Client - Driven Dynamic Database Updates

by

Preetika Tyagi

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved September 2011 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
K. Selçuk Candan
Hasan Davulcu

ARIZONA STATE UNIVERSITY

December 2011

ABSTRACT

This thesis addresses the problem of online schema updates where the goal is to be able to update relational database schemas without reducing the database system's availability. Unlike some other work in this area, this thesis presents an approach which is completely client-driven and does not require specialized database management systems (DBMS). Also, unlike other client-driven work, this approach provides support for a richer set of schema updates including vertical split (normalization), horizontal split, vertical and horizontal merge (union), difference and intersection. The update process automatically generates a runtime update client from a mapping between the old the new schemas. The solution has been validated by testing it on a relatively small database of around 300,000 records per table and less than 1 Gb, but with limited memory buffer size of 24 Mb. This thesis presents the study of the overhead of the update process as a function of the transaction rates and the batch size used to copy data from the old to the new schema. It shows that the overhead introduced is minimal for medium size applications and that the update can be achieved with no more than one minute of downtime.

DEDICATION

To My Family & Friends...

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Dr. Rida Bazzi for his continued guidance and support during my association with him as a Master's student. I learned many valuable lessons during this time that will be helpful to me in my professional pursuits in the future. I am grateful to Dr. Candan for his valuable suggestions that helped me grasp a good insight of the concepts during my thesis. I would like to thank Dr. Davulcu for being my committee member and guidance. I am also thankful to all my friends Archana, Sushovan, Amrit, Kanika, Ina, Shruti, Raj, Jyothi Swaroop, Anu and Vidhi for their constant help and encouragement. I am grateful to the faculty members and administrative staff for their support throughout the duration of my master's degree, here at Arizona State University. Last but not least, I owe gratitude to my parents, who always believed in me and provided invariable support. This work was supported in part by National Science Foundation grant CSR - 0849980. The findings do not necessarily reflect the opinion of NSF.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The relational data model is the most widely used model for database systems [8]. In the relational model [26], the database schema defines the structure of the data stored in the database and affects how queries are written to access the database. In addition to database schemas, databases have integrity constraints to ensure data accuracy and consistency. Database schemas and their associated constraints are not static entities; they can be modified (*upgraded*) to adapt to changed business needs, eliminating data redundancy through normalization, or improving system performance through data reorganization [30].

Traditionally, and except for some trivial changes [24], upgrading the database schema requires making the database unavailable while the data is *ported* from the old to the new schema. When the data is completely copied and formatted according to the new schema, the new database is started and the (new) application can resume accessing the database system. This process can cause downtime of a few hours for medium-size databases. Such upgrades are costly. According to a study of the estimated financial impact of outage on several industries, such as retail brokerage, credit card sales authorization, catalog sales centers, airline reservation centers, and ATM services fees, the cost of downtime varies from $14,500 an hour for the ATM service fee industry to $6.5 million an hour for the retail brokerage industry [23].

To mitigate the effects of system unavailability, system upgrade is typically done at a time when the system is least busy (2 am on a Saturday for example). Unfortunately, for some applications, there is no good time for upgrades. For example, in the healthcare field, a hospital needs to have its database system always available. This need is especially pronounced for emergency rooms; in order to admit patients and avoid negative drugs interactions, for example, physicians need to have continuous access both to lookup patient information and to update patient records (read/write access). Also, for systems that are accessed by worldwide clients, 2 am on a Saturday, does not exist (due to time zone differences)! For such applications, there is a need to minimize or completely eliminate, if possible, the time needed to apply an upgrade. It is schema upgrades for such applications that this thesis is addressing.

To minimize system downtime, clients[1] should have access to the database while the upgrade process is ongoing. This means that while the upgrade process is ongoing, clients should still be able to modify existing records and insert new records in the database, while at the same time be able to access records for reading without violating the database consistency. In this thesis, we consider updates that are *atomic*. This means that, as far as the client can tell, there is a period of time in which the database behaves like the old database and at a given point in time the database behaves like the new database. Atomic database updates require atomic client updates. In addition to switching between the old and the new database, the clients themselves should change from old version to a new version that knows how to interact with the new database. We assume that clients are written in such a way that they can switch to executing new client code when they detect that a new version of the database is available. The focus of this thesis is on developing the mechanisms needed to support online schema upgrades without the need to modify the database management system. Unlike other works, we adopt a client-driven approach. Given a mapping between the old and the new schemas, a specialized *update client* is automatically generated to: (1) prepare the database for update; (2) create a database according to the new schema; (3) copy data from the old to the new database; (4) deactivate the old database; and (5)

[1]We use the term *clients* to refer to the software processes that interact with the database system.

activate the new database. We use techniques developed by Løland [21, 20] to effect the dynamic database update. Unlike [20], our approach is client driven, whereas the approach of [20] requires the modification of the DBMS system, which makes it less flexible and non-portable.

We have shown that our approach is viable by implementing a prototype system that supports schema transformations including horizontal merge (with and without duplication), vertical merge (full outer join), intersection and difference, and horizontal and vertical splits. Our system automatically creates an *update client* using a description of the schema changes provided by the user and then runs the update client to dynamically apply the database update. We tested our system for a variety of update types and our performance results show that a client-driven approach is a viable approach for dynamic database updates. In Chapter 5, we demonstrate the results for a database of about 3.3 million records and 11 tables with an average of 5 columns and achieve the update with the downtime of less than one minute.

To summarize, the contributions of this thesis are the following:

- The first design and implementation of a DBMS-independent client-driven dynamic database update system.
- A thorough evaluation that shows the viability of the approach for dynamic database updates.

The rest of the thesis is organized as follows. Chapter 2 describes related work and compares it to our work. Chapter 3 illustrates the mechanism of client-driven dynamic database update framework. Chapter 4 covers the schema transformations supported by the framework. Chapter 5 gives an evaluation of the approach and presents the results. Chapter 6 concludes our work with a possible future direction of the work.

Chapter 2

RELATED RESEARCH

This chapter describes the relevant work in the area of relational schema changes with the main focus on the live database updates. Our method extends some of the ideas discussed here and we point that out in following chapters. We also summarize the features provided by various systems for relational schema evolution in table 2.1.

The most closely related work in its aims is that of Løland [20]. He designed and implemented a database management system prototype capable of performing non-blocking schema transformations. He supports six complex schema transformations including horizontal merge, vertical merge, horizontal split, vertical split, difference and intersection. The proposed technique has three phases: initial population, log propagation and synchronization. This is the same approach we followed, but in our approach, we achieve all these phases without any modification to database management system (DBMS). We believe that the difference is very significant as it allows the implementation of dynamic database updates on any DBMS that supports standard SQL functionality. Unlike our work, Løland supports mixed use schemas in which old and new schemas can coexist for some time.

Another work that attempts to provide non-blocking schema transformations is that of Ronström [27]. He presented a general framework based on triggers, but did not provide an implementation or an evaluation. His use of triggers differ from ours in that in his approach triggers are used to keep both schemas synchronized and are not used for maintaining a log table. His use of triggers is very expensive. While he did not present performance numbers, Løland [20] presented an analytical study of the overhead which indicates that his approach will result in overhead that is at least two or three times higher in terms of operations and the use of locks. Our approach can be thought of as a combination of the two approaches: we use triggers to provide a client-driven framework that provides the advantages and smaller overhead of [20] while providing the flexibility of the approach proposed in [27].

More recently, Mark Callaghan at Facebook [4] open sourced a tool for Online Schema Change (OSC) in the MySQL database system. It has four phases - *Copy*: The source table is copied to the output table. *Build*: The output table is modified to satisfy the new schema requirements. *Reply*: The source table changes, which occurred during the previous steps, are propagated to the new table. *Cut over*: The old and new tables are synchronized and clients are given access to the new table. This procedure supports the schema change of adding a column to a table with a default value and uses external (PHP) script for the implementation. This approach is limited because it does not use stored procedures like we do. In our approach, a Java client connects to the database to install the stored procedures required for the update process. The advantage of using stored procedure is that they run within the database engine. Therefore, it reduces the network related overhead which allows us to provide an efficient framework for implementing dynamic database updates for complex schema transformations.

Curino et al. [10] presents a workbench to predict and evaluate the effect of schema changes and also to perform schema evolution. It is useful for database administrators to gain the insight of a specified schema evolution process in terms of cost. As the result, the necessary changes can be made in the database to minimize the cost of schema evolution in databases. This framework also supports the query rewriting of the old queries to the new queries that can access the new schema [9].

A few commercial DBMSs have been working on providing the support for online and/or offline schema changes. Oracle [31] provides a feature *redefinition* to support online schema evolution capability, such as, add/drop/rename a columns and create/rebuid index. Redefinition process makes it possible to perform changes in a table while the users are allowed to access the table. The table is made offline only when the table is switched to the new updated table. IBM DB2 [19] provides a tool *Optim Data Studio Administrator* that allows the administrator to make changes in the schema using an interface. It also includes the functionality of verifying the impact of any changes in the schema and if it detects error due to intended schema changes, the user is notified before committing any changes. DB2 also provides the feature for performing online schema changes [11], such as, rename column and change column data type, by using the routine *ADMIN_MOVE_TABLE*. SQL Server [29] provides a tool *SQL Server Management Studio (SSMS)* by using which the user can perform schema changes. It includes GUI where the user can edit the diagram to define the required schema changes, such as, adding constraints and dropping columns. SSMS tracks all the changes made to the diagram and generates a script to propagate these changes to the database schema. The framework presented in this thesis includes complex schema transformations, for example, horizontal/vertical merging and splitting of tables.

Hartung and Terwilliger [15] describe the recent work in the relational and XML schema evolution along with the ontology evolution. It outlines general desiderata for schema evolution support, such as, rich set of schema transformation and minimizing the degradation of user transactions. Some researchers focused on the area of schema evolution in database systems along with the associated dependencies. Sometimes, the system imposes the restriction on the schema evolution in the presence of dependencies. Papastefanatos [25] developed an open-source tool *HECATAEUS* for enabling impact predictions and regulations of relational database schema evolutions. *HECATAEUS* includes an approach to propagate the schema changes to the dependent objects as well. The complete system is represented as a graph. The nodes of the graph represent the database objects and the associated interacting units (views, queries) and the schema evolution is simulated on the graph to study the impact of any changes on the other parts of the graph. The user can define the behavior of the changes – propagate or block the changes to the dependent objects, or prompt the user to ask for each change.

Hick and Hainut [16] present an approach to evolve the relational schema by using a conceptual model. *DB-MAIN* [28], a modeling framework, is used to draw the conceptual model for a relational schema and then the conceptual model is edited to perform the schema changes. These changes are traced down to propagate to the relational schema. Domínguez [12] presents another similar tool *MeDEA* to represent the relational database schemas as the conceptual models where edits can be performed and propogated to the relational schema. Cleve [7] worked towards providing a conceptual view of relational database along with a set of data manipulation APIs for data-intensive applications. These data manipulation APIs provides a mapping between the conceptual view and the relational schema. Whenever relational schema changes, there is an automatic generation of these APIs so that application programs are protected against schema changes. Terwilliger and Bernstein [32] present an Object-Relational Mapping (ORM) System *MoDEF*, an extension to Visual Studio that allows a developer to build a client model and map it to an existing store. *MoDEF* captures all the changes made to the client model and updates the mapping and the store model without additonal user input. Bounif [2] proposed an approach for smooth schema evolution by predicting possible future requirements. The problem of online scaling in databases has been studied by Bratsberg [3]. It targets declustering evolution in databases due to changes in data volume, processing power and access patterns.

A few researchers also addressed the problem of schema evolution in other data models, such as, object oriented, XML and RDF. Claypool [6] proposed *SERF* framework to perform complex schema transformations in object oriented database (OODB) systems. Monk [22] proposed a class versioning approach for schema evolution in OODBs. Oracle [1] provides the support for XML schema evolution. Chirkova [5] studied the problem of schema evolution for RDF data models.

Table 2.1 outlines certain features provided by various systems for relational schema changes including our client-driven framework. The description of these features is as follows.

1. DBMS modification: To support live schema changes, DBMS may need to be modified. It might be complicated to modify DBMS for such requirement. Alternatively, an application level mechanism can be developed on the top of DBMS. It doesn't require to deal with internals of a DBMS. It signifies the easiness with which a schema change feature can be integrated with DBMS to support schema evolution. However, there may be a scope of more efficient solution in case the DBMS is modified to support schema evolution mechanism.

2. Richness: it refers to the set of schema transformations supported by a schema evolution system. We categorize it into simple and complex schema transformations. Simple Simple schema changes include adding attributes, renaming attributes and dropping attributes whereas complex schema changes include merging and splitting of tables.

	DBMS modification	Richness		Online	Versioning
		Simple	Complex		
Client-driven framework	–	–	√	√	–
Løland	√	–	√	√	√
Ronström	–	–	√	√	√
OSC	–	√	–	√	–
Oracle	√	√	–	√	√
SQL Server	√	√	–	–	–
IBM DB2	√	√	√	√	–
PRISM	–	√	√	–	√
HECATAEUS	–	√	–	–	–
DB-MAIN/MeDEA	–	√	–	–	–

Table 2.1: Features provided by various systems for relational schema evolution

3. Online: it refers to the capability of a system to perform schema evolution while the database is still accessible to the users. There may be a short span of time during which the database is temporarily made unavailable to users. This short duration may be acceptable as compared to couple of hours that may cause a large impact on several industries. As we mentioned in previous chapter, this is important for highly available systems which have high cost of downtime.

4. Versioning: when a database schema evolutes to a new schema, we can either remove or maintain the old version of schema. If the old version of schema is removed, only new client applications can access the database. In case both versions of schema are maintained, both old and new client applications can access the database by using the corresponding version of schema. However, it imposes the requirement of maintaining the consistency of the database which is modified by both versions of client applications.

Chapter 3

CLIENT-DRIVEN DYNAMIC DATABASE UPDATE FRAMEWORK

In a database system, the database *schema* specifies the logical organization of the data as seen by the user. The same data can be organized according to different schemas, however, the schema design influences the business requirements and the database performance. Therefore, schemas do change to capture new business requirements and improve the database performance. In this thesis, we deal with relational database only. Relational schema consists of tables, columns, indices, procedures, functions, triggers and relationships (for example, foreign keys). The goal of dynamic database update is to achieve schema update with little or no down time. We define two models for achieving dynamic schema update. In one update model, we might allow queries that work with the old schema and queries that work with the new schema to access the database. We call this update model the *mixed update model*. The mixed model requires that two copies of the database be maintained one according to the old schema and one according to the new schema with extra synchronization (that might include internal tables not visible to the user) between the two copies to ensure access consistency. In another update model, we do not allow the mixing of queries. Before the update, the database is accessed by old queries and, after the update, the database is accessed by new queries. We call this update model the *pure* update model. The pure update model has the advantage of being simpler to reason about. It avoids the intricacies of preserving the consistency of both the old and new databases due to interaction of old and new queries.

In this thesis, we present a dynamic database update framework based on the *pure* model. Before the update, applications access the database of the old schema and after the update applications access the database of the new schema.

Since queries (to access data) depend upon the logical organization (schema) of data, the same set of queries may not work for a changed (new) schema and need to be modified along with the schema change. However, we restrict our attention to the dynamic schema updates only. The update is dynamic such that the update process does not block ongoing transactions on the database of the old schema while the update process is ongoing. Only when the system is ready to switch from the old schema to the new schema is the database system made unavailable for a short duration of time. Mullins [23] present the details on the acceptable downtime for the highly available systems (*five nines*) and defines the five minutes as the acceptable downtime per year. The goal of this thesis is to present a generic portable framework for dynamic database updates in relational databases along with minimizing this downtime.

The framework we propose is built upon and shares many similarities with the work of Løland [20]. Løland's approach asynchronously copies data from the old database tables to the new database tables while at the same time keeping track of all transactions on the old database. When the copying is complete, the log of the transactions that were executed during the copy phase is applied to the new database to synchronize both the databases. This process can repeat because the log can keep on growing during the synchronization process. Only when the log is small enough, the old database is made unavailable and a final fast synchronization is done, at which time the new database is up to date and can be made available to clients. This thesis presents the results with a downtime of less than one minute.

Unlike Løland's work, our approach does not require the modification of the database management system (DBMS) and is supported fully at the application level through a specialized *update client* (client-driven update). We call our framework the *client-driven framework* for dynamic schema updates. The client-driven framework has the advantage of being fully portable and more flexible than a DBMS-bound implementation. Given that we do not have access to the DBMS internals, we need to drive all the phases using the update client. At a high level, this is done as follows:

1. Creating the log table: since we do not have access to internal log tables, we need to create and maintain customized log tables at the application level. Maintenance of the log table is achieved using triggers (stored procedures) that are also inserted at the application level on the old database tables by the update client. The additional advantage of creating the user defined log table is that it has the scope of being modified to support any enhancements or modifications in the framework if required. Note that we apply the triggers on those old database tables which are involved in the schema transformation (database update) process.

2. Creating the new schema table(s): this step simply creates new tables that will be the target for copying information from the old database to the new database. When the copying is complete (in a sense to be explained later), the old tables are made inaccessible and the new tables are made accessible to the user application.

3. Creating auxiliary tables: these tables are required during the database update process while mapping the data from the old database to the new database and as well as during the synchronization phase(s).

4. Activating triggers on the old schema table(s): the *row - level* triggers that are activated keep track of every transaction (query) on every record of the old database. The record of these transactions is maintained in the created

log tables. Note that it will impose a small overhead on the user transactions that access the old database tables.

5. Asynchronous copying of data: this step is driven by the update client using *stored procedures* in which the data from the old database tables is transferred (copied) to the new database tables through a mapping process depending on the defined schema transformation.

6. Synchronizing copied tables: this step is also driven by the update client (using *stored procedures*) and reconciles the log table to reflect the changes in the new database tables. The details of this phase are described later in this chapter.

The phases of the dynamic update can be divided into three main phases: (1) setup, (2) initial transfer of partial data, and (3) synchronization. We explain each of these phases in details in what follows. Then we explain how the update client is created.

3.1 Update Setup

The setup phase prepares the database for being dynamically updateable. One field *myid* is added to all the old database tables (involved in schema transformation process) as part of the setup phase. This field is a unique identifier for each record in a table and records have their *myid* fields updated so that they are initially unique. Subsequently, the *myid* field is created for every new record by incrementing the highest value of *myid* previously created. The log table is also created as part of the update setup. This is a global table that reflects all transactions to the database that happen during the copying phase. We also create the *log maintenance triggers* that will update the log table (see below) whenever a record is modified in the database due to user transactions. These triggers are initially inactive.

Another part of the setup includes creating tables for the new schema as well as auxiliary tables to be used in the copying and synchronization phases. The update client executes several *stored procedures* during the update process which are generated and compiled during the setup phase. These *stored procedures* define the mapping mechanisms of the data from the old schema to the new schema. The information of the old and new schemas can be fed into the configuration files. Then the framework uses these configuration files to automatically generate the *stored procedures* required for the update process. The update client must be an administrative user in order to perform the updates in the database. This is necessary to avoid any permission issues while performing several activities involved in update process, such as, apply triggers, data transfer among the tables, compile stored procedures and manage access rights of the users on the databases. The information of all the current users is also provided to the framework by using configuration files to control their access to the database while switching the database from old schema to the new schema. The users may access the database of old schema before switching to the new schema. The update client does not provide access to the database of new schema unless the update process is completed and the system is ready to switch to the new schema. The final part of the setup includes activating the log maintenance triggers.

3.2 Initial Transfer of Partial Data

This phase comprises transferring the data from the database of old schema to the database of new schema while the old database is available to users. Since log maintenance triggers are activated before this phase, any insert, update, or delete of old database records by user transactions will be captured in the log table. In this phase, two copies of the database co-exist: the data according to the old schema which is continuously consistent during this phase and the data according to the new schema but this data is not consistent during this phase [17]. Only the data of according to the old schema are visible to and accessed by user transactions during this phase.

As a first step, the number of records that need to be copied should be determined. This is achieved by querying the database for the largest value of the *myid* field in a given table. Once the number of records that need to be copied is determined, the manner of copying needs to be established. That manner can drastically affect system performance. Records can be copied individually or in batches of records. We introduce two parameters that allow us to control the overhead of the update process: (1) the batch size used in copying from old tables to new tables, and (2) the time interval that separates copying two batches.

The transfer process reads the records in batches and holds read-lock on a single batch at a time. The smaller the batch size, the less interference there is between the copying process and ongoing user transactions. But the smaller batch size results in a longer time for the whole copying process and more time for the synchronization phase (see below).

The *inter-batch time interval* is the time gap between the processing of each batch. The upgrade process halts during the *time interval* and does not interfere with the concurrent user transactions. Thus larger values of *time interval* reduce interference with the concurrent user transactions, but increase total update time.

Note that since log maintenance triggers are activated before the copying starts, there is no risk of losing any records. The worst that can happen is that some records that are copied will also show up in the log table.

3.3 Synchronization

This phase applies the log entries to the copied tables to synchronize the data of the old and the new schemas. During the synchronization phase, user transactions are not automatically blocked. They are only blocked if it is estimated that the time it takes to process *all log entries* is smaller than a configurable threshold. If that time is estimated to be larger than the chosen threshold, user transactions are allowed to proceed which means that the log table would continue on growing while the logs are processed. So the log table is processed in multiple iterations. The size of the iteration is determined by the number of transactions that occurred while processing the old iteration. So, we assume that the transaction rate is such that the synchronization process will catch up with the growth of the log table so that each iteration is smaller than the previous one.

To estimate the time it takes for one iteration, we determine the number of records that need to be processed and compare it to the number of records of the previous iterations, with the assumption that the processing time is proportional to the number of record. The processing time of the last iteration is therefore used to estimate the time for the next iteration. In the last iteration, user transactions are blocked and at the end of the processing of the last synchronization iteration, the old and new databases are completely in sync and the system is ready to be switched to the new database. The users are given access to the new database.

In our experiments, we maintained the database of the old schema to compare it with the database of the new schema immediately after switching to the new schema. we distinguish between user data fields that capture data of interest to the user and database data that is not of interest to the user but that is used to link various tables, such as referential constraints and functions. We do not consider applying the constraints or functions on the new schema, dropping the old database tables or the additional columns appended to the new database tables (for the update process) while measuring switching time. In our implementation, we chose the batch size and inter-batch time interval so that the downtime was less than one minute.

Note that the *myid* field is crucial to synchronizing the logs table. Løland [20] used record identifiers (RID) and log sequence number (LSN) to propagate the log entries to the new database tables. RID uniquely identifies a record in the old database and is used to handle the insertion or deletion of the records in the new database. Moreover, if a record is updated in the old database, RID doesn't change and then LSN is used to identify if that particular record has already been updated in the new database. Our framework achieves the same goal by using a different strategy. The combination of *myid* and *old database table name* uniquely identifies a record in the old database whereas the content (field values) of the record itself (in *log table*) is used to identify if the record has already been updated in the new database. Entries in the *log table* contain the following information: (i) The old database table name, (ii) the entry type (insert/delete/update), (iii) the data record, (iv) the old data record if the entry type is *update*,(v) the value of the *myid* column in the old database table, and (vi) the schema transformation in which the old database table is involved (for example, vertical split). The framework attempts to apply an *update* type entry to the new database only if it finds a record that matches with the old data record in the log entry. If it doesn't find such record in the new database, it implies the fact that the current *log table* entry has already been applied to the new database. Here *myid* solves two purposes – (i) it is used in *Initial Transfer of Partial Data* to control the overhead of the system as mentioned earlier, and (ii) it is used in mapping mechanism of the schema transformations.

Although the new database tables may have *myid* as additional fields as the result of the previous update process, however, it can't be used for further database updates next time. When we append *myid* column to the old database table during the setup phase, it automatically increments its value and assigns an unique value for each new record. However, it doesn't hold the same property in the new database table. Therefore, it will impose an overhead on the user transaction coming to the new database since each new record will need to calculate the new *myid* column value that doesn't conflict with an existing value. In addition, these *myid* field values may not fit into the mapping mechanisms of some schema transformations. Similar to Løland's [20] use of LSN, we also have an option to maintain application-level LSN to handle the *update* type log entries by using triggers, however, it will impose an overhead on the user transactions to maintain the latest LSN value for each updated record in old database. The next chapter illustrates the schema transformations and provides an overview of how *myid* and *log table* entries are used to apply the changes to the database of the new schema.

Figure 3.1 presents an overview of the schema transformation framework and illustrates the steps involved in the dynamic database updates process.

- Step A: This represents the initial state of database server before the database update process begins. Database is being accessed by several concurrent user transactions.

- Step B: A special client (update client) connects to the database and initiates the process of database updates while database is still being accessed by ongoing user transactions. It maintains the hybrid state of the database i.e. databases according to both the old and the new schemas are maintained internally. This step involves all the tasks illustrated in the phases,

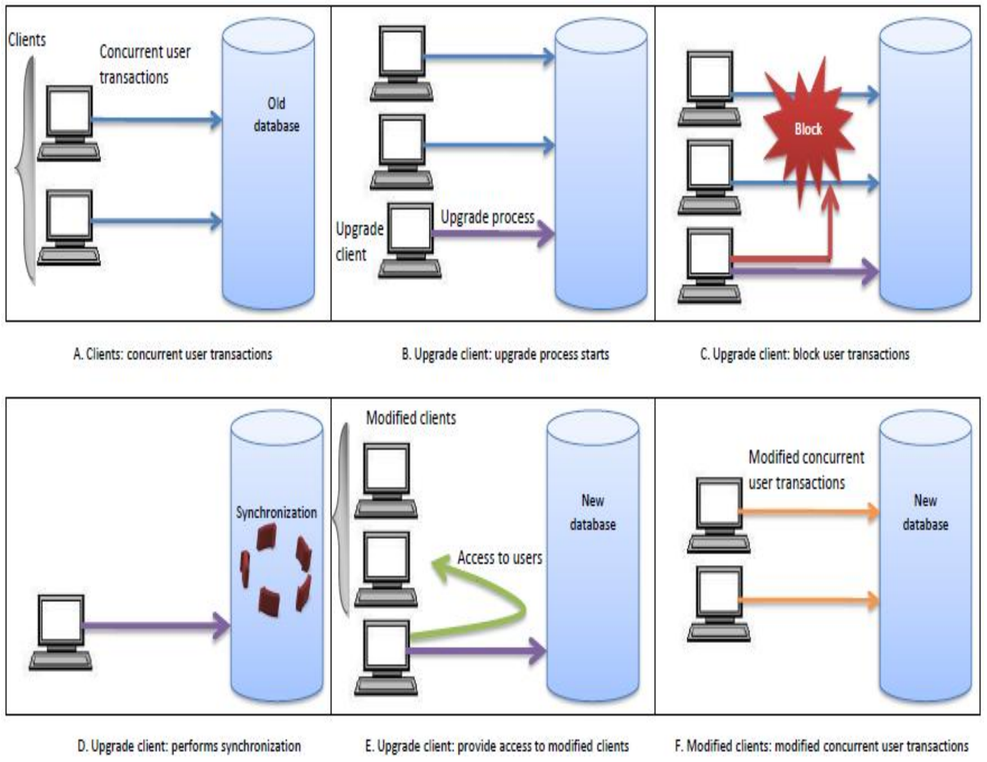

Clients
Concurrent user transactions
Old database
Upgrade client
Upgrade process
Block
A. Clients: concurrent user transactions
B. Upgrade client: upgrade process starts
C. Upgrade client: block user transactions
Synchronization
Modified clients
Access to users
New database
Modified concurrent user transactions
New database
D. Upgrade client: performs synchronization
E. Upgrade client: provide access to modified clients
F. Modified clients: modified concurrent user transactions

Figure 3.1: Client-Driven Dynamic Database Update Workflow

Update Setup, *Transfer of Partial Data* and *Synchronization*, before blocking the user transactions.

- Step C: The update client blocks ongoing user transactions since old and new databases are required to be synchronized.
- Step D: This step performs the last iteration of the *Synchronization* phase after blocking the user transactions. Old and new copies of databases are synchronized by applying the last chunk of the *log table* entries.
- Step E: Blocked users are given access to the new database. Here the assumption is that the client applications have been modified to access the new database.

- Step F: Update client is removed and modified user transactions resume on the database.

To validate our client-driven dynamic database update framework, we implemented it with support for a number of schema transformations. We implemented the following schema transformations: Horizontal Merge, Vertical Merge, Horizontal Split, Vertical Split, Difference and Intersection [20]. The description for each schema transformation is provided in next chapter.

3.4 Creating the Update Client

The update client is created using the description of a schema transformation. The description is a simple text file that specifies the tables involved in a particular transformation, the necessary fields (as appropriate) and the type of transformation. In our current implementation, we give the user a rudimentary interface to specify the information and the text file is generated from the provided information. The update client is then generated from the text file. The client is simply a generic implementation that is parametrized with the table and fields names.

3.5 Handling Failures

The current version of the framework doesn't support the automated detection of the failure cases and rollback of the update process. Here the failure case stands for any interruption occurred during the update process. If any failure case happens, all the tables and functions (included in the update process and the new schema) can be removed manually or by writing a simple SQL script. Then the update process can be restarted without creating any inconsistency in the database of the old schema.

Chapter 4

SCHEMA TRANSFORMATIONS

The framework introduced in Chapter 3 expects the user to provide for a given schema change stored procedures for (1) initial copying from the old database to the new database and (2) applying log table entries to reconcile the copied tables with the log. As we explained in Chapter 3, these stored procedures are generated by the system using input provided by the user. In this chapter, we give the details of what these procedures look like for a number of schema transformations that we support in our system. These schema transformations are – Horizontal Merge, Vertical Merge, Horizontal Split, Vertical Split, Difference and Intersection. Each schema transformation process has two phases of data mapping from the old to the new schema tables. Phase I includes transferring the data from the old schema table to the new schema table whereas phase II includes propagating the data from the *log table* to the new schema table. This chapter illustrates both phases of mapping for all the schema transformations.

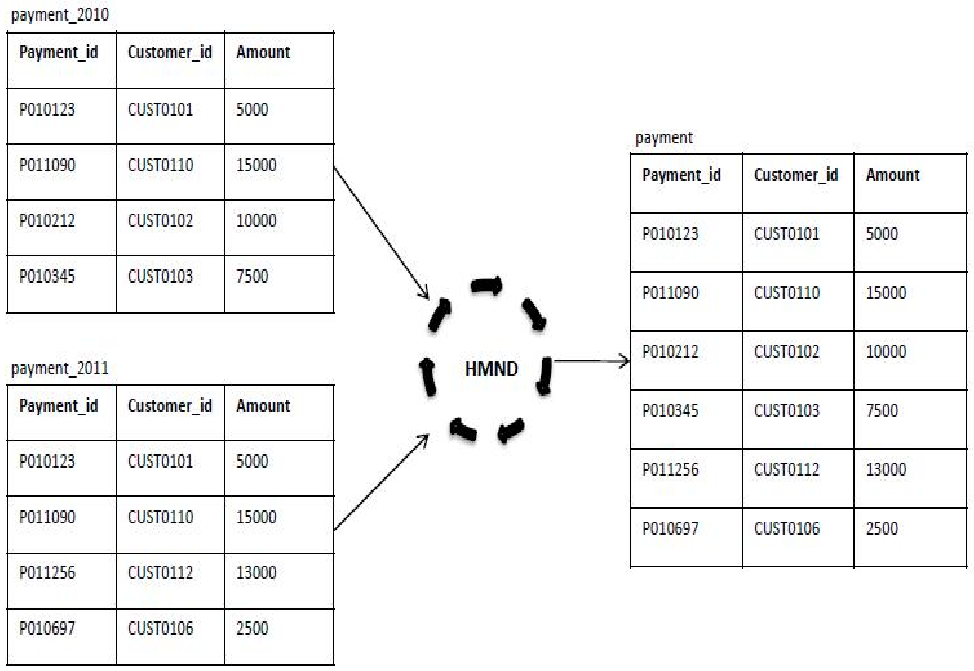
This thesis includes all the schema transformations that are presented in [20] along with an additional schema transformation. However, we have a different set of assumptions for some of the schema transformations. For example, we assume a primary key column in old schema tables for horizontal merge. Hence we compare the primary key column values to determine if two records are duplicate whereas Løland [20] compares all field values to determine duplicity. The mapping mechanisms are different from [20] for all schema transformations mainly due to our client-driven approach, but the handling of the transformations is essentially that of Løland [20].

In general, the initial copying phase is a straightforward and, with the exception of vertical merge, horizontal merge with no duplicates, difference and intersection, only requires knowing how columns in the old schema map to columns in the new schema. Horizontal merge (without duplicates), vertical merge, difference and intersection requires creating an auxiliary table to support initial copying. The log reconciliation phase is more complicated though. It requires auxiliary tables to support reconciliation for horizontal merge (without duplicates), difference and intersection. For some transformations, new columns are created in the target tables to support initial copying and the log reconciliation phases. These columns are dropped before the new database is made available to the user. Also, for some transformations we make the assumption that the old schema tables have primary keys. In what follows we present how each transformation is handled. As we mentioned for all but one case is identical to the work of Løland [20], but we include it for completeness.

4.1 Horizontal Merge

Horizontal Merge is equivalent to the *union* operator in the relational databases. It combines records of two tables having similar structure. There are two ways of merging records. One is to maintain all duplicate records in the new schema table whereas another excludes duplicate records from the new schema table. We implemented horizontal merge with and without duplicates. We assume that the old schema table has a primary key column and, hence, there are no duplicate records in each of the tables being merged. However, the two tables being merged might have records that are identical.

Horizontal Merge with No Duplicates (HMND): HMND excludes duplicate records from the new schema table. An auxiliary table is created consisting of the following columns – the primary key column as in the old schema table and the *old schema table name* column. HMND inserts and searches for the records in the auxiliary table during the merging process. The two records in the new schema table are considered duplicates if they have the same value for the primary key column. This assumption is different from Løland [20] where two records are considered duplicate when all the column values are identical for both the records. Figure 4.1 demonstrates an example of HMND. There are two old schema table – *payment_2010* and *payment_2011*. The new schema table consists of all records from both old schema tables and excludes duplicate records with primary key column (*payment_id*) value as *P010123* and *P011090*. Below is the description of mapping records in both phases.



payment_2010

Payment_id	Customer_id	Amount
P010123	CUST0101	5000
P011090	CUST0110	15000
P010212	CUST0102	10000
P010345	CUST0103	7500

payment_2011

Payment_id	Customer_id	Amount
P010123	CUST0101	5000
P011090	CUST0110	15000
P011256	CUST0112	13000
P010697	CUST0106	2500

payment

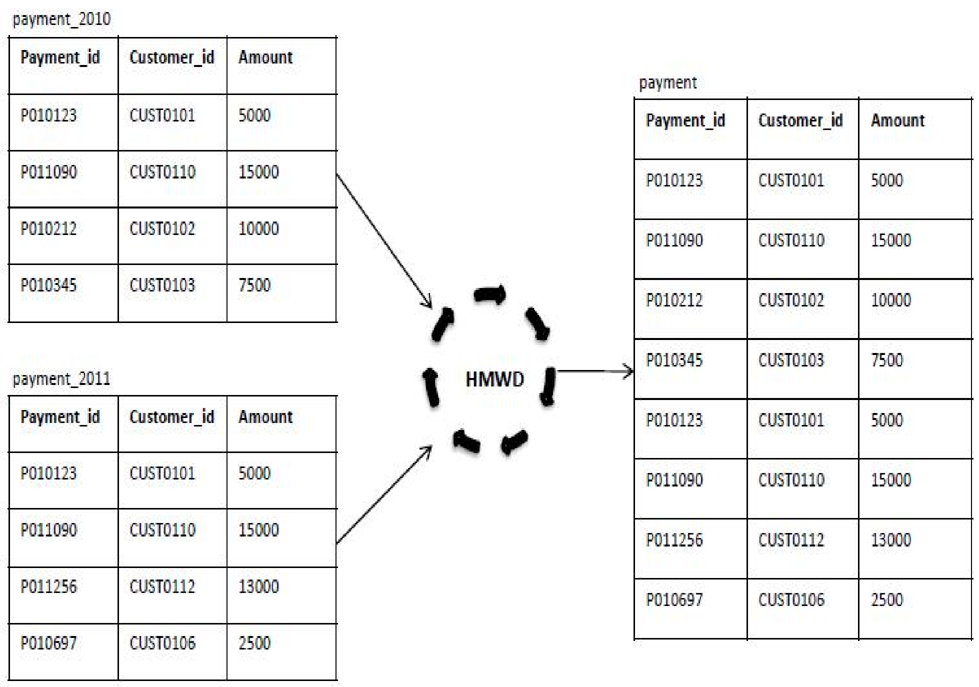
Payment_id	Customer_id	Amount
P010123	CUST0101	5000
P011090	CUST0110	15000
P010212	CUST0102	10000
P010345	CUST0103	7500
P011256	CUST0112	13000
P010697	CUST0106	2500

Figure 4.1: HMND: Horizontal Merge with No Duplicates

Phase I: We scan all records of both old schema tables. For each old schema table record, search for the record *r1* in auxiliary table by using the primary key value and *old schema table name*. If the record *r1* does not exist in the auxiliary table, then we search for another record *r2* in the auxiliary table by using the same primary key value and another *old schema table name*. If the record *r2* exists, we only insert the *r1* record in the auxiliary table. However, if the record *r2* does not exist, we also insert the record into the new schema table. Note that we do not insert any duplicate record in the new schema table and track the duplicity of records by using the auxiliary table.

Phase II: Insert- Search for the record in auxiliary table by using the primary key value and *old schema table name*. We ignore the log entry if the record already exists in the auxiliary table. Otherwise we insert the record in the new schema table as we do in phase I. *Delete-* Search for the record *r1* in the auxiliary table by using the primary key value and *old schema table name*. We ignore the log entry if no record is found. However, if the record *r1* exists, then search for the record *r2* in the auxiliary table by using the same primary key value and another *old schema table name*. If the record *r2* exists, we delete the record *r1* from the auxiliary table only; otherwise we also delete the record from the new schema table by using the primary key value. Note that we do not delete a record from the new schema table if it still exists in another old schema table. *Update-* Search for the record *r1* in auxiliary table by using the primary key value and *old schema table name*. If the record *r1* exists, then we search for the record in the new schema table by using the primary key column value and old record data from log entry. If such record exists in the new schema table, we update this record with the new record data from the log entry.

Horizontal Merge With Duplicates (HMWD): HMWD maintains all duplicate records in the new schema table while merging the records of both old schema tables. The new schema table consists of two additional columns – *myid* column as in the old schema table and *old schema table name* column. The combination of these two columns uniquely identify a record in the old schema table. Figure 4.2 demonstrates an example of HMWD. There are two old schema tables – *payment_2010* and *payment_2011*. The new schema table consists of all records from both old tables and maintains duplicate records with primary key column (*payment_id*) value as *P010123* and *P011090* in the new schema table. Below is the description of mapping records in both phases.



payment_2010

Payment_id	Customer_id	Amount
P010123	CUST0101	5000
P011090	CUST0110	15000
P010212	CUST0102	10000
P010345	CUST0103	7500

payment_2011

Payment_id	Customer_id	Amount
P010123	CUST0101	5000
P011090	CUST0110	15000
P011256	CUST0112	13000
P010697	CUST0106	2500

payment

Payment_id	Customer_id	Amount
P010123	CUST0101	5000
P011090	CUST0110	15000
P010212	CUST0102	10000
P010345	CUST0103	7500
P010123	CUST0101	5000
P011090	CUST0110	15000
P011256	CUST0112	13000
P010697	CUST0106	2500

Figure 4.2: HMWD: Horizontal Merge With Duplicates

Phase I: We scan all records of both old schema tables and insert each record into the new schema table along with corresponding *myid* and *old schema table name* column values.

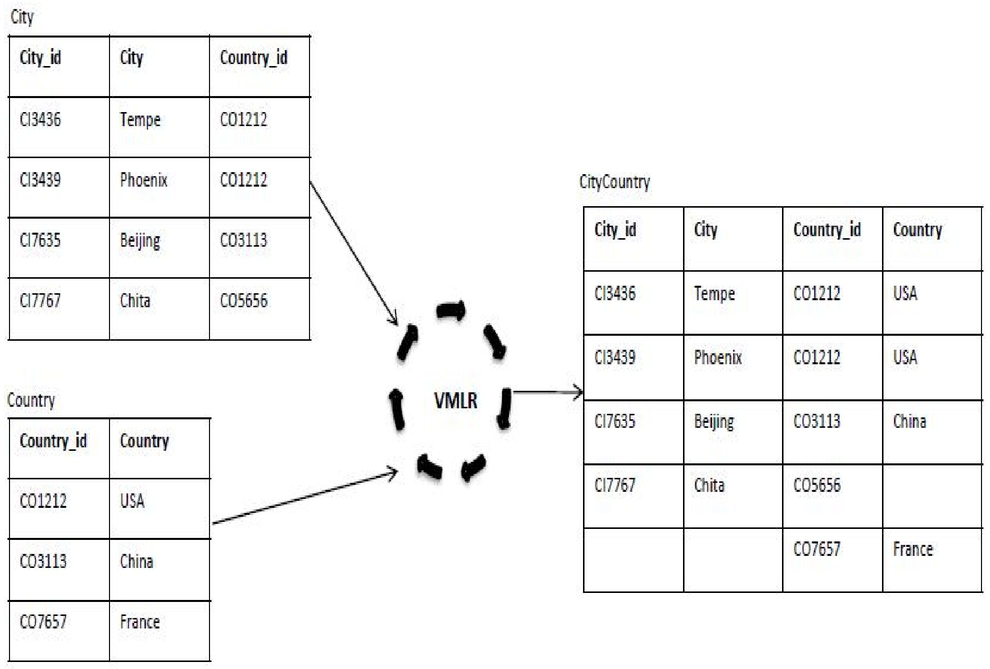
Phase II: Insert- Search for the record in the new schema table by using *myid* and *old schema table name* column values. If such record exists, we ignore the log entry; otherwise we insert the record into the new schema table as we do in phase I. *Delete*- Search for the record in the new schema table by using *myid* and *old schema table name* column values. If no such record exists, we ignore the log entry; otherwise we delete this record. *Update*- Search for the record in the new schema table by using *myid*, *old schema table name* column values and old record data from the log entry. If such record exists in the new schema table, we update the record with the new record data from the log entry.

4.2 Vertical Merge

Vertical Merge is equivalent to the *join* operator in relational database which merges *related records* from two tables into single record [13]. The most common *join* involves join conditions with equality comparisons. There are four types of join – Left outer join, Right outer join, Full outer join, and Inner join. Left outer join returns all combination of records in left and right table that are equal on their common attributes along with all the records in the left table for which there are no related records in the right table. Right outer join is similar to left outer join but the role of left and right tables are switched. Full outer join returns all combination of records in the left and right table that are equal on their common attributes along with the all records in the left table for which there are no related records in the right table and all records in the right table for which there are no related records in the left table. Inner join returns all combination of records in the left and right table that are equal on their common attributes. We implement full outer join since it is a lossless join. Left outer join, right outer join, and inner join can be derived from full outer join.

Vertical Merge Left Right (VMLR): VMLR merges two old schema tables, *Left* and *Right*, based on the value of a common attribute value. We refer to this common attribute as the merging column. Two records from the *Left* and *Right* tables are merged if the value of the common attribute is equal. VMLR mapping process requires an auxiliary table and four additional columns in the new schema table – *leftmyid* (value of *myid* in Left table), left table name, *rightmyid* (value of *myid* in Right table), and right table name. Figure 4.3 illustrates an example of

VMLR. We include all the records of *City* and *Country* tables in the new schema table *CityCountry*. Below is the description of mapping records in both phases.



City

City_id	City	Country_id
CI3436	Tempe	CO1212
CI3439	Phoenix	CO1212
CI7635	Beijing	CO3113
CI7767	Chita	CO5656

Country

Country_id	Country
CO1212	USA
CO3113	China
CO7657	France

CityCountry

City_id	City	Country_id	Country
CI3436	Tempe	CO1212	USA
CI3439	Phoenix	CO1212	USA
CI7635	Beijing	CO3113	China
CI7767	Chita	CO5656	
		CO7657	France

Figure 4.3: VMLR: Vertical Merge Left Right

Phase I: This phase starts with copying of the *Right* table to the auxiliary table. Then we scan the *Left* table and insert the merged record into the new schema table. This is done as follows. For each *Left* table record, search for the *Right* table record in the auxiliary table which have the same merging column value as that of *Left* table record. If there is no such *Right* table record, we insert the *Left* table record into the new schema table and leave the remaining columns blank as they belong to the *Right* table. However, if the corresponding *Right* table record exists, insert the merged record into the new schema table. When all the *Left* table records have been processed, scan auxiliary table for the *Right* table records which have not been copied to the new schema table. We insert all these remaining *Right* table records and leave the remaining columns blank as they belong to the *Left* table. Note that we maintain the lossless behavior of the vertical merge by keeping all the records of the old schema tables in the new schema table.

Phase II: There are two different mechanisms for mapping the records from *Left* and *Right* tables. *Left table insert-* Search the new schema table to determine if the record with the same *lmyid* and *left table name* column values already exists. We ignore the log entry if the record already exists in the new schema table. However, if no such record exists then we need to figure out the corresponding *Right* table record and then insert the merged record into the new schema table. To do so, we search for the record with same merging column value as that of the given *Left* table record and non blank *right table name* column value in the new schema table. If we find such *Right* table record, insert the merged record into the new schema table. However, if there is no corresponding *Right* table record, we insert the *Left* table record into the new schema table and leave the remaining columns blank as they belong to the *Right* table. Note that if the corresponding *Right* table record exists with blank *left table name* column value, we need to update this record itself to provide *Left* table column values instead of inserting a new merged record.

Left table delete- Search for the record in the new schema table by using given *lmyid* column value. We ignore the log entry if no such record exists. Otherwise, we need to delete this record. We need to handle two cases here. The record may or may not have the blank *right table name* column value. If the *right table name* column value is blank, then we just delete the record from the new schema table that has the given *lmyid* value. However, if the *right table name* column value is not blank, then we delete this records and also make sure that we do not loose any *Right* table record *r* in the new schema table. In order to do so, we search the *Right* table record *r* in the new schema table which has different value of *lmyid* column from that of the given *lmyid* value. If we find such *Right* table record, then we just delete the record from the new schema table that has the given *lmyid* value. Otherwise, we update this record in the new schema table that has the given *lmyid* value and make the *Left* table column values blank.

Left table update- It involves handling of two cases. If the merging column value is not changed, then we search for the record in the new schema table by using given *lmyid* column value which is also identical to the old record data of the log entry. Note that we only compare the *Left* table column values in the new schema table with the old record data to determine if they are identical. If we find such record, we just update the record in the new schema table by using the new record data from the log entry. However, if the merging column value is changed, then insert the new merged record in the new schema table in the similar way as we do in the *Left table insert* (described above). Also, we will need to delete the record from the new schema table which is identical to the old record data. This is done in the similar way as we do in the *Left table delete* (described above).

Right table insert- Search the new schema table to determine if the record with the same *rmyid* and *right table name* column values already exists. We ignore the log entry if the record already exists in the new schema table. However, if no such record exists then we need to figure out the corresponding *Left* table record and then insert the merged record into the new schema table. To do so, we search for the record with same merging column value as that of the given *Right* table record and non blank *left table name* column value in the new schema table. If we find such *Left* table record, insert the merged record into the new schema table. However, if there is no corresponding *Left* table record, we insert the *Right* table record into the new schema table and leave the remaining columns blank as they belong to the *Left* table. Note that if the corresponding *Left* table record exists with blank *right table name* column value, we need to update this record itself to provide *Right* table column values instead of inserting a new merged record.

Right table delete- Search for the record in the new schema table by using given *rmyid* column value. We ignore the log entry if no such record exists. Otherwise, we need to delete this record. We need to handle two cases here. The record may or may not have the blank *left table name* column value. If the *left table name* column value is blank, then we just delete the record from the new schema table that has the given *rmyid* value. However, if the *left table name* column value is not blank, then we delete this records and also make sure that we do not loose any *Left* table record *r* in the new schema table. In order to do so, we search the *Left* table record *r* in the new schema table which has different value of *rmyid* column from that of the given *rmyid* value. If we find such *Left* table record, then we just delete the record from the new schema table that has the given *rmyid* value. Otherwise, we update this record in the new schema table that has the given *rmyid* value and make the *Right* table column values blank.

Right table update- It involves handling of two cases. If the merging column value is not changed, then we search for the record in the new schema table by using given *rmyid* column value which is also identical to the old record data of the log entry. Note that we only compare the *Right* table column values in the new schema table with the old record data to determine if they are identical. If we find such record, we just update the record in the new schema table by using the new record data from the log entry. However, if the merging column value is changed, then insert the new merged record in the new schema table in the similar way as we do in the *Right table insert* (described above). Also, we will need to delete the record from the new schema table which is identical to the old record data. This is done in the similar way as we do in the *Right table delete* (described above).

4.3 Horizontal Split

Horizontal split distributes the records of an old schema table into two new schema tables depending on the value of a specific column which is called as horizontal split column. We assume that the two new schema tables are disjoint and their union is equal to the old schema table. This assumption is slightly different from that of Løland [20] where the resulting tables may have overlapping records. This thesis includes the horizontal split based on the equality condition (=) only, however, the schema transformations supporting other conditions (for example, less than <) can easily be integrated with the framework.

Horizontal Split on EQuality (HSEQ): HSEQ scans each record of the old schema table and inserts it into one of the new schema tables, table1 or table2, by checking the horizontal column value. Figure 4.4 illustrates an example of HSEQ. We scan each record of the old schema table *Category* and inserts it into either *Category1* or *Category2* table according to the horizontal split column value. Each new schema table has an additional *myid* column along with the indexing on it for the faster search during the mapping process. Below is the description of mapping records in both phases.

Phase I: We scan all the records of the old schema table. For each record, insert it into table1 or table2 according to the horizontal column value.

Phase II: Insert- Search for the record by using *myid* column value in one of the new schema tables depending on the horizontal split column value. The record is inserted if it doesn't exist already. *Delete-* Search for the record by using *myid* column value in one of the new schema tables depending on the horizontal split column value. The record is deleted if it exists already. *Update-* The update record case can be divided into two parts. If the horizontal split column value is not changed, then search for the record by using *myid* column value in one of the

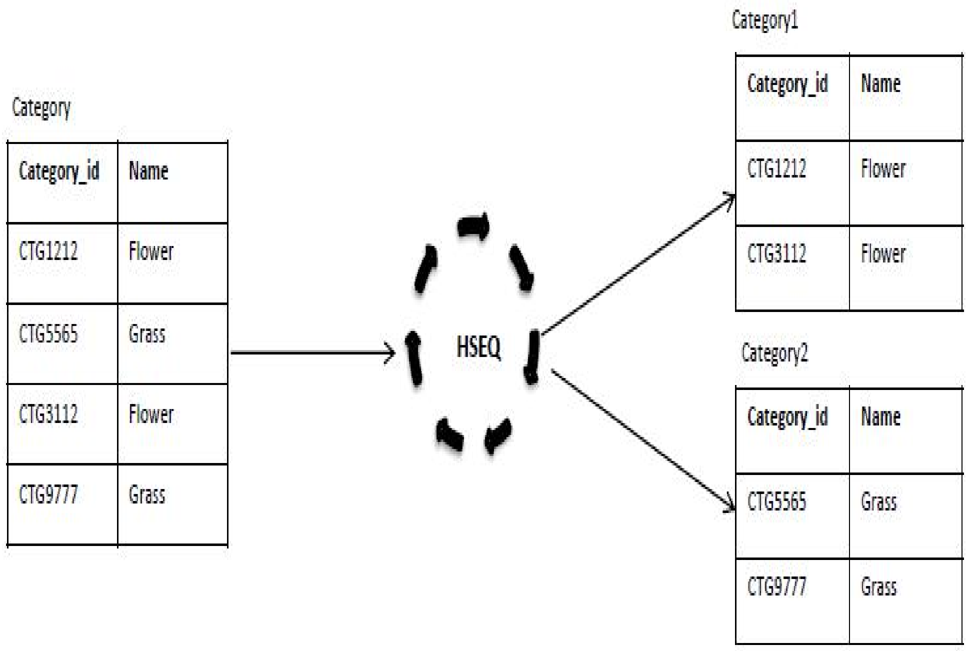


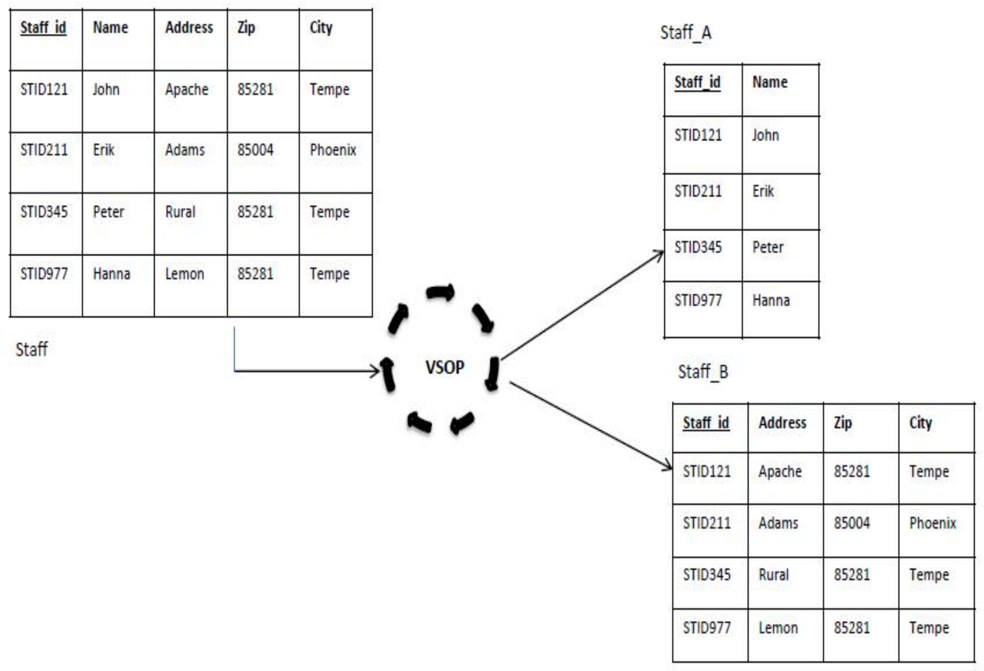
Figure 4.4: HSEQ: Horizontal Split on EQuality

new schema tables depending on the horizontal split column value. If such record exists and is also identical to the old record data of the log entry, we update this record. However, if the horizontal split column value is changed, then search for the old record data in table1 (or table2) and move the record to another new schema table with the new data record values of the log entry.

4.4 Vertical Split

Vertical Split divides a table vertically and produces two new schema tables. Each new schema table consists of a different subset of columns from the old schema table except one column (split column) which is shared by the two new schema tables. We describe two cases of vertical split. One divides the table based on a primary key, whereas another divides the table based on a non primary key which is often used in normalization (functional dependency) [13].

Vertical Split On Primary key (VSOP): This schema transformation involves vertical splitting of a table on the primary key column and this primary column is shared by both new schema tables, *split1* and *split2*. Figure 4.5 demonstrates an example of VSOP. The table *Staff* is divided into two new schema tables, *Staff_A* and *Staff_B*, and primary key column exists in both new schema tables. Below is the description of mapping records in both phases.



Staff_id	Name	Address	Zip	City
STID121	John	Apache	85281	Tempe
STID211	Erik	Adams	85004	Phoenix
STID345	Peter	Rural	85281	Tempe
STID977	Hanna	Lemon	85281	Tempe

Staff_id	Name
STID121	John
STID211	Erik
STID345	Peter
STID977	Hanna

Staff_id	Address	Zip	City
STID121	Apache	85281	Tempe
STID211	Adams	85004	Phoenix
STID345	Rural	85281	Tempe
STID977	Lemon	85281	Tempe

Figure 4.5: VSOP: Vertical Split On Primary key

Phase I: The columns in each new schema table represents the subset of the columns of the old schema table. Therefore, the combination of the records *rs1* and *rs2* represents the record *r* of the old schema table, where *rs1* represents the *split1* table record and *rs2* represents the *split2* table record. In this phase, we scan all the records of the old schema table. For each record, insert the records *rs1* and *rs2* into the new schema tables by using the record *r* of the old schema table.

Phase II: Insert- Search for the records *rs1* and *rs2* in new schema tables by using the primary column value. If records don't exist, we insert the records in the new schema tables as we do in phase I. *Delete-* Search for the records *rs1* and *rs2* in new schema tables by using the primary column value. The records are deleted if they exist already. *Update-* Search for the records *rs1* and *rs2* in new schema tables by using the primary column value and old record data from the log entry. If the search is a success, we update the records.

Vertical Split on Non Primary key (VSNP):

For the old schema table, we produce two new schema tables – *split1* and *split2*. The old schema table has primary key constraint on one of its columns. Each new schema table contains subset of the old schema table columns. In addition, *split1* table includes the primary key column as in the old schema along with other columns whereas the *split2* table makes the split column as the primary key column. Figure 4.6 illustrates the example of VSNP. Here *Staff_A* table is *split1* and *Staff_B* table is *split2*. *Staff* has the primary key column *Staff_id*, *Staff_A* has the primary key column *Staff_id* and *Staff_B* has the primary key column *Zip*. Below is the description of mapping records in both phases.

Phase I: The columns in each new schema table represents the subset of the columns of the old schema table. Therefore, the combination of the records *rs1* and *rs2* represents the record *r* of the old schema table, where *rs1* represents the *split1* table record and *rs2* represents the *split2* table record. In this phase, we scan all the records of the old schema table. For each record, insert the records *rs1* and *rs2* into the new schema tables by using the record *r* of the old schema table. However, we insert the record *rs2* into the *split2* table only if it does not violate the primary key constraint on the split column.

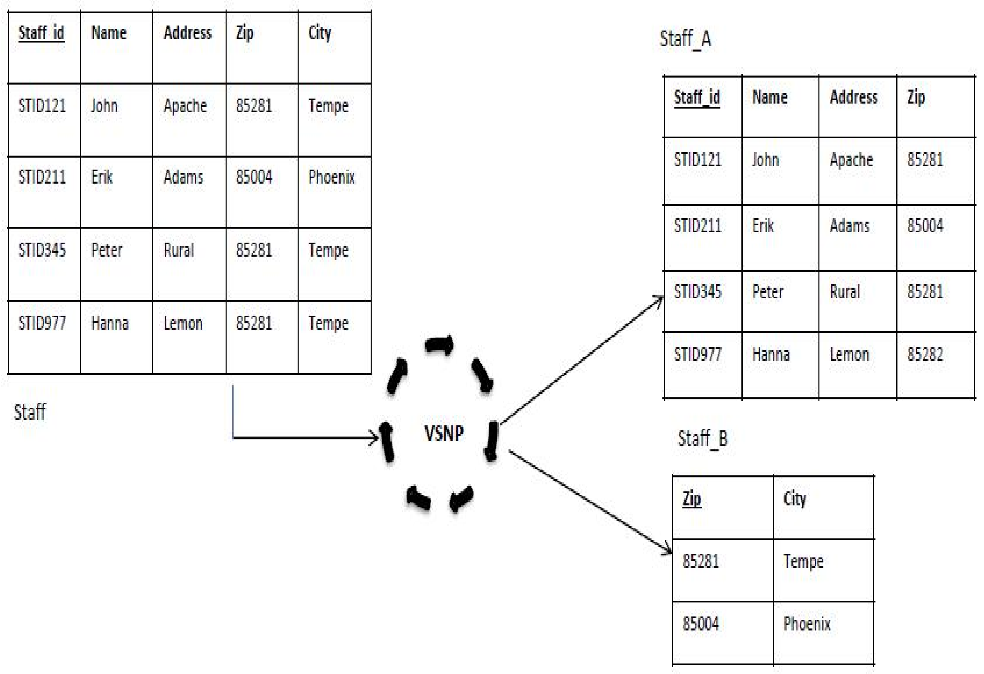


Figure 4.6: VSNP: Vertical Split on Non Primary key

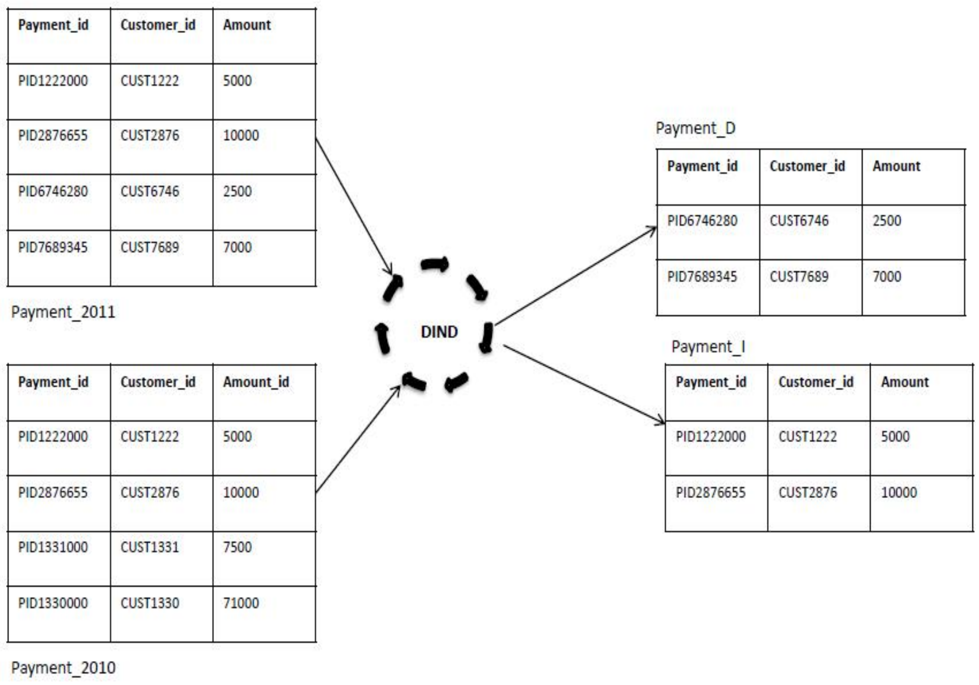
Phase II: Insert- Search for the record *rs1* in the *split1* table by using the primary key column value (or *myid* can be be used if no primary or unique column is provided). If the record is not found, insert the record in new schema tables in similar way as we do in phase I; otherwise, we ignore the log entry. *Delete*- Search for the record *rs1* in the *split1* table by using the primary key column value. If the record *rs1* exists, we delete it from the *split1* table. Then we search for the record *rs2* in *split2* table by using the split column value. We delete this record from *split2* table only if no other record in *split1* table has the same split column value.

Update- We need to handle two cases here. If the split column value is not changed, then search for the record *rs1* in *split1* table by using the primary key column value and search for the record *rs2* in *split2* table by using the split column value. If the combination of records *rs1* and *rs2* is identical to the old record data of the log entry, we just update the records *rs1* and *rs2* in new schema tables with the new record data of the log entry. However, if the split column value is changed, then search for the record *rs1* in *split1* table by using the primary key column value and search for the record *rs2* in *split2* table by using the old split column value. If the combination of records *rs1* and *rs2* is identical to the old record data of the log entry, we only update the record *rs1* in *split1* table with the new record data of the log entry. Then we search for the record in *split2* table by using the new split column value. If it exists, we update the record with the new record data of the log entry. Otherwise, we insert a new record in *split2* table by using the new record data of the log entry. Also, we search for the record in *split1* table by using the old split column value. If no record is found, then delete the record from *split2* table by using the old split column value.

4.5 Difference and Intersection

The difference and intersection operators apply to tables that have the same structure. The difference of two tables *Left* and *Right* is a new table that contains records in *Left* that are not in *Right*. The intersection of *Left* and *Right* is a new table that contains records that are in both *Left* and *Right*. As in [20], we implemented both schema transformations together. A part of the process, we create an auxiliary table as a part into which *Right* table is copied. It prevents conflicts with live transactions on the *Right* table while looking up for the records in the *Right* table during the update process. Below is the description of mapping records in both phases.

Difference and Intersection with No Duplicates (DIND): This operator produces two new schema tables, *Difference* and *Intersection*. *Difference* table consists of all records of *Left* table that do not exist in *Right* table. *Intersection* table consists of all records that exist both in *Left* and *Right* tables. DIND does not allow duplicate records in both old schema tables. Two records are considered duplicates if their field values are identical. Figure 4.7 illustrates an example of DIND. In the old schema table *Payment_2011*, the records with *Payment_id* column value as *PID6746280* and *PID7689345* do not exist in *Payment_2010*. Therefore, these records are inserted to the new schema intersection table *Payment_I*, where as other records are inserted to the new schema difference table *Payment_D*. Below is the description of mapping records in both phases.



Payment_2011

Payment_id	Customer_id	Amount
PID1222000	CUST1222	5000
PID2876655	CUST2876	10000
PID6746280	CUST6746	2500
PID7689345	CUST7689	7000

Payment_2010

Payment_id	Customer_id	Amount_id
PID1222000	CUST1222	5000
PID2876655	CUST2876	10000
PID1331000	CUST1331	7500
PID1330000	CUST1330	71000

Payment_D

Payment_id	Customer_id	Amount
PID6746280	CUST6746	2500
PID7689345	CUST7689	7000

Payment_I

Payment_id	Customer_id	Amount
PID1222000	CUST1222	5000
PID2876655	CUST2876	10000

Figure 4.7: DIND: Difference and Intersection with No Duplicates

Phase I: This phase begins with copying the *Right* table to the auxiliary table. For each *Left* table record, if an identical record exists in the auxiliary table then insert the *Left* table record into the *Intersection* table. Otherwise, insert the *Left* table record into the *Difference* table.

Phase II: Similar to VMLR, DIND also has two different mechanisms for mapping records from both, *Left* and *Right*, old schema tables.

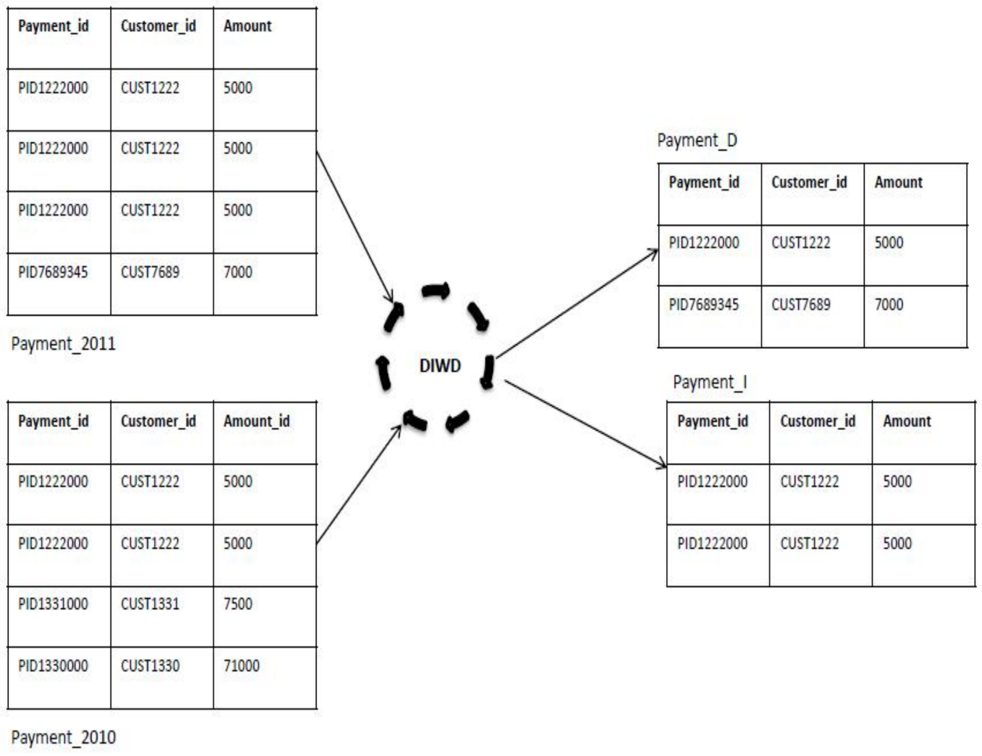
Left table insert- Search for the record in *Difference* and/or *Intersection* table by using the record data of the log entry. If such record already exists in one of the new schema tables, we ignore the log entry. Otherwise, insert the record into the new schema table in the similar way as we do in phase I. *Left table delete-* Search for the record in *Difference* and/or *Intersection* table by using the record data of the log entry. We delete the record if it exists in any of these two new schema tables. *Left table update-* Search for the record *r* in *Difference* and/or *Intersection* tables which is identical to the old record data of the log entry. If the record *r* exists in the *Difference* table, update the record *r* with the new record data of the log entry (say r_{new}). Then search for the record in the auxiliary table which is identical to the record r_{new}. If such record exists, move the r_{new} record from the *Difference* table to the *Intersection* table. However, if the record *r* exists in the *Intersection* table, update the record *r* with the new record data of the log entry (say r_{new}). Then search for the record in the auxiliary table which is identical to the record r_{new}. If such record does not exist, move the r_{new} record from the *Intersection* table to the *Difference* table.

Right table insert- Search for the record in the auxiliary table by using the record data of the log entry. We ignore the log entry if it exists already. Otherwise, we insert the record into the auxiliary table. Then search for an identical record in the *Difference* table. If such record exists, then we move it to the *Intersection* table. *Right table delete-* Search for the record in the auxiliary table by using the record data of the log entry. We ignore the log entry if it does not exist already. Otherwise, we delete the record from the auxiliary table. Then search for an identical record in the *Intersection* table. If such record exists, we move it to the *Difference* table. *Right table update-* Search for the record *r* in the auxiliary table which is identical to the old record data of the log entry. If the record *r* exists, update the record *r* with the new record data of the log entry (say r_{new}). Then search for the record in the *Difference* table which is identical to the record r_{new}. If such record exists, move it to the *Intersection* table. Also, search for the record in the *Intersection* table which is identical to the record *r*. If such record exists, move it to the *Difference* table.

Difference and Intersection With Duplicates (DIWD): DIWD is an additional schema transformation that is not covered in [20]. Unlike DIND, DIWD allows for duplicate records in the *Difference* and *Intersection* tables and, therefore, is somewhat more complex. The *Difference* table has an additional column *lmyid* that represents the *myid* column of the *Left* table. The *Intersection* table consists of two additional columns – *lmyid* that represents *myid* column of the *Left* table and the other is *rmyid* that represents the *myid* column of the *Right* table. We also create an auxiliary table into which *Right* table is copied during the process and it has *rmyid* column that represents the *myid* column of the *Right* table as an additional column.

Figure 4.8 illustrates an example of DIWD. We consider each duplicate record as a different record. In the old schema table *Payment_2011*, the records with *Payment_id* column value exist three times whereas the same record exists only twice in another old schema table *Payment_2010*. Therefore, this record is inserted into the new schema intersection table only twice. The remaining third record is inserted into the new schema difference table *Payment_D*. Below is the description of mapping records in both phases.

Phase I: This phase begins with copying of *Right* table to an auxiliary table. Then we scan all records of the *Left* table. For each *Left* table record, we search for an identical record *r* in the auxiliary table such that the record with same *rmyid* column value does not exist in *Intersection* table. If such record *r* exists, we



Payment_2011

Payment_id	Customer_id	Amount
PID1222000	CUST1222	5000
PID1222000	CUST1222	5000
PID1222000	CUST1222	5000
PID7689345	CUST7689	7000

Payment_2010

Payment_id	Customer_id	Amount_id
PID1222000	CUST1222	5000
PID1222000	CUST1222	5000
PID1331000	CUST1331	7500
PID1330000	CUST1330	71000

Payment_D

Payment_id	Customer_id	Amount
PID1222000	CUST1222	5000
PID7689345	CUST7689	7000

Payment_I

Payment_id	Customer_id	Amount
PID1222000	CUST1222	5000
PID1222000	CUST1222	5000

Figure 4.8: DIWD:Difference and Intersection With Duplicates

insert the *Left* table record into the *Intersection* table. Otherwise, insert this record into the *Difference* table.

Phase II: Left table insert- Search for the record in *Difference* and/or *Intersection* table by using the *lmyid* column value. If such record exists in any of these two new schema tables, we ignore the log entry. Otherwise insert the record either into the *Difference* or *Intersection* table in the similar way as we do in phase I.

Left table delete- Search for the record in *Difference* and/or *Intersection* table by using the *lmyid* column value. If such record exists in any of these new schema tables, we delete the record.

Left table update- Search for the record (say *r*) in the *Difference* and/or *Intersection* table by using the *lmyid* value and the old record data of the log entry.

If the record *r* exists in the *Difference* table, we update this record (say r_{new}) and then search for an identical record in the auxiliary table such that the record with same *rmyid* column value does not exists in *Intersection* table. If such record exists in the auxiliary table, we move the record r_{new} to the *Intersection* table. Moreover, if the record *r* exists in the *Intersection* table, we update this record (say $\mathrm{r}_{\mathrm{new}}$) and then search for an identical record in the auxiliary table such that the record with same *rmyid* column value does not exists in *Intersection* table. If such record exists in the auxiliary table, we update the *rmyid* column value of record r_{new} in the *Intersection* table. Otherwise, we move the record r_{new} to the *Difference* table.

Right table insert- Search for the record in the auxiliary table by using the *rmyid* column value. If the record already exists, we ignore the log entry. Otherwise, insert the record into the auxiliary table and search for an identical record in the *Difference* table. If we find the identical record in the *Difference* table, we move it to the *Intersection* table.

Right table delete- Search for the record in the auxiliary table by using the *rmyid* column value. If the record does not exist, we ignore the log entry. However, if the record exists, we delete the record from auxiliary table. Then we search for an identical record (say *r*) in the *Intersection* table by using the *rmyid* column value. If the record *r* exists in the *Intersection* table, then we search for another identical record in the auxiliary table such that the record with the same *rmyid* column value does not exists in the *Intersection* table. If such record exists, we update the *rmyid* column value of the record *r* in the *Intersection* table. Otherwise, we move the record *r* to the *Difference* table.

Right table update- Search for the record in the auxiliary table by using the *rmyid* value and the old record data of the log entry. If the record exists in the auxiliary table, we update this record and then search for an identical record in the

Intersection table by using the *rmyid* value and the old record data of the log entry. If such record (say *r*) exists in the *Intersection* table, we update the record (say r_{new}) and then search for an identical record in the auxiliary table such that the record with same *rmyid* column value does not exists in *Intersection* table. If such record exists in the auxiliary table, we update the *rmyid* column value of the record r_{new} in the *Intersection* table. Otherwise, we move the record r_{new} to the *Difference* table. Also, we search for another identical record for the old record data of the log entry (*r*) in the auxiliary table. If we find the record in the auxiliary table, the record *r* is inserted into the *Intersection* table. Otherwise, insert it into the *Difference* table.

Chapter 5

RESULTS

This chapter describes the test environment and experiments performed in order to evaluate the update framework. We show the impact of varying certain parameters on the overhead caused by the update process. It also shows the feasibility of the database updates with less overhead.

We performed dynamic database updates on the sample schema obtained from PgFoundry [14] including the following schema transformations: Horizontal Merge with No Duplicates, Horizontal Split, Vertical Merge, Vertical Split with Primary Key column, Vertical Split with No Primary Key column, Difference and Intersection with No Duplicates, Difference and Intersection With Duplicates. Any index mentioned in this thesis, which exists in the old schema or created as a part of the update process, represents a b-tree based index.

5.1 Old and New Tables for Schema Transformations

Here we describe the old and new scheme tables involved in each schema transformation.The complete details of old and new schemas are provided in the Appendix A for all schema transformations.

1. HMND: Old schema tables are *payment_p2007_04* and *payment_p2007_05*. New schema table is *tabledn*.

2. VMLR: Old schema tables are *City* and *Country*. New schema table is *tabledvm*.

3. VSOP: Old schema table is *customer* and new schema tables are *tabledp1* and *tabledp2*.

4. VSNP: Old schema table is *staff* and new schema tables are *tabledn1* and *tabledn2*.

5. HSEQ: Old schema table is *category* and new schema tables are *tablhs1* and *tablehs2*.

6. DIND: Old schema tables are *payment* and *payment_p2007_01*. New schema tables are *tabledifference* and *tableintersection*.

7. DIWD: Old schema tables are *payment_p2007_02* and *payment_p2007_03*. New schema tables are *tabledifferenceall* and *tableintersectionall*.

5.2 Experimentation Setup

We used two computers to simulate client-server architecture for the database management system. Both computers are connected through a coaxial cable to eliminate noise due to network latency fluctuation. One computer acts as the client and executes transactions on the database server running on the other computer. The client machine is a 1.8 GHz x86 CPU with 512 Mb of main memory running Debian GNU/Linux 4.0 and Java Development Kit (JDK) version: 1.6. The client machine uses a Java program to execute transactions on the database server continuously. The DBMS used is PostgreSQL 8.4 with 24 Mb of shared buffer size executing on a dual core 3.33 GHz x86 machine with 8 Gb main memory and 128 Mb effective cache size running Ubuntu 9.10. The shared buffer size is less than 10% of the size of the database and while the available memory is very large, the buffer size limits all in-memory database operations to 24 Mb to avoid a situation in which the database queries are running from main memory (and only writes to the log go to disk). We opted for the smaller size database and buffer size to be able to test our implementation within a reasonable amount of time while attempting to achieve system parameters that are a scaled down version of what one would find in larger databases. The database on which we tested the transformation consists of 11 tables with an average of around 5 columns per table.

The tables were populated with synthetic data of 300,000 records for a total of 300 Mbyte (approximately) of space (average of around 100 bytes per record).

5.3 Query Generation

In general, transactions can be complex in which case all the transactions might have multiple queries of the same or different basic types (update, delete, insert, select) or simple in which case a transaction has only one query. In the evaluation, we mainly considered simple queries. Simple client queries were generated randomly according to uniform distributions both across tables and across records. However, *insert* type of queries only insert new records in a table. As we mentioned earlier that there are 11 old schema tables, we generate the queries which randomly choose a table from old schema tables and then pick a random record from the chosen table. Each table has at least one integer column which is used as the condition to pick a record by a query. All records have values for this integer column in a pre-defined range. This range of numbers is used to generate the random client queries by using the random function provided by Java. Client queries are grouped according to the type of the transaction. Therefore, we have four set of transactions. These are – Update, Delete, Insert and Select. Each transaction set consists of one type of simple queries that inserts/update/deletes/selects a record from a table based on a given condition. For example, *Update Transaction Set* consists of all the update queries on the old schema tables. The samples of these transactions sets are provided in Appendix (B.1, B.2, B.3). This has the advantage of comparing how the performance parameters are affected by query types. It has the obvious disadvantage of limiting the scope of the evaluation as the real-world scenario may consist of transactions that include a group of mixed queries on multiple tables. For each basic transaction type, we generated a transaction set for the old schema and applied it against the database. In addition, we also generated two transaction sets of complex client transactions in which each transaction consists of multiple queries (update and delete) including multiple old schema tables. The samples of these complex transactions are provided in Appendix (B.4, B.5).

5.4 Empirical Validation

The dynamic database update process was executed on the database while client queries were executing on the old schema. For testing the correctness of the implementation, the content of the old database and new database is compared when the system is ready to switch from the old to the new database. We ran a *validation procedure* to ensure that the data in the new database matches the data in the old database for some experiments. We also performed this process manually for some experiments. The empirical validation is mainly done during the development phase and partially during the the performance analysis phase.

During the dynamic database update process, synchronization of the log file was achieved in as little as 2 and as high as 28 iterations over the log files. We describe the performance parameters and interpretation of the obtained results next.

5.5 Performance Parameters

We measure three performance parameters: (1) percentage overhead in the average response time per transaction; (2) execution time taken by the update process (time during which the response time overhead is incurred); and (3) *upper response time* which is the smallest time larger than the response time of 95% of all completed transactions [18]. While the first parameters are straightforward to understand, the last parameter attempts to capture the worst-case scenario for response time. The overhead is calculated as:

$$overhead = \frac{ART_{WU} - ART_{WOU}}{ART_{WOU}} \times 100 \tag{5.1}$$

where ART_{WU} is the average response time of user transactions with the update process and ART_{WOU} is the average response time of user transactions without the update process.

As mentioned in Chapter 3, the performance is measured as a function of two optimization parameters: (1) the batch size used in copying the old database to the new database; and (2) the inter-batch time interval. The effect of these parameters were evaluated by varying the parameters individually, while keeping the other constant.

The concurrent user transactions have continuous access patterns in which new queries are issued as soon as responses to the previous queries are received. For our system, the continuous access pattern of user transactions corresponds to a query frequency of approximately 60 transactions per second which is the highest feasible for the client and we have opted not to try to increase that rate by multi-threading the client.

5.6 Interpretation of Results

We provide performance analysis for all four types of transaction sets – update, delete, insert and select.

UPDATE: As we mentioned in the previous chapter, there are two optimization parameters: batch size and inter-batch time interval. The smaller value of batch size results in a read-lock on a smaller number of records in the old schema table during the update process. This implies that there will be less overhead of the update process on the user transactions for the smaller values of batch size. However, smaller values of the batch size result in longer execution time of the update process due to large number of batches. The inter-batch time interval is the time during which the update process is halted between the processing of each batch. The larger value of time interval provides more scope for the user transactions to execute and results in less overhead, but results in a larger total execution time of the update process.

We draw the performance graphs for percentage overhead, execution time of the update process and upper response time. Figures 5.1, 5.2 and 5.3 show the results for different time intervals when batch size is varied. The overhead tends to decrease with smaller value of batch size (Figure 5.1) whereas the execution time increases (Figure 5.2). Due to the smaller overhead, the upper response time also decreases for small value of batch size (Figure 5.3).

Figures 5.4, 5.5 and 5.6 present the results for different batch sizes when time interval is varied. The overhead tends to decreases with larger value of time interval (Figure 5.4) whereas the execution time decreases (Figure 5.5). Due to the smaller overhead, the upper response time also decreases for large value of time interval (Figure 5.6).

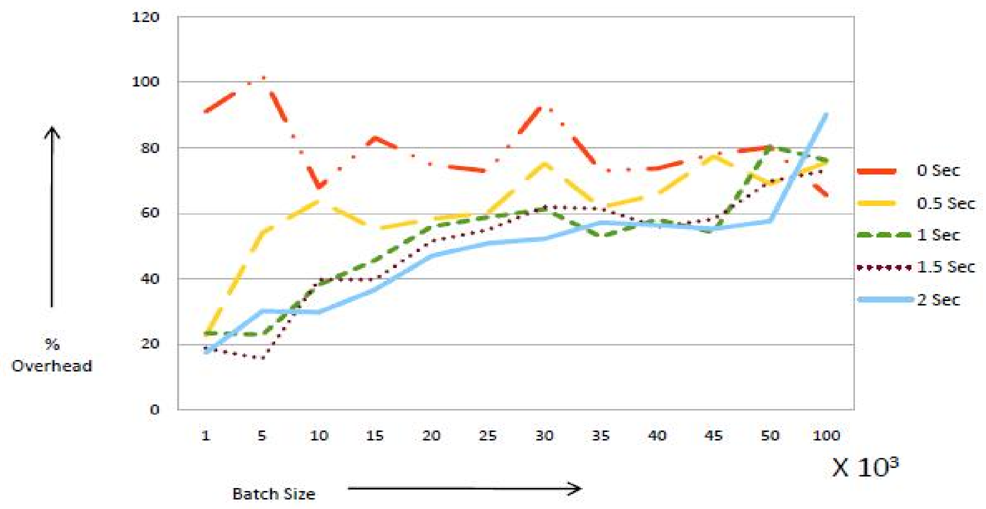


Figure 5.1: Overhead for Update: Vary batch size

DELETE and INSERT: Performance graphs for Delete and Insert transaction show patterns that are similar to those of the Update transactions. The performance graphs for Delete with different *batch sizes* are given in

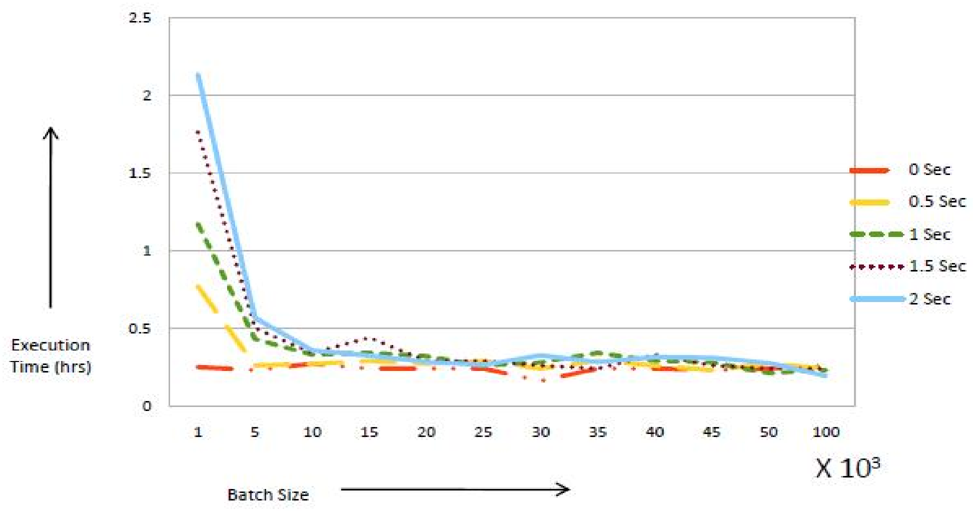


Figure 5.2: Execution Time for Update: Vary batch size

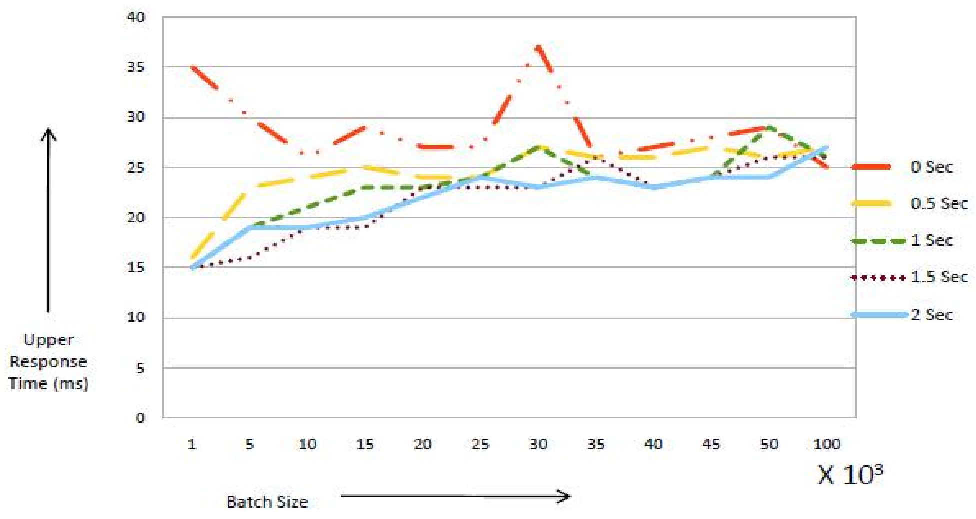


Figure 5.3: Response Time for Update: Vary batch size

Figures 5.7, 5.8, 5.9. The performance graphs for Delete with different *inter-batch time intervals* are given in Figures 5.10, 5.11, 5.12.

The performance graphs for Insert with different *batch sizes* are given in Figures 5.13, 5.14, 5.15. The performance graphs for Insert with different *inter-batch time intervals* are given in Figures 5.16, 5.17, 5.18.

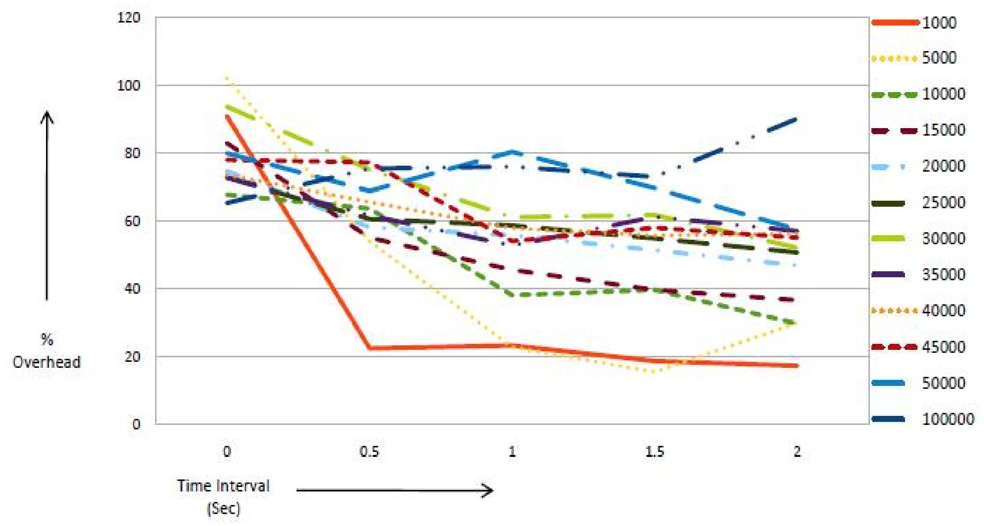


Figure 5.4: Overhead for Update: Vary time interval

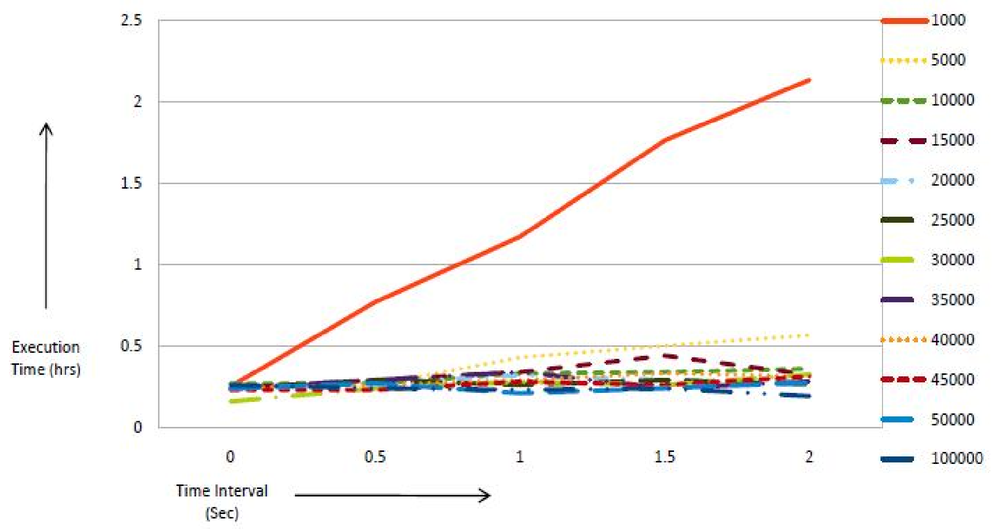


Figure 5.5: Execution Time for Update: Vary time interval

SELECT: The database update process holds read-lock on the old schema table and hence, does not interfere with the Select transactions. There will be an overhead due to sharing of common resources though.

We also perform a set of experiments using complex transaction sets and measure the overhead. We generate two complex transaction sets A and B

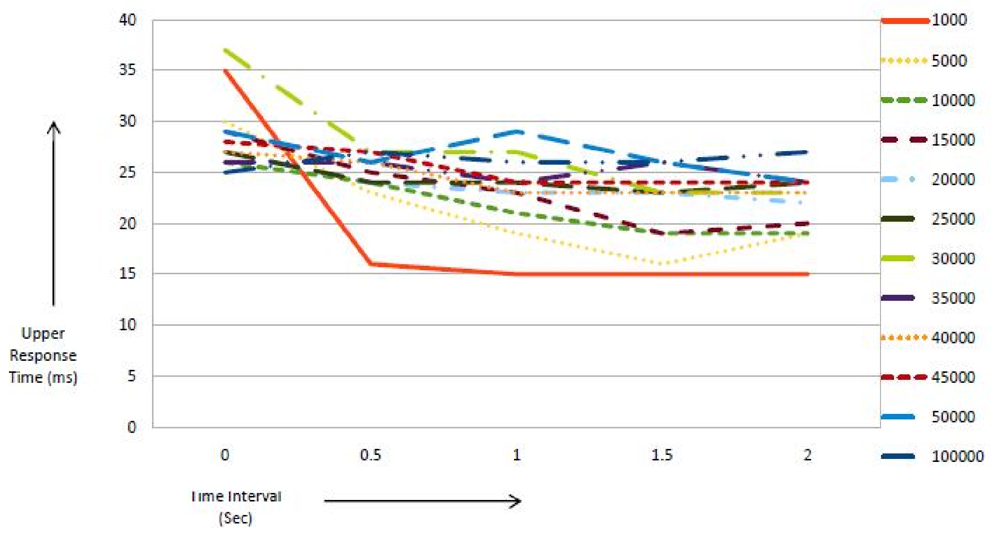


Figure 5.6: Response Time for Update: Vary time interval

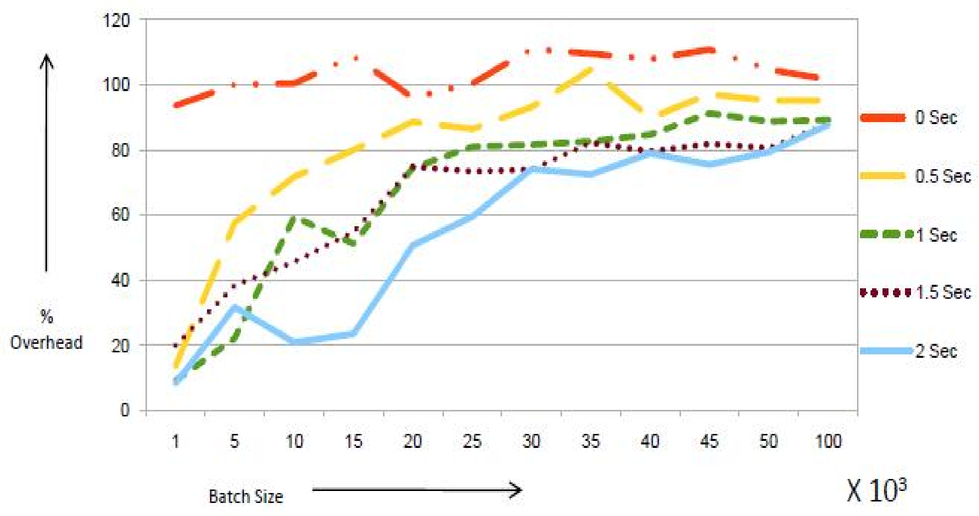


Figure 5.7: Overhead for Delete: Vary batch size

(Appendix B.4 and B.5) and study the overhead by varying batch size. In these complex transactions, each transaction involves more than one table to perform any changes in the database.

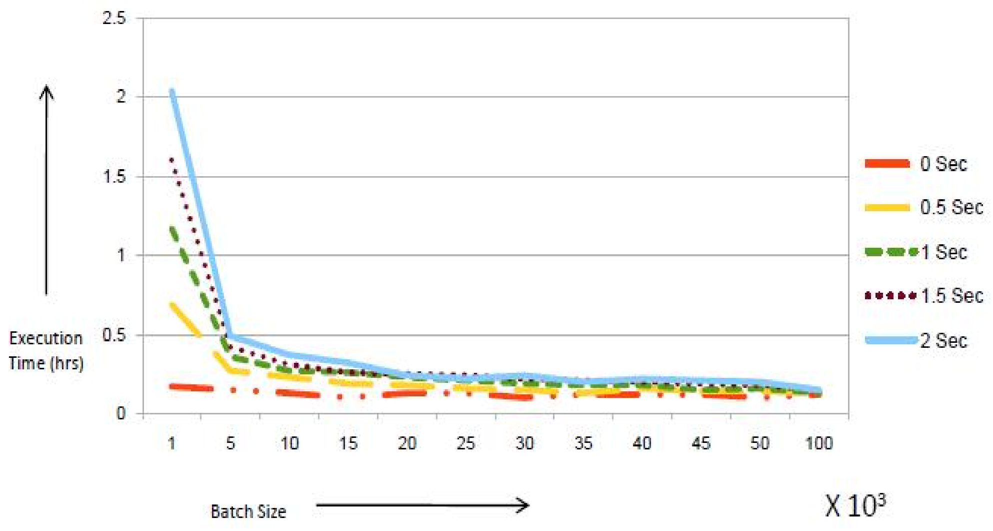


Figure 5.8: Execution Time for Delete: Vary batch size

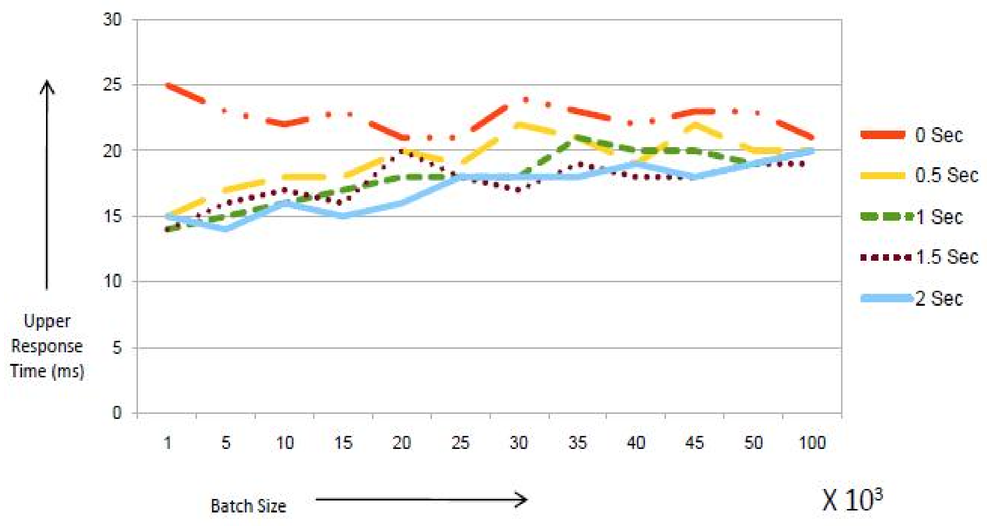


Figure 5.9: Response Time for Delete: Vary batch size

Complex Transaction Set A: For batch size = 5000 and inter-batch time interval = 1 second, the percentage overhead is 77.19. And, for batch size = 1000 and inter-batch time interval = 1 second, the percentage overhead is 35.6.

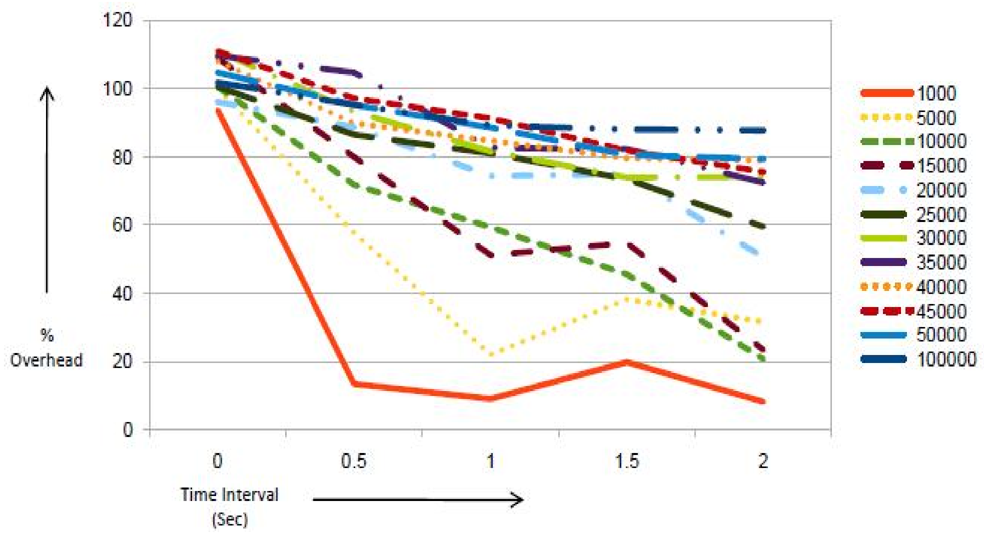


Figure 5.10: Overhead for Delete: Vary time interval

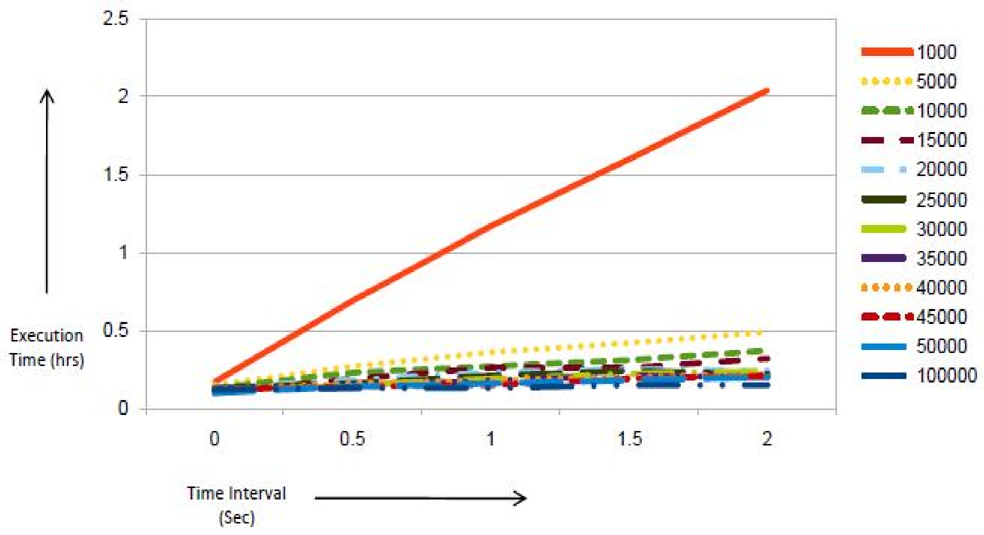


Figure 5.11: Execution Time for Delete: Vary time interval

Complex Transaction Set B: For batch size = 5000 and inter-batch time interval = 1 second, the percentage overhead is 40.76. And, for batch size = 1000 and inter-batch time interval = 1 second, the percentage overhead is 30.

Similar to the case of simple queries, the overhead tends to reduce with the smaller batch size for the complex transactions as well.

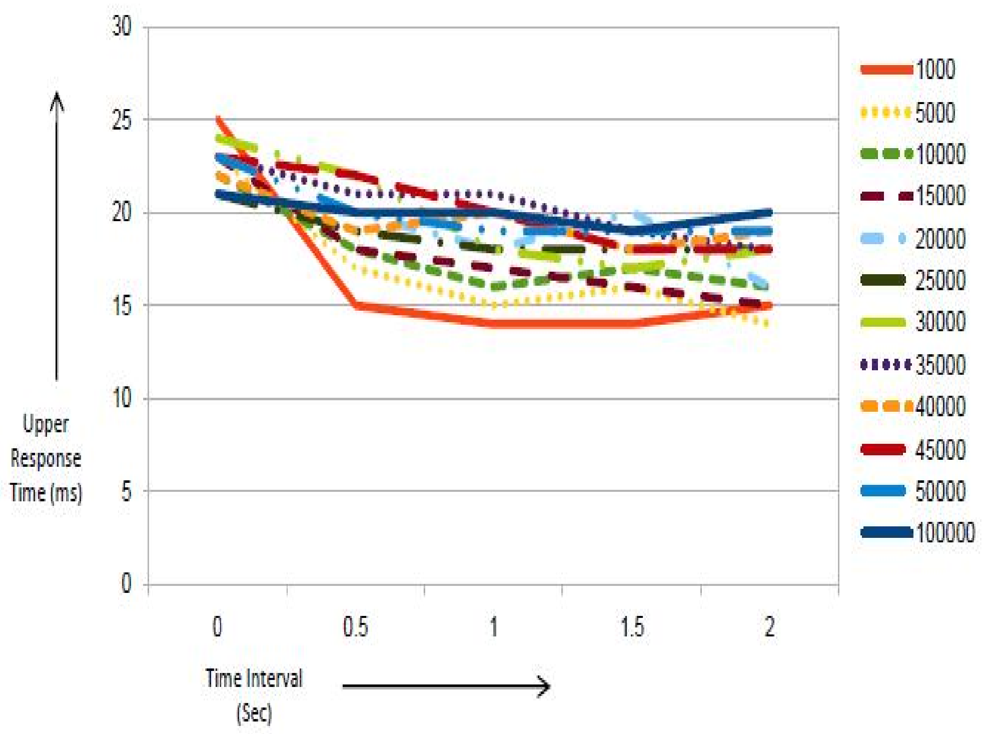


Figure 5.12: Response Time for Delete: Vary time interval

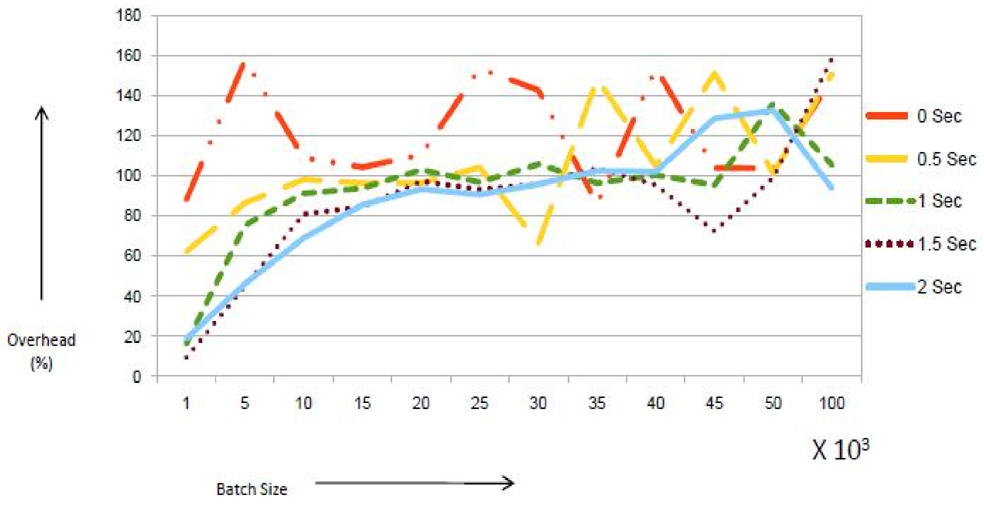


Figure 5.13: Overhead for Insert: Vary batch size

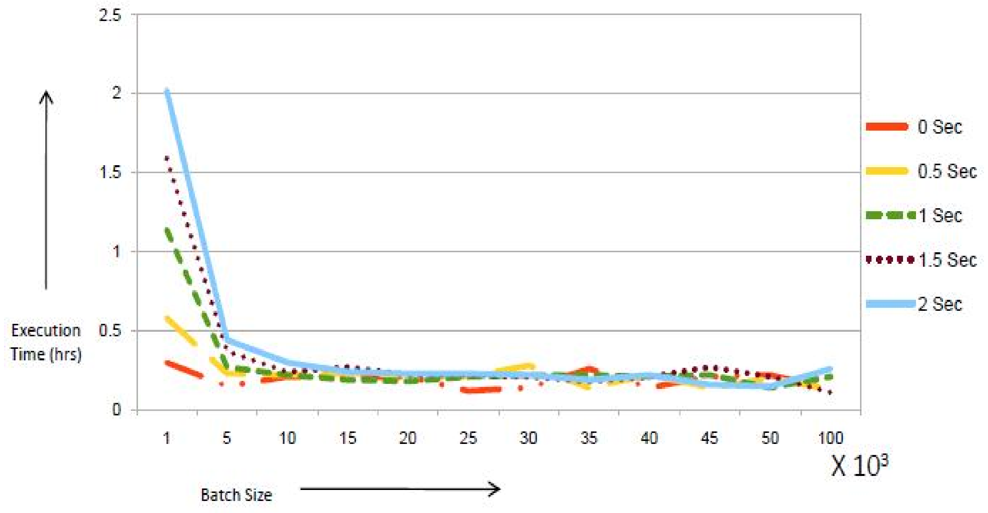


Figure 5.14: Execution Time for Insert: Vary batch size

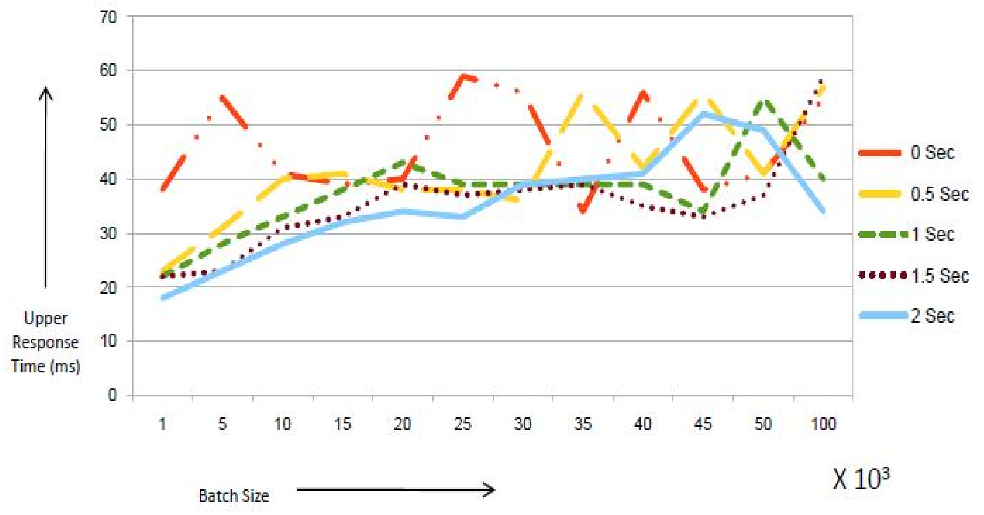


Figure 5.15: Response Time for Insert: Vary batch size

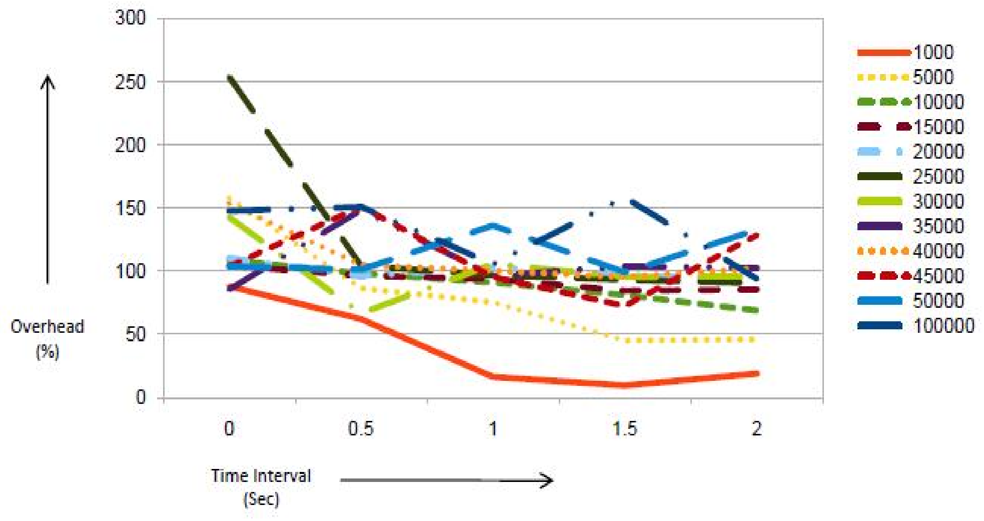


Figure 5.16: Overhead for Insert: Vary time interval

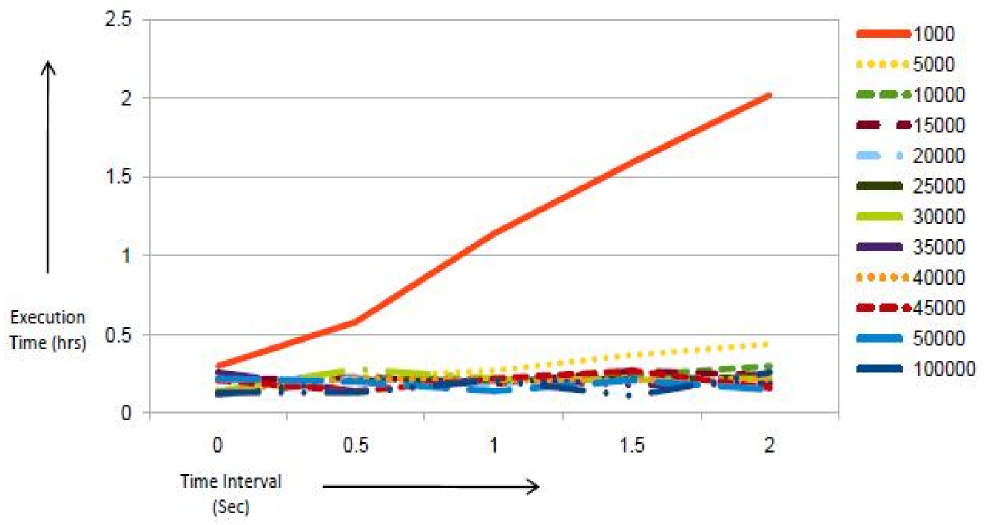


Figure 5.17: Execution Time for Insert: Vary time interval

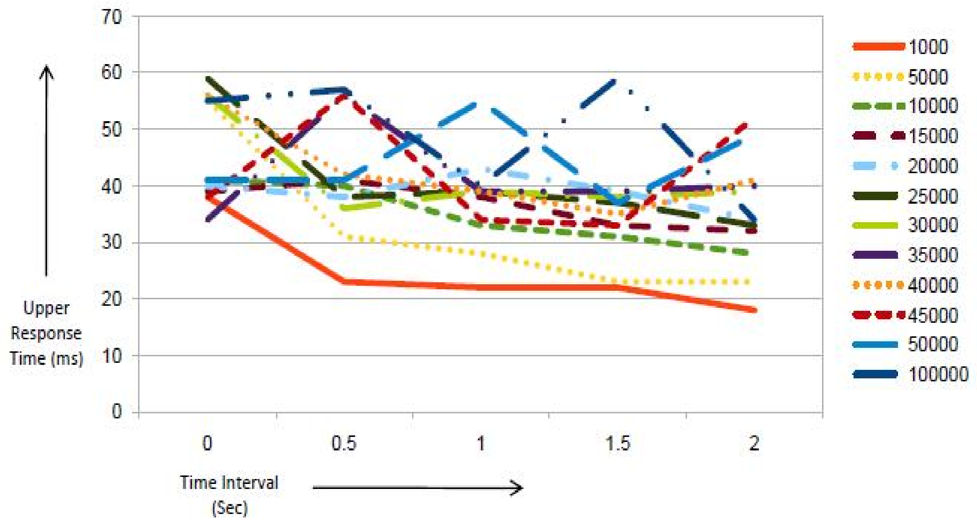


Figure 5.18: Response Time for Insert: Vary time interval

Chapter 6

CONCLUSION

We have presented a client-driven framework for dynamic database updates and have shown that it is a viable approach for implementing complex dynamic database updates. The client-driven mechanism of the framework provides the flexibility of its integration with already existing DBMSs. The update framework supports the schema transformations presented in the Løland [20] along with an additional schema transformation. The framework reads the specifications of old and new schemas using configuration files and generates the required mappings for these schema transformations. However, any other schema transformation can be integrated with the framework. To integrate a schema transformation, user needs to define certain mapping mechanisms required at various stages of the update process. User can also integrate the functionality of automatic generation of new schema transformations. Most of the implementation is done in structured query language which is a declarative language used by a wide range of relational databases. Therefore, it becomes easier for a developer to integrate the support for other schema transformations in the framework.

We test our framework with a number of user transactions (simple and complex) and study the impact of the performance parameters defined in the framework on the concurrent user transactions during the update process. We show how we can reduce the overhead of an update process by varying these performance parameters. The performance results, while not extensive, show the viability of the work.

6.1 Limitations and Future Work

There are a number of directions for further study:

1. The current implementation of the framework does not allow the coexistence of the old and new schemas. The framework can be extended to provide this functionality with the help of triggers and more complex implementation. However, this approach may cause a significant overhead.

2. This thesis focuses on the updates in relational schema only. However, the database updates may lead to the changes in the client applications. The dynamic updates of client applications along with the database updates is subject for future research.

3. There is a scope of optimization for better log processing. The current implementation scans the log sequentially to process each log entry. The log reconciliation becomes costly if each log entry fetches a new disk page to apply the changes to the new database table. Therefore, we need an efficient method for processing the log. One approach for efficient log processing is to rearrange the log such that it reduces the number of page faults and hence requires fewer disk accesses.

4. The performance results that we presented show the viability of our approach, but a more thorough evaluation is still needed with more realistic queries and database.

REFERENCES

[1] D. Adams. Oracle xml db developer's guide, 11g release 1 (11.1) b28369-01.

[2] H. Bounif. Predictive approach for database schema evolution. *Intelligent databases: technologies and applications*, page 286, 2007.

[3] S.E. Bratsberg and R. Humborstad. Online scaling in a highly available database. In *VLDB*, pages 451–460, 2001.

[4] M. Callaghan. Online schema change for mysql. Web., 2010. <http://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932>.

[5] R. Chirkova and G.H.L. Fletcher. Towards well-behaved schema evolution. Citeseer, 2009.

[6] K.T. Claypool, J. Jin, and E.A. Rundensteiner. Serf: schema evolution through an extensible, re-usable and flexible framework. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 314–321. ACM, 1998.

[7] A. Cleve, A.F. Brogneaux, and J.L. Hainaut. A conceptual approach to database applications evolution. *Conceptual Modeling–ER 2010*, pages 132–145, 2010.

[8] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[9] C. Curino, H.J. Moon, and C. Zaniolo. Automating database schema evolution in information system upgrades. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, page 5. ACM, 2009.

[10] C.A. Curino, H.J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772, 2008.

[11] IBM developerWorks. Db2: Online schema change. Web., 2009. <http://www.ibm.com/developerworks/data/library/techarticle/dm-0907db2outages/>.

[12] E. Domínguez, J. Lloret, Á.L. Rubio, and M.A. Zapata. Medea: A database evolution architecture with traceability. *Data & Knowledge Engineering*, 65(3):419–441, 2008.

[13] R. Elmasri and S. Navathe. *Fundamentals of database systems*, volume 2. Pearson Education India, 2008.

[14] PostgreSQL Development Group. pgfoundry. n.d. <http://pgfoundry.org/>.

[15] M. Hartung, J. Terwilliger, and E. Rahm. Recent advances in schema and ontology evolution. *Schema Matching and Mapping*, pages 149–190, 2011.

[16] J.M. Hick and J.L. Hainaut. Database application evolution: a transformational approach. *Data & Knowledge Engineering*, 59(3):534–558, 2006.

[17] S.O. Hvasshovd, T. Sæter, Ø. Torbjørnsen, P. Moe, and O. Risnes. A continously available and highly scalable transaction server: Design experience from the hypra project. In *IN PROCEEDINGS OF THE 4TH INTERNATIONAL WORKSHOP ON HIGH PERFORMANCE TRANSACTION SYSTEMS*. Citeseer, 1991.

[18] S.O. Hvasshovd, Ø. Torbjørnsen, S.E. Bratsberg, and P. Holager. The clustra telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477. Morgan Kaufmann Publishers Inc., 1995.

[19] IBM. Optim database administrator for db2. Web., n.d. <http://www-01.ibm.com/software/data/optim/database-administrator/features.html>.

[20] J. Løland. *Materialized view creation and transformation of schemas in highly available database systems*. PhD thesis, Norwegian University of Science and Technology (NTNU), Norway, 2007.

[21] J. Løland and S.O. Hvasshovd. Online, non-blocking relational schema changes. *Advances in Database Technology-EDBT 2006*, pages 405–422, 2006.

[22] S. Monk and I. Sommerville. Schema evolution in oodbs using class versioning. *ACM SIGMOD Record*, 22(3):16–22, 1993.

[23] Craig Mullins. *Database administration: the complete guide to practices and procedures*. Addison-Wesley, 2002.

[24] MySQL. Alter table syntax. Web., n.d. <http://dev.mysql.com/doc/refman/5.1/en/alter-table.html>.

[25] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. Hecataeus: Regulating schema evolution. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1181–1184. IEEE, 2010.

[26] Jan Paredaens, Paul De Bra, Marc Gyssens, and Dirk Van Gucht. *The structure of the relational database model*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[27] M. Ronström. On-line schema update for a telecom database. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 329–338. IEEE, 2000.

[28] Rever S.A. Db-main: The modelling framework. Web., n.d. <http://www.db-main.eu/?q=en/content/db-main-data-architecture tool/>.

[29] SQL Server. Using sql server management studio. Web., n.d. <http://msdn.microsoft.com/en-us/library/ms174173.aspx>.

[30] G.H. Sockut and B.R. Iyer. Online reorganization of databases. *ACM Computing Surveys (CSUR)*, 41(3):1–136, 2009.

[31] M. Subramaniam and J. Loaiza. Online reorganization and redefinition of relational database tables, November 15 2005. US Patent 6,965,899.

[32] J.F. Terwilliger, P.A. Bernstein, and A. Unnithan. Worry-free database upgrades: automated model-driven evolution of schemas and complex mappings. In *Proceedings of the 2010 international conference on Management of data*, pages 1191–1194. ACM, 2010.

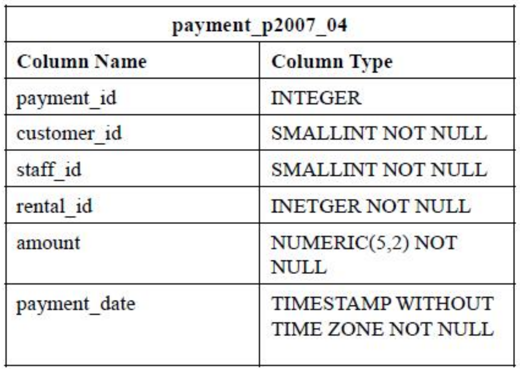
APPENDIX A

OLD AND NEW SCHEMA DETAILS

We describe the structure of old and new tables that demonstrate the schema changes applied to old schema tables involved in each schema transformation.

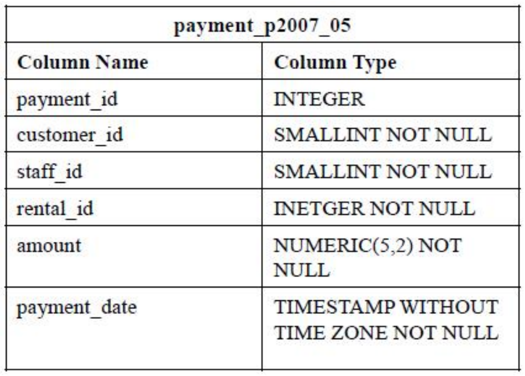
A.1 HMND: Horizontal Merge with No Duplicates

Figures A.1, A.2 and A.3 describe the schema of old and new tables involved in HMND.



payment_p2007_04	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT NOT NULL
staff_id	SMALLINT NOT NULL
rental_id	INETGER NOT NULL
amount	NUMERIC(5,2) NOT NULL
payment_date	TIMESTAMP WITHOUT TIME ZONE NOT NULL

Figure A.1: Old Schema Table for HMND



payment_p2007_05	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT NOT NULL
staff_id	SMALLINT NOT NULL
rental_id	INETGER NOT NULL
amount	NUMERIC(5,2) NOT NULL
payment_date	TIMESTAMP WITHOUT TIME ZONE NOT NULL

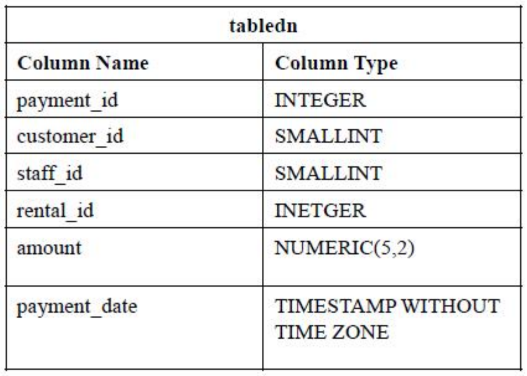
Figure A.2: Another Old Schema Table for HMND

A.2 HSEQ: Horizontal Split on EQuality

Figures A.4, A.5 and A.6 describe the schema of old and new tables involved in HSEQ.

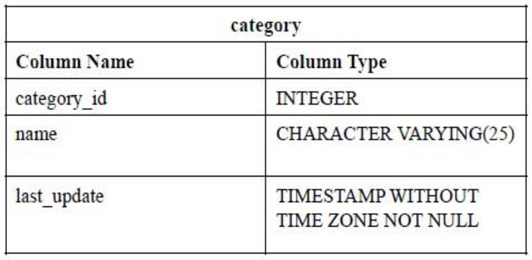
A.3 VMLR: Vertical Merge of Left and Right tables

Figures A.7, A.8 and A.9 describe the schema of old and new tables involved in VMLR.



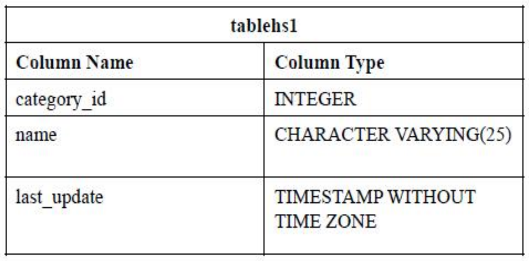
tabledn	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT
staff_id	SMALLINT
rental_id	INETGER
amount	NUMERIC(5,2)
payment_date	TIMESTAMP WITHOUT TIME ZONE

Figure A.3: New Schema Table for HMND



category	
Column Name	**Column Type**
category_id	INTEGER
name	CHARACTER VARYING(25)
last_update	TIMESTAMP WITHOUT TIME ZONE NOT NULL

Figure A.4: Old Schema Table for HSEQ



tablehs1	
Column Name	**Column Type**
category_id	INTEGER
name	CHARACTER VARYING(25)
last_update	TIMESTAMP WITHOUT TIME ZONE

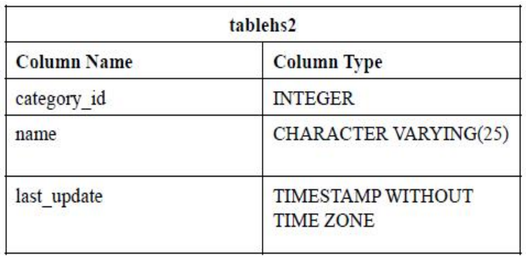
Figure A.5: New Schema Table for HSEQ

A.4 VSOP: Vertical Split on Primary key

Figures A.10, A.11 and A.12 describe the schema of old and new tables involved in VSOP.

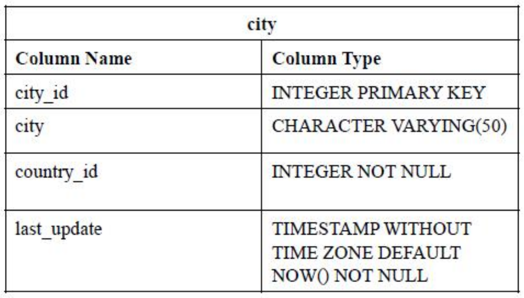
A.5 VSNP: Vertical Split on Non Primary key

Figures A.13, A.14 and A.15 describe the schema of old and new tables involved in VSNP.



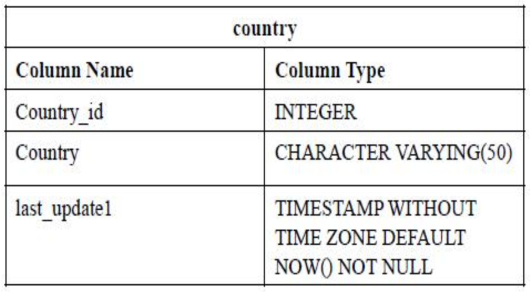
tablehs2	
Column Name	**Column Type**
category_id	INTEGER
name	CHARACTER VARYING(25)
last_update	TIMESTAMP WITHOUT TIME ZONE

Figure A.6: Another New Schema Table for HSEQ



city	
Column Name	**Column Type**
city_id	INTEGER PRIMARY KEY
city	CHARACTER VARYING(50)
country_id	INTEGER NOT NULL
last_update	TIMESTAMP WITHOUT TIME ZONE DEFAULT NOW() NOT NULL

Figure A.7: Old Schema Table for VMLR



country	
Column Name	**Column Type**
Country_id	INTEGER
Country	CHARACTER VARYING(50)
last_update1	TIMESTAMP WITHOUT TIME ZONE DEFAULT NOW() NOT NULL

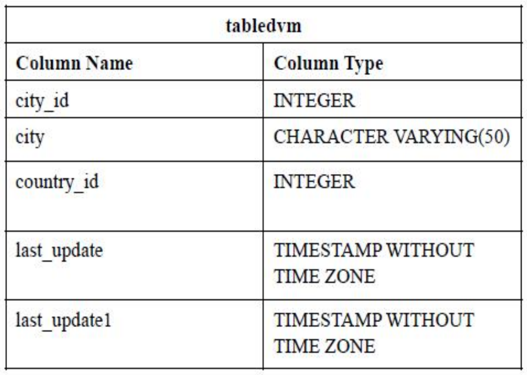
Figure A.8: Another Old Schema table for VMLR

A.6 DIND: Difference and Intersection with No Duplicates

Figures A.16, A.17, A.18 and A.19 describe the schema of old and new tables involved in DIND.

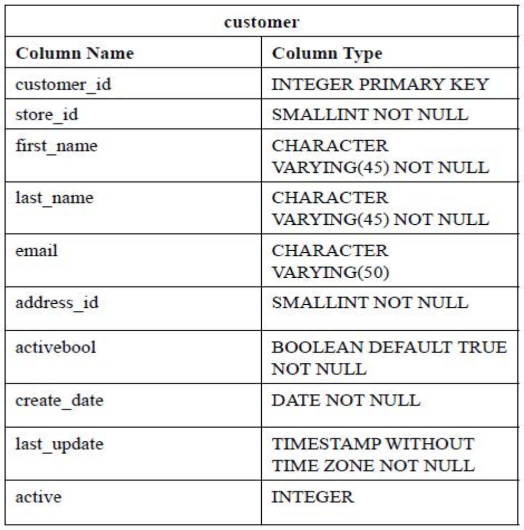
A.7 DIWD: Difference and Intersection With Duplicates

Figures A.20, A.21, A.22 and A.23 describe the schema of old and new tables involved in DIWD.



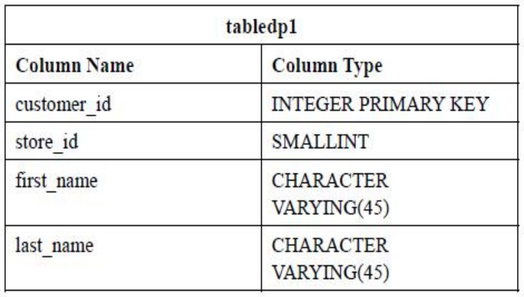
tabledvm	
Column Name	**Column Type**
city_id	INTEGER
city	CHARACTER VARYING(50)
country_id	INTEGER
last_update	TIMESTAMP WITHOUT TIME ZONE
last_update1	TIMESTAMP WITHOUT TIME ZONE

Figure A.9: New Schema Table for VMLR



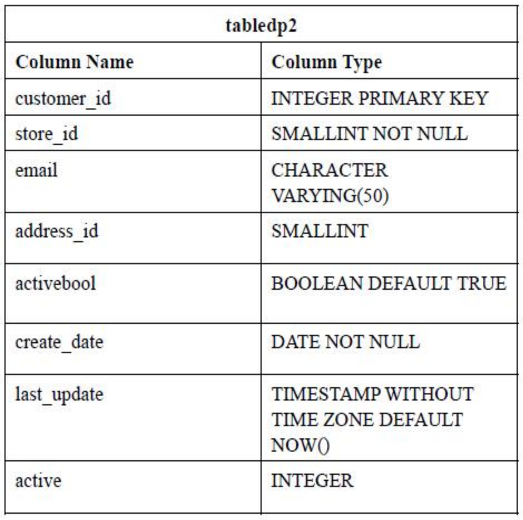
customer	
Column Name	**Column Type**
customer_id	INTEGER PRIMARY KEY
store_id	SMALLINT NOT NULL
first_name	CHARACTER VARYING(45) NOT NULL
last_name	CHARACTER VARYING(45) NOT NULL
email	CHARACTER VARYING(50)
address_id	SMALLINT NOT NULL
activebool	BOOLEAN DEFAULT TRUE NOT NULL
create_date	DATE NOT NULL
last_update	TIMESTAMP WITHOUT TIME ZONE NOT NULL
active	INTEGER

Figure A.10: Old Schema Table for VSOP



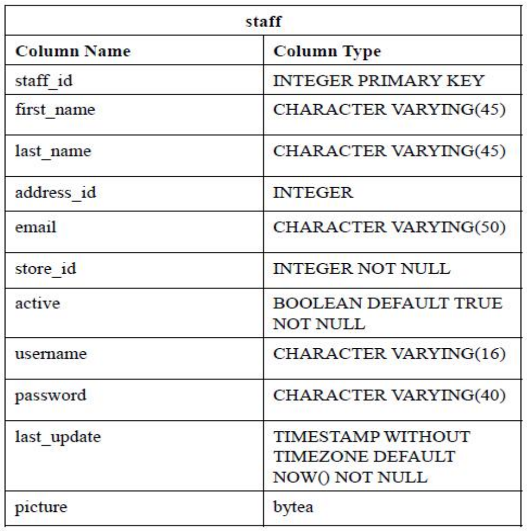
tabledp1	
Column Name	**Column Type**
customer_id	INTEGER PRIMARY KEY
store_id	SMALLINT
first_name	CHARACTER VARYING(45)
last_name	CHARACTER VARYING(45)

Figure A.11: New Schema Table for VSOP



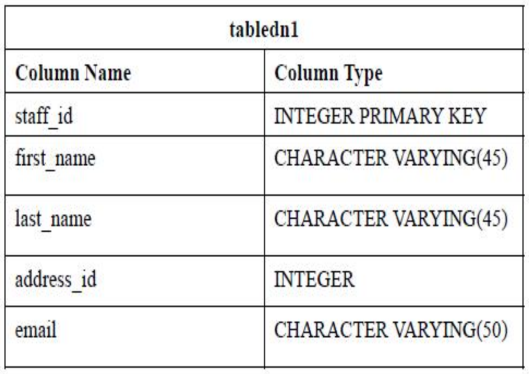
tabledp2	
Column Name	**Column Type**
customer_id	INTEGER PRIMARY KEY
store_id	SMALLINT NOT NULL
email	CHARACTER VARYING(50)
address_id	SMALLINT
activebool	BOOLEAN DEFAULT TRUE
create_date	DATE NOT NULL
last_update	TIMESTAMP WITHOUT TIME ZONE DEFAULT NOW()
active	INTEGER

Figure A.12: Another New Schema Table for VSOP



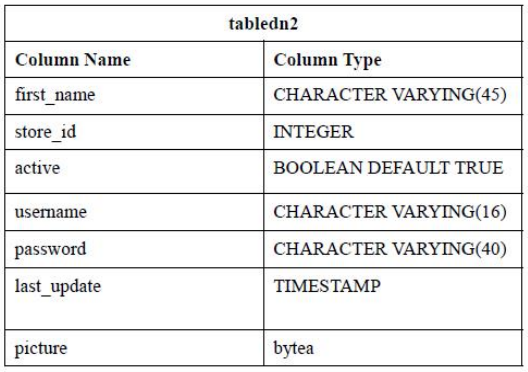
staff	
Column Name	**Column Type**
staff_id	INTEGER PRIMARY KEY
first_name	CHARACTER VARYING(45)
last_name	CHARACTER VARYING(45)
address_id	INTEGER
email	CHARACTER VARYING(50)
store_id	INTEGER NOT NULL
active	BOOLEAN DEFAULT TRUE NOT NULL
username	CHARACTER VARYING(16)
password	CHARACTER VARYING(40)
last_update	TIMESTAMP WITHOUT TIMEZONE DEFAULT NOW() NOT NULL
picture	bytea

Figure A.13: Old Schema Table for VSNP



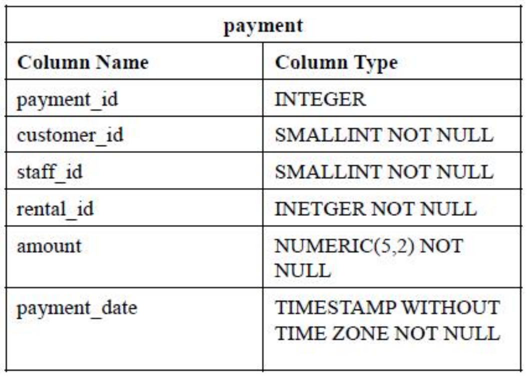
tabledn1	
Column Name	**Column Type**
staff_id	INTEGER PRIMARY KEY
first_name	CHARACTER VARYING(45)
last_name	CHARACTER VARYING(45)
address_id	INTEGER
email	CHARACTER VARYING(50)

Figure A.14: New Schema Table for VSNP



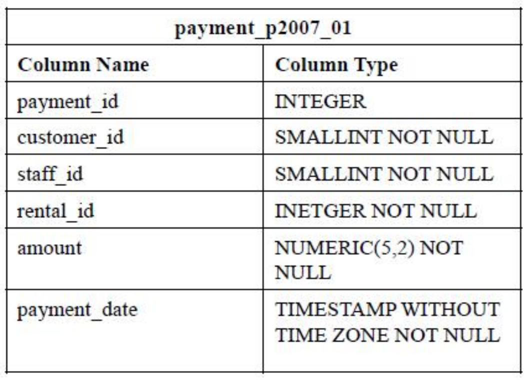
tabledn2	
Column Name	**Column Type**
first_name	CHARACTER VARYING(45)
store_id	INTEGER
active	BOOLEAN DEFAULT TRUE
username	CHARACTER VARYING(16)
password	CHARACTER VARYING(40)
last_update	TIMESTAMP
picture	bytea

Figure A.15: Another New Schema table for VSNP



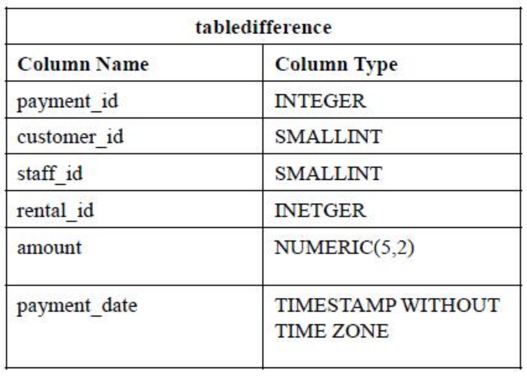
payment	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT NOT NULL
staff_id	SMALLINT NOT NULL
rental_id	INETGER NOT NULL
amount	NUMERIC(5,2) NOT NULL
payment_date	TIMESTAMP WITHOUT TIME ZONE NOT NULL

Figure A.16: Old Schema Table for DIND



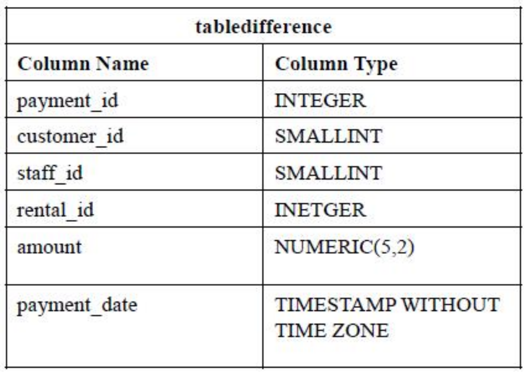
payment_p2007_01	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT NOT NULL
staff_id	SMALLINT NOT NULL
rental_id	INETGER NOT NULL
amount	NUMERIC(5,2) NOT NULL
payment_date	TIMESTAMP WITHOUT TIME ZONE NOT NULL

Figure A.17: Another Old Schema Table for DIND



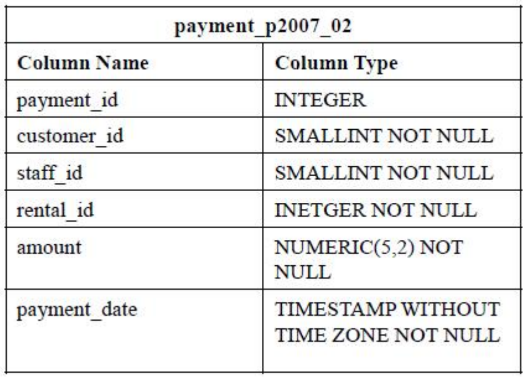
tabledifference	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT
staff_id	SMALLINT
rental_id	INETGER
amount	NUMERIC(5,2)
payment_date	TIMESTAMP WITHOUT TIME ZONE

Figure A.18: New Schema Table for DIND



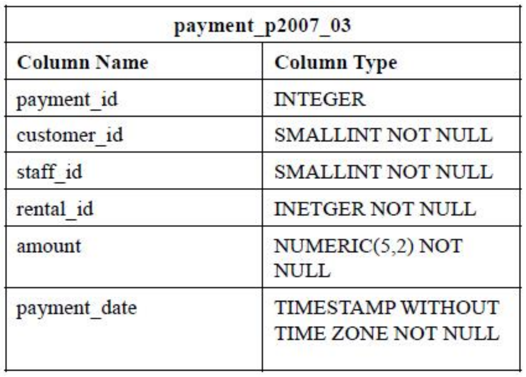
tabledifference	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT
staff_id	SMALLINT
rental_id	INETGER
amount	NUMERIC(5,2)
payment_date	TIMESTAMP WITHOUT TIME ZONE

Figure A.19: Another New Schema Table for DIND



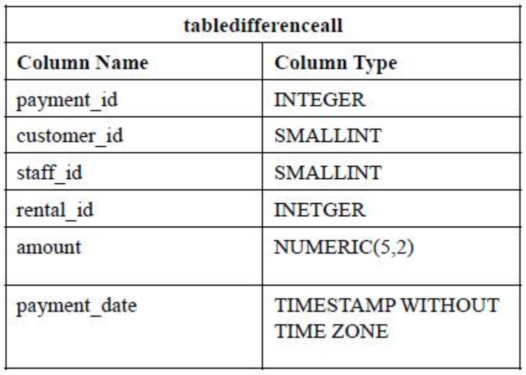
payment_p2007_02	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT NOT NULL
staff_id	SMALLINT NOT NULL
rental_id	INETGER NOT NULL
amount	NUMERIC(5,2) NOT NULL
payment_date	TIMESTAMP WITHOUT TIME ZONE NOT NULL

Figure A.20: Old Schema Table for DIWD



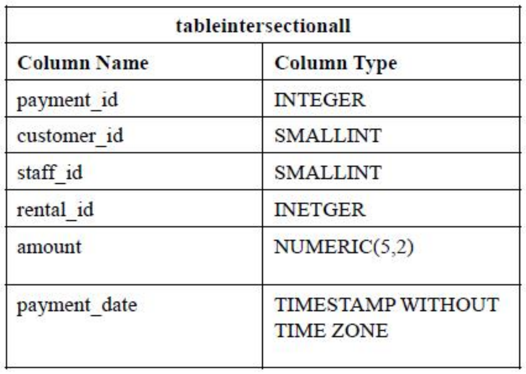
payment_p2007_03	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT NOT NULL
staff_id	SMALLINT NOT NULL
rental_id	INETGER NOT NULL
amount	NUMERIC(5,2) NOT NULL
payment_date	TIMESTAMP WITHOUT TIME ZONE NOT NULL

Figure A.21: Another Old Schema Table for DIWD



tabledifferenceall	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT
staff_id	SMALLINT
rental_id	INETGER
amount	NUMERIC(5,2)
payment_date	TIMESTAMP WITHOUT TIME ZONE

Figure A.22: New Schema Table for DIWD



tableintersectionall	
Column Name	**Column Type**
payment_id	INTEGER
customer_id	SMALLINT
staff_id	SMALLINT
rental_id	INETGER
amount	NUMERIC(5,2)
payment_date	TIMESTAMP WITHOUT TIME ZONE

Figure A.23: Another New Schema table for DIWD

APPENDIX B

SAMPLE TRANSACTION SETS

We provide the sampels of transaction sets that are generated during the query generation phase in chapter 5.

B.1 Simple Transaction Set INSERT

INSERT INTO customer(customer_id,store_id,first_name,last_name,email,address_id,active) VALUES(1348968,5577,'FName45360','LName45360','EMAIL45360',5577,45360); INSERT INTO payment_p2007_04(payment_id,customer_id,staff_id,rental_id,amount,payment_date) VALUES(5564709,1604,1604,28624,500,'2011-07-25 14:40:49'); INSERT INTO payment_p2007_05(payment_id,customer_id,staff_id,rental_id,amount,payment_date) VALUES(3636381,3642,3642,4925,500,'2011-07-25 14:40:49'); INSERT INTO category(name) VALUES('Action');

B.2 Simple Transaction Set UPDATE

UPDATE payment_p2007_05 SET rental_id='5555555' WHERE payment_id=296777; UPDATE customer SET last_name='NEW SURNAME' WHERE customer_id=282204; UPDATE payment_p2007_03 SET rental_id='5555555' WHERE payment_id=113084; UPDATE country SET country='NEWCOUNTRY' WHERE country_id=180526;

B.3 Simple Transaction Set DELETE

DELETE FROM payment_p2007_05 WHERE payment_id=64483; DELETE FROM payment WHERE payment_id=167610; DELETE FROM city WHERE city_id=154521; DELETE FROM payment_p2007_01 WHERE payment_id=144724;

B.4 Complex Transaction Set A

BEGIN; DELETE FROM staff WHERE staff_id IN (SELECT customer_id FROM customer WHERE customer_id = 144980); UPDATE payment_p2007_04 SET amount=200 WHERE payment_id IN (SELECT category_id FROM category WHERE category_id = 201807); COMMIT;

BEGIN; DELETE FROM payment WHERE payment_id IN (SELECT payment_id FROM payment_p2007_01 WHERE payment_id = 251465); UPDATE payment_p2007_05 SET amount=200 WHERE payment_id IN (SELECT category_id FROM category WHERE category_id = 224215); COMMIT;

B.5 Complex Transaction Set B

BEGIN; DELETE FROM city WHERE country_id IN (SELECT country_id FROM country WHERE country_id=125085); UPDATE customer SET email='NEWEMAIL' WHERE customer_id IN (SELECT staff_id FROM staff WHERE staff_id = 266504); DELETE FROM payment_p2007_03 WHERE payment_id IN (SELECT payment_id FROM payment_p2007_02 WHERE payment_id = 161026); UPDATE staff SET email='NEWEMAIL' WHERE staff_id IN (SELECT customer_id FROM customer WHERE customer_id = 283726); COMMIT;

BEGIN; DELETE FROM payment_p2007_04 WHERE payment_id IN (SELECT category_id FROM category WHERE category_id = 291665); UPDATE payment_p2007_05 SET amount=200 WHERE payment_id IN (SELECT category_id FROM category WHERE category_id = 7771); DELETE FROM payment WHERE payment_id IN (SELECT payment_id FROM payment_p2007_01 WHERE payment_id = 82235); UPDATE staff SET email='NEWEMAIL' WHERE staff_id IN (SELECT customer_id FROM customer WHERE customer_id = 264603); COMMIT;