Test-driven development of a performance-critical database record-tree balancing application

Jonathan Mack

Problem Report
submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Graduate Committee:
James D. Mooney, Ph.D. (Chair)
Bojan Cukic, Ph.D.
Timothy J. Menzies, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, WV
2008

Keywords: database, Java, performance, test-driven development, load balancing, record tree

Abstract

Test-driven development of a performance-critical database record-tree balancing application

Jonathan Mack

This Problem Report details the design, implementation, and testing of a multi-table database record tree balancing application created by the author during summer 2008.

The application is designed to move trees of database records between user objects known as Managers. Records are associated with Managers by their value in a certain field of that record. Along with a unique identifier and version ID number, these three fields form a composite key across all records in the database. The purpose of the application is to move all appropriate record trees in a user-specified database from existing Managers to desired ones, balancing them among the desired Managers, and maintaining composite key uniqueness. Additionally, since a database may contain millions of records, and since the database must be taken offline for this process, it is desired that this process complete as quickly as possible.

In this Report, application requirements are detailed, followed by application design decisions, ranging from high-level user configuration issues to details of specific fundamental methods used in the application. Performance and testing considerations are then detailed, followed by a final look at the state of the application and possible future work along its same lines.

Acknowledgments

I would like to thank my family and all of my instructors and teachers, who pointed me in a direction such that academic achievement of any type was possible. I would like to thank Strictly Business Computer Systems, Inc., for giving me the project on which this Report is based, and its employees (especially Jason Stadler) for their support and suggestions in its design and development. Also for his support and suggestions, I would like to thank my principal advisor, Dr. James Mooney.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DDL:	Database Definition Language
DML:	Database Manipulation Language
JDBC:	Java Database Connectivity
URL:	Uniform Resource Locator
XML:	extensible Markup Language

1. Introduction

This Problem Report details the design, implementation, and testing of a multi-table record tree balancing application.

In May 2008, the author began a summer software engineering internship with Strictly Business Computer Systems, Inc., of Huntington, WV. Strictly Business was working on a suite of applications for a client, but did not have any software engineers to spare for the balancing project. Additionally, since it was thought that the project was small enough in scope to be finished by one person in a summer, and independent enough that one person could work on it, the project was given to the author at the start of the internship.

The author worked independently on this project throughout the summer, interacting with other company software engineers for suggestions and advice. In addition, a code review of the application was conducted that included all pertinent software engineers in early July 2008. By the end of the internship in August 2008, the application had passed all required correctness and performance tests, and was delivered to the lead software engineer of the company for final approval and delivery to the customer.

This Report details the design, implementation, and testing of the balancing application. Application requirements are detailed, followed by application design decisions, ranging from high-level user configuration issues to details of specific fundamental methods used in the application. Performance and testing considerations are then detailed, followed by an investigation of the final state of the application and possible future work along its same lines.

2. Application requirements

Introduction

Application requirements were delivered to the author in late May 2008. Requirements for this project are discussed below; unless otherwise noted, all requirements were either designated so by the user, or were indirectly needed due to existing user database, hardware, or software installations.

Database/Table Configuration/Fields

At its most basic level, the application requires the user to be able to “balance” trees of records in multiple database tables. Before detailing exactly what balancing means, it is important to note the fields in the database tables related to balancing.

Tables may have multiple fields, but seven are of interest in terms of this application (data types are in parentheses after each field name):

- **tree_id** (varchar2): the ID of the tree with which the record is associated. All records within the same tree have the same **tree_id**, and each tree has a unique **tree_id**.
- **manager_id** (varchar2): the unique ID of the Manager with which the record is associated. With the **unique_identifier** and **version_id**, forms a composite key across all database tables being considered. All records within a tree have the same value of **manager_id**.
- **unique_identifier** (number(10)): along with the **version_id** and **manager_id**, forms a composite key across all database tables being considered.
- **version_id** (number(10)): along with the **unique_identifier** and **manager_id**, forms a composite key across all database tables being considered.
- **parent_id** (number(10)): the **unique_identifier** of this record's parent in the tree. Tree root records have a value of either 0 or null for this field. Some tables may not contain this field.
- **parent_version_id** (number(10)): the **version_id** of this record's parent in the tree. Tree root records have a value of either 0 or null for this field. Some tables may not contain this field.
- **live** (char(1 byte)): value indicating whether this record can be processed (i.e., is “balanceable”), either 'T' or 'F'. All records within the same tree have the same value of **live**.

Trees

Each record is a member of exactly one Record Tree (RT). Each parent in an RT may have any number of children. Each RT may have records from one or more tables. An example RT, using records from example tables **table1**, **table2**, and **table3**, might be:

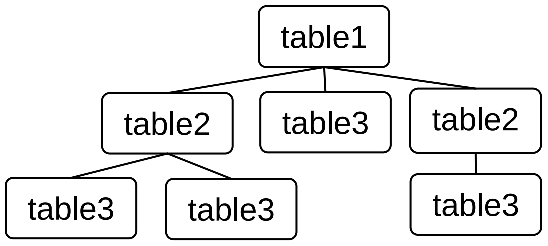

Figure 1: Sample Record Tree.

Tree Movement Requirements

The goal of the application is to balance the number of trees among a list of desired Managers, by "moving" them from one Manager to another. This is accomplished by changing the **manager_id** of the records in the tree from the one associated with the current Manager to the one associated with the desired Manager. It is possible, however, that the value of the **unique_identifier** and **version_id** of the record being moved may be the same as that of a record that already has the new **manager_id**. As **manager_id**, **unique_identifier**, and **version_id** are all parts of the database-wide composite key, the **unique_identifier** or **version_id** of a record being moved may need to be changed as well. In addition, since the combination of **unique_identifier/parent_id** and **version_id/parent_version_id** establishes parent/child relationships within a tree, this may necessitate an update of the **parent_id** or **parent_version_id** of records being moved as well.

User Configurability/Interface Requirements

The user's main goal is to move all live Record Trees associated with existing Managers to those associated with a user-specified list of desired Managers. The following additional requirements, however, were specified by the user:

1. The user may wish for some tables or Managers not to be balanced, and may desire to balance different databases. Database connection information, as well as the list of tables and Managers to be considered when balancing, will therefore be supplied by the user.
2. Each Manager has a current load and desired load. The current load is the number of live trees (trees whose records have value of **live** = 'T' associated with them) associated with that Manager, in the specified tables. The desired load is the number of trees in user-specified tables that should be associated with that Manager after balancing. For current but not desired Managers, the desired load is zero. The desired load should be roughly equal for all desired Managers. After balancing, current load should equal desired load for each Manager.
3. The list of desired Manager IDs may contain some, all, or none of the current Manager IDs, and may include IDs that aren't in the list of current IDs.
4. Some tables may consist of only root records, and do not have **parent_id** or **parent_version_id** fields. The application needs to check for this and respond accordingly.
5. No assumptions can be made regarding the locations of child and root records. Child and root records may exist in the same table.

Software and Performance Requirements

The following software and performance requirements were also specified:

1. The application must interface with an Oracle database, as this is how the user's data is stored.
2. There is no specific programming language in which the application must be implemented.
3. No assumptions can be made about the graphics capability of the user console; a text interface is therefore preferred.
4. The tables used for balancing may have millions of records, so performance (in terms of the total time needed for the application to execute) is critical. No specific performance requirements were given, but experience with the customer databases indicated that the application needed to be able to completely process databases with numbers of records in the low millions within no more than approximately five hours.

3. Application Design

Language design decisions are first discussed, followed by high- and low-level detailed application design. Performance design decisions involve many different aspects of the application, so design performance considerations are detailed in the Performance Section of this Report.

Development Language/Environment Selection

The project was implemented using the Java programming language. No user requirement specified this, but it was the language most used by both the client and Strictly Business, and it was also the language with which the author was most familiar. In addition, three extensions to Java were used in the creation of this application: dom4j, log4j, and JUnit.

dom4j is an open-source extension designed to, among other things, allow Java to interact with XML. The extension contains methods to allow the creation, editing, search, and retrieval of XML data. In this application, dom4j is used to retrieve information from the user configuration file, as well as the files used to create test database tables and records.

log4j is a highly-configurable open-source logging extension. It allows logging of constant and variable text strings to various locations, including the console and multiple text files. Since test-driven development was an integral part of the design, it was deemed important to be able to switch quickly between user-appropriate output (a small number of console-only messages) and test-appropriate output (a larger amount of console output, as well as database and record output to log files). To this end, log4j methods were used for all console output, including that to users. A logging configuration file (log4jconfig.txt) was used to switch between the two types of logging.

In general, a logging level of DEBUG was used for testing logging; this was primarily used to gather data sent to the logging file (log4jLogFile.txt). A logging level of INFO was used for all output sent to the console, and was the primary vehicle for user interaction. At the start of application execution, the logging level was determined and saved by Java; this allowed the application to skip entirely expensive data logging loops when performance was being tested. Performance testing of console output using log4j's INFO logging level versus Java-based output (using System.out), showed a negligible performance difference, justifying the use of log4j for all console output.

JUnit is a unit-testing extension for Java. It allows for creation of multiple user-defined testing routines that can be run on the same application. JUnit was used for testing the application's data-gathering methods, multiple current/desired Manager configurations on both random and premade databases, and the test database creation process itself.

Though not a Java extension, Javadoc should also be mentioned here. Javadoc-compliant commenting was used throughout the application and its test suite, and Javadoc html pages detailing application classes, methods, and attributes were generated for all classes.

The application was coded using the Eclipse IDE, and tested using Eclipse's JUnit Eclipse extension.

High-Level Application Design Detail

The application first retrieves and checks database and user configuration file information. Especially important is the number of live trees associated with each current Manager, as the total from all Managers is divided by the number of desired Managers to get the desired load of each desired manager. (Any remainder is distributed among the desired Managers).

The application then displays the database location URL, the list of **manager_id**s along with their current and desired loads, and asks the user if they wish to proceed. If that is the case, the application then balances RTs, moving all live record trees in specified tables from the current Managers to the desired ones, such that post-balance current loads equal desired loads.

unique_identifier/**version_id**/**manager_id** uniqueness and parent/child relationships are preserved by searching all records associated with the destination **manager_id**, and retrieving the maximum unique_identifier for each tree move. This value is then added to the **unique_identifier** and **parent_id** of each record being moved. (See *updateRecordTree* under TableGroup.java later in this section for more information.)

During balancing, progress updates are displayed to the user at regular intervals. After balancing, the user is shown the current and desired loads of each Manager, and the application ends.

Low-Level Application Design Detail

As much as possible, the principles of object-oriented design were adhered to for this application. Application functionality was therefore split into Java Classes that maximized cohesion. Application files are as follows:

- **LoadBalancer.java**: executor class, designed to interface with the user, catch exceptions, and execute methods in other classes in order to provide required functionality.
- **Manager.java**. This class models a Manager. It contains Manager-related attributes and getter/setter methods used by the TableGroup class.
- **Table.java**. This class models a database Table, and contains Table-related attributes and Methods.
- **TableGroup.java**. This class contains the most important methods in the application. Methods in this class interface with the Manager and Table classes, and are the methods used by the LoadBalancer class to execute the application.
- **A user configuration file**. This is not a java class, but an XML file where user configuration is set.

The structure of each file is next detailed.

Configuration File

In order to facilitate execution, the user supplies a file that contains database connection information, the list of tables and **manager_id**s to be considered when balancing (the current Managers), and the list of desired **manager_id**s. To ensure maximum compatibility and portability, it was formatted as an XML (eXtensible Markup Language) file. Its design is as follows:

```
<config>
 <currentManagers>
    <ID>Current Manager ID 1</ID>
    <ID>Current Manager ID 2</ID>
    …
 </currentManagers>
 <desiredManagers>
    <ID>Desired Manager ID 1</ID>
    <ID>Desired Manager ID 2</ID>
    …
 </desiredManagers>
 <tables>
    <name>Table Name 1</name>
    <name>Table Name 2</name>
    …
 </tables>
 <databaseInfo>
    <driver>Java database driver name</driver>
    <url>database URL</url>
    <id>user ID</id>
    <password>password</password>
 </databaseInfo>
</config>
```

LoadBalancer.java

This class contains only one method; the main method of the class, which also serves as the entry point of the application. When the application is begun, this method is supplied at least one user argument. The first contains the path to the location of the user's configuration file. The second argument is optional; if it is given, it contains the name of the configuration file. If it is not supplied, the default file name (defaultConfig.xml) is used.

The LoadBalancer class begins by creating a new TableGroup object. Methods within TableGroup then open and parse the XML configuration file, and retrieve associated database information. If there are database tables that contain all the fields needed for balancing, but aren't specified by the user, the user is notified of this, and asked if they wish to continue. If so, the user is shown the database URL and names of tables and Managers to be balanced, along with the Manager RT loads, both before and after balancing. If the user decides to do so, RT loads are then balanced among specified Managers, per the following:

- Unmovable RTs (either due to not being live, or not being in user-specified tables or Managers) are not moved
- All movable RTs are moved from specified current to specified desired Managers
- The current and desired load for each non-desired Manager is zero.

In addition to the aforementioned, LoadBalancer is the final destination for exceptions thrown by other classes in the application. If an exception was thrown, LoadBalancer displays its stack trace, and aborts the application.

TableGroup.java

This class forms the backbone of the application. It provides methods that gathers user and database information, and balances database loads. Due to its importance, all of its significant methods are detailed, in the order in which they're used in the application. The order presented here, unless otherwise noted, mirrors the order in which the methods are called by LoadBalancer.java.

parseConfig: This method opens the user's XML configuration file, and parses the information inside.

openConnection establishes the connection between Java and the appropriate database tables using the driver, URL, and user ID/password information found in the user's configuration file. If the file contains bad/missing data, this method prints an appropriate message, and throws an appropriate exception. Exceptions with explicitly defined user messages include missing data, an incorrect driver name, and incorrect URL, ID, or password.

setTables: This method sets the user-specified names of tables that have RTs that could be balanced. This method first retrieves all table names from the database specified in *openConnection*. It then retrieves all table names (to be considered when balancing) from the user's configuration file. It checks for the following conditions, with the associated action:

- <tables><name> XML tag with no data in configuration file: warn user with console message, but continue.
- Table name in configuration file, but not in database: display error message, and throw exception which aborts application.
- Duplicate table name in configuration file: warn user with console message, but continue.

If no fatal exceptions were thrown, Table objects (see Tables.java) are created for all tables in both the configuration file and database, and added to the mapping of all table names to table objects (*tables*) kept by each instantiation of TableGroup.

checkTables determines whether any balanceable tables (i.e., those with the requisite fields) exist in the database that weren't specified by the user. If any of these exist, they are returned to LoadBalancer.java. LoadBalancer then queries the user, to see if they wish to continue, knowing that some balanceable tables won't be considered when balancing.

setParentID: Some balanceable tables contain records that are not part of a multi-record tree, but may still be associated with balanceable tables, and current/desired Managers. The application treats these records as trees with one node. Tables for which this is true, however, do not have **parent_id** and **parent_version_id** fields. Since these fields are used in the actual method that balances loads (*balance*), and that method must be as fast as possible, it was deemed appropriate to determine table configuration beforehand. *setParentID* does this, setting the *parentID* attribute of each table's Table object to true if it does contain the **parent_id** field.

createPreparedStatements: This method uses JDBC (Java Database Connectivity), the Java standard for connecting to databases. One of its features is the PreparedStatement class, which allows developers to create general database statements (e.g., "SELECT first_name FROM Employees WHERE years_employed > ?") which are saved at the database level, and can be reused with various values in place of the '?'. In order to increase performance, PreparedStatements were heavily used in this application. The purpose of this method is to create the PreparedStatements used by *balance* and the methods that it calls. The following PreparedStatements are created and stored as variables accessible by all methods in the class:

- *retrieveRTs*: This PreparedStatement retrieves the **tree_id** of all live trees in user-desired tables associated with a given **manager_id**.
- *retrieveMaxUniqueID*: This PreparedStatement retrieves the maximum **unique_identifier** of a given manager across all live RTs and user-desired tables.

In addition, a PreparedStatement is created and set as an attribute for each Table object. This PreparedStatement updates all records with a given **tree_id**. The update increases the **unique_identifier** and **parent_id** (if it exists for the table in question) to their current values plus a given offset, and **manager_id** to a given value. (See the *balance* method for more information on how these PreparedStatement are used.)

setCurrentManagers sets the user-specified Managers that have RTs that could be balanced. This method first retrieves all **manager_id**s in user-specified tables, then retrieves all **manager_id**s given in the user configuration file. For each user-specified **manager_id**, the following exceptional conditions are checked, with the appropriate application response:

- Blank or duplicate **manager_id**: warn user with console message, but continue execution. Blank or duplicate values are ignored.
- User-specified **manager_id** not in list of **manager_id**s found in the specified tables: display error message and throw exception which aborts application.

If no exceptional conditions were found, the **manager_id** is added to the ArrayList of **manager_id**s (*currentManagerIDs*) available to all methods in the class. In addition, a new Manager object is created, and the value of its ManagerID field is set to the value of the current **manager_id**. This Manager object is also added to the ArrayList of current and desired Managers (*Managers*) available to all methods in the class. If no valid **manager_id**s were found, an exception is thrown and the application aborts.

setDesiredManagers: This method sets the Manager to which the user would like the RTs in the specified tables and current Managers to be moved. This method checks for the blank or duplicate **manager_id**, and warns the user if one is found. The application otherwise continues, however, and blank or duplicate values are ignored. If the **manager_id** was not blank or a duplicate, it is added to the *desiredManagerIDs* ArrayList, which is available to all methods in the class. If in addition the **manager_id** was not in the list of current Managers, a new Manager object is created, and added to the Managers ArrayList.

setCurrentLoads sets the number of movable record trees and records associated with each Manager. To determine this, all specified tables are queried, and the total number of movable RT IDs and records is retrieved and stored in the appropriate Manager object.

setNumMovableRecordTrees and *setNumMovableRecords*: These methods set the total number of movable Record Trees and records, by adding the RT/records associated with each manager to a running total. The resulting values are stored in the variables totalNumMovableRTs and totalNumMovableRecords, such that they're available to all methods in the class. totalNumMovableRecords is displayed by *balance* at the start of balancing. totalNumMovableRTs is used by *logStatus* to determine when to display a status message, and in calculation of the values used in the message string.

setDesiredLoads sets the desired RT load for each Manager object. Since the final load should be distributed evenly among all desired Managers, the final load for each Manager is determined by dividing the number of movable Record Trees by the number of desired Managers. The desired load for each desired Manager is then set to this value. Managers not desired remain at their initialization value of zero. If the number of Managers does not divide evenly into the number of movable record trees, 1 is added to the desired load of each Manager (starting with the first) until there is no remainder.

setMaxUniqueIDs and *setMaxUniqueID*: These methods set the maximum Unique ID for all Managers. *setMaxUniqueIDs* simply calls *setMaxUniqueID* for each Manager. *setMaxUniqueID* sets the maximum unique identifier for a Manager. Unlike other methods in this class, setMaxUniqueID **does** consider nonmovable records. This is done in order to preserve the uniqueness of the unique_identifier, version_id, and manager_id fields between all table records in the database. *setMaxUniqueID* obtains the maximum **unique_identifier** by sending the *retrieveMaxUniqueID* PreparedStatement to the database for execution.

These methods were split in order to provide a performance increase, as *updateManagerInfo* must set a maximum **unique_identifier** after RTs have been moved from one Manager to another, but only needs to do so for the destination Manager (*setMaxUniqueID*) and not all Managers (*setMaxUniqueIDs*).

logURL, *logTableNames*, and *logManagerInfo*: These three methods are called after all data has been gathered, but just before balancing. They allow the user to verify all specified databases, tables, and Managers are correct before balancing begins. *logURL* displays the database URL, *logTableNames* displays the table names on which balancing will be performed (so the user-specified tables), and *logManagerInfo* displays all current and desired Managers. After the LoadBalancer class has called these methods, the user is given the option to verify all values and continue. If the user elects to continue, records are balanced per the displayed data.

balance: By the time this method has been called, all appropriate information has been gathered from the user and database. Most important is the name and makeup (i.e., having a **parent_id** or not) of each table to be considered, and the current and desired load of each Manager to be considered. Armed with this information, *balance* actually balances the load, moving RTs from undesired current Managers to desired Managers as appropriate.

The algorithm executes inside a while loop, and is as follows: if the current and desired loads of any Manager are not equal to each other (as determined by *isBalanced*), this method searches through the list of Managers for one with too many RTs (current load greater than desired load). When one is found, *balance* then searches for a Manager with too few RTs (desired load greater than current load). It then takes the minimum of these differences. *balance* then uses the retrieveRTs PreparedStatement created by *createPreparedStatements* to retrieve the (minimum) number of RT IDs as calculated previously, as well as all tables containing records associated with that RT. For each RT ID and table combination, *balance* then calls *updateRecordTree*, which updates each record in the tree.

updateRecordTree's specific mechanism is explored fully in its entry, but for now it should be noted that it works by batching and periodically committing the appropriate database update statements. As update statements are committed only when the variable MAX_BATCH_SIZE (set to 100) is reached, it is possible that some update statements created by *updateRecordTree* are batched but not executed. To obviate this, *balance* executes and commits all leftover update statements after *updateRecordTree* has been called all RTs to be moved.

Finally, starting and ending Manager current load and maximum unique id information is updated (*updateManagerInfo*), as well as the number of RTs moved (as saved in the class-global variable *treesMoved*) and the process repeats until current and desired loads are equal for all Managers.

updateRecordTree changes the records associated with a given RT in a given table from its old Manager to its new one. The updateRT PreparedStatement associated with the table in question is first retrieved, then populated with the ending **manager_id** and its maximum **unique_identifier**, and the ID of the RT to be moved. If the table in question contains the **parent_id** field, the maximum **unique_identifier** is again added. The resulting update statement is as follows (values inserted by updateRecordTree are in italics):

UPDATE *tableName*
SET unique_identifier = unique_identifier + 1 + *maxUniqueIdentifier*,
manager_id = *endManagerID*,
parent_id = DECODE(parent_id, 0, 0, NULL, 0, parent_id + 1 + *maxUniqueIdentifier*)
WHERE tree_id = *treeID*

If the table does not contain the **parent_id** field, line four does not exist in the query. The arguments of the SQL DECODE statement are DECODE(field, criteria1, result1, criteria2, result2, …). This statement is required since an RT root record may have a value of either 0 or NULL for its **parent_id**, and this value should not be changed even if moved to a new Manager. If the **manager_id** of the record is 0 or NULL, then, its **manager_id** is updated to 0.

If the **parent_id** is not zero, both it and the **unique_identifier** are updated to their current values plus one plus the maximum **unique_identifier** for the new manager. As a constant value is added to both the **unique_identifier** and **parent_id** (and this value isn’t updated until all RTs to be moved between one manager and another are moved), tree parent-child relationships are preserved. Since the **unique_identifier** and **parent_id** of any records moved to the new Manager are always greater than the maximum **unique_identifier** currently associated with that manager, **manager_id**/**unique_identifier**/**version_id** uniqueness is not violated.

Each update statement is added to a table-specific batch. When the batch size reaches MAX_BATCH_SIZE (set to 100), all update statements in that batch are executed and committed.

updateManagerInfo: This method updates information in the starting and ending Managers involved in a move. This method adds the number of record trees moved to the current load of the ending Manager, and subtracts that number from the starting Manager. With a call to *setMaxUniqueID*, it also updates the maximum **unique_identifier** of the ending Manager.

isBalanced determines whether all user-specified tables have been balanced. This method iterates through all Manager objects. If current RT load equals desired RT load for each Manager, isBalanced returns true, and false otherwise.

logStatus: This method is used to periodically display execution status. At the start of each iteration of *balance*’s while loop, *logStatus* is called. If the number of RTs moved has been changed since the last time logStatus was called, the percentage of total RTs moved is recalculated. That percentage, along with the starting and ending **manager_id**s and number of RTs being moved is displayed.

4. Performance

The database on which the application is to run handles many thousands of updates and queries per hour. In addition, in order to preserve tree relationships and verify non-desired Managers truly have no RTs associated with them when the application has finished, the database must be taken offline for this application to run. As the database is likely only to be taken offline in order that the application may execute, application performance is critical. It was determined early on that each query and update statement were the main performance bottleneck, especially those in *balance*, *updateRecordTree*, and *updateManagerInfo*. Various efforts were made to both reduce the number of queries made by these methods, and reduce the cost of each.

One of the primary methods used in increasing performance was the use of PreparedStatements, which allowed most of a query to be stored in the database. When Java required the query, it sent only values specific to that particular instantiation (**unique_identifier**, **tree_id**, etc) to the database, reducing query time. In any case where a query or update statement was repeated more than a few times, PreparedStatements were used to store the query.

Along the same lines, database indexes were also used to increase performance. As they were primary keys for their tables, **manager_id**, **unique_identifier**, and **version_id** were already indexed. It was found, however, that creating an index on (**manager_id**, **live**) increased performance when searching for all live RTs corresponding to a certain **manager_id** at the start of *balance*. In addition, an index on **tree_id** was also helpful in retrieving all records corresponding to a specific RT in *updateRecordTree*. These indexes are created when test tables are used (see the Testing section), but it was desired that the customer version of the application not change database structure. It was therefore recommended that the customer create these indexes for all balanceable tables upon customer delivery and installation.

Batch execution of database UPDATE statements were also used to increase performance. For each table, UPDATE statements were inserted into a batch using the PreparedStatement class' *addBatch* method. When the batch limit was reached, all UPDATE statements for that table were executed and committed at once. Various values of the batch limit were tried; 100 yielded the highest performance and so was the value used.

One solution that was attempted multiple times, in various forms, was concurrent execution. Both manually creating threads and specific Java classes and packages (ThreadPool, ThreadFactory) were tried. PreparedStatements, however, can only be assigned to one thread at a time. After various threading attempts, it was determined that no threading approach was faster than the single-threaded version finally decided upon.

It was determined early on that in order to preserve uniqueness within a Manager, only one of the two remaining required uniqueness fields (**unique_identifier**, **version_id**) had to be checked when moving records to the Manager. The decision was therefore made to determine the maximum **unique_identifier** of any Manager to which records were about to be added, and require that the **unique_identifier** of the new records be greater than that of the existing ones. The **version_id** field therefore did not need to be checked, and indeed the application does not update **version_id** (or **parent_version_id**) of any records.

One potential issue with the final configuration of the application is its treatment of **unique_identifier**s (and hence **parent_id**s). As the current maximum **unique_identifier** is added to each new set of records moved to the Manager, it is conceivable that the Manager might eventually reach the upper limit of its data type. (This is made more likely due to the fact that only the maximum **unique_identifier** of ending Managers are updated after a move.) In the target database, that data type is number(10), for a maximum value of 9,999,999,999. In Java, the *long* data type was used to store maximum **unique_identifier**s; that maximum is 9,223,372,036,854,775,807. **unique_identifier**s of new records in the target database begin 0, however, and are always positive. In addition, customer experience with the database showed that it would not hold more than a few million records. The assumption was therefore made that reaching a maximum value was unlikely enough that the current configuration would be acceptable. If unique_identifiers ever did get close to a maximum, it was further noted that it would be possible to take the database offline, and write and execute an application that would move RTs to values of **unique_identifier** closer to 0.

The performance of the final version of the application was deemed sufficient for customer delivery. Performance values for various numbers of record tree creations are as follows:

Record Tree Creations	Time (s)	Time (hr)
5,000	4	0
10,000	6	0
50,000	35	0.01
100,000	77	0.02
300,000	308	0.09
500,000	745	0.21
1,000,000	4270	1.19

Table 1: Application execution time for various numbers of Record Tree creations.

Each RT corresponds to approximately 2.7 movable records. The Testing section of this Report can be referenced for detailed information on RT testing configuration. Information in the previous table is presented graphically below:

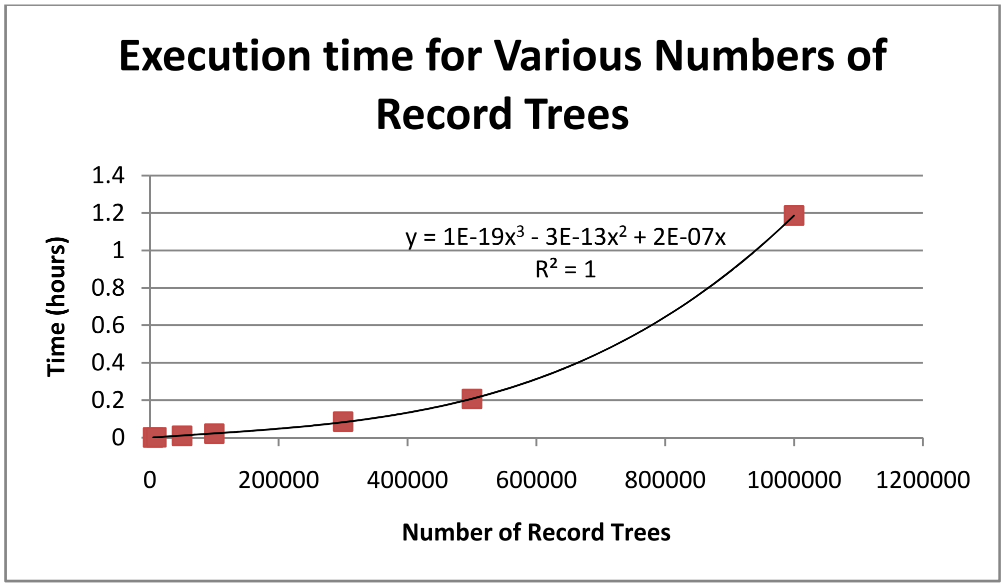

Figure 2: Execution Time for Various Numbers of Record Trees.

Polynomial regression produced the highest trendline R^2 values. Of the various orders, 2 gave a value of $R^2 = .9981$. All orders higher than 3 (up to 6, the maximum in the regression application used) gave values of 1. It is therefore assumed that the application runs in polynomial time in the number of records, with an estimated order of at least 3 ($O(n^3)$).

5. Testing

Introduction

As the application was designed using test-driven agile methods, and since application time performance was critical, testing was an integral part of the development process. In order to test the application, however, test database tables and records were required. Two types of records were created: premade, and randomly generated.

The following tables and record types are created, for both premade and randomly generated records:

- **root1** and **root2**: These tables contain only RT root records, though the trees with which they are associated may have only one record (the root record), and may or may not be live. The **root2** table was created to verify that the application can successfully handle situations where root records were in more than one table.
- **child1**: This table contains children of records in the **root1** and **root2** tables.
- **child2**: This table contains records that may be children of records in **child1**, **root1**, and **root2** tables.
- **not_in_tablenames**: This table resembles the previous ones, but is not included with the other tables in the user configuration file. **not_in_tablenames**, and the records within it, is included to verify that the application will not change records in tables not specified by the user.
- **no_child**: Unlike the previous, this table does not contain **parent_id** or **parent_version_id** fields. It is included to verify the application can successfully process RTs containing only one record.

This table parent/child configuration gives a maximum tree height of three. Although the application can accept database table names of any value, association of certain types of records with certain tables for testing purposes was deemed acceptable for the following reasons:

- Table generation and testing could be greatly simplified if the names of tables were set, and were associated with specific types of records.
- Record/table association mimicked customer database design.

Both premade and randomly-generated records are created using the TablesCreator class. This class' purpose is to instantiate an object of the Tables class, and call its *createTables* method. The constructor for a Tables object contains the following attributes (an alternate constructor for premade records only contains only the first two):

- *isPremade*: **true** for premade records, and **false** for randomly generated ones. If this value is **true**, all values for subsequent attributes after *test* are ignored.
- *test*: if **true**, an extra **test_id** field is created in each table. Each new record is assigned a unique **test_id**, which begins at 1 and is incremented for each record. This field is designed to allow comparison of individual records before and after

application execution. It is set to **false** when performance is tested.

- *numRoots*: the total number of RT roots (and therefore Record Trees) to be created.
- *numRootCreates*: Java ArrayLists are used in the creation of new RTs, and ArrayLists do not have infinite size. With standard values of the other variables (discussed below), setting *numRoots* to 100,000 was found to be a good ceiling on the number of roots that can be created at one time. When a greater number of RTs are desired (as used in performance testing, for example), *numRoots* is set to one divisor of the number of RTs desired, and *numRootCreates* is set to the other. To create 500,000 RTs, then, *numRoots* is set to 100,000, and *numRootCreates* is set to 5.
- *numManagers*: the total number of Managers that the created records may be associated with.
- *maxNumChild1*: The maximum number of **child1** records that can be associated with its parent record (so one in **root1** or **root2**).
- *maxNumChild2*: The maximum number of **child2** records that can be associated with its parent record (so one in **root1**, **root2**, or **child1**).
- *livePerDead*: The ratio of roots with values of ‘T’ for their **live** fields to those with values of ‘F’. This, then, is also the value of live (considered during execution) versus not-live (not considered) RTs across all tables.

Once a Tables object has been constructed, TablesCreator calls Tables’ *createTables* method, which actually constructs the tables. *createTables* first drops all existing tables with the same names as those created, then creates new versions of those tables. As with all other file-contained information, table DDL statements are stored as XML, in the *defineTables.xml* file.

Premade Records

A set of premade records were used during initial testing of the application, or after any significant changes. If the value of *isPremade* was **true** when the Tables object was constructed, premade records are created. This set of records is small (63 records), and is designed to populate all tables, with various configurations of Managers and Record Trees. Identifiers for each relevant field were chosen to be as similar as possible without violating uniqueness, to test whether the application could successfully handle those types of values. In addition, values of records in **not_in_tablenames** were deliberately chosen to match those in the other tables (and hence violate uniqueness), in order to verify the application would not consider these records. Each record, organized in tree form, is given below. Inside of each node is the node’s table location abbreviation, as defined below:

Table name	Abbreviation
root1	R1
root2	R2
child1	C1
child2	C2
not_in_tablenames	NIT
no_child	NC

Table 2: Table name abbreviations for each table.

This is followed in the node by the record's **unique_identifier** and **version_id**. The **tree_id** of each tree appears above each tree, and its **manager_id** below. Also below each tree is its **live** designation: 'T' for live trees and 'F' for trees that aren't live.

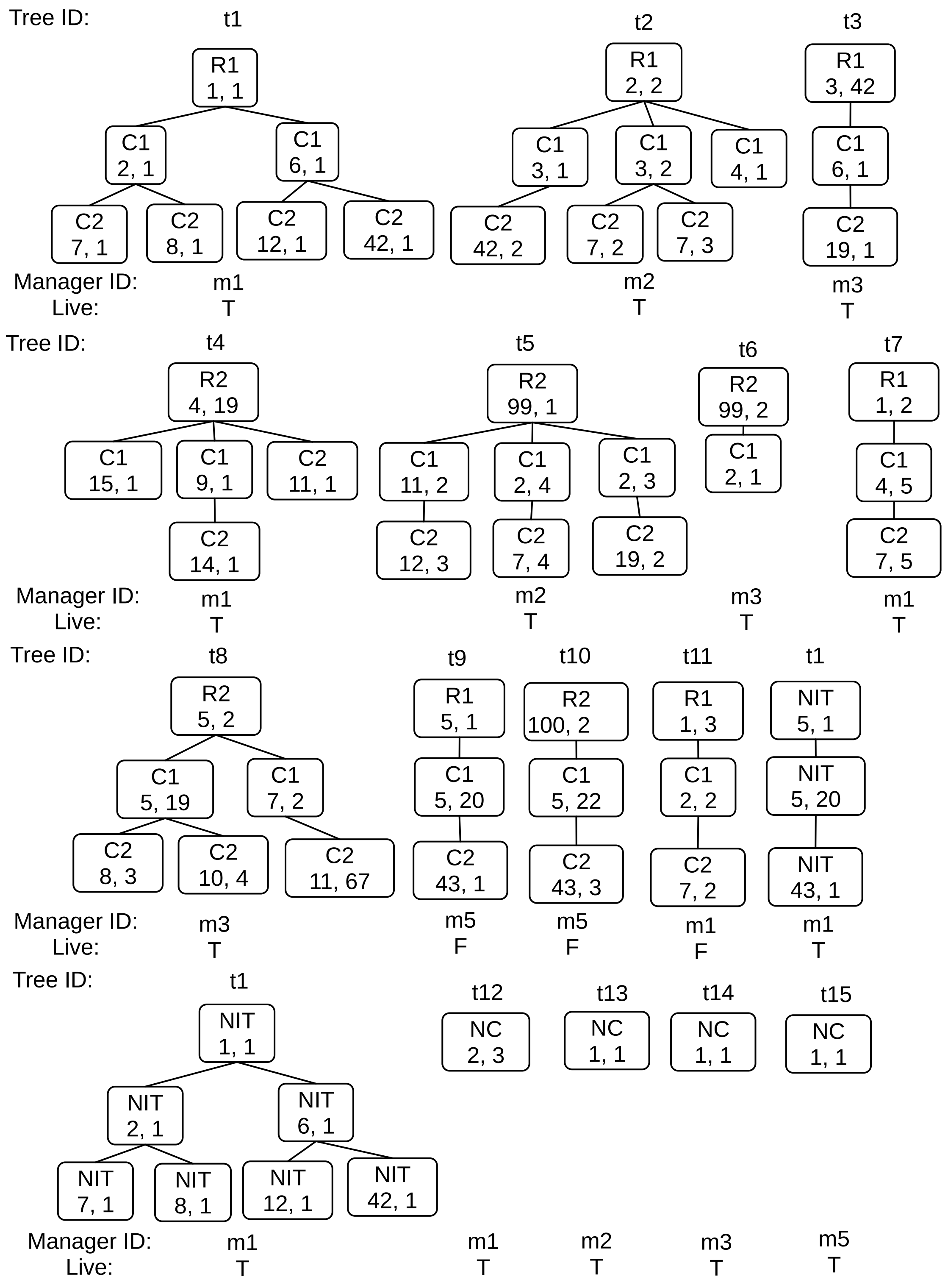

Figure 3: Premade Records.

Randomly Generated Records

If *isPremade* is **false**, (pseudo)randomly generated records are created. Records are created in terms of their Record Trees, with *numRoots * numRootCreates* new root records being first created. Each record is first randomly assigned to a certain table (**root1**, **root2**, **no_child**, or **not_in_tablenames**), with equal probability. Its **manager_id** (and therefore the **manager_id** of the rest of the RT) is then set, as one of the possible values "mN", where N: 1 ... *numManagers*.

The **unique_identifier** and **version_id** is next assigned. For the first record created, **unique_identifier** and **version_id** are assigned a value of 1. After this record, and for each subsequent record, either **unique_identifier** or **version_id** (the choice is randomly determined, with equal probability) is incremented, and the resulting two values are assigned to the next record created. (If the root is assigned to the not_in_tablenames table, the **unique_identifier** and **version_id** are deliberately not incremented, to verify that the application does not read those records.) To facilitate this process, the **manager_id** and current maximum values of **unique_identifier** and **version_id** are stored in an instance of the TablesManager class, one per Manager.

The **tree_id** for the root record is then set, as "tN", where N is 1 … *numRoots * numRootCreates* (so “t1”, “t2” …). For root records, 0 is used for both the **parent_id** and **parent_version_id**. The root is assigned a value of **live** = 'F' (default value 'T'), at a 1/(*livePerDead* + 1) probability. Finally, if *test* is true, a unique-across-all-tables Test_ID is assigned (starting at 1, and going to *numRoots * numRootCreates*).

Child records are then created, for all roots except for those in **no_child**. **child1**-type records are created first. The number of amendments created for each root is random, between 0 and *maxNumChild1*. For each amendment record, its **manager_id**, **tree_id**, and **live** values are taken from its parent (root) record. The latest value of **unique_identifier** and **version_id** are then assigned, as for root records. The **parent_id** and **parent_version_id** for the new record is set to that of the **unique_identifier** and **parent_version_id**, respectively, of its root record. If test is true, a **test_id** is added as before. Records created here are placed in the **child1** table, except for those whose parents are in **not_in_tablenames**, which are also placed in **not_in_tablenames**. Additionally, records created here that will go into **not_in_tablenames** do not have their UID and VID incremented, again to verify that the application does not read those records. Root records in **no_child** are not given **child1** children.

child2-type records are then created, in much the same way as **child1** records. **child2**-type records may be children of either a root or **child1** record, and there may be from 0 to *maxNumChild2* records (again randomly selected) for each parent. For each error record, its **manager_id**, **tree_id**, and **live** values are taken from its parent record. As before, the latest values of **unique_identifier** and **version_id** are used and randomly incremented for creation of the next record. (Also as before, **not_in_tablenames** records are not incremented.) **parent_id** and **parent_version_id**, and **test_id** are set as with **child1**-type records..

Once created, each record is stored in an instance of TablesRecord. When all records have been created, or the number of TablesRecord instances equals *numRoots*, all records are inserted. Similar to *updateRecordTree*, record INSERT statements are stored as PreparedStatements, and inserted as a batch of MAX_BATCH_SIZE statements (also currently set to 100).

Though various values of *numManagers*, *maxNumChild1*, *maxNumChild2s*, and *livePerDead* were used during testing, values used for most testing of LoadBalancer and TablesGroup coalesced at (3, 3, 3, 4). This translates to three Managers, 0 to 3 amendment-type records per root record, 0 to 3 error-type records per root or amendment record, and 80% of records having values of "T" for **live**. As noted, this translates to a number of balanceable records equal to approximately 2.7 times the number of *numRoots*.

Once all records have been created and inserted, indexes on **tree_id** and (**manager_id**, **live**) are created, per the Performance section of this Report.

Unit Testing

Testing is performed using a JUnit testing suite class (AllTests.java). Related test cases were separated into appropriate classes: TableTest.java, LoadBalancerTest.java, and TableGroupTest.java. Two helper classes, ManagerTest.java, and RecordTest.java, are also used. As database records are read into a Java ArrayList for testing, the value of *numRoots* is not set higher than 100,000 during unit testing.

TableTest.java

The TableTest class tests the database tables created by the Tables class. It does this by reading field values from each record in all created tables into an ArrayList, then analyzing those records. Specifically, records are tested against the following requirements:

- All **unique_identifier** and **version_id** > 0 (not a requirement for an actual table, but useful as a first test to verify no negative values were assigned).
- All records in root tables (**root1** and **root2** here) have **parent_id** = **parent_version_id** = 0.
- All records in non-root tables (**child1** and **child2** here) have nonzero **parent_id**s and **parent_version_id**s.
- For each Manager, no **unique_identifier/version_id** combinations are repeated.
- All records in an RT have a unique **tree_id**, and either are root records, (with **parent_id** = **parent_version_id** = 0), or are related to the root record or its children through parent/child relationships. (The **parent_id** and **parent_version_id** of the record in question equal the **unique_identifier** and **version_id** of exactly one other record in the RT.)
- User-specified table names are correct (**root1**, **root2**, **child1**, and **child2**).
- The number of created **tree_id**s equal the number specified.
- For each **tree_id**, the **manager_id** and **live** value for all records with that **tree_id**

is the same.

- The number of records from all tables equal the number created (as determined by the final value of **test_id**).
- The number of distinct **tree_id**s equals the number created (per *numRoots*).
- The number of distinct **manager_id**s equals *numManagers*.
- Both **live** = true- and false-valued records should exist.

The TableTest class contains two testing methods: *testPremadeTable* and *testRandomTable*. *testPremadeTable* creates tables with premade tables, and tests them against the above criteria. *testRandomTable* does the same with randomly-generated records. Since randomly-generated records may not create problems detectable by testing suites, the table- and record-creation and testing process is done ten times for each run of TableTest.java.

TableGroupTest.java

The TableGroupTest class tests the (pre-balance) information-gathering methods of the TableGroup class. For each test, it implements a manually-created database, and uses a standard configuration file containing correct database connection information, the required tables, three existing **manager_id**s, and three desired **manager_id**s different than the current ones. The following is tested using the methods of this class:

- All records are associated with the specified table names.
- Tables that should contain the **parent_id** and **parent_version_id** fields, do.
- The number and names of Managers is correct, for both current and desired Managers.
- The current load of each Manager is correct.
- The number of live RTs is correct.
- The total number of records is correct.
- The desired load of each Manager is correct.
- The maximum **unique_identifier** associated with each Manager is correct.

LoadBalancerTest.java

This class tests the LoadBalancer application as a whole. This class contains various methods to set up the appropriate tables and records, retrieve data, balance loads, and tear down tables. Most importantly, however, this class contains two methods, *testRecords* and *testManagers*. *testRecords* tests each record in the database against record-level requirements for load balancing. Specifically, the following requirements are tested:

- Pre- and post-balance table name, **tree_id**, **version_id**, **parent_version_id**, and **live** values are the same.
- If record was not live, or was not specified as a table to balance, all other new values equal old ones.
- The pre- and post-balance values of the **parent_id** for each root record are the same.

- Parent-child relationships (through the **unique_identifier**/**parent_id** and **version_id**/**parent_version_id** fields) should be preserved.
- If table has no **parent_id**, the value of old and new **parent_id** and **parent_version_id** for any record in it equal the RecordTest object initialization values of -1.
- The combination of the **manager_id**, **unique_identifier**, and **version_id** fields are unique throughout all records in user-specified tables.
- The number of records retrieved is greater than zero.

testManagers tests each Manager in the database against Manager-level requirements for load balancing. Specifically, the following are tested:

- Pre- and post-balance desired loads are equal.
- Undesired Managers have zero post-balance current loads.
- The current and desired loads of desired Managers are equal.

In addition, this class contains methods to conduct tests using individual user configuration files, at one file per method. For each test method, both premade and randomly-generated records are created. 48 different files are tested, evaluating application execution for cases ranging from incorrect configuration file names to all existent tables and current Managers specified, with different desired Managers.

RecordTest.java

The RecordTest class stores pre- and post-balance values for each record. This class is used by LoadBalancerTest.java to test that loads have been correctly balanced, and that the appropriate record, record tree, Manager, and parent/child relationships have been preserved.

ManagerTest.java

This class stores information used by LoadBalancerTest.java in testing of the LoadBalancer class. It consists of pre- and post-balance versions of current and desired loads for each Manager.

6. Conclusions

Initial design of the application commenced in May 2008. By early June, the application appeared to satisfy all but performance requirements. The author at this point began to work with other software engineers in the company and conduct research in an effort to determine how performance might be improved. This process culminated in a code review with all pertinent software engineers in the company in early July.

Once the program appeared to satisfy correctness requirements, the author also began creating a testing environment for the application. The eventual result of this process was a full test suite that was able to evaluate program correctness and performance for both premade and randomly-generated table records, of arbitrary number.

By August 2008, testing had shown that the application satisfied all performance and correctness requirements. It was then delivered to the chief software engineer of the company for final approval and delivery to the customer. As far as the author is aware, that is still the current status of the application.

Final value achieved was significant. The author certainly gained valuable understanding of and experience in “real-world” software development. Perhaps most importantly, however, the customer received what it asked for: a database record-tree load balancing application that satisfied all correctness and performance requirements.

There are two possible directions for future work. The first would reduce the likelihood of a possible integer overflow error in the application. As noted in the Performance section of this Report, RTs moved to a desired Manager have their **unique_identifier**s updated to their current values plus the maximum **unique_identifier** currently in the desired Manager. This guarantees **manager_id**, **unique_identifier**, and **version_id** uniqueness at best performance, but at the possible cost of the **unique_identifier** more quickly nearing the maximum value of **unique_identifier** as specified by the database or Java. A future add-on to the application, would, for each record in a Manager, change its **unique_identifier** to the smallest positively-valued unused one in that manager, and appropriately update its relationship to its parent and child(ren) in the tree.

Another possible direction for future work would be to deal with unmoved RTs in non-desired Managers. Other customer applications work with Managers, and it is desired that Managers themselves be “deactivated” once all records are moved from it. The application, however, only moves all live records in user-specified tables associated with a Manager. This is per user requirements, but it would be useful to move all non-live RTs, and perhaps those in non-specified tables, to another Manager such that a non-desired Manager can be completely deactivated.

7. Bibliography

1. **Atkins, John.** *An Oracle 10g Release 2 Tutorial and Reference.* Fairmont, WV : ManTech International Incorporated: Enterprise Integration Center (e-IC), 2007.
2. dom4j - Introduction. *dom4j.* [Online] October 6, 2007. http://www.dom4j.org/.
3. Eclipse. *Eclipse.org home.* [Online] February 14, 2008. http://www.eclipse.org/.
4. **Gűlcű, Ceki.** *log4j: The complete manual.* Lausanne : QoS.ch, 2004.
5. **Horstmann, Cay.** *Java Concepts, Fourth Edition.* Hoboken : John Wiley and Sons, Inc., 2005.
6. JUnit. *JUnit.org.* [Online] February 12, 2008. http://www.junit.org.
7. Log4j. *Apache Log4j 1.2.* [Online] September 7, 2007. http://logging.apache.org/log4j/1.2/index.html.
8. **Sanjay Mishra, Alan Beaulieu.** *Mastering Oracle SQL.* Sebastopol, CA : O'Reilly, 2004.
9. **Sun Microsystems, Inc.** API Specification. *Java 2 Platform, Standard Edition, v 1.4.2.* [Online] Febuary 11, 2008. http://java.sun.com/j2se/1.4.2/docs/api/.
10. **Sun Microsystems, Inc.** Class Thread. *Java 2 Platform SE v1.3.1.* [Online] 2001. [Cited: November 26, 2001.] http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html.
11. **Sun Microsystems, Inc.** Executors. *Java 2 Platform SE 5.0.* [Online] 2004. [Cited: November 26, 2008.] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executors.html.
12. **Sun Microsytems, Inc.** Interface PreparedStatement. *Java 2 Platform, Std. Ed. v1.4.2.* [Online] July 1, 2008. [Cited: July 1, 2008.]
13. **Vajhøj, Arne.** Re: JDBC PreparedStatement in a multi-threaded environment. *Der Keiler Coding.* [Online] November 16, 2008. [Cited: November 26, 2008.] http://coding.derkeiler.com/Archive/Java/comp.lang.java.programmer/2008-11/msg01541.html.

Appendix

Sample configuration File

```
<config>
   <currentManagers>
      <ID>d1</ID>
      <ID>d2</ID>
      <ID>d3</ID>
   </currentManagers>
   <desiredManagers>
      <ID>d4</ID>
      <ID>d5</ID>
      <ID>d6</ID>
   </desiredManagers>
   <tables>
      <name>Amendments</name>
      <name>Errors</name>
      <name>Others</name>
      <name>Transactions</name>
      <name>no_child</name>
   </tables>
   <databaseInfo>
      <driver>oracle.jdbc.driver.OracleDriver</driver>
      <url>jdbc:oracle:thin:@localhost:1521:xe</url>
      <id>jmack</id>
      <password>vgh4vb4</password>
   </databaseInfo>
</config>
```

Sample Output

```
Load balancer starting: using configuration file defaultConfig.xml
The following tables contain the requisite fields for balancing, but are not
included in the list of tables to balance:
not_in_tablenames
Do you wish to continue? (Enter ‘y’ to do so.) y
Database URL: jdbc:oracle:thin@localhost:1521:xe
Database tables on which balancing will be performed:
root1
root2
child1
child2
no_child
ManagerID: m1, current load: 4, desired load 0
ManagerID: m2, current load: 3, desired load 0
ManagerID: m3, current load: 3, desired load 0
ManagerID: m4, current load: 0, desired load 4
ManagerID: m5, current load: 0, desired load 3
ManagerID: m6, current load: 0, desired load 3
Do you wish to perform this operation? (Enter ‘y’ for yes.) y
Load balance 0% complete: currently moving 4 records from Manager m1 to
Manager m4
Load balance 36% complete: currently moving 3 records from Manager m2 to
Manager m5
Load balance 64% complete: currently moving 3 records from Manager m3 to
Manager m6
ManagerID: m1, current load: 0, desired load 0
ManagerID: m2, current load: 0, desired load 0
ManagerID: m3, current load: 0, desired load 0
ManagerID: m4, current load: 4, desired load 4
ManagerID: m5, current load: 3, desired load 3
ManagerID: m6, current load: 3, desired load 3
Load balancer ended
Load balancer run time 1 second(s)
```

LoadBalancer.java

```
package com.LoadBalancer;

import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Scanner;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.dom4j.DocumentException;

/**
 * The LoadBalancer class balances the database RT (record tree) load between
 * user-specified Managers in a Table Group (TG), using the methods of the
 * TableGroup class.
 *
 * @author Jonathan Mack
 */

public class LoadBalancer
{
   /** Instantiation of log4j logging object. */
   static Logger logger = Logger.getLogger(LoadBalancer.class);

   /**
    * Balances the loads between user-specified Managers. To accomplish this,
a
    * new TableGroup object is first created. Methods within TableGroup then
    * open and parse an XML configuration file, and retrieve associated
    * database information. If there are database tables that contain all the
    * fields needed for balancing, but aren't specified by the user, the user
    * is notified of this, and asked if they want to continue. If so, the
user
    * is shown the database URL and names of tables and Managers to be
    * balanced, along with the Manager RT loads, both before and after
    * balancing. If the user decides to do so, RT loads are then balanced
among
    * specified Managers, such that unmovable RTs are not moved, all movable
    * RTs are moved from specified current to specified desired Managers, and
    * the ending number of desired records minus number of current RTs for
each
    * Manager is zero.
    *
    * @param args
    *            Array of user arguments. args[0] contains the path to the
    *            location of the user's config file. args[1] is optional; if
it
    *            exists, it contains the name of the user's config file. If
it
    *            is null, the default file (defaultConfig.xml) is used.
    */
   public static void main(String[] args)
   {
      // get start time for program execution
      long startTime = System.currentTimeMillis();
      // if no path specified, abort execution
      if (args.length == 0)
      {
         throw new IllegalArgumentException(
            "Must specify directory location of config files");
      }

      // get path and config file name (if it exists); configure logger
      String path = "";
      String fileName = "defaultConfig.xml";
      path = args[0];
      if (args.length == 2)
      {
         fileName = args[1];
      }
      PropertyConfigurator.configure(path + "log4jConfig.txt");

      logger.info("Load balancer starting: using configuration file "
         + fileName);

      TableGroup tg = new TableGroup();
      try
      {
         // get database table information
         tg.parseConfig(path + fileName);
         tg.openConnection();
         tg.setTables();

         // check for balanceable tables in database that aren't in
         // user-specified list
         ArrayList<String> notInTableNames = new ArrayList<String>();
         notInTableNames = tg.checkTables();
         Scanner in = new Scanner(System.in);

         // non-specified balanceable tables found; determine if user wants
         // to continue
         if (notInTableNames.size() > 0)
         {
            logger
               .info("The following tables contain the requisite fields for "
                  + "balancing, but are not included in the list of tables to 
"

                  + "balance: ");

            // display non-specified balanceable tables
            for (String s : notInTableNames)
            {
               logger.info(s);
            }
            logger.info("Do you wish to continue? (Enter 'y' to do so.) ");

            // exit if user wishes to abort
            if (!in.next().toLowerCase().equals("y"))
            {
               logger.info("Load balancer aborting per user request: "
                  + " no records will be changed");
               return;
      }
   }

   // get remainder of load balance information
   tg.setParentID();
   tg.createPreparedStatements();
   tg.setCurrentManagers();
   tg.setDesiredManagers();
   tg.setCurrentLoads();
   tg.setNumMovableRecordTrees();
   tg.setNumMovableRecords();
   tg.setDesiredLoads();
   tg.setMaxUniqueIDs();

   // log pre-balance database info to console
   tg.logURL();
   tg.logTableNames();
   tg.logManagerInfo();

   // verify user wants to continue
   logger.info("Do you wish to perform this operation? "
      + "(Enter 'y' for yes.) ");
   if (!in.next().toLowerCase().equals("y"))
   {
      logger.info("Load balancer aborting per user request: "
         + " no records will be changed");
      return;
   }

   tg.balance();
   tg.logManagerInfo();
   tg.closePreparedStatements();
   tg.closeConnection();
   logger.info("Load balancer ended");
   logger.info("Load balancer run time: "
      + (System.currentTimeMillis() - startTime) / 1000
      + " second(s)");

}
// catch errors
catch (ClassNotFoundException e)
{
   e.printStackTrace();
}
catch (SQLException e)
{
   e.printStackTrace();
}
catch (DocumentException e)
{
   e.printStackTrace();
}
catch (IllegalArgumentException e)
{
   e.printStackTrace();
}
// close database connection objects
```

```
        finally
        {
           try
           {
              tg.closePreparedStatements();
           }
           catch (Exception e)
           {
           }
           try
           {
              tg.closeConnection();
           }
           catch (Exception e)
           {
           }
        }
     }
  }
```

TableGroup.java

```
package com.LoadBalancer;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Vector;

import org.apache.log4j.Logger;
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Node;
import org.dom4j.io.SAXReader;

/**
 * The TableGroup class allows the balancing of record trees (RTs) attached
to
 * various Managers (Managers) within a database of tables. This group of
tables
 * is known as a Table Group. Each RT consists of a tree of records uniquely
 * identified by their Manager ID, Unique Identifier, and Version ID. Each RT
is
 * also associated with exactly one RT ID. Parent/child relationships (if
they
 * exist; some trees contain no children) are determined by the use of a
Parent
 * ID and Parent Version ID, which correspond to the Unique ID and Version ID
of
 * the parent record. Records in each RT are associated with exactly one
 * Manager. Finally, all records have a boolean value in a field known as
 * live ('live', for the purposes of this documentation). All records
 * associated with a certain record tree have the same value of live.
 * <p>
 * The user supplies an XML configuration file which contains information on
the
 * connection to the database where the RT records are stored, the tables
that
 * contain records that may be moved, the ManagerIDs from which records may
be
 * moved (current Managers), and the ManagerIDs in which the user desires the
 * records to be placed (desired Managers). Desired ManagerIDs may include
some,
 * all or none of ManagerIDs currently used, and may include ManagerIDs that
are
 * not associated with any record.
 * <p>
 * NOTE: As load balancing involves moving records from one Manager to
another,
 * and this only entails (for the purposes of this package) changing the
 * Manager_id, unique_id, and (if it exists) parent_id values in a record,
 * "balance", "change" and "move" are used interchangeably throughout this
 * documentation. Also, 'movable', for the purposes of this documentation,
means
 * any record or RT that is live and is associated with user-specified tables
 * and current ManagerIDs.
 * <p>
 * Methods in this class obtain the configuration information, and "move" RTs
by
 * changing the Manager_IDs of their records, pursuant to the following
 * requirements:
 * <ul>
 * <li>Parent-child relationships are preserved.</li>
 * <li>The composite of Manager_ID, Version_ID, and Unique_Identifier remains
 * unique throughout all tables.</li>
 * <li>All records in a particular RT have the same Manager.</li>
 * <li>All records in a Manager that isn't desired by the user are moved to
 * one(s) that is/are, except for those RTs which aren't live or are in
tables
 * or current Managers not specified by the user.</li>
 * <li>After balancing, the RTs are balanced such that the desired number of
 * RTs associated with a desired Manager (its 'load') is equal to its current
 * load.</li>
 * </ul>
 * Manager and Table are helper classes used by TableGroup methods to store
 * appropriate information.
 *
 * @author Jonathan Mack
 */

class TableGroup
{
   /** log4j Logger object used to log information. */
   static Logger logger = Logger.getLogger(TableGroup.class);
   /** Maximum size of updateRTs PreparedStatement batches. */
   private static final int MAX_BATCH_SIZE = 100;
   /**
    * List of user-specified database table names containing the records the
    * user wants to balance.
    */
   private ArrayList<String> tableNames;
   /**
    * List of Managers to be considered when balancing, drawn from both the
    * list of current and desired Managers in the user's config file.
    */
   private ArrayList<Manager> Managers;
   /**
    * User-specified list indicating Managers that may be considered when
    * balancing loads.
    */
   private ArrayList<String> currentManagerIDs;
   /**
    * User-specified list indicating the Managers in which the user wants the
    * RTs in the list of current Managers and table names to reside after
    * balancing.
    */
   private ArrayList<String> desiredManagerIDs;
   /**
    * XML Document object containing user-specified database connection,
table,
    * current Manager, and desired Manager information.
    */
   private Document config;
   /**
    * Connection object used to connect to the database, used by multiple
    * methods in the class.
    */
   private Connection conn;
   /** Number of record trees that have been moved. */
   private long treesMoved = 0;
   /**
    * Total number of record trees that can be moved. Includes only live RTs
    * found in user-specified tables and current Managers.
    */
   private long totalNumMovableRTs = 0;
   /**
    * Total number of records that can be moved. Includes only live records
    * found in user-specified tables and current Managers.
    */
   private long totalNumMovableRecords = 0;
   /** Displayed percent of total record trees moved. */
   private int percentMoved = 0;
   /** Displayed ID of Manager from which RTs are being moved. */
   private String startManager;
   /** Displayed ID of Manager to which RTs are being moved. */
   private String endManager;
   /**
    * Variable used to improve performance when the effective level of the
    * logger is not DEBUG. Used especially to guarantee that logging loops
will
    * not be traversed.
    */
   private final boolean debug = logger.isDebugEnabled();
   /**
    * PreparedStatement to retrieve all record trees corresponding to a
certain
    * Manager.
    */
   private PreparedStatement retrieveRTs;
   /**
    * PreparedStatement to retrieve the maximum unique_identifier associated
    * with a certain Manager.
    */
   private PreparedStatement retrieveMaxUID;
   /** Map from table name to Table object. */
   private Map<String, Table> tables = new HashMap<String, Table>();

   /** Constructs a new TableGroup object. */
   public TableGroup()
   {
      tableNames = new ArrayList<String>();
      Managers = new ArrayList<Manager>();
      currentManagerIDs = new ArrayList<String>();
      desiredManagerIDs = new ArrayList<String>();
      totalNumMovableRTs = 0;
   }

   /**
    * Opens the user's XML configuration file, and parses the information
    * inside.
    *
    * @param fileName
    *            the name of the configuration file
    * @throws DocumentException
    */
   public void parseConfig(String fileName) throws DocumentException
   {
      if (debug) logger.debug("Parsing configuration file");

      SAXReader reader = new SAXReader();
      config = reader.read(fileName);
   }

   /**
    * Opens the connection between Java and the appropriate database tables
    * using information found in the user's config file. If the file contains
    * bad/missing data, this method prints an appropriate message, and throws
    * an appropriate exception.
    *
    * @throws ClassNotFoundException
    * @throws SQLException
    * @throws NullPointerException
    */
   public void openConnection() throws ClassNotFoundException, SQLException
   {
      logger.info("Opening database connection");

      try
      {
         // read database connection info
         String driverName = config.selectSingleNode(
            "/config/databaseInfo/driver").getStringValue();
         String url = config.selectSingleNode("/config/databaseInfo/url")
            .getStringValue();
         String id = config.selectSingleNode("/config/databaseInfo/id")
            .getStringValue();
         String password = config.selectSingleNode(
            "/config/databaseInfo/password").getStringValue();

         // get driver connection
         Class.forName(driverName);
         conn = DriverManager.getConnection(url, id, password);
         conn.setAutoCommit(false);
      }
      // catch errors, append appropriate description, and throw
      catch (NullPointerException e)
      {
         throw new NullPointerException(
            "Could not find all database connection information in
configuration file");
      }
      catch (ClassNotFoundException e)
      {
         throw new ClassNotFoundException(
            "Database driver name incorrect in configuration file");
      }
      catch (SQLException e)
      {
         throw new SQLException(
            "Database connection information (url, id, or password) incorrect
in configuration file");
      }
   }

   /**
    * Returns the database connection.
    *
    * @return the database connection object
    * @throws SQLException
    */
   public Connection getConnection() throws SQLException
   {
      return conn;
   }

   /**
    * Closes the database connection.
    *
    * @throws SQLException
    */
   public void closeConnection() throws SQLException
   {
      if (conn != null) conn.close();
   }

   /**
    * Sets the user-specified names of tables that have RTs that could be
    * balanced. This method checks for table names not in the specified
    * database URL, and blank and duplicate table names. If the table names
    * aren't in the database, or if no valid table names were found, an
    * exception is thrown. If the table name was valid, a new Table object is
    * also created, and added to the <tableName, Table> 'tables' Map.
    *
    * @throws SQLException
    * @throws IllegalArgumentException
    */
   @SuppressWarnings("unchecked")
   public void setTables() throws SQLException
   {
      logger.info("Setting table names");

      // get list of tables in database
      Statement stmt = null;
      ResultSet rs = null;
      ArrayList<String> dbTableNames = new ArrayList<String>();
      try
      {
         stmt = conn.createStatement();
         rs = stmt.executeQuery("SELECT table_name FROM user_tables");
         String dbName = "";
         while (rs.next())
         {
            dbName = rs.getString(1).toLowerCase();
            dbTableNames.add(dbName);
         }
      }
      // catch database errors
      catch (SQLException e)
      {
         e.printStackTrace();
         throw new SQLException();
      }
      // close ResultSet and Statement objects
      finally
      {
         try
         {
            if (rs != null) rs.close();
         }
         catch (Exception e)
         {
         }
         try
         {
            if (stmt != null) stmt.close();
         }
         catch (Exception e)
         {
         }
      }

      // read table names in from user's XML file
      List<Node> namesList = new Vector<Node>();
      namesList = config.selectNodes("/config/tables/name");
      String fileName = "";
      // check each table name: if not blank, a duplicate, or not in
database,
      // add it to the list
      for (Node n : namesList)
      {
         fileName = n.getStringValue().toLowerCase();
         // check for blank table names, warn but continue if found
         if (fileName.length() == 0)
         {
            logger.info("Blank table name element in config file: "
               + "element ignored");
         }
         // check for table names in config file, but not in database
         else if (!dbTableNames.contains(fileName))
         {
            throw new IllegalArgumentException("Table name '" + fileName
               + "' specified as existing in config file not found in "
               + "database");
         }
         // check for duplicate table names
         else if (tableNames.contains(fileName))
         {
            logger.info("Duplicate table name '" + fileName
               + "' in config file; duplicate will not be added");
         }
         // all requirements met: add table name
         else
         {
            tableNames.add(fileName);
            Table aTable = new Table(fileName);
            tables.put(fileName, aTable);
         }
      }

      // log table names
      if (debug)
      {
         for (String s : tableNames)
         {
            logger.debug("Table name: " + s);
         }
      }

      // check for no valid table names specified; throw exception if so
      if (tableNames.size() == 0)
      {
         throw new IllegalArgumentException(
            "Must specify at least one database table");
      }
   }

   /**
    * Returns the list of database tables that have the requisite fields for
    * load balancing, but aren't included in the user's config file. If the
    * size of this list is greater than zero, LoadBalancer.class warns the
    * user, and asks if they want to continue.
    *
    * @throws SQLException
    */
   public ArrayList<String> checkTables() throws SQLException
   {
      Statement stmt = null;
      ResultSet rs = null;
      ResultSet fieldRS = null;
      Statement fieldStmt = null;
      ArrayList<String> notInTableNames = new ArrayList<String>();

      try
      {
         // get all table names
         stmt = conn.createStatement();
         rs = stmt.executeQuery("SELECT table_name FROM user_tables");
         while (rs.next())
         {
            // get fields in each table
            fieldStmt = conn.createStatement();
            fieldRS = fieldStmt
         .executeQuery("SELECT column_name FROM user_tab_columns "
            + "WHERE table_name NOT LIKE 'BIN$%' AND "
            + "lower(table_name) = '"
            + rs.getString(1).toLowerCase() + "'");

      // add fields to tableFields ArrayList
      ArrayList<String> tableFields = new ArrayList<String>();
      while (fieldRS.next())
      {
         tableFields.add(fieldRS.getString(1).toLowerCase());
      }

      // if table contains all fields needed for balancing, and is not
      // in the list of user-specified tables, add it to the list of
      // tables to return
      if (!tableNames.contains(rs.getString(1).toLowerCase())
         && tableFields.contains("unique_identifier")
         && tableFields.contains("version_id")
         && tableFields.contains("Manager_id")
         && tableFields.contains("root_object_id")
         && tableFields.contains("live"))
      {
         notInTableNames.add(rs.getString(1));
      }
   }
}
// catch SQL exception
catch (SQLException e)
{
   e.printStackTrace();
   throw new SQLException();
}
// close ResultSet and Statement objects
finally
{
   try
   {
      if (rs != null) rs.close();
   }
   catch (Exception e)
   {
   }
   try
   {
      if (stmt != null) stmt.close();
   }
   catch (Exception e)
   {
   }
   try
   {
      if (fieldRS != null) fieldRS.close();
   }
   catch (Exception e)
   {
   }
   try
```

```
      {
         if (fieldStmt != null) fieldStmt.close();
      }
      catch (Exception e)
      {
      }
   }
   return notInTableNames;
}

/**
 * Gets names of tables that have records that could be balanced.
 *
 * @return the ArrayList of table names
 */
public ArrayList<String> getTableNames()
{
   return tableNames;
}

/**
 * Gets the Map of table names to Table objects.
 *
 * @return the Map of table names to Table objects
 */
public Map<String, Table> getTables()
{
   return tables;
}

/**
 * Sets the parentID field in each Table object in the 'tables' Map.
 *
 * @throws SQLException
 */
public void setParentID() throws SQLException
{
   Statement stmt = null;
   ResultSet rs = null;

   // get the field names for each specified database table, and set
   // the parentID attribute of the associated Table object to true
   // if it contains the 'parent_id' field.
   try
   {
      stmt = conn.createStatement();
      for (String s : tableNames)
      {
         rs = stmt
            .executeQuery("SELECT column_name FROM user_tab_columns "
               + "WHERE lower(table_name) = '" + s + "'");
         while (rs.next())
         {
            if (rs.getString(1).toLowerCase().equals("parent_id"))
            {
               tables.get(s).setParentID(true);
            }
            }
         }
      }
      // catch SQL exception
      catch (SQLException e)
      {
         e.printStackTrace();
         throw new SQLException();
      }
      // close ResultSet and Statement objects
      finally
      {
         try
         {
            if (rs != null) rs.close();
         }
         catch (Exception e)
         {
         }
         try
         {
            if (stmt != null) stmt.close();
         }
         catch (Exception e)
         {
         }
      }
      if (debug) logHasParentID();
   }

   /**
    * Creates the PreparedStatements used to speed execution of the methods
in
    * the TableGroup class. Three PreparedStatements are created: one each to
    * find the maximum Unique ID (including nonlive records) and all live
RTIDs
    * associated with a specified Manager, and one to update each RT to its
new
    * values.
    *
    * @throws SQLException
    */
   public void createPreparedStatements() throws SQLException
   {
      // create subquery to retrieve record trees to move
      StringBuffer treeSubBuffer = new StringBuffer();
      treeSubBuffer.append("SELECT tree_id FROM (");
      for (String s : tableNames)
      {
         treeSubBuffer.append("SELECT tree_id FROM " + s
            + " WHERE live = 'T' AND manager_id = ? UNION ");
      }
      treeSubBuffer.delete(treeSubBuffer.length()
         - " UNION ".length(), treeSubBuffer.length());
      treeSubBuffer.append(") WHERE rownum <= ?");

      // create query to retrieve record trees to move
      StringBuffer treeBuffer = new StringBuffer();
      for (String s : tableNames)
      {
         treeBuffer.append("SELECT DISTINCT tree_id, '" + s
            + "' AS table_name FROM " + s + " WHERE tree_id IN (");
         treeBuffer.append(treeSubBuffer.toString());
         treeBuffer.append(") UNION ALL ");
      }
      treeBuffer.delete(treeBuffer.length()
         - " UNION ALL ".length(), treeBuffer.length());

      // create PreparedStatement to retrieve RTs
      retrieveRTs = conn.prepareStatement(treeBuffer.toString());

      // create query to get maximum unique_id
      StringBuffer maxBuffer = new StringBuffer();
      maxBuffer.append("SELECT MAX(unique_identifier) FROM (");
      for (String s : tableNames)
      {
         maxBuffer.append("SELECT unique_identifier FROM " + s
            + " WHERE Manager_id = ? UNION ALL ");
      }
      maxBuffer.delete(maxBuffer.length() - " UNION ALL ".length(), maxBuffer
         .length());
      maxBuffer.append(")");

      // create PreparedStatement to get the maximum Unique ID associated
with
      // a Manager
      retrieveMaxUID = conn.prepareStatement(maxBuffer.toString());

      // create PreparedStatement to update RTs
      for (String s : tableNames)
      {
         // PreparedStatement for tables with the parent_id field
         if (tables.get(s).isParentID())
         {
            PreparedStatement ps = conn
               .prepareStatement("UPDATE "
                  + s
                  + " SET unique_identifier = unique_identifier + 1 + ?,"
                  + " manager_id = ?, parent_id = DECODE(parent_id, 0, 0,
NULL,"
                  + " 0, parent_id + 1 + ?) WHERE root_object_id = ?");
            tables.get(s).setPs(ps);
         }
         // PreparedStatement for tables without the parent_id field
         else
         {
            PreparedStatement ps = conn.prepareStatement("UPDATE " + s
               + " SET unique_identifier = unique_identifier + 1 + ?,"
               + " manager_id = ? WHERE root_object_id = ?");
            tables.get(s).setPs(ps);
         }
      }

   }
   /**
    * Closes the PreparedStatements used in this class.
    *
    * @throws SQLException
    */
   public void closePreparedStatements() throws SQLException
   {
      try
      {
         if (retrieveRTs != null) retrieveRTs.close();
      }
      catch (Exception e)
      {
      }

      for (String s : tableNames)
      {
         try
         {
            if (tables.get(s).getPs() != null)
               tables.get(s).getPs().close();
         }
         catch (Exception e)
         {
         }
      }
      try
      {
         if (retrieveMaxUID != null) retrieveMaxUID.close();
      }
      catch (Exception e)
      {
      }
   }

   /**
    * Sets the user-specified Managers that have RTs that could be balanced.
    * This method checks for manager_ids not in the specified database URL,
and
    * blank and duplicate manager_ids. If the manager_ids aren't in the
    * database, or if no valid manager_ids were found, an exception is
thrown.
    * If the manager_id was valid, the manager_id is added to the
    * currentManagerIDs ArrayList, and a new Manager object is created and
    * added to the Managers ArrayList.
    *
    * @throws SQLException
    * @throws IllegalArgumentException
    */
   @SuppressWarnings("unchecked")
   public void setCurrentManagers() throws SQLException,
      IllegalArgumentException
   {
      logger.info("Setting IDs of current Managers");

      // create query StringBuffer to get list of ManagerIDs in database
// tables
StringBuffer queryBuffer = new StringBuffer();
String subquery = "";
for (String s : tableNames)
{
   subquery = "SELECT DISTINCT Manager_id FROM " + s
      + " WHERE live = 'T' UNION ";
   queryBuffer.append(subquery);
}
queryBuffer.delete(queryBuffer.length() - " UNION ".length(),
   queryBuffer.length());

// execute query and retrieve ManagerIDs in database tables
ArrayList<String> ManagerIDsInTables = new ArrayList<String>();
Statement stmt = null;
ResultSet rs = null;
try
{
   stmt = conn.createStatement();
   rs = stmt.executeQuery(queryBuffer.toString());
   while (rs.next())
   {
      ManagerIDsInTables.add(rs.getString(1));
   }
}
// catch database exceptions
catch (SQLException e)
{
   e.printStackTrace();
   throw new SQLException();
}
// close ResultSet and Statement objects
finally
{
   try
   {
      if (rs != null) rs.close();
   }
   catch (Exception e)
   {
   }
   try
   {
      if (stmt != null) stmt.close();
   }
   catch (Exception e)
   {
   }
}

// add ManagerIDs in config file to lists of all and current ManagerIDs
List<Node> idList = new Vector<Node>();
idList = config.selectNodes("/config/currentManagers/ID");

// add Manager IDs from config file
for (Node n : idList)
{
```

```
         // check for blank ManagerIDs, warn but continue if found
         if (n.getStringValue().length() == 0)
         {
            logger.info("Blank current Manager ID element found in config "
               + "file: element ignored");
         }
         // Specified Manager not in specified tables: abort program
         else if (!ManagerIDsInTables.contains(n.getStringValue()))
         {
            throw new IllegalArgumentException("Manager ID '"
               + n.getStringValue() + "' specified as existing "
               + "in config file not found in specified tables");
         }
         // check to see if Manager ID is a duplicate; discard if so
         else if (currentManagerIDs.contains(n.getStringValue()))
         {
            logger.info("Duplicate current Manager ID '"
               + n.getStringValue()
               + "' in config file; duplicate will not be added");
         }
         // add new Manager ID to list of current Managers
         else
         {
            currentManagerIDs.add(n.getStringValue());
            Managers.add(new Manager(n.getStringValue()));
         }
      }

      // throw exception if no Managers were found
      if (currentManagerIDs.size() == 0)
      {
         throw new IllegalArgumentException(
            "No Managers were specified from which to move records");
      }
   }

   /**
    * Sets the Manager to which the user would like the RTs in the specified
    * tables and current Managers to be moved. This method checks for blank
and
    * duplicate manager_ids. If no valid manager_ids were found, an exception
    * is thrown. If the manager_id was valid, the manager_id is added to the
    * desiredManagerIDs ArrayList. If in addition the manager_id was not in
the
    * list of current manager_ids, a new Manager object is created, and added
    * to the Managers ArrayList.
    *
    * @throws SQLException
    */
   @SuppressWarnings("unchecked")
   public void setDesiredManagers() throws SQLException
   {
      logger.info("Setting desired Managers");

      // get list of desired ManagerIDs from user config file
      List<Node> idList = new Vector<Node>();
      idList = config.selectNodes("/config/desiredManagers/ID");
      // check each ID for null, duplicate
      for (Node n : idList)
      {
         // check for blank ManagerIDs, warn but continue if found
         if (n.getStringValue().length() == 0)
         {
            logger.info("Blank desired Manager ID element found in config "
               + "file: element ignored");
         }
         // check to see if Manager ID is a duplicate; discard if so
         else if (desiredManagerIDs.contains(n.getStringValue()))
         {
            logger.info("Duplicate desired Manager ID '"
               + n.getStringValue()
               + "' in config file; duplicate will not be added");
         }
         // add new Manager ID to list of desired Managers
         else
         {
            desiredManagerIDs.add(n.getStringValue());

            // if not in list of Managers, add to list
            if (!currentManagerIDs.contains(n.getStringValue()))
            {
               Managers.add(new Manager(n.getStringValue()));
            }
         }
      }

      // throw exception if no Managers were found
      if (desiredManagerIDs.size() == 0)
      {
         throw new IllegalArgumentException(
            "No Managers were specified to which to move records");
      }
   }

   /**
    * Gets the list of Manager objects.
    *
    * @return the ArrayList of Manager objects
    */
   public ArrayList<Manager> getManagers()
   {
      return Managers;
   }

   /**
    * Gets the list of current Manager_IDs.
    *
    * @return the list of current Manager_IDs
    */
   public ArrayList<String> getCurrentManagerIDs()
   {
      return currentManagerIDs;
   }
```

```
   /**
    * Gets the list of desired Manager_IDs.
    *
    * @return the ArrayList of desired Manager_IDs
    */
   public ArrayList<String> getDesiredManagerIDs()
   {
      return desiredManagerIDs;
   }

   /**
    * Sets the number of movable record trees and records associated with
each
    * Manager. For each Manager, all specified tables are queried, and the
    * total number of movable RTIDs and records is retrieved and stored in
the
    * appropriate Manager object.
    *
    * @throws SQLException
    */
   public void setCurrentLoads() throws SQLException
   {
      logger.info("Setting current loads for each Manager");

      // get ManagerIDs and the number of live record trees in each from
      // tables
      StringBuffer queryBuffer = new StringBuffer();
      queryBuffer
         .append("SELECT Manager_id, COUNT(DISTINCT root_object_id), "
            + "COUNT(root_object_id) from (");
      for (String s : tableNames)
      {
         queryBuffer.append("SELECT Manager_id, root_object_id FROM " + s
            + " WHERE live = 'T' UNION ALL ");
      }
      queryBuffer.delete(queryBuffer.length() - " UNION ALL ".length(),
         queryBuffer.length());
      queryBuffer.append(") GRTUP BY Manager_id");

      // execute query and insert results into Managers
      Statement stmt = null;
      ResultSet rs = null;
      try
      {
         stmt = conn.createStatement();
         rs = stmt.executeQuery(queryBuffer.toString());

         // set each current load
         while (rs.next())
         {
            for (Manager d : Managers)
            {
               if (rs.getString(1).equals(d.getManagerID()))
               {
                  d.setCurrentRTload(rs.getLong(2));
                  d.setCurrentRecordLoad(rs.getLong(3));
               }
            }
         }
      }
      // catch database errors
      catch (SQLException e)
      {
         e.printStackTrace();
         throw new SQLException();
      }
      // close ResultSet and Statement objects
      finally
      {
         try
         {
            if (rs != null) rs.close();
         }
         catch (Exception e)
         {
         }
         try
         {
            if (stmt != null) stmt.close();
         }
         catch (Exception e)
         {
         }
      }
   }

   /**
    * Sets the total number of movable record trees.
    *
    * @throws SQLException
    */
   public void setNumMovableRecordTrees()
   {
      logger.info("Setting total number of movable record trees");

      // add number of movable RTs associated with each Manager to running
      // total
      for (Manager d : Managers)
      {
         totalNumMovableRTs = totalNumMovableRTs + d.getCurrentRTload();
      }

      if (debug)
      {
         logger.debug("Total number of movable record trees: "
            + totalNumMovableRTs);
      }
   }

   /**
    * Gets the total number of movable record trees.
    *
    * @return the total number of movable RTs
```

```
    */
   public long getNumMovableRootObjects()
   {
      return totalNumMovableRTs;
   }

   /**
    * Sets the total number of movable records.
    *
    * @throws SQLException
    */
   public void setNumMovableRecords() throws SQLException
   {
      logger.info("Setting total number of movable records");

      // add number of movable records associated with each Manager to
running
      // total
      for (Manager d : Managers)
      {
         totalNumMovableRecords = totalNumMovableRecords
            + d.getCurrentRecordLoad();
      }

      if (debug)
      {
         logger.debug("Total number of movable records: "
            + totalNumMovableRecords);
      }
   }

   /**
    * Gets the total number of movable records.
    *
    * @return the total number of movable records
    */
   public long getNumMovableRecords()
   {
      return totalNumMovableRecords;
   }

   /**
    * Sets the desired RT load for each Manager object. Since the final load
    * should be distributed evenly among all desired Managers, the final load
    * for each desired Manager is determined by dividing the number of
movable
    * record trees by the number of desired Managers. The desired load for
each
    * desired Manager is then set to this value. Managers not desired remain
at
    * their initialization value of zero. If the number of Managers does not
    * divide evenly into the number of movable record trees, 1 is added to
the
    * desired load of each Manager (starting with the first) until there is
no
    * remainder.
    */
   public void setDesiredLoads()
   {
      logger.info("Setting desired loads for each Manager");

      // If number of desired Managers does not divide evenly into the number
      // of
      // record trees, calculate the remainder so it can be evenly
distributed
      // among as many Managers as possible
      int remainder = (int) (totalNumMovableRTs % desiredManagerIDs.size());

      for (Manager d : Managers)
      {
         // only add remainder to desired ManagerIDs
         if (desiredManagerIDs.contains(d.getManagerID()))
         {
            // desired load = total number of record trees divided by number
            // of desired Managers
            d.setDesiredRTload(totalNumMovableRTs
               / desiredManagerIDs.size());

            // If remainder is greater than zero, add one record tree to the
            // current Manager, and decrement the remainder
            if (remainder > 0)
            {
               d.setDesiredRTload(d.getDesiredRTload() + 1);
               remainder--;
            }
         }
      }
      // initialization value of desiredLoad = 0, so no need to set it for
      // non-desired Managers
   }

   /**
    * Sets the maximum Unique ID for all Managers.
    */
   public void setMaxUniqueIDs() throws SQLException
   {
      // loop through each table and Manager
      for (Manager d : Managers)
      {
         setMaxUniqueID(d);
      }
   }

   /**
    * Sets the maximum unique identifier for a Manager. Unlike other methods
in
    * this class, setMaxUniqueID <strong>does</strong> consider nonmovable
    * records. This is done in order to preserve the uniqueness of the
    * unique_id, version_id, and manager_ID fields between all table records
in
    * the Table Group.
    */
   private void setMaxUniqueID(Manager Manager) throws SQLException
   {
      // execute query to retrieve maximum unique_id for all Managers
      ResultSet rs = null;
      try
      {
         for (int i = 1; i <= tableNames.size(); i++)
         {
            retrieveMaxUID.setString(i, Manager.getManagerID());
         }
         rs = retrieveMaxUID.executeQuery();
         rs.next();
         Manager.setMaxUniqueIdentifier(rs.getLong(1));

      }
      // catch database exceptions
      catch (SQLException e)
      {
         e.printStackTrace();
         throw new SQLException();
      }
      // close ResultSet
      finally
      {
         try
         {
            if (rs != null) rs.close();
         }
         catch (Exception e)
         {
         }
      }

   }

   /**
    * Balances movable RTs among desired Managers, moving them from undesired
    * current Managers as appropriate. If the current and desired loads of
any
    * Manager is different, this method searches for a Manager with too many
    * RTs (current load greater than desired load) and one with too little
    * (desired load greater than current load). It then takes the minimum of
    * these differences, and moves that minimum number of RTs associated with
    * the first Manager to the second, by batching and periodically
committing
    * the appropriate update statements. Manager current load and maximum
    * unique id information is then updated, and the process repeats until
    * current and desired loads are equal for all Managers.
    *
    * @throws SQLException
    */
   public void balance() throws SQLException
   {
      logger.info("Starting load balance: " + totalNumMovableRTs
         + " movable record trees, " + totalNumMovableRecords
         + " movable records");

      // move RTs until all Managers balanced
      int startIndex = 0; // index of Managers ArrayList for prospective
start
      // Manager
      int endIndex = 0; // index of Managers ArrayList for prospective end
      // Manager
      long recordTreesInTransit = 0;
      while (!isBalanced())
      {
         // look for start Manager with RTs to move, and end Managers with
         // space for
         // them
         if (Managers.get(startIndex).getCurrentRTload() > Managers.get(
            startIndex).getDesiredRTload()
            && Managers.get(endIndex).getCurrentRTload() < Managers.get(
               endIndex).getDesiredRTload())
         {
            // get number of RTs to move
            recordTreesInTransit = Math.min(Managers.get(startIndex)
               .getCurrentRTload()
               - Managers.get(startIndex).getDesiredRTload(), Managers
               .get(endIndex).getDesiredRTload()
               - Managers.get(endIndex).getCurrentRTload());

            // display balance status on console
            logStatus(Managers.get(startIndex).getManagerID(), Managers
               .get(endIndex).getManagerID(), recordTreesInTransit);

            // get calculated number of RTs using PreparedStatement
            ResultSet rs = null;
            try
            {
               int i;
               // populate PreparedStatement
               for (int j = 0; j < tableNames.size(); j++)
               {
                  for (i = 1; i <= tableNames.size(); i++)
                  {
                     retrieveRTs.setString(i + j
                        * (tableNames.size() + 1), Managers.get(
                        startIndex).getManagerID());
                  }
                  retrieveRTs.setLong(i + j * (tableNames.size() + 1),
                     recordTreesInTransit);
               }
               // execute PreparedStatement
               rs = retrieveRTs.executeQuery();

               // update each RT retrieved
               while (rs.next())
               {
                  updateRecordTree(rs.getString(1), rs.getString(2),
                     endIndex);
               }

               // commit leftovers in each batch
               for (String s : tableNames)
               {
```

```
            tables.get(s).getPs().executeBatch();
            conn.commit();
            tables.get(s).getPs().clearBatch();
            tables.get(s).setBatchSize(0);
         }
      }
      // catch database errors
      catch (SQLException s)
      {
         s.printStackTrace();
         throw new SQLException();
      }
      // close ResultSet object
      finally
      {
         try
         {
            if (rs != null) rs.close();
         }
         catch (Exception e)
         {
         }
      }

      // update Manager current load info
      treesMoved = treesMoved + recordTreesInTransit;
      updateManagerInfo(Managers.get(startIndex), Managers
         .get(endIndex), recordTreesInTransit);

      if (debug) logManagerInfo();
      if (debug) logDBrecords();
   }
   /*
    * prospective destination Manager either had no room for new RTs,
    * or has just had RTs transferred to it; look at next destination
    * Manager in either case
    */
   endIndex++;
   /*
    * all destination Managers for this starting Manager have been
    * investigated; increment starting Manager, and reset ending
    * Manager to 0
    */
   if (endIndex == Managers.size())
   {
      endIndex = 0;
      startIndex++;
   }
   // both start and destination Manager have reach their maximum value
   // without records having been balanced; reset both to 0 and start
   // over
   if (startIndex == Managers.size())
   {
      startIndex = 0;
      endIndex = 0;
   }
}
```

```
   }

   /**
    * Changes the records associated with a given RT in a given table from
its
    * old Manager to its new one. The appropriate PreparedStatement is first
    * populated, then added to the batch appropriate for that table name.
When
    * any batch reaches MAX_BATCH_SIZE, its contents are executed, committed,
    * and cleared.
    *
    * @param recordTreeID
    *            the ID of the RT being moved
    * @param tableName
    *            the table name where the records being moved reside
    * @param endManagerindex
    *            the index (in {@link #Managers}) of the destination Manager
    *
    * @throws SQLException
    */
   private void updateRecordTree(String recordTreeID, String tableName,
      int endManagerindex) throws SQLException
   {
      // populate the update RT PreparedStatement
      tables.get(tableName).getPs().setLong(1,
         Managers.get(endManagerindex).getMaxUniqueIdentifier());
      tables.get(tableName).getPs().setString(2,
         Managers.get(endManagerindex).getManagerID());

      // add parent_id field value, if exists in this table, add RTID
      // regardless
      if (tables.get(tableName).isParentID())
      {
         tables.get(tableName).getPs().setLong(3,
            Managers.get(endManagerindex).getMaxUniqueIdentifier());
         tables.get(tableName).getPs().setString(4, recordTreeID);
      }
      else
      {
         tables.get(tableName).getPs().setString(3, recordTreeID);
      }

      // add update statement to batch and increment batch size
      tables.get(tableName).getPs().addBatch();
      tables.get(tableName).setBatchSize(
         tables.get(tableName).getBatchSize() + 1);

      // execute, commit, and clear batch if table's batch is greater than
      // maximum
      if (tables.get(tableName).getBatchSize() >= MAX_BATCH_SIZE)
      {
         tables.get(tableName).getPs().executeBatch();
         conn.commit();
         tables.get(tableName).getPs().clearBatch();
         tables.get(tableName).setBatchSize(0);
      }
   }
   /**
    * Updates information in the starting and ending Managers involved in a
    * move. This method adds the number of record trees moved to the current
    * load of the ending Manager, and subtracts that number from the starting
    * Manager. It also updates the maximum unique_identifier of the ending
    * Manager.
    *
    * @param startManager
    *            the Manager from which RTs were moved
    * @param endManager
    *            the Manager to which RTs were moved
    * @param rootObjectsMoved
    *            the number of RTs moved
    * @throws SQLException
    */
   private void updateManagerInfo(Manager startManager, Manager endManager,
      long rootObjectsMoved) throws SQLException
   {
      // subtract number of RTs moved from current RT load of start Manager
      startManager.setCurrentRTload(startManager.getCurrentRTload()
         - rootObjectsMoved);

      // add number of RTs moved to current RT load of destination Manager
      endManager.setCurrentRTload(endManager.getCurrentRTload()
         + rootObjectsMoved);

      // update maximum unique ID of destination Manager
      setMaxUniqueID(endManager);
   }

   /**
    * Determines whether all Managers have been balanced. If current RT load
    * equals desired RT load for each Manager, isBalanced returns true, and
    * false otherwise.
    *
    * @return whether the specified table/Manager combination has been
balanced
    */
   private boolean isBalanced()
   {
      boolean isBalanced = true;
      // if current and desired RT loads for any Manager are different, the
      // table/Manager combination hasn't been balanced
      for (Manager d : Managers)
      {
         if (d.getCurrentRTload() != d.getDesiredRTload())
            isBalanced = false;
      }
      return isBalanced;
   }

   /**
    * Logs information (name, current and desired RT load) for all Managers.
If
    * the logging level is set to debug, the maximum unique ID is also
logged.
```

```
 */
public void logManagerInfo()
{
   // if debugging, log name, current and desired RT load, and maximum
   // unique ID for each Manager
   if (debug)
   {
      for (Manager d : Managers)
      {
         logger.debug("ManagerID: " + d.getManagerID()
            + ", current RT load: " + d.getCurrentRTload()
            + ", desired RT load: " + d.getDesiredRTload()
            + ", maximum UID: " + d.getMaxUniqueIdentifier());
      }
   }
   // if not debugging, log only name and current and desired RT load
   else
   {
      for (Manager d : Managers)
      {
         logger.info("ManagerID: " + d.getManagerID()
            + ", current load: " + d.getCurrentRTload()
            + ", desired load: " + d.getDesiredRTload());
      }
   }
}

/**
 * Logs LoadBalancer-specific record values from all specified tables.
 *
 * @throws SQLException
 */
private void logDBrecords() throws SQLException
{
   logger.debug("Logging records from database");

   Statement stmt = null;
   ResultSet rs = null;
   try
   {
      stmt = conn.createStatement();
      // iterate through each table
      for (String s : tableNames)
      {
         // log DB records from tables that do not have a parent_id field
         if (!tables.get(s).isParentID())
         {
            rs = stmt
               .executeQuery("SELECT unique_identifier, "
                  + "version_id, Manager_id, root_object_id, live FROM "
                  + s);

            logger.debug("Table name: " + s);
            while (rs.next())
            {
               logger.debug("UID: " + rs.getLong(1) + ", versionID: "
                  + rs.getLong(2) + ", ManagerID: " + rs.getString(3)
```

```
                     + ", RTID: " + rs.getString(4) + ", STP_live: "
                     + rs.getString(5));
               }
            }
            // log DB records from tables that don't have a parent_id field
            else
            {
               rs = stmt.executeQuery("select unique_identifier, "
                  + "version_id, Manager_id, root_object_id, parent_id, "
                  + "parent_version_id, live from " + s);

               logger.debug("Table name: " + s);
               while (rs.next())
               {
                  logger.debug("UID: " + rs.getLong(1) + ", versionID: "
                     + rs.getLong(2) + ", ManagerID: " + rs.getString(3)
                     + ", RTID: " + rs.getString(4) + ", PID: "
                     + rs.getLong(5) + ", PVID: " + rs.getLong(6)
                     + ", STP_live: " + rs.getString(7));
               }
            }
         }
      }
      // catch database exceptions
      catch (SQLException s)
      {
         s.printStackTrace();
         throw new SQLException();
      }
      // close ResultSet and Statements objects
      finally
      {
         try
         {
            if (rs != null) rs.close();
         }
         catch (Exception e)
         {
         }
         try
         {
            if (stmt != null) stmt.close();
         }
         catch (Exception e)
         {
         }
      }
   }

   /**
    * Displays the load balancing status. The last displayed starting
Manager,
    * ending Manager, and percent complete are compared against the current
    * values for those variables. If any are different, a new status line is
    * displayed, giving the percentage of total record trees that have been
    * moved, and the starting and ending Manager of record trees that are
    * currently being moved.
```

```
     *
     * @param sManager
     *            the current Manager from which record trees are being moved
     * @param eManager
     *            the current Manager to which record trees are being moved
     */
    public void logStatus(String sManager, String eManager,
       Long numRootObjectsBeingMoved)
    {
       // assign new values to status variables
       int currentPercent = (int) ((double) treesMoved
          / (double) totalNumMovableRTs * 100);
       percentMoved = currentPercent;
       startManager = sManager;
       endManager = eManager;

       // log values
       logger.info("Load balance " + percentMoved
          + "% complete: currently moving " + numRootObjectsBeingMoved
          + " record tree(s) from Manager " + startManager + " to Manager "
          + endManager);
    }

    /**
     * Logs whether each specified table has a parent_id field. The table name
     * is logged, along with a boolean value; true if the table has a
parent_id
     * field, and false if not.
     */
    private void logHasParentID()
    {
       logger.debug("Logging whether table has parent_id:");
       for (String s : tableNames)
       {
          logger.debug("Table name: " + s + ", has parent_id? "
             + tables.get(s).isParentID());
       }
    }

    /**
     * Logs the name of each specified table. Currently designed to be called
     * just before the user is asked whether they wish to engage in the
     * balancing process.
     */
    public void logTableNames()
    {
       StringBuffer nameBuffer = new StringBuffer();
       for (String s : tableNames)
       {
          nameBuffer.append(s + ", ");
       }
       nameBuffer.delete(nameBuffer.length() - ", ".length(), nameBuffer
          .length());
       logger.info("Database tables on which balancing will be performed: "
          + nameBuffer);
    }
```

```
   /**
    * Logs the database URL.
    */
   public void logURL()
   {
      logger.info("Database URL: "
         + config.selectSingleNode("/config/databaseInfo/url")
            .getStringValue());
   }
}
```

Manager.java

```
package com.LoadBalancer;

/**
 * The Manager class models a Manager. Each Manager is
 * associated with multiple record trees (RTs), each with its own unique Root
 * Object ID (RTID). Each RT is composed of one or more records, with the
 * maximum value of unique_id from all those records also stored in the
 * attributes of this class. A record or RT is considered "movable" if it is
 * live (i.e., the value in its live field(s) = "T") and is associated with a
 * user-specified table and current Manager.
 *
 * @author Jonathan Mack
 */
class Manager
{
   /** The Manager ID. */
   private String managerID;
   /**
    * The number of live RTs in user-specified tables associated with this
Manager.
    */
   private long currentRTload;
   /**
    * The number of live records in user-specified tables associated with
this
    * Manager.
    */
   private long currentRecordLoad;
   /**
    * The optimal number of RTs associated with this Manager, such that the
    * difference between current and desired load is zero for all Managers
after
    * balancing.
    */
   private long desiredRTload;

   /**
    * The maximum Unique Identifier of all records associated with this
Manager.
    * Unlike other fields, this includes both movable and nonmovable records.
    */
   private long maxUniqueIdentifier;

   /**
    * Creates a new Manager object, with the Manager name as its
    * parameter.
    *
    * @param aManagerID
    *            the ID of the Manager
    */
   public Manager(String aManagerID)
   {
      managerID = aManagerID;
      currentRTload = 0;
      currentRecordLoad = 0;
   desiredRTload = 0;
   maxUniqueIdentifier = 0;
}

/**
 * Sets the Manager ID.
 *
 * @param ManagerID
 *            the Manager ID
 */
public void setManagerID(String ManagerID)
{
   this.managerID = ManagerID;
}

/**
 * Gets the Manager ID.
 *
 * @return the Manager ID
 */
public String getManagerID()
{
   return managerID;
}

/**
 * Gets the current number of movable RTs.
 *
 * @return the number of movable RTs
 */
public long getCurrentRTload()
{
   return currentRTload;
}

/**
 * Sets the current number of movable RTs.
 *
 * @param currentRTload
 *            the number of movable RTs
 */
public void setCurrentRTload(long currentRTload)
{
   this.currentRTload = currentRTload;
}

/**
 * Gets the current number of movable records.
 *
 * @return the current number of movable records
 */
public long getCurrentRecordLoad()
{
   return currentRecordLoad;
}

/**
```

```
     * Sets the current number of movable records.
     *
     * @param currentRecordLoad
     *            the current number of movable records
     */
   public void setCurrentRecordLoad(long currentRecordLoad)
   {
      this.currentRecordLoad = currentRecordLoad;
   }

   /**
     * Gets the optimum number of movable RTs.
     *
     * @return the optimum number of movable RTs
     */
   public long getDesiredRTload()
   {
      return desiredRTload;
   }

   /**
     * Sets the optimum number of movable RTs.
     *
     * @param desiredRTload
     *            the optimal number of movable RTs
     */
   public void setDesiredRTload(long desiredRTload)
   {
      this.desiredRTload = desiredRTload;
   }

   /**
     * Gets the maximum Unique ID of all records associated with this Manager.
     *
     * @return the maximum Unique ID of all records associated with this
Manager
     */
   public long getMaxUniqueIdentifier()
   {
      return maxUniqueIdentifier;
   }

   /**
     * Sets the maximum Unique ID of all records associated with this Manager.
     *
     * @param maxUniqueIdentifier
     *            the maximum Unique ID of all records associated with this
Manager
     */
   public void setMaxUniqueIdentifier(long maxUniqueIdentifier)
   {
      this.maxUniqueIdentifier = maxUniqueIdentifier;
   }
}
```

Table.java

```
package com.LoadBalancer;

import java.sql.PreparedStatement;

/**
 * The Table class provides objects and methods to store, alter, and retrieve
 * information on the database tables updated by the TableGroup class.
 * Attributes include the following:
 * <ul>
 * <li>The table name</li>
 * <li>Whether or not it has parent_id (and by association parent_version_id)
 * fields</li>
 * <li>The PreparedStatement needed to update record trees (RTs) in this
table</li>
 * <li>The number of unexecuted update statements associated with this
table's
 * PreparedStatement batch</li>
 * </ul>
 *
 * @author Jonathan Mack
 */
public class Table
{
   /** The name of the table. */
   private String tableName;
   /** Whether or not the table contains parent_id and parent_version_id
fields. */
   private boolean parentID;
   /** The PreparedStatement used to update RTs associated with this table.
*/
   private PreparedStatement ps;
   /**
    * The number of unexecuted update statements associated with this table's
    * PreparedStatement batch.
    */
   private int batchSize;

   /** Creates a new table object, with the specified String as its name. */
   public Table(String aTableName)
   {
      tableName = aTableName;
      parentID = false;
      ps = null;
      batchSize = 0;
   }

   /**
    * Gets the table name.
    *
    * @return the table name
    */
   public String getTableName()
   {
      return tableName;
   }
   /**
    * Sets the table name.
    *
    * @param tableName
    *            the table name
    */
   public void setTableName(String tableName)
   {
      this.tableName = tableName;
   }

   /**
    * Returns whether the table contains the parent_id and parent_version_id
    * fields.
    *
    * @return whether the table contains the parent_id and parent_version_id
    *         fields
    */
   public boolean isParentID()
   {
      return parentID;
   }

   /**
    * Sets whether the table contains the parent_id and parent_version_id
    * fields.
    *
    * @param parentID
    *            whether the table contains the parent_id and
parent_version_id
    *            fields
    */
   public void setParentID(boolean parentID)
   {
      this.parentID = parentID;
   }

   /**
    * Gets the PreparedStatement used to update RTs associated with this
table.
    *
    * @return the PreparedStatement used to update RTs associated with this
    *         table
    */
   public PreparedStatement getPs()
   {
      return ps;
   }

   /**
    * Sets the PreparedStatement used to update RTs associated with this
table.
    *
    * @param ps
    *            the PreparedStatement used to update RTs associated with
this
```

```
     *            table
     */
    public void setPs(PreparedStatement ps)
    {
       this.ps = ps;
    }

    /**
     * Gets the number of unexecuted update statements associated with this
     * table's PreparedStatement batch.
     *
     * @return the number of unexecuted update statements associated with this
     *         table's PreparedStatement batch
     */
    public int getBatchSize()
    {
       return batchSize;
    }

    /**
     * Sets the number of unexecuted update statements associated with this
     * table's PreparedStatement batch.
     *
     * @param batchSize
     *            the number of unexecuted update statements associated with
     *            this table's PreparedStatement batch
     */
    public void setBatchSize(int batchSize)
    {
       this.batchSize = batchSize;
    }
 }
```