Automated Database Refresh in Very Large and Highly Replicated Environments

Eric King
Regis University

Follow this and additional works at: https://epublications.regis.edu/theses

Part of the Computer Sciences Commons

Recommended Citation
https://epublications.regis.edu/theses/471

This Thesis - Open Access is brought to you for free and open access by ePublications at Regis University. It has been accepted for inclusion in All Regis University Theses by an authorized administrator of ePublications at Regis University. For more information, please contact epublications@regis.edu.
Disclaimer

Use of the materials available in the Regis University Thesis Collection (“Collection”) is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the “fair use” standards of the U.S. copyright laws and regulations.
AUTOMATED DATABASE REFRESH IN VERY LARGE AND HIGHLY
REPLICATED ENVIRONMENTS

A THESIS

SUBMITTED ON 26TH OF AUGUST, 2011

TO THE DEPARTMENT OF COMPUTER SCIENCE OF THE SCHOOL OF
COMPUTER & INFORMATION SCIENCES OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF
SCIENCE IN SOFTWARE ENGINEERING AND DATABASES TECHNOLOGIES

BY

Eric King

APPROVALS

Dr. Ernest Eugster, Thesis Advisor

Donald J. Ina - Ranked Faculty

Program Coordinator
Abstract

Refreshing non-production database environments is a fundamental activity. A non-productive environment must closely and approximately be related to the productive system and be populated with accurate, consistent data so that the changes before moving into the production system can be tested more effectively. Also if the development system has more related scenario as that of a live system then programming in-capabilities can be minimized. These scenarios add more pressure to get the system refreshed from the production system frequently. Also many organizations need a proven and performant solution to creating or moving data into their non-production environments that will neither break business rules, nor expose confidential information to non-privileged contractors and employees. But the academic literature on refreshing non-production environments is weak, restricted largely to instruction steps. To correct this situation, this study examines ways to refresh the development, QA, test or stage environments with production quality data, while maintaining the original structures. Using this method, developer's and tester's releases being promoted to the production environment are not impacted after a refresh. The study includes the design, development and testing of a system which semi-automatically backs up (saves a copy of) the current database structures, then takes a clone of the production database from the reporting or Oracle Recovery Manager servers, and then reapplies the structures and obfuscates confidential data. The study used an Oracle Real Application Cluster (RAC) environment for refresh. The findings identify methodologies to perform the refresh of non-production environments in a timely manner without exposing confidential data, and without over-writing the current structures in the database being refreshed. They also identify areas of significant savings in terms of time and money that can be made by keeping the structures for developers with freshened data.
Acknowledgements

This project would not have been possible without the co-operation and help from the staff and friendships I have made over the last three years. Merrill Lamont and Jason Buchanan were mentors who helped me grow and love this profession with their examples.

To Dr. Ernest Eugster, who was always available for our weekly phone conferences and provided excellent responses to my questions and offered further questions to keep the ideas going and the project moving forward.

To my family and siblings, my father Pat and my mother Mary, my mother-in-law Jean, Eileen, Patrick, Peter, Brian and Róisín and my sister-in-law Terri and my brother-in-law Nick; for providing me the initial loan to pursue this thesis, their understanding and their patience throughout this project timeline. Without their love and support this project would not be possible. Thank you for the chocolate chip cookies.

To my many nieces and nephews: Isabella, Tessa, Liam, Conor, Fiona, Orla, Emma, Nora, Isabella Grace, Harrison, and the little bump – your uncle is back to play again.

Finally to my wonderful wife Meghan for her constant support, unflagging energy and well-wishes made this project possible. For providing me the space and time to complete this in spite of our busy work schedules. This was not only greatly appreciated, it was totally necessary to getting this done on time. I hope that one day I can return the favor.

Thank you everyone for your help and support through this thesis, I greatly appreciate it.

– Eric.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Literature Review</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Methodology</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Data Analysis and Findings</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>47</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Installation Instructions</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.1 - Traditional Four Tier Development Environment

Figure 3.1 - EMC SAN Copy – Pull/Push Strategies

Figure 3.2 - Onsite and Offsite Infrastructure Layout

Figure 3.3 - The Automatic Database Refresh Architecture / Process Flow Diagram

Figure 4.1 - Elapsed Timeline Chart Showing Overlapping Processes
List of Tables

Table 4.1 - Total Effort Timeline Breakdown
Table 4.2 - Detailed Total Resource Cost Breakdown Comparison
Table 4.3 - Compare the Old Refresh Process with the New Refresh Process
Chapter 1 – Introduction

Many IT companies struggle with the issue of database refresh from a security, cost and
downtime perspective. It would be a folly to believe one can simply copy production into a non-
production environment. Many issues need to be addressed including data obfuscation, gateways
and data replication. The code being promoted up the stack will be over-written. Furthermore, a
balance must be made between frequency of refreshes and amount of downtime the development
team can absorb. The reason for this balance requirement is that developers need near-prod
quality data – the process to provide this prod-quality data can take up to a week. During this
week the developers cannot code which is a poor use of resources. For every change a developer
makes, it makes the environment less like production. If the gaps between refreshes are too long,
then code will not execute correctly in the production because the data quality is stale.

This chapter introduces basic IT concepts and provides a brief history of technology
structures used. It also examines the typical technology stack created by an organization and
what actions are targeted at what stage in the technology stack. This researcher developed a
methodology that could be followed to refresh the development, QA, test or stage environments
with production quality data, while maintaining the original structures.

1.1 Background - A History of Technology Structures

When programmers first started their trade, the idea was to place the client, application
and database in a single machine. Some machines were made famous by this concept, including
the ENIAC and the Turing Machine.

Because programmers needed to book time on costly mainframes, programming was
initially done on sheet cards and fed individually into the computer. Programs were small in size
and were slow to create; often weeks. FORTRAN and COBOL allowed formulae to be entered
directly into the computer. Over time, dropping hardware costs allowed monitors and direct
interaction via keyboards. This greatly sped up computer program development, as the user
interacted with a mainframe computer via a “dumb” terminal.

In the 1980s and 1990s with the advent of networking and the World Wide Web –
centralized computing was no longer attractive. Lower-cost, network connected commodity-
based hardware could be leveraged for a fraction of mainframe costs. Programmers also had
access to enormous computational power.

In the 90s, with the rise of n-tier layered systems - the database, middleware and client
layer model allowed for modular work to be done. Independence between objects and layers
enabled an environment that was easier to maintain, support and enhance than mainframe
environments. But this brought additional problems that needed to be solved. One problem came
with the business rules and validations, namely in what location should these business rules be
stored? Problems arise if the logic is stored on the client then changes required finding and
changing similar code on different screens. If the logic is stored within the database server, then
changing database vendors required high porting costs and introduced risk of unsupported
features.

With the object-oriented world growing, applications were architected into three primary
layers: presentation, domain and data storage. Thus, changes in one layer would not impact the
other layers. This saved time and money.

The removal of highly expensive monolithic mainframes for smaller, more numerous
clustered servers were further advanced with the advent of real application clusters and highly
available Virtual Internet Protocol (VIP) based servers. Not only can lower costs be achieved,
but if a server fails, all data including data in flight within the network can be rolled-back and
failed over to an available server. This allowed for higher system availability which meant less IT outages which impact revenue and costs.

Businesses now place their environments (production, stage) on commodity based hardware configurations. For the programmers, coding via a text editor would not be competitive enough to deal with the explosion of speed and complexity of the enterprise level software environments. Software development, CASE, IDE and SDKs and version control systems would all need to be harnessed to create and modify the complex systems of today's market leading products. As a result, staff had to collaborate to handle the complexities of the technology stack – system analyst, object-oriented programmers, system administrators, database administrators, testers and software engineers.

To help coordinate the different responsibilities, a four phase environment was developed where projects would be divided into:

- Development
- Integration
- Staging
- Production

Each phase is shown in Figure 1.1. The figure shows the flow of promoting code from the development environment, to bug testing on the integration and stage environments until final promotion to production environments through the co-ordinated efforts of the QA team and the release manager.
1.2 Development Environment

The development environment is the fundamental process in which developers create new content, fix bugs or modify content. Referring to Figure 1.1 step 1, a developer writes code changes on their own desktop environment or in a linked development environment with other developer’s code in flux. This provides an area with semi-current code and with near up-to-date code structures for the developer's to not code in a vacuum or be aware of the impact their code changes can have on other objects in the environment. When performing unit testing, the
developers need to be able to get the most current versions of test/dev so they have a common baseline and not coding too specifically or else issues may arise when they bring their code together. Version control of the baseline and of codes must be done as a standard, as shown in Figure 1.1 step 2. This version control is normally handled by Subversion. Referring to steps 3, 4 and 5 from Figure 1.1, bugs are reported and fixed by the developers and checked into subversion. The benefit to the development environment is the developer can focus on the unit of code they are tasked to create or modify, without external factors affecting this endeavor. The developer would typically be responsible for testing and validating their unit of code and providing exception catching.

The disadvantage to the development environment is that the developer is coding in a vacuum. Another disadvantage is that the data within the test environment is typically divorced from production quality, since it has been edited or changed due to other development work. Other developer have tasks to modify and create code of their own, and it will not be known if these two units will work together until they are placed in an integrated or testing environment.

1.3 Integration Environment

The integration environment is a proving ground for the code integration of the units of code built and promoted from the development environment. These units of code to be promoted are determined by the developers and release manager. In Figure 1.1 step 7 the release manager would also be responsible for packaging what code is to be promoted to the stage environment. Within the integration environment, sub-components of the system are tested to ensure the modified and created pieces of code from the developers work as expected and do not introduce bugs into the system. Interfaces between the software components will be tested at this point to ensure each unit can communicate with corresponding units of code.
The integration environment typically has a subset of data from production which is refreshed frequently to prevent “garbage” build up. This task can be done initially as a refresh from production followed by flashback of the database to a point-in-time to allow iterative testing with the same data set. This saves time and can be done each evening to allow for a cleaned up environment each morning for the developers to work in.

**1.4 Stage Environment**

In the *stage environment*, which is a hardware, software and hopefully minimal delta change version of production, the quality assurance (QA) team tests the code with coverage, load, smoke, stress and exception tests as well as functional unit and system tests. This is made clear in step 8 of Figure 1.1. Reports are sent to the release managers who will determine which developer's code needs to fix the bugs, see Figure 1-1 and refer to steps 9, 10 &11. Once these bugs are fixed, the release manager again inspects for issues and checks into Subversion and promotes to stage environment. This is an iterative process until the QA team is satisfied that the code is of a high enough standard to release to production, see Figure 1.1 steps 12 and 13. The release manager then packages the release version and co-ordinates promotion to the production environment; see Figure 1.1 steps 14, 15, 16.

The staging environment can be used for running demos for clients or as a training environment. The staging environment is typically updated via log updates or batch job updates from production to ensure data is current.

**1.5 Production Environment**

The *production environment* is the live environment clients view when connecting to the system. It should be bug and error free and meet customer service level agreements with ease, to allow for growth of the system and during times of exceptional load (typically end of year). The
production environment should house the best hardware the company can afford to meet client’s needs. The production environment typically has standby and back-up servers in case of fail-over or power loss of the primary production site. These standby servers can be located in a different part of the building or in different locations of the city or country to allow up-time of the system in case of catastrophic failure, for example, a failure of a utility or a fire at the primary location. The standby servers are constantly written to and updated from the primary production servers so that the mean-time-to-recovery is short and preferably un-noticed.

1.6 Traditional Four Tier Application Development Structure – Advantages and Disadvantages

The advantages of the traditional four tier structure is it focuses on catching bugs and issues as early in the process as possible, thus reducing expensive fixes later in the Software Development Life Cycle (SDLC).

Initially the focus is on the unit of code, typically in object oriented environments this would be a class. Bug discovery at this stage results in quick and efficient fixes for incorrect outputs and false exception handling. This is the cheapest and easiest form of bug fixing because the concept of the unit is still fresh in the developer's mind and can be adjusted quickly to produce the desired results.

Another advantage is the iterative process that is inherent in so many of the SDLC methodologies. Starting with the initial unit of code, then adding other developer's units to the test, then components of the system are tested and finally the entire system is tested – this allows for bugs and issues to be found prior to the client finding them. An addition to this is that, with the different environment tier, new features can be applied in a structured approach prior to releasing the new feature to the client.
The main disadvantage of the traditional four tier structure is cost. Hardware, storage and cooling all take their toll on company profits. Another disadvantage to the tiered system is the additional steps and disruptions to developers work when it comes time to refresh environments. In small and well-designed organizations, this is less troublesome as there are fewer inter-dependencies on objects. However, for products that have evolved over a long period of time, large organization and poorly-planned projects with many artifacts which have not been removed from the database, the process can take a great deal of resources and time to ensure the stability of the environment post-refresh. These resources could be put to better use building and improving the product and developing and testing other new products.

1.7 Research Methodology

Refreshing non-production database environments is a fundamental activity to ensuring accurate development and testing of new features and bug fixing. Without production-like business rules and data, the developers and QA testers would be unable to promote patches, bug fixes and new features to the production environment.

But the academic literature is weak on details and lacks concrete applicable examples from industry. The existing literature focuses on instruction steps and theory. To correct this situation, this study examines ways to refresh the development, QA, test or stage environments with production quality data, while maintaining the original structures. Using this method, developer's and tester's releases being promoted to the production environment are not impacted after a refresh. The study includes the design, development and testing of a system which semi-automatically backs up (saves a copy of) the current database structures, then takes a clone of the production database from the reporting or Oracle Recovery Manager servers, and then reapplies the structures and obfuscates confidential data. The goal of the research is to fill the gap of over-
simplistic examples and encourage a comprehensive approach to the refresh methodologies.

This research examines ways to refresh the development, integration and stage environments with production quality data while maintaining the original structure. Using this method, developer's and tester's releases being promoted to the production environment are not impacted after a refresh.

This researcher developed a refresh process that could be followed was to identify the correct methodologies to perform the refresh of non-production environments in a timely manager without exposing confidential data, and without over-writing the current structures in the database being refreshed. Significant savings in terms of time and money can be made by keeping the structures for developers with freshened data. The findings hope to prove greater flexibility and better utilization of resources available to a company. It will not only test backup and fail-over systems on a monthly basis when extra computational load is expected due to copy from production backups, as opposed to when there is a serious failure and the backups have never been tested.

The primary research activity consisted of designing, developing and testing a system which semi-automatically backed up (saved a copy of) the current database structures. The researcher then took a clone of the production database from the reporting or Recovery Manager servers, and then reapplied the structures and obfuscates confidential data. The study used an Oracle Real Application Cluster (RAC) databases for refresh. The study will be kept to Oracle databases – since this will be the only vendor database available to this researcher.

The research method chosen was Grounded Theory. By using Grounded Theory to take existing schema statistics on separate environments and applying the hypothesis that by combining them together on a production server, we should not really affect load. The Grounded
Theory was the most logical research methodology for this research. This researcher used what has already been proven as an industry and academic standard, and adapting it to develop the study on refreshing non-production databases. The advantage of Grounded Theory is that it ventures little from what is already proven; giving you a solid bedrock to test from, and using comparative analysis on other theses and secondary research of what other academics have discovered can provide credence to your hypothesis. By using what is proven already to steer this thesis, it should be easier to prove/disprove what is to be accomplished. And by making initial steps with each of the methods stated in the first paragraph above, a divide-and-conquer method can be employed to further bolster findings.

The methodology also included interviews. The researcher interviewed MIT Computer Science graduates, IEEE and Oracle forums and Oracle Users Group's discussion boards of Puget Sound and New England Oracle User's Group. The thesis was explained to senior members of the IS team, and followed up with fail-over and back up that many production servers have which development environments do not. With a production system, because it was not viewed as a cost-limiter, but rather maximizes profit, many financial managers saw the necessity of paying for premium production systems and parts. However, development environments were often neglected and are of a much cruder construct. As a result, the movement of data from Production to non-Production environments can take a longer than expected amount of time. This reduced the time required to smoke-test and validate.

1.8 Success criteria

To focus on research objectives and establish targets for evaluating progress, this researcher identified the following four success criteria,
1. Provide clear steps on how to perform a database environment refresh of data while maintaining the non-production structures.

2. Ensure that no accidental production incursions occur post-refresh.

3. All supporting technologies used to perform the database refresh must themselves provide clear error handling, log writing and give engineers a means to determine root cause analysis of an issue.

4. The refresh must be completed and smoke tested over a weekend period for up to twenty terabytes of data.

1.9 Conclusions

This chapter provided a brief history of the software architecture model and the need for four environments to capture bugs and errors prior to deployment to the client facing production environment. Database refresh brings real-time client facing business rules to the non-client facing environments to allow developers and testers to create new features and fix bugs in a close to real-environment as possible. Without a frequent and robust refresh process the four stages of the traditional software model quickly breaks down. Companies invest a great deal of resources in purchasing and maintaining these environments and must be cognizant of not only the hardware but also the data and metadata components. If the data is not close to production like quality – errors, bugs and wasted effort becomes part of the process over head.

Chapter two provides a review of the literature which details previous research in this area. It shows differences of opinion on whether to use live or synthetic data and the challenges associated with both approaches – and gaps in the research which allows improvements to the database refresh activity to be explored and presented.
Chapter 2 – Literature Review

Chapter 1 introduced the need for a clear and secure method of refreshing non-production database environments, without exposing confidential data, in an automated and performant manner. It also explained the research methodology used in the study.

This chapter explores the existing literature on database refresh. Considerable research does exist. But it is limited to the description of schema refresh or entire database refresh without thought of the sequences and underlying metadata which constitutes the clear identity of the original target environment. This chapter examines the research and thoughts of some of the great minds in the area. This thesis provides a more holistic approach to the refreshing of non-production environments while maintaining the underlying metadata and code still being promoted up the chain of environments.

2.1 Fundamental Relational Database Research

E.F. Codd (1990), considered by many to be the father of relational databases, defined the most important motivation for relational database research that resulted in the relational model which was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management. “Relational processing entails the treating of whole relationships as operands. Its primary purpose is loop-avoidance, an absolute requirement for end users to be productive at all, and a clear productivity booster for application programmers” (Codd, 1990).

Other researchers built upon Codd’s research. Khurana (2002), an author of many introductions to relational database books, provided examples regarding the advantages of the relational database over the traditional file processing system. He noted that because “same
information may be duplicated in several files...this duplication of data is known as *data redundancy*, which leads to wastage of storage space. The other problem with file processing systems is that the data may not be updated consistently.” Khurana goes further into the matter of data inconsistency. He argued that constraints and business rules should prevent false data, which would break business rules, from being entered into a table. “In a file processing system, all these rules need to be explicitly programmed in all application programs, which are using that particular data item” (Khurana, 2002). For example, entering letters into a column which should only expect numbers would break business rules and a database should enforce these rules. To take this example further, adding more business rules would require a complete re-write of the application, rather than a separation of database and application. Another key feature is data file level security. On a file processing system, all similar users have access to all files in the directory, without significant administrative overhead, private or confidential data could unwittingly become exposed. Worse still is in case of an accident or catastrophic occurrence; files could be lost without the possibility of recovery.

Researchers have also noted the risks of exposing confidential data when populating database with live or mock data. If populating with mock data, then how are business rules and unique and relational constraints ensured? Wiederhold and Bilello (1997) forwarded the idea of row-level security. Essentially, row-level security (RLS) would, at least in theory, ensure data confidentiality. It would add an additional predicate condition to all queries, such that would effectively obfuscate the confidential data. For example, if one wanted to obfuscate data in a table which matched the following condition EMP_ID = 123, then one could create a database policy which appends ' not in EMP_ID = 123' to all queries in the database. Effectively, this data would not be output to unauthorized users.
Wu, Wang and Zheng (2003) provided instructions on how to create mock databases for application software testing without revealing any confidential information from the live database. Their article was an improvement over Wiederhold's work – which was to provide a synthetic database which would be valid (preserves business rules of live production database), resembles the original data and preserves privacy (no confidential data is inferred). Wu, Wang and Zheng claimed that the main disadvantages of Wiederhold's proposal – which was to apply the row-level security on queries as follows: “The live data may not reflect a wide enough scope to test beyond the current state of the database. It may be difficult to identify correct users queries which may or may not negatively impact the live database to a point that it no longer reflects the live production database. Another condition that is not immediately apparent is the performance impact the implicit predicate addition will have on the database. Troubleshooting such issues may prove difficult if execution statistics do not include the predicate (Wu, Wang & Zheng, 2003)”

By leverage *a-priori* knowledge and leveraging Entity-Relationship (E-R) Models and metadata engines - Wu, Wang and Zheng hoped to create a database that would not contain the same data as production, but would satisfy all the rules of production.

Although Wu, Wang and Zheng promoted tight security policies, the question of database constraints was never entirely answered. The resultant database could contain many spurious and invalid result sets which break and ignore the business rule set. Another drawback would be the level of maintenance and upkeep required to make edits to all the E-R Models and metadata and rebuild in time for each refresh. A final drawback was the absence of performance from the article – could this work be done in a time that was worth the effort – without answering this question, it was not likely this approach could be followed in large highly replicated databases.
Chays (2000) emphasized the axiom of “populating a database with meaningful data that satisfies constraints. This is the first component of a comprehensive tool that will ultimately semi-automatically generate test cases.” He argued that by using live data – all constraints and referential integrity was maintained – which is a key tenet of a relational database. Another advantage would be that performance could be also tested if the metadata was copied down as well as the live data – this would allow near real-prod testing in the non-production environments and provide accurate details to testers and developers on the impact any changes to objects would have before promoting to production. Chays’ main concern with live data testing was the possible security and privacy concerns of an application tester viewing confidential and secure data.

The disadvantages of synthetic data testing was that many testing teams only considered the population of tables with random data in a hope of having a large enough data footprint with which to test. A glaring hole in the synthetic tester’s theory was the complete absence and near certain breakdown of relational integrity when using random data in a database containing more than one table. The RDBMS engine requires accurate integrity constraints being upheld within the database to be able to create logically correct output from SQL queries which can be parsed. Once relational integrity breaks down, the database is no better than a series of non-relatable tables – a crude Excel spreadsheet.

Kuhn (2007) wrote about potentially cloning the source database (production) to the target database (development, test or stage) leveraging the Oracle Recovery Manager (RMAN) tool. RMAN is an acronym for the Oracle Recovery Manager, which can be considered the Oracle database file backup and restoration utility. To clone the production database, the RMAN tool would take a full datafile backup of the environment and restore it over the existing non-
production environment database. This will effectively over-write the target database information at the core database level – the test database “thinks” it’s the production database. Significant work will be required by the DBA to restore the test database identity and environment parameters. Also, user passwords will be production level passwords and confidential information will be now located in the test environment. It is this researcher's belief that having production level passwords and confidential information in a lower environment poses grave and serious security issues, and may not be viewed favorably by senior members of business. In a replicated environment which could also store its queuing and replication settings within database tables, accidental production database incursion could be done from a lower environment without detailed smoke-testing and validation by the database administrator and QA testing teams.

2.2 Gaps in Research

The existing research still relies on the need to consider the “relations” within a relational database to prevent foreign keys and constraints from breaking the system or producing false errors. Although synthetic data is an attractive and potentially viable offering – the databases that this researcher worked with were highly integrated and multi-terabyte in size. As a result, the effort to create the synthetic data would be highly inefficient use of this researcher's time and effort. Since data will be copied from production – only classified and confidential data is stored in only a dozen columns throughout these multi-terabyte databases. As a result, a quick PL/SQL scrubbing application can be developed so that this classified and confidential information can be obfuscated and protected, while at the same time maintaining the correct constraints.

Another item that was missing from the above feature set was the restoration of the test environment structure once the environment had been over-written by production. By creating a
structured only export of the original target, then comparing the source db and the structure only copy to determine the deltas, then copying down the source and applying the deltas – you have effectively restored the old target structure with new data. Without these structures, many development code promotions may be hampered or QA teams may need to spend hundreds of man-hours to re-write their testing scripts.

2.3 Conclusions

A great deal of research exists on database refresh activity. But without ensuring referential integrity via foreign keys and constraints – non-production environments quickly fall apart and the database loses cohesiveness. The use of live or synthetic data has its pros and cons. There are many ways to refresh an environment - copying files, using Oracle export utilities, RMAN, or data pump. But these only copy over the production level data to the target. Once in the target database, significant work must be done to prevent confidential information from becoming exposed, user passwords will need to be re-built and test structures need to be restored or development promotion in the non-production environment could be lost.

The next chapter examines the research methodology. It details the research environment, apparatus used and the means to measure results.
Chapter 3 – Methodology

Chapter 2 examined the literature produced by academic minds on the subject of database refresh and presented the debate between live production data versus mock data and the associated advantages and disadvantages of both approaches. It also noted the gaps in current research where this research work can provide clarity and present a way to refresh the non-production databases without exposing confidential data in a performant manner.

Chapter 3 discusses the research methodology. Grounded Theory was chosen because it was the most logical research methodology for comparing what has already been proven as an industry and academic standard, and adapting it to develop the study on refreshing non-production databases. The advantage of Grounded Theory is that it ventures little from what was already proven; providing future researchers a solid bedrock to test from, and using comparative analysis on other research. By using what has been proven already to steer this researcher’s work, it should be easier to prove/disprove the success criteria. Also, by making initial steps with each of the methods stated in the first paragraph above, a divide-and-conquer method to further bolster the findings can be gained. The chapter describes the five research phases used in the current environment and what changes would be implemented by a new refresh methodology. Details of the database, hardware, storage and infrastructure can be found in Appendix A. It provides instructions on re-producing the environment. Finally, a high-level explanation of each piece of the automated database refresh application closes the chapter. A more detailed explanation can be found in Appendix B.
3.1 Research Method Chosen – Grounded Theory

Grounded Theory (GT) was the methodology chosen for several reasons. Numerous researchers have explained the significance of GT for research. Martin and Turner (1986, p.141) defined Grounded Theory as an "inductive theory discovery methodology that allows the researcher to develop a theoretical account of the general features of the topic while simultaneously grounding the account in empirical observations of data.” Grounded Theory offers “a logically consistent set of data collection and analysis procedures aimed to develop theory” (Charmaz, 2001 p. 245). By identifying patterns in the data and studying the before and after effects “the theory-building process is so intimately tied with the evidence that it is likely that the resultant theory will be consistent with empirical observation” (Eisenhardt, 1989).

Urquhart (2001) emphasized two key canons of Grounded Theory: (a) the researcher has to set aside theoretical ideas; and, (b) the concepts are developed through constant comparison. It is this researcher's goal to monitor the current methodology, insert a single variance and study how this variance impacts the system as a whole. Through this methodology a greater understanding of the variables involved can be explained and contrasted against the empirical body gathered to prove the theories put forward.

Grounded Theory was also considered after researching an article (Calloway & Knapp 2000) that compares two studies that employed the theory to investigate information systems development tools. The researchers used ethnography while interviewing designers and engineers in the computer industry. Calloway and Knapp discovered that many of the workers “develop strong attachments to their tools that they express in highly emotional language. These attitudes, attitude patterns and beliefs that accompany tool usage and systems design are learned.” This led to the realization that “research on information systems development tools has
focused on the tools themselves and not on the use of the tools in an organizational context” (Wynekoop & Conger, 1990). Using Grounded Theory and coding, highly disparate studies can use a similar methodology and coding practice to analyze and understand interview data. Due to the need for high levels of collaboration throughout the Software Development Life Cycle process, it became clear that the Grounded Theory methodology would be the best method to use for this researcher’s work.

3.2 Research phases

The research proceeded in five phases. In the first phase, which was covered in Chapter 2, examined the existing literature. It showed that for the scope and size of the databases that needed to be refreshed and tested in the Test and Stage environments – synthetic database creation would be too laborious and would become obsolete so quickly with the highly volatile nature of the database objects.

In the second phase, requirements workshops were setup to gather requirements and brainstorm ways of improving the old process. Managers wanted a scalable and low-cost solution. Support technicians wanted something that would be easy to maintain. Developers and testers wanted a refresh of their environment that was quick and error free. During the workshop, volunteers from key stakeholder groups were contacted for follow-up interviews. Open-ended questions about the old process and what end-users would like to see in the new process were recorded. These interviews were followed up with a synopsis of what was covered to ensure high quality of requirements gathering.

Observations of end-users following the old process of refresh were employed to gather data about the manual and difficult effort that was needed to complete the tasks. Ad hoc requests for assistance would not have been recorded had observation practices not been undertaken.
In the third phase, a research environment was created using existing hardware and software made available by this researcher’s employer. The database system used was Oracle 11gR2 Enterprise Edition. The database was installed on commodity hardware, Oracle Unbreakable Linux operating systems and Raid 5 – EMC Celerra SAN storage systems. The technology leveraged to copy the data from one environment to the next environment was a combination of Oracle Recovery Manager and EMC SAN Copy.

EMC SAN Copy allows businesses to quickly and securely move data from one storage system to a different storage system which is platform and server independent. According to EMC, SAN Copy ensures that valuable server cycles are reserved for servicing applications and user requests by performing data copy and movement as a function of storage. The product distributes data without interruption to production while protecting server CPU cycles and preventing LAN bandwidth bottlenecks. SAN Copy does not steal CPU cycles through file transfer overheads.” SAN Copy, as shown in the Figure 3.1, has two modes: push and pull. Push is used over short distances and increases confidence in data quality. Pull, typically used over long distances, does not wait for status, but has two fewer checks than push, which increases throughput (EMC, 2010).
3.3 Refresh Process

Figure 3.1 shows, at a high level, the process of how the data was replicated from the source to target environment and how the process of refreshing the non-production environments was undertaken. The read-only Oracle RMAN server (which was used for backups and reporting) has archive-logs directly applied to it from production. The RMAN server was then used as a source to refresh the Read-Write Clone server (used for production testing and fixing errors). The Clone server was finally used as source for refreshing the System Integration Testing (Test) environment and the smaller Dev environments. This environment was Onsite – typically in the building/location where primary business is conducted – housing the production server for performance and security reasons. Offsite sites are those which do not house primary source users, but are important in case of accident, fire or catastrophic failure of systems at the primary site.
Offsite sites are typically on a separate electrical power, water and ancillary utility grid with backup generators / water tanks in case of failure. Offsite sites are useful for holding disaster recovery (DR) sites – so in case of emergency, business continuity is ensured. DR is a physical replica of the production database, so performance is ensured if the systems fail-over to the DR server. Disaster recovery (which is kept in Read-only log-archive apply mode) was used as the source for the Stage environment. The stage environment was used as the final area to test code changes before promotion to the Production environment.

Production databases house the client facing portion of the business. For disaster recovery and risk mitigation, the production database writes its updates to the DR Server space (which acts as a physical standby to production) and the RMAN server (which can be used to copy files, take backups or as a reporting, read-only, database). To reduce the risk further and
reduce performance penalties, no copies of production required by the non-production environments are taken directly from the production server - but rather the RMAN server is considered “prod-quality” and used as the source database instead. A read-write production-like environment – called a clone– exists to reproduce production level bugs and errors and fix them, again without having to directly accessing production, and then promote the fix to production environment - once it has gone through the correct QA and promotion process via test and stage environments.

The process differed from traditional refresh of non-production environments because the researcher’s employer organization does not simply overwrite the contents of the test or stage environments with the contents of production. The reason for this may seem obscure, but the test and stage environments are where all the development and testing and new features are being generated – to overwrite this would essentially negate dozens of man-months of work.

Due to the elastic and dynamic nature of employees, and the inter-dependencies of teams on core tables within the database, a holistic version control system can only be leveraged with the application code and not necessarily the structural changes which constitute all the changes to the database for a particular version of the product. Therefore, the organization took a structure-only backup of the database and placed it online. To test the environment, production-like files from the RMAN server (read-only) were copied over. If the organization starts up the Test database now, it would contain and reflect an exact copy of the RMAN server database settings. Scripts must be executed to rename the SID and global name and the data-file headers. Once the Test server has been brought online, a metadata-comparison is made between the structure-only backup of the old Test environment is made and the prod-like fresh copy in the current test environment. All of the columns which have been added or removed in the Test environment
but have not yet been promoted to production is outputted to a diff script. These scripts are then executed to place the structure of the old Test environment back the way it was, but with production quality data. Finally, all data replication, middleware, gateways and queues must also be converted to Test settings to avoid a potential for production level incursions – again scripts have been generated to prevent accidents- human or otherwise. Once the process is completed the environment is now refreshed – and QA testers may begin smoke testing the environment to ensure nothing has been missed.

The review of the literature and interviews from Chapter 2 showed that theoretically it would be possible to engineer a prototype to test the thesis hypotheses. Permission was granted to design, test and code the automated database solutions and to place it on the organizations integration environment initial and pending the feedback from developers place it on the backup servers of the production environment. This would allow the company to operate without the automated database system in production, and monitor the effects the system had on their non-production environments without sacrificing service level agreements with clients. Resources and technical staff were provided to be on standby should any incident arise from the changes. Programmers were made available to consult should any questions or programmatic roadblocks need to be solved, and to also monitor any sudden loss of service due to the changes proposed.

In phase four, the system would be tested and metrics would be gathered with production and development environments in parallel.

Finally in phase five, the results would be observed for stability, accuracy and performance. These metrics would then prioritize the focus on what fixes would provide the best return on resource investment.
3.4 Environment Refresh and Synchronization Process

The section below includes the following topics:

- DBSync Components (Application Directories and Files)
- Architecture
- Application Setup
- Configurable parameters
- Usage

The only topic that is not covered in detail is the schema compare utility of Toad, offered by the Quest Company and how to record and run the schema compare as an action from command line. The detailed steps are included in Appendix C.

3.4.1. DBSYNC

DBSYNC is a repeatable semi automated script based solution that helps in overriding all the production parameters and production configurations when a non-production environment is refreshed. The refresh of the non-production environment was done using a combination of Oracle RMAN and EMC SAN Copy solutions that clones the production instance. This process basically created copies of production database there by helping the development teams to have access to pristine production like image of data on regular intervals. This process in turn helps improve the quality of code that goes into production because most of the code gets tested with the exact image of production to meet all the challenges well before the code is deployed to the production environment.

This researcher found the DBSYNC process answered two important requirements that make these environments usable as a part of the ongoing development process.

- Bring the database up to date with all the structure changes as they existed in the
environment before the refreshes process.

- Override the production parameters and replace all the prod references to the environment specific reference (stage / test).

The first requirement specified is critical since this researcher’s employer organization was an ongoing 24*7 development process. As a result, the stage and test environments contained critical application patches / releases that had not yet been promoted to production. These requirements usually translated to structural changes or changes to the core data model. It is important to restore these changes to what they were prior to refresh. This is where we largely use the Schema Compare capabilities of Quest Software’s TOAD to accomplish. For details, see Appendix C. The DBSYNC process comprised of several modules of usable and reusable components that could be added / removed and customized to suit each database needs. This is basically handled by a set of configuration files discussed in detail in installation and configuration section of the document.
3.4.2. Automated Database Refresh Architecture / Process Flow Diagram

![Automated Database Refresh Architecture / Process Flow Diagram](image)

Figure 3.3 – The Automatic Database Refresh Architecture / Process Flow Diagram

Figure 3.3 shows that the process flow diagram was split into three parts: pre-process, refresh process and post-process. The pre-process of refreshing an environment involved various preparations to ensure a smooth and error-free experience for all teams. Initially,
communications to all developers, QA teams, managers and clients were provided up to four weeks in advance to allow for plenty of time for clients to request a delay to the schedule and to set expectations with their customers. Once all administrative green lights were given, the Business Continuity Volumes (BCV) synchronizes the backup mirrors with the Production-like environment source (RMAN server) – which provides a block-for-block accuracy level copy of the source environment.

Since the Structure Only exports of the original target environments were completed already, the second step started. This step compared the source database to the original target database structures and outputs all the diffs, also known as deltas. These diffs are stored in files on mounts which are SAMBA mounted so they can be opened from either Unix or Windows platforms. The third step generated additional scripts from the target DB. For example, the original passwords were extracted and the database links and their synonyms were backed up. Scripts were validated for accuracy, and any discrepancies were corrected before stopping the listeners and database and shutting down the Shareplex replication.

In the refresh process, the EMC SAN Copy copied the point-in-time copy to the target database, thereby overwriting the original SIT database with production-like structure and data – so the target and source databases are clones of one another.

Once the refresh process was completed, it was time to begin the post-process. It began with starting the databases in restricted mode, which included starting alternate listeners so that developers and QA testers cannot accidentally connect to the database. The MQ Gateways were stopped and synchronization of the environment with the earlier backups taken. All schema/user passwords were reset to their original settings for the target environment. All public and private database links and synonyms were reset to their original. All the original sequence values will be
restored and their currval and nextval would be restored. If the environment uses messaging gateways and queues or Shareplex – then further restoring of queues and recreation of subscribers and schedules are necessary. For the Shareplex piece, a restoration of the original Shareplex configuration will be restored and Shareplex replication is compare/repaired and restarted.

3.5 Conclusions

This chapter provided a high-level explanation for the four phase research methodology used. Grounded Theory was considered the best research methodology to use. The technologies are explained at a high level of detail, and further instructions for installing the instructions and screenshots have been moved to Appendices A, B and C.

The next chapter will provide the results collected and gathered when the methodologies were executed.
Chapter 4 – Data Analysis and Findings

Chapter 3 discussed the research methodology used to refresh the development, QA, test or stage environments with production quality data, while maintaining the original structures. It identified instruments and devices that were used to re-create the process. This chapter focuses on the data analysis. The analysis suggests several findings that support the thesis that the new process developed by this researcher refreshes the non-production environment in a low-risk, automated manner. This has the potential to reduce the chance of production incursions and reduces the need for skilled workers to execute the process. Each milestone in the process emails the senior members of the team so they have a status update as the process successfully move ahead, or an error log if something goes wrong.

4.1 Results

After meeting with developer team leads, architects, and managers, it was quickly determined that, although a copy of the source DB would be the quickest solution, it was not the desired methodology to pursue. The reasons for not simply copying the source DB were threefold:

1. The source database contained confidential data that could not be exposed to third-party vendors with access only to the non-production environments. The company could face litigation, scandal and irreparable damages to its pristine reputation.

2. Due to the highly replicated nature of the environment, by copying down directly from source, hundreds of settings would still be set to the production environment. This would invite a higher risk of accidentally causing a production incursion. A production incursion is when a process, or person, believes they are operating in a non-production environment, submit commands and commit the changes – only to realize they have
actually made changes on the client/world facing environment of the company. Again, civil and criminal litigation proceedings and scandal would destroy the company if such an occurrence happened.

3. Structures being promoted through the development stack would be overwritten by copying directly from source. Developers would needlessly have to re-write their changes which are an unacceptable use of resources at the company.

Breaking down the results in terms of effort, performance and cost savings over the old methodology – the focus was determining where savings could be made and performance improved in the overall process. The resultant data has been tabulated for ease of reference and comparison.

4.1.1 People Effort

Table 4.1 shows the task milestones. It describes whether these tasks were automated, executed manually, or not part of the feature set in past. Finally, a breakdown of resources used to fulfill the tasks. Focusing on the total person hours, the old refresh process took 65 hours, or up to 8.125 man-days. Comparing this figure to the new refresh process, the effort was reduced to only 5.4 hours, or less than 0.75 man-days. This is equivalent of a 1080% improvement in terms of human resources for every refresh.

Table 4.1 also shows that the old refresh process was a highly manual process with almost 40% of the task features unable to be accomplished. This evidence is apparent in the second and third columns which compare that mode of operation. The new researched refresh process was highly automated; highly monitored and error-resistant. Human error is minimized in an automated process.
Comparing the effort in hours between the old and new processes – there were few improvements in terms of hours in duration of effort. One advantage was fewer human resources were required for each task. The savings become quickly evident with only three people required during validation (which actually breaks down to one database administrator and two middleware engineers) as compared to the old process four people, all of which are database administrators.

Table 4.1: Total effort Timeline breakdown.

Table 4.1 further shows that the new process was more than 95% automated – which results in less manual activity and error opportunities for the database administrator. Freed from mundane work, the administrator can focus on higher value and more strategic activities. The
new process also added numerous features over the manual process. The following features were not possible under the initial manual refresh process:

- retrieving role information for all schemas,
- determining which grants were assigned to which roles,
- re-creating users which had not been granted access to production environments,
- rebuilding all snapshot and Multi-Master Replication tables,
- the entire DB synchronization post-process

All actions previously requiring manual activity were now automated and the jobs placed into the batch job server – to be called and monitored when the environment was to be refreshed.

In addition, less monitoring and transparency was available in the old method which encouraged human error and negative steps to be included in the final process. The current methodology avoided such negative impacts to the process and allowed for best practices and continued standard operating procedures to be applied via scripts. Although the overall effort has changed little, the effort for human resources was minimal and the scope of risk within the refresh activity was mitigated significantly. Finally, steps that were manual would require a database administrator to monitor the progress of the job. For example, synchronizing the read-only environment with the read-write environment (which would act as the source) took up to six hours and required a database administrator to finally place the read-only database into standby mode again. The new process was now automated and leveraged a batch job which was executed by the Database administrator and when the job was completed, a report of the job was
emailed. The total effort of work on the database administrator was reduced from six hours to approximately 30 seconds.

4.1.2 - Efficiency Gain

Table 4.2 tends to support the research study; that the new refresh procedure is more secure and stable, lowers production incursion risks, and is a faster method of refreshing the environment. The new process significantly reduced costs: $4 million less than the manual process per year. Figure 4.2 compares costs between the old and new processes. It is based upon the following assumptions:

- Only 150 resources were considered for the Calculation (1/3 of the whole organization)
- Cost: $100 per hr
- 30% productivity loss when the environment was not available (A * 0.30): $216,000.
- To explore the findings further, interviews were conducted which revealed the following:
  - Application team effort (pre- and Post-refresh) for each refresh (Old: 3 resources for 1 week; New: 8 hrs/refresh)
  - Average productivity loss due to bad data in the SIT was around 15 to 20%. Used only 12% for the calculation.
  - 10% inefficiencies still existed in the SIT environment after implementing MDR.
  - Total cost of refreshes per year was calculated as the number of resources (R) multiplied by average resource cost (C) multiplied by length of down time (5 weeks or 240 hours) generated by bad data in SIT environment (B) – which equals R*C*B=cost of bad data in the SIT environment (Z). This equals $3,600,000 per year.
• There are 15 applications (T) – so if a single application team is impacted the cost is \( Z / (T) = (A) = 3600000 / 15 = \$240,000 \)

• Several applications exist on a single environment – which would take one week to restore \( Z / B * 40 = \$720,000 \)

Based on the calculations from Table 4.1 – four resources were required for pre-process and eight resources for post-process in the old refresh method. In the new method, only four hours of work in both pre- and post-processes were required.

Tabulating the costs of the old process based on the above figures - adding the above totals up for the Total Cost of Refresh per year (\$13,000), the cost of a single week’s downtime for an SIT environment (\$720,000), the cost of resources for the pre-process (\$120,000), the cost of resources for the post-process (\$240,000), the cost of bad data in the System Integration Testing environment (\$3,600,000) resulted in a total of \$4,693,000.

Tabulating the costs of the new process based on the above figures - adding the above totals up for the Total Cost of Refresh per year (\$6,480), the cost of a single week’s downtime for an SIT environment (\$216,000), the cost of resources for the pre-process (\$72,000), the cost of resources for the post-process (\$72,000), the cost of bad data in the System Integration Testing Environment (\$360,000) resulted in a total of \$726,480.

Taking the cost of the new process from the old process results in a cost savings of performing a refresh of the System Integration Environment of \$3,966,520.
Table 4.2 Detailed Total Resource Cost Breakdown Comparison

<table>
<thead>
<tr>
<th>Cost Assumptions</th>
<th>Old Process</th>
<th>New Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only 150 resources were considered for the Calculation (1/3 of whole organization)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spoke with managers, Architects and Senior members of staff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Applications</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Avg number of resources per application (6 dev, 3 qa and 1 analyst)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Cost/hour</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>No of Refresh / year</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

**Refresh Cost**

| Person hours for System Integration Testing Environment                           | 65          | 6.4         |
| Cost Per Refresh                                                                  | 6500        | 540         |

**Total Cost of Refresh per year**

| $13,000                           | $6,480      |

**Environment Downtime Cost**

| Assuming 30% of Time is lost per person due to environment downtime                |             |             |
| Business Days lost due to environment downtime                                     | 10          | 0.5         |

**Environment Downtime Cost**

| $720,000                           | $216,000    |

**Cost of Application’s Team pre-work before the refresh**

| Assuming only developers are impacted                                             |             |             |
| Pre-work effort per person in hours (old 40 hrs and new 4 hrs)                    | 8           | 4           |

**Cost of Application’s Team pre-work before the refresh**

| $120,000                           | $72,000     |

**Cost of Application’s Team Post-work after the Refresh**

| Post-work Effort per person in hrs (old 80 hours and new 4 hours)                  |             |             |

**Cost of Application’s Team Post-work after the Refresh**

| $240,000                           | $72,000     |

**Cost of bad data in the System Integration Testing Environment**

| Based on high level conversations with various teams. There is 15 to 20% loss due to Bad Data which converts to 8 to 10 Weeks |             |             |
| Being conservative used 12% for calculation which converts 6 weeks per person for the old process |             |             |
| 10% inefficiencies still exists in the System Integration Testing environment after implementing the Refresh process |             |             |
| Time lost due to bad data in System Integration Testing (5 Weeks per person per year in hours) | 2400        | 24          |

**Cost of bad data in the System Integration Testing Environment**

| $3,600,000                          | $360,000    |

**Total Cost per year**

| $4,693,000                          | $726,480    |

**Cost Saving after new Refresh method is implemented.**

| $3,966,520                          |             |
System Integration Testing environments. Considering the small to medium size of the company, the costs are quite significant.

### 4.2 Analysis

#### 4.2.1 Projected: People Effort (Old vs. New Process)

Figure 4.1 Elapsed Timeline Chart showing overlapping processes

Figure 4.1 presents the results from Table 4.1 and overlaps the human resource effort of both the new and old processes. Based on the old refresh process, the people effort required per refresh was around 65 hrs. Based on the new refresh process, the people effort required per refresh is around 5.4 hrs. The process was almost halfway completed before our first human resource was required for one hour. The automated process continued until it was time to validate the structures synchronization and finally the activation of Shareplex. When compared to the old process, human resources were required from the beginning of the process and quickly...
began to cost both in terms of hours worked and number of workers required to complete the work. The new process is modular and should new environments be added to the process, it is scalable enough to accommodate the extra workload without adding significant time to the overall process.

**4.2.2 High Level Process Refresh Comparison**

Table 4.3 presents the results from comparing the old refresh process against new refresh process. In the old process scripts were written and executed manually. In the new process, scripts were batched and highly automated. This greatly reduced the chances of human error, and increased productivity.

Because of the similar naming conventions used between environments and the requirement to connect to a production quality data-source – the risk of impacting production was high due to the manual nature of the process. In the new process – the scripts were fenced off in a batch server which does not have connection privileges to any production servers except the read-only datasource – and only during the refresh window.

The highly manual nature of the old process required 65 hours of human work, which equaled $6500 per refresh. From a cost efficiency and development impact standpoint this meant a refresh could only happen twice a year. Under the new process, the human effort was only five hours, with estimated costs at $500 per refresh (or 1/13th the cost using the manual process). Assuming 12 refreshes per year, the costs to the company are 50% less than the old process.

Because the EMC SAN Copy process ran overnight if anything went wrong, it required intervention from teams offshore. The new process leveraged privileged user batch jobs which
produced reports for each environment that completed – so a baseline could be established the next day if the SAN Copy was completed. Finally, no application team resources were required in the new process – all objects were restored, so objects were identical to when the process started.

Table 4.3 shows that the automated process is more efficient and effective. The data forecasts a $4 million efficiency gain.

<table>
<thead>
<tr>
<th></th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Manual</td>
<td>Automated</td>
</tr>
<tr>
<td>Impact to production</td>
<td>High Risk</td>
<td>Minimized</td>
</tr>
<tr>
<td>People effort</td>
<td>~ 65 hrs</td>
<td>~5 hrs</td>
</tr>
<tr>
<td>Cost/refresh estimate</td>
<td>~ $6,500</td>
<td>~$500</td>
</tr>
<tr>
<td>Elapsed time/refresh</td>
<td>11 days</td>
<td>2 days (Projected)</td>
</tr>
<tr>
<td>Refreshes/year</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>People off-hours effort</td>
<td>Required</td>
<td>Minimal</td>
</tr>
<tr>
<td>Application teams involvement</td>
<td>Required</td>
<td>None</td>
</tr>
<tr>
<td><strong>Projected Efficiency Gain</strong></td>
<td></td>
<td>~$4 million</td>
</tr>
</tbody>
</table>

Table 4.3 Comparison of the old refresh process to new refresh process

4.3 Conclusions

The new refresh process supports this researcher’s findings of a scalable and performant database refresh method. It met or exceeded every one of the success criteria outlined in Chapter 1. The new refresh process provided a holistic way of not only refreshing the environment, but putting the structures back in place which will minimized the impact to developers and testers who had made structural changes to the lower environments during their promotion of new features to the technology stack. Not only were over 60 man hours of effort saved with the new
process for every refresh – an expected cost savings of over $4 million per year – but a more robust and scalable solution was provided which minimized risk and increased productivity.

The next chapter provides observations on the contributions of this study to the literature and makes recommendations for future research.
Chapter 5 - Conclusions

Refreshing non-production environments is a fundamental database activity. Without a recent version of production quality data, the development, integration and staging environments quickly become stale. Testing will not expose errors or bugs early in the testing cycle. In Chapter 2, it was noted that refresh is an under-researched area. It also provided evidence of earlier research in this area; highlighting the advantages and disadvantages of refreshing non-production databases with live data, or creating mock data. Chapter 3 explained the four phase research process based upon Grounded Theory and an iterative testing methodology. Chapter 4 presented findings that refreshing the development, QA test or stage saved on average $4 million per year per environment. It also showcased how scalable and more frequent refreshes could become by leveraging automation and reducing and eliminating application team involvement in the process.

This chapter begins by recapping what this researcher has learned and then discusses the obstacles overcome. Finally, it concludes with suggestions for future research.

5.1. Database Production Environment Revisited

The researcher believes this project contributes to the IT literature by providing a clear format on how to refresh non-production environments without exposing confidential data or disrupting developer’s promotion of code through the technology stack. The findings help to explain how changes to the lower environments will be done to production because the metadata will already be captured and applied to lower environments. Leveraging metadata baselines for evaluation of new code in production environments, developers appreciated that not only data, but metadata could be copied down from the production environments. This means statistical
and data quality is kept in-line with production environment workloads and performance that could be expected, when the code would eventually be promoted to production servers. This reduced unexpected performance or data-related surprises on the day of deployment and also provided a better baseline for QA testers.

Security issues were avoided by the use of Oracle 11g features, which included, but are not limited to, the new identity (NID) utility (which updates the database name and files with a new identify) and ‘rename database’ features as well as the ‘global_name’ feature – all of which can be used to change the Oracle SID (Oracle System Identifier) to a different name – thereby keeping non-production environment refreshes safe and clearly labeled.

Post-refresh all former non-production environment passwords, roles and privileges were restored. Developers and QA team members who typically enjoy liberal privileges while developing and testing code, but would have restricted roles and privileges in the live production environments. Also, developers appreciated the exception tables list, which removed certain tables in the environment from the refresh action. This guaranteed metadata and lower environment specific data were retained. For example, the currency table may have pseudo-currencies being considered in lower environments. Rather than overwriting this data and having to re-publish it, one could exclude this table from refresh – thereby saving the Developer and DBA time and effort.

5.2 Commodity-based Considerations

With organizations moving to commodity-based, clustered infrastructure – RAC refresh has been proven as compatible with this method of refreshing the non-production environments. The additional steps of shutting down all but one instance of the clustered database and then
follow the refresh procedures as outlined in Chapter 3 was all that was required. The risks and human effort are greatly reduced which provides cost savings and automation.

This research also found that the jobs of security team’s jobs were improved because all commands can be automated via a batch scheduler. This limits the access to only specific users during specific times, and prevents ad-hoc changes to the commands issued – unless done through a controlled and visible release cycle.

The final product was deployed because of the high level of cost savings both in terms of resources and time and due to the increased robustness of the process and mitigated risk to unwanted errors. QA testers appreciated receiving new data without the new structures so existing QA framework tests did not require an update with every refresh from production. Developers were also pleased with the reduced impact to their version control and development efforts. Additional improvements were implicit obfuscation of tables – which was captured during design talks with developers and managers. Post-refresh scripts were created which removed key data from tables from the lower databases with every database. Developers and testers also reported less down time typically caused by earlier methods of refreshing the development and integrated environments.

5.3. Lessons Learned and Future Research

In designing the research, this researcher used a variety of procedures and controls to validate the work. Using the initial process as a baseline, capturing the metadata when changes were made provided accurate metrics which revealed how the changes will behave when promoted to production. Oracle Real Application provided the tools to perform this baseline performance comparison. Also, breaking the process into self-contained pieces and then
studying how the original manual refresh compared to the new automated refresh and where improvements were made allowed for a higher level of confidence in calculating the overall savings.

But this researcher does acknowledge that the methodology had to overcome some obstacles.

- Licensing costs and rights differ greatly between the non-RAC and RAC environments – this caused many issues with deploying the automated database refresh process to the production and RAC environments.
- Re-building the environment due to a change in the researcher’s employment eventually took time and required re-work due to changes in industry rules at the new position of employment, delayed research.
- Acquiring buy in from the management should be done as early in the process as possible. Document each change no matter how innocuous so that facts and data received from testing, does not escape the process and decisions that seem trivial are recorded and reviewed at a later date.

This study was designed to prove the successful and cost effective means of refreshing an Oracle non-production environment, over a weekend, without impacting the development teams database object's structures. But it should be considered an early attempt to an overall refresh of any class of database, vendor agnostic, and be able to do expect the same results or better. This researcher suggests that other researchers begin with a simple environment, with low-coupling and low-replication and try to study this alone – incrementally add complexity to the environment until fully integrated, multi-master, shareplexed, snapshot intensive, highly replicated environment which is 24*7*365 and tuned for performance – and then change vendor
technologies. Future researchers can then appreciate the value of robust RDBMS engines and how even through complexity, albeit bad architecture may be to blame for many issues that as engineers we study to provide solutions and build them. It is this researchers hope that an open-source RDBMS vendor will soon be able to market a comparable system. This may encourage a more competitive marketplace and level the playing field for companies with less IT budget, but with great minds and passion to engineer a better solution.

Other items that could be considered for further research within the Oracle vendor stack are CPU profiling – which would give certain jobs priorities within the refresh and give high priority jobs more CPUs or more CPU time compared to low priority jobs. Another study that holds promise would be to determine whether better performance is gained with the Enterprise class Oracle installations versus standard installation. Nothing this researcher used truly leveraged the Enterprise cost structure – since Oracle made the availability of RAC a standard feature of version 11g in 2007. This researcher believes that more research would produce a more complete and useful picture of refreshing non-production environments, with high-levels of replication and high-degrees of coupling without risk to the company's confidential data in a performant manner.
References


Appendix A – Installation Instructions

The following instructions were followed to install the environments within the lab to conduct the comparison between the new and old refresh processes.

This appendix is split into four parts:

- Grid infrastructure Installation
- ASM Diskgroup and ASM Cluster filesystem setup
- The installation of the Oracle database binaries
- ASM storage setup

Grid Infrastructure Installation

1) Download the Grid Infrastructure Binaries from Oracle

2) Log into the account that will own the Grid Infrastructure, “grid”

3) Start the runInstaller
Select “Install and Configure Grid Infrastructure for a Cluster”.
Select Advanced Installation

- **Typical Installation**: Perform a full grid infrastructure installation with basic configuration.
- **Advanced Installation**: Allows advanced configuration options such as alternative storage choices, additional networking flexibility, integration with IPS, and more granularity in specifying Automatic Storage Management roles.
Select English and Click Next >
Enter the cluster name you planned out and the associated SCAN name that was configured. The cluster name is the same as the SCAN name minus the appended domain name. For research build purposes, the CLUSTERNAME was test_server-cl1 (15 character limit). The SCAN NAME was test_server-cl1-scan.company.com. Add in all the nodes in the cluster and their associated VIP that has been configured.
After all nodes have been added, click on the “SSH Connectivity” button, enter the grid account and password and finally select the “Test” button. Added company.com suffix to hostnames and vips. We also used the grid password then clicked setup. Oracle performs some background operations to ensure ssh password-less connectivity is assured. Once the test comes back successfully, click on the “Setup” button.
Select the correct NIC for the public and private networks. There will likely be bonded and unbonded ports configured for the public, private, and IP storage configurations. It is important to select the correct ports that have been configured for these specific uses and also mark the ports configured for IP storage as “DO NOT USE”.

Bond 0       PUBLIC

Bond 1       PRIVATE
Choose ASM storage for the clusterware devices
The clusterware devices require normal redundancy and three physical devices. The name of the disk group should be “CLUSTERWARE_DG” and the devices configured as DISK_C# should be selected for this disk group. Choose the 3 * 4GB devices from the list.
Create the password for the ASM instance

The new Automatic Storage Management (ASM) instance requires its own SYS user with SYSASM privileges for administration. Oracle recommends that you create a less privileged ASMSNMP user with SYSDBA privileges to monitor the ASM instance.

Specify the password for these user accounts.

- Use different passwords for these accounts

<table>
<thead>
<tr>
<th></th>
<th>Password</th>
<th>Confirm Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASMSNMP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Use same passwords for these accounts

Specify Password: [ ] Confirm Password: [ ]

Messages:
- Specify Password [NLS-30011] The password entered does not conform to the Oracle recommended standards.
Do not configure the IPMI
Select the appropriate OS groups to manage the ASM environment for role based management.
Specify the correct Oracle base and the software location for the grid infrastructure. These directory placements are critical due to directory ownership and permissions and must be followed precisely.
Enter the location for the Oracle inventory

You are starting your first installation on this host. Specify a directory for installation files. This directory is called the "inventory directory". The installer automatically sets up subdirectories for each product to contain inventory data. The subdirectory for each product typically requires 150 kilobytes of disk space.

Inventory Directory: /ora/app/orainventory

Members of the following operating system group (the primary group) will have write permission to the inventory directory (orainventory):

orainventory Group Name: admin
When prompted to run a few scripts as root on each node, it is important to run the first script (orainstRoot.sh) on each node, and then run the second script (root.sh) on each node. Have the system administrators (UNIX administrators) run these scripts. Do not proceed to the second script until the first one has run on each node.
This completes the Grid Infrastructure installation which consists of the Oracle clusterware and the ASM instances on each node. Check that all resources are built. `crsctl status resource` – creates your local listeners as well as the LISTENER_SCAN
2) Create ASM Disk group and associated ASM Clustered Filesystem for the database “admin” directory structures.

Caveats:

1) Log into the account that will owns the Grid Infrastructure, “grid”.

2) Startup the ASM Configuration Assistant (ASMCA)
Make sure an ASM instance is running on each node in the cluster
Click on the Disk Groups tab and you should see the existing Disk group that was configured for the clusterware devices.
Click on the “Create” button and create the new disk group “DB_ADMIN_DG”. Select devices provisioned with the DISK_A# naming convention. The example below shows external redundancy but we will very likely be mirroring across devices from different failure groups. Make sure you evaluated those requirements. Sort disks by size, then claim the 2 * 16GB devices.
Create the volume within the new disk group. The redundancy will be inherited by the disk group. For our purposes we chose size 20GB.
Create the filesystem as a “Database Home Filesystem”. This will inform the clusterware to treat this resource as a pre-requisite to databases instances when starting up the services.

NOTE: Select Database Home File System, and ensure Database Home Owner Name is set to Oracle.
Run a script as root which will change the permissions on the mount point and add the resource to clusterware.

Run `crsctl status resource -t` to see the new features added are offline. Each new feature will need to be brought online.
AUTOMATED DATABASE REFRESH

Make sure the new filesystem is mounted on all nodes in the cluster.
3) **Install Oracle 11.2 database binaries**

The following are the high-level steps that are needed to install the Oracle database binaries.

- Download the Oracle database 11.2 binaries
- Log into the account that will own the database software installation, “Oracle”.
- Change directory to the $INSTALLER_HOME/bin directory and start the runInstaller.
Choose “Install database software only”
Select ALL nodes in the cluster to install the binaries. Click the “SSH Connectivity” button and enter the Oracle account and password and then test the User equivalency.
Accept the default language of “English.”
Select Enterprise Edition

Which database edition do you want to install?

- **Enterprise Edition (4,29K9)**
  Oracle Database 11g Enterprise Edition is a self-managing database that has the scalability, performance, high availability, and security features required to run the most demanding, mission-critical applications.

- **Standard Edition (4,220K8)**
  Oracle Database 11g Standard Edition is ideal for work groups, departments, and small to medium-sized businesses looking for lower-cost solutions.
Install all of the components
Select the Oracle database as well as the path for the software installation making sure to follow the standards.
You’ll receive a warning regarding the placement of the Oracle database in the home directory of a user account, in this case the Oracle account. It is safe to ignore this.
Select the appropriate OS group “dba” for the sysdba and sysoper based upon the standards.
Review the settings chosen for accuracy.
The installation will proceed on all nodes.
The Execute Configuration scripts prompts to run a script as root on each of the nodes in the cluster.
This completes the Oracle 11.2 database binary installation.
4) Create ASM Disk Groups for Database Storage

- Standardized “DATA” and “FRA” disk groups will be created and shared by all databases.

- Additional disk groups can be configured as needed.

- Log into the account that is part of the asmdba OS group, “grid”.

- Startup the ASM Configuration Assistant (ASMCA).
On the Disk Group tab, click on the “Create” button.
Create the disk group name as “DB_DATA_<DEVICE TYPE>_<PROTOCOL>_#”. This example demonstrates normal redundancy which requires two distinct failure groups. In this case, the devices are mirrored across storage frames to prevent loss of service due to a storage frame failure. Also, notice that the devices chosen have the same size and performance characteristics.
Create the flash recovery area disk group name as “DB_FRA_<DEVICE_TYPE>_<PROTOCOL>_#”. For the FRA disk group it is usually not necessary to provide ASM mirroring as it will only store multiplexed copies of the controlfile and redo logs, archivelogs, flashback logs, and backups.
Appendix B – Detailed Environment Refresh and Synchronization Process

The section includes the following topics

- DBSync Components (Application Directories and Files)
- Architecture
- Application Setup
- Configurable parameters
- Usage

The only topic that is not covered in the documentation is TOAD which is basically specific to how to use the Schema compare utility of Toad and how to record and run the schema compare as an action from command line. There will be a separate document for toad that would cover these topics with some known issues and learning's.

DBSYNC is a repeatable semi automated script based solution that helps in overriding all the production parameters and production configurations when a non production environment is refreshed.

The refresh of the non-production environment was done using a combination of Oracle RMAN and EMC SAN Copy solutions that helped in cloning the production instance. This process basically creates copies of production database there by helping the development teams to have access to pristine production like image of data on regular intervals. This process in turn helped improve the quality of code that goes into production because most of the code gets tested with the exact image of production to meet all the odds well before the code was deployed to production environment.

This researcher found the DBSYNC process answers two important requirements that make
these environments usable as a part of the ongoing development process.

- Bring the database up to date with all the structure changes as they existed in the environment before the refreshes process.

- Override the production parameters replace all the prod references to the environment specific reference (STAGE / TEST).

The first requirement specified was critical because of an ongoing 24*7 development process as a result the STAGE and TEST environments contains critical application patches/releases that have not yet been promoted to production. These requirements usually translate to structural changes or changes to the core data model. It was important to restore these changes to as they were that could always translate to a better appreciation of data refreshes by the application team. This was one requirement where we largely use the Schema Compare capabilities of TOAD (tool from Quest Software) to accomplish.

The DBSYNC process was built into smaller chunks of usable and reusable components that could be added / removed and customized to suit each databases’ needs. This was basically handled by a set of configuration files which will be discussed in detail in installation and configuration section of the document.

**B.2. Architecture / Process Flow Diagram**

Details regarding the flow of this diagram can be found in Chapter 3.
B.3 Installation and Configuration

B.3.1 Directory Structure

Unix Directory structure details

```
DBSYNC
DBSYNC/sqlFiles
DBSYNC/envFiles
DBSYNC/output
DBSYNC/toadFiles
DBSYNC/toadFiles/TEST
DBSYNC/genSQLFiles
```

The DBSYNC application can be hosted under any parent directory in the above example. It was created under the home directory as the hosting directory.

DBSYNC: Parent / Application home directory.

DBSYNC/sqlfiles: SQL Scripts repository. This directory hosts a list of SQL files that are executed on the target instance by the DBSYNC process and then in turn generate a series of executable SQL files by reading the metadata from the target databases.

DBSYNC/envFiles: This directory contains all the critical application configuration files. The settings in these configuration files can be tuned to drastically change the behavior of the application. It is advised to read through the configuration section of the document that deals in depth with the configurations files before any changes are done to these files.

DBSYNC/toadFiles: This contains all the toad Sync files that will be applied to the target database. This directory is usually succeeded by the Environment and SID directories. For example, when executing the TEST database refresh for the SID instance then the other directories that would succeed the DBSYNC/toadfiles directory would be TEST/TEST_SID. Resultant directory structure in this case would be DBSYNC/toadFiles/TEST/TEST_SID.
Format of the same will be described as <ENV> (that will signify TEST / STAGE) and <SID> in the rest of the document.

**DBSYNC/output**: This directory contains the output of all the scripts that are run on the target database. Any abnormal execution exceptions or application asserts will be logged in this directory. This directory also succeeds by the <ENV> and <SID> directories.

**DBSYNC/genSQLfiles**: This directory contains all the SQL Scripts generated by reading the Target Database metadata. These are the actual script files that will be executed on the target database to override the production / non compliant environment parameters to more environment specific values. This directory also succeeds by the <ENV> and <SID> directories.

### B.3.1.1 Some Important Facts about Environment / SID Directories

- The current valid values for <ENV> are TEST and STAGE only.
- The values of <ENV> are case sensitive and the Environment directories must be created in Upper Case values only.
- The case of <SID> depends on the actual SID value of the database. Hence this directory name can be in upper case / lower case / alphanumeric value. Nevertheless this value should exactly match with the Target Database SID on which the operation is performed else application will not work.

### 3.6.3.2 Application Files

This section provides a list of mandatory files that are required for the application to run without errors. The contents of the DBSYNC folder are as follows:
**cleanTBSMoveAndRebuild.pl**: Perl script used to clear all the "ALTER TABLE MOVE" commands created by TOAD.

**Changification.sql**: Script to install mandatory database objects to the NON_PRODUCTION_USER or DEMO_USER. These objects are mandatory and are critical to the proper operation of the product. The table below describes detail list of objects installed by this script.

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Object Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFN_SCHEMA_GENERATE</td>
<td>TABLE</td>
<td>This has a list of schema's that exists in the lower Non production environments and have objects created in it. These are list of schemas that have not been promoted to production. Space DBSYNC application creates a script that will Create these Schema's and Export Import the objects in these schema. <strong>Note</strong>: As agreed upon the DBSYNC process only restores the structure for such schema's no data movement is handled in the current incarnation of the application.</td>
</tr>
<tr>
<td>CFN_SEQUENCES</td>
<td>TABLE</td>
<td>This table contains some critical information about the sequences whose current numbers will need to be altered to an environment specific OFFSET value. This table must be a preloaded table as a one time</td>
</tr>
<tr>
<td>Table Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CFN_SYS_SCHEMALIST</td>
<td>TABLE</td>
<td>This contains a predefined list of Exclusion schema (mainly list of Oracle SYS / SYSTEM schema) that will need to be excluded for all computations in the DBSYNC process. Make an entry into this table to exclude any schema from generating script files.</td>
</tr>
<tr>
<td>GENCREATEMVIEWSCRIPT</td>
<td>PROCEDURE</td>
<td>This is a stored PL/SQL routine that generates materialized view source code. Since the DBMS_METADATA for Mviews works only on an invoker rights model and since the AUTHID definition is restricted to a Procedure / Function only is the reason why this researcher has not been able to accommodate this routine into the main CREATEDBSCRIPT package.</td>
</tr>
<tr>
<td>CREATEDBSCRIPT</td>
<td>PACKAGE</td>
<td>This is a stored PL/SQL package that helps the DBSYNC process generate various object source codes.</td>
</tr>
<tr>
<td>CHANGIFICATION.Utility</td>
<td>PACKAGE</td>
<td>This is a stored PL/SQL package that helps the DBSYNC process write the output to files.</td>
</tr>
<tr>
<td>CFDIR</td>
<td>DIRECTORY</td>
<td>This directory controls the end destination where the CHANGIFICATION.Utility will create and write</td>
</tr>
</tbody>
</table>
SyncPreProcess.sh: This shell script contains all the pre process routines that will be run on the environment prior to refresh. This process collects all the vital configuration information specific to the environment database that is being refreshed.

SyncPostProcess.sh: This shell script contains all the post process routines that will be run on the environment post / after the environment refresh. This process also generates a series of executable SQL scripts from the new target database and also applies all the files generated by the pre process and the initial phase of the post process in a specific order. The order of the routines calling the execution functions are completely controlled by the preProcessEnv and postProcessEnv which will be discussed in more detail in the later sections.

B.3.3 Database Setup

B.3.3.1 Setting up the NON_PRODUCTION_USER or DEMO_USER

Script to Create NON_PRODUCTION_USER

Prompt User NON_PRODUCTION_USER;

--

-- NON_PRODUCTION_USER (User)

--

CREATE USER NON_PRODUCTION_USER

IDENTIFIED BY <PWD>

DEFAULT TABLESPACE <TBS_NAME>

TEMPORARY TABLESPACE <TEMP TBS NAME>

PROFILE DEFAULT
ACCOUNT UNLOCK;

-- 7 Roles for NON_PRODUCTION_USER

GRANT CONNECT TO NON_PRODUCTION_USER;
GRANT RESOURCE TO NON_PRODUCTION_USER;
GRANT EXP_FULL_DATABASE TO NON_PRODUCTION_USER;
GRANT IMP_FULL_DATABASE TO NON_PRODUCTION_USER;
GRANT SELECT_CATALOG_ROLE TO NON_PRODUCTION_USER;
ALTER USER NON_PRODUCTION_USER
    DEFAULT ROLE CONNECT, BROWSER_RL,
        SELECT_CATALOG_ROLE;

-- 10 System Privileges for NON_PRODUCTION_USER

GRANT CREATE TYPE TO NON_PRODUCTION_USER;
GRANT CREATE TABLE TO NON_PRODUCTION_USER;
GRANT CREATE PROCEDURE TO NON_PRODUCTION_USER;
GRANT SELECT ANY TABLE TO NON_PRODUCTION_USER;
GRANT ALTER ANY SEQUENCE TO NON_PRODUCTION_USER;
GRANT SELECT ANY SEQUENCE TO NON_PRODUCTION_USER;
GRANT CREATE ANY DIRECTORY TO NON_PRODUCTION_USER;
GRANT UNLIMITED TABLESPACE TO NON_PRODUCTION_USER;
GRANT CREATE PUBLIC SYNONYM TO NON_PRODUCTION_USER;
GRANT SELECT ANY DICTIONARY TO NON_PRODUCTION_USER;
Run the Changification.sql in the target database after creating the NON_PRODUCTION_USER. Run the script as SYS. The script will create the above list of the objects in the NON_PRODUCTION_USER and required PUBLIC synonyms and make the required GRANTS to these objects for other users to run these routines without having to logon to the NON_PRODUCTION_USER.

B.3.4 Configuration

B.3.4.1 Important Application Configuration Files – located in the DBSYNC/envFiles

<table>
<thead>
<tr>
<th>Permissions</th>
<th>Owner</th>
<th>Group</th>
<th>Size</th>
<th>Date</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-rwxr-xr--</td>
<td>1</td>
<td>oracle dba</td>
<td>4235</td>
<td>Jan 6 19:06</td>
<td>syncenv</td>
</tr>
<tr>
<td>-rwxr-xr--</td>
<td>1</td>
<td>oracle dba</td>
<td>385</td>
<td>Jan 6 19:06</td>
<td>sqlwrapstart</td>
</tr>
<tr>
<td>-rwxr-xr--</td>
<td>1</td>
<td>oracle dba</td>
<td>25</td>
<td>Jan 6 19:06</td>
<td>sqlwrapend</td>
</tr>
<tr>
<td>-rwxr-xr--</td>
<td>1</td>
<td>oracle dba</td>
<td>2020</td>
<td>Jan 6 19:06</td>
<td>preProcessEnv</td>
</tr>
<tr>
<td>-rwxr-xr--</td>
<td>1</td>
<td>oracle dba</td>
<td>1908</td>
<td>Jan 6 19:06</td>
<td>postProcessEnv</td>
</tr>
</tbody>
</table>

**syncenv**: Contains SID specific environment set up parameters.

**ORACLE_HOME**: Home directory where Oracle software binaries are installed.

**ORACLE_BASE**: Oracle binaries base directory

**OB / OH**: Setting aliases for Oracle Base and Oracle Home.

**LD_LIBRARY_PATH**: Oracle LD Library Path.

**PATH**: Oracle Binaries PATH setting.

**ORACLE_SID**: Oracle Target SID.

**Note**: This value must match the value in the first comment line. The common notation across the DBSYNC application to differentiate the entries between SID is delimited between the # /*
SID */ and # /* End */ all the parameters appearing within them are specific to the SID mentioned.

**Note:** All the configuration files will have one set of entries specific to SYNCDB. This is basically a template entry. The end users may either use this as a reference or replicate this entry and make the suitable changes for the application to uniquely identify the SID specific parameter demarcations.

**HOMEDIR:** DBSYNC application Home directory.

**ENVDIR:** DBSYNC application environment files directory path (referred internally by application.)

**SQLINDIR:** SQL File repository location.

**SQLOUTDIR:** SQL File destination location.

**OUTDIR:** Script output directory.

**TOADDIR:** Toad Sync Script location details.

**TEMP_PASSWORD:** Temp password that will be assigned to list of schema's that have DB Links. Later after creation of the Private Database links this password will be overridden with the original password.

**LOGMAILLIST:** List of mail-ids to mail the application logs after the process completes. Multiple mail-ids may be provided with comma separated values however this feature to send error report to multiple mail-ids has not been tested. This could also be replaced with a group mail-id.

**MGWCONNINFOUSERNAME / MGWCONNINFOPASSWORD / MGWCONNINFOENV:** These are the messaging gateway specific parameters that are user by
the `reOrgSIDMSGGateway` and `reOrgSID2MSGGateway` routines when we "exec `dbms_mgwadm.db_connect_info" to add the connection info.

**sqlwrapstart:** Contains list of standard sqlplus commands that will be wrapped to all the files at the start.

```sql
SET PAGES 0
SET LINES 1000
SET NEWPAGE 0
SET SPACE 0
SET HEAD OFF
SET FEEDBACK ON
SET PAUSE OFF
SET ECHO ON
SET VERIFY OFF
SET RECFSEP OFF
SET TERR OFF
SET DOCU OFF
SET TRIMSPool ON
TTITLE OFF
DTITLE OFF

define spool_file = "&1"
spool &spool_file
```

**sqlwrapend:** Contains list of standard sqlplus commands that will be wrapped to all the files at the end of file.

```sql
COMMIT WORK;
SPool OFF
```

**preProcessEnv:** Contains the list of the functions that will be run by the pre process and there specific order. The details of each routine will be discussed in the later sections.
postProcessEnv: Contains the list of the functions that will be run by the post process and there specific order. The details of each routine will be discussed in the later sections.
DBSYNC Pre Process Usage

Usage:

./SyncPreProcess.sh <"SID"> <"ENV">

ENV Valid Values "STAGE" / "TEST"
In the above example, the pre-process was run on a database SID it9 (SID Structure only database running in test database). Hence, the SID value of it9 is passed, which is the SID of the database.

The process generates all the SQL files in the below location.
DBSYNC Post Process Usage

Usage:

./SyncPostProcess.sh <"SID"> <"ENV">

ENV Valid Values "STAGE" / "TEST"

Running the post process is similar to running the pre-process with no exception. Please note the post-process on a database is running with SID value icl9. This is a structure only clone of the SID_UAT instances. So as an exception for the example to run smoothly, a copy of all files that
were generated by it9 in the pre process will be manually sent to the HOMEDIR/genSQLFiles/TEST/it9 to HOMEDIR/genSQLFiles/TEST/icl9. This exception is specifically because that is how the test environment for testing the DBSYNC application is configured. However in real time this is not the case. Since the real process would use the SAN Copy from prod / UAT instance to create these TEST / STAGE instances; for this example, it will be running the pre process and post process on the same machine with only exception that the preprocess will run before the SAN Copy and the post process will run after SAN Copy.

In the above example, the post-process was run on a database SID icl9 (SID Structure only
database running in test database). Hence the SID value of icl9 is passed which is the SID of the database.

The process generates / uses all the SQL files and associated schema dump files (which have been blacked out to protect identities) in the below location.

```
[133]:cd genSQLFiles/TEST/icl9/
[134]:ls -lrt

total 12572
-rw-r--r-- 1 oracle dba 10 Jan  6 14:24 start_msg_gateway.sql
-rw-r--r-- 1 oracle dba 137 Jan  6 14:20 shut_msg_gateway.sql
-rw-r--r-- 1 oracle dba 1573 Jan  6 14:28 reset_to_temp_password.sql
-rw-r--r-- 1 oracle dba 279656 Jan  6 14:28 refresh_stage_passwords.sql
-rw-r--r-- 1 oracle dba 141261 Jan  6 14:28 grant_quotas_to_stage_users.sql
-rw-r--r-- 1 oracle dba 14174 Jan  6 14:28 drop_stage_mview_logs.sql
-rw-r--r-- 1 oracle dba 2580 Jan  6 14:28 create_stage_users.sql
```

All the application output / exceptions and logs are recorded under the output directory into the SID specific sub directory as shown below.
The following code extract keeps the environment variables set for both the source and target schemas – this prevents incorrect settings and possible human-error to arise within the automated process

```bash
# /* SYNCDB */
ORACLE_HOME=/ora/app/Oracle/product/9.2/db1
ORACLE_BASE=/ora/app/Oracle
OB=$ORACLE_BASE
OH=$ORACLE_HOME
```
LD_LIBRARY_PATH=$ORACLE_HOME/lib
LIBPATH=$ORACLE_HOME/lib
PATH=$ORACLE_HOME/bin:$ORACLE_HOME/jdk/bin:$PATH/sbin:$ORACLE_BASE/admin/utils:$ORACLE_BASE/admin/utils/src:$PATH
SQLPATH=$OH/rdbms/admin:$OB/tns/ora_utils
JAVA_HOME=$ORACLE_HOME/javavm
ORACLE_SID=SYNCDB
HOMEDIR=/test/dbsync/Automate/DBSYNC
HD=$HOMEDIR
ENVDIR=$HD/envFiles
SQLINDIR=$HD/sqlFiles
SQLOUTDIR=$HD/genSQLFiles
OUTDIR=$HD/output
TOADDIR=$HD/toadFiles
TEMP_PASSWORD=TEMP
LOGMAILLIST=Tester@company.com
MGWCONNINFOUSERNAME=mgw_agent
MGWCONNINFOPASSWORD=mgwagt_TEST
MGWCONNINFOENV=SID_TEST
INITFILE_LOC=$HD
AUTOMATED DATABASE REFRESH

TOADCONTROLFILE=${HD}/toadCTLFile

SOURCECONN=SYNCDB_SOURCE

TARGETCONN=SYNCDB_TARGET

CLONECONNSTRING=SYNCCl

CLONEGLOBALNAME=SYNCCl

CLONEUSERPASS=CLONEPASS

TESTUSERPASS=TESTPASS

DMLAREA=/test/dbsync/

COMPILE_ITERATIONS=4

SYNCFILES=/test/dbsync

BINDIR=${HD}/bin

RUNTIME=${HD}/runTime

EXPORTMETHOD=exp

DBVERSION=11
Appendix C Toad Schema Compare Documentation

Toad is one of the key components of the DBSync process used to generate the schema sync scripts. The scope of this document will be to understand some of the usage and features of Toad to generate the schema sync scripts for the DBSync process. Toad supports three methods to generate the schema sync scripts.

- Command Line Mode
- GUI Mode
- Using Toad Scheduler

2. Record Toad Schema Sync as Action

Some of the key steps to record Toad Schema sync as an action are:

Step 1: Connect to source Database as DEMO_USER account.

Step 2: Connect to Target Database as DEMO_USER account.

Step 3: Choose the baseline Schema

Step 4: Choose the Target Schema

Step 5: Set up the Sync Files destination.

Step 6: Choose the Options for Schema Compare

Step 7: Save AS Action
Step 1: Connect to Source Database - DEMO_USER account.

Note the User name – Demo_User

Note the Service Name – Demo_DB_SRC
Step 2: Connect to Target Database as DEMO_USER account.
Step 3: Choose the baseline Schema
Step 4: Choose the Target Schema
Step 5: Set up the Sync Files destination.
Step 6: Choose Options Toad Schema Compare.

<table>
<thead>
<tr>
<th>Schemas</th>
<th>Options</th>
<th>Object Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Types to Compare</td>
<td>Object Type Specific Options</td>
<td>Misc Options</td>
</tr>
<tr>
<td>□ No Recycle Bin Obj</td>
<td>Libraries</td>
<td>Refresh Groups</td>
</tr>
<tr>
<td>☑ Clusters</td>
<td>Materialized Views</td>
<td>Scheduler Jobs</td>
</tr>
<tr>
<td>☑ Constraints</td>
<td>Comments</td>
<td>Scheduler Programs</td>
</tr>
<tr>
<td>☑ Check Constraints</td>
<td>Column Comments</td>
<td>Scheduler Schedules</td>
</tr>
<tr>
<td>☑ FK Constraints</td>
<td>Materialized View Logs</td>
<td>Tables</td>
</tr>
<tr>
<td>☑ PK Constraints</td>
<td>Outlines</td>
<td>Comments</td>
</tr>
<tr>
<td>☑ UK Constraints</td>
<td>Packages</td>
<td>Column Comments</td>
</tr>
<tr>
<td>☑ Composite</td>
<td>Partitions</td>
<td>TRIiggers</td>
</tr>
<tr>
<td>☑ DB Links</td>
<td>Policies</td>
<td>Types</td>
</tr>
<tr>
<td>☑ Dimensions</td>
<td>Policy Groups</td>
<td>Views</td>
</tr>
<tr>
<td>☑ Directories</td>
<td>Procedures</td>
<td></td>
</tr>
<tr>
<td>☑ Functions</td>
<td>Private Synonyms</td>
<td>Comments</td>
</tr>
<tr>
<td>☑ Grants</td>
<td>Public Synonyms</td>
<td>Column Comments</td>
</tr>
<tr>
<td>☑ Indexes</td>
<td>Queues</td>
<td>Column Procs/Scale</td>
</tr>
<tr>
<td>☑ Java Source</td>
<td>Queue Tables</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schemas</th>
<th>Options</th>
<th>Object Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Types to Compare</td>
<td>Object Type Specific Options</td>
<td>Misc Options</td>
</tr>
<tr>
<td>PL/SQL Objects</td>
<td>Tables</td>
<td></td>
</tr>
<tr>
<td>☑ Exclude Whitespace</td>
<td>☑ Compare Storage Columns</td>
<td></td>
</tr>
<tr>
<td>Exclude Double Quotes</td>
<td>☑ Compare &quot;ANALYZED&quot;</td>
<td></td>
</tr>
<tr>
<td>Exclude SQL Comments</td>
<td>☑ Compare Row Count with NUM_ROWS</td>
<td></td>
</tr>
<tr>
<td>Ignore Case</td>
<td>☑ Compare Row Count with Select Count(*)</td>
<td></td>
</tr>
<tr>
<td>List Stats</td>
<td>NULL=Blank for column data defaults</td>
<td></td>
</tr>
<tr>
<td>Format before comparison</td>
<td>Exclude Column Order</td>
<td></td>
</tr>
<tr>
<td>PL/SQL Objects</td>
<td>Sequences</td>
<td>Misc</td>
</tr>
<tr>
<td></td>
<td>☑ Compare Current Val</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Drop &amp; Create to Sync</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don't compare Manual</td>
<td></td>
</tr>
<tr>
<td>PL/SQL Objects</td>
<td>Triggers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>☑ Never parse for schema names</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don't compare schema name in description</td>
<td></td>
</tr>
</tbody>
</table>
### Automated Database Refresh

#### Script Options
- Always use `;` to end SQL statements
- Include "Set define OFF"
- Include SQLPlus 'Prompt' comments in script
- Include schema name in migration ddl
- Include Comments in Sync Script
- List Dependencies

#### Inclusion Options
- Only Compare Object Names like...
  - `%`
  - `IgnoreCase`
- Stop when # of differences reaches...
  - `1000`

#### Object Types to Compare
- Tablespace
- User Storage
- Logging/MetaLogging
- Cache/NoCache
- Compress/NoCompress
- Monitoring/NoMonitoring
- Degree
- Instances
- RetrIn
- RetiUsed

#### Object Type Specific Options
- InTrans
- MaxTrans
- Initial
- Next
- MinExtents
- MaxExtents
- PetIncrease
- PrivIds
- Freelist Groups
- Buffer Pool
2.7 Step 7: Save AS Action

![Save/Load Window Snapshot](image)

3 Executing To Generate Schema Sync Script

3.1 GUI Option

![GUI Option](image)
3.2 Command Line Option

```bash
D:\> D:\\> "D:\\QuestSoftware\TOAD\Toad for Oracle"

D:\\QuestSoftware\TOAD\Toad for Oracle>
D:\\QuestSoftware\TOAD\Toad for Oracle>
D:\\QuestSoftware\TOAD\Toad for Oracle>
D:\\QuestSoftware\TOAD\Toad for Oracle>
D:\\QuestSoftware\TOAD\Toad for Oracle>TOAD.exe -a"DEMO_APP"
```

3.3 Toad Scheduler Option
AUTOMATED DATABASE REFRESH

Schedule Action Item

Enter the path to Toad

Application: \{Path\}

Parameters: "DEMO_APP"

Work Dir: 

Comments: 

Schedule Action Item

Enter the task name and execution frequency

Name: DEMO

Frequency:
- Daily
- Weekly
- Monthly
- One-time only
Schedule Action Item

Enter task schedule

Start time: 10:09:03 PM
Start date: 01/20/2009

Perform:
- Every day
- Every [ ] days

< Back  Next >  Cancel