The development of a query cost model for the weakly dependent subquery optimization of the Stack-Based Query Language

Gregory Carlin

Byram Hills High School

Armonk, NY

Abstract

The applications of databases range from social networks to mapping the human genome. The object-oriented database is a specific type of database that models data in a fashion similar to the model used by many popular programming languages. These types of databases, still largely in the experimental stage, are growing in popularity due to their ease of use. One of these experimental databases is the Object Database for Rapid Application Development (ODRA). This system interprets queries written in the Stack-Based Query Language (SBQL). Several optimizations can rewrite queries before executing them in order to make them faster. Some optimizations always produce a faster query. Others, like the weakly dependent subquery optimization, do not. These latter optimizations need a cost model in order to function properly. This model predicts the execution times for queries before and after optimization (but before execution) in order to predict which version will be faster. This study focused on the creation of this cost model, *which had not previously been developed for SBQL despite the number of optimizations requiring it*. Benchmarks were run on test queries in order to create statistical models for the execution times of SBQL operators. These individual operator models were then unified into a single cost model and implemented into ODRA. The implementation recursively traverses a query, summing the estimations for each operator. The implemented model was then tested using example queries and achieves the goal of accurately predicting at least 90% of all SBQL queries. This ultimately results in faster executions of queries.

1 Introduction

The quantity of information stored and analyzed is increasing exponentially. From 1986 to 2007, data stored in the world was estimated to have increased over 100-fold [1]. In almost any situation where a great quantity of data is stored and managed, a database management system (DBMS) is used. DBMSs organize the contents of databases, manage database connections, and execute queries. There are several varieties, including relational and object-oriented. One focus of object-oriented database management systems (OODBMSs) is ease of access for programmers. More complex systems create more opportunities for the inadvertent introduction of bugs. It has been estimated that half of the development time of an average programming project is devoted to debugging and that this amount of time is usually dramatically underestimated by developers [2]. Additionally, large amounts of data are being stored by many commercial applications. As a result, DBMSs are ubiquitous, and any significant optimization that can be made in any part of a DBMS would be a welcome improvement. My study is an attempt to create a query cost model that can predict the execution times of queries. This enables the database to effectively take advantage of certain query optimizations that do not always improve the execution time of a query by informing the DBMS when these particular optimizations can be used. The ability to utilize these optimizations in certain situations ultimately causes an increase in efficiency.

2 Review of Literature

Currently, the most popular type of DBMS is the relational database management system (RDBMS) [3]. This type of DBMS remains popular due to its high efficiency and reliability. However, like many popular programming languages and unlike RDBMSs, OODBMSs are based on the object-oriented model.

2.1 Relational Databases and The Object-Oriented Model

The organization of a database is defined by its *schema*. In a relational database (RDB), a schema contains the names of tables as well as the names and types of columns, among other information. In a complete system, there is usually more than a DBMS. Most application logic is done in application code, which is written in one or more programming languages and is

distinct from the database. The most popular languages for this application code follow the *object-oriented paradigm*. This is usually due to the reusability of objects and is similar to conceptual logic. Instead of columns, the object model has *attributes*. Fully-fledged DBMSs contain all the usual features of objects in programming languages including a typing system, encapsulation, and *methods*. Many of the example queries in this paper use the same example situation. For this situation, suppose the "Object Query Corporation" wanted to keep track of its employees in a database by creating an object for each of their employees. Each object could have several attributes: a *name* attribute that stores the employee's name as a string, a *gender* attribute that is set to either "male" or "female", and a *salary* attribute that stores an employee's weekly salary as an integer. Each employee could also have a method associated with it called *changeSal*(*integer*) that can be used to change his or her salary.

Since applications are usually written in object-oriented languages, but data is usually stored in relational databases, data needs to be converted between the two formats when it is read and written. This defining problem is called *object-relational impedance mismatch* and increases with the complexity of the data model. The use of an OODBMS eliminates this problem [4]. Graph database management systems, which are a subset of OODBMSs, are currently growing faster than any other type of DBMS with an increase of over 250% in usage during 2013 [5]. While RDBMSs still dominate the market commercially (8 out of 10 of the most popular DBMSs are relational [5]), OODBMSs manage hundreds of terabytes of data. They are used by the Large Hadron Collider at CERN and the Stanford Linear Accelerator, among other places [6]. If a data model is well-suited for the object-oriented paradigm, an OODBMS can be upwards of 1000 times faster than a RDBMS [7]. One OODBMS currently in its developmental stages is the Object Database for Rapid Application Development (ODRA). Because of the use of OODBMSs in advanced research, it is crucial to continue making improvements. My study offers one such improvement that focuses on query optimization in ODRA.

2.2 Querying Databases

In many database systems, the application code communicates with the DBMS via queries, or lines of text formatted in a syntax specific to the DBMS. Many popular RDBMSs read queries written in Structured Query Language (SQL). When a DBMS receives a query, it can rewrite

the query into a semantic equivalent that may take less time to execute than the original. This is called *optimization by query rewriting*.

In the ODRA system, queries are handled in two primary stages. The first one is the compile-time stage. Each query used in a DBMS only goes through the compile-time stage once in its lifetime. It takes place as soon as the query is received by the DBMS from the application. Most query optimization takes place at this stage and is called static optimization [8]. After static optimization, the query is converted to bytecode [9]. This bytecode can be saved in the DBMS and reused.

The second stage of a query's life is the run-time stage. This stage uses the bytecode produced in the compile-time stage. The bytecode is executed every time the query is called upon. This means that the query does not need to be re-optimized every time it is run. It also means the time the optimizer takes to rewrite the query is not of critical importance. This study focuses on static optimizations of queries written in the Stack-Based Query Language (SBQL).

2.3 The Stack-Based Query Language

SBQL is unique for a query language in that it also serves as a fully-fledged object-oriented programming language [10]. For example, there is no extra syntactical difference between expressions such as `2+2` and queries such as *`Employee`* **`where`** *`Salary`* `> 2000` [10]. This allows for the existence of statements such as *`Employee`* **`where`** *`Salary`* `> 2+2` and `2+(`*`Employee`* **`where`** *`Name`* `= "Smith").`*`Salary`*. This unification of both volatile and persistent variables is not a feature of most popular systems today [7] and further reduces the problem of object-relational impedance mismatch, making the system easier to use [10]. SBQL has many features, some of which are novel and some of which are adapted from other systems. Many optimizations rely on the presence of these features, which include the static environment stack and the metabase.

The environment stack. SBQL relies on an environment stack (ENVS) known from practically all programming languages as the *call stack*. In comparison to more popular languages, ENVS has new features dedicated to the processing of queries. Each section of ENVS contains information on some run-time environment. The bottom sections of ENVS contain references

to global information. They serve as a connection to the database and contain information on the current user session, computer libraries, and computer environment variables [10].

Each section contains structures called *binders*. These binders can be written as $n(x)$ where n is an external name that can be used in queries and referred to from outside the database, and x is an internal program entity (e.g., a reference to an object or a value). When a query is processed, ENVS is searched from the top to the bottom for the names in that query. Additionally, sections of ENVS may be omitted from the search if required by the current scope [11]. There are several optimizations that operate using ENVS, including the independent subquery optimization and the weakly dependent subquery optimization, both of which are described later in this paper.

The static environment stack. During static optimization, a static version of ENVS is used. This version is called S_ENVS and contains the signatures of objects instead of their values. These signatures primarily contain the object's type, but can also be linked to the metabase. S_ENVS is used to ensure that all names found in a query exist in the current environment and to perform calculations that determine what optimizations are possible for a given query.

The metabase and the schema. Another component of the stack-based architecture (SBA) is the metabase. It is compiled from the database schema, which presents the organization of the database. Additionally, the metabase can include statistics like the counts of different object types. The metabase is available at compile-time and is used in strong type checking as well as optimization [11]. An example schema (as stored in the metabase) similar to the one described in the Object Query Corporation example can be modeled in a diagram similar to those described by the Unified Modeling Language Standard (Figure 1, overleaf).

In the diagram, each large box represents a type of object. The name of the object is at the top of the box. This schema shows three possible object types: *Person*, *Emp*, and *Dept*. If this schema was to be used by the Object Query Corporation, the three types could represent a person the company would like to store data on, an employee of the company, and a department of the company, respectively. Next to each name is a cardinality, which defines how many instances of that object can be present in the database. Each is defined by two values: a minimum and a maximum. An asterisk indicates an unlimited number. In this example, there can be as many *Emp* objects in the database as needed, but there can also be no *Emp* objects. Below each

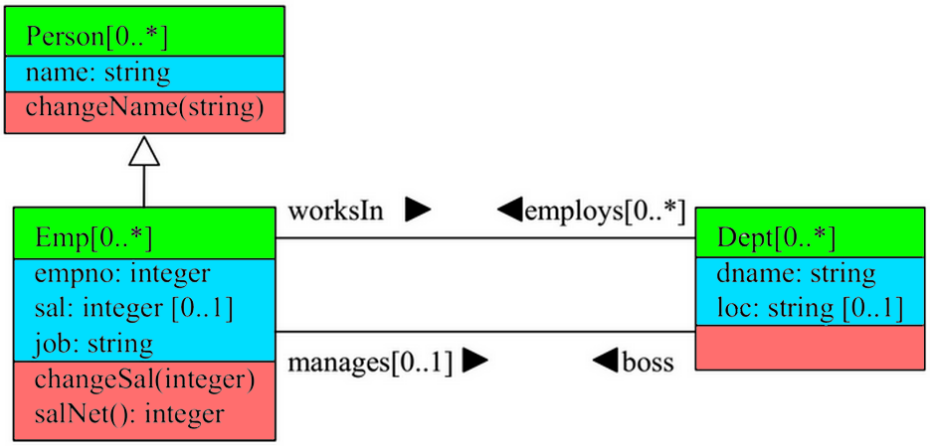

Fig. 1. An example schema showing the contents of and relationships between three types of objects. Adapted from *Object Oriented Query and Programming Languages - the Stack Based Approach* (p. 60) [11].

name is the list of attributes for that object, along with each attribute's type. In this case, *Emp* objects have the attribute *sal*. In addition to objects, attributes also have cardinalities. If a cardinality is not explicitly written, [1..1] is assumed (meaning exactly one of that attribute must be present). The *sal* attribute has a cardinality of [0..1], meaning that it may or may not be present in any given *Emp* object. Objects that have methods have them listed below the attributes. This schema also shows pointers between objects. In the recurring example, every *Emp* object has a pointer called *worksIn* to a *Dept* object that holds data on the department where an employee works. Pointers also have cardinalities. In this schema, the *Emp* object inherits from the *Person* object. Inheritance is fully supported by ODRA. Using these ideas, suppose the Object Query Corporation has an employee named "Smith" in their database. Suppose "Smith" worked in a *Dept* named "Accounting". He could be assigned an *empno* of 27 and a *job* of "Accountant". He may or may not have an assigned *sal*. "Smith" is defined as an *Emp*, but he is also considered a *Person* because *Emp* inherits from *Person*.

2.4 The Independent Subquery Optimization

Many optimizations developed for SBQL use features like the metabase to operate. The independent subquery optimization "factors out" part of a query that does not rely on previous parts of the same query. A limited version of it, not covering all cases, is implemented in SQL. The SBQL version relies primarily on S_ENVS and is best demonstrated through an example

query. Suppose that the Object Query Corporation wanted to know which of its employees were earning more than Smith. This can be accomplished by the following query:

```
Emp where sal > ((Emp where name = "Smith").sal)
```

As this query is currently written, the DBMS will evaluate it by iterating through each instance of *Emp* in the database. For each instance, it will compare the *sal* attribute of that *Emp* to the right side of the comparison operator. However, this right subquery is evaluated separately for each instance of *Emp* in the database, which means the DBMS will run a separate search for Smith for each stored employee. This particular subquery only needs to be evaluated once, until it finds Smith's salary. Not all queries fit this model.

To determine whether a query contains an independent subquery, the DBMS first determines (at compile-time) the level each name is bound to on ENVS. S_ENVS is used for this, along with other entities. Below is the example query rewritten with a number below each name and non-algebraic operator indicating its assigned section of ENVS.

```
Emp where sal > ((Emp where name = "Smith").sal)
 1    2    2        1    3    3             3 3
```

None of the names in the subquery `((Emp where name = "Smith").sal)` are bound to a section of ENVS opened by the first *where* operator, indicating that the subquery is independent of the left side of the query. Note that the distinction around the first *where* operator is specific to this particular example. This independence means that the subquery can be "factored out." The resulting value of this subquery is also assigned to a new name, so it can be referenced in the comparison. The result is below.

```
(((Emp where name = "Smith").sal) groupas c).
                    (Emp where sal > c)
```

This optimization is dominating. In other words, for every query it can be applied to, the rewritten query will execute at a time equal to or less than that of the unmodified version [12]. Thus, it can be used in a production environment without modification. However, not all optimizations

have this property. The improvement made in this study targets these latter optimizations. *This study is the first to make this improvement for SBQL, and the model created works for several query optimizations that are unique to SBQL.*

2.5 The Weakly Dependent Subquery Optimization

The weakly dependent subquery optimization is based on the independent subquery optimization. It "factors out" parts of a query that rely on an earlier part of the query but only take on a certain number of predefined values. It is not implemented in SQL. The best demonstration is through example. Consider the query below (*get every female employee whose salary is greater than 2000 as well as every male employee whose salary is greater than Smith's salary*). Written below each name and non-algebraic operator in the query is the section of ENVS it is bound to.

```
Emp as e where e.sal > ((2000 where e.sex = "female") union
 1          2    23 3               3    24 4
     (Emp where e.sex = "male" and name = "Smith").sal)
       1     3   24 4                  3              3 3
```

The subquery located to the right of the comparison operator is not independent of the subquery context. This is shown by the name *e*, which is bound to the second section of the stack, which is induced by the first *where* operator. This section is opened on the left of the operator. More conceptually, as the query processor iterates through the *Emp* objects in the database, the condition of the operator depends on a value located in the *Emp* object currently being evaluated. As a result, this query cannot be optimized via the independent subquery optimization. However, since the value depended upon is of the enumerated type, the weakly dependent subquery optimization is able to rewrite the query into a potentially faster version [13].

Query rewriting. The weakly dependent subquery optimization starts by breaking the original query into sections for each possible value of the enumerated type. The example query is rewritten below. The right side of the query is assigned to a temporary function *wdq* for clarity.

```
wdq(e.sex) = ((2000 where e.sex = "female") union
(Emp where e.sex = "male" and name = "Smith").sal)
```

```
Emp as e where if e.sex = "male" then e.sal > wdq("male")
                else e.sal > wdq("female")
```

After this initial rewriting, the contents of the function can then be substituted back into the query. The result of this is below.

```
Emp as e where if e.sex = "male"
then e.sal > ((2000 where "male" = "female") union
(Emp where "male" = "male" and name = "Smith").sal)
else e.sal > ((2000 where "female" = "female") union
(Emp where "female" = "male" and name = "Smith").sal)
```

This query contains several literal comparisons that can be evaluated at compile-time. Other optimizations already exist for SBQL that can then remove these. The result is below.

```
Emp as e where if e.sex = "male" then
e.sal > (Emp where name = "Smith").sal else e.sal > 2000
```

An independent subquery `(Emp where name = "Smith").sal` is now contained within the query. The weakly dependent subquery optimization does not always result in independent subqueries, but when it does (as in this case), the query can be further rewritten [13]. The final query is below.

```
((Emp where name = "Smith").sal group as aux).
(Emp as e where if e.sex = "male" then e.sal > aux
                else e.sal > 2000)
```

This query can actually be further optimized via methods such as indexing, but this is beyond the scope of this paper.

The query cost model. Unlike the independent subquery optimization, the weakly dependent subquery optimization is not dominating. Because of this, *there is no guarantee that a rewritten query will be faster than the original query.* Cases like this often occur when an independent

subquery is not produced or the enumerated type the subquery depends on has many possible values. The DBMS needs a method to predict at compile-time if a given query will benefit from the weakly dependent subquery optimization. Importantly, the weakly dependent subquery optimization cannot be used in a production environment without this method [14]. This method has not yet been created for SBQL and is therefore the focus of this study.

To predict the execution time of a given query, the DBMS can use a query cost model. This type of model looks at the contents of the query as well as the metabase to make its prediction. The prediction is based on the assumption that the DBMS will behave in the same way across different instances. This assumption would only fail in the case of multiple implementations of the same language or different versions of the language. This is not an issue as long as the cost model is updated to reflect changes in an implementation and the query is compiled on the same DBMS instance on which it is run [15]. For an optimization like the weakly dependent subquery optimization, a prediction can be made for both the pre-optimization and post-optimization queries. The predictions can then be compared in order to determine which version will execute more quickly.

A query cost model is considered to be acceptable if it can accurately predict which version of a query will execute more quickly for 90% or more of all queries [16]. The hypotheses of this paper are based on this number.

3 Hypotheses

H_0: The query cost model developed will be able to accurately predict whether a given query will benefit from optimization for less than 90% of all queries.

H_1: The query cost model developed will be able to accurately predict whether a given query will benefit from optimization for *greater than or equal to* 90% of all queries.

4 Objectives

1. Determine a statistical model that predicts the execution time of all SBQL operators based on statistics available in the metabase and the organization of the query.
2. Combine the models into one unified cost model.
3. Implement the cost model as an addition to the ODRA system.

4. Test the effectiveness of the cost model by running test queries.

5 Methods

I created this study under the guidance of my mentor, Dr. Kazimierz Subieta of the Polish-Japanese Institute of Information Technology in Warsaw, Poland. In the concluding sections of *The Optimization of Object-Oriented Queries involving Weakly Dependent Subqueries*, an article suggested by Dr. Subieta, I found the need for a query cost model discussed. I decided that I wanted to try this project and received approval for it. I secured the version of ODRA modified with enums from Dr. Michał Bleja, a colleague of Dr. Subieta. I was already familiar with Java, the language ODRA is written in, as I had taught it to myself over the course of several years prior to starting this project. I spent a good deal of time setting up ODRA on my computer and getting it to compile correctly. Because the issues were specific to my system, I did most of this independently. I learned how to work with compilation software and encountered unexpected issues. Several times a version of ODRA would cease to work, the code would have to be recompiled and the data rewritten. I also learned how to write queries in SBQL, a task made more difficult by its unstable nature in its implementation in ODRA. Queries I found in articles would often not run due to bugs and the fragmentation of the ODRA source code.

I completed the study with minimal assistance from Dr. Subieta and his colleagues, only receiving direction when a problem or question arose that I could not solve. All modifications of ODRA (done in Java), as well as the set up and execution of benchmarks, were done without external support. The parts of ODRA that needed to be modified were found independently. I also recognized the need to account for JVM warm-ups and modified the benchmarking utility accordingly. Code was written in Scala, a language based on Java, to generate test queries, organize the data, and manipulate the benchmark results. This included the subtraction of data sets from each other when an operator could not be completely isolated in the original test and sanitizing the data so it could be analyzed without issue. I taught myself Scala as I worked. The operators tested, as well as the differences between their cases, were chosen by me, and I analyzed the results of the benchmarks using Mathematica, which I also taught myself.

5.1 Benchmarking

The ODRA system was written with a built-in benchmarking tool. It allowed for queries to be run multiple times, with the execution times of those queries outputted to a file. However, ODRA is written in Java. Because of the nature of Java's implementations, benchmarks involving the language are prone to the "warm-up" problem. The Java Virtual Machine, which executes written Java code, may not immediately compile a given piece of code to completion [17]. As a result, if a given piece of code is being run many times in a row, the execution times of the first few iterations may be significantly greater than the execution times for the later iterations. ODRA's built-in benchmarking tool had to be modified to run queries a certain number of times before benchmarking in order to force compilation. This was done inside the ODRA source code. A warm-up of ten queries was chosen and implemented.

In order to further reduce error, each test query was executed 100 times after this warm-up. This means each test query was executed a total of 110 times. The time used in the analysis was the average of the last 100 executions.

The tool was also not designed for its output to be analyzed automatically. The output files contained column headers and extraneous data that might be useful to a human reader but got in the way of computer analysis. Additionally, a new output file was generated for each query run. The benchmarking tool was modified to output data in a simpler format and to allow for multiple related benchmarks to be output to the same file, thus allowing for easier analysis.

Queries were broken down to contain the least possible number of operators. Ideally, a query test case would only contain a single operator - the one being benchmarked. However, some operators require the use of additional operators in order to properly test. The execution times for these query test cases had to be adjusted in order to accurately reflect the execution time of the operator being tested. The execution times of the corresponding benchmarks for any extraneous operators were subtracted from the results of the test case. This adjusted data set is what was analyzed.

For each SBQL operator, a query test case was created. This test case manifested in a file that contained a long list of queries to be executed by ODRA. The differences between queries in a single test case were planned to isolate a single variable that could be correlated to the

execution time of that query. For example, the test case for the **bag** operator first ran a query that created an empty bag. The next query created a bag with a single number in it. The next query created a bag with two numbers. This continued until the end of the test case. Cases usually had 512 to 1024 queries, except for a certain few where the variable being tested could not take on 512 different states. Because the query test cases were so large, and each query was executed 110 times, query test cases could take a very long time to execute (i.e., anywhere from an hour to several days).

5.2 Benchmark Analysis

The files output from the modified benchmark utility were analyzed in Wolfram Mathematica 10. The best model for a given data set was determined manually using a scatterplot of the data. Different models were tested and overlaid on the scatterplot to determine the best fit. A coefficient of determination and p-values were calculated for each model.

5.3 Implementation

First, the optimization framework was modified to save a pre-optimized copy of a query. The optimization framework works by running both the pre-optimized query and the optimized query through the cost model. The model makes predictions on both versions, and the version predicted to be faster is outputted. If the predictions are equal, the optimized version is used. Exactly equal predictions will typically be a result of a given query not being applicable to optimization.

The bulk of the implementation was in the cost model itself. The model recursively traverses a query, predicting how much time each operator will take to execute. These predictions are made using the models found during analysis. It is important to note that these predictions are not absolute. In other words, the predictions made by the cost model should not be compared to the actual execution time of a query. The actual execution time will vary greatly by the system the database is run on. However, the predictions are relative and are designed to be compared to each other.

Because most models are based on the size of the arguments passed to an operator, the cost model also contains functionality that predicts the number of objects a given query returns. This

function uses cardinality information from the metabase to make predictions. It uses a worst case scenario approach. For example, a query that references a name is predicted to return the maximum cardinality of that name in the metabase. This means that optimization will be more accurate if the database developer has provided accurate cardinalities when creating the database schema.

5.4 Efficacy

After the implementation of the model, test queries were run through the system to check the efficacy of the model. Most test queries were based on example queries used in articles describing different optimizations. Due to ODRA's current development state, most test queries were designed to be optimized by either the independent subquery method or the weakly dependent subquery method. Examples from other articles would not execute properly due to issues with their optimizations' implementations.

6 Results

For every query tested, the cost model successfully predicted which version of the query would be faster. Twenty-eight queries were tested, yielding a p-value of 0.03888 from a 1-proportion z-test. With α set at 0.05, this result provides significant evidence that the model can predict at least 90% of all SBQL queries.

6.1 Model Components

Table one enumerates the different variables present in Table two (overleaf). Each variable is limited to the certain data type defined.

Table 1. Variable types present in results.

Variable	Data Type
n	Number (Any Real or Integer)
b	Boolean (True or False)
s	String
t	A Type (Real, Integer, Date, Boolean, or String)
o	Any Single Object
g	A Group of Objects (May only contain one)
$name$	A Name (Either local or external)

Table two is a list of the operators tested and the models that predict their execution times. Since these are models and are inexact, each one is checked dynamically to ensure that it does not predict a value below zero. The models all contain a single variable x. The value from this variable comes from the estimation of the number of items utilized in an operation. In a unary operation, this value is the number of items returned by the inner expression. In a binary operation, this value is usually the sum of the number of items returned by both the left and right expressions. However, if x is defined in the example query (e.g., in the second **union** query), x in the model uses the value of x in the example. The exception to these rules is the **where** operator, as it relies on two variables. x is the number of items returned by the left side of the expression, and z is the number of conditions on the right side of the expression.

It is important to note that every subquery in a given query is analyzed. This ensures that no part of the query can affect the model in a way that has not been properly taken into account. Even if a model defines an operator as negligible, the subqueries within that operator's expression are still analyzed and their execution times are predicted.

A notable omission from Table two is the **in** operator. This is because this operator is converted by ODRA to an equivalent **contains** operator.

Table 2. Tested SBQL operators and their results.

Query	**Model**	r^2	p**-values**
n **+** n	Negligible (Values too low)	-	-
n **–** n	Negligible (Values too low)	-	-
n **/** n	Negligible (Values too low)	-	-
n ***** n	Negligible (Values too low)	-	-
n **%** n	Negligible (Values too low)	-	-
-n	Negligible (Values too low)	-	-
$n > n$	Negligible (Values too low)	-	-
$n >= n$	Negligible (Values too low)	-	-
$n < n$	Negligible (Values too low)	-	-
$n <= n$	Negligible (Values too low)	-	-
b **and** b	Negligible (Values too low)	-	-
b **or** b	Negligible (Values too low)	-	-
g **as** $name$	Negligible (Values too low)	-	-
g **groupas** $name$	$-0.00591995 + 2.01902 \times 10^{-7}x^2$	0.316911	$0.619181, < .01$
avg(g)	$0.168342631 + 0.002959577x + 7.91336 \times 10^{-7}x^2$	0.994094	$< .01, < .01, < .01$
bag(g)	Negligible (Values too low)	-	-
s **+** s	Negligible (Values too low)	-	-
count(g)	Unpredictable (Values too dispersed)	0.000418795	0.374396, 0.368428
$o_1, o_2, ..., o_x$	$0.232373793 - 0.005537560x + 0.000014719x^2 + 4.80612 \times 10^{-8}x^3$	0.840262	$< .01, < .01, < .01$

Query	Model	r^2	p-values
if(b)	Unpredictable (Not enough data)	-	-
g **contains** o	$-0.0438918 + 6.17165 \times 10^{-7}x^2$	0.906365	$< .01, < .01$
ref(g)	Negligible (Values too low)	-	-
deref(g)	Negligible (Values too low)	-	-
g.$name$	Negligible (Values too low)	-	-
o = o	Negligible (Values too low)	-	-
o <> o	Negligible (Values too low)	-	-
exists($name$)	Negligible (Values too low)	-	-
forall(g) (b)	$0.368421 + 0.000115264x$	0.53822	$< .01, < .01$
forsome(g) (b)	$-0.0188783 + 6.62542 \times 10^{-7}x^2$	0.912928	$0.134027, < .01$
$g[n]$	Negligible (Values too low)	-	-
g **intersect** g	Unpredictable (Values too dispersed)	0.131829	$< .01, 0.824746$
g **join** g	$0.662582 + 0.0000856859x^2$	0.999975	$< .01, < .01$
max(g)	Unpredictable (Values too dispersed)	0.00486482	$0.866849, 0.0847042$
min(g)	$0.248804 - 0.00107x + 9.2684 \times 10^{-7}x^2$ *	0.794276	$< .01, < .01, < .01$
now()	Unpredictable (Not enough data)	-	-
g **orderby** $name$	$-10.963285 + 0.893485000x$	0.999835	$< .01, < .01$
g **rangeas** $name$	Negligible (Values too low)	-	-
$s \sim\sim s$	if(x $<$ 103) 0.676408 else 2.00131	0.979792	$< .01, < .01$
struct(g)	$3.9893 + 0.0000150523x^2$	0.83348	$< .01, < .01$
g **subtract** g	Negligible (Values too low)	-	-
sum(g)	$-0.16496286 + 0.003835860x$	0.989362	$< .01, < .01$
g **union** g	$0.198532 - 0.000831706x + 8.05861 \times 10^{-7}x^2$ **	0.791186	$< .01, < .01, < .01$
o_1 **union** ... **union** o_x	$-0.288195 + 0.0000292202x^2$	0.99956	$< .01, < .01$
unique(g)	Negligible (Values too low)	-	-
uniqueref(g)	Negligible (Values too low)	-	-
(t) o	Negligible (Values too low)	-	-
g **where** b	$9.25131 + 9.13024x + 0.0650857z$	0.999906	$< .01, < .01, 0.823875$

* If $x < 832$, an execution time of 0 should be assumed.

** If $x < 655$, an execution time of 0 should be assumed.

6.2 The Combined Model

Twenty-eight different queries were used to test the effectiveness of the cost model. These queries can be found in Table three.

Table 3. Tested SBQL queries.

Query	Optimization	Gain
Emp **where** $sal >$ (Emp **where** $lname$ = "Smith").sal	Independent	49.35x
Emp **where** $sal >$ **avg**($Emp.sal$)	Independent	52.13x
Emp **where** sal = **max**($Emp.sal$)	Independent	55.55x
Emp **where** sal = **min**($Emp.sal$)	Independent	55.50x
Emp **where** $hire\_date >$ (Emp **where** $lname$ = "Smith").$hire\_date$	Independent	49.21x
Emp **where** $hire\_date$ = **max**($Emp.hire\_date$)	Independent	55.60x
Emp **where** $hire\_date$ = **min**($Emp.hire\_date$)	Independent	55.43x

Query	Optimization	Gain
Emp **where** $birthday > (Emp[42]).birthday$	Independent	3.00x*
Emp **where** $works\_in.Dept$ = (Emp **where** $lname$ = "Smith").$works\_in.Dept$	Independent	41.68x
$Dept$ **where count**($employs$) > **avg**(($Dept$ **join count**($employs$) as x).x)	Independent	193.15x
$Dept$ **where count**($employs$) = **max**(($Dept$ **join count**($employs$) as x).x)	Independent	186.99x
$Dept$ **where count**($employs$) = **min**(($Dept$ **join count**($employs$) as x).x)	Independent	187.12x
$Part$ **where** $detailCost > (Part$ **where** $name$ = "Kemp").$detailCost$	Independent	49.84x
$Part$ **where** $kind$ = ($Part$ **where** $name$ = "Kemp").$kind$	Independent	49.25x
$Part$ **where** $kind$ = ($Part$[42]).$kind$	Independent	2.50x*
$Part$ **where** $detailMass < (Part$ **where** $name$ = "Kemp").$detailMass$	Independent	47.76x
$Part$ **where count**($component$) > **avg**(($Part$ **join count**($component$) as x).x)	Independent	74.78x
$Part$ **where count**($component$) = **max**(($Part$ **join count**($component$) as x).x)	Independent	68.17x
$Part$ **where count**($component$) = **min**(($Part$ **join count**($component$) as x).x)	Independent	65.20x
$Part$ **where exists**($component$) **and sum**($component.amount$) > **avg**(($Part$ **where exists**($component$) **join sum**($component.amount$) as x).x)	Independent	33.54x
(Emp **as** e) **where** $e.sal >$ **avg**((Emp **as** f **where** $e.sex = f.sex$).$f.sal$)	Weakly Dependent	47.44x
(Emp **as** e) **where** $e.sal >$ **avg**((Emp **as** f **where** $e.position = f.position$).$f.sal$)	Weakly Dependent	33.02x
(Emp **as** e) **where** $e.sal =$ **max**((Emp **as** f **where** $e.sex = f.sex$).$f.sal$)	Weakly Dependent	46.62x
(Emp **as** e) **where** $e.sal =$ **max**((Emp **as** f **where** $e.position = f.position$).$f.sal$)	Weakly Dependent	33.89x
(Emp **as** e) **where** $e.sal =$ **min**((Emp **as** f **where** $e.sex = f.sex$).$f.sal$)	Weakly Dependent	46.63x
(Emp **as** e) **where** $e.sal =$ **min**((Emp **as** f **where** $e.position = f.position$).$f.sal$)	Weakly Dependent	33.82x
(Emp **as** e) **where** $e.sal >$ **avg**((Emp **as** f **where** $e.works\_in.Dept = f.works\_in.Dept$).$f.sal$)	Error	1.77x*
$Part$ **as** p **where** $p.detailCost >$ **avg**(($Part$ **as** q **where** $q.kind = p.kind$).$q.detailCost$)	Error	1.63x*

* Indicates a query where no serious gain was realized from the optimization framework.

Twenty of these queries targeted the independent subquery optimization, six of them targeted the weakly dependent subquery optimization, and the remaining two targeted other optimizations. These queries were also benchmarked both before and after optimization. Four of these queries (as indicated by an asterisk in the "Gain" column) did not demonstrate a large decrease in execution times, but all others did. The cost model successfully predicted no gain from optimization for these four queries, and predicted a gain for all others. The success of these predictions yields a p-value of 0.03888 from a 1-proportion z-test for the tested hypothesis.

7 Discussion

7.1 Model Components

Most operators show significant evidence that the model created for them is accurate, as evidenced by their high r^2 and low p values. The exact values of these variables can be found in Table two. 42 out of 47 of the operators tested yielded a determinate result. Any testing completed on a system different from the one these tests were run on could not be immediately

included into the cost model since times between systems are not comparable. However, it is theoretically possible that a relationship could be found between the benchmarks in this study and ones completed on another system. One could replicate the tests in this study to find this relationship and then adjust the models for previously untested operators to match the models already implemented in ODRA.

7.2 The Combined Model

The operators that could not be benchmarked were implemented as negligible. However, if warnings are turned on in the database, any use of those operators will alert the user that the operator is not included in the cost model. This enables the database user to be fully aware of situations when he or she may receive unexpected results and can adjust the database configuration accordingly. For the remaining operators, the cost model created has been shown to be effective in a variety of situations.

8 Conclusion

This study is the first of its kind to develop a query cost model for SBQL [14]. It succeeded in achieving its goal and demonstrated significant evidence favoring the desired hypothesis. The null hypothesis was rejected. Models were successfully developed for 15 different SBQL operators, and 27 more were found to be negligible. This leaves only five testable operators with indeterminate results. In total, 47 operators were tested.

A novel cost model has been created for the ODRA system that has been shown to be effective for at least 90% of all SBQL queries. It is general, and therefore offers the advantage of use in conjunction with any arbitrary optimization or series of optimizations.

Importantly, the query cost model can be used on any optimization implemented in ODRA without any further changes. Every query ran at compile-time is automatically analyzed by the model, without any further work by the user. However, it can easily be switched off with little modification to the ODRA source code. The model allows many optimizations to be used in ODRA that could not be previously. These optimizations include the compound weakly dependent subquery optimization [18], the large and small collection optimization [19], optimization via indexing [9], and the weakly dependent subquery optimization [13]. The creation of this

cost model enables the optimization of many more queries in SBQL and improves the feasibility of SBQL and ODRA to be used in future projects.

Several operators could not be benchmarked for various reasons. The operators **leavesuniqueby**, **!∼**, **dateprec**, and **not** could not be executed properly due to errors in ODRA. Additionally, the operators **closeuniqueby**, **leavesby**, and **closeby** could not be benchmarked because their execution times in the testing environment were too great to be tested in a timely manner while maintaining the number of executions used by other operators. The testing of these operators is subject to further research.

Additionally, the testing of the cost model was limited by errors in the implementations of several optimizations. Notably, the last two queries in Table three should be optimizable by the optimization framework, but due to errors were not properly rewritten. Many other test queries could not even be executed. Further and more extensive testing of the unified model could also be the subject of future work.

The model also does not distinguish between operators that are theoretically dominant and those that are not. This could also be a future improvement.

The model was *heretofore undeveloped for SBQL in the literature*. It ultimately makes DBMSs more efficient and has the potential to improve speeds in everything from a small business paying its employees to a large social network connecting friends. It also has applications in massive research projects like the Brain Research through Advancing Innovative Neurotechnologies (BRAIN) Initiative that will require data to be stored in formats that are unsupported by the more popular RDBMSs of today.

9 Acknowledgements

This project and paper could not have been possible without the assistance and guidance of my mentor, Dr. Kazimierz Subieta, as well as my research teachers: Mr. David Keith, Mrs. Stephanie Greenwald, and Mr. Ken Kaplan. I also appreciate the direction given to me by my computer science teacher Mr. Christopher Lewick and the continued support and assistance of my friends, parents, and siblings.

References

1. M. Hilbert and P. Lòpez. **The world's technological capacity to store, communicate, and compute information.** *Science*, 2001, **332** (6025), pp.60-65.

2. F. P. Brooks. The Mythical Man-Month. *In: The mythical man-month: Essays on software engineering*. 35th ed. Boston: Addison-Wesley, 1995, pp.19-20.

3. P. Andlinger. *Open source RDBMS are gaining in popularity, but jobs are found elsewhere*, 2014. [Online] Available from http://db-engines.com/en/blog_post/35 [Accessed 25 Sep 2014].

4. K. Subieta. **Object database systems.** Unpublished manuscript, 2000.

5. M. Gelbmann. *Graph DBMSs are gaining in popularity faster than any other database category*, 2014. [Online] Available from http://db-engines.com/en/blog_post/26 [Accessed 25 Sep 2014].

6. D. Obasanjo. *An exploration of object oriented database management systems*, 2001. [Online] Available from http://antares.itmorelia.edu.mx/~fmorales/TopAvBdatos/01%20ModelosEmergentes/BD%20Orientadas%20a%20Objetos/70%20WhyArentYouUsingAnOODBMS.pdf [Accessed 25 Sep 2014].

7. K. Subieta. *Recent Article and Idea*. Personal email to: G. Carlin, 1 Apr, 2013.

8. J. Płodzień and K. Subieta. **Applying low-level query optimization techniques by rewriting.** *Lecture Notes in Computer Science,* 2001, **2113**, pp.867-876.

9. J. Płodzień and K. Subieta. Static analysis of queries as a tool for static optimization. In *Proceedings of the 2001 International Symposium on Database Engineering & Applications*, Washington, DC, USA: IEEE Computer Society, 16-18 Jul. 2001, p.117.

10. R. Adamus, P. Habela, K. Kaczmarski, M. Letner, K. Stencel, and K. Subieta. **Stack-based architecture and stack-based query language.** *ICOODB,* 2008, pp.77-96.

11. K. Subieta. **Object oriented query and programming languages - the stack based approach.** Unpublished manuscript, 2011.

12. J. Płodzień and A. Kraken. **Object query optimization through detecting independent subqueries.** *Information Systems,* 2000, **25** (8), pp.467-490.

13. M. Bleja, T. Kowalski, R. Adamus, and K. Subieta. **Optimization of object-oriented queries involving weakly dependent subqueries.** *Lecture Notes in Computer Science,* 2009, **5936**, pp.77-94.

14. K. Subieta. *Weakly Dependent Subquery Optimization with a Query Cost Model*. Personal email to: G. Carlin, 20 Nov, 2013.

15. G. Gardarin, F. Sha, and Z. Tang. **Calibrating the query optimizer cost model of IRO-DB an object-oriented federated database system.** In *Proceedings of the 22nd Conference on Very Large Data Bases*, Mumbai, India: Very Large Data Base Endowment Inc, 3-6 Sep. 1996, pp.378-389.

16. K. Subieta. *Several Questions*. Personal email to: G. Carlin, 11 Feb, 2014.

17. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. **A benchmark suite for high performance Java.** *Concurrency - Practice and Experience,* 2000, **12** (6), pp.375-388.

18. M. Bleja, T. Kowalski, and K. Subieta. **Optimization of object-oriented queries through rewriting compound weakly dependent subqueries.** *Lecture Notes in Computer Science,*

2010, **6261**, pp.323-330.

19. M. Bleja, K. Stencel, and K. Subieta. **Optimization of object-oriented queries addressing large and small collections.** In *Proceedings of the International Multiconference on Computer Science and Information Technology*, Mragowo, Poland: IEEE, 12-14 Oct, 2009, pp.643-650.