An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions*

Shahram Ghandeharizadeh and Ankit Mutha

Database Laboratory Technical Report 2014-04

Computer Science Department, USC

Los Angeles, California 90089-0781

{shahram,ankit.mutha}@usc.edu

November 6, 2014

Abstract

With object-oriented programming languages, Object Relational Mapping (ORM) frameworks such as Hibernate have gained popularity due to their ease of use and portability to different relational database management systems. Hibernate implements the Java Persistent API, JPA, and frees a developer from authoring software to address the impedance mismatch between objects and relations. In this paper, we evaluate the performance of Hibernate by comparing it with a native JDBC implementation using a benchmark named BG. BG rates the performance of a system for processing interactive social networking actions such as view profile, extend an invitation from one member to another, and other actions. Our key findings are as follows. First, an object-oriented Hibernate implementation of each action issues more SQL queries than its JDBC counterpart. This enables the JDBC implementation to provide response times that are significantly faster. Second, one may use the Hibernate Query Language (HQL) to refine the object-oriented Hibernate implementation to provide performance that approximates the JDBC implementation.

A Introduction

Hibernate [9] is an object-relational mapping (ORM) framework for the Java programming language. It supports three key features. First, given the persistent class definitions of an application, it generates its relational database schema. Second, it populates instances of a persistent class by issuing SQL queries to the underlying relational database management system (RDBMS) *transparently*. A persistence class has a primary key and each of its instances correspond to a row of a table in the RDBMS. Third, Hibernate propagates an application's changes[1] to the underlying database

*In Proceedings of the 16th ACM International Conference on Information Integration and Web-based Applications and Services, Hanoi, Vietnam, December 2014.

[1] A change to a persistent instance might be an update to one or more of its property values, deletion or creation of an instance.

transparently and in a transactional manner as necessary. These three features render a Hibernate implementation of an application portable to a wide variety of RDBMSs such as MySQL, Oracle, PostgreSQL among others. Moreover, pundits point out that Hibernate enhances productivity of developers by relieving them from common data persistence related programming tasks, enabling them to focus on the requirements of the application and their implementation using an object-oriented language.

A natural question is what is the run-time overhead of using the Hibernate ORM? In this paper, we answer this question using a social networking benchmark named BG [1, 2]. BG populates an RDBMS with a social graph consisting of a fixed number of members, friends per member, and resources per member. It consists of social networking actions such as extend a friendship invitation to a member and view a member profile. See Table 2 for a list of the BG actions considered in this study. We use BG to establish two important metrics: Average response time ($\overline{RT}$) and Social Action Rating (SoAR). $\overline{RT}$ is the average amount of time elapsed from when BG issues an action to the time the action is processed using an idle system with no background load. SoAR of an implementation is its highest number of supported concurrent actions (system throughput measured in requests per unit of time) that satisfy a pre-specified Service Level Agreement, SLA. The SLA used throughout this study requires 95% of requests to observe a response time equal to or faster than 100 milliseconds. We compare $\overline{RT}$ and SoAR of a Hibernate implementation of BG's actions with a JDBC implementation. The implementation with the faster $\overline{RT}$ is superior. Similarly, the system with the highest SoAR is more desirable as it processes a larger amount of work while providing the same quality of service. We observe that for certain actions of BG, two different implementations may provide similar SoARs while one is considerably faster than the other (provides a lower $\overline{RT}$). This is due to a resource of the RDBMS server such as its networking card becoming fully utilized and dictating the same SoAR for all implementations.

We chose BG for this evaluation because it has a conceptual data model (see Figure 1) that consists of multiple entity sets and relationship sets with different mapping cardinalities, both one-to-many and many-to-many. BG's data model is more complex than that of the YCSB [7] and YCSB++[12] benchmarks that would consist of a single entity class with Hibernate and reduce to one table. BG's use of both foreign and primary keys highlights the different system behavior observed with Hibernate. We also considered the use of TPC-C/E [8] and an e-commerce benchmark such as RUBiS [6] and RUBBoS [11]. Ultimately, we selected BG because of its emphasis on social networking actions, its ability to quantify both the response time and SoAR of the alternative implementations [3, 4, 10], and the amount of unpredictable (stale, erroneous, simply wrong) data produced by an implementation. The latter is useful for our future investigation of Hibernate with caches, see Section D.

Our objective is to understand the impact of using an object-oriented representation, termed *OO-Hibernate*, on both the response time and SoAR of different social networking actions when compared with an implementation that uses the relational representation directly. The later is termed the *JDBC* implementation because it requires an application developer to author SQL queries and provide software to address the impedance mis-match between objects and tables. We also analyze the use of the Hibernate Query Language (HQL), a query syntax similar to SQL for issuing queries

at the object level, to improve upon OO-Hibernate and enhance its performance. This implementation is named *HQL-Hibernate*.

Our key findings are as follows. First, Hibernate implements BG's schema effectively with MySQL, producing a finely-tuned physical relational data design with correct index structures and foreign-key relationships. Second, both HQL-Hibernate and JDBC outperform OO-Hibernate by implementing BG's actions with fewer queries. OO-Hibernate issues more queries for those actions that navigate either a many-to-many or a one-to-many relationship. With a many-to-many relationship, its issued number of queries is the cardinality of the relationship. For example, to list friends of a member A, if A has 100 friends then OO-Hibernate issues 100 SQL queries. JDBC issues only one SQL query using the join operator. One may realize the same by issuing an HQL query as implemented in the HQL-Hibernate. Third, while HQL-Hibernate provides slower response times than JDBC, its SoAR is comparable to JDBC for most of BG's actions.

We evaluate the different implementations using the same hardware and software platform, consisting of one RDBMS server PC and one or more BGClient PCs connected using a Gigabit switch. A BGClient implements either the JDBC, OO-Hibernate, or HQL-Hibernate. The RDBMS server hosts the MySQL relational database management system (RDBMS) versions 5.5 configured with 3 Gigabyte of memory [2]. The server hosting the MySQL RDBMS is an Intel dual core CPU E8600 (3.33GHz) with 4 Gigabytes of memory. Its operating system is the Ubuntu 12.04 LTS.

BG is a scalable benchmark [2] and we replicated the BGClients as necessary to prevent the benchmarking infrastructure from becoming the bottleneck. In all reported SoAR, a resource in the RDBMS becomes fully utilized to dictate the system performance. This explains why HQL-Hibernate provides comparable SoAR ratings to the JDBC implementation.

To the best of our knowledge, this is the first study to evaluate Hibernate for processing interactive social networking actions. An interesting outcome of this investigation is the observation that the OO-Hibernate implementation issues more SQL queries to implement an action than its JDBC and HQL alternative. The rest of this paper is organized as follows. Section B presents BG's conceptual schema and an implementation of this schema along with BG's actions using Hibernate version 3.6.10 [9]. Section C rates this implementation using BG and compares it with a JDBC implementation. Section D provides brief conclusions.

B A Hibernate Implementation of BG

Figure 1 shows the conceptual design of BG's social graph used for this evaluation. (See [1, 2] for a comprehensive description of BG.) The Members entity set contains those users with a registered profile. It consists of a unique identifier and a fixed number of string attributes. In our experiments, each member has two images: a 2 KB thumbnail image and a 12 KB profile image. Thumbnail images are displayed when listing friends of a member and profile

[2]Hibernate supports a wide range of RDBMSs and our decision to use MySQL is arbitrary. Our quantitative observations such as Hibernate issuing more queries than necessary are independent of the choice of an RDBMS.

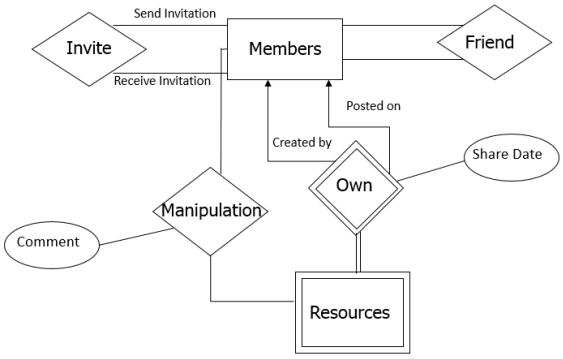

Figure 1: Conceptual design of BG's database.

image (which can be of higher resolution) is displayed when visiting a profile. A member may extend an invitation to or be friends with another member, represented using Friends and Invite relationship sets, respectively. A resource may pertain to an image, a posted question, a technical manuscript, etc. These entities are captured in one set named "Resources". In order for a resource to exist, it must be "Owned" by a member. A member may post a resource, say an image, on the profile of another member, represented as a relationship between two members and a resource. A member may comment on a resource. This is implemented using the "Manipulation" relationship set.

We used Hibernate's annotations to design entity classes Users and Resource corresponding to the Members and Resources entity sets in the conceptual model. We modeled the relationship sets Friend, Invite, Own, and Manipulation using entity classes Friendship, Invitation and Manipulation, respectively. We created collections of entity classes as required to represent one-to-many and many-to-many associations in Hibernate. This results in the following collections:

- Seven collections with the User entity class: Friends, FriendsOf, CreatedResources, WallResources, Manipulations[3], PendingFriends, and InvitedFriends[4], and
- One collection with the Resource entity class, named Manipulations.

Table 1 shows each collection and its entity class type.

During the entity-class design step, we used the default setting of Hibernate that populates collections in a lazy manner. This means a collection such as Friends of a User object is not populated unless a getter method that references this collection is invoked.

We used Hibernate to generate the relational schema (i.e., hbm2ddl) for MySQL. It authors the following tables (the underlined attributes are indexed while each of the italic attributes denote a foreign key):

[3]These are the comments posted by a user. This collection is not used to implement a BG action. However, it is included to be consistent with the conceptual design shown in Figure 1. Its presence has no impact on the performance of Hibernate.

[4]This collection is not used to implement a BG action. However, in contrast to the Manipulations collection, Hibernate requires the InviteFriends collection to implement the many-to-many Invite relationship set between Members (instances of the User entity class).

Collection	Entity class
Friends	Friendship
FriendsOf	Friendship
Createdresources	Resource
WallResources	Resource
Manipulations	Manipulation
Pending Friends	Invitation
Invited Friends	Invitation

Table 1: Collections and their entity class type.

BG Social Actions	Type	Very Low (0.1%) Write	Low (1%) Write	High (10%) Write
View Profile, VP	Read	40%	40%	35%
List Friends, LF	Read	5%	5%	5%
View Friend Requests, VFR	Read	5%	5%	5%
Invite Friend, IF	Write	0.04%	0.4%	4%
Accept Friend Request, AFR	Write	0.02%	0.2%	2%
Reject Friend Request, RFR	Write	0.02%	0.2%	2%
Thaw Friendship, TF	Write	0.02%	0.2%	2%
View Top-K Resources, VTR	Read	49.9%	49%	45%
View Comments on a Resource, VCR	Read	0%	0%	0%
Post Comment on a Resource, PCR	Write	0%	0%	0%
Delete Comment from a Resource, DCR	Write	0%	0%	0%

Table 2: Three mixes of social networking actions.

- User(userid, username, pw, firstname, lastname, dob, gender, jdate, ldate, address, email, tel, profileImage, thumbnailImage)
- Friendship (*friend1, friend2*)

  The foreign key dependence is on the User row with userid equal to either friend1 or friend2.
- Invitation(*inviter, invitee*)

  The foreign key dependence is on the User row with userid equal to either inviter or invitee.
- Resource(rid, *creator, walluser*, type, body, doc)

  The foreign key dependence is on the User row with userid equal to either creator or walluser.
- Manipulation(mid, *rid*, *modifier*, timestamp, type,content)

  The foreign key dependence is on the User row with userid equal to modifier and Resource row with the same rid.

One may use BG to create different social graphs with a fixed number of members, friends per member, and resources per member. It is an extensible benchmark that supports different interactive social networking actions. An experimentalist may define arbitrary workload with a mix of actions by manipulating the frequency of each action, see

Table 2. Below, we describe the different actions and their implementation using Hibernate. With the first two actions, View Profile and List Friends, we analyze the behavior of the Hibernate implementation as a function of the number of friends per member with a social graph consisting of 10,000 members and 100 resources per member.

The Hibernate implementation of the different BG actions is as follows. Each action starts by opening a session with the session factory, begins a transaction with the session, followed with an instantiation of different entities and invocation of their getter and setter methods to implement the action. An action ends by committing the transaction and closing the session.

With both OO-Hibernate and HQL-Hibernate, we quantify the number of SQL queries generated to implement a BG action by configuring Hibernate to print its issued SQL queries. This was realized by setting the configuration property called "show_sql" to true in the configuration file "hibernate.cfg.xml". (This property was set to false when evaluating Hibernate.) Next, we use BG's command-line tool to issue an action, causing Hibernate to print the SQL queries issued for that action.

HQL-Hibernate issues fewer SQL queries for all actions except Post Comment on Resource (PCR) because it inserts rows in tables and HQL does not support an equivalent command[5]. In general, the HQL syntax and its use case resembles SQL using JDBC. HQL-Hibernate is faster than OO-Hibernate and provides comparable SoAR to the JDBC native implementation.

We now describe a Hibernate implementation of the eleven BG actions in turn.

View Profile, VP: The VP action emulates a socialites visiting the profile of either herself or another member. Its input include the socialite's id and the id of the referenced member, U_r. A socialite's id may equal U_r, emulating a socialite referencing her own profile. The output of VP is the profile information of U_r, including U_r's attribute values and the following two aggregate information: U_r's number of friends and number of resources. If the socialite is referencing her own profile (socialite id equals U_r's id) then VP retrieves a third aggregate, U_r's number of pending friend invitations.

OO-Hibernate implements VP by instantiating a User entity with userid equal to U_r (using session.get by specifying the User entity class and U_r). Next, it populate the collections of this object by invoking its getters. For the three aggregates, it computes the size of their respective collections. This implementation (transparently) issues the following five SQL queries to the MySQL RDBMS:

- An SQL query with an exact match selection predicate referencing the userid of the User table to retrieve the member attribute values to populate the primitives of the instantiated User object.
- An SQL query with an exact match selection predicate referencing the walluser of the Resource table, populating the WallResources collection in response to the getter for the size of this collection.
- Two SQL queries with an exact match selection predicate referencing each attribute of Friend1 and Friend2 of

[5]Hibernate HQL supports only "Insert into ... SELECT ..." commands. It does not support "Insert Into .... Values ..." command.

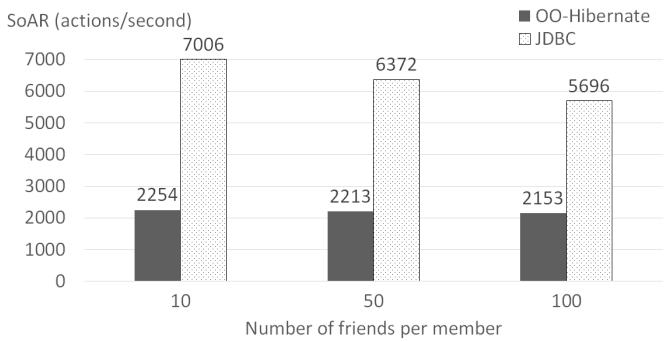

Figure 2: SoAR of the VP action with 10K members as a function of the number of friends per member.

the Friendship table. These two queries fetch all confirmed friendships for the member as a friendship between two members is represented as one row.

In our design of friendship relation we have two collections of Friendship entity class in User entity class, one maintains the friendship invitations initiated by U_r that were confirmed (named Friends) and the second consist of friendship invitations received and confirmed by U_r. The total number of U_r's friends is the sum of the size of these two collections.

An alternative may have been to represent a friendship relationship as two rows. This would have reduced the number of queries to compute U_r's friends to one. Hibernate does not provide a mechanism to model a friendship in this manner.

- An SQL query with an exact-match selection predicate referencing Invitee attribute of the Invitation table, computing all pending friendship invitations extended to the member with userid equal to U_r.

Note that the last query is issued only when the socialite id equals U_r's id. Thus the total number of SQL queries is either 4 or 5.

Figure 2 shows the SoAR of the OO-Hibernate implementation with the VP action as we vary the number of friends per member from 10 to 50 and 100. As a comparison, we show the observed SoAR with the JDBC implementation of Section C. (The JDBC implementation provides a higher performance because it issues four queries instead of five, see discussions of Section C).

With the HQL-Hibernate, we replace the two queries used to retrieve the friends of U_r with one HQL query that employs the Boolean *or* connective: Friend1=U_r or Friend2=U_r. This enhances SoAR by almost a factor of two from 2,153 to 4,101 actions with 100 friends per member.

List Friend, LF: The LF action emulates a socialite A viewing member U_r's list of friends. Similar to the discussion of VP, U_r may equal A to emulate the socialite viewing her own list of friends. LF retrieves the profile information of each friend including their thumbnail image and excluding their profile image.

Using OO-Hibernate, we implement LF in three steps. First, we instantiate a User entity with userid equal to A.

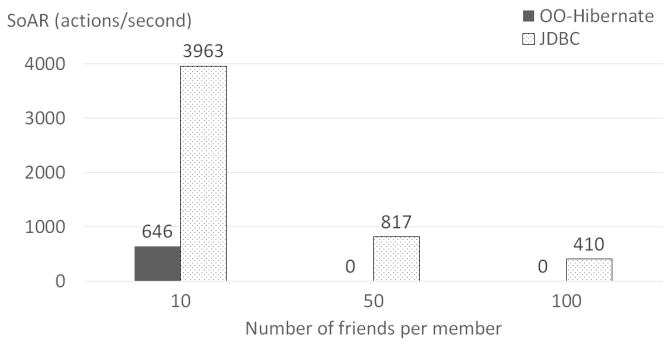

Figure 3: SoAR of LF with 10K members as a function of the number of friends per member.

Second, we populate the Friends and FriendsOf collections by invoking the getter method of the instantiated User object. Third, we iterate each collection of Friendship entities, and invoking the getter method on friend1 and friend2 attributes of the Friendship entities which gives us a User entity for each friend in order to retrieve the attributes of that member including their thumbnail image. This implementation issues one SQL query for the first step, two SQL queries for the second step, and ϕ queries for the third step where ϕ is the number of friends per Member U_r.

The SQL queries issued by the first and second step are similar to those with the VP action. The two SQL queries issued by the second step examine member A in Friend1 and Friend2 columns of each row of the Friendship table as a friendship is represented by one row of this table.

The performance of OO-Hibernate's implementation of LF is sensitive to the number of friends per member, ϕ, and issues ϕ+3 queries. Figure 3 shows SoAR of this implementation as a function of 10, 50, and 100 friends per member for a social graph consisting of 10,000 members. SoAR of OO-Hibernate drops to zero with 50 and 100 friends per member because its provided response time is greater than 100 milliseconds even with one socialite accessing the system.

The HQL-Hibernate improves upon the third step of LF to issue a single query instead of ϕ queries. With this change, the SoAR of HQL-Hibernate improves to 412 actions per second with 100 friends per member, ϕ=100. This is comparable to the SoAR observed with JDBC as the one Gigabit network card of the RDBMS server becomes fully utilized, dictating the SoAR of these two implementations.

View Friend Request, VFR: The VFR action emulates a Socialite A retrieving her pending friend invitations. This retrieves the profile of each member who has extended a friend invitation to A. Its OO-Hibernate implementation instantiates a User entity with userid equal to A. This issues an SQL query similar to VP and LF to retrieve the user row corresponding to A in order to populate the object. Next, VFR invokes a getter method that populates the collection PendingFriends by issuing an SQL query similar to Step 2 of LF. A key difference here is that PendingFriends is asymmetric, resulting in one SQL query instead of two as member A is only checked in invitee column of Invitation table. Next, we iterate this collection of Invitation entities and invoke getter method for inviter attribute to instantiate the User entity corresponding to each userid, causing OO-Hibernate to (transparently) issue an SQL query for each

inviter. Similar to the discussion of LF, the number of issued SQL queries is A's number of pending friend invitations plus 2.

The SoAR of VFR with OO-Hibernate and its HQL-Hibernate derivative is similar to LF and not repeated.

Invite Friend, IF: The IF action emulates a Socialite A extending an invitation to Member U_r. Its OO-Hibernate implementation instantiates two objects of the User entity class, corresponding to Member A and U_r, respectively. This results in 2 SQL queries similar to the discussion of VP. Next, it instantiate an object of Invitation entity class by passing the given members to session.get to check if this object already exists. It skips duplicating the creation if it already exists. This check requires one SQL query. If no invitation corresponding to given members exists, OO-Hibernate instantiate an object of the Invitation entity class with invitee as the object U_r and the inviter as the object A. Finally, it persists the Invitation object by issuing session.save command. This causes OO-Hibernate to issue the following insert SQL command: insert into Invitation(Inviter, Invitee) values (?, ?) using object A and U_r's userid. In sum, IF issues 4 SQL commands.

HQL-Hibernate improves upon the OO-Hibernate by not issuing the query to check if the friendship exists, reducing the total number of queries to 3.

Accept Friend Request, AFR: When a Socialite A accepts the friend invitation of Member U_r, the OO-Hibernate implementation instantiates an object of the User entity class for A and U_r, issuing 2 SQL queries. Next, we instantiate an object of the Invitation entity corresponding to U_r inviting A, causing OO-Hibernate to issue an SQL query to populate this object. We delete this Invitation object, causing OO-Hibernate to issue an SQL command to delete the corresponding row from the Invitation table. Next, we instantiate an object of the Friendship entity using entity class constructor with U_r and A as two participating members. Finally, we save the Friendship object, causing OO-Hibernate to issue an SQL insert command to insert a row in the Friendship table. In sum, this command issues 5 SQL commands with Hibernate.

HQL-Hibernate reduces the number of SQL commands to 4 by simply deleting the invitation row without retrieving it. (The OO-Hibernate issues one SQL query to populate the invitation object and a second one to delete it while the HQL-Hibernate simply deletes the invitation.)

Thaw Friendship, TF: When Socialite A thaws friendship with Member U_r, the OO-Hibernate implementation instantiates an object of the User entity class for A and U_r, issuing 2 SQL queries. Next, it instantiates one object of the Friendship entity using session.get by specifying A and U_r as the primary key of entities of interest. If the returned entity is null then we reverse the order of A and U_r with the session.get method. This requires up to two SQL queries, as the friendship table could have Friend1 as A and Friend2 as U_r or it's reverse. We delete this Friendship entity using session.delete and commit. With OO-Hibernate, TF issues either 4 or 5 SQL commands[6].

HQL-Hibernate reduces the number of issued SQL commands to one by simply deleting the object (row) for the friendship.

[6]This is because one row represents a friendship relationship. If we had represented the friendship between A and U_r as two rows [5] then the number of issued queries would have been six.

Reject Friend Request, RFR: When Socialite A rejects U_r's friend invitation, the OO-Hibernate processing is similar to TF with the difference that it manipulates an object of the Invitation entity. However, this action issues 4 SQL queries. This is one less than the TF action because friendships are symmetric but stored as one row in join table while invitations are asymmetric.

Similar to TF, HQL-Hibernate reduces the number of issued SQL commands to one by simply deleting the object (row) for the pending friend invitation.

View Top-K Resource, VTR: The OO-Hibernate implementation of VTR instantiates an object of the User entity class for the Socialite A invoking the action, issuing one SQL query. Next, it populates the collection WallResource by invoking the corresponding getters, issuing one SQL query. Each element of WallResource collection is of type Resource and has all attributes of that resource because this is a one-to-many relationship from a user to resources. Thus, our implementation returns the top K Resource entities without issuing additional SQL queries. This is in sharp contrast to the LF action where friendship is a many-to-many relationship. In sum, the VTR action issues 2 SQL queries.

HQL-Hibernate issues one HQL query that references the Resource class with WallUser equal to Socialite A, sorts them using "order by" clause, and specifies the limits with the value k (using setMaxResults method). The syntax looks very similar to SQL.

View Comment on a Resource, VCR: When Socialite A invokes VCR on a resource with rid R, the OO-Hibernate implementation instantiates its corresponding Resource object, issuing an SQL query using the specified R. Next, it populates the manipulation collection of this object by invoking its getter that retrieves all posted comments and their attribute values. This issues a second SQL query. Finally, the BGClient iterates the collection and retrieves the different manipulation objects. In sum, VCR issues two SQL queries.

HQL-Hibernate issues one HQL query that references the Manipulation class with the specified R (resource id), reducing the number of SQL queries to one.

Post Comment on a Resource, PCR: When Socialite A posts a comment on a resource with rid R, the OO-Hibernate implementation instantiates an object of the Resource entity using the specified rid R, issuing an SQL query. Next, it instantiates an object of the Manipulation entity and populates it with the input attribute values of the comment being posted. Finally, we save the manipulation object, issuing one SQL insert command. In sum, PCR issues two SQL commands.

HQL-Hibernate cannot reduce the number of SQL queries further because HQL does not support a command equivalent to "Insert into ... Values ...".

Delete Comment on Resource, DCR: When Socialite A deletes a comment posted on a resource with rid R, the OO-Hibernate implementation instantiates an object of the resource entity using the specified rid R, issuing an SQL query. Next, we instantiate an object of the Manipulation entity class using this resource object and the provided mid, issuing a second SQL query. Finally, we delete this object and commit, issuing an SQL delete command. In total,

	VP	LF	VTR	10% Mix
OO-Hibernate	121	93	252	106
HQL-Hibernate	1.9	5.5	0.6	26
JDBC	1.1	3.4	0.3	21

Table 3: Average response time, $\overline{RT}$, in milliseconds for the View Profile (VP), List Friend (LF), View Top-K Resource (VTR), and a mix of read and write actions with a 100K social graph, 100 friend and 100 resources per member.

DCR issues 3 SQL commands.

HQL-Hibernate issues one HQL delete command that references the Manipulation class with the specified R, resource id, reducing the number of SQL commands to one.

C A Comparison with JDBC

This section compares the Hibernate implementations of Section B with a JDBC implementation of BG consisting of the following five tables:

- Users(userid, username, pw, fname, lname, gender, dob, jdate, ldate, address, email, tel, profileImage, thumbnailImage)
- Friendship (*friend1, friend2*)
- Invitation(*inviter, invitee*)
- Resource(rid, *creatorid, walluserid*, type, body, doc)
- Manipulation(mid, *rid*, *modifierid,creatorid*, timestamp, type,content)

Table 3 shows the average response time with a single threaded BG benchmark issuing different workloads: 100% View Profile (VP), 100% List Friend (LF), 100% View Top-K Resource (VTR), and a 10% mix of read and write actions (see Table 2). These results show the JDBC implementation is faster than the two Hibernate implementations. Both JDBC and HQL-Hibernate issue fewer queries than OO-Hibernate and outperform it. HQL-Hibernate incurs the overhead of copying data across extra software layers, providing a slower response time than JDBC.

Next, we compare SoAR of different implementations with alternative workloads. SoAR of OO-Hibernate is zero in all experiments due to its high response time (see Table 3) for processing the different actions. Even with one thread, OO-Hibernate fails to provide a response time of 100 milliseconds or faster for the different workload.

Figure 4 shows SoAR of HQL-Hibernate and the JDBC implementation using MySQL configured with 3 Gigabyte of memory. With the VP action, JDBC outperform HQL-Hibernate by more than a factor[7] of two. With the LF action,

[7]When MySQL is configured with a different amount of memory, this trend may change. For example, with a 1 Gigabyte buffer pool, SoAR of VP with HQL-Hibernate and JDBC is 363 and 263, respectively. This is a 50% difference. We speculate the trend changes because of the 12 KB profile image and its representation as a blob using MySQL. See [5] for a discussion of alternative representations of the profile image to enhance SoAR.

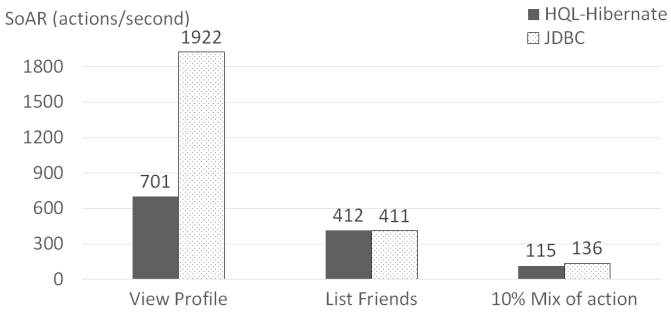

Figure 4: SoAR of HQL-Hibernate and JDBC with a 100K social graph, 100 friends and resources per member.

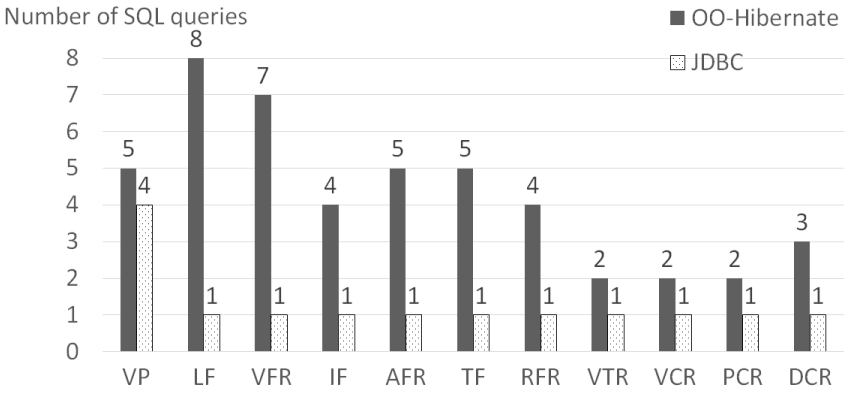

Figure 5: Number of SQL queries issued with OO-Hibernate and JDBC for each BG action assuming 5 friends per member and 5 pending friend invitations per member.

the observed throughput is almost identical with both JDBC and HQL-Hibernate with the network card of the server hosting MySQL becoming fully utilized. With the 10% mix of write actions, the View Profile action constitutes 35% of the workload (see Table 2) enabling JDBC to provide a slightly higher SoAR than HQL-Hibernate.

D Conclusion

This paper evaluates the Hibernate ORM by comparing it with a JDBC implementation using the BG benchmark. We observed Hibernate to produce a finely tuned physical design of a relational database for BG's schema. A purely object-oriented Hibernate implementation of BG's actions issues more SQL queries than a JDBC implementation, see Figure 5. With actions that navigate a many-to-many relationship, the number of SQL queries is a function of the mapping cardinality of the relationship. For example, with the List Friend (LF) action referencing a member with five friends, five SQL queries are issued. This higher number of SQL queries slows down the purely object-oriented Hibernate when compared with a JDBC implementation. One may use the Hibernate Query Language, HQL, to reduce the number of issued SQL queries. This speeds up the Hibernate implementation dramatically, approximating the response times observed with a JDBC implementation. When a resource of the RDBMS server such as its network interface card becomes fully utilized, the observed throughput with the HQL-Hibernate and the JDBC implementation will be approximately the same.

A future research direction is to investigate alternative physical designs similar to those explored in [5] and their impact on the Hibernate implementation. We also intend to analyze the use of caches and optimization techniques (using HQL) that close the gap between the Hibernate and the JDBC implementations.

E Acknowledgments

We thank Sumita Barahmand for her valuable comments on an earlier draft of this paper.

References

[1] S. Barahmand. Benchmarking Interactive Social Networking Actions, Ph.D. thesis, Computer Science Department, USC, 2014.

[2] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR*, January 2013.

[3] S. Barahmand and S. Ghandeharizadeh. Expedited Benchmarking of Social Networking Actions with Agile Data Load Techniques. *CIKM*, 2013.

[4] S. Barahmand, S. Ghandeharizadeh, and D. Montauk. Extensions of BG for Testing and Benchmarking Alternative Implementatios of Feed Following. *ACM SIGMOD Workshop on Reliable Data Services and Systems (RDSS)*, 2014.

[5] S. Barahmand, S. Ghandeharizadeh, and J. Yap. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. *CIKM*, 2013.

[6] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *OOPSLA*, pages 246–261, 2002.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[8] Transaction Processing Performance Council. TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp.

[9] C. Bauer G. King, M. R. Andersen, E. Bernard, S. Ebersole, and H. Ferentschik. *HIBERNATE - Relational Persistence for Idiomatic Java*. Red Hat Middleware, Inc., June 2009.

[10] S. Ghandeharizadeh, R. Boghrati, and S. Barahmand. An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions. *TPC Technology Conference on Performance Evaluation and Benchmarking*, 2014.

[11] ObjectWeb. RUBBoS: Bulletin Board Benchmark, http://jmob.ow2.org/rubbos.html.

[12] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Cloud Computing*, New York, NY, USA, 2011. ACM.