Introduction

Project Management System 2004

Project Management System 2004 is an organisational tool used by business to aid the internal processing of information. It is able to organise material and human resources within projects, such as personnel, events, files and projects. It is orientated towards time management as well as the effective transfer of information in what could be a large and complicated project.

About Scalability

Scalability is the relationship between the increase in the number of items in the database and the increase in time experienced by a user performing certain functions. In particular, functions involving high orders of n may cause large problems when there are a large number of items in the database. Scalability is important because there are costs associated with poor computer performance in the business world.

Since the system should be able to handle arbitrarily large projects taken on by clients, the question of scalability will inevitably arise, especially due to the complex querying system, sophisticated file-handling capabilities and high-level integration between the different types of resources within projects. Hence, it is important to test scalability in order to see if the project is fit for the consumers for whom it has been designed. Without an analysis of scalability, the set of tasks that this system will be able to perform may be limited to the small scale.

Report Abstract

In normal operation of this business database, we would expect the following operations to be performed regularly:

- Insertion of database items (INSERT queries);
- Selection of database items based on their unique ID or on some other property (SELECT queries), and optionally perform operations on the resultant dataset (UPDATE queries);
- Deletion of database items based on some identifying property (DELETE queries);
- Shutting down and restoring the database from file; and
- Computing statistics on the items stored in the database

Hence, if these functions were to be performed on the database regularly, it is important that the developers are aware of the scalability of the software, so that appropriate adjustments can be made to particular sections of the code base before the client uses it in the (demanding) real world. By running scalability tests on various components of the system, and then extrapolating the results to predict the performance of the system under high load, developers can see at a glance the scalability issues and prioritise development time accordingly.

In this scalability report, the developers have elected to use “Big Oh” notation, which indicates the scalability on the worst-case scenarios, as its asymptotic behaviour is a good

measure of how the database will perform with large data sets experienced in a business environment. Classes that are utilised in database operations are analysed individually; this is followed by a theoretical analysis of the functions listed above, created by combining the results from analysing the classes individually. A model will be created for each of the functions, outlining its behaviour at larger load strains.

Following the theoretical analysis, will be the analysis of the experimental data. The experimental results will be compared against the theoretical predictions, and any potential explanations for discrepancies will be discussed. The models produced in the theoretical section are modified if necessary. Finally, the results are extrapolated to discuss the scalability of the above functions to 100,000 items.

Theoretical Analysis

Core Database Classes

Database

Method Name	*Big Oh Calculations*	*Big Oh*
getItem	O(table.getItem)	O(n)
initialise	O(1)	O(1)
loadBlank	O(1)	O(1)
load	O(n)	O(n)
save	$O(n^2)$	$O(n^2)$
processRequest	O(createQuery + processQuery)	Depends on type of query function

DatabaseTable (includes ProjectManager, EventManager, UserManager, FileManager and AccessManager)

Note: Although the implementation varies from manager to manager, they were determined to all have the same complexity.

Method Name	*Big Oh Calculations*	*Big Oh*
getIDList	O(n)	O(n)
getItem	O(n)	O(n)
toXML	$O(n \times object.toXML)$	$O(n^2)$
add	O(Vector.add)	O(n)
remove	O(Vector.remove)	O(n)
search	$O(n \times DatabaseObject.Get)$ [All the DatabaseObject classes currently use linear search]	$O(n^3)$
getPropertyEnumeration	O(1)	O(1)

All DatabaseObject Classes (includes Project, Event, User, File and AccessControlList)

Note: Although the implementation varies from database object to database object, they were determined to all have the same complexity.

Method Name	*Big Oh Calculations*	*Big Oh*
get Methods	O(authentication)	$O(n^2)$
set Methods	O(authentication)	$O(n^2)$
add Methods	O(authentication + Vector.add)	$O(n^2)$
remove Methods	O(authentication + Vector.add)	$O(n^2)$
toXML()	O(n)	O(n)

All DatabaseFolder Classes

Method Name	*Big Oh Calculations*	*Big Oh*
addChild()	O(Vector.add)	O(n)
removeChild()	O(Vector.remove)	O(n)

Access Control

AccessManager

Method Name	*Big Oh Calculations*	*Big Oh*
authenticate()	Calls isAdministrator() and searches Administrators group list using linear search – O(n), where n is the number of users in the database; checks whether the user's ID is in the specified access control list using linear search – O(n), where n is the number of users; checks all access groups to which the user belongs – O(n x m), where n is the (number of access groups + number of users) and m is the (number of access groups) i.e. O($n + n + n^2$)	O(n^2)

Query classes

The query system is complex to analyse in Big Oh notation; however, these are a few of the values for the execution of different types of queries:

Type of Query	*Big Oh Calculations*	*Big Oh*
Select Query	O(m × DatabaseTable.search) where m = number of variables to test)	O(n^3) – if m is held constant
Insert Query	O(DatabaseTable.add)	O(n)
Update Query	O(select + n × m) where m = number of actions to perform	O(n^3) – if m is held constant
Delete Query	O(select + n × DatabaseTable.remove)	O(n^3)

XML classes

The XML classes are also best described in terms of function.

Function	*Big Oh Calculations*	*Big Oh*
Parsing the document	Each character is analysed once only, as the file is read from beginning to end – O(n) where n is the number of bytes in the file, which is proportional to the number of items in the database; it is then validated, where each item is visited once – O(n) i.e. O($n + n$) = O(n)	O(n)
Getting a sub-element from an element	Linear search – O(n), where n is the number of sub-elements belonging to that element, which in worst case is the number of items in database	O(n)
Adding sub-elements to an element	Addition to Vector – O(n), where n is defined as above	O(n)
Getting an attribute from an element	Linear search – O(n), where n is the number of attributes belonging to that element (it is independent of the size of the database)	O(n)
Adding an attribute to an element	Addition to Vector – O(n), where n is defined as above (independent of database)	O(n)
Converting the database in XML tree form to a String	All database objects can have references to any other database object, so terminals are O(n); folders then collate together all the strings from	O(n^2)

	the terminals – O(n); there is a fixed number of database tables, each of which contain one root folder – O(1) i.e. O(n x n x 1) = O(n^2)	
Restoring the database from an XML tree (ObjectCreator)	O(n × DatabaseTable.add}	O(n^2)

Miscellaneous Classes

IDGenerator

In all cases for IDGenerator.java, n refers to the total number of items in database (noting that the number of IDs in system equals the number of database objects).

Method Name	*Big Oh Notation Working*	*Big Oh*
getUniqueID()	Calls: getUniqueNumber() – O(n); addToArray(long) – O(n + addToArray(int, int, long)) = O(n + log n) = O(n) noting that addToArray(int, int, long) uses binary search; longToID(String) – O(1). i.e. O(n + n + 1) = O(n)	O(n)
addID()	Uses binary search to find location – O(log n) ; internal array may need to be resized – O(n). i.e. O(n + log n) = O(n)	O(n)
getExistingIDs()	Iterates through all items in the array – O(n); adds to Vector with correct internal size O(1). i.e. O(n + 1) = O(n)	O(n)
isIDValid()	Checks the given fixed-length string character by character – O(1)	O(1)
remove()	Traverses the array – O(n); resizes array – O(n); adds to array – O(1) i.e. O(n + n + 1) = O(n)	O(n)
deleteLog()	All operations are O(1)	O(1)

TransactionLog

Method Name	*Big Oh Notation Working*	*Big Oh*
saveToFile	O(1)	O(1)
getLogEntry	O(1)	O(1)
close	O(1)	O(1)
recordQuery	O(1)	O(1)

WildcardServiceProvider

Method Name	*Big Oh Notation Working*	*Big Oh*
match()	O(1)	O(1)

Theoretical Analysis by Function

Insertion

- Consists of insert query O(n), authentication O(n^2), add to database O(n), get ID O(n), transaction log O(1)
- Overall – O(n^2)

Access based on Property or Identifier

- Consists of select query O(n^3), authentication O(n^2), search in DatabaseTable O(n^3), transaction log O(1)
- Overall – O(n^3)

Deletion

- Consists of delete query O(n^3), authentication O(n^2), search in DatabaseTable O(n^3), transaction log O(1), remove in DatabaseTable O(n)
- Overall – O(n^3)

Computing Simple Statistics

- The current count() method in the console user interface relies on getting a list of objects first via select queries. Hence it has the same complexity, i.e. O(n^3).

Saving to Files

- Consists of translating each database object into XML O(n) (n due to cross referencing) × translating each database root folder into XML O(n) (n due to n children) = O(n^2), converting database in XML tree to String O(n^2)
- Overall – O(n^2)

Restoring from Files

- Consists of parsing the document O(n), restoring the database using ObjectCreator O(n^2)
- Overall – O(n^2)

Experimental Analysis

Introduction to Experimental Analysis

The following experimental analysis was carried out to gain an empirical insight into how efficiently the database runs on a personal computer compared to the theoretical Big-Oh predictions derived earlier. The test machine was running Windows 98 using Java SDK 1.4.1, on a AMD Athlon 1800 XP+ 1599MHz processor and 256 MB RAM.

The method that will be used in order to experimentally test the scalability of the database will involve a testing script that will run methods in a particular order and record the amount of time that it takes for each method to run, these methods will be able to be told to run X amount of iterations and each of these iteration's execution time will be recorded. This data will be output to a CSV file which can then be imported into Microsoft Excel for analysis and processing.

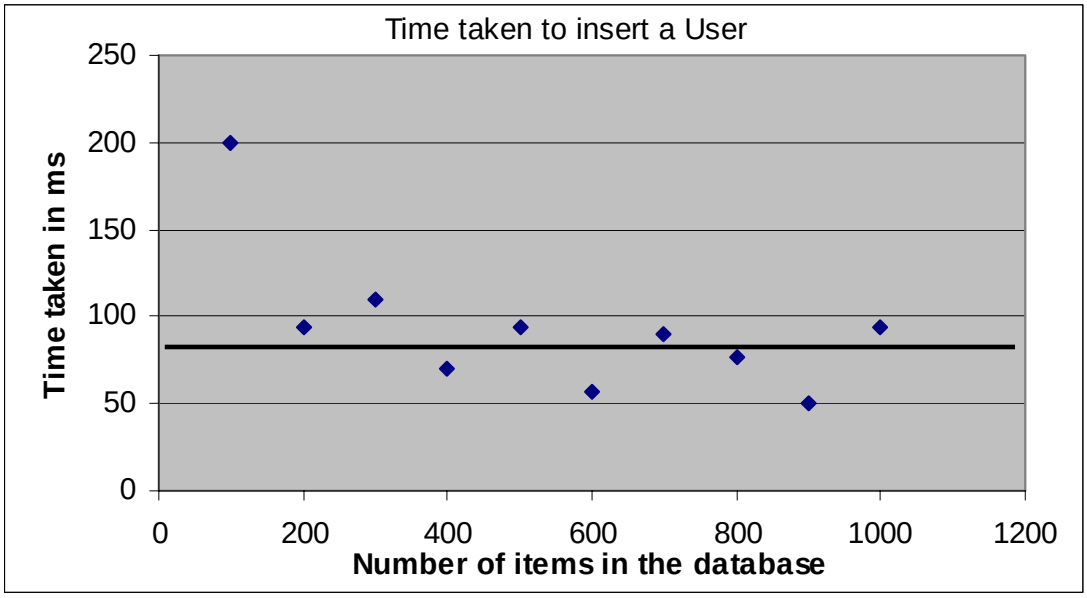
To test the insertion of items each object will be added in increments of a hundred, and the time it takes for 100 items to be added will be recorded, this will be recorded for up to 1000 items and then the process will be repeated again in order to develop an average execution time at each level. This will be carried out for each type of object.

To test the searching and deletion of each type of object, items will be added to the database and then some searches and some deletions will be run. Then the system will add another increment of items and the process will continue. This will also be carried out for each type of item.

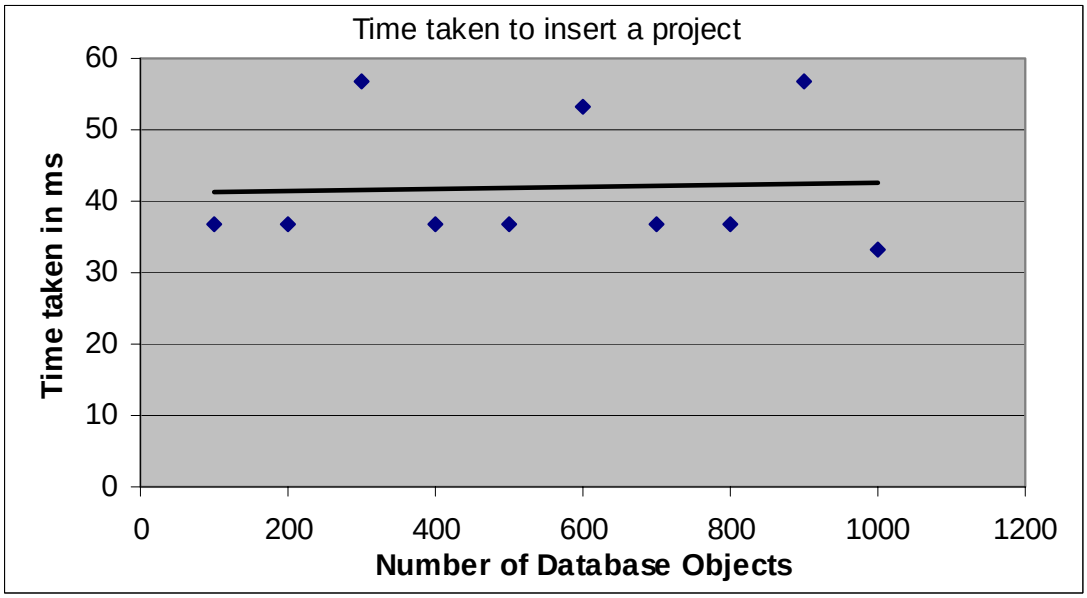
To test the loading and saving functionality, the database will be asked to insert a number of items then save them to a file, the database will then be asked to load the file again, then more items will be added. This process will continue up to 2500 items. Each load and save will occur 10 times.

Warning

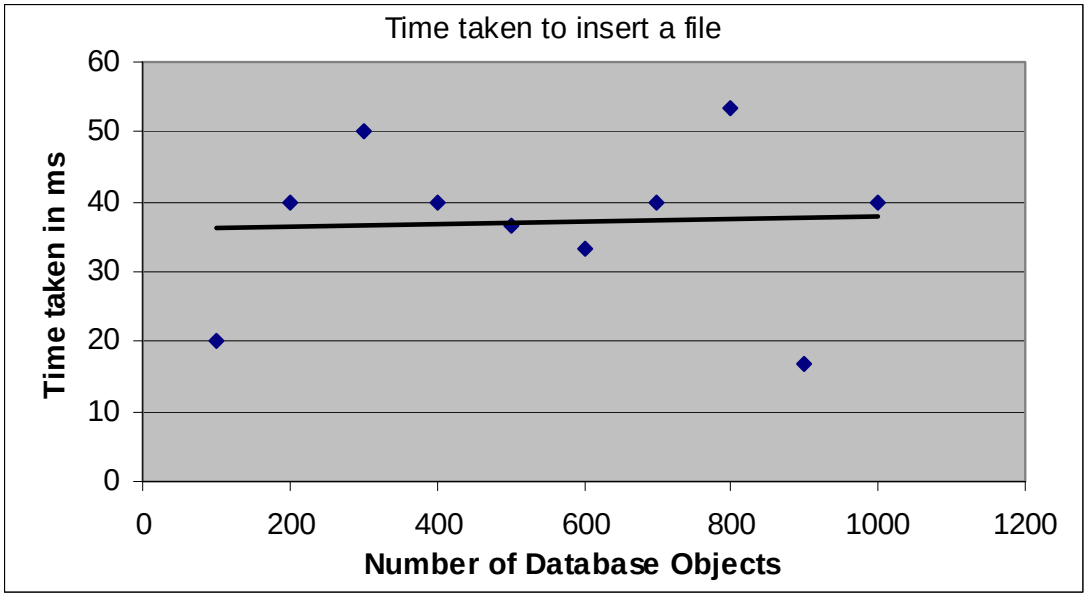
It is predicted that for many functions predicted to be n^2, the increase observed when we use a timer will be linear, because the `AccessManager.authenticate()` method is n when validating Administrators and n^2 when validating everyone else. We ran the tests as Administrator.



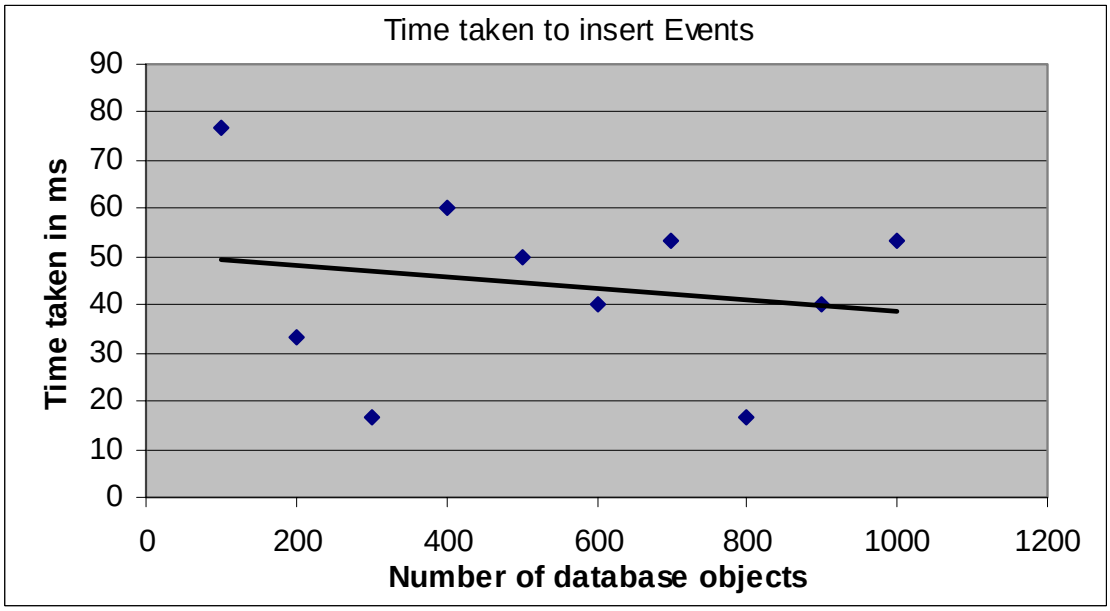
The trendline for user insertion is very clearly a horizontal line, signifying an independent relationship with the number of objects in the database. The initial surge may be because of the Java Virtual Machine initialising since User insertion comes first. Otherwise, ignoring the relatively small fluctuations it can be safely stated that user insertion is O(1). It was predicted that insertion is O(n^2) – however, we added as Administrator (see note above), and for Java Vectors, O(n) is worst case that only occurs when it resizes.



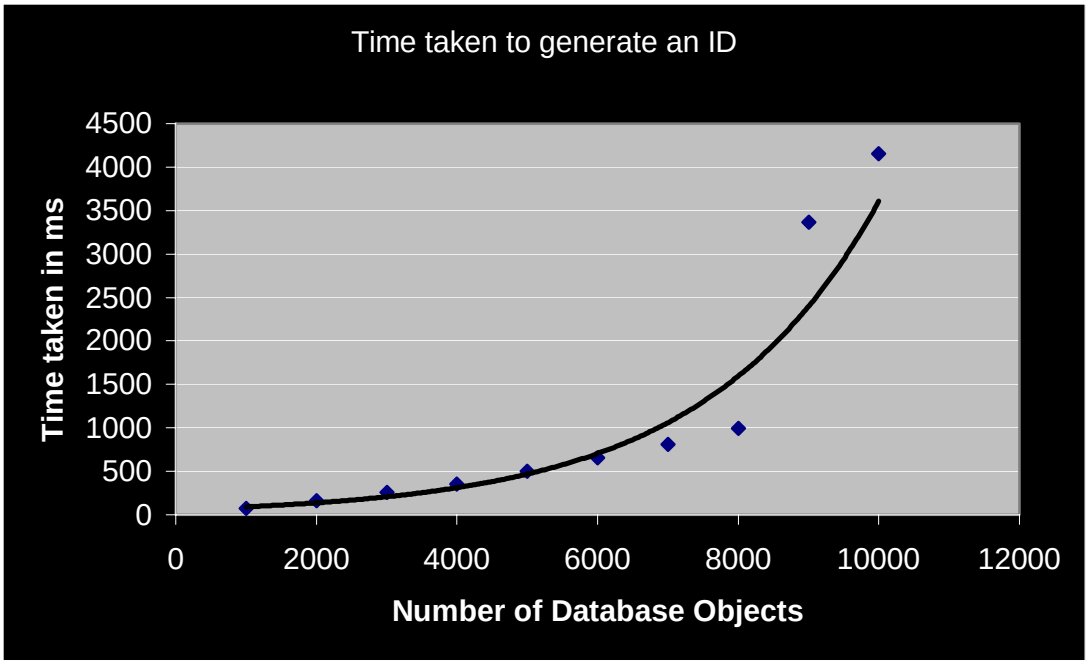
Although the graph for the insertion of a project seems much more fluctuating than the insertion of users, it is important to realise that the scale of this graph is different. There is a lesser dependence on many O(1) methods and thus methods that give a more random result, such as IDGenerator and other factors such as memory allocation and the java virtual machine may have a greater impact. However overall it is seen that the general trend is horizontal, also meaning that the time for insertion is independent of the number of objects in the database.



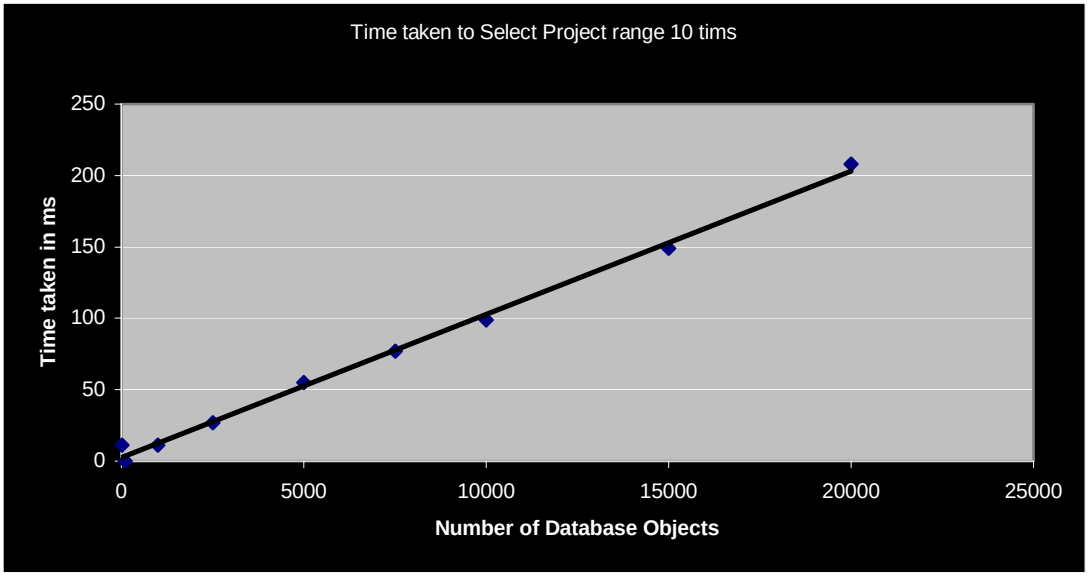
The trendline for project insertion is also horizontal, also leaning towards its being independent to the number of items within the database. From similar arguments for insertion of projects, we can account for the fluctuations.

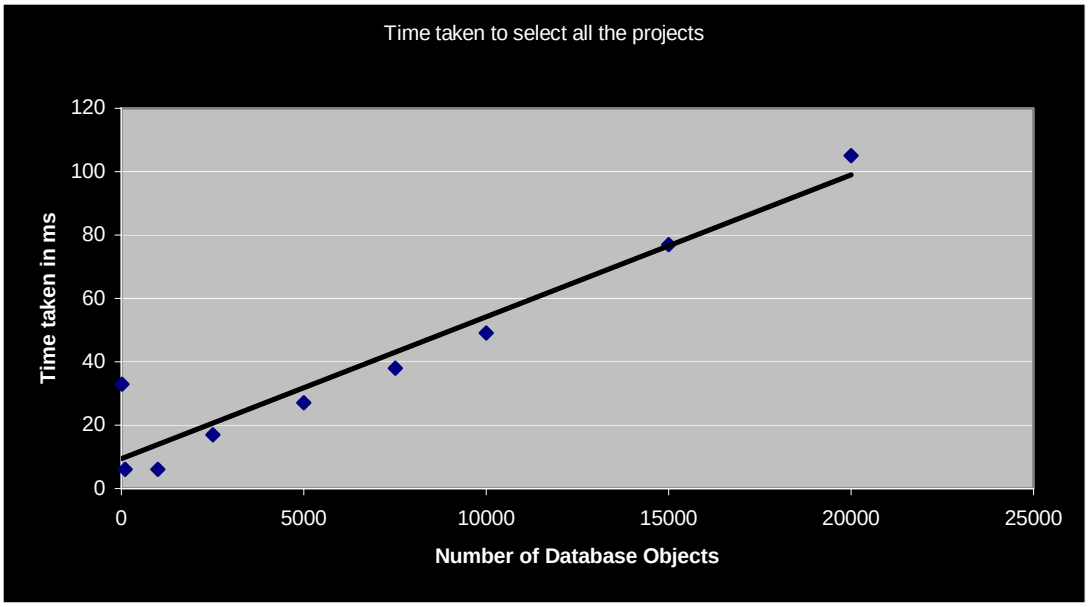


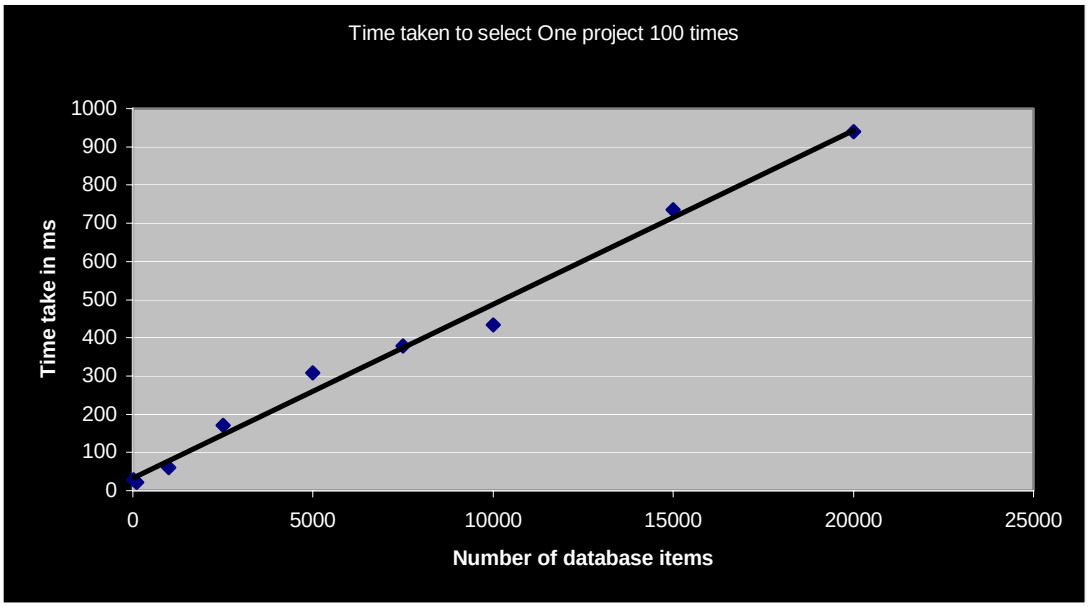
The trendline for event insertion is generally horizontal, also leaning towards its being independent to the number of items within the database. The decreasing gradient should be explained by the pervasiveness of the randomness rather than a decrease in insertion time in relation with number of objects in database. From similar arguments for insertion of projects, we can account for the fluctuations.



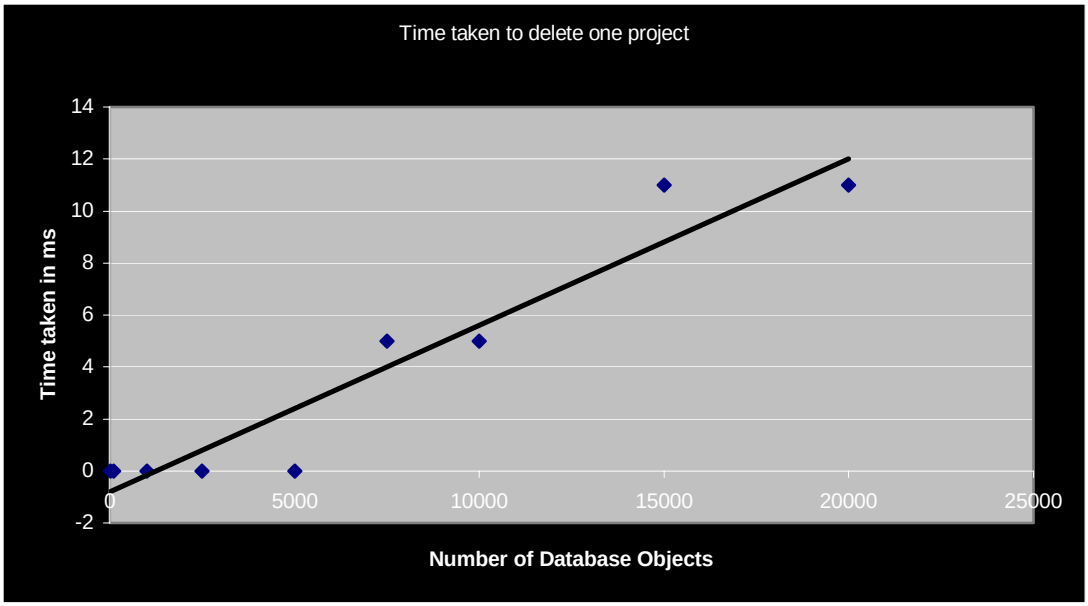
The exponential trendline fitted to the IDGenerator data is indicative of poor scalability. When the database grows to a very large number of items, the time taken to insert each item may be constant, but the time taken to ensure that the item is uniquely identifiable is increasingly large. Thankfully, the base which we raise to the nth power (where n is greater than 1) is experimentally determined to be very close to 1, and so the growth is slow for less than 5000 DatabaseObjects.

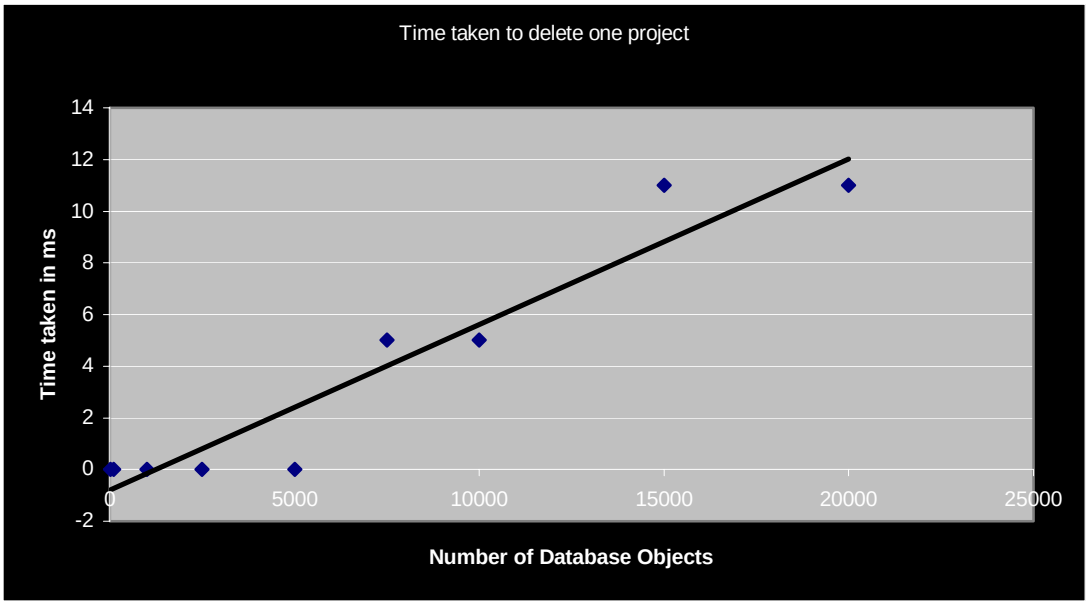






As strongly suggested by the straight trendlines in the preceding graphs, the time taken to select any number of users is propotional to the number of DatbaseObjects. As the number of projects to select increases, the entropy of the graph decreases, and the trendline fits better.





From the above two graphs, we notice that the same applies for deletion. The explanation for this phenomenon stems from the 0{n} speed of access of a DatabaseObject. An Object may be added in O{1} time without the need for sorting, but the cost of this quick addition is the need to iterate through all items to find a particular one.
This trade-off was chosen because of the way the database is loaded from a file upon boot (or a recovery file after a crash). A user would grow impatient with the long wait while the database loads.

Prediction and Anticipated Behaviour

We found the equations using the trendline function in Microsoft Excel.

Function	Equation	Predicted Value at 100000 (ms)
Select all projects	y = 0.0045x + 9.3926	456.3926
Select project range	y = 0.01x + 2.7901	2790.1
Delete one project	y = 0.0006x - 0.7923	59.2077
Select one project 100 times	y = 0.0455x + 32.866	149540.3

We have used linear regression lines here, because in practice that is what they are mostly to be, based on our experimental work. As demonstrated here, the database will slow down, but not to an extent that it becomes unusable.

For example, deleting a project would still only take 59 milliseconds, which is quite an achievement. However, almost by definition, as the database size grows, there are likely to be more users on the system, and hence the system will slow down somewhat because there are simply more users using the system at any one time.

Evaluation and Conclusion

After a thorough experimental and theoretical analysis of the system, we find that the Project Management System database is fairly robust under a yardstick load of 100,000 DatabaseObjects. The general access speed of a Database object is O(n), while to add an Object takes O(1). The object is added quickly without sorting but the cost of this quick addition is the need to iterate through all items to find a particular one.

This functionality was deliberately planned because of frequent interaction with files of entire databases that must be loaded. For example, a database is loaded from a file upon boot (or a recovery file after a crash). The user specifies which database file they wish to load. However, a user would grow very impatient with the long wait while the database loads if the system sorted as it added. Additionally, it is embarrassing to cause the program to crash while loading a crash recovery file.

The developers believe that the system is scalable up to 100,000 items, memory permitting. For very heavy use, where the contents of the database are being accessed, selected, and updated constantly, the system would perform poorly. However, the Project Management System 2004 was designed principally to store large amounts of data, rather than to act principally as a data manipulation tool. Because the most important functionality of adding items from a file takes O(n) time, loading the database would be feasible, and the system would perform well.

Appendix: Code Listings of Scalability Test Drivers and XML Definitions

```
<!—IDGenTest by Enoch Lau -->
<ClassDriver>
  <DriverName>soft1902.util.scaling.DatabaseDriver2</DriverName>
  <TestList>
    <Test>
      <!-- ID Generation First Set of results -->
      <Name>IDGenerator 1000 Times (#1)</Name>
      <Iterations>10</Iterations>
      <MethodName>generate1000IDs</MethodName>
      <ResetDriver>true</ResetDriver>
    </Test>
    <Test>
      <!-- ID Generation Second Set of results -->
      <Name>IDGenerator 1000 Times (#2)</Name>
      <Iterations>10</Iterations>
      <MethodName>generate1000IDs</MethodName>
       <ResetDriver>true</ResetDriver>
    </Test>
    <Test>
      <!-- ID Generation Third Set of results -->
      <Name>IDGenerator 1000 Times (#3)</Name>
      <Iterations>10</Iterations>
      <MethodName>generate1000IDs</MethodName>
      <ResetDriver>true</ResetDriver>
    </Test>
  </TestList>
</ClassDriver>
```

```
/*
 * Filename: DatabaseDriver.java
 * Created: 14/09/2004
 * Author: Josh Green
 * Description:
 *    The database testing driver used to Drive the database through
 * generic routines. each of these functions listed in this class
 * are accessed through an XML script file, and each function is
timed
 * each time it is called
 *
 * History: Josh Green - Stubbed (14/09/2004)
 *
 */
package soft1902.util.scaling;
import soft1902.server.data.*;
import java.util.Vector;

public class DatabaseDriver implements ClassTestDriver{
  //Instance variables
  Database db;
  User rootUser;

  Vector lastUser;
  Vector lastFile;
  Vector lastEvent;
  Vector lastProject;
  IDGenerator idGen;

  /**
   * Driver Constructor
   * Make sure that everything is null to begin with this, (avoids
null pointer)
   */
  public DatabaseDriver(){
    db = null;
    rootUser = null;
    deinitialise();
  }

  /**
   * Initialise function
   * Called by the TestDriver to reset the database Object
   * and any other objects stored in the driver back to
   * their original states
   */
  public void initialise(){
    deinitialise();

    System.out.print("Reinitialising Database: ");

    if(db==null){
      db = new Database("ScalabilityTest");
    }else{
      db = new Database("ScalabilityTest2");
    }
    rootUser = db.getRootUser();

    lastUser = new Vector();
    lastFile = new Vector();
    lastProject = new Vector();
```

```
    lastEvent = new Vector();

    idGen = new IDGenerator();
  }

  /**
   * Deinitialise the database and delete any files associated with
it
   * This function is a must otherwise the database trys to recover
   * using it's temporary log file, (not good when it tells you it
just
   * recovered 4000 queries and is about to run another 1000)
   */
  public void deinitialise(){
    if(db!=null)db.close();
    java.io.File f = new java.io.File("ScalabilityTest.xml");
    if(f.exists())f.delete();
    f = new java.io.File("ScalabilityTesttempLog.txt");
    if(f.exists())f.delete();
    f = new java.io.File("ScalabilityTestIDlog.txt");
    if(f.exists())f.delete();
    f = new java.io.File("ScalabilityTest2.xml");
    if(f.exists())f.delete();
    f = new java.io.File("ScalabilityTest2tempLog.txt");
    if(f.exists())f.delete();
    f = new java.io.File("ScalabilityTest2IDlog.txt");
    if(f.exists())f.delete();
  }

  /**
   * Generate 1000 id's,
   * using the ID generator
   */
  public void generate1000IDs()
  {
    for (int i = 0; i < 1000; i++)
    {
      String id = idGen.getUniqueID();
    }
  }

  /* ALL THESE INSERT A NUMBER OF OBJECTS INTO THE DATABASE
   */
  public void insertUserItem(int n){
    try{
      Vector objs = db.processRequest("INSERT user set
name=\"user"+n+"\"",rootUser);

if(lastUser.size()<=0||n<5)lastUser.add(((DatabaseObject)objs.get(0))
.getID());
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void insert10UserItems(){
    for(int i=0;i<10;i++){
      insertUserItem(i);
    }
  }
```

```
  public void insert100UserItems(){
    for(int i=0;i<100;i++){
      insertUserItem(i);
    }
  }

  public void insertProjectItem(int n){
    try{
      Vector objs = db.processRequest("INSERT project set
name=\"project"+n+"\"",rootUser);

if(lastProject.size()<=0||n<5)lastProject.add(((DatabaseObject)objs.g
et(0)).getID());
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void insert10ProjectItems(){
    for(int i=0;i<10;i++){
      insertProjectItem(i);
    }
  }

  public void insert100ProjectItems(){
    for(int i=0;i<100;i++){
      insertProjectItem(i);
    }
  }

  public void insertFileItem(int n){
    try{
      Vector objs = db.processRequest("INSERT file set
name=\"file"+n+"\"",rootUser);

if(lastFile.size()<=0||n<5)lastFile.add(((DatabaseObject)objs.get(0))
.getID());
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void insert10FileItems(){
    for(int i=0;i<10;i++){
      insertFileItem(i);
    }
  }

  public void insert100FileItems(){
    for(int i=0;i<100;i++){
      insertFileItem(i);
    }
  }

  public void insertEventItem(int n){
    try{
      Vector objs = db.processRequest("INSERT event set
name=\"event"+n+"\"",rootUser);

if(lastEvent.size()<=0||n<5)lastEvent.add(((DatabaseObject)objs.get(0
)).getID());
```

```
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void insert10EventItems(){
    for(int i=0;i<10;i++){
      insertEventItem(i);
    }
  }

  public void insert100EventItems(){
    for(int i=0;i<100;i++){
      insertEventItem(i);
    }
  }

   /* ALL THESE ARE SELECTING ENTIRE SETS OF OBJECTS
   */
  public void selectAllUser(){
    try{
      for(int i=0;i<10;i++){
      Vector objs = db.processRequest("SELECT user",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectAllProject(){
    try{
      for(int i=0;i<10;i++){
      Vector objs = db.processRequest("SELECT project",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectAllFile(){
    try{
      for(int i=0;i<10;i++){
      Vector objs = db.processRequest("SELECT file",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectAllEvent(){
    try{
      for(int i=0;i<10;i++){
      Vector objs = db.processRequest("SELECT event",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  /* ALL THESE ARE SELECTING SPECIFIC OBJECTS WITHIN A SET
   */
```

```
  public void selectOneUser(){
    try{
      for(int i=0;i<100;i++){
        Vector objs = db.processRequest("SELECT user where
id=="+lastUser.get(lastUser.size()-1),rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectOneProject(){
    try{
      for(int i=0;i<100;i++){
        Vector objs = db.processRequest("SELECT project where
id=="+lastProject.get(lastProject.size()-1),rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectOneFile(){
    try{
      for(int i=0;i<100;i++){
        Vector objs = db.processRequest("SELECT file where
id=="+lastFile.get(lastFile.size()-1),rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectOneEvent(){
    try{
      for(int i=0;i<100;i++){
        Vector objs = db.processRequest("SELECT event where
id=="+lastEvent.get(lastEvent.size()-1),rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  /* ALL THESE ARE SELECTING A RANGE OF THE SET
   */
  public void selectUser(){
    try{
      for(int i=0;i<10;i++){
        Vector objs = db.processRequest("SELECT user where
name>=user5",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectProject(){
    try{
      for(int i=0;i<10;i++){
```

```
        Vector objs = db.processRequest("SELECT project where
name>=project5",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectFile(){
    try{
      for(int i=0;i<10;i++){
      Vector objs = db.processRequest("SELECT file where
name>=file5",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void selectEvent(){
    try{
      for(int i=0;i<10;i++){
      Vector objs = db.processRequest("SELECT event where
name>=event5",rootUser);
      }
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  /* ALL THESE DELETE AN ITEM FROM THE SET
   */
  public void deleteUser(){
    try{
      Vector objs = db.processRequest("DELETE user where
id=="+lastUser.get(lastUser.size()-1),rootUser);

if(objs.size()>0)lastUser.remove(((DatabaseObject)objs.get(0)).getID(
));
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void deleteProject(){
    try{
      Vector objs = db.processRequest("DELETE project where
id=="+lastProject.get(lastProject.size()-1),rootUser);

if(objs.size()>0)lastProject.remove(((DatabaseObject)objs.get(0)).get
ID());
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void deleteFile(){
    try{
      Vector objs = db.processRequest("DELETE file where
id=="+lastFile.get(lastFile.size()-1),rootUser);
```

```
if(objs.size()>0)lastFile.remove(((DatabaseObject)objs.get(0)).getID(
));
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  public void deleteEvent(){
    try{
      Vector objs = db.processRequest("DELETE event where
id=="+lastEvent.get(lastEvent.size()-1),rootUser);

if(objs.size()>0)lastEvent.remove(((DatabaseObject)objs.get(0)).getID
());
    }catch(Exception e){
      System.out.println("Error");
    }
  }

  /**
   * Load a database from a file
   * @param name - the Database name that is to be loaded
   */
  public void load(String name){
    db = new Database(name);
  }

  /**
   * Save the current database to a file
   */
  public void save(){
    db.shutdown();
  }
}
```