

オープンソフトウェア利用促進事業

「自治体・企業等の情報システムへの

Ruby 適用可能性に関する調査」

技術検証報告書

平成 21 年 9 月

独立行政法人 情報処理推進機構

## 目次

1. はじめに .....	3
1.1 検証の背景 .....	3
1.2 検証の目的 .....	3
1.3 検証の概要 .....	3
1.4 検証環境 .....	4
2. パフォーマンス検証 .....	5
2.1 パフォーマンス検証概要 .....	5
2.2 パフォーマンス検証実施環境 .....	14
2.3 Ruby 版パフォーマンス検証実施内容詳細 .....	17
2.4 Java 版パフォーマンス検証実施内容詳細 .....	26
2.5 パフォーマンス検証実施結果 .....	31
3. スケーラビリティ検証 .....	42
3.1 スケーラビリティ検証概要 .....	42
3.2 スケーラビリティ検証実施環境 .....	54
3.3 スケーラビリティ検証実施内容詳細 .....	60
3.4 スケーラビリティ検証実施結果 .....	68
4. 考察 .....	74
4.1 パフォーマンス検証 .....	74
4.2 スケーラビリティ検証 .....	74
4.3 まとめ .....	75
付録1 パフォーマンス検証における検証ケースおよびチューニング一覧 .....	76
付録2 スケーラビリティ検証における検証ケースおよびチューニング一覧 .....	82

## 1. はじめに

本技術検証報告書は、「自治体・企業等の情報システムへの Ruby 適用可能性に関する調査」内で実施した技術検証作業を、別途まとめた報告書である。

### 1.1 検証の背景

Ruby (MRI) の言語そのものの処理性能については、例えば業務アプリケーション開発に広く用いられている Java などに比べて、「Ruby は遅い」という批判がある。実際に、他言語や他の処理系とのベンチマークレポートが公開されており、言語そのものの処理性能については確認することができる。(参考：<http://shootout.alioth.debian.org/>)

しかし、実際のアプリケーションの処理プロセスでは、リレーショナルデータベース操作などの「入出力処理」が頻繁に行われており、言語そのものの処理性能もさることながら、外部ライブラリ等を用いた入出力処理の性能が、アプリケーション全体の処理性能を考えた場合の重要なファクターとなってくる。

現状では、このような観点に基づいたベンチマークデータやレポートは少なくとも日本語では公開されていない。そのため、「Ruby でアプリケーションを開発して大丈夫なのか？」との問いに対して定量的な回答を示すことが困難な状況である。

### 1.2 検証の目的

本技術検証では、上記の状況を踏まえ、言語そのものの処理性能について測定・考察するのではなく、Ruby を用いた「入出力処理」や、入出力処理によって構成される「アプリケーションとしての処理プロセス」を想定したベンチマークを取得し、幅広いアプリケーション開発に Ruby を用いる際に参考となる計測データやチューニング手法とその効果を抽出することを目的とした。

### 1.3 検証の概要

本技術検証では、Ruby を用いたアプリケーションの「入出力処理」について、O/R マッピングライブラリを利用したデータベースアクセスの性能に着目し、それを調査するための検証（以下、パフォーマンス検証）と、「アプリケーションとしての処理プロセス」について、実際の Web アプリケーションでどれくらいの性能が出せるか調査するための検証（以下、スケーラビリティ検証）の二つの検証を行った。

#### 1.4 検証環境

本技術検証環境として、富士通株式会社より “Platform Solution Center (プラットフォームソリューションセンター)” に設けられた検証ルームと検証用機器をお貸し頂いた。検証用機器、ネットワーク環境については、「2.2 パフォーマンス検証実施環境」、「3.2 スケーラビリティ検証実施環境」に記載する。



サーバールーム



検証ルーム

Platform Solution Center : <http://jp.fujitsu.com/facilities/psc/>

## 2. パフォーマンス検証

本技術検証で実施した「パフォーマンス検証」について詳細を以下に記載する。

### 2.1 パフォーマンス検証概要

#### (1) 検証概要

本パフォーマンス検証で作成するプログラムは、一般的なアプリケーションで頻繁に行われる「入出力処理」の性能を評価するため、Ruby で作成されたアプリケーションで広く利用されている Ruby on Rails 標準の O/R マッピングライブラリである、「ActiveRecord」を利用して作成し、データベース処理の性能について測定・考察することとした。Ruby では MySQL/Ruby のようなライブラリがあり O/R マッピングライブラリを利用しなくともデータベース操作を行えるが、本検証では実際のアプリケーションで利用するシーンを想定し O/R マッピングライブラリである ActiveRecord を利用してデータベース処理を記述することとした。

パフォーマンス検証で利用するプログラムでは、MySQL に付属する Perl で記述されたベンチマークツールである「Sql-bench」を Ruby で書き換え、データベース処理で扱うデータ量を変化させながら検証を行った。また、Ruby 版だけでなく、Java 版のプログラムも作成し、業務アプリケーション開発に広く用いられている Java との比較も行う。Java 版プログラムで利用する O/R マッピングライブラリには、シェアが高く稼働実績の多い「Hibanate」を利用して実装を行うこととした。また、検証対象として「Sql-bench」を採用する上で「Sql-bench」には様々なシチュエーションを想定したテストプログラムが含まれているが、その中の一部のプログラムを対象として利用することとした。選定したプログラムは以下の通り。

- test-ATIS
- test-transaction

上記二つのプログラムの選定理由としては、多くのシステムで利用されるデータ抽出処理を中心に検討した結果、test-ATIS が SQL で様々な SELECT 文を利用していることから選定した。また、一般的なシステムにおいては、データの抽出処理以外にもデータの格納や更新、削除処理などが利用されるが、そこで利用されるトランザクション処理についても性能を評価すべく、トランザクション処理に特化したテストプログラムである test-transaction を選定した。

また上記の理由を前提とし、さらに以下の検証が出来ることを選定要件とした。

- データ量を変更した検証
- トランザクション処理の検証
- SELECT 文の検証
- WHERE 句を利用して抽出条件を指定した SELECT 文の検証
- 抽出フィールドを指定した SELECT 文の検証

- ・ 範囲検索を利用した SELECT 文の検証
- ・ 複数のテーブルを指定した SELECT 文の検証
- ・ JOIN を利用した SELECT 文の検証
- ・ DISTINCT を利用した SELECT 文の検証
- ・ COUNT を利用した SELECT 文の検証
- ・ GROUP BY を利用した SELECT 文の検証
- ・ SUM を利用した SELECT 文の検証
- ・ AVG を利用した SELECT 文の検証
- ・ MIN、MAX を利用した SELECT 文の検証

ここでは、Ruby 版プログラムのボトルネックについてあらかじめ仮説を立て、改良を加えた Ruby 版プログラムでどこまで性能が伸ばせるか、検証を行う。

また、実際のアプリケーションにおいてはデータ量の増大が、パフォーマンスにどのような影響を与えるかが課題となる。そこで、本検証でもレコード数を増やした際に、パフォーマンスにどう影響するかを調査し、処理速度やリソースの使用状況についても仮説を立て、解決策を施したプログラムを用意して検証することとする。

検証結果として以下「表 1」を取得する。CPU 時間、実行時間に関しては Ruby 及び Java で作成した検証プログラムから取得する。また、メモリ使用量に関しては、プログラム実行時に ps コマンドにより各プロセスのメモリ使用量を取得し、最大メモリ使用量を表へ記載する。

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位： KB)	
	User	System	Total		VSZ	RSS

表 1：検証結果記入表

(2) 検証プログラム概要

前述の通り、本検証で作成するプログラムは、Sql-bench における test-ATIS、test-transaction の二つのプログラムとした。オリジナルのそれぞれのプログラムで実行される SQL は以下の「ア. SQL」である。

また、プログラムで利用するスキーマについては、test-ATIS、test-transaction それぞれオリジナルで提供されているものと同じスキーマを利用することとした。

データ量については、オリジナルで提供されているデータ量と、テーブル毎のデータ量を増やした場合の二通りの場合で検証できるようプログラムを作成した。増やすデータ量はそれぞれのテーブルで 10 万件前後を基本とし、スキーマの制約から 10 万件まで増やすことが出来ないテーブルについては制約の限界まで増やすこととした。

ア. SQL

a. test-ATIS

No	種別	SQL
1	select_simple_join	SELECT city.city_name,state.state_name,city.city_code FROM city,state WHERE city.city_code='MATL' AND city.state_code=state.state_code
2		SELECT city.city_name,state.state_name,city.city_code FROM state,city WHERE city.state_code=state.state_code
3		SELECT month_name.month_name,day_name.day_name FROM month_name,day_name WHERE month_name.month_number=day_name.day_code
4		SELECT month_name.month_name,day_name.day_name FROM month_name,day_name WHERE month_name.month_number=day_name.day_code AND day_name.day_code >= 4
5		SELECT flight.flight_code,aircraft.aircraft_type FROM flight,aircraft WHERE flight.aircraft_code=aircraft.aircraft_code
6	select_join	SELECT airline.airline_name,aircraft.aircraft_type FROM aircraft,airline,flight WHERE flight.aircraft_code=aircraft.aircraft_code AND flight.airline_code=airline.airline_code
7	select_key_prefix_join	SELECT fare.fare_code FROM restrict_carrier,airline,fare WHERE restrict_carrier.airline_code=airline.airline_code AND fare.restrict_code=restrict_carrier.restrict_code
8	select_distinct	SELECT DISTINCT category FROM aircraft
9		SELECT DISTINCT from_airport FROM flight
10		SELECT DISTINCT aircraft_code FROM flight
11		SELECT DISTINCT * FROM fare
12		SELECT DISTINCT flight_code FROM flight_fare
13		SELECT DISTINCT flight.flight_code,aircraft.aircraft_type FROM flight,aircraft WHERE flight.aircraft_code=aircraft.aircraft_code
14		SELECT DISTINCT airline.airline_name,aircraft.aircraft_type FROM aircraft,airline,flight WHERE flight.aircraft_code=aircraft.aircraft_code AND flight.airline_code=airline.airline_code
15		SELECT DISTINCT airline.airline_name,aircraft.aircraft_type FROM flight,aircraft,airline WHERE

		flight.aircraft_code=aircraft.aircraft_code AND flight.airline_code=airline.airline_code
16	select_group	SELECT day_name.day_name,day_name.day_code,COUNT(*) FROM flight_day,day_name WHERE day_name.day_code=flight_day.day_code GROUP BY day_name.day_name,day_name.day_code ORDER BY day_name.day_code
17		SELECT day_name.day_name,COUNT(*) FROM flight_day,day_name WHERE day_name.day_code=flight_day.day_code GROUP BY day_name.day_name
18		SELECT month_name,day_name from month_name,day_name WHERE month_number=day_code AND day_code>3 GROUP BY month_name,day_name
19		SELECT day_name.day_name,flight_day.day_code,COUNT(*) FROM flight_day,day_name WHERE day_name.day_code=flight_day.day_code GROUP BY flight_day.day_code,day_name.day_name ORDER BY flight_day.day_code
20		SELECT SUM(engines) FROM aircraft
21		SELECT AVG(engines) FROM aircraft
22		SELECT AVG(engines) FROM aircraft WHERE engines>0
23		SELECT COUNT(*),MIN(pay_load),MAX(pay_load) FROM aircraft WHERE pay_load>0
24		SELECT MIN(flight_code),MIN(flight_code) FROM flight
25		SELECT MIN(from_airport),MIN(to_airport) FROM flight
26		SELECT COUNT(*) FROM aircraft WHERE pay_load>10000
27		SELECT COUNT(*) FROM aircraft WHERE pay_load<>0
28		SELECT COUNT(*) FROM flight WHERE flight_code >= 112793
29		SELECT COUNT(if(pay_load,1,NULL)) FROM aircraft
30		SELECT STD(engines) FROM aircraft
31		SELECT from_airport,to_airport,AVG(time_elapsed) FROM flight WHERE from_airport='ATL' AND to_airport='BOS' GROUP BY from_airport,to_airport
32		SELECT city_code, AVG(ground_fare) FROM ground_service WHERE ground_fare<>0 GROUP BY city_code
33		SELECT COUNT(*), ground_service.city_code FROM ground_service GROUP BY ground_service.city_code
34		SELECT category,COUNT(*) AS totalnr FROM aircraft WHERE engines=2 GROUP BY category having totalnr>4
35		SELECT category,COUNT(*) FROM aircraft WHERE engines=2 GROUP BY category HAVING COUNT(*)>4
36		SELECT flight_number,range_miles,fare_class FROM aircraft,flight,flight_class WHERE flight.flight_code=flight_class.flight_code AND flight.aircraft_code=aircraft.aircraft_code AND range_miles<>0 AND (stops=1 OR stops=2) GROUP BY flight_number,range_miles,fare_class
37		SELECT DISTINCT from_airport.time_zone_code,to_airport.time_zone_code,(FLOOR(a rrival_time/100)*60+MOD(arrival_time,100)-FLOOR(departure_t ime/100)*60-MOD(departure_time,100)-time_elapsed)/60 AS

	time_zone_diff FROM flight,airport AS from_airport,airport AS to_airport WHERE flight.from_airport=from_airport.airport_code AND flight.to_airport=to_airport.airport_code GROUP BY from_airport.time_zone_code,to_airport.time_zone_code,arrival_time,departure_time,time_elapsed
38	SELECT DISTINCT from_airport.time_zone_code,to_airport.time_zone_code,MOD((FLOOR(arrival_time/100)*60+MOD(arrival_time,100)-FLOOR(departure_time/100)*60-MOD(departure_time,100)-time_elapsed)/60+36,24)-12 AS time_zone_diff FROM flight,airport AS from_airport,airport AS to_airport WHERE flight.from_airport=from_airport.airport_code AND flight.to_airport=to_airport.airport_code AND MOD((FLOOR(arrival_time/100)*60+MOD(arrival_time,100)-FLOOR(departure_time/100)*60-MOD(departure_time,100)-time_elapsed)/60+36,24)-12 < 10
39	SELECT from_airport,to_airport,range_miles,time_elapsed FROM aircraft,flight WHERE aircraft.aircraft_code=flight.aircraft_code AND to_airport NOT LIKE from_airport AND range_miles<>0 AND time_elapsed<>0 GROUP BY from_airport,to_airport,range_miles,time_elapsed
40	SELECT airport.country_name,state.state_name,city.city_name,airport_service.direction FROM airport_service,state,airport,city WHERE airport_service.city_code=city.city_code AND airport_service.airport_code=airport.airport_code AND state.state_code=airport.state_code AND state.state_code=city.state_code AND airport.state_code=city.state_code AND airport.country_name=city.country_name AND airport.country_name=state.country_name AND city.time_zone_code=airport.time_zone_code GROUP BY airport.country_name,state.state_name,city.city_name,airport_service.direction ORDER BY state_name
41	SELECT airport.country_name,state.state_name,city.city_name,airport_service.direction FROM airport_service,state,airport,city WHERE airport_service.city_code=city.city_code AND airport_service.airport_code=airport.airport_code AND state.state_code=airport.state_code AND state.state_code=city.state_code AND airport.state_code=city.state_code AND airport.country_name=city.country_name AND airport.country_name=state.country_name AND city.time_zone_code=airport.time_zone_code GROUP BY airport.country_name,state.state_name,city.city_name,airport_service.direction ORDER BY state_name DESC
42	SELECT airport.country_name,state.state_name,city.city_name,airport_service.direction FROM airport_service,state,airport,city WHERE airport_service.city_code=city.city_code AND airport_service.airport_code=airport.airport_code AND state.state_code=airport.state_code AND state.state_code=city.state_code AND airport.state_code=city.state_code AND airport.country_name=city.country_name AND airport.country_name=state.country_name AND city.time_zone_code=airport.time_zone_code GROUP BY

	airport.country_name,state.state_name,city.city_name,airport_service.direction ORDER BY state_name
43	<pre> SELECT from_airport,to_airport,fare.fare_class,night,one_way_cost,rnd_trip_cost,class_days FROM compound_class,fare WHERE compound_class.fare_class=fare.fare_class AND one_way_cost &lt;= 825 AND one_way_cost &gt;= 280 AND from_airport='SFO' AND to_airport='DFW' GROUP BY from_airport,to_airport,fare.fare_class,night,one_way_cost,rnd_trip_cost,class_days ORDER BY one_way_cost </pre>
44	<pre> SELECT engines,category,cruising_speed,from_airport,to_airport FROM aircraft,flight WHERE category='JET' AND engines &gt;= 1 AND aircraft.aircraft_code=flight.aircraft_code AND to_airport NOT LIKE from_airport AND stops&gt;0 GROUP BY engines,category,cruising_speed,from_airport,to_airport ORDER BY engines DESC </pre>

b. test-transaction

No	種別	SQL
1	insert_commit	SET AUTOCOMMIT=0 INSERT INTO bench1 values (0,9999,'A',0,0) . . INSERT INTO bench1 values (9999,0,'J',370.333333333333,0) COMMIT
2	insert_autocommit	SET AUTOCOMMIT=1 INSERT INTO bench2 values (0,9999,'A',0,0) . . INSERT INTO bench2 values (9999,0,'J',370.333333333333,0)
3	insert rollback	SET AUTOCOMMIT=0 INSERT INTO bench1 values (10000,10000,'K',370.37037037037,0) . . INSERT INTO bench1 values (10099,9901,'B',374.037037037037,0) ROLLBACK INSERT INTO bench1 values (10100,9900,'C',374.074074074074,0) . . ROLLBACK
4	Update rollback	UPDATE bench1 set updated=2 WHERE idn=0 . . UPDATE bench1 set updated=2 WHERE idn=99 ROLLBACK UPDATE bench1 set updated=2 WHERE idn=100 . . UPDATE bench1 set updated=2 WHERE idn=9999 ROLLBACK
5	delete rollback	DELETE FROM bench1 WHERE idn=0 . . DELETE FROM bench1 WHERE idn=99 ROLLBACK . . DELETE FROM bench1 WHERE idn=9999 ROLLBACK
6	Update commit	UPDATE bench1 set updated=1 WHERE idn=0 . . UPDATE bench1 set updated=1 WHERE idn=9999 COMMIT
7	update autocommit	SET AUTOCOMMIT=1 UPDATE bench2 set updated=1 WHERE idn=0 . . UPDATE bench2 set updated=1 WHERE idn=9999

8	delete commit	SET AUTOCOMMIT=0 DELETE FROM bench1 WHERE idn=0 ・ ・ DELETE FROM bench1 WHERE idn=9999 COMMIT
9	delete autocommit	SET AUTOCOMMIT=1 DELETE FROM bench2 WHERE idn=0 ・ ・ DELETE FROM bench2 WHERE idn=9999

イ. スキーマとデータ量

a. test-ATIS

テーブル名	カラム数	レコード数	
		デフォルト	追加後
City	5	11	11,000
State	3	63	63
month_name	2	12	12
day_name	2	7	7
Flight	14	579	579
Aircraft	13	135	46,656
airline	3	314	314
restrict_carrier	2	614	115,833
fare	8	534	534
flight_day	3	448	134,400
ground_service	4	33	132,000
flight_class	2	2,895	104,803
flight_fare	2	2,998	119,764
airport	6	9	49,786
airport_service	5	14	140,000
compound_class	9	143	47,988

#### b. test-transaction

テーブル名	カラム数	レコード数	
		デフォルト	追加後
bench1	5	10,000	100,000
bench2	5	10,000	100,000

#### ウ. 利用方法

##### a. test-ATIS

test-ATIS は Ruby 及び Java で実装し検証を行う。それぞれのプログラム言語で作成されるプログラムについては、抽出結果を CSV ファイルとして出力し、Ruby、Java 版両方の抽出結果が同一であることを担保し、パフォーマンス測定時の計測で公平性が保たれるように配慮した。

また、Ruby 版の ActiveRecord の一般的な利用方法で記述したプログラムについては、そのプログラムのボトルネックからパフォーマンスを改善する方法が無いか仮説を立て、仮説を元にチューニングを施した上で、再度検証を行う。

##### b. test-transaction

test-transaction も test-ATIS 同様 Ruby 及び Java で実装し検証を行う。なお、このプログラムでは抽出データが無い削除処理などは CSV ファイルとしての出力は行わない。また、test-transaction も test-ATIS 同様、Ruby で作成されるプログラムについてはチューニングを施したプログラムも検証する。

## 2.2 パフォーマンス検証実施環境

### (1) ハードウェア

本検証で利用した環境は次の通り。



1台のアプリケーションサーバ上で Ruby 版 Sql-bench と Java 版 Sql-bench を実行し、1台のデータベースサーバに対して処理を行う。

#### 【アプリケーションサーバ】

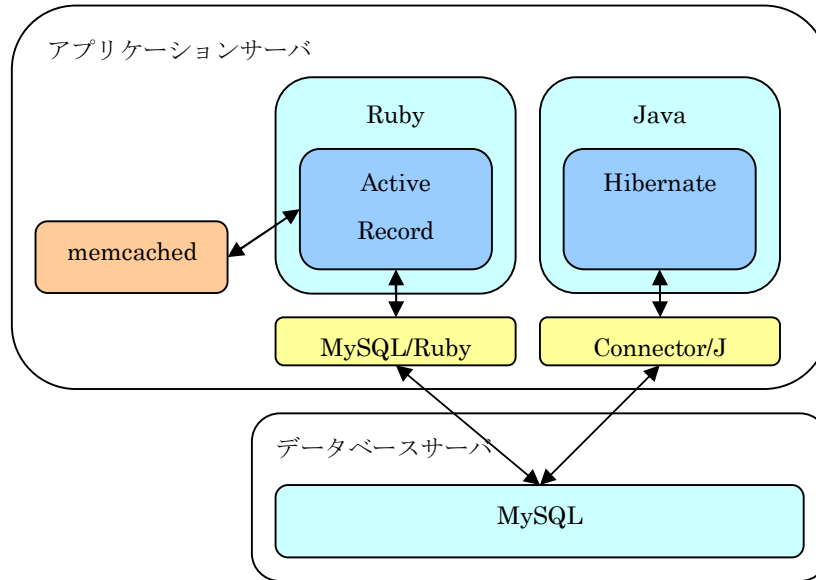
APサーバ	1台
製品名(シャーシ)	FUJITSU PRIMERGY BX600 S3
製品名	FUJITSU PRIMERGY BX620 S4
CPU	QuadXeon X5355 2.66GHz/2x4MB *2
メモリ	8GB
DISK	147GB(RAID1)

#### 【データベースサーバ】

DBサーバ	1台
製品名	FUJITSU PRIMERGY RX300 S4
CPU	QuadXeon5460 3.16GHz/12MB *2
メモリ	4GB
DISK	294GB(RAID5)

## (2) ソフトウェア

本検証で利用したソフトウェアの環境は次の通り。



1台のアプリケーションサーバ上で Ruby 版 Sql-bench と Java 版 Sql-bench を実行し、O/R マッピングライブラリを用いて 1 台のデータベースサーバに対して処理を行う。Ruby 版 Sql-bench の改良版である memcached 利用版に関しては、アプリケーションサーバ上で起動している memcached を利用する。memcached はデータベースへの問い合わせ結果を一時的にメモリにキャッシュし、データベースへのアクセス回数を減らすために利用されるソフトウェアであり、Web サイトを高速化する仕組みとして現在多くのサイトで利用されている。また、本プログラムで使用している Connector/J は Sun Microsystems 社が提供している MySQL 用の公式 JDBC ドライバであり、Java プログラミング言語から MySQL に接続するときには使用するものである。

### 【アプリケーションサーバ】

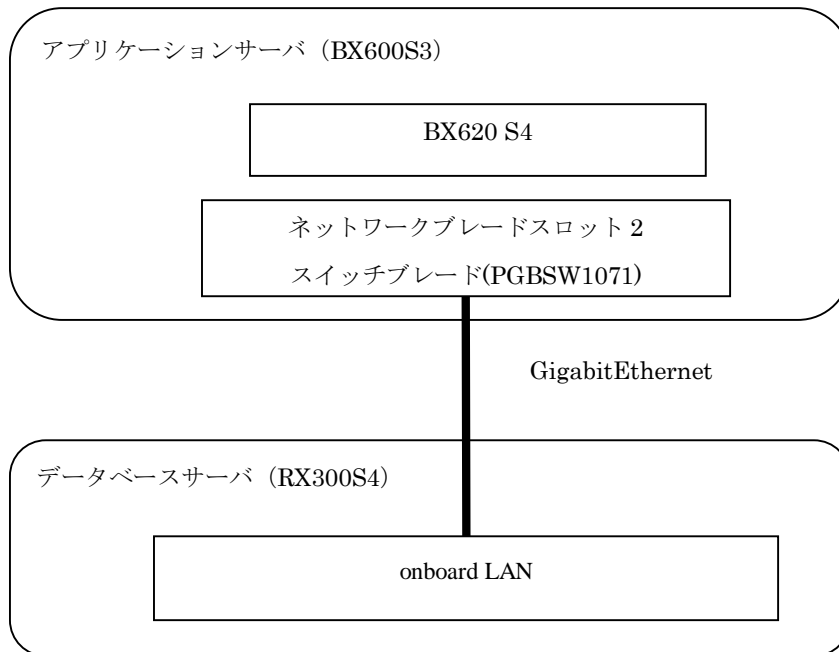
OS	Red Hat Enterprise Linux 5.2 (for x86) 2.6.18-92.el5.PAE 32 ビット
Ruby	ruby Version 1.8.7-p72
Ruby Framework	Ruby on Rails Version 2.2.2
Ruby ConnectorAPI	mysql-ruby Version 2.8
Java	Java Version 1.6.0_11
Java Framework	Hibernate Version 3.3.1
Java ConnectorAPI	JDBC:MySQL Connector/J Version 5.1.7
その他	Memcached Version 1.2.6 Libevent Version 1.4.9

【データベースサーバ】

OS	Red Hat Enterprise Linux 5.2 (for Intel64) 2.6.18-92.el5 64 ビット
RDBMS	MySQL Version 5.1.30

(3) ネットワーク

本検証で利用したネットワーク構成は次の通り。



アプリケーションサーバとして利用する BX620 S4 は、BX600S3 シャーシに内蔵されている 2 つのネットワーク・スイッチに内部的に接続されており、サーバブレードと外部 LAN との間を接続している。本検証では、内蔵スイッチ 2 を使用し、データベースサーバとの間をギガビット・イーサケーブルで接続している。サーバ間にネットワーク機器を挟んでいない為、ボトルネックが発生しにくい構成としている。アプリケーションサーバからデータベースサーバへの接続には IP アドレスを使用し、データベースサーバの MySQL 設定では DNS を無効にしており、ホスト名逆引きなどの余計なコストをかけない設定にしている。また、全ての試験に関してネットワークにエラーがないことを `ifconfig` コマンドで確認している。

## 2.3 Ruby 版パフォーマンス検証実施内容詳細

Ruby で作成された検証プログラムについて、`test-ATIS`、`test-transaction` それぞれ以下のプログラムを用意し検証を行った。

### ○ Ruby 版 `test-ATIS` プログラム一覧

- ・ ノーマル版  
ActiveRecord の一般的な利用方法で記載されたプログラム
- ・ `find` メソッド改良版  
ActiveRecord の `find` のオプションを変更し、効率の良い SQL を出力するよう改善したプログラム
- ・ `find_by_sql` 利用版  
ActiveRecord#`find_by_sql` メソッドを利用し、オリジナルの `test-ATIS` と同じ SQL を出力するよう改善したプログラム
- ・ `memcached` 利用版  
ActiveRecord で取得したデータを、`memcached` を利用してキャッシングし、取得データを再利用するよう改善したプログラム

### ○ Ruby 版 `test-transaction` プログラム一覧

- ・ ノーマル版  
ActiveRecord の一般的な利用方法で記載されたプログラム
- ・ `find` メソッド改良版  
効率の良い SQL を出力するよう改善したプログラム

`test-ATIS`、`test-transaction` それぞれに改善を施したプログラムについて、ノーマル版が Java と比べて明らかにパフォーマンスが劣ると予想し用意した。それぞれ、O/R マッピングライブラリの仕様で非効率的な SQL を生成することによるパフォーマンス低下、O/R マッピングライブラリの SQL 生成の処理速度が遅いことによるパフォーマンス低下、そして、大量のデータベースアクセスによるパフォーマンス低下、といった原因を仮説立て改善プログラムを作成した。各プログラムの詳細は次に示す。

#### (1) `test-ATIS` プログラム詳細

##### ア. 各プログラムでの共通事項

`test-ATIS` は、Ruby 添付ライブラリの `Benchmark` クラスを利用し、ベンチマークを取得する。ベンチマークは以下のようにデータ抽出処理毎に取得する。

```
atis-measure.rb
```

```
require 'benchmark'

module AtisMeasure
  .
```

```

.
.
def start(title)
  result = Benchmark.measure do
    yield
  end
  if @@results[title]
    @@results[title] += result
  else
    @@titles << title
    @@results[title] = result
  end
end
end
.
.
.
end

```

atis.rb

```

records = []

AtisMeasure.start("Query 1 :") do
  records = City.find(:all, :include => :state,
                    :conditions => ["city.city_code = :city_code", {:city_code =>
'MATL'}]).map do |city|
    [city.city_name, city.state.state_name, city.city_code]
  end
end

GenerateCSV.generate(records, 1)

```

また、Java で作成された test-ATIS プログラムのデータ抽出が同一であることを確認するため、上記のように「GenerateCSV.generate」を利用して抽出したデータを CSV ファイルとして出力することとした。なお、CSV ファイルを出力する処理はデータベースアクセスとは関連性が無いため、本検証で取得するベンチマークからは外すこととした。

イ. ノーマル版（ActiveRecord の一般的な利用方法で記載されたプログラム）

このプログラムでは、O/R マッピングライブラリのメリットの一つである SQL を意識しない方法で実装を行った。ActiveRecord の find メソッドのオプションには SQL の断片を記述しなければいけないオプションがあるが、極力そのような指定は使わずに実装した。この方針により、オリジナルのプログラムで発行される SQL とは違う SQL でデータ抽出を行うが、CSV ファイルとして出力するまでのところではオリジナルと同じ抽出結果になるよう作成した。

この例では include オプションと conditions オプションを利用している。

atis.rb

```
records = City.find(:all,
                   :include => :state,
                   :conditions => ["city.city_code = :city_code",
                                   {:city_code => 'MATL'}])
```

オリジナルの SQL と上記 find メソッドで実行される SQL は以下の通り。

オリジナルの SQL

```
select city.city_name,state.state_name,city.city_code from city,state
where city.city_code='MATL' and city.state_code=state.state_code
```

上記 find メソッドで実行される SQL

```
SELECT * FROM `city` WHERE (city_code = 'MATL')
SELECT * FROM `state` WHERE (`state`.`state_code` = 'GA')
```

このように、オリジナルの SQL と比べて実行される SQL の数や、O/R マッピングライブラリを利用することで出力される SQL の内容が違う場合の性能を調査する。

ウ. find メソッド改良版 (ActiveRecord の find のオプションを変更し、効率の良い SQL を出力するよう改善したプログラム)

test-ATIS ではいくつかのデータ抽出処理において ActiveRecord#find メソッドのオプションである include オプションを利用しており、通常の問い合わせより多く SQL が発行される。また、include を利用することで複数のテーブルのカラムを取得できるが、その反面、取得するカラムの指定は行うことができない。そのため、この変更では joins や select オプションを利用し、取得カラム数の軽減や SQL の問い合わせ数を減らす方針で改良を行う。改良前のプログラムは以下の通り。

atis.rb

```
records = City.find(:all,
                   :include => :state,
                   :conditions => ["city.city_code = :city_code",
                                   {:city_code => 'MATL'}])
```

このように「ActiveRecordの一般的な利用方法で記載されたプログラム」では include オプションを利用し SQL を意識しない方法で実装を行っており、include オプションを指定することにより、関連するテーブルに対して SELECT 文を発行し、その結果をもとにレシーバ (検索対象のオブジェクト) のテーブルに対して SELECT 文が発行されている。つまり、SELECT 文が 2 回発行され非効率である (ただし、find メソッドの引数に conditions を指定した場合は JOIN 句を含む SQL が発行され include で指定するモデルの数により SELECT 文の回数は変わる)。また、include を指定した場合は取得するカラムの指定ができないため、全てのカラムを

取得することになり、メモリの使用量を消費することになる。抽出するレコード件数が少ない場合は問題になることは無いが、抽出するレコードが多い場合にはその分リソースを圧迫することになる。

以上のことを踏まえ、この改善ではSQL文の発行回数を減らしデータベースアクセスを少なくすることで処理速度の向上を図り、取得するカラムを制限することでリソースの消費を抑えるようにした。改善したプログラムは以下の通り。

#### atis-improved.rb

```
records = City.find(:all,
                    :select => "city.city_name,state.state_name,city.city_code",
                    :joins => :state,
                    :conditions => ["city.city_code = :city_code",
                                   {city_code => 'MATL'}])
```

このプログラムでは include オプションの利用を止め、joins オプションを利用した。また、select オプションを利用して取得するカラムを制限した。

エ. find\_by\_sql 利用版 (ActiveRecord#find\_by\_sql メソッドを利用し、オリジナルの test-ATIS と同じ SQL を出力するように改善したプログラム)

find\_by\_sql を利用することで ActiveRecord での SQL 文の生成が無くなり、またオリジナルの test-ATIS と同じ SQL を発行することでパフォーマンスの向上を図る。実際のプログラムは以下の通り。

#### atis-find-by-sql.rb

```
records = City.find_by_sql("select city.city_name,state.state_name,city.city_code
                           from city,state where city.city_code='MATL' and
                           city.state_code=state.state_code")
```

オ. memcached 利用版 (ActiveRecord で取得したデータを、memcached を利用してキャッシングし、取得データを再利用するように改善したプログラム)

データベースアクセスそのものの負荷を減らすことでパフォーマンスの向上を図ることを検討した。そこで、検索結果をメモリ上に格納することができる memcached を利用することにした。memcached を利用することで、2 回目の検索以降はメモリ上のデータを利用するようになり、データの内容が変わらず何度も同じ検索を行う場合は有効であると考えた。実際のプログラムは以下の通り。

#### atis-memcache.rb

```
cache = MemCache.new(<memcached サーバ名>, <オプション>)

unless records = cache["1"]
  records = データ抽出処理
  cache["1"] = records
end
```

このプログラムでは `memcached` に検索対象のキャッシュが存在するか確認し、キャッシュが無ければデータを抽出し、キャッシュが存在するとデータ抽出処理を省くよう実装している。今回の検証では同じ問い合わせを繰り返し実行するため、二回目以降のデータ抽出処理は行われず、処理速度が向上することが見込まれる。

## (2) test-transaction プログラム詳細

### ア. 各プログラムでの共通事項

test-transaction は、test-ATIS と同様に Ruby 添付ライブラリ Benchmark を利用し、ベンチマークを取得する。

#### transaction.rb

```
t = Benchmark.measure do
  <各種処理>
end
```

また、データの格納や更新処理に関しては「GenerateCSV.generate」を利用して抽出したデータを CSV ファイルとして出力することとし、削除処理に関しては対象となるデータが無いため、CSV ファイルの出力処理は行わないこととした。なお、CSV ファイルを出力する処理はデータベースアクセスとは関連性が無いため、本検証で取得するベンチマークからは外す。

トランザクション処理についても以下のように ActiveRecord の transaction メソッドを利用して実現する。

#### test-transaction.rb

```
Bench1.transaction do
  <各種処理>
end
```

### イ. ノーマル版 (ActiveRecord の一般的な利用方法で記載されたプログラム)

このプログラムでは、test-ATIS と同様に O/R マッピングライブラリのメリットの一つである SQL を意識しない方法で実装を行った。INSERT 文を利用するデータの格納処理では ActiveRecord の create メソッドを利用し、更新処理では update メソッド、削除処理では destroy メソッドを利用した。作成したプログラムは以下の通り。

#### transaction.rb

```
.
.
.
# データ格納処理
@class_object.create(:region => region,
                     :idn => id,
                     :rev_idn => rev_id,
                     :grp => grp,
                     :updated => 0)
.
.
.
# データ更新処理
```

```

@class_object.update(id, :updated => 1)
.
.
.
# データ削除処理
@class_object.destroy(id)
.
.
.

```

#### ウ. find メソッド改良版（効率の良い SQL を出力するよう改善したプログラム）

test-transaction ではデータの挿入・更新・削除の処理がトランザクションで実行される。ActiveRecord の一般的な利用方法で開発したプログラムの場合、更新・削除処理に関しては、更新・削除を行うメソッドを実行する前に、更新・削除の対象となるレコードを ActiveRecord のオブジェクト(モデル)としてインスタンス化して実行するため、インスタンス生成のコストがかかる。

よって test-transaction の改良を施したプログラムでは、上記のインスタンス化がボトルネックであると仮説を立て、更新処理は ActiveRecord.update\_all メソッドを、削除処理は ActiveRecord.delete、または ActiveRecord.delete\_all メソッドを使用することとした。なお、挿入処理に関しては同様のメソッドが用意されていないため、改善は行わない。挿入処理に限っては ActiveRecord::Schema.execute メソッドを利用し、直接 SQL を実行する改良を施すこととした。

#### transaction-improved.rb

```

# データ格納処理
def insert_data
  rev_id = @loop_count
  sql = "insert into #{@class_object.table_name} "
  sql += "(idn ,rev_idn, region, grp, updated) VALUES "
  values = []
  @loop_count.times do |id|
    rev_id -= 1
    grp = id / OPT_GROUPS
    region = (65 + id % OPT_GROUPS).chr
    values << "#{id}, #{rev_id}, '#{region}', #{grp}, 0"
  end
  sql += values.join(" , ")
  ActiveRecord::Base.connection.execute sql
end

# データ更新処理
def update_data
  @loop_count.times do |id|
    @class_object.update_all("updated = 1", "idn = #{id}")
  end
end

# データ削除処理
def delete_data
  @loop_count.times do |id|

```

```
@class_object.delete(id)
end
end
```

### (3) Ruby 版プログラム検証手順

#### ア. 共通手順

データベースサーバのクエリキャッシュは無効にした状態で検証を行う

#### イ. オリジナルデータ量での test-ATIS 検証手順

ノーマル版 test-ATIS プログラム

1. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS normal --loop 100
```

find メソッド改良版 test-ATIS プログラム

1. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS improved --loop 100
```

find\_by\_sql 利用版 test-ATIS プログラム

1. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS sql --loop 100
```

memcached 利用版 test-ATIS プログラム

1. memcached を再起動する
2. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS memcache --loop 100
```

#### ウ. データ量を増やした場合の test-ATIS 検証手順

ノーマル版 test-ATIS プログラム

1. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS normal --loop 100 --data increase
```

find メソッド改良版 test-ATIS プログラム

1. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS improved --loop 100 --data increase
```

find\_by\_sql 利用版 test-ATIS プログラム

1. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS sql --loop 100 --data increase
```

memcached 利用版 test-ATIS プログラム

1. memcached を再起動する
2. 起動パラメータを次の通り指定し、100 回繰り返し処理を実行する

```
$ test-ATIS memcache --loop 100 --data increase
```

エ. オリジナルデータ量での test-transaction 検証手順

ノーマル版 test-transaction プログラム

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ test-transactions normal --record 10000 --transaction 100
```

find メソッド改良版 test-transaction プログラム

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ test-transactions improved --record 10000 --transaction 100
```

オ データ量を増やした場合の test-transaction 検証手順

ノーマル版 test-transaction プログラム

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ test-transactions normal --record 100000 --transaction 100
```

find メソッド改良版 test-transaction プログラム

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ test-transactions improved --record 100000 --transaction 100
```

## 2.4 Java 版パフォーマンス検証実施内容詳細

Java で作成された検証プログラムについて、test-ATIS、test-transaction 共に以下のプログラムを用意し検証を行った。

### ○ Java 版 sql-bench プログラム

- a. test-ATIS
- b. test-transaction

本プログラムは Hibernate + Java Persistence API を使用方法でプログラムを記述した。Java Persistence API を利用している理由としては、各種フレームワークを利用する場合には Hibernate 単品での実装は一般的ではなく、Java Persistence API を使用方法が一般的なためである。また、一般的な使用方法の EntityManager を使用方法ではなく、Annotation Configuration を使用方法をとっている。Annotation Configuration の選択理由としては、設定ファイル (hibernate.cfg.xml) を jar ファイル外出しにすることが可能であり、実行環境ごとの設定をコンパイルなしで行えることにある。

test-ATIS、test-transaction プログラムは同一の jar ファイルに同梱して作成した。各プログラムの詳細は次に示す。

### (1) Java 版 sql-bench 詳細

#### ア. 各プログラムでの共通事項

test-ATIS、test-transaction 両プログラムともベンチマークを取得する。ベンチマークは以下のようにデータ抽出処理毎に取得する。

TestBase.java

```
import jp.co.sstyle.util.Benchmark;
public abstract class TestBase {
    ...
    ...
    public void addBenchResult(String keyname, BenchmarkResults result)
    {
        BenchmarkResults tmpResult = RunAllTests.benchResults.get(keyname);
        if (tmpResult != null) {
            result.setTestCount(result.getTestCount()
                + tmpResult.getTestCount());
            result.setUserTime(result.getUserTime() + tmpResult.getUserTime());
            result.setSysTime(result.getSysTime() + tmpResult.getSysTime());
            result.setCpuTime(result.getCpuTime() + tmpResult.getCpuTime());
            result.setTaskTime(result.getTaskTime() + tmpResult.getTaskTime());
        }
        RunAllTests.benchResults.put(keyname, result);
    }
    ...
    ...
}
```

## TestATIS.java

```
startTime = System.nanoTime();
startBench = new Benchmark();
objectQ1 = cityDao.findBy();
count = count + 1;
endBench = new Benchmark(startTime);
endBench.setName("Query 1");
endBench.setTestCount(1);
addResults("Query 1", startBench, endBench);
```

## CityDao.java

```
public City findBy() throws HibernateException
{
    City city = null;
    try {
        city = (City) session.load(City.class, "MATL");
    } catch (HibernateException hex) {
        RunAllTests.logStackTrace(hex.getStackTrace(), hex.getMessage());
    }
    return city;
}
```

### イ. Java 版 sql-bench test-ATIS 詳細

test-ATIS の実装方針としては、レコードを取得する処理 (HibernateSession#load メソッド、HibernateCriteria#list メソッド)において、オプションなどを極力省き、簡潔に記載するようにしているが、最終的に実行される SQL は、オリジナルの test-ATIS プログラムで利用されている SQL と可能な限り同じ SQL が実行されるように実装を行った。ただし、HibernateSession#load メソッド、HibernateCriteria#list メソッドで実行不可な SQL を実行する必要がある場合 (※1、※2) に限り、クエリ生成部分のみ HibernateSession#createSQL メソッドを用いて SQL 文の直接記述によるクエリ実行をしている。ただし、この場合であってもデータ取得処理に関しては Hibernate を介した O/R マッピングを行っている。

※1 SQL 関数を使用している SQL

Query. 29,30,34,35,37,38

※2 実装困難な SQL

Query. 6, 7, 36, 39, 44, 40, 41, 42

実際のプログラムは以下の通り。

## CityDao.java

```
public List<City> findAll() throws HibernateException
{
    List<City> cities = null;
    try {
        Criteria criteria = session.createCriteria(City.class);
        criteria.setFetchMode("state", FetchMode.JOIN);
        criteria.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
        cities = (List<City>) criteria.list();
    } catch (HibernateException hex) {
        RunAllTests.logStackTrace(hex.getStackTrace(), hex.getMessages());
    }
    return cities;
}
```

## AircraftDao.java

```
public Double stdEngines() throws HibernateException
{
    Double stdEngines = null;
    try {
        String sql = "select std(a.engines) as stdEngines from aircraft a";
        List<Double> std = (List<Double>) session.createSQLQuery(sql)
            .addScalar("stdEngines", Hibernate.DOUBLE).list();
        stdEngines = std.get(0);
    } catch (HibernateException hex) {
        RunAllTests.logStackTrace(hex.getStackTrace(), hex.getMessages());
    }
    return stdEngines;
}
```

## ウ. Java 版 sql-bench test-transaction 詳細

transaction の実装方針として次の項目を考慮し作成を行った。

レコード挿入処理 : `HibernateSession` の `save` メソッドを利用する

レコード更新処理 : `HibernateSession` の `update` メソッドを利用する

レコード削除処理 : `HibernateSession` の `delete` メソッドを利用する

トランザクション処理 : `HibernateSession` の `begin` メソッドを利用し、`HibernateSession#begin` から `HibernateSession#rollback` もしくは `HibernateSession#commit` に囲まれた範囲内でトランザクション内の処理を記述する。

コミット処理 : `HibernateSession#commit` を利用する

ロールバック処理 : `HibernateSession#rollback` を利用する

実際のプログラムは以下の通り。

## Bench1Dao.java

```
public void create(int idn, int revIdn, String region, int grp, byte updated)
    throws HibernateException
{
    try {
        Bench1 entity = new Bench1();
        entity.setIdn(idn);
        entity.setRevIdn(revIdn);
        entity.setRegion(region);
        entity.setGrp(grp);
        entity.setUpdated(updated);
        session.save(entity);
    } catch (HibernateException hex) {
        RunAllTests.logStatckTrace(hex.getStackTrace(), hex.getMessages());
    }
}
```

```
public void updateUpdated(int idn, byte updated) throws HibernateException
{
    try {
        Bench1 entity = null;
        entity = (Bench1) session.load(Bench1.class, idn);
        entity.setUpdated(updated);
        session.update(entity);
    } catch (HibernateException hex) {
        RunAllTests.logStatckTrace(hex.getStackTrace(), hex.getMessages());
    }
}
```

```
public void delete(int idn) throws HibernateException
{
    try {
        Bench1 entity = null;
        entity = (Bench1) session.load(Bench1.class, idn);
        session.delete(entity);
    } catch (HibernateException hex) {
        RunAllTests.logStatckTrace(hex.getStackTrace(), hex.getMessages());
    }
}
```

## (2) Hibernate 設定ファイル

本検証プログラムでを使用した `hibernate.cfg.xml` 設定内容は次の通りである。

```
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="hibernate.show_sql">>false</property>
<property name="hibernate.max_fetch_depth">3</property>
<property name="hibernate.use_sql_comments">>false</property>
<property name="hibernate.format_sql">>true</property>
<property name="hibernate.use_outer_join">>true</property>
<property name="hibernate.hbm2ddl.auto">none</property>
<property name="hibernate.connection.autocommit">>true</property>
<property name="hibernate.jdbc.batch_size">30</property>
<property name="hibernate.cache.use_second_level_cache">>false</property>
```

`jdbc.batch_size` プロパティ に関しては、Hibernate は新しく挿入するためのオブジェクトをキャッシュしていくが、その際に `OutOfMemoryException` が発生してしまう。そこで、`jdbc.batch_size` に値をセットし、JDBC バッチを利用するように設定を行った。

`use_second_level_cache` プロパティに関しては、挿入を行ったオブジェクトは再度利用されることが本プログラム中では存在しないため、二次キャッシュを設定ファイルレベルで無効にし、メモリ使用量の抑制を行った。

上記 2 点以外の設定内容に関しては Hibernate を使う上での標準的な設定である（コネクションプーリングは使用していない）。

### (3) Java 版プログラム検証手順

#### ア. 共通手順

データベースサーバのクエリキャッシュは無効にした状態で検証を行う

#### イ. オリジナルデータ量での test-ATIS 検証手順

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ sql-bench.jar -runtest atis -fast -datafiledir '/var/scripts/java-sql-bench/Data/ATIS/'
```

#### ウ. データ量を増やした場合の test-ATIS 検証手順

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ sql-bench.jar -runtest atis -fast -datafiledir '/var/scripts/java-sql-bench/Data/ATIS-increase/'
```

#### エ. オリジナルデータ量での test-transaction 検証手順

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ sql-bench.jar -runtest transactions -fast -transaction -loopcount 10000
```

#### オ. データ量を増やした場合の test-transaction 検証手順

1. 起動パラメータを次の通り指定し、処理を実行する

```
$ sql-bench.jar -runtest transactions -fast -transaction -loopcount 100000
```

## 2.5 パフォーマンス検証実施結果

パフォーマンス検証実施結果を以下に示す。

### (1) Ruby 版 test-ATIS の検証結果

ア. オリジナルデータ量での test-ATIS の検証結果

#### a. ノーマル版 test-ATIS プログラム

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位： KB)	
	User	System	Total		VSZ	RSS
select simple join	2.94	0.02	2.96	3.19	28,332	23,364
select join	3.91	0.01	3.92	4.07		
select key_prefix_join	28.45	0.49	28.95	53.22		
select distinct	15.66	0.08	15.74	16.44		
select group	30.34	0.13	30.47	32.69		
Total time				109.61		

#### b. find メソッド改良版 test-ATIS プログラム

処理項目	CPU 時間(単位：秒)			実行時間 (単位：秒)	メモリ使用量(単位： KB)	
	User	System	Total		VSZ	RSS
select simple join	1.00	0.01	1.01	1.23	28,080	23,060
select join	1.25	0.00	1.25	1.44		
select key_prefix_join	7.92	0.01	7.93	8.69		
select distinct	5.83	0.04	5.87	7.13		
select group	2.64	0.03	2.67	4.37		
Total time				22.86		

#### c. find\_by\_sql 利用版 test-ATIS プログラム

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位:KB)	
	User	System	Total		VSZ	RSS
select simple join	1.04	0.01	1.05	1.21	27,988	23,024
select join	1.27	0.00	1.27	1.40		
select key_prefix_join	7.87	0.01	7.89	8.64		
select distinct	5.44	0.02	5.46	6.80		
select group	2.45	0.04	2.49	4.17		
Total time				22.22		

#### d. memcached 利用版 test-ATIS プログラム

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位： KB)	
	User	System	Total		VSZ	RSS
select simple join	0.10	0.01	0.11	0.11	28,312	23,328
select join	0.09	0.01	0.10	0.12		
select key_prefix_join	0.42	0.03	0.45	0.47		
select distinct	0.38	0.04	0.42	0.39		
select group	0.31	0.03	0.34	0.40		

Total time				1.49		
------------	--	--	--	------	--	--

イ. データ量を増やした場合の test-ATIS の検証結果

a. ノーマル版 test-ATIS プログラム

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
select_simple_join	2.84	0.03	2.87	8.33	138,332	133,200
select_join	3.84	0.02	3.86	4.01		
select_key_prefix_join	27.53	0.51	48.35	57.12		
select_distinct	232.73	1.15	233.88	245.71		
select_group	291.26	1.17	292.43	387.33		
Total time				702.50		

b. find メソッド改良版 test-ATIS プログラム

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
select_simple_join	1.05	0.01	1.05	5.04	28,856	23,776
select_join	1.26	0.01	1.27	1.48		
select_key_prefix_join	7.87	0.01	7.88	8.65		
select_distinct	5.63	0.02	5.65	18.76		
select_group	2.65	0.03	2.69	52.89		
Total time				86.82		

c. find\_by\_sql 利用版 test-ATIS プログラム

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
select_simple_join	0.99	0.01	1.00	5.01	28,864	23,852
select_join	1.27	0.01	1.28	1.46		
select_key_prefix_join	7.82	0.03	7.85	8.61		
select_distinct	5.41	0.02	5.43	18.08		
select_group	2.41	0.03	2.45	52.91		
Total time				86.07		

d. memcached 利用版 test-ATIS プログラム

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
select_simple_join	0.12	0.01	0.13	0.18	28,792	23,788
select_join	0.10	0.00	0.11	0.12		
select_key_prefix_join	0.45	0.01	0.46	0.48		
select_distinct	0.42	0.02	0.44	0.55		
select_group	3.55	0.23	3.77	4.32		
Total time				5.65		

(2) Java 版 test-ATIS の検証結果

ア. オリジナルデータ量での test-ATIS の検証結果

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位：KB)	
	User	System	Total		VSZ	RSS
select simple join	0.82	0.09	0.90	1.23	1,210,308	81,444
select join	0.16	0.02	0.18	0.33		
select key_prefix_join	0.40	0.03	0.43	1.14		
select distinct	0.85	0.07	0.92	2.33		
select group	0.87	0.22	1.09	3.25		
Total time				8.28		

イ. データ量を増やした場合の test-ATIS の検証結果

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位：KB)	
	User	System	Total		VSZ	RSS
select_simple_join	0.76	0.06	0.82	6.07	1,218,076	73,224
select_join	0.12	0.01	0.13	0.34		
select_key_prefix_join	0.39	0.03	0.42	1.18		
select_distinct	0.82	0.08	0.89	13.62		
select_group	0.93	0.30	1.23	51.88		
Total time				73.09		

(3) Ruby 版 test-transaction の検証結果

ア. オリジナルデータ量での test-transaction の検証結果

a. ノーマル版 test-transaction プログラム

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位： KB)	
	User	System	Total		VSZ	RSS
insert_commit	5.85	0.10	5.95	7.24	42,820	37,848
insert_autocommit	6.66	0.21	6.87	10.20		
insert_rollback	6.51	0.09	6.60	8.04		
update_rollback	6.97	0.15	7.12	10.10		
delete_rollback	3.20	0.15	3.34	6.22		
update_commit	7.00	0.17	7.17	10.03		
update_autocommit	7.75	0.28	8.03	13.02		
delete_commit	3.19	0.16	3.35	6.06		
delete_autocommit	3.66	0.28	3.94	8.62		
Total time				79.53		

b. find メソッド改良版 test-transaction プログラム

処理項目	CPU 時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位： KB)	
	User	System	Total		VSZ	RSS
insert_commit	0.02	0.00	0.02	0.15	42,100	37,104

insert_autocommit	0.03	0.00	0.03	0.15		
insert_rollback	6.43	0.09	6.53	7.94		
update_rollback	7.02	0.17	7.19	10.16		
delete_rollback	3.15	0.16	3.31	6.20		
update_commit	0.50	0.05	0.56	1.88		
update_autocommit	0.52	0.06	0.58	2.08		
delete_commit	0.77	0.07	0.84	2.09		
delete_autocommit	0.81	0.07	0.88	2.42		
Total time				33.07		

イ. データ量を増やした場合の test-transaction の検証結果

a. ノーマル版 test-transaction プログラム

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
insert_commit	58.66	0.86	59.53	72.51	174,672	169,688
insert_autocommit	81.09	2.11	83.20	115.85		
insert_rollback	94.72	0.83	95.56	110.90		
update_rollback	93.14	1.60	94.74	124.25		
delete_rollback	40.56	1.50	42.06	70.93		
update_commit	94.61	1.61	96.22	124.66		
update_autocommit	111.37	2.92	114.29	163.71		
delete_commit	51.77	1.49	53.26	80.58		
delete_autocommit	51.25	2.76	54.01	100.89		
Total time				964.28		

b. find メソッド改良版 test-transaction プログラム

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
insert_commit	0.37	0.01	0.39	1.31	174,876	169,804
insert_autocommit	0.68	0.00	0.68	1.58		
insert_rollback	93.79	0.83	94.63	109.87		
update_rollback	92.03	1.66	93.69	123.16		
delete_rollback	40.10	1.48	41.58	70.69		
update_commit	5.78	0.65	6.43	19.72		
update_autocommit	7.14	0.68	7.82	22.85		
delete_commit	12.00	0.67	12.67	25.44		
delete_autocommit	11.12	0.65	11.77	27.24		
Total time				401.87		

(4) Java 版 test-transaction の検証結果

ア. オリジナルデータ量での test-transaction の検証結果

処理項目	CPU 時間(単位 : 秒)			実行時間 (単位 : 秒)	メモリ使用量(単位 : KB)	
	User	System	Total		VSZ	RSS
insert_commit	1.02	0.10	1.12	2.30	1,222,620	83,500
insert_autocommit	0.90	0.09	0.99	2.46		

insert_rollback	0.06	0.00	0.06	0.07		
update_rollback	1.64	0.11	1.75	3.29		
delete_rollback	0.92	0.12	1.05	2.56		
update_commit	0.90	0.16	1.06	3.94		
update_autocommit	1.24	0.18	1.41	4.64		
delete_commit	0.87	0.19	1.06	3.81		
delete_autocommit	1.04	0.18	1.22	4.43		
Total time				27.49		

イ. データ量を増やした場合の test-transaction の検証結果

処理項目	CPU時間(単位：秒)			実行時間 (単位： 秒)	メモリ使用量(単位：KB)	
	User	System	Total		VSZ	RSS
insert_commit	2.25	0.76	3.01	15.44	1,217,136	267,336
insert_autocommit	4.12	0.84	4.96	19.12		
insert_rollback	0.27	0.00	0.27	0.28		
update_rollback	8.34	1.07	9.41	24.17		
delete_rollback	7.84	1.09	8.93	23.62		
update_commit	9.11	1.76	10.87	39.17		
update_autocommit	10.90	1.92	12.82	45.31		
delete_commit	7.82	1.85	9.67	37.48		
delete_autocommit	9.92	1.90	11.82	43.93		
Total time				248.52		

## (5) 性能比較

### ア. オリジナルデータ量での test-ATIS

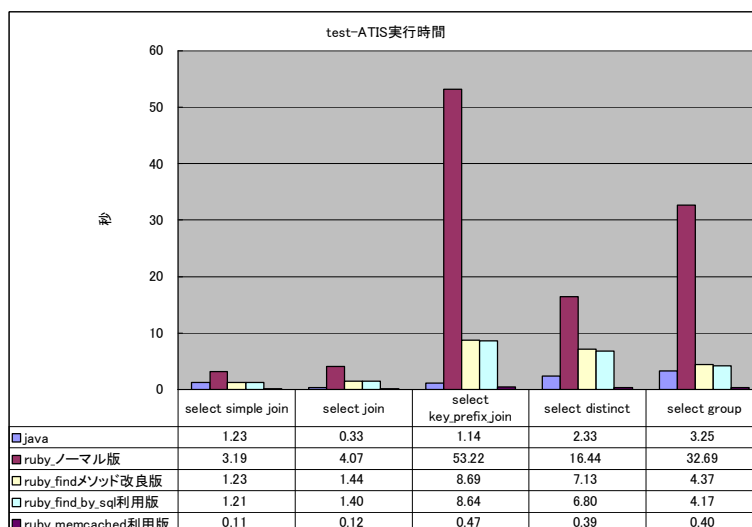


図 1 : SQL 種別実行時間 (test-ATIS)

データ取得処理に関して、図 1 より、Ruby で作成したノーマル版「ActiveRecord の一般的な利用方法で記載されたプログラム」の実行時間と Java 版プログラムの実行時間に大きな差があることが確認できる。特に select join に関しては Java 版の 12 倍、select\_key\_prefix\_join に関しては 46 倍、select distinct に関しては 7 倍、select group に関しては 10 倍程度処理時間がかかっていることがグラフより確認できる。

ノーマル版「ActiveRecord の一般的な利用方法で記載されたプログラム」に関しては select join、select\_key\_prefix\_join、select distinct、select group において極端な性能差がみられた。

select join、select\_key\_prefix\_join の各処理時間が長くなる原因としては、include を使用して関連する 3 テーブルのデータを結合する際に、それぞれのテーブルに対して SQL 文を発行しているためと考えられる。

select distinct は、SQL レベルで distinct を使用しておらず、重複するレコードも一旦 Ruby 側にオブジェクトとして取得し、Ruby 側で重複するデータを削除する処理を行っている。そのため、取得する元データが多くなること、また Ruby レベルでの distinct に該当する処理に時間がかかることが、処理時間が長くなる原因として考えられる。

select group は、SQL レベルで group を使用しておらず、group 化できるレコードも一旦 Ruby 側にオブジェクトとして取得し、Ruby 側で group 化を行っている。また、order by が指定されてない場合は、Ruby レベルでソートを行っている。これらの理由で処理時間が長くなったと考えられる。

これらを踏まえ改善を施した find メソッド改良版「ActiveRecord の find のオ



次にリソースの使用状況（図 2、図 3）を比較すると CPU に関して Java 版は複数の CPU コアを利用できているのに対し、Ruby 版は 1 コアのみ利用となった。これは MRI(本検証で用いているインタプリタ)の実装仕様が影響していると考えられる。

#### イ. オリジナルデータ量での test-transaction

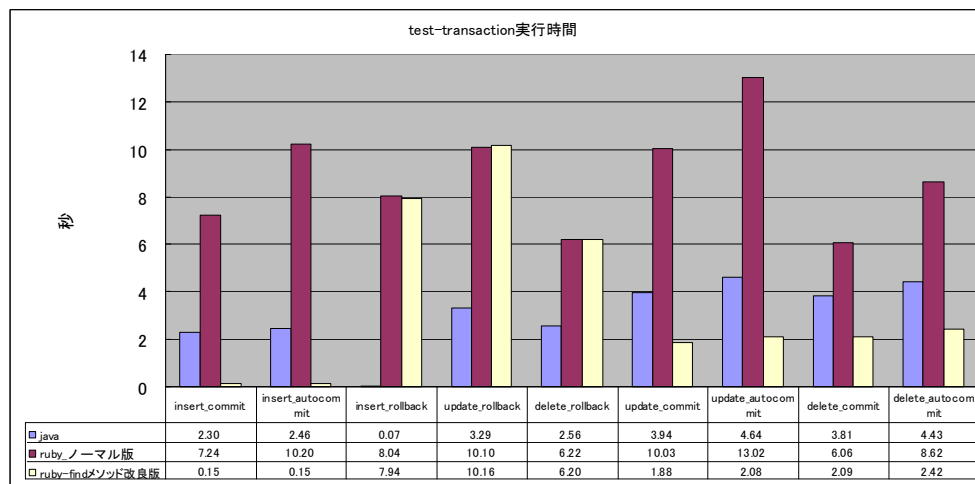


図 4 : SQL 種別実行時間 (test-transaction)

一般的なシステムにおけるトランザクションを想定した test-transaction のデータ更新処理に関しても、Ruby と Java を比較すると全ての処理において性能差があることが図 4 の Java と Ruby (ノーマル版) の比較から確認できる。

ノーマル版「ActiveRecordの一般的な利用方法で記載されたプログラム」の処理時間が長くなる原因としては、ActiveRecordが生成するSQLの数やSQL生成部分の速度が挙げられる。更新・削除処理は、各処理前に対応するレコードをインスタンス化し、それに対して更新・削除処理を行うという処理になるが、この場合に発行されるクエリとして、SELECT 文とUPDATE文またはDELETE文が発行される。また、挿入処理は処理の性質上SELECT文の発行は無いが、生成されたレコードに対応するオブジェクトの生成処理が行われる。つまり、インスタンス化のコストがかかるため処理速度に影響を与える。この点に関しては以下の改善を行うことで処理速度の向上が見込まれる。

1. 更新処理はActiveRecord.update\_all メソッドを使用
2. 削除処理はActiveRecord.delete、またはActiveRecord.delete\_all メソッドを使用
3. 挿入処理に関してはActiveRecord::Schema.execute メソッドを使用し直接 SQL を実行

#### 4. 挿入処理はmultiple insert

findメソッド改良版「効率の良いSQLを出力するよう改善したプログラム」は、rollback処理であるinsert\_rollback、update\_rollback、delete\_rollbackを除くすべての処理に関して性能が改善し、Java版より性能が上回っていることが、図4より確認できる（Java版は、Rubyノーマル版同様に、更新・削除処理は、各処理前に対応するレコードをインスタンス化し、それに対して更新・削除処理を行うという処理であるため、比較対象としては有効ではないが、改善効果を見る上での指標としている）。なお、Rollback処理に関しては改善方法が存在しないため、改善前、改善後と変化がみられなかった。

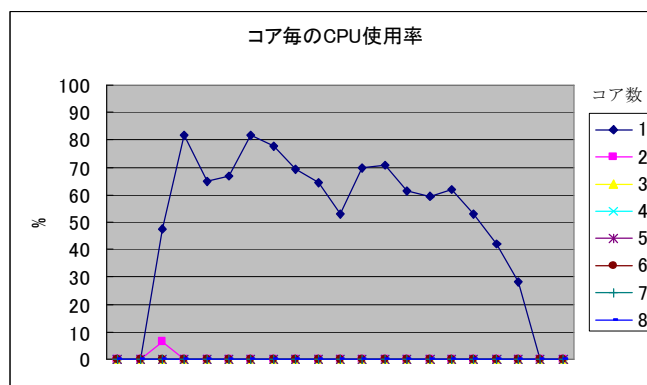


図 5 : Ruby ノーマル版 test-transaction における CPU リソースの使用状況

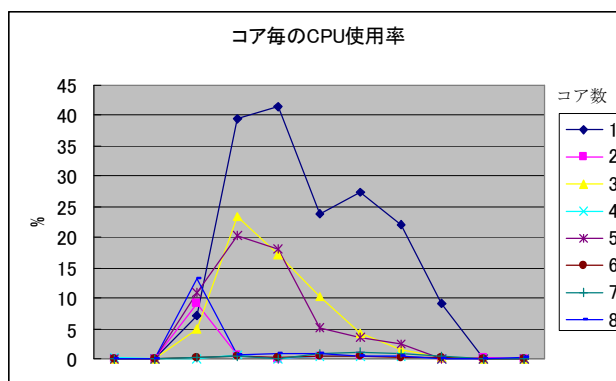


図 6 : Java test-transaction における CPU リソースの使用状況

リソース使用状況についても、Ruby 版と Java を比較したが、結果は test-ATIS と同様の傾向がみられた（図 5、図 6）。

ウ. データ量を増やした場合の test-ATIS および test-transaction

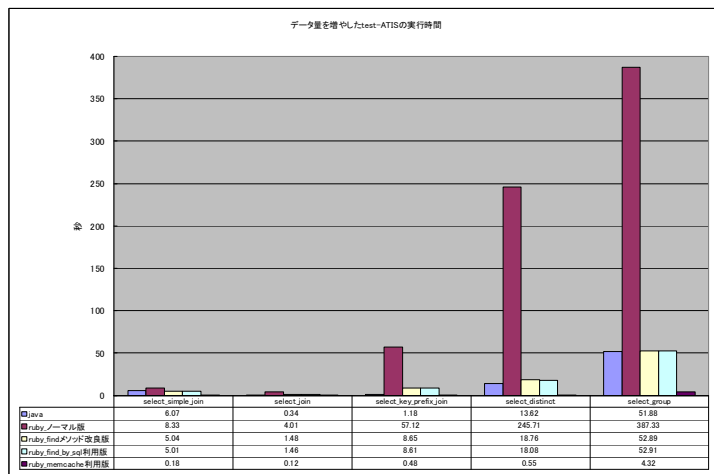


図 7 : データ量を増やした場合の SQL 種別実行時間 (test-ATIS)

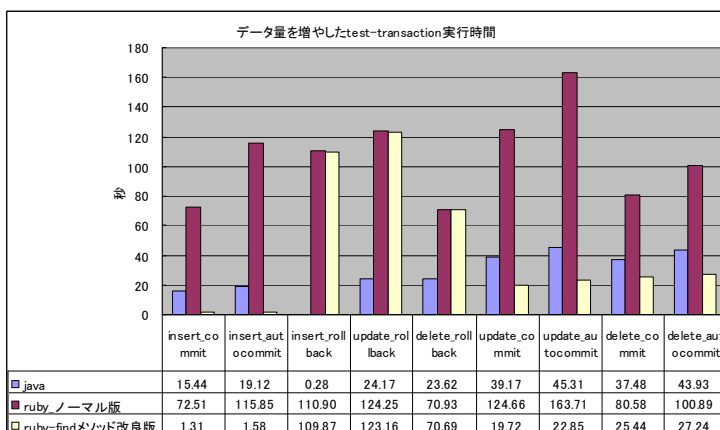


図 8 : データ量を増やした場合の SQL 種別実行時間 (test-transaction)

データ量を増やした場合の測定では、図 7 のとおり、test-ATIS ではデータ取得処理に関しては各 SQL 種別においてデータを増やす前とほぼ同様の傾向を示している（スキーマの成約によりデータ量の増加に違いがあるため、SQL 種別での比較をすると傾向は異なる）。しかし、select group においては、find メソッド改良版「ActiveRecord の find オプションを変更し、効率の良い SQL を出力するよう改善したプログラム」、find\_by\_sql 利用版「ActiveRecord#find\_by\_sql メソッドを利用し、オリジナルの test-ATIS と同じ SQL を出力するよう改善したプログラム」と Java 版プログラムでは実行時間の違いがほとんど見られなかった。これは、select group に複雑な SQL が多く、データを増やしたことで SQL 自体の処理時間の割合が増加し、O/R マッピングライブラリによる影響が相対的に小さくなったのではないかとと思われる。

また、図 8 のとおり、test-transaction では、データを増やす前と後ではグラフの形状に大きな変化は見られない。特に、データを増やす前の Ruby 版のプログラム（改善前と改善後）の実行時間の差（割合）と、データを増やしたときと同様

の実行時間の差（割合）にほとんど変化がみられない。このことから、改善前のプログラムであっても、データが増えたときに爆発的に処理時間が増加することは無いと考えられる。

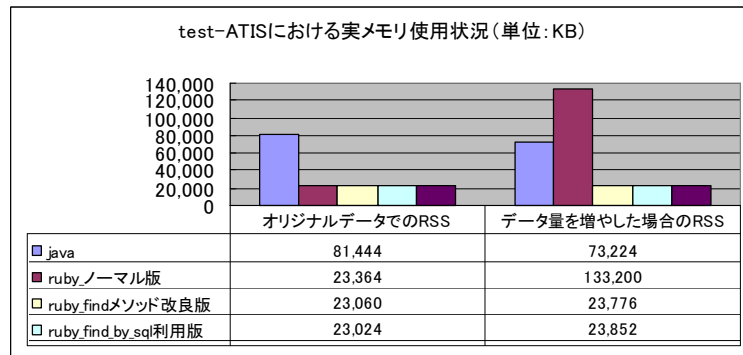


図 9 : test-ATIS における実メモリ使用状況

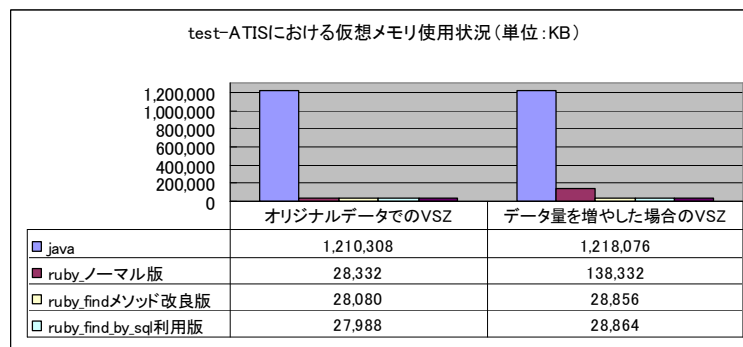


図 10 : test-ATIS における仮想メモリ使用状況

メモリの使用状況に関して、Java 版と Ruby 版は、処理系の仕様だけでなく、O/R マッピングフレームワークのアーキテクチャなどメモリ使用方法に関して考慮すべき点が多いため、データを増やす前後の傾向を見るために比較した。図 9、図 10 のとおり、ノーマル版「ActiveRecord の一般的な利用方法で記載されたプログラム」を実行した場合のみ、データ量を増やす前後で顕著な違いが現れ、データを増やす前より実メモリの使用量で 6 倍近く、仮想メモリを確保している量で 5 倍近く増加した。これは、ノーマル版「ActiveRecord の一般的な利用方法で記載されたプログラム」では SQL レベルで `distinct` 及び `group` を使用しておらず、最初にデータを取得してから Ruby レベルでまとめるという処理が行われるためである。データを増加させた場合、最初にデータを取得するところで大量のメモリが必要になるため、メモリ使用量が増加したと思われる。

### 3. スケーラビリティ検証

本技術検証で実施した「スケーラビリティ検証」について以下に詳細を記載する。

#### 3.1 スケーラビリティ検証概要

##### (1) 検証概要

本スケーラビリティ検証は、負荷ツール等を用い実際の運用環境を想定した多重度対応の側面から、Ruby で作成されたアプリケーションの「アプリケーションとしての処理プロセス」の性能を評価する。検証で利用するアプリケーションは HTTP 経由にて実行するベンチマークアプリケーションである、PetStore を利用した。PetStore は Java で実装されている JPetStore があるが、本検証では Ruby on Rails で実装された Rails PetStore(下記 URL 参照)を Ruby on Rails バージョン 2.2.2 に対応させたものを用いて測定を行った。

Rails PetStore : <http://tw-commons.rubyforge.org/svn/petstore/trunk/>

本検証では、パフォーマンス検証と同様に一般的な記述方法で実装したアプリケーションと、一般的な記述方法で実装した場合に想定されるボトルネックについて仮説を立て、改良を施したアプリケーションを使用して検証を行う。スケーラビリティ検証は HTTP 経由での検証を行うが、そのベンチマークを取得する方法としてパフォーマンス測定・負荷テストツールである、JMeter を利用して検証を行う。検証の内容を以下に示す。

##### ア. Web サーバソフトウェアの性能検証と選定

Web サーバソフトウェアは、Ruby on Rails 動作環境として広く利用されている Mongrel (Apache mod\_proxy\_balancer との併用)と、Ruby on Rails のデプロイメントが容易かつ高速な Passenger を利用し、スループットや検証中のリソースの状態を記録しながら最適な Web サーバソフトウェアを選定する。

Mongrel : <http://mongrel.rubyforge.org/>

Passenger : <http://www.modrails.com/>

##### イ. スケーラビリティの向上を検討・改善したアプリケーションでの性能検証

この検証では、想定されるボトルネックを仮説として立て、改善を施した Rails PetStore も用意して検証を行う。また、Web サーバソフトウェアは、アの検証結果を元に性能結果の良い Web サーバソフトウェアを利用して検証を行う。

##### (2) 検証プログラム概要

検証で利用する Rails PetStore は Ruby on Rails バージョン 2.2.2 に対応して検証を行うが、改変の概要と各プログラムの修正箇所を以下に示す。なお、変更箇所につい

ては差分表示（-は削除、+は追加）で記載する。

- カートに商品を格納し、商品を注文できる機能の追加
- ユーザ変更機能の追加

#### ア. app/controllers/accounts\_controller.rb

カートの有無でリダイレクト先を分け、かつユーザ変更機能を追加した。

```
@@ -7,7 +7,12 @@
  @account = Account.new(params[:account])
  if @account.save
    session[:account_id] = @account.id
-   redirect_to :controller => 'shop', :action => 'index'
+   #redirect_to :controller => 'shop', :action => 'index'
+   if current_cart.empty?
+     redirect_to :controller => 'shop', :action => 'index'
+   else
+     redirect_to :controller => 'orders', :action => 'new'
+   end
@@ -16,4 +21,17 @@
  def edit
    @account = current_account
  end
+
+  def update
+    begin
+      @account = current_account
+      @account.password_confirmation = params[:password][:confirmation]
+      @account.update_attributes!(params[:account])
+      flash[:message] = 'Account was successfully updated.'
+    rescue
+      flash[:message] = 'Account update Error.'
+    end
+    redirect_to :controller => 'shop', :action => 'index'
+  end
```

#### イ. app/controllers/authentication\_controller.rb

カートの有無でリダイレクト先を分けた。

```
@@ -1,12 +1,17 @@
-
-  class AuthenticationController < ApplicationController
+
+    def login
+    end

+    def login_post
+      if account = Account.find_by_username_and_password(params[:username],
params[:password])
+        session[:account_id] = account.id
-        render :template => "shop/index"
+        #render :template => "shop/index"
+        if current_cart.empty?
```



オ. app/models/cart\_item.rb

Forwardable#def\_delegators に:~id を追加した。

```
@@ -1,6 +1,9 @@
+# Addition for class 'CartItem'
+require 'forwardable'
+
+ class CartItem
+   extend Forwardable
+
+ def_delegators :@item, :attribute1, :attribute2, :attribute3, :attribute4, :attribute5, :in_stock?, :list_price, :product, :string_id
+
+ def_delegators :@item, :attribute1, :attribute2, :attribute3, :attribute4, :attribute5, :in_stock?, :list_price, :product, :string_id, :id
+   attr_accessor :quantity
```

カ. app/models/item.rb

LineItem モデルクラスとのアソシエーションを追加した。

```
@@ -1,5 +1,6 @@
+ class Item < ActiveRecord::Base
+   belongs_to :product
+   has_one :line_item
```

キ. app/models/line\_item.rb

LineItem モデルクラスを追加した。

```
@@ -0,0 +1,4 @@
+class LineItem < ActiveRecord::Base
+  belongs_to :order
+  belongs_to :item
+end
```

ク. app/models/order.rb

Order モデルクラスに LineItem モデルクラスへのアソシエーションを追加した。

また、カートの商品を同クラスに追加するためのメソッド(add\_cart\_item、add\_line\_item)を追加した。

```
@@ -1,8 +1,10 @@
+ class Order < ActiveRecord::Base
+
+   has_many :line_items, :foreign_key => "order_id"
+
+   def self.new_with_defaults(attributes = {})
+     returning(new(attributes)) do |order|
+       order.credit_card = "9999 9999 9999 9999"
+     end
+   end
+   order.credit_card = "9999 9999 9999 9998"
+   order.expiration_date = "12/03"
+ end
+
+@@ -17,4 +19,21 @@
+ self.billing_zip = account.zip
+ self.billing_country = account.country
```

```

end
+
+ def add_cart_items(cart_items)
+   self.total_price = 0
+   cart_items.each do |cart_item|
+     self.add_line_item(cart_item)
+     self.total_price = self.total_price + cart_item.total
+   end
+ end
+
+ def add_line_item(cart_item)
+   line_item = LineItem.new(:line_number => self.line_items.size + 1)
+   line_item.quantity = cart_item.quantity
+   line_item.unit_price = cart_item.list_price
+   line_item.item_id = cart_item.id
+   self.line_items << line_item
+ end

```

#### ケ. app/views/accounts/edit.rhtml

ユーザ変更画面のパスワード入力フィールドを変更した。

```

@@ -1,4 +1,4 @@
-<form method="post" action="/shop/editAccount.shtml">
+<form method="post" action="/shop/updateAccount.shtml">
  <html:hidden name="accountBean" property="validation" value="edit" />
  <html:hidden name="accountBean" property="username" />
  <table cellpadding="10" cellspacing="0" align="center" border="1" bgcolor="#dddddd">
@@ -16,7 +16,10 @@
    </tr>
    <tr bgcolor="#FFFFFF88">
      <td>Repeat password:</td>
+
+    <!--
      <td><html:password name="accountBean" property="repeatedPassword"
/></td>
+
+    -->
+    <td><%= password_field('password', 'confirmation') %></td>

```

#### コ. app/views/orders/confirm.rhtml

商品注文確認画面で商品が表示されるように変更した。

```

@@ -7,14 +7,16 @@
  </table>

  <p>
+ <!--
  <center>
    <b>Please confirm the information below and then press continue...</b>
  </center>
+ -->
  <p>
    <table width="60%" align="center" border="0" cellpadding="3" cellspacing="1"
    bgcolor="#FFFFFF88">
      <tr bgcolor="#FFFFFF88">
        <td align="center" colspan="2">
-          <font size="4"><b>Order</b></font>
+          <font size="4"><b>Order</b><%= "&nbsp;&nbsp;&nbsp;&##{@order.id}" unless
@order.id.nil? %></font>

```

```

        <br />
        <font          size="3"><b><bean:write          name="orderBean"
property="order.orderDate" format="yyyy/MM/dd hh
:mm:ss" /></b></font>
    </td>
@@ -75,12 +77,47 @@
    <tr bgcolor="#FFFF88">
        <td>Country: </td><td><%= @order.shipping_country %></td>
    </tr>
+   <!-- Status Section Start -->
+   <tr bgcolor="#FFFF88">
+       <td colspan="2">
+           <b><font color="GREEN" size="4">Status:</font><%= @order.status %></b>
+       </td>
+   </tr>
+   <tr bgcolor="#FFFF88">
+       <td colspan="2">
+           <table align="center" bgcolor="#008800" border="0" cellspacing="2"
cellpadding="5" width="100%"
>
+           <tr bgcolor="#cccccc">
+               <td><b>Item ID</b></td>
+               <td><b>Description</b></td>
+               <td><b>Quantity</b></td>
+               <td><b>Price</b></td>
+               <td><b>Total Cost</b></td>
+           </tr>
+           <% @order.line_items.each do |line_item| %>
+               <% item = line_item.item %>
+               <tr bgcolor="#FFFF88">
+                   <td><b><%= link_to(item.string_id ,
"/shop/viewItem.shtml?item=#{item.string_id}") %></b>
</td>
+                   <td><%= item.attribute1 %> <%= item.attribute2 %> <%=
item.attribute3 %> <%= item.attribu
te4 %> <%= item.attribute5 %> <%= item.product %></td>
+                   <td align="center"><%= line_item.quantity %></td>
+                   <td align="right"><%= "$%5.2f" % line_item.unit_price %></td>
+                   <td align="right"><%= "$%5.2f" % (line_item.unit_price *
line_item.quantity) %></td>
+               </tr>
+           <% end %>
+           <tr bgcolor="#FFFF88">
+               <td colspan="7" align="right"><b>Sub Total: <%= "$%5.2f" %
@order.total_price %></b><br/></
td>
+           </tr>
+       </table></td>
+   </tr>
+   <!-- Status Section End -->
</table>
<p>
    <center>
+   <!--
    <a href="/shop/newOrder.shtml?confirmed=true">
        
    </a>
+   -->

```

```
+      <%= link_to(image_tag("../images/button_continue.gif"), submit_order_url) if
@order.id.nil? %>
      </center>
```

#### サ. config/environment.rb

vendor 以下のツール群を削除したことにより、will\_paginate プラグインを require で呼び出すように変更した。

Ruby on Rails のバージョンを 2.2.2 に変更した。

```
@@ -1,4 +1,5 @@
-RAILS_GEM_VERSION = '1.2.3' unless defined? RAILS_GEM_VERSION
+#RAILS_GEM_VERSION = '1.2.3' unless defined? RAILS_GEM_VERSION
+RAILS_GEM_VERSION = '2.2.2' unless defined? RAILS_GEM_VERSION
require File.join(File.dirname(__FILE__), 'boot')

if RAILS_ENV =~ /jdbc/
@@ -22,8 +23,10 @@
  if RUBY_PLATFORM =~ /java/
    config.plugins = %w(jrubyworks will_paginate)
  else
-   config.plugins = %w(will_paginate)
+#   config.plugins = %w(will_paginate)
  end
end

Object.send :undef_method, :id
+
+require 'will_paginate'
```

#### シ. config/routes.rb

ユーザ変更及び商品注文処理へのアクセスルールを追加した。

```
@@ -14,10 +14,12 @@
  map.cart_summary "shop/checkout.shtml", :controller =>
"cart", :action => "summary"
  map.new_order "shop/newOrderForm.shtml", :controller =>
"orders", :action => "new"
  map.edit_account "shop/editAccountForm.shtml", :controller =>
"accounts", :action => "edit"
+  map.update_account "shop/updateAccount.shtml", :controller =>
"accounts", :action => "update"
  map.update_cart "shop/updateCartQuantities.shtml", :controller =>
"cart", :action => "update"
  map.remove_from_cart "shop/removeItemFromCart.shtml", :controller =>
"cart", :action => "remove"
  map.confirm_order "shop/newOrder.shtml", :controller =>
"orders", :action => "confirm"
  map.orders "shop/listOrders.shtml", :controller =>
"orders", :action => "list"
```

```

st"
  map.order          "shop/viewOrder.shtml",          :controller =>
"orders",          :action => "sh
ow"
  map.search_products "shop/searchProducts.shtml",    :controller =>
"products",        :action => "se
arch"
+  map.submit_order   "shop/submitOrder.shtml",        :controller =>
"orders",          :action => "su
bmit"
end

```

ス. db/migrate/002\_add\_column\_to\_line\_item.rb

line\_items テーブルに item\_id カラムを追加した。

```

@@ -0,0 +1,9 @@
+class AddColumnToLineItem < ActiveRecord::Migration
+  def self.up
+    add_column :line_items, :item_id, :integer
+  end
+
+  def self.down
+    remove_column :line_items, :item_id
+  end
+end

```

セ. vendor/以下

Ruby on Rails のバージョンを 2.2.2 に変更することにより動作に影響がある、vendor/gems、vendor/plugins、vendor/rails 以下に配置されていたプログラムをそれぞれ削除した。

本検証で利用するデータベーススキーマとデータは以下の通り。

- accounts
- categories
- items
- line\_items
- order\_status
- orders
- products
- suppliers

accounts テーブル

カラム	型	キー	その他
Id	int(11)	プライマリーキー	auto_increment
Username	varchar(255)		

Password	varchar(255)		
Email	varchar(255)		
first_name	varchar(255)		
last_name	varchar(255)		
Status	varchar(255)		
address_line_1	varchar(255)		
address_line_2	varchar(255)		
City	varchar(255)		
State	varchar(255)		
Zip	varchar(255)		
Country	varchar(255)		
Phone	varchar(255)		
language_preference	varchar(255)		
favorite_category_id	int(11)		
my_list_option	tinyint(1)		
banner_option	tinyint(1)		

categories テーブル

カラム	型	キー	その他
Id	int(11)	プライマリーキー	auto_increment
string_id	varchar(255)		
Name	varchar(255)		
Descripting	varchar(255)		
Banner	varchar(255)		

items テーブル

カラム	型	キー	その他
Id	int(11)	プライマリーキー	auto_increment
product_id	int(11)		
string_id	varchar(255)		
list_price	decimal(10,2)		
unit_cost	decimal(10,2)		
Supplier	int(11)		
Status	varchar(255)		
attribute1	varchar(255)		
attribute2	varchar(255)		
attribute3	varchar(255)		
attribute4	varchar(255)		
attribute5	varchar(255)		

quantity	int(11)		
----------	---------	--	--

line\_items テーブル

カラム	型	キー	その他
id	int(11)	プライマリーキー	auto_increment
order_id	int(11)		
line_number	int(11)		
quantity	int(11)		
unit_price	decimal(10,2)		
item_id	int(11)		

order\_status テーブル

カラム	型	キー	その他
id	int(11)	プライマリーキー	auto_increment
order_id	int(11)		
line_number	int(11)		
created_at	datetime		
status	varchar(255)		

orders テーブル

カラム	型	キー	その他
id	int(11)	プライマリーキー	auto_increment
account_id	int(11)		
created_at	datetime		
shipping_address_line_1	varchar(255)		
shipping_address_line_2	varchar(255)		
shipping_city	varchar(255)		
shipping_state	varchar(255)		
shipping_zip	varchar(255)		
shipping_country	varchar(255)		
billing_address_line_1	varchar(255)		
billing_address_line_2	varchar(255)		
billing_city	varchar(255)		
billing_state	varchar(255)		
billing_zip	varchar(255)		
billing_country	varchar(255)		
courier	varchar(255)		
total_price	decimal(10,2)		
bill_to_first_name	varchar(255)		

bill_to_last_name	varchar(255)		
ship_to_first_name	varchar(255)		
ship_to_last_name	varchar(255)		
credit_card	varchar(255)		
expiration_date	varchar(255)		
card_type	varchar(255)		
locale	varchar(255)		

products テーブル

カラム	型	キー	その他
id	int(11)	プライマリーキー	auto_increment
string_id	varchar(255)		
category_id	int(11)		
name	varchar(255)		
description	varchar(255)		

suppliers テーブル

カラム	型	キー	その他
id	int(11)	プライマリーキー	auto_increment
name	varchar(255)		
status	varchar(2)		
address_line_1	varchar(255)		
address_line_2	varchar(255)		
city	varchar(255)		
state	varchar(255)		
zip	varchar(255)		
phone	varchar(255)		

### (3) ボトルネックの仮説

実際に Web アプリケーションを運用した場合を想定し、Ruby の開発実績が豊富な「Enterprise Ruby コンソーシアム」の知見より、本検証で利用する Rails PetStore で検証した際に想定されるボトルネックを仮説として立て、改善を施したアプリケーションを用意した。具体的には、データ抽出時の DB アクセス及び HTML の表示部分(レンダリング)を基本的なボトルネックと想定し、以下の 3 つの改善版を用意した。

#### ア. アプリケーション改良版 Rails PetStore

Rails PetStore をアプリケーションレベルで改善したものである。この改善では、

Ruby on RailsでWebアプリケーションを実装した場合に一般的に実施されるチューニング方法を施し、性能が改善されるかを評価する。チューニングの内容としては、ファインダを利用した場合に、効率的なSQL文が発行されないこと、また部分テンプレートからHTMLを生成（レンダリング）する処理がオーバーヘッドになることがボトルネックになりうるため、findメソッドでファインダ利用している点を改良し、また部分テンプレートを廃止しレンダリング部分の改善を行う。

#### イ. memcached利用版Rails PetStore

この改善では、使用頻度の高いデータベースアクセス処理に対して、取得したデータをmemcachedに格納し処理スピードを向上させる。クライアントからアクセスされた機能がデータベースアクセスをしている場合、アクセス数の増加に伴い、アプリケーションとデータベース間の負荷が高くなることがボトルネックとなりえる。この問題を改善するために、使用頻度の高い機能に対して、一度データベースから取得したデータをmemcachedに格納し再利用することによって、データベースとの通信を少なくする実装を行った。

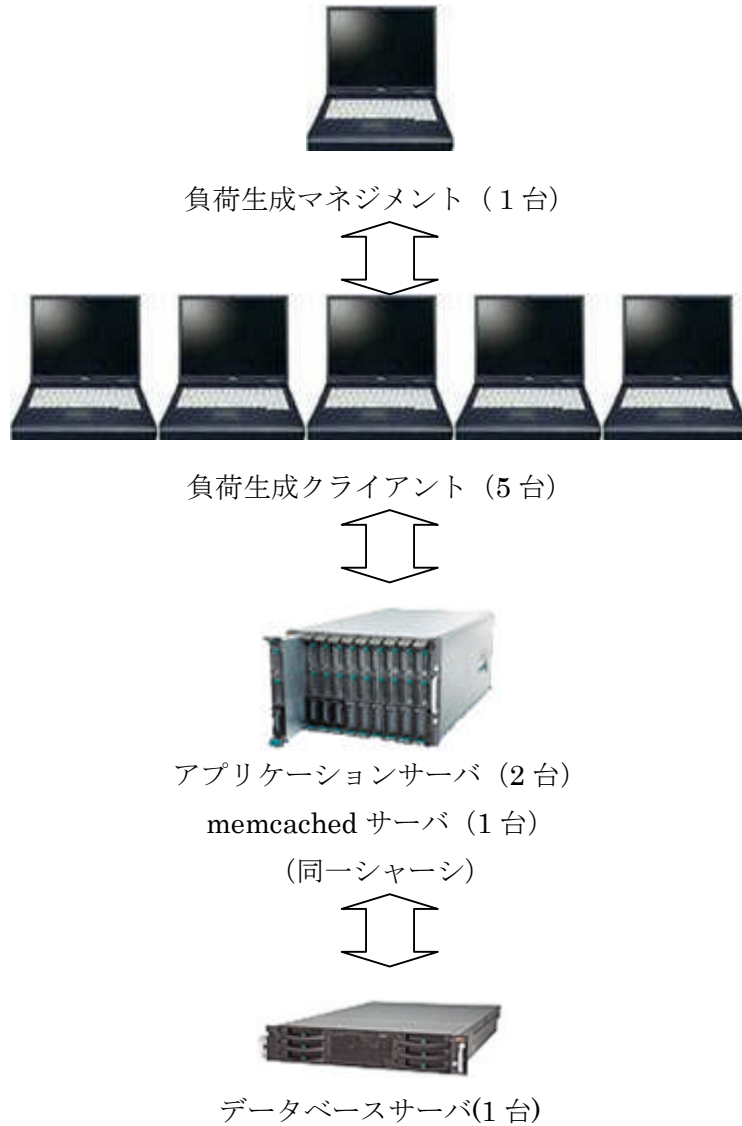
#### ウ. 全改良適用版Rails PetStore

この改善では、上記アとイの両方を適用した Rails PetStore で検証することとした。

### 3.2 スケーラビリティ検証実施環境

#### (1) ハードウェア

本検証で利用した環境は次の通り。



5 台の負荷クライアントから Rails PetStore を稼動させた 2 台のアプリケーションサーバへアクセスを行う。Rails PetStore からアクセスするデータベースサーバ、memcached サーバは 1 台とする。

#### 【負荷生成クライアント】

負荷生成クライアント	6 台 (内 1 台は、マネジメント用)
製品名	FUJITSU FMV830NA
CPU	Pentium 4 3.2GHz ×1

メモリ	2GB
DISK	80GB

【アプリケーションサーバ】

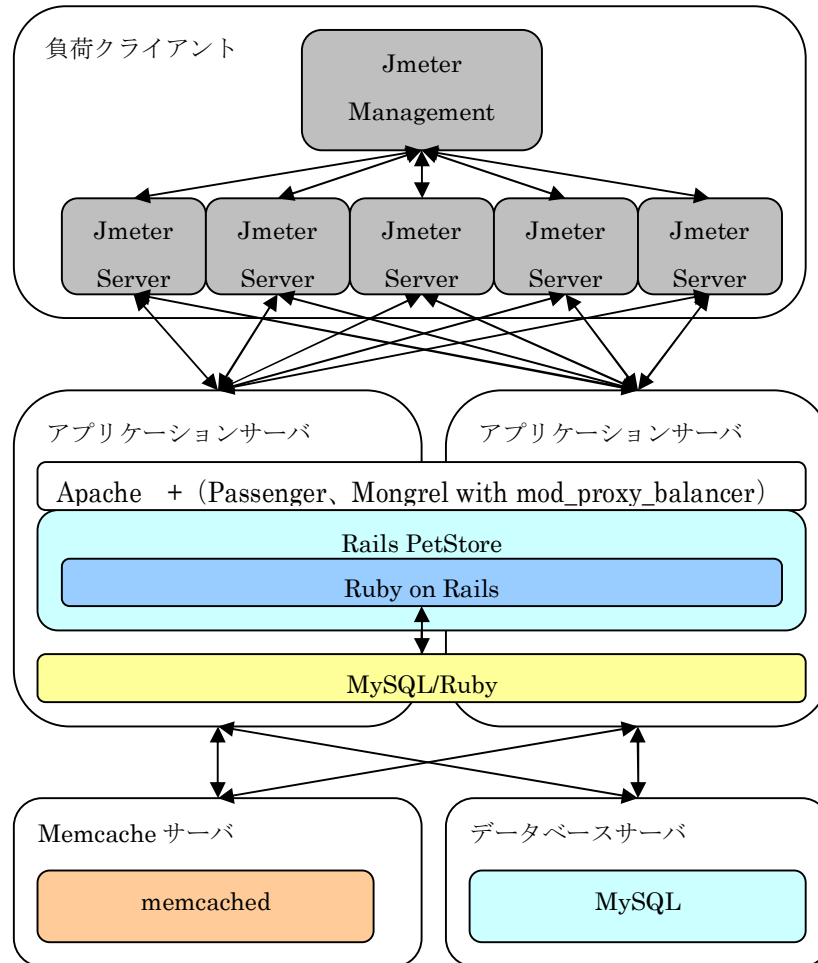
AP サーバ	3 台 (内、2 台を Web サーバとし 1 台を memcached サーバとする)
製品名 (シャーシ)	FUJITSU PRIMERGY BX600 S3
製品名	FUJITSU PRIMERGY BX620 S4
CPU	QuadXeon X5355 2.66GHz/2x4MB *2
メモリ	8GB
DISK	147GB(RAID1)

【データベースサーバ】

DB サーバ	1 台
製品名	FUJITSU PRIMERGY RX300 S4
CPU	QuadXeon5460 3.16GHz/12MB *2
メモリ	4GB
DISK	294GB(RAID5)

## (2) ソフトウェア

本検証で利用したソフトウェアの環境は次の通り。



1台の Jmeter Management から Server モードで起動させた 5 台の Jmeter リモートクライアントに対し処理を要求する。5 台の Jmeter リモートクライアントは、設定シナリオ通りの画面遷移処理を、http プロトコルを利用してアプリケーションサーバで起動している Web サーバに対してリクエスト送信する。

アプリケーションサーバで稼働している Rails PetStore アプリケーションからアクセスするデータベースサーバ、memcached サーバはそれぞれ専用で 1 台稼働させる。

また、計測毎にファイルキャッシュをクリアし、Apache, memcached は再起動を行う。

### 【アプリケーションサーバ】

OS	Red Hat Enterprise Linux 5.2 (for x86) 2.6.18-92.el5.PAE 32 ビット
----	---

Web サーバ ソフトウェア	Apache Version 2.2.3
	Mongrel Version 1.1.5
	Mongrel Cluster Version 1.0.5
	Passenger Version 2.0.6
Ruby	Ruby Version 1.8.7-p72
Ruby Framework	Ruby on Rails Version 2.2.2
Ruby ConnectorAPI	mysql-ruby Version 2.8

【memcached サーバ】

OS	Red Hat Enterprise Linux 5.2 (for x86) 2.6.18-92.el5.PAE 32 ビット
分散メモリ キャッシュサーバ	Memcached Version 1.2.6
	Libevent Version 1.4.9

【データベースサーバ】

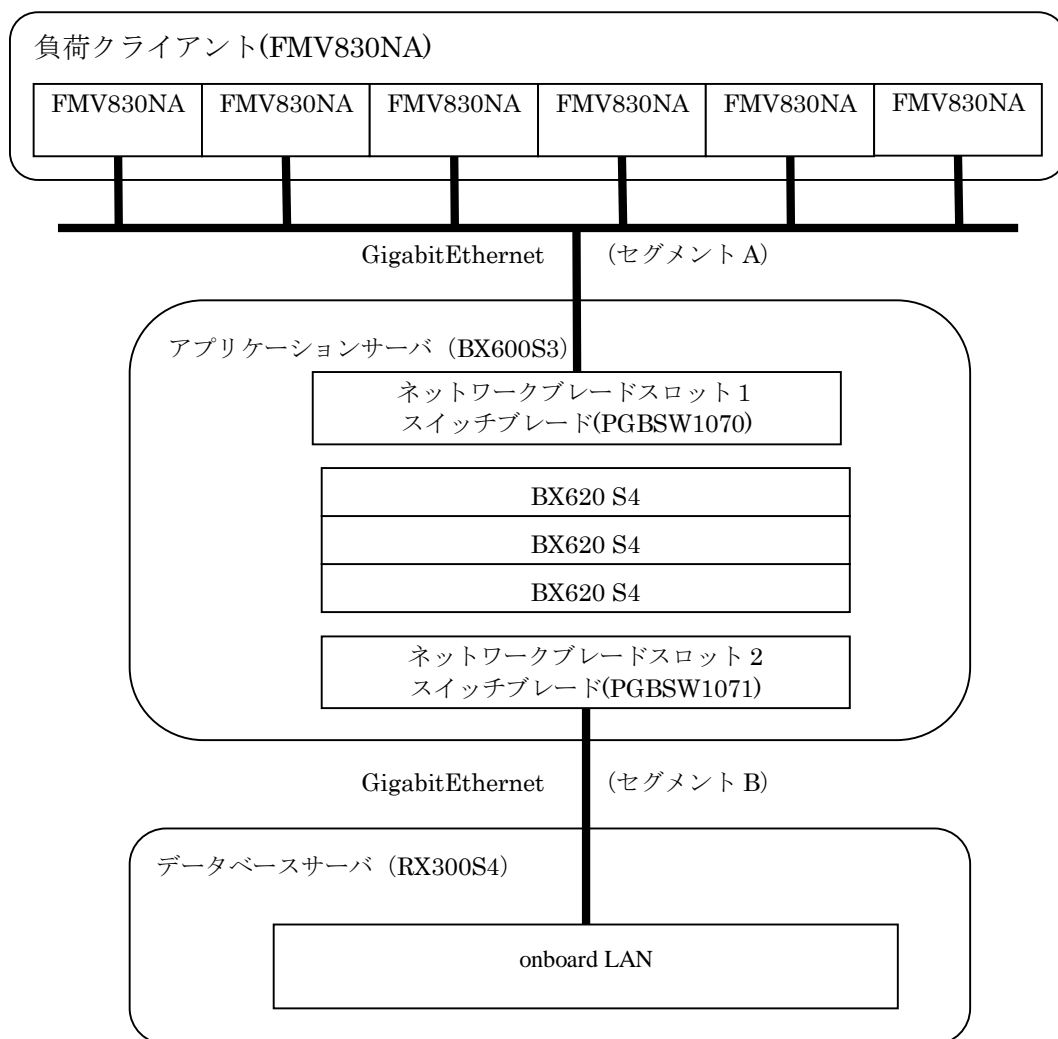
OS	Red Hat Enterprise Linux 5.2 (for Intel64) 2.6.18-92.el5 64 ビット
RDBMS	MySQL Version 5.1.30

【負荷生成クライアント】

OS	Windows XP Professional SP2 32 ビット
負荷生成 ソフトウェア	jakarta-jmeter Version 2.3.2 Java Jre Version 6u11

### (3) ネットワーク

本検証で利用したネットワーク構成は次の通り。



アプリケーションサーバとして利用する BX620 S4 は、BX600S3 シャーシに内蔵されている 2つのネットワーク・スイッチに内部的に接続されており、サーバブレードと外部 LAN との間を接続している。

負荷クライアントとして利用する FMV830NA からアプリケーションサーバへの接続はアプリケーションサーバの内蔵スイッチ 1 を使用し、ギガビット・イーサケーブルで接続しており、セグメント A で設定している。

アプリケーションサーバとデータベースサーバの間は内蔵スイッチ 2 を使用し、ギガビット・イーサケーブルで接続しており、セグメント B で設定している。

負荷クライアント、アプリケーションサーバ、データベースサーバ間にネットワーク機器を挟んでいない為、ボトルネックが発生しにくい構成としている。負荷クライアントとアプリケーションサーバ間、アプリケーションサーバとデータベースサーバ間は別セグメントを利用している。負荷クライアントとアプリケーションサーバ間、アプリケーションサーバとデータベースサーバ間の接続には IP アドレスを使用し、データベースサーバの MySQL 設定では DNS を無効にしており、

ホスト名逆引きなどの余計なコストをかけない設定にしている。また、全ての試験に関してネットワークにエラーがないことをifconfigコマンドで確認している。

### 3.3 スケーラビリティ検証実施内容詳細

#### (1) Web サーバソフトウェアの性能検証と選定

Ruby on Rails を稼働させる最適な Web サーバを性能検証の結果から選定する。

Web サーバは Apache mod\_proxy\_balancer+Mongrel と Apache+Passenger を利用し、Web サーバに対しては負荷生成クライアントで起動している Jmeter からリクエストを行う。

検証時に利用するソフトウェアと設定は次の通り。

#### ア. 共通

##### a. MySQL

設定ファイルは、インストール時に付属するサンプル設定ファイルの my-innodb-heavy-4G.cnf を使用する。サンプル設定ファイルから変更した点は以下の通り。

```
以下のパラメータを[mysql]セクションへ追加
character-set-server=utf8
skip-character-set-client-handshake
skip-name-resolve
以下のパラメータをコメントアウトし無効とする
#log-bin=mysql-bin
#binlog_format=mixed
以下のパラメータを変更する
default_table_type = InnoDB
innodb_flush_log_at_trx_commit = 2
```

##### b. Apache

設定ファイルは、インストール時に付属するデフォルトの httpd.conf を利用する。

##### c. memcached

以下のパラメータ設定を使用して起動する。

複数起動する場合は、ポート番号を変更し起動する。

```
/usr/local/bin/memcached -m 300 -p 11211 -u root -c 2000 -v -d
```

#### イ. Mongrel 検証

本検証では、プロセス数を増やしながら性能検証を行う為、Mongrel に Apache の mod\_proxy\_balancer を利用した構成で実施する。

##### a. Mongrel

mongrel\_cluster の設定

アプリケーションディレクトリ配下の config/mongrel\_cluster.yml を次のよう

に記述する。<起動プロセス数>の箇所に、起動するプロセス数を設定する。

```
user: apache
cwd: /var/www/petstore
log_file: log/mongrel.log
port: "3000"
environment: production
group: apache
pid_file: tmp/pids/mongrel.pid
servers: <起動プロセス数>
```

#### b. Apache 設定

Mongrel で複数のプロセスを利用するため、次のコマンドを実行し、モジュールを有効にする。なお、単一のプロセスで利用する場合も同様に `mod_proxy_balancer` を利用する。

```
$ a2enmod proxy
$ a2enmod proxy_balancer
```

デフォルトの Apache 設定ファイルに、次の設定を追加する。

`/etc/httpd/conf/httpd.conf`

```
<VirtualHost *>
  ServerName localhost.localdomain
  DocumentRoot "/var/www/petstore/public"
  <Proxy *>
    Order deny,allow
    allow from all
  </Proxy>

  ProxyPass / balancer://localhost/
  ProxyPassReverse / balancer://localhost/

  <proxy balancer://localhost/>
    BalancerMember http://localhost:3000
    BalancerMember http://localhost:3001
    .
    (起動するプロセス数に応じて、ポート番号のみを変更し記述する)
    .
  </proxy>
</VirtualHost>
```

#### ウ. Passenger (mod\_rails) 検証

`Passenger-install-apache2-module` コマンドで出力される内容を設定する。

##### a. Apache 設定

デフォルトの Apache 設定ファイルに、次の設定を追加する。<起動プロセス数>の箇所に起動するプロセス数を設定する。

/etc/httpd/conf/httpd.conf

```
LoadModule Passenger_module
/usr/local/lib/ruby/gems/1.8/gems/Passenger-2.0.6/ext/apache2/mod_Passenger.so
PassengerRoot /usr/local/lib/ruby/gems/1.8/gems/Passenger-2.0.6
PassengerRuby /usr/local/bin/ruby

PassengerMaxPoolSize <起動プロセス数>
PassengerUseGlobalQueue off
<VirtualHost *>
    ServerName localhost.localdomain
    DocumentRoot /var/www/petstore/public
</VirtualHost>
```

## エ. ApacheMPM 検証

Apache の processing model (MPM)方式を Prefork と Worker 共に検証する為、以下の設定を検証項目に応じて変更する。

### a.Prefork を利用した検証の場合 (Apache のデフォルト設定)

/etc/sysconfig/httpd

```
以下の設定をコメントアウトする
#HTTPD=/usr/sbin/httpd.worker
```

### b.Worker を利用した検証の場合

/etc/sysconfig/httpd

```
以下の設定を有効にする
HTTPD=/usr/sbin/httpd.worker
```

以下のファイルを削除もしくは、適当な場所へ移動する。

/etc/httpd/conf.d/php.conf

(デフォルトインストール状態で配置されているファイルだが php が Worker に対応しておらず、Apache 起動時にファイルが存在するとエラーとなる為)

## オ. 検証プログラム

ノーマル版 Rails PetStore (3-1. (3) 記載の改善を施していない RailsPetStore)

## カ. 検証データ・スキーマ

検証データ・スキーマは Rails PetStore のデフォルトデータを利用する。

デフォルトデータ件数は次の通り。

テーブル名	初期データ件数
accounts	2
categories	5
items	28

line_items	0
order_status	0
orders	0
products	16
schema_migrations	2
sessions	0
suppliers	2

#### キ. Jmeter 検証シナリオ

次の画面遷移を Jmeter のシナリオ内で行う

トランザクション名	処理内容
TopPage	トップページの表示
ProductSearch	商品の検索
Product	プロダクトの選択
AddCart	商品をカートへ追加
UpdateCart	購入数を変更し、変更を保存
Proceed to Checkout	チェックアウトに進む
Continue to Checkout	チェックアウトを続ける
AccountPage	RegisterNow をクリックし、 アカウント登録画面に進む
RegisterAccount	新規アカウントを登録
Submit	チェックアウトを決定する
Confirm	購入を確認する
SignOut	サインアウト

本シナリオを 1 回実行する際に発行されるクエリ数は次の通り

	ノーマル版	アプリケーション改良版	memcached 利用版	全改良適用版
Delete	1	0	0	0
Insert	5	3	3	3
update	32	0	0	0
Select	44	6	8	4
計	82	9	11	7
参照割合	53.66	66.67	72.73	57.14

#### ク. Jmeter 負荷設定

項目	設定値	内容
----	-----	----

スレッド数	25 スレッド	全ての負荷クライアントから 同時実行するスレッド総数
Ramp-up 期間	0 秒	指定したスレッド数を同時に起動する
持続時間	180 秒	実行時間

## ケ. 検証手順

### a. 共通手順

計測毎に利用する全テーブルを再作成し、初期データを投入する  
実行コマンド

```
mysql -uroot -e "DROP DATABASE IF EXISTS petstore_production"
```

```
sudo -u apache rake db:create RAILS_ENV=production
```

```
sudo -u apache rake db:migrate RAILS_ENV=production
```

```
sudo -u apache rake db:load RAILS_ENV=production
```

### b. Mongrel の最適プロセス数、最大性能検証

(ア) 起動プロセス数を変更し Mongrel\_cluster を開始

実行コマンド (アプリケーションディレクトリ配下で実行)

```
sudo -u apache mongrel_rails cluster::start
```

(イ) Apache を開始

実行コマンド

```
/etc/init.d/httpd start
```

(ウ) Jmeter から 1 台の Web サーバに対してリクエスト送信

### c. Passenger の最適プロセス数、最大性能検証

(ア) 起動プロセス数を変更し Apache を開始

実行コマンド

```
/etc/init.d/httpd start
```

(イ) Jmeter から 1 台の Web サーバに対してリクエスト送信

### d. Apache prefork/worker 性能・リソース使用状況検証

(ア) 起動 MPM を変更し Apache を開始

実行コマンド

```
/etc/init.d/httpd start
```

(イ) Jmeter から 1 台の Web サーバに対してリクエスト送信

## (2) スケーラビリティの向上を検討・改善したアプリケーションでの性能検証

この検証では「(1) Web サーバソフトウェアの性能検証と選定」の検証結果から想定されるボトルネックに対する改善策を施したプログラムを使用して検証を行う。

## ア. Web サーバソフトウェア

「(1) Web サーバソフトウェアの性能検証と選定」の検証結果から Passenger を 8 プロセスで起動して利用する

## イ. 検証プログラム

### a. ノーマル版 Rails PetStore

ノーマル版 Rails PetStore のプログラムについては「3.1 スケーラビリティ検証概要」を参照。3-1 (3) で記載した改善を施していない Rails PetStore。

### b. アプリケーション改良版 Rails PetStore

本プログラムは、ノーマル版 Rails PetStore において、データベース処理とレンダリング部分がボトルネックになりえるとの仮説を立て、Ruby on Rails に標準で備わっている機能を利用してボトルネックの改良を行ったものである。

データベース処理では以下のようなノーマル版 Rails PetStore のソースコードを変更した。

app/controllers/categories\_controller.rb

```
# @category = Category.find_by_string_id(params[:category].upcase)
@category = Category.find(:first, :select => 'id, name',
                          :conditions => ["string_id = ?", params[:category].upcase])
```

1行目がノーマル版 Rails PetStore のコードで、ActiveRecord の動的ファインダ機能を利用してデータ抽出を行っている。ActiveRecord の動的ファインダ機能は SQL 生成にあたり、たとえば `find_by_name` といったようなメソッド名を Ruby が解釈してから、メソッドの引数を合わせて SQL を生成する必要があり、単純な `find` メソッドがパラメータ解析のみで SQL 文を生成するのに比べて処理コストが高くなる。そのため、より処理コストが低い `find` を利用するように 2 行目以降のコードで改良を行った。また、取得するカラムに関しても必要最低限のカラムのみ取得するように変更し、Ruby が使用するメモリ量を抑えるようチューニングを行った。

レンダリング部分については、Ruby on Rails ではサブテンプレートを多用すると HTML 生成処理コストが高くなる傾向がある。これは、HTML の生成を Ruby on Rails が行うため、多段のテンプレートや繰り返しを利用したサブテンプレートなどを使用することで HTML 生成部分に Ruby の処理がかかり、パフォーマンスに影響を及ぼすためである。本改良版ではサブテンプレートとレイアウトファイルを廃止し、1 つのアクションメソッドで 1 つのビューファイルを利用するようチューニングを行った。これらの修正を今回の検証で利用する処理全てに対して可能な限り適用した。

### c. memcached 利用版 Rails PetStore

この Rails PetStore は、実際のアプリケーションの運用環境を想定し、データベースサーバへのアクセスを減らすことで高速化すると仮説を立て、チューニングを行った。データベースサーバへのアクセスを減らす方法として、memcached を利用して一度データベースから取得したデータをメモリに格納し、二度目以降のデータの抽出処理ではデータベースにアクセスせず、memcached にアクセスしデータの抽出を行うようにした。

#### app/controllers/products\_controller.rb

```
# @products = Product.find_by_keywords(keywords, params[:page])
unless @products = CACHE[params[:keyword].gsub(' ', '')]
  @products = Product.find_by_keywords(keywords, params[:page])
  CACHE[params[:keyword].gsub(' ', '')] = @products
end
```

1 行目ではノーマル版 Rails PetStore のソースコードで、データベースにアクセスしている。これを 2 行目以降のソースコードに変更した。2 行目では memcached にアクセスし、この処理に必要なデータが存在するかどうかチェックする。存在すると @products インスタンス変数にデータを格納し処理が終了する。次に 3 行目では memcached にデータが存在しない場合、データベースにアクセスしデータを取得する。4 行目では 3 行目のソースコードで取得したデータを memcached に格納する。これらの修正を今回検証で利用する処理全てに可能な限り適用した。

### d. 全改良適用版 Rails PetStore

この Rails PetStore はアプリケーション改良版 Rails PetStore と memcached 利用版 Rails PetStore で施した改良を同時に適用した Rails PetStore とする。

#### ウ. 検証データ・スキーマ

検証データ・スキーマは Rails PetStore の標準データを利用する。

「(1) Web サーバソフトウェアの性能検証と選定」と同様

#### エ. 検証シナリオ・手順

「(1) Web サーバソフトウェアの性能検証と選定」と同様

#### オ. Jmeter 負荷設定

項目	設定値	内容
スレッド数	5 スレッド	全ての負荷クライアントから
	10 スレッド	同時実行するスレッド総数
	15 スレッド	※負荷量を変化させ検証を行う

	20 スレッド 25 スレッド 50 スレッド	
Ramp-up 期間	0 秒	指定したスレッド数を同時に起動する
持続時間	300 秒	実行時間

カ. スケーラビリティ検証手順

a. 同時実行ユーザ数をスケールした検証

(ア) 利用する全テーブルを作成し、初期データを投入

実行コマンド

```
mysql -uroot -e "DROP DATABASE IF EXISTS petstore_production"
```

```
sudo -u apache rake db:create RAILS_ENV=production
```

```
sudo -u apache rake db:migrate RAILS_ENV=production
```

```
sudo -u apache rake db:dataload RAILS_ENV=production
```

(イ) Apache を開始

実行コマンド

```
/etc/init.d/httpd start
```

Jmeter から 1 台の Web サーバに対してリクエスト送信

b. スケールアウト検証

(ア) 利用する全テーブルを作成し、初期データを投入

実行コマンド

```
mysql -uroot -e "DROP DATABASE IF EXISTS petstore_production"
```

```
sudo -u apache rake db:create RAILS_ENV=production
```

```
sudo -u apache rake db:migrate RAILS_ENV=production
```

```
sudo -u apache rake db:dataload RAILS_ENV=production
```

(イ) Apache を開始

実行コマンド

```
/etc/init.d/httpd start
```

(ウ) Jmeter から 1 台と 2 台の Web サーバに対してリクエスト送信

JmeterManagement クライアントは CUI を利用する

### 3.4 スケーラビリティ検証実施結果

#### (1) Web サーバソフトウェアの性能検証と選定

##### ア. Mongrel と Passenger の起動プロセス数の最適値の検証結果

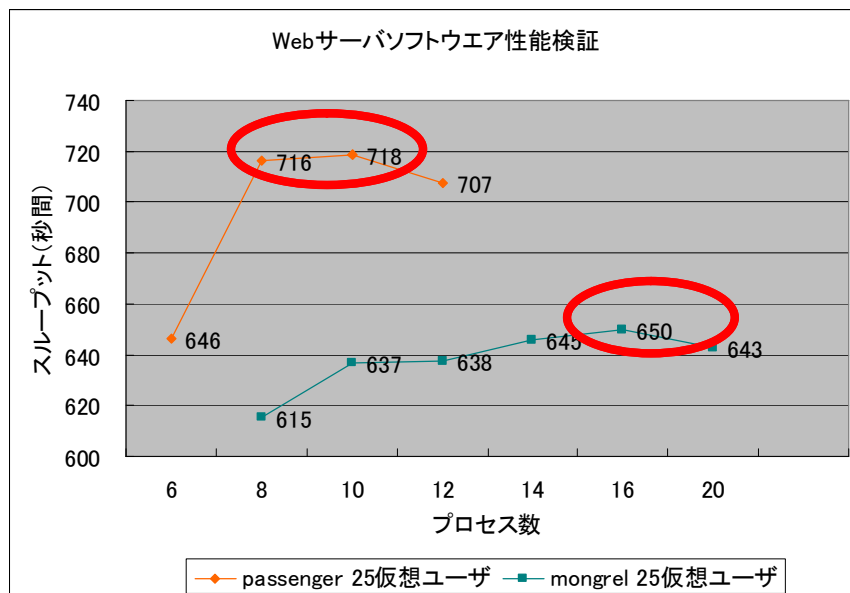


図 11 : Web サーバソフトウェアの性能比較

Mongrel と Passenger に関してはアーキテクチャの違いから同一プロセス数での比較ができない為、検証環境においてそれぞれの Web サーバソフトウェア毎で最も性能が出るプロセス数を検証し、最大性能時での比較を行った。

本検証環境においては、Passenger は 8~10 プロセスを起動している場合、Mongrel は 16 プロセス起動している場合が共に最もスループットが高くなるプロセス数であることが図 11 より確認できる。最大性能を比較すると約 70 スループット程 Passenger の性能が Mongrel より上回る結果となった。また、Mongrel は起動するプロセス数が固定であるのに対し、Passenger は設定により適宜プロセス数を増やすことが出来る。この違いにより、Mongrel はプロセス全てが処理中の場合にクライアントからのリクエストが送られた場合、リクエストは待たされることになるが、Passenger はプロセス起動数に余裕があれば新たにプロセスを増やし対応することができる。こういったアーキテクチャの違いについても運用環境では性能差に影響すると考えられる。

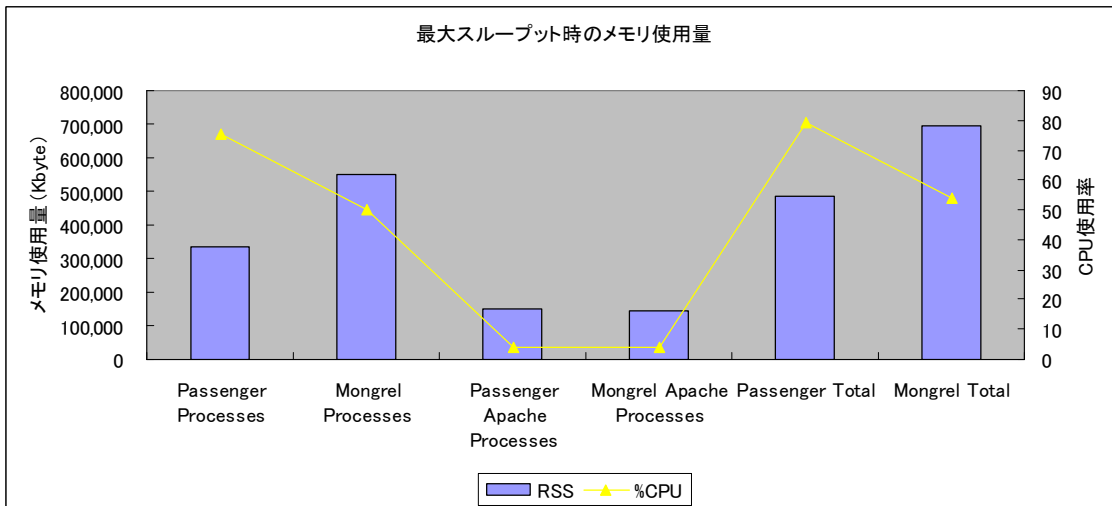


図 12：最大スループット時のメモリ使用量

次に Passenger、Mongrel 共に Apache のデフォルト MPM である Apache prefork MPM 利用時のリソースの使用状況を確認する。図 12 のグラフでは Passenger は 8 プロセス、Mongrel は 16 プロセス起動した場合の数値を抽出している。

各プロセス数の選定理由は、Passenger は 10 プロセス起動している場合、性能は 8 プロセスと同等であってもプロセス毎の CPU 使用率は負荷を上げて低いままであったため、本検証環境でノーマル版 Rails PetStore を動作するのに最適なプロセス数は 8 プロセスであると判断した。

また、Mongrel は、起動プロセス数を上げることにより CPU 使用率は上がるが、性能は横這いである為、最大性能値が出ていた 16 プロセスが最適であると判断した。

メモリ使用量に関しては、プロセスを多く起動している分 Mongrel の方が多く使用している結果となった。また、それぞれの Web サーバソフトウェアが使用する Apache の CPU 使用率に関しては差がみられないものの、Passenger のプロセスと Mongrel のプロセスが使用している CPU 使用率を比較すると 25% 程 Passenger が多く使用しており、Passenger がより効率的に CPU を使用しているという結果となった。

以上の検証結果から、スケーラビリティ測定に関しては Passenger を採用し検証を行うこととする。

#### イ. Apache マルチプロセッシングモジュール (MPM)のの違いによる性能とリソース消費についての検証結果

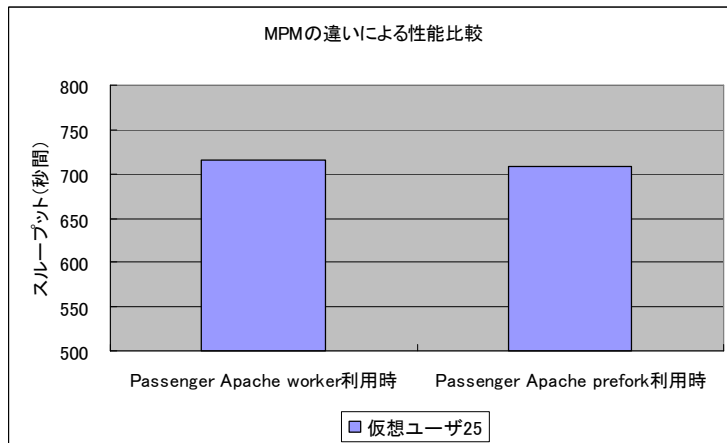


図 13 : Apache MPM の違いによる性能比較

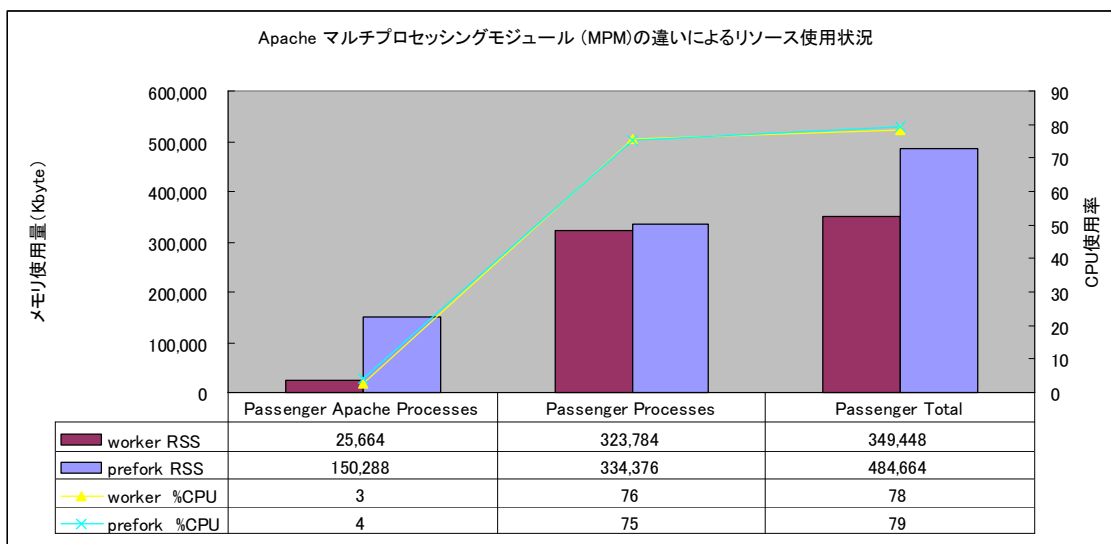


図 14 : Apache MPM の違いによる CPU リソースの使用状況

Passenger のバージョン 2.1 から Apache Worker MPM に対応したことを受けて、Apache Prefork MPM と Apache Worker MPM の性能とリソース使用状況について検証を行った。

図 13 より、Prefork と Worker の違いによるスループット（秒間）の性能差はないことが確認できる。

また、リソース使用状況については、図 14 より CPU 使用率に関しては Prefork、Worker の違いによる差がないことが確認できる。一方、メモリ使用量に関しては Prefork と比較して Worker で動作させることにより、Passenger 自体のプロセスが使用しているメモリ使用量に変化はみられないものの、Apache が使用するメモリ使用量が約 135MB 程低減していることが確認できる。

本検証結果から、スケーラビリティ測定に関しては Apache Worker MPM を採用し検証を行うこととする。

## (2) スケーラビリティの向上を検討・改善したアプリケーションでの性能検証

### ア. memcached の最適プロセス数の検証結果

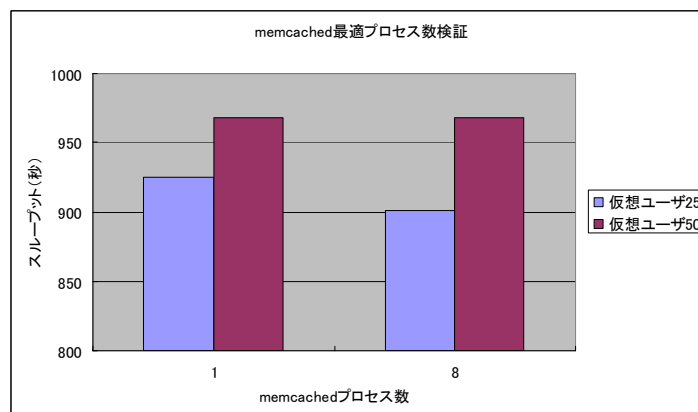


図 15 : memcached を利用した場合の、プロセス数の違いによるスループット比較

今回の検証環境で利用する memcached を利用した Rails PetStore アプリケーションで使用する memcached の起動プロセス数を決めるにあたって、1 プロセス、8 プロセスでのスループットを取得した。

1 プロセスと 8 プロセスを選択した理由は、Passenger のプロセス数が 8 プロセスで処理しているということと、現時点のバージョンの memcached はシングルスレッドで動作する仕様である為、今回の検証マシンの CPU コア数に合わせて 8 プロセスを検証に含めてスループットを比較検証した。

検証の結果、図 15 のとおり、仮想ユーザ 25 の場合は 1 プロセスが、若干性能がよく、仮想ユーザ 50 の場合は同等性能となった。検証実施時、memcached サーバの CPU 負荷は常に 1%以下であった。検証結果から、今回使用した検証アプリケーションでは memcached にかかる負荷は低く 1 プロセスで十分処理できると判断した。

本検証結果から、memcached プロセスは 1 プロセスのみ起動して検証を行うこととする。

## イ. Rails PetStore 検証結果

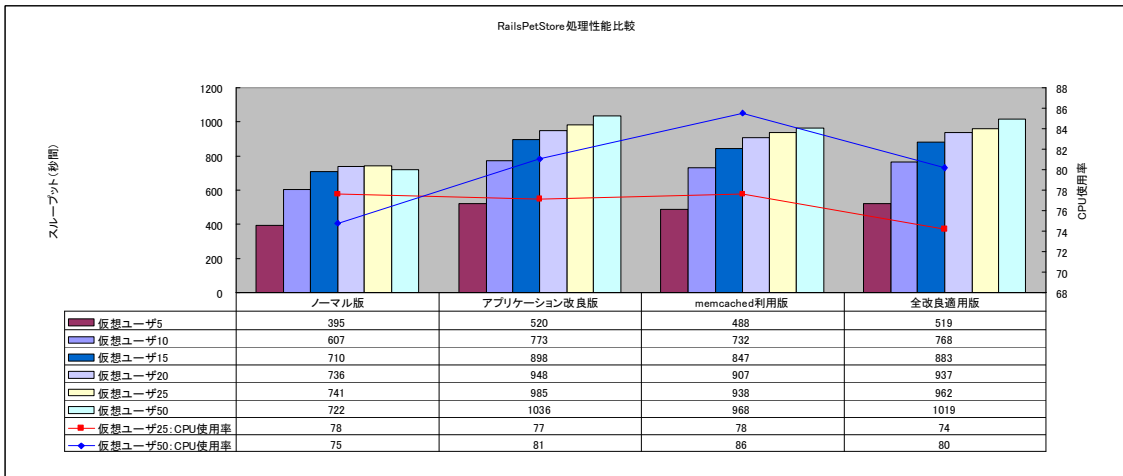


図 16 : Rails PetStore 性能比較

Rails PetStore を用いて、同時実行ユーザ数をスケールした検証の結果は図 16 のとおりである。

ノーマル版では仮想ユーザ数 20 から性能は頭打ちになり、仮想ユーザ数を増やしてもスループットが向上しない結果となった。

一方、改善を施した各 Rails PetStore では仮想ユーザ数の増加と共に性能も向上していることが確認できる。アプリケーション改良版ではレンダリング及び DB アクセスの高速化、memcached 利用版ではデータベースアクセス数の低減化により、スループットが向上していることが判り、アプリケーション改良版と memcached 利用版では同等の性能が出ていることが確認できる。両方の改良を適用した全改良適用版では、ノーマル版と比較すると性能は向上しているもののアプリケーション改良版、memcached 利用版と比較すると同等性能となっていることが確認できる。

この結果、データベース処理に関しては、一度目のアクセスにてアプリケーション改良版で改良したデータベース処理が行われるが、二度目以降のデータベース処理では memcached へのアクセスになるため、両方の改良を施した全改良適用版ではデータベース処理に関しては他の改良版と比べて大きく性能向上が見込めない。

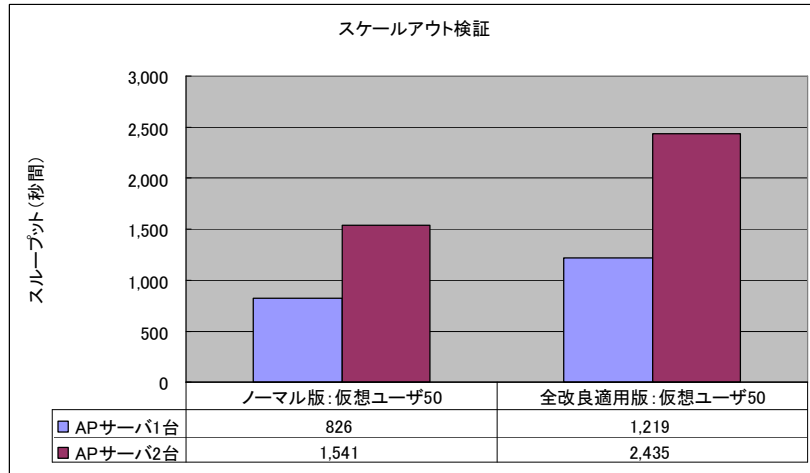


図 17: ノーマル版と全改良適用版のスケールアウト比較

スケールアウト検証として、負荷クライアントの負荷軽減の為に Jmeter は CUI モードで起動し、全改良適用版を 2 台まで Web サーバを増やして計測した。

図 17 の検証結果より、ノーマル版と全改良適用版とではスループットに差はあるものの、Web サーバ以外にボトルネックがない場合は、共にリニアに性能が出ていることが確認できる。

## 4. 考察

本技術検証の結果についての考察を以下に記載する。

### 4.1 パフォーマンス検証

パフォーマンス検証結果から、Ruby で ActiveRecord を利用したプログラムではデータ抽出処理に関して find メソッドをチューニングすることにより、性能が大きく向上することが判った。参照系システムでは、特に複数のデータベースにまたがる検索処理 (select join)、一意のデータを検索する処理 (select distinct)、集計処理 (select group) は、非効率な SQL 文が発行されないよう、find メソッドのチューニングをするとパフォーマンス向上に効果がある。しかし、今回の検証下で比較対象とした Java 版に比べるとまだ性能が及ばなかった。データ抽出処理が多い参照系システムでは memcached を利用し、データ抽出処理性能を向上させるような仕組みを導入することで入出力処理の性能を担保するとよい。

データ格納や更新、削除処理といった一連のトランザクション処理に関しても、find メソッドのチューニングによって、性能を改善できることが判った。ただし、ロールバック処理に関しては現状の ActiveRecord では改善策が無いため、期待する性能を実現するためには、データベースエンジン側のストアードプロシージャを利用するなど Ruby アプリケーションのチューニング以外での対策が必要である。

リソースの使用については、Ruby 版は CPU を 1 コアしか利用できていない。これは MRI の実装仕様から 1 つの Ruby のプロセスから複数の CPU (コア) を使えないことが原因として考えられる。そのため、処理を拡大する際は、スケーラビリティ検証の Web サーバソフトウェアに対するプロセス数の設定を参考に、プロセス数を増やすことで、スループットを向上させることができる。また、メモリ使用量に関しては、SQL レベルで対処できるデータ抽出処理を、Ruby 側で処理するのではなく、データベース側で処理させるように実装することで、データ増加によるメモリ使用量増加を抑制させることができる。

### 4.2 スケーラビリティ検証

スケーラビリティ検証結果から、ノーマル版での結果を基準値とし、他の改善したアプリケーションを検証した場合、平均して秒間 300 スループット程度向上した。実際の Web アプリケーションの現場でも、利用者の増大へ対応する (スケールする) 環境を構築する際に、アプリケーション改良版で実施した基本的なチューニング (P66) による HTML レンダリングの高速化、DB アクセスの高速化、あるいは memcached の利用によるデータベースアクセス数の低減化は効果があり、スループットを向上させることができる。

さらにより多くの処理を行うためには、近年多くの開発現場で取られている、サーバ数を増やすスケールアウトの手段が、Ruby での開発においても有効で、リニアに性能が出ることが確認できた。

また、Ruby によるプログラミングに注目した情報や書籍は増えてきている一方、

Ruby でシステムを構築する際の、サーバの設定などの周辺環境についての情報や書籍は少ないのが現状であるが、アプリケーションとしての処理性能を高めるために Web サーバソフトウェアに対して行った、設定値の調整や、組み合わせの検討プロセスは、実際の Web アプリケーション構築の現場における手引きとなる。

#### 4.3 まとめ

本検証を通じて、Ruby を用いたアプリケーションの「入出力処理」と入出力処理によって構成される「アプリケーションとしての処理プロセス」を想定したベンチマークを取得し、Ruby をアプリケーション開発に用いる際に参考となる計測データやチューニング手法とその効果を抽出することができた。

今回得られた計測データは、実際のアプリケーション開発で Ruby 活用の可否を判断する場合の参考となり、その計測データを得るための手法とプログラムは所与の環境で性能評価する際にも活用できる。

また、本検証におけるチューニング手法の検討とそのチューニング結果から、Ruby 単体やライブラリなどの利用で改善が出来ない場合や、さらなる速度を求める場合でも、他の外部ツールをうまく活用することで十分運用に耐えるソフトウェアを開発できるケースがあることが確認できた。ただし、memcached などの外部ツールを利用する際は、Ruby のクライアントライブラリの情報や、バージョンの整合性など、ソフトウェアの実装や保守面で考慮すべき事項が増える。そのため、まずは今回の検証の中で効果が確認された Ruby または Ruby on Rails の標準機能を利用したチューニング手法の適用を優先し、それでも十分な性能が得られない場合には外部ツールを利用する、といった順序で検討するとよい。

付録1 パフォーマンス検証における検証ケースおよびチューニング一覧

【パフォーマンス検証共通事項】

共通事項	内容	参照ページ
テストの実施方法	検証プログラムを実行し、プログラムが出力するログ（ベンチマーク）を確認する	P17-18 (1) test-ATIS プログラム詳細 ア. 各プログラムでの共通事項 P22 (2) test-transaction プログラム詳細 ア. 各プログラムでの共通事項
想定している業務プロセスと情報処理	業務の拡大等に伴い、 1. データ量が増大した場合における、データ抽出処理（test-ATIS） 2. データ量が増大した場合における、データ格納、更新、削除処理（test-transaction） を想定している。	
計測対象	CPU 時間	
	実行時間	
	メモリ使用量（VSZ,RSS）	
計測条件	2 パターンのデータ量での計測 ・ ベンチマークプログラム（test-ATIS、test-transaction）のデフォルトデータ量 ・ 10 万件相当のデータ量	P12-13 イ.スキーマとデータ量 a.test-ATIS b.test-transaction
データベースの稼働環境（ソフトウェア）	MySQL 5.1.30	P16

## 1. データ抽出処理 (test-ATIS)

### ・チューニングを施す前のプログラム

No.	テストプログラムの名称	テストプログラムの概要	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
1-1	test-ATIS ノーマル版	test-ATIS にある SQL 文と同等の結果が得られるようなリクエストを、ActiveRecord 経由で発行し、SQL 文を意識しない実装を行い、結果を Ruby のオブジェクトとして取得する Ruby スクリプト	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P18-19 P36-P38

### ・チューニング一覧

No.	テストプログラムの名称	テストプログラムの概要	チューニング前に想定された課題 (仮説)	チューニング内容	チューニング結果	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
1-2	test-ATIS find メソッド改良版	1-1(test-ATIS ノーマル版)に対して、ActiveRecord の (find メソッドの) オプションを変更したもの	SQL 文の発行回数が多くなることによりデータベースアクセスがボトルネックになる。取得するカラムが多いことによりメモリリソースの圧迫が生じる。	<p>効率の良い SQL を出力し、SQL 文の発行回数を減らす。取得するカラムを制限する。</p> <p>&lt;チューニング例&gt;</p> <ul style="list-style-type: none"> <li>・ include オプションの利用を止め、joins オプションを利用する。</li> <li>・ select オプションを利用して取得するカラムを制限する。</li> </ul> <p>&lt;サンプルコード&gt;</p> <pre>records = City.find(:all,   :select =&gt;     "city.city_name,state.state_name,city.city_code",   :joins =&gt; :state,   :conditions =&gt; ["city.city_code = :city_code",     {:city_code =&gt; 'MATL'}])</pre>	(比較対象とした Java 版には劣ったものの) データ抽出処理において大きな改善が見られた。データ量が増えても、同等比の改善が見られた。	1-1(test-ATIS ノーマル版)と同じ。 Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P19 P36-37 P39-41

1-3	test-ATIS find_by_sql 利用版	1-1(test-ATIS ノーマル版)に対して、SQL 文生成を ActiveRecord まかせにせず、test-ATIS オリジナルの SQL 文が生成されるように ActiveRecord の使用方法を変更したもの	1-2 の想定課題に加え、ActiveRecord のクエリ生成処理そのものがボトルネックになる。	SQL を直接記述し、ActiveRecord による SQL 文の生成をなくす。 <チューニング例> ・ find_by_sql メソッドを利用してオリジナルと同じ SQL 文を直接記述する。 <サンプルコード> records = City.find_by_sql( "select city.city_name,state.state_name,city.city_code from city,state where city.city_code='MATL' and city.state_code=state.state_code")	1-2 (test-ATIS find メソッド改良版) と改善状況は同等であり、ActiveRecord のクエリ生成処理そのものにボトルネックは見られなかった。	1-1(test-ATIS ノーマル版)と同じ。 Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P20 P36-37 P40-41
1-4	test-ATIS memcached 利用版	1-1(test-ATIS ノーマル版)に対して、データベースからの取得結果を memcached を利用してキャッシングし再利用するようにしたもの	SQL 文の発行回数が多くなることによりデータベースアクセスがボトルネックになる。	検索結果をメモリ上に格納し、メモリ上のデータを利用させることにより、データベースアクセスそのものを減らす。 <チューニング例> ・ memcached に検索対象のキャッシュデータが存在するか確認し、キャッシュが無い場合にデータベースアクセスからデータを抽出する。 ・ キャッシュが存在する場合はデータベースからのデータ抽出処理を省く (memcached のデータを利用する)。 <サンプルコード> cache = MemCache.new(<memcached サーバ名>, <オプション>) unless records = cache["1"] records = データ抽出処理 cache["1"] = records end	大幅な改善が見られた。 データ量が増えても、同等比の改善が見られた。	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8 Memcached 1.2.6	P20-21 P37 P40-41

・(参考) 比較対象のプログラム

No.	テストプログラムの名称	テストプログラムの概要	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
1-5	test-ATIS Java 版	test-ATIS にある SQL 文と同等の結果が得られるようなリクエストを、Hibernate 経由で発行し、結果を Java のオブジェクトとして取得する Java プログラム	Java 1.6.0_11 hibernate 3.3.1 (Annotation Configuration) JDBC:MySQL Connector/J 5.1.7	P27-28

2. データ格納、更新、削除処理 (test-transaction)

・チューニングを施す前のプログラム

No.	テストプログラムの名称	テストプログラムの概要	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
2-1	test-transaction ノーマル版	test-transaction にある SQL 文と同等の結果が得られるようなリクエストを、ActiveRecord 経由で発行し、SQL 文を意識しない実装を行った Ruby スクリプト <ul style="list-style-type: none"> <li>・追加 (INSERT) → create メソッド</li> <li>・更新 (UPDATE) → update メソッド</li> <li>・削除 (DELETE) → destroy メソッド</li> </ul>	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P22-23 P38-41

・チューニング一覧

No.	テストプログラムの名称	テストプログラムの概要	チューニング前に想定された課題 (仮説)	チューニング内容	チューニング結果	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
2-2	test-transaction メソッド改良版	2-1 (test-transaction ノーマル版) で実行されるデータベース操作に対して、データのインスタンス化が行われないよう使用するメソッドを変更したもの	インスタンス生成がボトルネックになる。	メソッドを変更し、インスタンス生成を行わないようにする。 <チューニング例> ・更新処理では、ActiveRecord.update メソッドではなく、ActiveRecord.update_all メソッドを使用する。 <サンプルコード> <pre># チューニング前 @class_object.update(id, :updated =&gt; 1)</pre>	(比較対象とした Java 版を上回る程の) 大幅な改善が見られた。 データ量が増えても、同等比の改善が見られた。	2-1 (test-transaction ノーマル版) に同じ。 Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P23-24 P38-P41

				<pre> # チューニング後 @class_object.update_all("updated = 1", "idn = #{id}")  &lt;チューニング例&gt; -- 削除処理では、ActiveRecord.destroy メソッドでは なく。ActiveRecord.delete メソッドまたは、 ActiveRecord.delete_all メソッドを使用する。 &lt;サンプルコード&gt; # チューニング前 @class_object.destroy(id) # チューニング後 @class_object.delete(id)  &lt;チューニング例&gt; -- 挿入処理では、ActiveRecord::Schema.execute メソ ッドを使用して、SQL で直接データを挿入する。 &lt;サンプルコード&gt; # チューニング前 @class_object.create(:region =&gt; region, : idn =&gt; id, : rev_idn =&gt; rev_id, : grp =&gt; grp, : updated =&gt; 0) # チューニング後 : [SQL 文を組み立てる処理] : ActiveRecord::Base.connection.execute sql </pre>		
--	--	--	--	---	--	--

・(参考) 比較対象のプログラム

No.	テストプログラムの名称	テストプログラムの概要	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
2-3	test-transaction Java 版	<p>test-transaction にある SQL 文と同等のデータベース操作 (追加・更新・削除) が行えるようなリクエストを、Hibernate のメソッドを利用して発行するプログラム</p> <ul style="list-style-type: none"> <li>・追加 (INSERT) → HibernateSession の save メソッド</li> <li>・更新 (UPDATE) → HibernateSession の update メソッド</li> <li>・削除 (DELETE) → HibernateSession の delete メソッド</li> </ul>	<p>Java 1.6.0_11</p> <p>hibernate 3.3.1 (Annotation Configuration)</p> <p>JDBC:mysql Connector/J 5.1.7</p>	P28-29

付録2 スケーラビリティ検証における検証ケースおよびチューニング一覧

【スケーラビリティ検証共通事項】

共通事項	内容	参照ページ
テストの実施方法	パフォーマンス測定・負荷テストツールである <b>JMeter</b> で負荷をかけ、計測対象を記録する	
想定している業務プロセスと情報処理	一連の <b>Web</b> アプリケーションの処理 ( <b>HTTP</b> 経由での検索、更新、削除等の処理) において、利用者の増加等に伴い、同時アクセス数が増大した場合を想定している。	
計測対象	スループット<JMeter より>	
	メモリ使用量<AP サーバより>	
	CPU 使用率<AP サーバより>	
データベースの稼働環境 (ソフトウェア)	<p><b>MySQL 5.1.30</b></p> <p>my-innodb-heavy-4G.cnf に、 下記の修正を加えた。</p> <ul style="list-style-type: none"> <li>- 以下のパラメータを[mysqld]セクションへ追加</li> </ul> <pre>character-set-server=utf8 skip-character-set-client-handshake skip-name-resolve</pre> <ul style="list-style-type: none"> <li>-以下のパラメータをコメントアウトし無効とする</li> </ul> <pre># log-bin=mysql-bin # binlog_format=mixed</pre> <ul style="list-style-type: none"> <li>-以下のパラメータを変更する</li> </ul> <pre>default_table_type = InnoDB innodb_flush_log_at_trx_commit = 2</pre>	<p>P57</p> <p>P60</p> <p>ア.共通</p> <p>a.MySQL</p>

## 1. Web サーバソフトウェアの性能検証と選定

### ・ 共通事項

共通事項	内容	参照ページ
テストプログラムの名称	ノーマル版 Rails PetStore	
テストプログラムの概要	Ruby on Rails 2.2.2 で動作するよう変更を加えた Rails PetStore	P42-P49

### ・ 検証ケース一覧

No.	検証内容	計測条件	設定内容	検証結果	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
1-1	検証環境下での Mongrel プロセス数の最適値を求める	Mongrel の起動 プロセス数 (6, 8, 10, 12, 14, <b>16</b> , 20)	<p>①mod_proxy_balancer の利用設定 以下の2つのコマンドをサーバ上で実行する</p> <pre>\$ a2enmod proxy \$ a2enmod proxy_balancer</pre> <p>②Mongrel の起動プロセス数変更</p> <p>②-1 [RAILS_ROOT/config/mongrel_cluster.yml] ファイルへの定義追加</p> <pre>user: apache cwd: /var/www/petstore log_file: log/mongrel.log port: "3000" environment: production group: apache pid_file: tmp/pids/mongrel.pid servers: &lt;起動プロセス数&gt;</pre> <p>②-2 [/etc/httpd/conf/httpd.conf]ファイルへの定義追加</p> <pre>&lt;VirtualHost *&gt;   ServerName localhost.localdomain   DocumentRoot "/var/www/petstore/public" &lt;Proxy *&gt;</pre>	<p>16 プロセス起動時が最もスループットが高くなった。</p> <p>最大性能時において、Mongrel より秒間約 70 スループット上回った。</p> <p>Mogrel に比べプロセスが有効に CPU を使用していることが判った。</p> <p>⇒後段の「スケーラビリティの向上を検討・改善したアプリケーションでの性能検証」においては、Passenger を採用することとなった。</p>	<p>Ruby 1.8.7-p72</p> <p>Ruby on Rails 2.2.2</p> <p>Mysql-ruby 2.8</p> <p>Mongrel 1.1.5</p> <p>Apache 2.2.3</p>	<p>P60-61</p> <p>P64</p> <p>P68-69</p>

			<pre> Order deny,allow allow from all &lt;/Proxy&gt;  ProxyPass / balancer://localhost/ ProxyPassReverse / balancer://localhost/  &lt;proxy balancer://localhost/&gt; BalancerMember http://localhost:3000 BalancerMember http://localhost:3001 . (起動するプロセス数に応じて、ポート番号のみを変更し記述する) &lt;/proxy&gt; &lt;/VirtualHost&gt; </pre>			
1-2	検証環境下での Passenger プロセス数の最適値を求める	Passenger の 起動プロセス数 (6, 8, 10, 12, 14, 16, 20)	<p>①Passenger の起動プロセス数変更</p> <pre> [etc/httpd/conf/httpd.conf]ファイルへの定義追加 LoadModule Passenger_module /usr/local/lib/ruby/gems/1.8/gems/Passenger-2.0.6/ext/apache2/mod_Pa ssenger.so PassengerRoot /usr/local/lib/ruby/gems/1.8/gems/Passenger-2.0.6 PassengerRuby /usr/local/bin/ruby  PassengerMaxPoolSize &lt;起動プロセス数&gt; PassengerUseGlobalQueue off # VirtualHost の指定 &lt;VirtualHost *&gt; ServerName localhost.localdomain DocumentRoot /var/www/petstore/public &lt;/VirtualHost&gt; </pre>	8~10 プロセス起動時が最もスループットが高くなった。	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8 Passenger 1.0.5 Apache 2.2.3	P61-62 P64 P68-69

1-3	検証環境下での Apache の MPM を決定する	Apache の MPM (prefork, worker)  # Passenger プロセス数は 8	<p>prefork と worker の切り替え</p> <p>①prefork を利用した検証の場合 (Apache のデフォルト設定)</p> <p>[etc/sysconfig/httpd]ファイルで以下の定義をコメントアウトする</p> <pre># HTTPD=/usr/sbin/httpd.worker</pre> <p>②Worker を利用した検証の場合</p> <p>[etc/sysconfig/httpd]ファイルで以下の定義を有効にする</p> <pre>HTTPD=/usr/sbin/httpd.worker</pre> <p>③共通の設定 (本件証で用いた Apache の場合)</p> <p>[etc/httpd/conf.d/php.conf]ファイルを削除もしくは、適当な場所へ移動する。 (Apache をデフォルトインストールした状態で配置されているファイルだが、php が Worker に対応しておらず、Apache 起動時にこのファイルが存在するとエラーとなる為)</p>	<p>秒間当たりのスループットおよび CPU 使用率において、Prefork と Worker には性能差がなかった。</p> <p>Worker の方が、Apache が使用するメモリ使用量を低減していることが確認できた。</p> <p>⇒後段の「スケーラビリティの向上を検討・改善したアプリケーションでの性能検証」においては、Apache Worker MPM を採用することとなった。</p>	<p>Ruby 1.8.7-p72</p> <p>Ruby on Rails 2.2.2</p> <p>Mysql-ruby 2.8</p> <p>Passenger 1.0.5</p> <p>Apache 2.2.3</p>	<p>P62</p> <p>P69-70</p>
-----	----------------------------	---	--	---	---	--------------------------

## 2. スケーラビリティの向上を検討・改善したアプリケーションでの性能検証

### ・チューニングを施す前のプログラム

No.	テストプログラムの名称	テストプログラムの概要	計測条件	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
2-1	ノーマル版 Rails PetStore	Ruby on Rails 2.2.2 で動作するよう変更を加えた Rails PetStore	仮想ユーザ数 (5, 10, 15, 20, 25, 50) Passenger プロセス数 (8) アプリケーションサーバ台数 (1,2)	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P42-49

### ・チューニング一覧

No.	テストプログラムの名称	テストプログラムの概要	チューニング前に想定された課題 (仮説)	チューニング内容	チューニング結果	テストプログラムの稼働環境 (ソフトウェア)	参照ページ
		計測条件					
2-2	アプリケーション改良版 Rails PetStore	2-1 (ノーマル版 Rails PetStore) のデータベース処理と画面レンダリング処理に対して Rails の標準機能を用いて改善を施したプログラム  仮想ユーザ数 (5, 10, 15, 20, 25, 50) Passenger プロセス数 (8)	効率的な SQL 文が発行されないことがボトルネックになる。 部分テンプレートから HTML を生成(レンダリング)する処理がオーバーヘッドになる。	①より処理コストが低くなるよう、find メソッドの記述方法を見直す。  <チューニングの例> ・動的ファインダ(find_by_string_id)を通常の find メソッドに変更する。 ・:select オプションを用いて必要なカラムのデータのみ取得するようにする。  <サンプルコード> <pre># @category = Category.find_by_string_id( #   params[:category].upcase) @category = Category.find(:first,   :select =&gt; 'id, name',   :conditions =&gt; ["string_id = ?",                  params[:category].upcase])</pre>	仮想ユーザ数の増加とともにスケールした。スループットも向上した。	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8	P52-53 P65 P72

				<p>②1つのアクションメソッドで1つのビューファイルを利用するよう、サブテンプレートとレイアウトの利用を廃止する。</p> <p>&lt;チューニングの例&gt;</p> <ul style="list-style-type: none"><li>• VIEW 中にサブテンプレートの記述内容を移動する。</li><li>• VIEW 中にレイアウトの記述内容を移動する。また、レイアウトを使用しないアクションをコントローラに定義する。</li></ul> <p>&lt;サンプルコード&gt;</p> <pre>layout 'application', :except =&gt; [アクション名 1, アクション名 2, ..]</pre>			
--	--	--	--	--	--	--	--

2-3	<b>memcached 利用版 Rails PetStore</b>	2-1 (ノーマル版 Rails PetStore) のセッション情報など使用頻度の高い(読み取り専用) データアクセスに memcached を使用するように改善を施したプログラム	アクセス数の増加に伴い、アプリケーションとデータベース間の負荷が高くなることがボトルネックとなる。	使用頻度の高い機能に対して、データベースから取得したデータメモリ上に格納し、メモリ上のデータを利用させることにより、データベースアクセスそのものを減らす。 <チューニング例> ・ memcached を利用して一度データベースから取得したデータをメモリに格納し、二度目以降のデータの抽出処理ではデータベースにアクセスせず、memcached にアクセスしデータの抽出を行うようにする。 <サンプルコード> <pre> # @products = #   Product.find_by_keywords( #     keywords, #     params[:page]) unless @products =   CACHE[params[:keyword].gsub(' ', '')]   @products =     Product.find_by_keywords(       keywords,       params[:page])   CACHE[params[:keyword].gsub(' ', '')] =     @products end </pre>	アプリケーション改良版と同等程度のスループット改善が見られた。	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8 Memcached 1.2.6	P53 P66 P71-72
		仮想ユーザ数 (5, 10, 15, 20, 25, 50) memcached プロセス数 (1, 8)					

2-4	<b>全改良版</b> <b>Rails PetStore</b>	2-2 (アプリケーション改良版) と 2-3 (memcached 利用版) の両改善を組み合わせたプログラム	2-2 および 2-3 の両方の課題。	2-2 および 2-3 の両方のチューニングを合わせて適用する。	本検証アプリケーション下では、memcached プロセスにかかる負荷は低く、1 プロセスでスケラビリティ検証を行うこととした。アプリケーション改良版および memcached 利用版と同等程度のスループット改善となった。アプリケーションサーバ台数を増やしたスケールアウト検証においては、ノーマル版とともにリニアにスケールした。	Ruby 1.8.7-p72 Ruby on Rails 2.2.2 Mysql-ruby 2.8 Memcached 1.2.6	P53 P66 P71-73
		仮想ユーザ数 (5, 10, 15, 20, 25, 50) memchaced プロセス数 (1, 8) アプリケーションサーバ台数 (1,2)					

オープンソフトウェア利用促進事業

自治体・企業等の情報システムへの Ruby 適用可能性に関する調査  
技術検証報告書

独立行政法人 **情報処理推進機構**

Copyright(c) Information-technology Promotion Agency, Japan. All rights reserved 2009