STUDIA INFORMATICA 2013
Volume 34 Number 2A (111)

Ewa PŁUCIENNIK-PSOTA, Tomasz PŁUCIENNIK
Silesian University of Technology, Institute of Computer Science

USING OBJECT DATABASE AS A CACHE FOR A RELATIONAL DATABASE − PERFORMANCE AND USAGE CONSIDERATIONS

Summary. Every non trivial application cooperates with database, usually of relational type. In most cases this cooperation decreases the performance. The fastest, but not too handy, way to access a relational data from an object application is SQL. As alternative one can use object-relational mapping and/or object or NoSQL database. Object database paradigm is the same as object application uses. This makes it possible to avoid characteristic concept dualism (so-called impedance mismatch) and resulting from this need to transform relations into objects and vice-versa. This article presents an attempt to use ORM and object database to increase performance of accessing relational database.

Keywords: object application, relational database, object database, impedance mismatch, object relational mapping, caching, JPA, Hibernate, db4objects, data replication system

WYKORZYSTANIE OBIEKTOWEJ BAZY DANYCH JAKO PAMIĘCI PODRĘCZNEJ DLA RELACYJNEJ BAZY DANYCH – ROZWAŻANIA DOTYCZĄCE WYDAJNOŚCI I UŻYTKOWANIA

Streszczenie. Każda nietrywialna aplikacja współpracuje z bazą danych, zwykle typu relacyjnego. W większości przypadków konieczność tej współpracy obniża wydajność. Najszybszym, choć nie najwygodniejszym sposobem dostępu do relacyjnych baz danych jest język SQL. Jako alternatywę można wykorzystać narzędzia mapowania obiektowo-relacyjnego (ORM) i/lub obiektowe bazy danych lub bazy NoSQL. Paradygmat obiektowych baz danych jest taki sam jak obiektowej aplikacji. Pozwala to na uniknięcie swoistego dualizmu pojęć (tzw. niedopasowania impedancji), a co za tym idzie – konieczności transformacji relacji na obiekty i vice-versa. Niniejszy artykuł przedstawia próbę wykorzystania narzędzi ORM i obiektowej bazy danych do zwiększenia wydajności dostępu do relacyjnej bazy danych.

Słowa kluczowe: aplikacja obiektowa, relacyjna baza danych, obiektowa baza danych, niezgodność impedancji, mapowanie obiektowo-relacyjne, pamięć podręczna, JPA, Hibernate, db4objects, system replikacji danych

1. Introduction

In present-days majority (if not all) production applications cooperate with a database, typically a relational one. In a relational database data is stored in tables of rows related to each other, containing columns of values with specified types [1, 2]. In terms of object applications (i.e. created using object programming languages like Java or C#) accessing the databases is much more convenient using objects instead of tables and rows. One have to build bridges between these two realms that have different paradigms which leads to a very adverse effect called object-relational impedance mismatch[1]. One way to avoid this discrepancy is to use an object database instead of a relational database. As for now object databases have only small part of the market. Relational model is standardized and have been widely used for decades. Its strength lies in common query language − SQL. Although attempts were made to create a counterpart query language for object databases since early 1990s [3, 4], they were not successful. Object Query Language defined by Object Data Management Group standard never achieved the same popularity as SQL. Still most of IT projects consist of an object application, a relational dataset and (in many cases) an additional layer called Object-Relational Mapping (ORM).

Using ORM increases system load. Of course ORM solution can be tuned to increase the performance [5, 6]. However, an object database, which internally stores objects instead of tables and has its own access interface, will probably perform faster than ORM with a relational database. This article present first practical tests of using an object database as a cache for relational database, the idea presented in [7].

2. Caching in Database Access

In this paper it is assumed that the source data originate from a relational database which stays a main data storage (e.g. a legacy database). In case of already existing and maintained data it is hard to migrate everything into new structures. The idea here is to increase performance of access to such data without undertaking a revolution. Instead an evolution is proposed which eventually could led to superseding the relational database by an object one.

[1] Term "impedance mismatch" comes from electrical engineering and stands for a resistance mismatch of source and receiver, which causes loss of power.

From this point on focus will be put on Java-based technologies. There are tools for Java platform, which can help in proving the validity of the idea without implementing everything from scratch. The idea presented in this paper is to replace the second level cache in any JPA (Java Persistence API) implementation with an object database. This new second level cache would also be used more extensively. Load of the relational database will be limited to rare updates of its state against the new cache an vice-versa. Using ODBC/JDBC and SQL is considered faster but from the developer point of view using JPA and objects (POJOs) is more convenient. Furthermore, the object database access should be faster than querying tables and be even faster in comparison to JPA.

The components chosen to verify the idea usefulness are: Hibernate as a representative of the object-relational mapping tools and Versant's db4o object database [8]. There are of course other candidates with similar capabilities, like Versant VOD (which is a non-free product), ObjectDB[2] or any other object database. However, db4objects can be easily synchronized with other relational data stores using dRS (db4o Replication System) system [8].

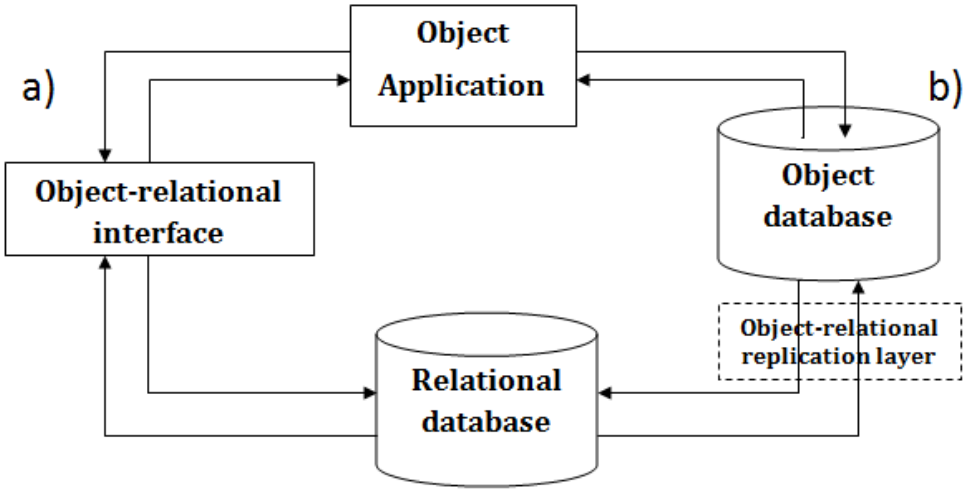


Fig. 1. Traditional (a) and object (b) access to relational database
Rys. 1. Tradycyjny (a) i obiektowy (b) dostęp do relacyjnej bazy danych

As show in Fig. 1 between the client application and the relational database an intermediate module is placed. It contains the object database and replication layer responsible for keeping both databases synchronized (using JPA). All traffic between the application and the relational database is routed through the object database which should limit response times on the client side. Bidirectional synchronization of the databases occurs periodically in the background, completely transparent for the end user. Frequency of updates will be related to data character (e.g. often changing data should be updated as soon as possible). It is also required to consider what is the cost-effective update type (full update, incremental update).

[2] http://www.objectdb.com/

3. Test Environment

Instead of implementing the new JPA or even accommodating an existing one a prototype system was created.

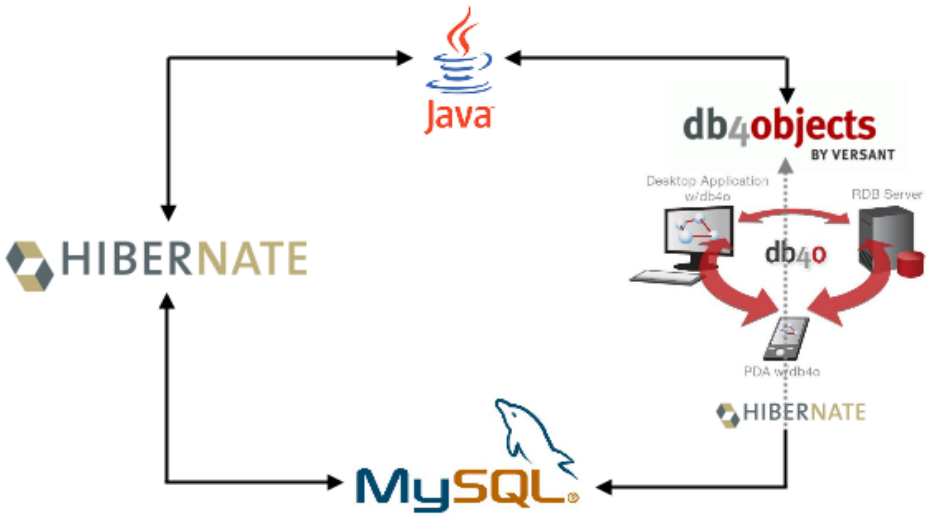


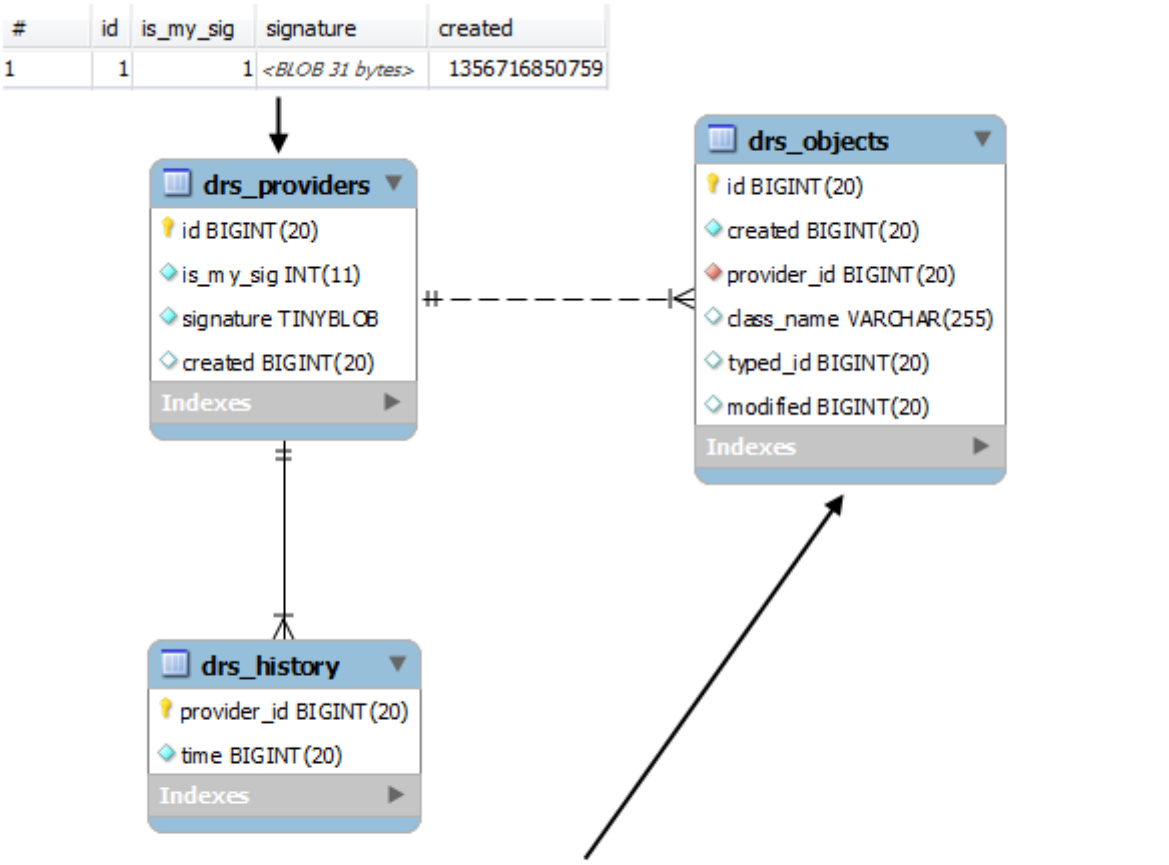
Fig. 2. Elements of the test system corresponding to Fig. 1
Rys. 2. Elementy systemu testowego odpowiadające rys. 1

Fig. 2 represents the test system with elements corresponding to the Fig. 1. MySQL 5.5 database is the typical representative of a relational database. It can be accessed using Hibernate library, which is one of the classic solutions. db4o database and dRS are accordingly the object database cache and the module responsible for synchronization of the data stores. dRS has the capability to communicate with the relational database using Hibernate, so together they correspond to object-relational replication layer in Fig. 1. On both previous figures no ORM was placed between the client application and the object database (ORMs for object databases do exist, like e.g. DataNucleus, however it is not supporting db4objects any more [9]). Instead of actual JPA for the tests db4o interface is used directly, since JPA is only a standard defining the API and its implementation is not needed at this point. Separately Hibernate and direct JDBC SQL access is used for comparison needs. All software used in the test system is open source and available under the GPL/LGPL license.

Versant provides users with data replication system which is currently (newest version of db4o and dRS is 8.1) able to communicate with their own database and any SQL database through Hibernate. dRS system, though useful, have its limitations:

- requires identifier of type *long* in every table, which complicates relations adding: one have to change any foreign keys into required long values and modify the source database,
- it keeps additional information in its own tables (containing information about registered data providers, tracked objects and changes history − Fig. 3), so the legacy database will require further modifications,

- it needs so-called replication session active during loading of the source database to update its metadata, which negatively impacts on performance (refer to Fig. 5),
- Hibernate entities can be configured only via XML,
- unfortunately is does not work with Hibernate 4 and therefore the used version is 3.6.10.



#	id	is_my_sig	signature	created
1	1	1	<BLOB 31 bytes>	1356716850759

#	id	created	provider_id	class_name	typed_id	modified
1	1	44456898312699905	1	pl.cds.drs.entity.GeoName	1	44456898312699904
2	2	44456898312962048	1	pl.cds.drs.entity.GeoName	2	44456898312699904
3	3	44456898312962049	1	pl.cds.drs.entity.GeoName	3	44456898312699904
4	4	44456898312994816	1	pl.cds.drs.entity.GeoName	4	44456898312699904

Fig. 3. dRS metadata with example rows
Rys. 3. Metadane dRS z przykładowymi wierszami

Additionally, it is encouraged by the db4o creators to use indexes on all object's attributes being queried, however it requires using objects instead of primitive types in entity's fields [10]. On the other hand dRS have some positives. It is free of charge, available on Java and .NET platforms as is db4o, it works with Hibernate, and generally it is very easy to use.

For tests presented in this paper only one table and entity class will be used to reduce the negative impact of dRS's shortcomings. The test data was taken from GeoNames database[3].

[3] www.geonames.org

The data contained 53584 objects, representing geographical names for Poland. This data rarely change and therefore in the production environment the presented solution would be the most effective.

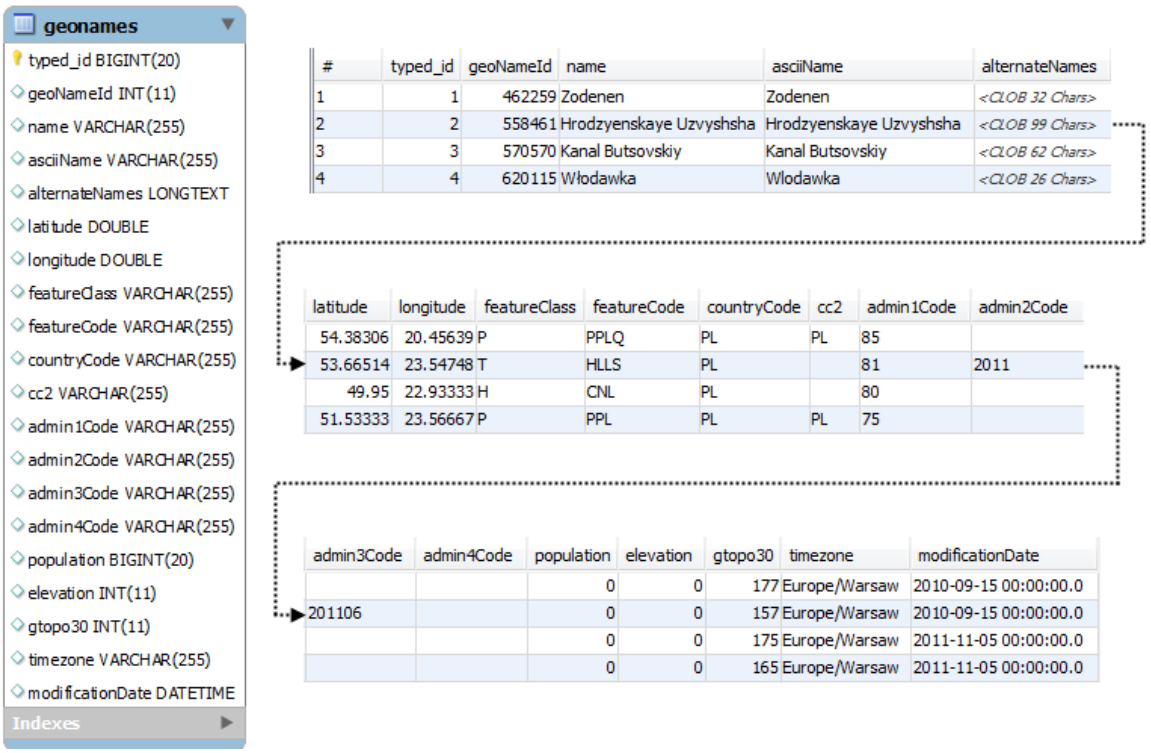


Fig. 4. The simple test database model and example rows
Rys. 4. Prosty model testowej bazy i przykładowe wiersze danych

Fig. 4 presents the test entity schema. The corresponding Java class with short description of every field is presented as following:

```
public class GeoName implements Serializable {

    // id required by dRS
    private Long typed_id;
    // integer id of record in geonames database
    private Integer geoNameId;
    // name of geographical point (UTF-8)
    private String name;
    // name of geographical point in plain ascii characters
    private String asciiName;
    // alternatenames, comma separated
    private String alternateNames;
    // latitude in decimal degrees (WGS-84)
    private Double latitude;
    // longitude in decimal degrees (WGS-84)
    private Double longitude;
    // group of the objects to which the object belongs to
    private String featureClass;
    // type of the map object
    private String featureCode;
    // ISO-3166 2-letter country code
    private String countryCode;
    // alternate country codes, comma separated, ISO-3166 2-letter country code
    private String cc2;
    // fipscode (subject to change to iso code)
    private String admin1Code;
```

```
    // code for the second administrative division
    private String admin2Code;
    // code for third level administrative division
    private String admin3Code;
    // code for fourth level administrative division
    private String admin4Code;
    // bigint (8 byte int)
    private Long population;
    // in meters
    private Integer elevation;
    // average elevation of 900m x 900m area in meters
    private Integer gtopo30;
    // the timezone id
    private String timezone;
    // date of last modification in yyyy-MM-dd format
    private Date modificationDate;
    // ...
}
```

Additional tables containing administrative codes descriptions were also available but ignored because of relation mapping issues mentioned above. The chosen entity class have the biggest amount of attributes useful for testing the selection operations performance.

All data stores were placed in a single machine environment to be able to ignore LAN/WAN network influence on the test results. The operations (queries) conducted on the data stores can be divided into following groups:

- load test − supplying the databases with the data,
- selection tests − querying the data with different conditions,
- update test − insert, update delete operations,
- retrieval tests − tests focusing on getting a single object by its identifier or all objects since some data stores offer special, usually optimized, procedures to access them,
- replication test − checking how much time is consumed by the synchronization process and how big impact it will have in the proposed solution.

4. Test Types and Results

The main goal of the test is to check and prove that object database is faster than an ORM cooperating with a relational database. Tests results contain already mentioned load, selection, update, retrieval and replication comparisons. Additionally retrieval and selection tests will be run against the MySQL JDBC connector. Following subchapters will describe in detail what and how is being compared and present the results.

4.1. Data Load

This test assumes empty data store to be filled with GeoNames objects. In theoretical working system it is assumed that source data is stored in the relational database but here

loading of the object database will also be tested. The loading of the source database have to fill the dRS metadata, so the further use of replication system will report any changes that have to be forwarded into the object cache. Fig. 5 presents comparison of load times in various configurations.

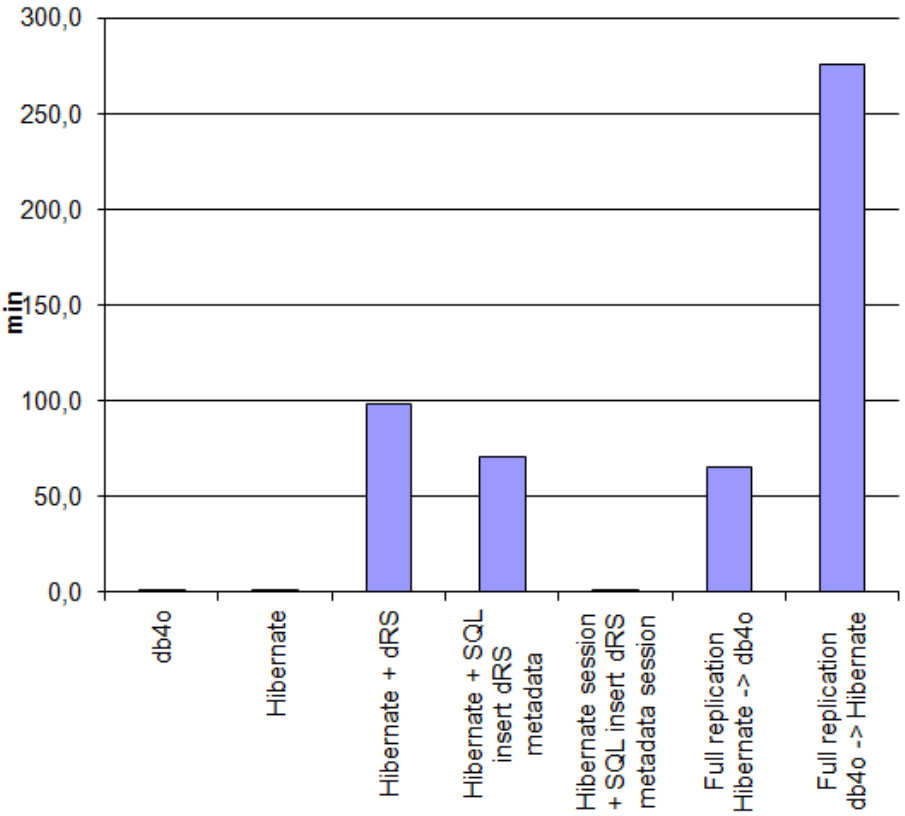


Fig. 5. Times of data load in various configurations
Rys. 5. Czasy ładowania danych w różnych konfiguracjach

Load of the relational database with active dRS system dramatically extends the time required to store the data (98 minutes 41 seconds). Inserting the metadata manually, knowing its structure reduces the time by about 30% (70 minutes 31 seconds) but this is still very long comparing to clean load of the GeoNames data. After the data is stored in the database (of course with the required *long* identifiers), one have to recreate the dRS metadata and loading them as a separate session takes very short time. Both phases took only 20 seconds to complete. Additionally the load time of db4o (which is joined with dRS in a way that metadata is see-through for the user) is even faster than simple load of the relational table through Hibernate (7 and 16 seconds respectively). What is more, db4o was creating indexes during loading, which slowed it down more or less two times.

In production environment it is assumed that the system starts with an relational database and the data is replicated to the object database(s) as needed. Therefore the object database does not have to have all the data stored. This next test checks how long it takes to replicate the full dataset. For a moment it is assumed that the object database is the source database.

This might be also corresponding to a massive update of the relational database. Full replication of the GeoNames for Poland from db4o to MySQL through Hibernate took 275 minutes and 50 seconds, which is almost 5 hours. On the other hand in the target solution, in which the object database is filled based on source database, the full replication was completed in 65 minutes 15 seconds. Please note, that this will be done only once.

4.2. Data Querying: Selection and Updates

For more reliable results all selection queries are conducted in two phases: so-called *cold* and *hot cache*. First, the cold phase is done for the DBMS to fill its buffers before the actual test. The hot phase corresponds to the database acting later, when it has achieved its optimal performance. The result of hot phase are the ones expected during the majority of queries in the application lifecycle. Both phases contain a 10 time repetition of a query and the average value is the actual result. The second phase result is the more probable one but first phase result are also presented.

Table 1 shows the select queries in SQL form. Each of the queries have other goal to achieve. Queries represent accordingly:

- retrieving all the data,
- a simple condition based on one parameter,
- a simple spatial query returning objects placed inside the given Minimal Bounding Rectangle [11],
- multiple conditions in one query as a example of more complicated query,
- a simple condition based on one parameter but with no matching results.

Table 1

Test select queries

No.	Name	SQL
1	QUERY_ALL	select * from geonames
2	QUERY_NAME	select * from geonames where name = 'Jurassic park'
3	QUERY_BBOX	select * from geonames where longitude <= 21.0 and longitude >= 20.0 and latitude <= 51.0 and latitude >= 50.0
4	QUERY_MULTIPLE	select * from geonames where name like 'A%' and population > 0
5	QUERY_NON_EXISTING_NAME	select * from geonames where name = 'Eiffel Tower'

These SQL queries can be easily translated info Hibernate HQL queries. In db4o there are three main query types: predicates, query by example and SODA [10]. Query by example is not applicable here, because example structure can provide only either a single value of an attribute to find by or no value at all. SODA is db4o's internal query interface and it is sug-

gested not to use it if not necessary [10]. The third query type are predicates which are simple methods returning logical *true* if a given object fulfils the query conditions. Predicates have limitations but are very versatile and easy to translate from SQL. Because of a bug in db4o, preventing from using indexes when logical alternative is used in the query condition [12], such operation was not included in the test.

In case of update operations no cold and hot cache phases are distinguished simply because every time a different object is updated. Operations are repeated 10 times, however every repetition have different parameters i.e. 10 random objects added, updated or deleted and average values for every operation are the final results. Only object identifiers are preserved between operations so that at the end both databases are in the same state as before the test. SQL access is this time was not taken into account.

Table 2

Queries (average execution times in milliseconds over 10 executions)

Query ID	db4o predicate		Hibernate HQL		SQL		Object count
	cold cache	hot cache	cold cache	hot cache	cold cache	hot cache	
ALL	50,0971	15,3033	490,2861	290,4702	531,3170	952,7941	53584
NAME	3,8136	1,0095	91,6430	91,9786	91,2974	92,4229	1
BBOX	126,7103	87,9322	104,5975	104,5928	108,5443	108,6018	1918
MULTIPLE	60,4562	49,2289	101,0946	101,0859	100,9114	100,5565	52
NON_EX	0,9205	0,6855	90,8131	90,7453	90,5254	90,6929	0
INSERT	143,0374		6,2444				10
UPDATE	97,7620		3,4613				10
DELETE	177,1562		1,7889				10

Results of select and update operations are presented in Table 2. The table provides also the affected object count. All corresponding select query results were equal. Although db4o is slower in terms of updates it has undeniable superiority over Hibernate in select operations. It is on average 50 times faster. The result is owed to extensive use of indexes in db4o. Unfortunately they slow down the update operations. Hibernate uses SQL in its core however it is able to use cache and provide performance boost over direct JDBC.

4.3. Data Retrieval by Identifiers

Reading object by its identifier is a very common operation. It does not receive any other conditions on the object so there is plenty of room for optimization. Both db4o and Hibernate have special methods of returning an object by ID.

In case of db4o beside the standard predicate-based queries a mechanism called *query by example* is available. In short it searches for objects matching the provided example filled with interesting values (in this case the identifier). Both predicates and example-based queries

are translated into internal SODA queries. db4o also provides user with lazy loaded result sets [10] based on Java collections. All types of queries were checked and as it turns out predicates were the slowest reaching an average time of 1.8 ms to execute. Query by example was an order of magnitude faster. Using SODA directly gave a little bit of acceleration (about 10-20 microseconds), but still half of the time the object was read from the result set. For comparison the query by example was chosen.

Table 3

Read single by ID (average execution times in milliseconds over 10 executions)

Object	db4o query by example		Hibernate get by ID		SQL	
	cold cache	hot cache	cold cache	hot cache	cold cache	hot cache
1	3,2992	0,4168	2,6487	0,0265	0,7729	0,7209
2	0,5639	0,4023	0,2842	0,0271	0,7066	0,6906
3	0,5286	0,4024	0,2754	0,0270	0,8073	0,5835
4	0,5649	0,3965	0,2559	0,0274	0,6679	0,6604
5	0,4825	0,4052	0,2626	0,0274	0,6333	0,6426
6	0,4794	0,4367	0,2808	0,0272	0,6185	0,6066
7	0,5025	0,4042	0,2805	0,0279	0,7013	0,6791
8	0,4420	0,3669	0,2589	0,0278	0,6447	0,5618
9	0,4141	0,3370	0,2305	0,0277	0,5779	0,5593
10	0,4144	0,3402	0,2500	0,0274	0,5654	0,5464
Average	0,7691	0,3908	0,5028	0,0273	0,6696	0,6251

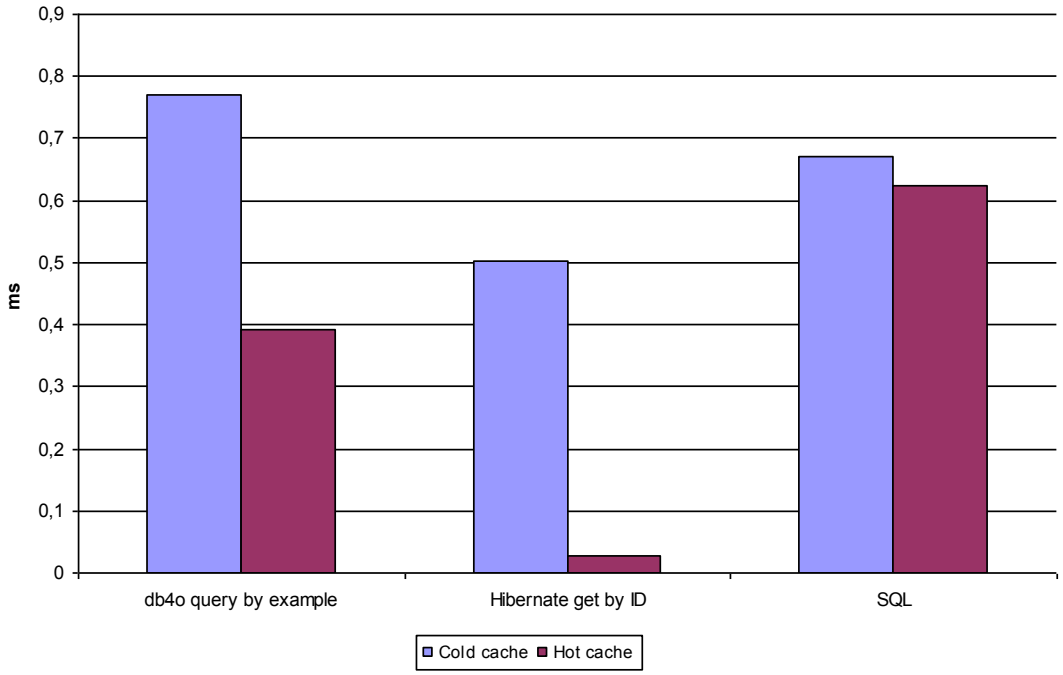


Fig. 6. Operation time comparison of reading a single object by ID

Rys. 6. Porównanie czasu operacji pobrania pojedynczego obiektu po identyfikatorze

Hibernate can simply return on abject based on its mapped identifier. In SQL a typical *where* phrase is processed. The test results are presented in Table 3 for 10 executions over 10

objects. As previously tests were conducted in cold and hot cache phases. Fig. 6 shows graphical comparison of the results which states that although Hibernate is very fast when caches are filled, using object database is faster than traditional SQL access and even beats Hibernate if it is initiated later than db4o (e.g. in some dispersed system with multiple types of clients to one database).

4.4. Full Read

Full read of the dataset was also tested with distinction into cold and hot cache phases. Here db4o provides a special way of getting the whole result by being able to query the data without any predicate. Although a large amount of data was returned every time, all access methods (even SQL access) achieved better results in the second phase as shown in Table 4 and Fig. 7. Hibernate again made use of its own caches, but db4objects was more than 10 times faster. Lazy loaded result sets are very helpful here. They do not require copying and in every test the session to a database is open throughout the test so the data is returned as needed.

Table 4

Read all (average execution times in milliseconds over 10 executions)

Phase	db4o no predicate	db4o predicate	Hibernate HQL	SQL
Cold cache	43,3834	24,0600	485,5906	535,1678
Hot cache	15,6460	13,1701	290,4488	444,9254

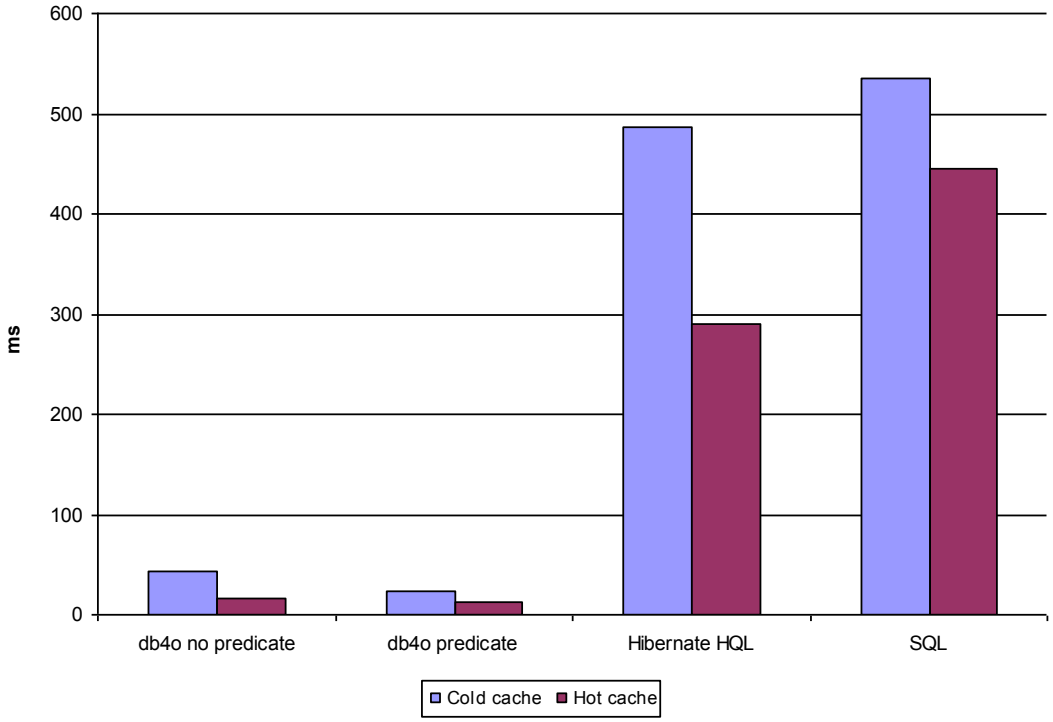


Fig. 7. Operation time comparison of reading all objects
Rys. 7. Porównanie czasu operacji pobrania wszystkich obiektów

The fact that providing db4o with a predicate always returning *true* gave performance increase over simply getting all the objects is a bit confusing. However results are not that different and may vary based on temporal system state and the fact that the database was initiated once and the test was performed in one JVM in order presented in the table.

4.5. Data Replication

Replication is considered as a two-way check of changes between the two data sources and synchronization of any updated objects. Until this time the update tests were conducted without the replication. To simplify the next test it is assumed that replication is done between any atomic operations. Now the whole update test is repeated multiple times but in between the operations a replication phase is conducted. First, replication is done before and after the whole test, then after updates on both databases and finally after every single update. Table 5 presents the replication run times and number of replicated changes in various states of both databases. Separately presented are times for both directions of replication and replication of deletions.

There is of course a correspondence in single direction replications − the more objects were modified the loner their replication will take compare to replication of the other database. Unfortunately in current version of dRS deletions have to be replicated separately and it takes a lot of time even if nothing was deleted (dRS checks both providers like in replication of other updates). As it turns out there are still bugs in deletions replication, especially with Hibernate [13]. There was also strange case of replicating one object less than was actually changed (this happened every time the test was conducted). The replication mechanism is still being developed and one can hope future versions will be more stable.

Table 5

Replication

Type	db40 -> Hibernate		Hibernate -> db4o		deletions (bidirectional)		Sum
	ms	changes	ms	changes	ms	changes	ms
Clean DBs	0,4156	0	4,0702	0	2,7257	0	7,2115
Both updates	0,5248	0	1,8958	0	1,6948	0	4,1154
db4o update	0,2188	0	1,4438	0	1,2866	0	2,9492
Hibernate update	0,2556	0	1,3104	0	1,1368	0	2,7029
db4o insert	163,9086	10	5,2421	0	50,8574	0	220,0081
db4o update	124,7600	10	2,4179	0	45,0489	0	172,2269
db4o delete	0,2449	0	3,8491	0	25,6885	0 (bug)	29,7826
Hibernate insert	0,4618	0	49,8423	10	62,4449	0	112,7490
Hibernate update	1,8819	0	35,0438	9 (bug)	39,3655	0	76,2912
Hibernate delete	0,2319	0	1,3785	0	34,7103	10	36,3206

Full replication (all updates bidirectional) times are comparable with executing a selection query (the "Sum" column). Therefore in the target system it can be assumed that during typical replication of new changes one query will run twice as long as normally, which is acceptable.

5. Conclusion

To summarize the conducted tests it has to be stated, that though update operations took more time in object database, selections were significantly faster. In typical application there are a lot of reads and much less updates. The exception is supplying of the data which is generally done during low network traffic. Surprisingly supplying of the object database is also faster and one can imagine synchronizing the new data to the relational database in the background and/or during the night.

Because of general acceleration in database responsiveness, there is plenty of time to conduct replication. Theoretically, if at least a bit less than the whole database idle time is spent on synchronization, the proposed system is still faster. In real system replication of large datasets will be rare. More common will be small (tens or hundreds of objects) updates done by users.

It has to be stated that object cache can be local for every client or global for the whole system. In first case one have to deal with replication changes made via one of the object database instances to the rest of them but accessing local database it faster. In second case, clients connect to the one object database instance via network but when any replication is needed we can use hot swap between two instances of object databases – operational and replicated. Source database can be updated directly by a legacy application in which case all object caches have to be updated in non-scheduled mode. Amount of updates might be significant and again hot swap is suggested. In every synchronization process there can occur conflicts. This is a typical problem and dRS has mechanisms for handling such situations.

Presented test system still has problems originating from db4o and dRS themselves. Limitations on ORM usage, unresolved bugs in the replication system may be the cause of choosing other products in the future. However these tools were easy enough to use and present the idea of caching through an object database. Of course the presented solution should be tested with another object database e.g. VOD or Neodatis. It is worth mentioning that DataNucleus (JPA and JDO[4] implementation for diverse data stores like RDBMS, NoSQL, XML, *xlxs* files, etc.) also offers data replication functionality [14]. Although is not automated it should

[4] Java Data Objects

be considered for future testing. Tests were conducted using slow changing data of moderate amount to tentatively apprize proposed solution usefulness. Further tests should encompass larger datasets with different changing characteristics and more complicated structure (related entities).

BIBLIOGRAPHY

1. Garcia-Molina H., Ullman J. D., Widom J.: Database System Implementation. Prentice Hall, 2000.
2. Ullman J. D., Widom J.: First Course in Database Systems, 3rd Edition. Prentice Hall, 2007.
3. Lausen G., Vossen G.: Models and Languages of Object-Oriented Databases. Addison-Wesley, 1997.
4. Kim W.: Introduction to Object-Oriented Databases. The MIT Press, 2008.
5. Bauer C., King G.: Hibernate in Action. Manning Publications, 2005.
6. Linwood J., Minterd.: Beginning Hibernate, Second Edition. Apress, 2010.
7. Płuciennik-Psota E., Płuciennik T.: Object Database-Based Optimization of Relational Database Access. Proceedings of the VIth International Conference on Computer Science and Information Technologies, Lviv, Ukraine 2011.
8. Versant db4o Object Database, http://www.versant.com/products/db4o-object-database [online, 2013-01-05].
9. DataNucleus, http://www.datanucleus.org/ [online, 2013-01-05].
10. db4o Reference, http://community.versant.com/documentation/reference/db4o-8.1/java/reference/ [online, 2013-01-05].
11. Longley P. A., Goodchild M. F., Maguire D. J., Rhind D. W.: Geographic Information Systems and Science. John Wiley & Sons Ltd, 2005.
12. db4o Issue Tracker, http://tracker.db4o.com/browse/COR-1409 [online, 2013-01-05].
13. db4o Issue Tracker, http://tracker.db4o.com/browse/DRS-106 [online, 2013-01-05].
14. DataNucleus AccessPlatform v.3.2 User Guide, http://www.datanucleus.org/products/accessplatform_3_2/datanucleus-accessplatform-docs.pdf [online, 2013-01-12].

Wpłynęło do Redakcji 16 stycznia 2013 r.

Omówienie

We współczesnym świecie większość nietrywialnych aplikacji współpracuje z bazą danych, przeważnie relacyjną. Ponieważ zwykle aplikacja jest obiektowa, mamy do czynienia ze współpracą dwóch światów opartych na zupełnie innych paradygmatach, co pociąga za sobą konieczność stosowania warstwy pośredniczącej w postaci narzędzi mapowania obiektowo-relacyjnego, tzw. ORM. Każda dodatkowa warstwa w aplikacji może oznaczać spowolnienie jej działania. Alternatywą dla takiego rozwiązania jest użycie obiektowej bazy danych. Tego typu bazy pozostają jednak w cieniu baz relacyjnych, które przez lata ugruntowały swoją pozycję. Trudno więc się spodziewać, że przedsiębiorstwo, które od lat stosuje relacyjną bazę danych, zaryzykuje przejście na bazę obiektową.

W artykule przedstawiono pierwsze testy rozwiązania, które pozwala wykorzystać obiektową bazę danych jako pamięć podręczną do intensywnej współpracy z obiektową aplikacją. Pamięć ta jest okresowo synchronizowana z relacyjną bazą danych. Częstotliwość synchronizacji zależy oczywiście od charakteru danych. Testy przeprowadzone na danych nazw geograficznych z użyciem obiektowej bazy db4o, biblioteki Hibernate oraz MySQL pokazały, że baza obiektowa może przyspieszyć działanie aplikacji klienckiej. Operacje selekcji, najczęściej dotyczące największej ilości danych, można było dzięki nowemu podejściu przyspieszyć średnio o kilkadziesiąt razy. Odbywa się to kosztem dłuższych operacji zapisu, aktualizacji i usuwania, które jednak nie są wykonywane aż tak często. Mając więc ogólnie szybszy system, można część zyskanego czasu przeznaczyć na wykonywanie synchronizacji obu baz danych. Pierwsze testy przeprowadzono dla umiarkowanej liczby danych o mało zmiennym charakterze, aby przekonać się, czy zaproponowane rozwiązanie przynosi korzyści i czy warto przeprowadzać dalsze testy, które powinny objąć większe zestawy danych o większej zmienności i bardziej złożonej strukturze.

Addresses

Ewa PŁUCIENNIK-PSOTA: Silesian University of Technology, Institute of Computer Science, Akademicka 16, 44-100 Gliwice, Poland, Ewa.Pluciennik-Psota@polsl.pl.
Tomasz PŁUCIENNIK: Silesian University of Technology, Institute of Computer Science, Akademicka 16, 44-100 Gliwice, Poland, Tomasz.Pluciennik@polsl.pl.