rs){
    String str = entity.getProperty("プロパティ").toString();
    Pattern pattern = Pattern.compile("検索ワード");
    Matcher matcher = pattern.matcher(str);

    if(matcher.find()){
      //キーワードを含んでいればリストに追加
      list.add(entity);
   }
}
```

図 3 Like 演算子

- 制約 4：複数のプロパティに対する不等号を用いた検索

  Bigtable には，一つのプロパティに対してしか不等号による検索を行えないという制限が存在する．複数のプロパティに対して不等号を用いる場合，まず一つのプロパティに対して検索した後，for 文による繰り返しで 2 つ目以降のプロパティを抜き出し，if 文で対象エンティティのみを取得する (図 4).

```
select * from テーブル where A > 10 and B < 100
```

```
Query query = new Query("テーブル");
query.addFilter("A", FilterOperator.GREATER_THAN, 10);

List<Entity> filterlist = ds.preparety(query).asList(OPT);
List<Entity> list = new ArrayList<Entity>();
for(Entity entity: filterlist){
   //int型に変換
    int value = Integer.valueOf(entity.getProperty("B")
                                .toString()).intValue();
    if(value < 100){
       list.add(filterlist);
    }
}
```

図 4 複数プロパティへの不等式検索

- 制約 5：OR による論理結合

  1 つ目の条件に対して検索を行い，更に 1 つめの条件を除いた状態で 2 つ目の条件に対して検索を行う．その後，それらの結果を結合する (図 5).

```
select * from テーブル where A > 10 or B < 100
```

```
//Filter1 一つ目の条件に対して検索
Query query = new Query("テーブル");
query.addFilter("A", FilterOperator.GREATER_THAN, 10);
List<Entity> list1 = ds.prepare(query).asList(OPT);

//Filter2  2つ目の条件に対して検索
//(1つ目の条件は除く)
    省略
  list2.add(filterlist);
}}

//検索結果を結合
List<Entity> result = new ArrayList<Entity>();
result.addAll(list1);
result.addAll(list2);
```

図 5 OR による論理結合

5 評価

5.1 評価方法

本研究で提案したクエリの書き換えの方法を用いて，Web アプリケーションの書き換えを行う．評価には次の 2 つの Web アプリケーション [5] を用い，Web アプリケーションの稼働成否及び実行速度の比較を行う．

- 家計簿アプリケーション
- ショッピングサイトアプリケーション

4.1 節のテーブルの再設計方法について，正規化したまま外部テーブルを利用する方法と，非正規化を行いテーブルに関連した情報を 1 つのエンティティにまとめる方法の実行速度の比較を行う．外部テーブルの参照を行う場合，アプリケーション側でキャッシュを利用することで，一度取得したデータを再び参照を行う必要がなくなる．

制約 2 の集約関数は，Web アプリ処理で代替する方法と集約エンティティを利用する方法に分けて比較を行う．

5.2 評価結果

評価に利用した 2 つのアプリケーションは，本研究で提案した書き換え方法を利用して，移行前と同様に動作することを確認できた．

表 4 と表 5 に，書き換えを行った Web アプリケーションの実行時間の平均時間の結果を示す．

表 4 の更新処理は，4.1 節の表 2 のカテゴリテーブルの name プロパティと表 3 の category_name プロパティを変更する処理であり，登録された件数の 3 分の 1 を更新している．表 5 の登録処理は，どれも 1 件のデータを登録にかかった実行時間である．

表 4 実行時間結果：テーブル設計 (ms)

		外部テーブル	非正規化
(500 件)	検索	213.2(85.4)	96.2(46.2)
	更新	101.5(15.0)	6017.5(838.9)
(1500 件)	検索	1023.0(121.8)	59.7(35.0)
	更新	125.8(33.2)	21564.5(4023.2)

括弧内は標準偏差

表 5 実行時間結果：集約関数 (ms)

		Web アプリ処理	集約エンティティ
(500 件)	登録	197.3(61.9)	191.2(67.9)
	検索	240.2(71.7)	37.8(1.0)
(1500 件)	登録	136.5(33.2)	225.8(34.5)
	検索	297.7(8.0)	47.2(4.9)

括弧内は標準偏差

6 考察

6.1 テーブル再設計に対する考察

4.1 節のプロパティの更新頻度に関わるテーブルの設計について，検索処理は非正規化を行う方が速く，更新処理では外部テーブルを利用する方が速い．しかし，正規化した場合の検索処理に比べて，非正規化した場合の更新処理の実行時間はデータの件数が増えるほど大幅に遅くなる．一般的な Web アプリケーションであれば正規化したテーブルでの実装を行う方が全体的な実行時間では望ましい．しかし，更新処理をほとんど行わない場合や，検索処理の実行時間を重視する場合では，非正規化を行う方が望ましいことがある．

6.2 集約関数に対する考察

集約関数への対応方法について，集約エンティティを利用する方法が Web アプリ処理を行う方法に比べて，検索処理が大幅に速いことが確認できた．登録処理については，Web アプリ処理を行う方法が速いが，大きな差はない．扱うデータの件数が増加した場合，Web アプリ処理を行う方法の検索処理は，集約エンティティを用いる方法の登録処理よりも実行時間の増加量が大きくなると考えられる．集約関数の機能を代替する場合は，集約エンティティを用いた書き換えが優位である．しかし，条件を指定しての検索などあらかじめ取得範囲を決めておくのが困難なクエリを想定した場合は，Web アプリ処理を行う方法が望ましいと考える．

6.3 JDO を用いた書き換え方法の考察

4.2 節では，SQL クエリに Low Level API を用いた書き換え方法を提案したが，JDO を用いた書き換えについて考察する．SQL クエリの書き換え方法は，データストアからのデータ取得処理と取得したリストに対する処理に大きく分けられる．データ取得処理は JDO を用いても記述できる．Low Level API によるデータ取得ではテーブルを指定し，データに対する条件をフィルタとして設定する．宣言的 JDO でも同様にテーブルを指定し，条件をフィルタとして設定して，データを取得することができる [2]．データ取得後の処理は，Low Level API と JDO でエンティティクラスの名前などに違いがあるが，リストに対する処理は同じである．図 4 の処理を JDO に書き換えた例を図 6 に示す．1 行目から 3 行目が JDO によるデータ取得で，5 行目以降がリストに対する処理である．

```
PersistenceManager pm = PMF.get().getPersistenceManager();
Query query = pm.newQuery("クラス.class");
query.setFilter("A < 10");

List<クラス> filterlist = (List<クラス>)query.execute();;
List<クラス> list = new ArrayList<クラス>();
for(クラス entity: filterlist){
    int value = entity.getB();
    if(value < 100){
        list.add(filterlist);
    }
}
```

図 6 JDO を用いた書き換え例

7 おわりに

本研究では，RDB を利用している Web アプリケーションを KeyValue 型のデータストアを利用する Web アプリケーションへの書き換えを支援することを目的として，データストアの違いを考慮したデータベースのテーブルの再設計方法の提案とクエリの書き換え方法の提案を行った．異なるデータストアへの移行の際に問題となるクエリの機能の差については，元のクエリの機能を代替する手段を GAE のクエリの組み合わせや Web アプリケーション側で実装を行うことで対応を行った．書き換えのサンプルに用いた Web アプリケーションで出現したクエリの機能は実装することができた．

しかし，本研究では SQL の集約関数の 1 つである GROUP BY 句を記述していた場合の SQL クエリの対応ができていない．また，アプリケーションの内容によってはデータアクセス部以外の書き換えが必要となる．例として，Basic 認証を用いたログイン機能が存在した場合，ローカル環境と GAE では実装方法が大きく異なる．加えて，Struts 以外のフレームワークが併用されていた場合もデータアクセス部の書換えのみでは対応出来ない可能性があり，Struts 以外のフレームワークで構成された Web アプリケーションに書き換え方法を適用した場合の有効性について検討が必要であると考えられる．

今後の課題としては，SQL クエリの集約関数の 1 つである GROUP BY 句を使用した SQL 文，及びデータアクセス部以外への対応方法の考案が挙げられる．

参考文献

[1] Google Dvelopers, “ Google App Engine Google Developers, ” https://developers.google.com/appengine/，2012.

[2] 中田秀基，すっきりわかる Google App Engine for Java クラウドプログラミング，SoftBank Creative，2010.

[3] ひがやすお，小川信一，Slim3 on Google App Engine for Java，秀和システム，2010.

[4] 倉持和彦，原田雅史, “ Struts アプリケーションのパブリッククラウド (Google App Engine) への移行に対する考察，”情報処理学会全国大会講演論文集第 72 回平成 22 年 (1)，pp.343-344，2010.

[5] 高安厚思，西川麗，Struts による Web アプリケーションスーパーサンプル，SoftBank Creative，2010.