Ask and You Shall Receive: Empirically Evaluating Declarative Approaches to Finding Data in Unstructured Heaps

William F. Jones and Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

Abstract

This paper reports on experience with the engineering and empirical evaluation of data management software that stores objects in collections like the `ArrayList` or `Vector`. While many programs may retrieve an object from a collection by iteratively evaluating each object according to a set of condition(s), this imperative retrieval process becomes more challenging and error-prone as it applies many complex criteria to find the matching objects in multiple collections. Query languages for unstructured Java virtual machine (JVM) heaps present an alternative to the imperative approach for finding the matching objects. Using a benchmarking framework that measures the performance of declarative approaches to identifying certain objects in the JVM heap, this paper empirically evaluates two query languages, JQL and JoSQL. Both the experiences and the experimental results reveal trade-offs in the performance and overall viability of the query languages and the imperative approaches.

1 Introduction

Programming in an object-oriented language (e.g., Java or C++) often involves the instantiation of objects and the invocation of methods in order to perform a desired operation. A Java program normally uses a collection, such as an `ArrayList` or a `Vector`, in order to manage the many objects that it allocates to the Java virtual machine (JVM) heap while executing. One approach to finding an object in a collection involves the iterative evaluation of each object according to an established set of condition(s). For instance, a Java-based Web browser may use a `Vector` of `Site` objects that stores data about all of the Web sites that a user is currently browsing. When a user disconnects from a specific Web site, the browser must traverse the `Vector` of `Site`s and remove the object associated with the recently closed Web page. This *imperative* approach to object retrieval is straightforward and easy to implement when the Java application uses a small number of collections and objects. Yet, as the number of collections and objects increases, imperative programming may lead to applications that are complicated, error-prone, and hard to maintain [11].

Instead of implementing the operations that are needed to identify the matching objects, the *declarative* approach to data manipulation specifies the requirements that the desired objects must meet. Declarative techniques, such as those found in a relational database management system (RDBMS), are commonly coupled with modules that perform query execution and optimization [9]. The query executor uses a specification for the required objects in order to automatically construct a plan for finding the desired data points. Since the RDBMS provides direct support only for relational data, it cannot transparently manage queries concerning the objects that a program stores in the heap of a JVM.

However, the Structured Query Language for Java Objects (JoSQL) [1] and Java Query Language (JQL) [10] furnish a declarative alternative to finding objects in heap-resident Java collections. JoSQL uses Java's reflection mechanism [2] during the execution of queries that are specified in a variant of the traditional structured query language [1]. JQL utilizes aspect-oriented programming (AOP) through AspectJ [6] and a unique querying syntax in order to support object queries. Even though there is a wealth of information concerning the efficiency of declarative querying in relational databases (e.g., [4, 8]), there is a relative dearth of performance data for object-oriented programs that query unstructured heaps. In order to close this knowledge gap, this paper presents a benchmarking framework that evaluates the performance of declarative approaches to querying JVM heaps. Leveraging our experience with this framework, we identify fundamental trade-offs in the response time characteristics of JQL, JoSQL, and several imperative methods. In summary, the main contributions of this paper are:

1. A benchmarking framework that measures the performance of declarative approaches to finding data in unstructured heaps (Sections 2 and 3).
2. Empirical evaluation of several factors (Section 4):
    (a) The performance trade-offs associated with

an aspect-oriented (JQL) and a reflection-based (JoSQL) approach to object querying.

(b) The impact that the type of object and collection have on the performance of both the declarative and imperative query executors.

(c) The viability of JQL and JoSQL when compared to several imperative methods.

2 Query Language Overview

In order to ensure that the paper is self-contained, this section provides a brief overview of JQL and JoSQL. For more details about these Java query languages, please refer to [1, 10]. Figure 1 describes the process associated with using JQL to perform a query in a Java program. Currently, JQL supports queries concerning objects that reside in any type of container that extends the Java class `java.util.Collection`. JQL also handles queries for objects that are instances of both the Java standard library and user-defined classes. JQL's caching mechanism stores matching objects and subsequently monitors the changes to these objects in order to efficiently process repeated queries. Since JoSQL does not furnish caching functionality, we reserve the evaluation of this JQL feature for future work. Finally, JQL queries can span multiple collections that vary according to their type and contents.

As depicted in Figure 1, a programmer must initially implement a Java program that contains specialized JQL code bodies (i.e., the JQL file). The JQL compiler transforms the JQL code, as shown in code segment one, into Java source code that conforms to the syntax and semantics of the current Java language. Using a standard Java compiler, the programmer transforms the JQL-enhanced source code into normal Java source code. Compiling and executing these bytecodes with a JVM causes the program to pass the JQL queries to a query executor that automatically develops and runs a query plan designed to return the matching objects. JQL programs contain AspectJ advice that tracks the allocation of objects to the JVM heap. The JQL runtime system uses a linear processing tree [7] during query evaluation and it supports both the nested-loop and hash joins [10].

(1) *Find all Integers with a value greater than ten (JQL):*

```
List queryResults = selectAll(Integer i: list
                              | i > 10 );
```

Figure 2 demonstrates the procedure for executing a JoSQL query in a Java program. Unlike JQL, JoSQL limits queries to objects that exist in a single collection. Similar to JQL, JoSQL handles queries for any Java object that resides in a container that extends `java.util.List`, a sub-interface of `java.util.Collection`. JoSQL also provides support for many standard SQL features such as aggregation, ordering clauses, and nested queries [1]. Yet, JoSQL does not offer complete SQL functionality since it lacks the join and union operators.

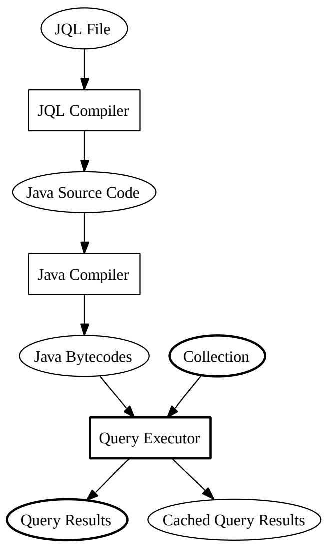

Figure 1: Query Execution with JQL.

As shown in Figure 2 and code segment two, a Java program using JoSQL expresses a query as a String containing `SELECT` statements. JoSQL's reliance upon Strings eliminates the pre-processing step that JQL requires. After constructing a new `Query`, a JoSQL program uses this object to parse the String and run the query. Employing Java's reflection mechanism to access the specified fields of each object, an `execute` method scans the objects in the heap-resident `list`. Finally, `queryResults` holds the object(s) that meet the conditions specified in the query.

(2) *Find all Integers with a value greater than ten (JoSQL):*

```
String sql= "SELECT *
             FROM java.lang.Integer
             WHERE intValue > 10";
Query q = new Query();
q.parse(sql);
List queryResults = q.execute(list);
```

3 Experiment Goals and Design

Benchmarking Framework. As illustrated in Figure 3, the execution of a benchmark starts with the input of a configuration to the Random Collection Generator (RCG) and Benchmark Initializer (BI). The user-defined configuration data includes details about the type of object, collection, and benchmark. The RCG uses `java.util.Random` to construct Strings, Integers, Graphs, and other abstract data types. For

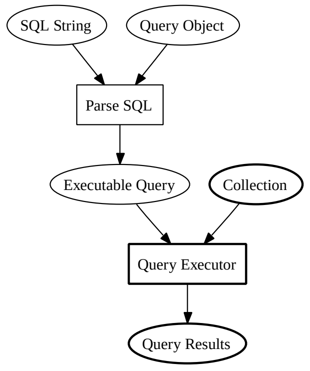

Figure 2: Query Execution with JoSQL.

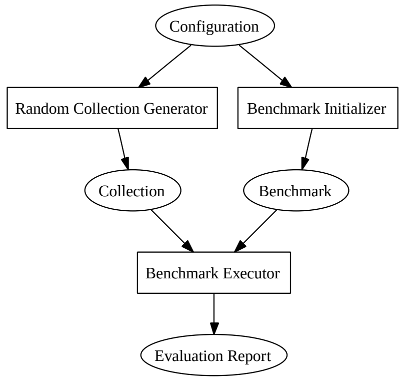

Figure 3: Benchmarking Framework.

instance, RCG successively selects and concatenates single characters to a String until it reaches the desired size. Following the design in [5], RCG can also build an undirected Graph object $G = \langle V, E \rangle$ such that each $v_i \in V$ is randomly connected to between one and three unique vertices. Each graph G contains two collections that store the vertices and edges while a vertex includes a collection for its incident edges. RCG randomly populates an `ArrayList`, `Vector`, or `LinkedList` since each of these collection types extends `java.util.List` and is thus compatible with both JQL and JoSQL. Finally, the BI configures a benchmark object so that the Benchmark Executor (BE) will run the experiment, collect and store the chosen evaluation metrics (e.g., response time), and statistically analyze the resulting data sets.

For each of the five benchmarks in Table 1, the collection size parameter stands for the number of unique objects that the framework stores in the collection. The Query benchmark selects either the Integers or Strings in the list that respectively match a specified Integer or contain a chosen character. For this benchmark, the object size corresponds to either the maximum value of the Integers or the total number of characters within each String. Figures 5 and 6 summarize the results from running Query. The Join benchmark joins two input lists containing randomly generated Integers and Strings in order to identify the Integers whose values match the length of the Strings. The object size for this benchmark represents the maximum size of both the Integer value and String length. We also hand coded an implementation of the Join benchmark, called HC-HJ, that uses a hash join to find the matching Integers and Strings. Figure 7 furnishes the outcomes from executing Join.

The Sub-Query benchmark searches a collection of undirected graphs for vertices that store a specified String. In this context, object size represents both the number of vertices within the graph and the maximum size of the Strings contained in each vertex. Figure 8 gives the empirical results created by the Sub-Query benchmark. Finally, we designed the Aggregate and Ordering benchmarks to evaluate the performance of the aggregation and order-by operators within the two query languages. Due to space constraints, this paper does not include the response time results from these benchmarks since they did not reveal experimental trends not already identified by the first three benchmarks. Similarly, we do not report the results from executing the Query benchmark with the small and medium object sizes. Nevertheless, the framework includes a complete implementation of each benchmark and we have successfully executed the benchmarks in every configuration stated in Table 1.

During the implementation of the framework, it became clear that differences in the syntax and functionality of JQL and JoSQL would make it difficult to fully compare these methods. For instance, while JQL includes several join operators, JoSQL does not furnish similar functionality. In support of a fair empirical comparison, we integrated standard join methods into JoSQL and enhanced JQL with ordering and sub-querying functions. We also implemented several hand-coded techniques, denoted HC, that use `for` loops to iteratively search the collections.

Goals and Metrics. The fact that JQL constructs queries at compile-time while JoSQL exclusively operates at run-time suggests that (i) JQL is faster than JoSQL and (ii) JoSQL is more flexible than JQL. As such, we designed the experiments to determine which querying technique exhibited the lowest response time as we varied the object size, object type, collection size, collection type, and various other factors. Specifically, we wanted to observe how response time changed as we systematically altered the (i) characteristics of the query, (ii) quantity and size of the objects, and (iii) type of collection. We also compared the declarative approaches (i.e., JQL and JoSQL) to the imperative methods (i.e., HC and HC-HJ).

The performance metric for this study is the time in milliseconds it takes to perform a single query operation on the input collections(s)

Name	Operation	Object(s)	Object Size			Collection Size		
			Small	Medium	Large	Small	Medium	Large
Query	Simple queries for objects in one list	String	10	50	100	5,000	50,000	500,000
		Integer	100	1,000	10,000	10,000	100,000	1,000,000
Join	Query/Join objects in multiple lists	String/Integer	10	50	100	500	1,500	3,000
		Integer	100	1,000	10,000	500	1,000	1,500
Sub-Query	Query object sub-collections in one list	Graph	10	50	100	500	1,000	2,000
Aggregate	Perform aggregates on one list	Integer	1,000	10,000	100,000	100,000	1,000,000	10,000,000
Ordering	Order objects by attributes in one list	Integer	1,000	10,000	100,000	1,000	10,000	100,000

Table 1: Overview of the Performance Evaluation Benchmarks.

and the framework measures response time with methods from the standard Java library (e.g., `System.currentTimeMillis()`). We compared the response time characteristics of (i) JQL 0.3.1 coupled with ANTLR 2.2.7 and AspectJ 1.5 and (ii) JoSQL 1.8. We conducted all of the experiments on identical Pentium 4 2.8GHz workstations with 1 GB of RAM. The execution environment of each workstation includes the (i) GNU/Linux operating system with a 2.6.22.9-91.fc7 kernel, (ii) Java 1.6.0 compiler, and (iii) Java 1.6.0 virtual machine in HotSpot^{TM}client mode.

Data Analysis Techniques. The bar charts in this paper depict the average time across twenty runs of the specific benchmark. Within each bar grouping, we organized the individual bars to report the response time measurements in the order JQL, hand-coded technique, and JoSQL. In Figures 5 through 8, the error bars at the top of a bar represent the 95% confidence interval (CI) for the arithmetic mean. The framework calculates the CIs using the default configuration of the Student's t-test implemented in the R language for statistical computation [3]. For each response time mean in the confidence interval $[l, u]$, we can be 95% certain that the mean of the response time values from subsequent experiments will fall between the lower bound l and the upper bound u. Therefore, small confidence intervals suggest that our benchmarking framework generates empirical outcomes in a repeatable and predictable manner. In fact, many of the resulting confidence intervals are so tight that they are not clearly visible in Figures 5 through 8.

As evidenced by Figure 4, this paper also employs regression tree models to describe the trends in the data sets. In particular, we use a recursive partitioning algorithm [3] to determine how the explanatory variables (e.g., query method, collection type, collection size, and object size) impact the response time metric. We selected this type of simple hierarchical model because it furnishes a clear view of the interactions between the explanatory variables. The root of a regression tree corresponds to the most important explanatory variable for a data set. By following a path from the root to a leaf node, it is possible to determine the mean value for the specified subset of the data.

In the split points of the regression trees (e.g., "Method:HC,JQL" in the first model of Figure 4), the word before the colon is a categorical explanatory variable and the word(s) to the right are the descriptors associated with the left sub-tree. Splits may also partition numerical variables with comparison operators. Moreover, the right sub-tree always corresponds to the data values that are not included in the split's descriptor. For instance, the root of the first tree model indicates that the left sub-tree describes the response time of HC and JQL while the right sub-tree gives that of JoSQL. The first tree in Figure 4 also reveals that HC and JQL have an average response time of 38.65 ms for the Integer-based Query benchmark.

4 Experimental Results

Query Benchmark. The examination of the first tree model in Figure 4 and the bar chart in Figure 5 shows that JQL's performance is comparable to HC and as such it is substantially faster than JoSQL. The tree model also reveals that when the Query benchmark uses either an `ArrayList` or a `Vector`, JoSQL's response time is an order of magnitude higher than the average response time of HC and JQL (i.e., 38.65 ms versus 309.40 ms). If we replace JQL with JoSQL in the `ArrayList`-based Query benchmark, then calculations with the results in Figures 5 and 6 indicate that, at the largest collection size, the response time increases by 874.8% and 180.1%, respectively. Overall, the tree models and the bar charts in Figures 5 and 6 confirm the same trend: JQL is faster than JoSQL. We attribute JoSQL's poor performance to the fact that it must process query Strings and use Java's expensive reflection mechanism [2] to access the collections during query execution. In contrast, JQL's pre-processing step avoids the costly operations that

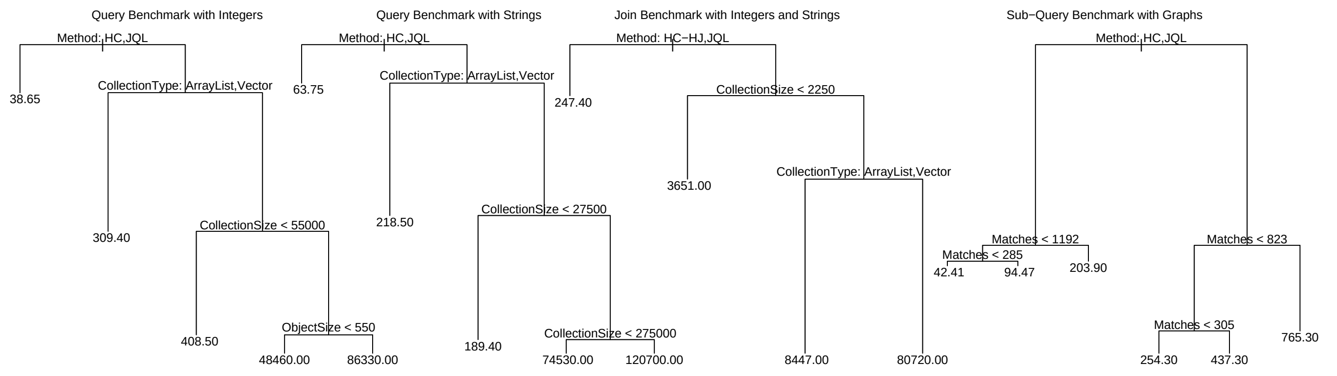

Figure 4: Regression Tree Models for the Response Time Metric.

are fundamental to JoSQL. JoSQL's reliance upon reflection also leads to high initialization overheads that further increase the response times. For instance, running the Query benchmark with JoSQL on an empty collection takes an average of 90 ms while JQL incurs an overhead of less than 1 ms for the same operation.

In the context of the tree model for the Query Benchmark with Integers, the leaves contained in JoSQL's right sub-tree have high values that range from 408.50 to 86330.00 ms. Therefore, it is evident that the `LinkedList` causes JoSQL to incur very high time overheads (we find a similar trend in the tree for the String-based Query benchmark). When JoSQL uses reflection to retrieve the data in an `ArrayList` or `Vector`, it gains access to the objects after one invocation of a reflective operation. In contrast, JoSQL must perform reflection on each vertex within the `LinkedList` in order to retrieve all of the data objects. Finally, the first two tree models and Figures 5 and 6 expose the fact that JQL and HC exhibit higher response times when Query uses Strings instead of Integers. We attribute this result to the fact that the String containment check with `contains(CharSequence s)` is more expensive than equality testing for Integers. In comparison to the Integer-based benchmark, a Query with Strings only has lower response times because we configured it to use collections that are half the size of those used in the Integer version (see Table 1 for more details).

Join Benchmark. As explained in Section 3, the Join benchmark scans the two input collections in order to determine which Integers have values that match the length of the Strings. When a match is found, this benchmark creates a temporary array of size two, stores the matching String and Integer in the array, and places the array in a collection housing the final results. The first, second, and third tree models in Figure 4 also demonstrate that, for both HC-HJ and JQL, Join is more computationally intensive than Query (i.e., 247.40 ms versus 63.75 ms). As anticipated, the Join benchmark's tree model also confirms that JoSQL's response time values are always one or two orders of magnitude higher than the arithmetic mean for either HC-HJ or JQL (i.e., 247.40 ms versus 3651.00, 8447.00, or 80720.00 ms).

The bar charts in Figure 7 also reveal that JQL is normally slower than HC-HJ when the Join benchmark analyzes the medium and large objects and collections. We attribute this to the fact that JQL uses AspectJ advice to (i) track the Strings, Integers, and temporary arrays allocated by Join and (ii) subsequently traverse a complex heap graph when executing the join operator. Consequently, HC-HJ exhibits lower response times because it avoids the execution of the tracking advice required by JQL. The results in Figure 7 also show that the performance of HC-HJ and JQL improves as the object size increases from 10 to 100. This phenomenon occurs because the number of matching objects decreases as the maximum Integer value and String length increases, thus decreasing the number of temporary arrays allocated by HC-HJ and JQL, and thereby reducing response time.

Sub-Query Benchmark. The benchmarks also record the number of objects within the collection that stores the results from executing the chosen query (we use the "Matches" explanatory variable to denote the size of the collection). The last tree model in Figure 4 shows that the number of matching objects is the dominant factor in explaining the response time characteristics of the Sub-Query benchmark with Graphs. The number of matching objects best explains the response time of Sub-Query because we set up the benchmark to operate with object and collection sizes that are relatively small when compared to the other benchmarks. When configured as such, the selectivity of a query may dominate an application's performance. The regression tree model also indicates that HC and JQL exhibit lower average response times than JoSQL

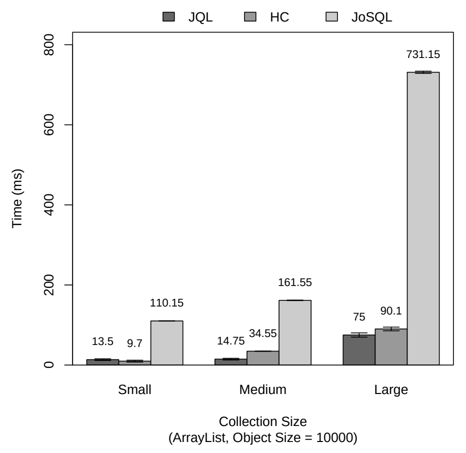

Figure 5: Query Benchmark with Integers.

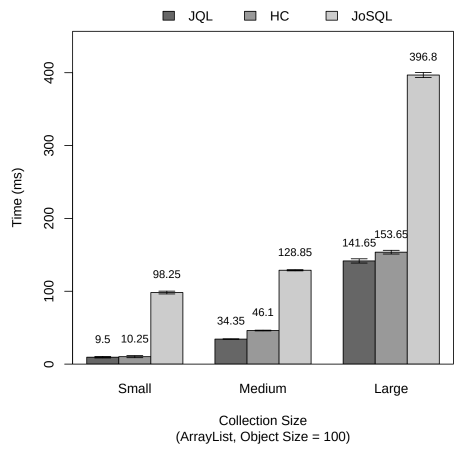

Figure 6: Query Benchmark with Strings.

(e.g., 42.41 versus 254.30 ms for Sub-Query benchmarks with comparatively few matches).

The bar charts in Figure 8 also expose the fact that JQL's response times are often lower than HC's when Sub-Query analyzes the small and medium sized collections of Graph objects. This result suggests that JQL's use of AspectJ advice to construct and traverse the heap graph can be more efficient than the iterative approach taken by HC. However, we also observe that JQL's response times increase as the size of both the objects and the collection increases (e.g., the "Large" bar group in the bottom right graph of Figure 8 shows the respective time overhead of JQL and HC at 216.25 and 194.10 ms). Once again, this phenomenon can be traced to the fact that JQL must use advice to track all of the objects and collections that the benchmark allocates to the JVM's heap. This trend is particularly noticeable for Sub-Query because each Graph object uses two collections to respectively store vertices and edges, with each vertex containing an additional collection for all of its incident edges.

5 Conclusions and Future Work

While tangentially connected to prior empirical studies of query execution in databases (e.g., [4, 8]), this article primarily relates to Willis et. al's description and evaluation of JQL [10]. Yet, to the best of our knowledge, this paper furnishes the first empirical comparison of two representative methods for finding data in unstructured JVM heaps (i.e., JQL and JoSQL). In conclusion, this paper presents a framework that uses simple benchmarks and automatically constructed tree models to yield insight into the response time characteristics of JQL and JoSQL. For instance, Figures 4 through 8 clearly reveal that JQL outperforms JoSQL by often exhibiting response time characteristics that are similar to the hand-coded implementations. The experiences in this paper also demonstrate the feasibility of using our benchmarks to make intelligent choices about query methods and subsequently avoid degrading the performance and/or correctness of a program's data analysis routines.

In future work, we intend to integrate new benchmarks and object types, while also considering different sizes of objects and collections. After further enhancing the benchmarks, we will also study the impact that caching can have on the performance of different methods for finding data in Java collections. Moreover, we will leverage more advanced statistical methods (e.g., analysis of variance, multiple comparison tests, and random forest generators) in order to further understand the trade-offs in response time. Ultimately, the combination of this paper and future work will yield a framework that enables developers to handle the software and data engineering challenges associated with using declarative and imperative approaches to finding data in unstructured heaps.

References

[1] G. Bentley, 2011. `http://josql.sourceforge.net/`.

[2] W. Cazzola. SmartReflection: efficient introspection in Java. *Journal of Object Technology*, 3(11), 2004.

[3] M. J. Crawley. *The R Book.* John Wiley & Sons, Inc., 2007.

[4] V. Ercegovac, D. J. DeWitt, and R. Ramakrishnan. The TEXTURE benchmark: measuring performance of text queries on a relational DBMS. In *Proc. of 31st VLDB*, 2005.

[5] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, Inc., 2006.

[6] G. Kiczales, E. Hillsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10), 2001.

[7] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of 12th VLDB*, 1986.

[8] J. Shao, D. A. Bell, and M. E. C. Hull. An experimental performance study of a pipelined recursive query processing strategy. In *Proc. of 2nd SDPDS*, 1990.

[9] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, Inc., 7th edition, 2010.

[10] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying in Java. In *Proc. of 20th ECOOP*, 2006.

[11] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *Proc. of 30th ICSE*, 2008.

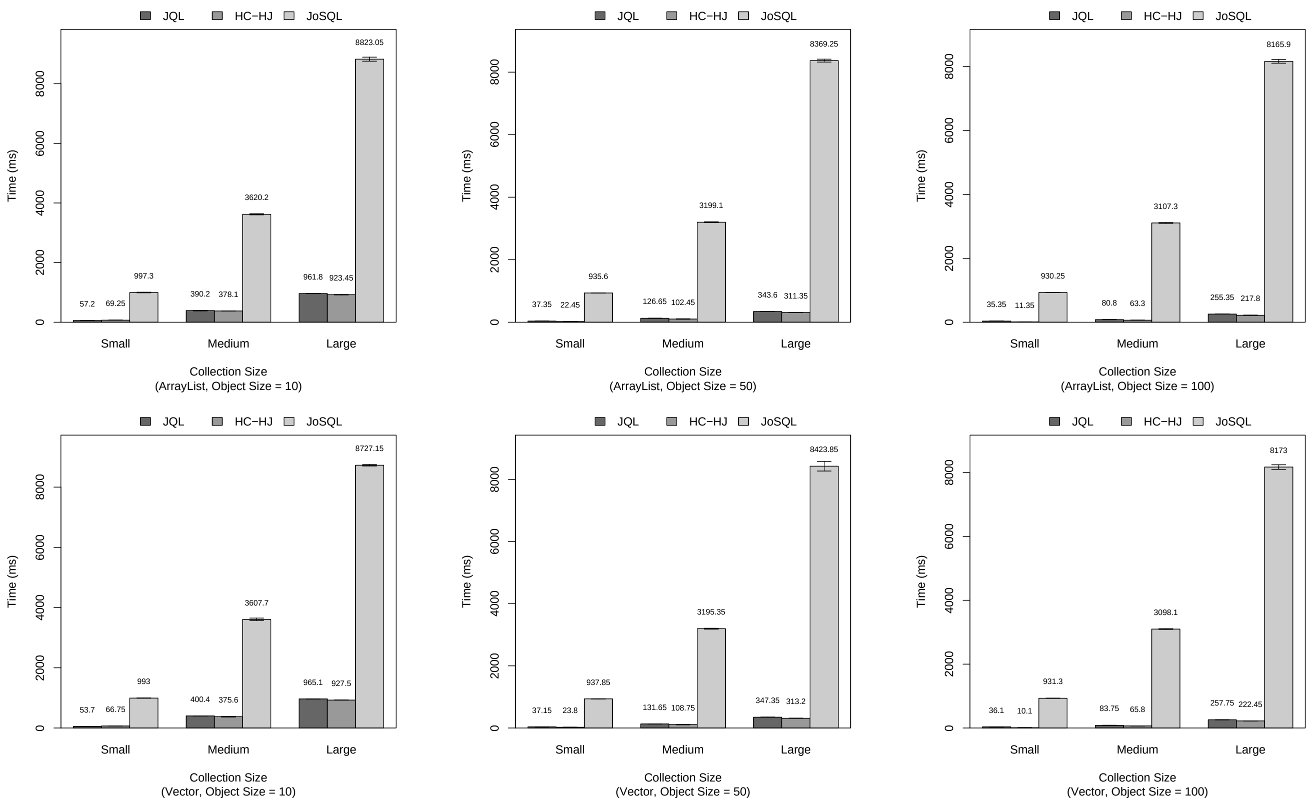

Figure 7: Join Benchmark with Integers and Strings.

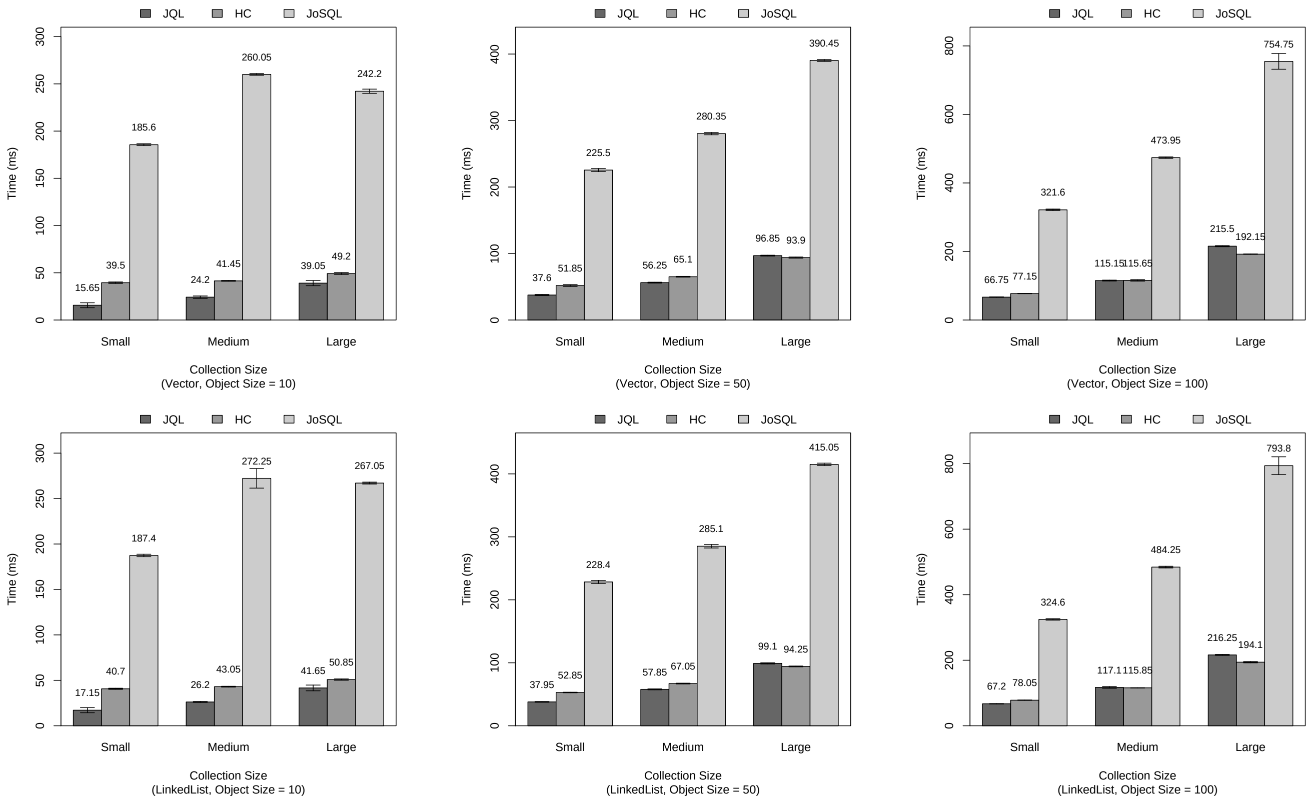

Figure 8: Sub-Query Benchmark with Graphs that Contain Strings.