Using an In-Memory Database for Efficient Querying of Java Objects

David Moskowitz

Doctoral Student

Nova Southeastern University

Fall 2010

Abstract

This paper explores adding object query capability to the Java language through the use if an in-memory database. Java contains no native object query capability. Searching for specific objects, or relating two or more object collections, must be done in a manual and iterative manor, increasing the likelihood of an inefficient implementation. This manual approach contrasts the capabilities of relation database systems that support high level querying via SQL. This paper looks at providing the same query capabilities on Java objects through the use of an in-memory database. A prototype system is presented that provides transparent querying of programmatic Java objects. Related research on in-memory databases and querying of programmatic objects is discussed and suggestions for future research are proposed.

In their 2006 paper "Efficient Object Querying for Java", Willis, Pearce, and Noble proposed an extension to the Java language to enable high-level object query capabilities for the Java language. A prototype system that compiled SQL like queries into native Java code was developed. This system, called the Java Query Language (JQL) was implemented as a Java library that encapsulated several common database query techniques, such as nested loops and hash joins (Willis, Pearce, & Noble, 2006).

In a 2010 unpublished paper, Moskowitz criticized this approach, specifically JQL's reimplementation of common database query processing functionality. JQL indexes all programmatic objects at object creation time and performs queries against this index. This approach is similar to what a typical RDBMS provides, without the concern for efficient disk access. In other words, JQL essentially reproduced an in-memory database (Moskowitz, Research Critique of "Efficient Object Querying for Java", 2010).

While the motivation for JQL was correct and the prototype implementation sound, the effort involved in creating the necessary query processing capabilities was high. In the conclusion of the paper, Moskowitz proposed incorporating a true in-memory database to provide the same (and likely much more) query capabilities envisioned for JQL. This document will examine such a system.

The Problem of Querying Program Objects

Before we get into the specifics of querying Java objects, we first discuss the concept of querying of programmatic objects. Programmatic objects refer to any data structure created by an application at run time. In an object-oriented language like Java, we create objects instances and collections of object instances. Enabling query capabilities on these programmatic objects was the motivation for the development of JQL and is the motivation for the system presented in this paper.

Even though the implementation of JQL was in Java, the authors describe the lack of object query capabilities in most major programming languages. An example from the JQL paper and paraphrased in (Moskowitz, 2010) is the following:

```
Given a domain of Students and Faculty, find all Students who are
also Faculty
```

Such a query could be expressed in SQL as:

```
SELECT students.* FROM students, Faculty
WHERE students.name = faculty.name
```

The JQL authors argue that most programmers would implement such a search using a nested loop, such as in the following code example:

```
List<Tuple2<Faculty,Student>> matches = new ArrayList<..>();
for(Faculty f : allFaculty) {
      for(Student s : allStudents) {
            if(s.name.equals(f.name)) {
                  matches.add(new Tuple2<Faculty,Student>(f,s));
}}}
```

In most cases, a hash join approach, though slightly more complicated, would be much more efficient. Benchmark results, presented later in this document, confirm this assertion.

JQL proposed using a SQL-like syntax in Java source files. Using JQL, the above query could be expressed as:

```
matches = selectAll(Faculty f, Student s:
                    f.name.equals(s.name));
```

JQL, like SQL-based systems, hides the implementation details and determines the best execution plan for the given query.

The need to query programmatic objects is demonstrable, but support in programming languages has been low. Nothing of the sort currently exists for Java. Several other languages, particularly those that support closures, have limited query capabilities. An example of the type of query support some languages provide is the Filter operation from Groovy (Groovy JDK API Specification). This examples illustrates simple lists selection capabilities

```
assert [2,4] == [1,2,3,4].findAll { it % 2 == 0 }
```

In this example a closure, checking if modulo 2 of the list element is 0, is passed to the findAll method and applied to each element in the collection. This operation is analogous to the Selection operation of relation algebra, and could be expressed in SQL as:

```
select num from nums where num%2      =0
```

Another example is Groovy's version of the Cartesian Join, or Product from relational algebra.

```
assert [['a', 'b'],[1, 2, 3]].combinations()
     == [['a', 1], ['b', 1], ['a', 2], ['b', 2], ['a', 3],
        ['b', 3]]
```

In SQL, this would be written as:

```
Select nums1.num,nums2.num from nums1, nums2
```

We could simulate a Theta Join from relational algebra by chaining the results of the findAll method from the first example to the results of the Cartesian join. Such an approach to theta joins would be logically correct but without the possibility of optimization, such as the use of indexes. Therefore, performance is likely to be poor. It is also likely that Groovy is implementing the join using a simple nested loop, where a hash join would be more efficient. Such Groovy operators are not meant to take the place of database queries and rarely have the elegance of SQL

Even limited capabilities such as those described above are rare in most programming languages. One notable exception is the Microsoft .NET platform and its LINQ extensions (Box & Hejlsberg, 2007). A similar query to the one above example could be written in C# LINQ as:

```
List<String> words = new List<String> { "apple", "book" };
List<String> moreWords = new List<String> { "apple",
                           "book", "cat", "do" };

var query = from word in words
            join word in moreWords
             on word equals words. word
            select new { word = words.word};
```

The results will be the expected theta join:

```
{"apple" ,"book" }
```

This example shows how LINQ enables the use of SQL-like operators to programmatic objects. Similar syntax is applicable to relational database sources as well as non-relational sources such as XML. This was the approach envisioned for JQL, though JQL is hampered by the lack of closure support in Java[1]. At the time JQL was developed, Microsoft LINQ was just recently announced and not yet integrated into the .NET platform. Since then, this approach has become a standard way to access data from the .NET application.

LINQ takes a similar approach to Microsoft's original common data access technology, ODBC (Microsoft , 2010). As ODBC provided a relational SQL-like model on top of multiple sources, relational and non relational (Microsoft Excel for example), LINQ provides query capabilities to relational databases , and additional sources such as XML files, and, most pertinent to our discussion, programmatic objects. As was the case for ODBC, the same API is used no matter what the data source. Indeed LINQ can be seen as the latest generation of ODBC-like technologies. Though specific to the .NET platform, ODBC can be used from a variety of platforms and tools, including Java. The same is not true for LINQ, which is .NET specific.

[1] A good description of closures and their potential use in the Java language is (Gafter, 2007)

In many ways, the approach taken by Hibernate (Relational Persistence for Java and .NET, 2010) is similar the object query functionality we are trying to implement. Hibernate is the most popular of the class of object/relational mapping (ORM) tools for Java[2]. Hibernate shields the programmer from the specifics of the database layer, allowing him to work solely at the Java object level. A mapping is created during development that describes how Java objects map to database tables. Operations and queries executed against the object layer are translated to equivalent SQL statements and executed against the backing database.

Our SQL example from earlier:

```
SELECT students.* FROM students, Faculty
WHERE students.name = faculty.name
```

could be written using Hibernate Query Language (HQL) as:

```
SELECT s from Student s, Faculty f WHERE f.name = s.name
```

Note that the HQL and SQL queries are almost identical. The primary difference (though HQL supports additional features not available in SQL) is the use of class names and not database table names. In most cases, an actual database query will be executed and the Java objects loaded according to the defined mapping. Hibernate does maintain an in memory cache, but this is not a cache of the database, only a cache of loaded objects. A requested object may exist in and be retrievable from the cache, but for complex joins a database retrieval is usually needed. The main similarity between HQL and native object queries is that HQL queries are executed against programmatic objects, not against database tables. Tools such as ODBC, and the Java equivalent, JDBC, enable SQL statements against database tables.

While the programming model is similar, Hibernate focuses exclusively on managing *persistent* objects (objects that are read and/or written to disk) in a transparent manner. Though not the

[2] A .NET implementation is also available

equivalent of in memory query, this approach could conceivably be extended to work on both in-memory objects and persistent database storage. Such an approach would be analogous to LINQ, where the same query syntax is used for in memory and database objects. Given the widespread use of Hibernate, and new Java ORM technologies such as the Java Persistence API, a likely avenue to incorporate object querying into Java would be as an addition to one of these existing Java technologies.

In-Memory Databases

In this section, we explore in-memory databases. Such systems are generally used in place of, or in conjunction with, an on-disk database in order to improve performance. An in-memory database is not a replacement for object querying, in the same way an SQL database is not such a replacement. Programmatic objects will continue to hold the transitory data while the database will hold the persistent data. The use of an in-memory database may in fact be transparent to the application, as the application is only aware that it is using a database via a standard interface such as SQL.

An in-memory database (IMDB), also called a main memory database (MMDB) is a variant of a relational database management system where the entire database is resident in memory. This organization is in contrast to a standard relational database management system (RDBMS) which stores data in secondary storage, usually a hard disk.

In-memory databases were first proposed in the 1980s as researchers envisioned improvements to RDBMS systems due to the continually rising capacity and falling costs of computer memory. Exemplary of this thought is a 1986 paper by Lehman and Carey, who write:

> It is projected that memory chip densities will continue their current trend of doubling every year for the foreseeable future, and, as a result, it is expected that main memory

> sizes of a gigabyte or more will be feasible and perhaps even common within the next decade. (Lehman & Carey, 1986)

This predication turned out to be correct. Moore's law, the basis for the prediction, continues to hold today. Main memory of several gigabytes is now standard on even common desktop computers.

Much of the research and techniques proposed in the seminal articles of the 1980s and 1990s are still the state of the art to this day. Commercial systems first became available in the late 1990s as main memory costs dropped and capacities rose. The progress since that time has been made in the number of available IMDB systems. The choices include high-end commercial and low-end embedded and open source solutions.

Even with the increase in memory capacity, some databases will never fit entirely in memory. Large repositories such as data warehouses, or emerging technologies such as satellite image storage or Web indexing, will always need secondary storage. However, IMDBs have seen widespread use in areas where real time and high performance data access is needed. Examples of these fields include financial services (high volume transactions) and telecommunications (real-time switching).

In-memory databases are claimed to be 10 times faster than on-disk RDBMS. This often-repeated metric inspired the name of one of the earliest commercial entries into the IMDB field, Times Ten, who was recently purchased by Oracle (Oracle, 2005). The Times Ten software is marketed as a standalone IMDB as well as an in-memory caching layer for Oracle's flagship database.

In a 1992 paper, Garcia-Molina and Salem describe the early history of in-memory databases. Most of the systems described at that time were prototypes research projects in various levels of development. The sole commercial product, Fast Path, was a component of the hierarchical IBM IMS database. Fast Path was added into IMS in 1975 (Gawlick & Kinkade, 1985) to support higher performance processing through increases in memory storage of recently used data, referred to as "hot spots". The system supported both in memory and on disk data.

A key point mentioned in journal papers, and used frequently in IMDB product literature, is the distinction between an IMDB and an on-disk databases with a large cache. Even if the cache were large enough to hold a copy of the entire database, the systems would have fundamentally different implementations. On-disk databases are optimized to limit disk access. Index structures such as B-trees are commonly used for storage while buffers are used for data transfer between disk and memory. Running such an on disk database in a RAM drive, for example, will still incorporate disk-optimized structures and buffering techniques. While such caching will boost performance and should certainly be used when appropriate, systems that need to gain every last bit of performance will generally look towards, true in-memory optimized databases.

In-memory databases are tuned for CPU efficiency/speed and reduced memory footprint. Structures such as T-trees, first proposed in 1986 (Lehman & Carey) for IMDBs, are commonly employed. Recently, further optimized index structures have been developed to take advantage of speed improvements at the CPU memory cache level. These indexes are sometimes called Cache Conscious indexes and optimize performance based on the speed differential between cache and main memory. Rao and Ross (1999) observed that throughout the 1980s and 1990s, processor/cache performance has increased by 60% per year while DRAM/Main memory performance only increased by 10% per year. This metric, they claim, raised questions as to the conclusions of earlier researchers who proposed T-trees as optimal IMDB structures. Instead, they propose Cache-Sensitive Search Trees (CSS-trees) as a better solution. Later researchers proposed additional cache sensitive index structures, such as cache sensitive B-trees. Cache sensitive solutions attempt to minimize cache misses, thereby performing most lookups in faster cache memory. This scenario is analogous to earlier (and ongoing) techniques to minimize "misses" to main memory lookups and avoid disk access. The major difference is that disk sensitive operations are under the control of the DBMS, while cache sensitive performance is dictated by the computer hardware and therefore non-deterministic.

Such in-memory systems are gaining popularity as hardware capacity increases and costs decline. Several open source IMBDs are available. Such tools make excellent candidate for incorporation into JimQL, due to their price point (free) and open source license. Modern IMDBs are available that support SQL.

Also available are "no SQL" or key-value store databases. Key stores are used to cache data so it is available for processing later. Key-value pairs provide a generic data mode usable by many disparate system types. These systems can provide optimal performance for certain applications, as the entire data model is limited to an in-memory hash table.

We will use an SQL oriented database for JimQL, as our goal is to enable simple high level querying. Most no-SQL systems require programmatic access to data similar to hierarchical and network models of the past.

Embedded Databases

Embedded databases are designed for use by an application and only accessed through the application. This approach contracts non-embedded databases that can be access by multiple and heterogeneous client tools. Embedded databases can be disk or memory based. Most will persist data to disk (as most MMDBs do for durability) in order to achieve long-term storage of application information.

One of the earliest embedded databases was Btrieve, developed in 1982. This tool offered programmatic access to an ISAM structure and was popular through the mid 1990s. In 1992, Microsoft released Open Database Connectivity (ODBC) that provided a common programming interface to disparate databases. Most importantly, this tool opened up SQL database access to emerging client-server programming environments, such as Microsoft's own Visual Basic. Visual Basic 3.0 included support for Microsoft Access databases through the Jet library. Though not a true DBMS, Access supported SQL queries and a relational view of data. ODBC and SQL back-ends soon became the standard for client-server development. Many systems in need of an embedded Database used Access files controlled by Jet engine embedded visual basic applications. Java Database Connectivity (JDBC), released in 1997 as part of JDK 1.1, is the Java equivalent of ODBC and has been the standard method for data access on the Java platform through the mid 2000s. In the past decade, new data access methodologies like LINQ for Microsoft platforms and Enterprise Java Beans and Hibernate for the Java platforms have gained popularity. However, SQL based access through JDBC remains a commonly used data access approach.

Embedded databases usually have a small memory footprint. Many of these systems are geared towards embedded devices like set-top boxes or other hardware that may not have large memory capacities. Our approach to Java object querying assumes enough memory to store our needed objects. It is understood that whenever specific performance or memory requirements issues arise in development, lower level implementation routines may be faster and necessary.

Implementation

For the needs of our prototype Java object query processor, an in-memory embedded database is the most appropriate. Providing data access from outside the application is not necessary, nor is disk persistence. We also require SQL support to enable high-level queries. We will refer to our implementation as Java In-memory Query Language, or JimQL.

Design Goals

Like JQL, our primary requirement is to provide a high level SQL like query capability on programmatic Java objects. The user should be able to simply specify the objects included in the query and the select or join criteria to apply to those objects.

The two recommendations made for future research in the Moskowitz paper will be addressed as well. These recommendations are:

1. Use a Java compatible syntax
2. Incorporate a commercial or open source in memory database as the query processing engine

We also add the following requirement to our implementation

3. enable transparent plug in support for multiple IMDB implementations

This last requirement will allow easy comparison of multiple query engines and allow the incorporation of additional IMDBs that may meet specific needs.

The system should also maintain reasonable performance for simple queries and superior performance for complex queries. Ideally, the performance of our system will be much better than hand coded implementations, especially when query requirements are complex and the hand-coded implementation may not take advantage of available query optimization strategies.

Java Compatible Syntax

One problem with the JQL approach mentioned by Moskowitz is that the query syntax is not Java compatible and an additional query compilation step is needed. Such requirements makes JQL code difficult to integrate into a modern IDE.

Java syntax can be easily achieved by using String parameters to represent queries. This approach is used by similar technologies such as Object-Graph Navigation Language (OGNL) and Hibernate HQL. While using strings makes type checking more difficult[3], the flexibility of this approach makes it preferable.

[3] A separate type checker could be developed and perhaps integrated into an IDE or, as JQL, through an additional compile stage.

To gain Java compatibility we could modify the JQL query syntax from the earlier example to the following:

```
matches = JQL.selectAll("Faculty f, Student s",
            "f.name.equals(s.name)");
```

JimQL will use a slightly different syntax. The following is the JimQL representation of the above query:

```
matches = .query("f.name = s.name",f,s);
```

This method uses a String argument for the join condition and a varargs argument for a variable number of collection objects.

Incorporating an In-memory Database

Rather than implementing an in-memory query processor for Java objects, and recognizing the difficulty of such an approach, we instead choose to incorporate an existing embedded in-memory database. Of the available systems, we choose the following for initial implementation and testing:

1. HSQLDB (The HSQL Development Group)
2. H2 Database Engine (H2 Database Engine)
3. Apache Derby[4] (Apache Derby)

These systems are all open source and freely available and distributable. There are also several commercial implementations (ex. ExtremeDb (eXtremeDB Embedded Database In-Memory Database System), SolidDb (solidDb Product Family)) that could be used. Open source systems are generally preferable in a project such as ours, since the source code can be modified if necessary for integration purposes or to satisfy specific functionality requirements. Closed source projects are

[4] Derby in included in Java version 6 distribution as JavaDB

more appropriate when used as- is, and, given our design goal of easy plug in transparency, could be easily incorporated when appropriate.

The basic approach in JimQL can be described as follows

1. Load source Java objects into target IMDB
2. Execute a query against target IMDB
3. Return results as Java objects

Intuition would hold that step 1 would be the bottleneck, as this operation is not needed for most other data query techniques. We can verify this assumption and gauge the performance penalty from such an approach through benchmark testing. If Step 1 does prove to be the bottleneck, we can take steps to avoid reloading collections that do not need to be loaded. We can do this by

- reusing existing, already loaded, collections if they are identical
- not loading specific tuples if they already exist

Step 2, query execution, will be handled by the chosen In-memory database engine. Our requirement here is to formulate the query from the input parameters. As will be shown, this is a straightforward translation.

Consider step 3, retrieving the result. Step 2 works on a representation of the actual source objects; In this case, a copy of the objects loaded into the IMDB. Therefore, what is returned from the query may not be the actual objects, but only a copy or close facsimile. Two options exist to return actual source objects:

- create new instances of the original class using the returned objects
- store a reference to all loaded objects and retrieve that reference for each row in the result

We will use the later approach, as it satisfied our requirements at the expense of slightly greater memory usage. To implement this technique, we will create a hash table containing each original source object loaded into the IMDB. While we could use a unique hash function based on the object properties, the objects are not guaranteed to contain a unique key. The simplest approach is to create an incrementing counter and associate this counter with a specific source object. In effect, we are creating an incrementing primary key, similar to an Oracle sequence or an Identity field in other database systems. Such a technique proves to be relatively straightforward and provided a direct mapping between source object and database rows.

Querying with JimQL

To illustrate the object query capabilities of JimQL, we will use the following domain:

- State(id,stateName)
- City(id,cityname,stateId)
- Zipcode(id,cityId,zip)

As a first example, we examine joining cities to their corresponding zip codes.

In SQL, this query would be written as:

```
SELECT city.id,   cityname,  zip  FROM zipcode
INNER JOIN city ON (zipcode.city = city.id)
```

The corresponding JimQL:

```
JimQL  jimql = new JimQL (JimQL.DBTYPE.HSQLDB);
List<Map<String, Object>> list = jimql. loadAndQuery
                    ("zipcode.city = city.id", cities, zipcodes);
```

In the above example, we first create a JimQL object, (optionally) specifying the database implementation to use. We then execute the loadAndQuery method. This method first loads each collection specified (in this case, cities and zipcodes) into an in-memory representation, then executes the corresponding join/filter using the criteria specified. The syntax of the criteria is similar to OGNL (OpenSymphony) and other Java-bases expression languages and corresponds directly to the structure of the source objects.

Behind the scenes, this query will be translated into the following SQL:

```
SELECT City.*, Zipcode  .* FROM City, Zipcode
WHERE Zipcode  .city = City.id
```

As can be inferred, JimQL executes the Cartesian product on the included relations and applies the filter condition on the result (the equivalent of a Theta Join in relational algebra). Since we are using an IMDB to do the actual query processing and optimization, we expect this operation will be more efficient that a true Cartesian join, especially if indexes exist on the joined columns. In fact, the necessary indexes will be created during the database load step.

JimQL does not currently support queries on nested complex objects types. For instance, if a City has a property latLong of type LatLong, an object containing a Latitude and Longitude value, JimQL does not support a query such as "city.latLong.longitude = 65.443" [5]. However, we can simulate this capability by adding the appropriate getter method to the City class:

```
public double getLongitude(){}
     return latLong.getLongitude
}
```

Adding this additional getter method would force our load routine to create a column for longitude and this column would then be available for querying. This technique has the drawback of forcing retrieval of each longitude value, from another class, for each source object loaded. This getter could be an expensive operation such as database or network call. As a future enhancement, Java annotations could be used to include or exclude specific getter methods or provide other directives to JimQL.

[5] Supporting such nested objects would require loading multiple tables per source object and runs the risk of loading much more data than is needed for the actual query being processes. More research is needed to determine the feasibility of supporting this feature.

Loading the Data

The data load step of our approach incurs the most overhead. Other query approaches work with data in place, be it a true database or in-memory objects[6]. In order to take advantage of IMDB query capabilities, we must load the data into the database. In most IMDB implementations, the data is loaded from persistent storage on startup, or for a continually available system, on a recovery event. For programmatic objects, the data needs to be loaded dynamically at query time. To achieve this, we implement the following algorithm:

```
for each Java source objects
      create a table in the target IMDB;
      Create an identity column using an incrementing integer value;
      for each public getter method in the  source object
            create a column in the new table;
            if the getter method is included in the query criteria
                  create an index on that column[7];
            end;
      Store the source object in a hash table using the identify value
            as the hash key;
end;
```

Another possible data loading approach, not currently implemented, is to load *all* programmatic objects, or those specifically annotated, into the target IMDB. JQL uses this technique. Such an approach would eliminate any redundant reloading of data, as all data would be only loaded once upon object creation. This approach would incur a larger memory usage overhead and additional facilities would be necessary to keep objects and IMDB data in sync.

[6] This statement is not entirely accurate, as disk-based DBMSs must transfer persistent date to in-memory buffers for processing. However, this process can be optimized in any manner appropriate and the appearance of a single persistent data store is maintained. JimQL requires loading of the entire relations into memory.
[7] Creating indexes may be done during query execution if the requested criteria is not available at load time.

It should be noted that JimQL requires, and assumes, enough available memory to load full relations. This is a reasonable requirement, as all data loaded already exists as transitory Java objects. While we have not benchmarked the exact memory requirements, a reasonable estimate is that the memory required for a Java object will double.

Querying the Data

Once the data is loaded and if the criteria is correctly formed, we can execute the query using the SQL interface of the target DB. The basic form of the database query created JimQL is:

```
SELECT * FROM Table1,table2,...,TableN WHERE Criteria
```

The tables in the query are the tables created from the target java objects. The criteria in the query is used as the join clause passed into JimQL. No modifications to the criteria are made so this query must be valid SQL or a run time error will be thrown.

Transforming the Query Results

The query is executed using standard Java JDBC and the results made available as a JDBC Resultset. These results needs to be translated back into Java objects, ideally, the same object instances passed into the JimQL query, appropriately filtered and joined. This is accomplish by creating an index of the source objects as they are loaded into the IMDB. A hash table is used to store this index, with the primary key of the database row created as the hash key and the original source object as the hash value. For each returned row, we look up the original source object from the hash table index and add that object to our results List. As we may have multiple source objects, each returned row can include more than one object type. We handle this by returning a List of Map objects. This typing can be seen in the JimQL query syntax described earlier.

The following algorithm is used to transform database query results into Java objects:

```
Execute SQL query;
for each row in query results
    create a Map representing that row ;
    for each Java source object in the query
        get the value of the identity column corresponding to the
            source object from the query results;
        Lookup up the actual source object from the hash table
                created during Load;
        add the source object to  current row Map;
    end;
    add the current map to the result List;
end;
```

For our sample city-zipcode join, the following database query result:

City.id	Zipcode.id
1	10
2	5
2	6

will be transformed into a List of Maps:

List Index		
1	key:"City" Value: City-1	Key:"Zipcode" Value: Zipcode-10
2	key:"City" Value: City-2	Key:"Zipcode" Value: Zipcode-5
3	key:"City" Value: City-2	Key:"Zipcode" Value: Zipcode-6

In the result List, City-n and Zipcode-n are instances of our original target Java object.

Performance Benchmarks

Benchmark tests were done to gauge the performance of JimQL compared to other query processing methods. The tests use the City, Zipcode, and State relations described above. The sizes of the relations are:

• City:	29982 rows
• State:	51 rows
• Zipcode:	41986 rows

These relations are stored in a Java ArrayList or database table. We will refer to these as target relations in the discussion below, as tests may be applied to either ArrayLists or Tables.

All tests were performed on an AMD Athlon II X4 630 Processor 2.80 GHZ desktop with 6GB Ram running Windows7 Home Premium. All external database servers run on the local machine.

Tables 1 through 3 shows the test results and descriptions for the benchmarks performed. All results are measured in seconds. Not all tests are applicable to all methods. Algorithmic complexities of various tests and methods are included where applicable, except where the queries are handled directly by the DBMS. In such cases, and on a fully indexed relation, the performance will vary between constant and linear complexity. Algorithmic complexities only consider in-memory operations, as no disk retrievals are needed.

While these benchmarks are informal, they are intended to show the feasibility incorporating an in-memory database into Java query processing. Additional testing is needed to verify further explain the results recorded. Specifically, each needs to be broken down into their components to determine exactly where each technology accelerates or hinders performance.

Table 1

Benchmark Results

		Test					
	Method	**Find Id 500X**	**Find City Name**	**2-way Join**	**3-way Join**	**2-way Select**	**Load All**
Java	**Nested-loop**	0.28		56.78			
	Ordered Nested-loop			27.22			
	Hash Join			0.04			
Database	**MySQL**	0.24		0.38	0.46	0.02	
	MySQL - No Indexes	14.29		91.09	121.62	0.11	
	SQL Server - No Indexes	1.58		0.19	0.15	0.02	
IMDB	**H2**	4.42	0.27	1.27	1.11	0.67	0.42
	HSQL	4.28	0.21	0.84	0.69	0.48	0.24
	Derby	11.85	1.85	5.41	4.93	4.16	3.6
*on-disk Mode	**MySql***	1301.69	1412.61	3052	3036	3080	3126.4
	SqlServer*	10.73	12.69	31.48	23.94	26.03	27.04

Execution time in seconds

A big surprise is the performance differential between Apache Derby, a tool included in recent Java distributions, and the two other IMDBs. Derby is generally 3-5 times slower across all tests. While H2 is a close second, HSQLDB is clearly the fastest of the three IMDBs tested and would be the obvious choice for inclusion in our system.

The tests show the stark difference between native Java nested joins and hash joins. The nested join tests, simulating a naive search implementation, perform much worse than all of our JimQL tests. Also surprising is that our binary join on a non-indexed database performs worse than even our nested join Java implementation. Our test database, MySQL, apparently performs no additional optimizations. SQL Server, by contrast, performs well even when not indexed. SQL Server also performs well in our On-disk Simulation tests. It's performance is not as good as the true in memory databases, but is reasonable considering the table creation and loading that is taking place on the slower disk medium. SQL server has a (well-deserved) reputation for under the hood optimization. It is also possible that a commercial IMDB would have superior performance to the systems tested here.

Table 2

Benchmark Tests

Test	**Description**
Find ID 500X	Choose a random city id and look up the corresponding entry in the source object collection. Perform this operation 500 times. For IMDBs load the target data once We can describe the complexity of this and other test based on the size of the City and Zipcode relations (C and Z, respectively) Algorithmic complexity of this test varies based on the method used (as described in table 3). Best case = 500 for indexed collections Average case = 500C/2 , or O(N), for non-indexed, table scan approaches.
Find City Name	Return all cities where the cityname = 'Albany'. The same list of random cities is used for all methods in this test.
2-way Join	Join City to Zipcode on matching City ids
3-way Join	Join State to City to Zipcode on matching City and State ids
2-way Select	Same as 2-way Join, but filter the results where City.cityname = 'Albany'. This test is similar to the Find City Name test, but includes zip codes in the result. (By the way, there are 15 cities in the U.S. named Albany comprising 72 zip codes)
Load All	Load the City, Zipcode, and State relations into the database table. This test should measure the overhead that our load before query approach has over a system of query in place. Loading complexity is always proportional to the number of records loaded, or O (N). Further improvements could be made by performing all loads in parallel.

Table 3

Test Methods

Class	Method	Description/Notes
Java This class of tests includes Java operations on native data structures. These tests simulate the manner in which most Java based object querying is implemented in practice.	Nested-loop (JDK 1.6.20)	A linear or nested loop on one or more Java collections We can describe the complexity of this and other test based on the size of the City and Zipcode relations (C and Z, respectively) Unary relation : C/2 or O(n) Binary relation: CZ or O (n^2) Note: Without adequate indexes, database queries will have similar performance as our nested joins, as our no-index DMSB tests show.
	Ordered Nested-loop (JDK 1.6.20)	For a nested loop join, optimize join order so the inner loop can be aborted when a result is found. In joining City to Zip, if we use Zip as the outer loop, we can abort the inner city look when a single city is found. If City is the outer loop, we must continue checking each zip in the inner loop since more than one zip can be found for each City complexity: CZ/2 or $O(n^2)$
	Hash Join (JDK 1.6.20)	A hash join can be used when the join condition is an equality. This will provide the best (linear) performance. Complexity: C+Z or O(n) An example hash join implementation is given in the appendix.
Database - On Disk We use an on-disk MySQL database as a comparison to our in-memory approach.	MySQL(5.1.47)	Remember, we are not targeting our IMDB as a replacement for an ODDB, but as a replacement for low-level operations on programmatic Java objects. This method will not have the overhead incurred when we load the IMDB.

Non Indexed We include several tests against non-indexed databases, as those are similar to a non-indexed java collection.	MySQL (5.1.47) - no Indexes	In this test, the DBMS will be forced to perform full table scans for select and join operations (unless it performs independent optimizations).
	SQL Server (2008) - no indexes	
In-memory Database The actual JimSQL implementation	HSQL(2.0.1)	HSQL exhibits the best overall performance of the three JimQL implementations.
	H2 (1.3.146)	H2 is a close second to HSQL in most tests.
	Derby(10.6.2.1)	Derby is surprisingly slower in all tests than the other two IMDBs.
In-memory Database - On-disk Simulation We also test our two disk based DBMS, but use these as if they were in memory databases. This test measures the performance gain by using an IMDB compared to an on-disk DBMS. In most cases, it would not be feasible to load on-disk tables to perform such queries. For small relations, even nested loops may provide adequate performance. For larger relations, the load time would be a limiting factor.	MySQL (5.1.47)	MySQL exhibits extremely poor performance.
	SQL Server (2008)	SQL Server exhibits relatively good performance in our load tests. Overall, this is a very versatile DBMS.

Also surprising is that the in-memory loads times do not appear to incur the majority of test time. This bodes well for our approach, as loading is the major overhead compared to in place operations on databases. Further research is needed to determine specifically where any bottlenecks occur within each test.

In summary, the performance of the in-memory databases are commendable and sufficient for our approach to be of value within normal application development.

Future Research

This study has proven the feasibility and utility of using an in-memory database to efficiently query native of Java objects. Several immediate enhancements could be made to the software to provide incremental improvement. The most obvious improvement to be made is in parallel loading of target relations. Since load operations are independent of one another, we could load all these into our IMDB in parallel. Further determination would need to be made as to the optimal concurrency level. This optimal concurrency load factor would likely be at least as high as the number of tables involved in the query, so all tables could be loaded simultaneously.

The development of the JimQL prototype did not focused largely on performance. However, performance issues are always an important concern and perhaps a limiting factor for widespread interest or adoption in any similar technology. Our target database can likely be tuned. For some cases, it might be possible to bypass the target database altogether. For small relations and simple criteria, JimQL could implement in place queries on Java objects, using nested loops or hash joins, and avoid loading the IMDB back end. Bypassing the IMDB should remain limited, as we risk gradually building a full IMDB. Avoiding building such a system was the primary motivation for JimQL.

Additional SQL features, such as group-by capabilities, could also be supported. The group-by operation could be done by the database engine or by JimQL while processing the result set. Optimization analysis will determine which approach is best. As before, we should lean towards letting the IMDB to the actual query processing.

JimQL is also a likely candidate for an open source project. Posting on an open source repository, such as Google Code, would help gauge interest in this approach and perhaps solicit input and additional participation.

While our approach using JimQL enables querying of Java objects, we are using SQL, which lacks some of the functionality available in imperative languages like Java. Research is already underway to include closures in Java (Gafter, Closures (Lambda Expressions) for the Java Programming Language). Such a feature would allow native Java querying on collections and make a LINQ-like implementation for Java possible. Query capabilities like that offered by JimQL could be incorporated directly into the Java language itself. Derby (or something better performing) querying could also be enabled using the approach we have outlined. Having both closure and SQL-based query capabilities would allow developers to choose the appropriate query language based on the specific developments and would provide Java an edge over .NET and other languages and technologies in this area.

Conclusion

This paper has shown how Java objects can be queried by incorporating an in-memory database to provide the query processing capabilities. A prototype implementation, JimQL, was introduced. Benchmark tests were performed and presented. These tests show the capabilities and performances of the prototype system are sufficient and can for the basis for future development and research, or even incorporated into the Java language.

Appendix A - Sample JimQL Code

The following code illustrates the use of the JimQL library for selected benchmark tests described above. Non-JimQL search routines are included in the Appendix B.

Find City Name

```
Jimql jimql = new Jimql(Jimql.DBTYPE.HSQLDB);

List<Map<String, Object>> list = jimql.loadAndQuery("city.cityName =
                                      'Albany'", cities);
```

3-way Join

```
Jimql jimql = new Jimql(Jimql.DBTYPE.HSQLDB);

List<Map<String, Object>> list = jimql.loadAndQuery(
           "city.stateId = state.id and zipcode.city  = city.id",
           cities, states, zipcodes);
```

2-way Select

```
Jimql jimql = new Jimql(Jimql.DBTYPE.HSQLDB);

List<Map<String, Object>> list = jimql.loadAndQuery(
           "zipcode.city = city.id and city.cityName = 'Albany'",
           cities, zipcodes);
```

Load Data

```
Jimql jimql = new Jimql(Jimql.DBTYPE.HSQLDB);

jimql.loadData(cities, states, zipcodes);
```

Appendix B - Native Java Search Implementation

Nested Join

```
List<City> citiesAndZips = new ArrayList<City>();

for (City city : cities) {
    for (Zipcode zipcode : zipcodes) {
        if (zipcode.getCity().equals(city.getId())) {
                City cityAndZip = new City();
                cityAndZip.setId(city.getId());
                cityAndZip.setCityName(city.getCityName());
                cityAndZip.setZip(zipcode.getZip());
                citiesAndZips.add(cityAndZip);
         }
     }
}

assertTrue(citiesAndZips.size() == zipcodes.size());
```

Nested Join with optimal order

```
List<City> citiesAndZips = new ArrayList<City>();

for (Zipcode zipcode : zipcodes) {
    for (City city : cities) {
           if (zipcode.getCity().equals(city.getId())) {
                City cityAndZip = new City();
                cityAndZip.setId(city.getId());
                cityAndZip.setCityName(city.getCityName());
                cityAndZip.setZip(zipcode.getZip());
                citiesAndZips.add(cityAndZip);
                break;
            }
     }
 }

 assertTrue(citiesAndZips.size() == zipcodes.size());
```

Hash Join

```
List<City> citiesAndZips = new ArrayList<City>();

Map<Integer, City> cityMap = new HashMap<Integer, City>();

for (City city : cities) {
    cityMap.put(city.getId(), city);
}

for (Zipcode zipcode : zipcodes) {
     City city = cityMap.get(zipcode.getCity());
     City cityAndZip = new City();
     cityAndZip.setId(city.getId());
     cityAndZip.setCityName(city.getCityName());
     cityAndZip.setZip(zipcode.getZip());
     citiesAndZips.add(cityAndZip);
 }

 assertTrue(citiesAndZips.size() == zipcodes.size());
```

References

Apache Derby. (n.d.). Retrieved November 25, 2010, from http://db.apache.org/derby/

Box, D., & Hejlsberg, A. (2007, February). *LINQ: .NET Language-Integrated Query*. Retrieved November 13, 2010, from MSDN: http://msdn.microsoft.com/library/bb308959.aspx

eXtremeDB Embedded Database In-Memory Database System. (n.d.). Retrieved from http://www.mcobject.com/standardedition.shtml

Gafter, N. (2007, January 28). *A Definition of Closures*. Retrieved 28 2010, November, from Neal Gafter's blog: http://gafter.blogspot.com/2007/01/definition-of-closures.html

Gafter, N. (n.d.). *Closures (Lambda Expressions) for the Java Programming Language*. Retrieved November 28, 2010, from Javac.info: http://javac.info/

Gawlick, D., & Kinkade, D. (1985). Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin , 8* (2), 3-10.

Groovy JDK API Specification. (n.d.). Retrieved November 20, 2010, from groovy.codehaus.org: http://groovy.codehaus.org/groovy-jdk/

H2 Database Engine. (n.d.). Retrieved November 25, 2010, from http://www.h2database.com

Lehman, T. J., & Carey, M. J. (1986). A Study of Index Structures for Main Memory Database Management Systems. *Proceedings of the 12th International Conference on Very Large Data Bases*, (pp. 294-303).

Microsoft . (2010, July 2). *ODBC--Open Database Connectivity Overview*. Retrieved November 28, 2010, from Microsoft.com: http://support.microsoft.com/kb/110093

Moskowitz, D. (2010). Research Critique of "Efficient Object Querying for Java". Unpublished paper.

OpenSymphony. (n.d.). *OGNL*. Retrieved November 27, 2010, from opensymphony.com: http://www.opensymphony.com/ognl/

Oracle. (2005, June 20). *Oracle Completes Acquisition Of TimesTen* . Retrieved Novembe 17, 2010, from Oracle.com: http://www.oracle.com/corporate/press/2005_jun/ttcomplete.html

Rao, J., & Ross, K. A. (1999). Cache Conscious Indexing for Decision-Support in Main Memory. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)* (pp. 78-89). San Francisco, CA: Morgan Kaufmann Publishers Inc.

Relational Persistence for Java and .NET. (2010). Retrieved November 15, 2010, from Hibernate Web Site: http://www.hibernate.org/

solidDb Product Family. (n.d.). Retrieved from IBM.com: http://www-01.ibm.com/software/data/soliddb/

The HSQL Development Group. (n.d.). *HSQLDB*. Retrieved November 13, 2010, from hsqldb.org: http://hsqldb.org/

Willis, D., Pearce, D. J., & Noble, J. (2006). Efficient Object Querying for Java. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* , 28-49.