CDR Structure and Related Processes

Contents

[Document Purpose 2](#_Toc409441495)

[Current Design 3](#_Toc409441496)

[Data Storage 3](#_Toc409441497)

[Business Processes 3](#_Toc409441498)

[CDR Import Process 3](#_Toc409441499)

[Pricing Process 4](#_Toc409441500)

[Repricing Process 4](#_Toc409441501)

[Problems in Current Design 5](#_Toc409441502)

[Proposed Solution 7](#_Toc409441503)

[Data Storage 7](#_Toc409441504)

[Database Design 7](#_Toc409441505)

[Data Partitioning 7](#_Toc409441506)

[Business Processes 9](#_Toc409441507)

[CDR Import and Pricing Processes 9](#_Toc409441508)

[Repricing Process 12](#_Toc409441509)

[Loading Big Configuration Data 13](#_Toc409441510)

[Impact of Proposed Solution 13](#_Toc409441511)

[Problems Solved 13](#_Toc409441512)

[Appendix I – Business Process Approach 15](#_Toc409441513)

[Business Process Framework 15](#_Toc409441514)

[Business Process Activities 17](#_Toc409441515)

[Parallel Execution 18](#_Toc409441516)

[Appendix II - Data Access Layer 19](#_Toc409441517)

[SQL Server Data Access Component 19](#_Toc409441518)

# Document Purpose

This document consists of two parts. First part shows the current design of CDR module in TOne and discusses the problems that are being faced due to this design. The second part proposes a new design and discusses what problems it should solve.

This document basically discusses reengineering the following parts of the CDR module:

* Data Storage
* Business Processes

In addition, it discusses the impact of this reengineering on the other parts (e.g. web pages, reports…)

# Current Design

## Data Storage

Currently, the CDRs are stored in the main TOne database which is an SQL server database. The CDR module consists of the following SQL tables:

|  |  |
| --- | --- |
| Table Name | Description |
| CDR | Raw CDR records as they are received from the Switch |
| Billing\_CDR\_Main | CDR records that are correctly mapped and with positive call duration |
| Billing\_CDR\_Invalid | CDR records with:   * missing information (e.g. supplier, customer) * or zero duration |
| Billing\_CDR\_Cost | The cost data of the CDR main records:   * reference to Billing\_CDR\_Main table |
| Billing\_CDR\_Sale | The sale data of the CDR main records:   * reference to Billing\_CDR\_Main table |
| TrafficStats | Time interval statistics based on grouping and aggregation of the records in the Billing\_CDR\_Main and Billing\_CDR\_Invalid tables |
| TrafficStatsDaily | Daily statistics based on grouping and aggregation of the records in the Billing\_CDR\_Main and Billing\_CDR\_Invalid tables |
| Billing\_Stats | Daily statistics based on grouping and aggregation of the records in the Billing\_CDR\_Main, Billing\_CDR\_Cost, and Billing\_CDR\_Sale tables |

All data are stored in the above tables for the lifetime of the system. Each table might have billions of records.

## Business Processes

The business processes that feed the CDR tables are:

* CDR Import process
* Pricing process
* Repricing process

The 3 above processes are part of TOne TABS component. Each of which is defined as a Runnable Task that runs based on time schedule or user request.

### CDR Import Process

|  |  |
| --- | --- |
| CDR Import Process | |
| Runnable Task | CDRImportRunner |
| Trigger | Time Schedule |
| Job Summary | It iterates over all switches defined in TOne and it fetches the new CDR records from each switch. It then update TOne database with new CDRs |
| Tables Affected | * CDR: insert * Billing\_CDR\_Main: insert * Billing\_CDR\_Invalid: insert * TrafficStats: insert, update * TrafficStatsDaily: insert, update |
| Sequence of Execution | 1. Get CDRs from switches 2. Store CDRs in CDR table 3. Build CodeMap object in memory (all code matches) 4. Build billing records (Main and Invalid) and insert them to Billing\_CDR\_Main and Billing\_CDR\_Invalid tables 5. Build traffic records and insert/update records in the TrafficStats and TrafficStatsDaily tables |

### Pricing Process

|  |  |
| --- | --- |
| Pricing Process | |
| Runnable Task | CDRPricingRunner |
| Trigger | Time Schedule |
| Job Summary | Generate Cost and Sale data from the new added records to Billing\_CDR\_Main table (check CDR Import process) |
| Tables Affected | * Billing\_CDR\_Cost: insert * Billing\_CDR\_Sale: insert |
| Sequence of Execution | 1. Retrieve new added records to Billing\_CDR\_Main table 2. Build PricingGenerator object in memory (all pricing) 3. Build cost and sale records and insert them to Billing\_CDR\_Cost and Billing\_CDR\_Sale tables |

### Repricing Process

|  |  |
| --- | --- |
| Repricing Process | |
| WCF Solution | TOne.WCFRepricingService |
| Trigger | User Request for a specific day(s) |
| Job Summary | Delete billing and statistics records for a specific day and regenerate them |
| Tables Affected | * Billing\_CDR\_Main: delete and insert * Billing\_CDR\_Invalid: delete and insert * Billing\_CDR\_Cost: delete and insert * Billing\_CDR\_Sale: delete and insert * TrafficStats: delete and insert * TrafficStatsDaily: delete and insert * Billing\_Stats: delete and insert |
| Sequence of Execution | 1. Build CodeMap object in memory (all code matches) 2. Build PricingGenerator object in memory (all pricing) 3. Retrieve CDRs from the CDR table for the selected day 4. Delete billing records from the database 5. Build billing records and insert them to Billing\_CDR\_Main, Billing\_CDR\_Invalid, Billing\_CDR\_Cost, and Billing\_CDR\_Sale tables 6. Delete statistic records 7. Build statistic records and insert them to TrafficStats, TrafficStatsDaily, and Billing\_Stats tables |

## Problems in Current Design

The system is currently facing the following problems:

|  |  |  |
| --- | --- | --- |
| Problem | Category | Details |
| Database Backup | Data Storage | It is hard to do periodic database backup due to the size of the data |
| Database Archiving | Data Storage | No archiving solution currently available |
| Deleting Data | Data Storage | The repricing process deletes data from the Billing/Statistics tables which consist of huge amount of data. This has two problems:   * The delete operation is very slow * The Indexes on the tables need to be rebuilt and this operation is also very slow |
| Inserting Data | Data Storage | The insert data to tables is slow. This is because each table has huge amount of data and is configured with indexes. |
| Database Design | Data Storage | * The Billing\_CDR\_Invalid table stores the three types of data: Invalid, Failed and Interconnect\*. The Invalid type is the most important type and it forms only a small percentage of the entire data in the table. This causes the query to Invalid CDRs to be very slow * The Cost and Sale data are stored in tables separate from the Main table. Whereas each Cost and Sale record are additional information to the Main record and the three tables could be merged in a single table |
| Slow Queries | Data Storage | * Reports that query Invalid CDRs are slow due to the types of records stored in the Billing\_CDR\_Invalid * Join between CDR and Billing\_CDR\_Invalid tables on Invalid CDRs is slow * Invoices generated directly on the raw billing data, when Carrier time shift needs to be considered, is slow. This is because it needs to join the three tables: Billing\_CDR\_Main, Billing\_CDR\_Cost, and Billing\_CDR\_Sale * Building Billing statistics (Billing\_Stats table) is slow because it also needs to join the three tables: Billing\_CDR\_Main, Billing\_CDR\_Cost, and Billing\_CDR\_Sale |
| Table with Huge Data | Data Storage | Each table would have a huge amount of data (which might reach billions of records). This also affect the queries to the tables.  There is a need for a solution to partition the data in each table (time-based partition) |
| Hard to Maintain Processes | Business Process | It is a bit hard to understand and maintain the execution of the current processes and to solve any issues. |
| Process Dependency | Business Process | Whenever there is a dependency between two processes, the dependency is controlled by using static variables. this approach causes scalability problems and disallow us to have each process running on a separate server or under a separate application |
| High Memory | Business Process | Each process is taking high memory on the server in case the system consists of too many suppliers. This is because the CodeMap and PricingGenerator objects load all data from the database to the Memory |
| Sequential Execution | Business Process | Each process is implemented in a sequential manner except some portions which are implemented using parallel threading. This causes the process to execute slowly and might cause blocking between independent steps |
| Data Access | Business Process | There is no layering in the current coding approach. The data access and business logic are almost merged in the same component. In addition, there are multiple approaches used to access the database (e.g. NHibernate, ADO.net) |

\*The differences between the types of the records that are currently stored in the Billing\_CDR\_Invalid table are:

* Invalid: the CDR has missing information and could not be mapped to TOne Supplier/Customer
* Failed: valid CDR with zero call duration
* Interconnect: this type is created with a call that is received on one switch and transmitted over another switch. Two CDRs are created in this case, one is valid and another is interconnect (Invalid table currently)

# Proposed Solution

The proposed solution is divided into three parts:

* Data Storage
* Business Processes
* Loading Big Configuration Data

## Data Storage

The following sections discusses the changes that are proposed on the database level

### Database Design

The changes to database design would be:

* Divide the Billing\_CDR\_Invalid into 3 tables:
  + Billing\_CDR\_Invalid: only stores CDRs that are invalid (with missing information)
  + Billing\_CDR\_Failed: stores CDRs with zero duration
  + Billing\_CDR\_Incorrect: stores CDRs of type incorrect (check definition in previous section)
* Merge the Billing\_CDR\_Main, Billing\_CDR\_Cost, and Billing\_CDR\_Sale in one table. This is because there is almost one to one relation between Billing\_CDR\_Main and Billing\_CDR\_Cost/ Billing\_CDR\_Sale tables

### Data Partitioning

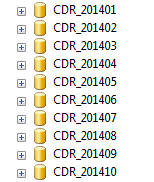
Currently TOne consists of a single SQL database. In the new solution, TOne would have the following databases:

|  |  |  |
| --- | --- | --- |
| Database | Number Of Instances | Type of Data |
| Business Entities | 1 | This database would have all TOne business entities and configuration tables. Everything in current TOne database except the CDR/Billing/Statistic tables |
| CDR | 1 instance each time period (e.g. each 1 month) | Each instance of this database will have the tables to store CDR raw data and CDR billing raw data. A table of each type would be also created for each day in the time period assigned to this database instance |
| Statistics | 1 | This database would have the traffic statistic and billing statistic tables. This includes:   * TrafficStats * TrafficStatsDaily * Billing\_Stats |

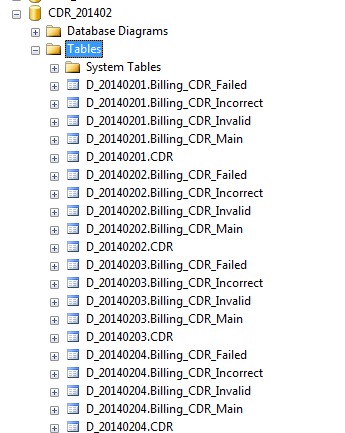
The following section discusses the CDR databases in more details

#### CDR Databases Design

A database would be automatically created each predefined time-period (each month for example):



In each database, a table of each type would be automatically created for each day:



The following challenges are imposed here and would be addressed in the proposed solution:

* TOne business processes (e.g. CDR Import, Pricing, Repricing…) will automatically routes CDR data based on the CDR Attempt date to the corresponding tables in the corresponding database.
* On the other side, TOne processes and screens that read data of this type (i.e. CDR/Billing) will automatically connect to the corresponding database(s) and query the corresponding tables based on the requested dates.
* If the output result is retrieved from multiple daily tables, they would be merged in one result set.

#### CDR Databases Backup and Archive

In the proposed solution, a number of databases would be available online based on a configured parameter. For instance, recent 6-month CDR data would be always online which corresponds to 6 databases.

* The backup process would only backup the databases that have changes from the last performed backup
* An archiving process would be implemented to detach the databases that are older than 6 months and copy its files to an archiving storage

## Business Processes

We would follow a new approach in reengineering the business processes. If not yet checked, please check the Appendix I and Appendix II to have an idea about this proposed approach

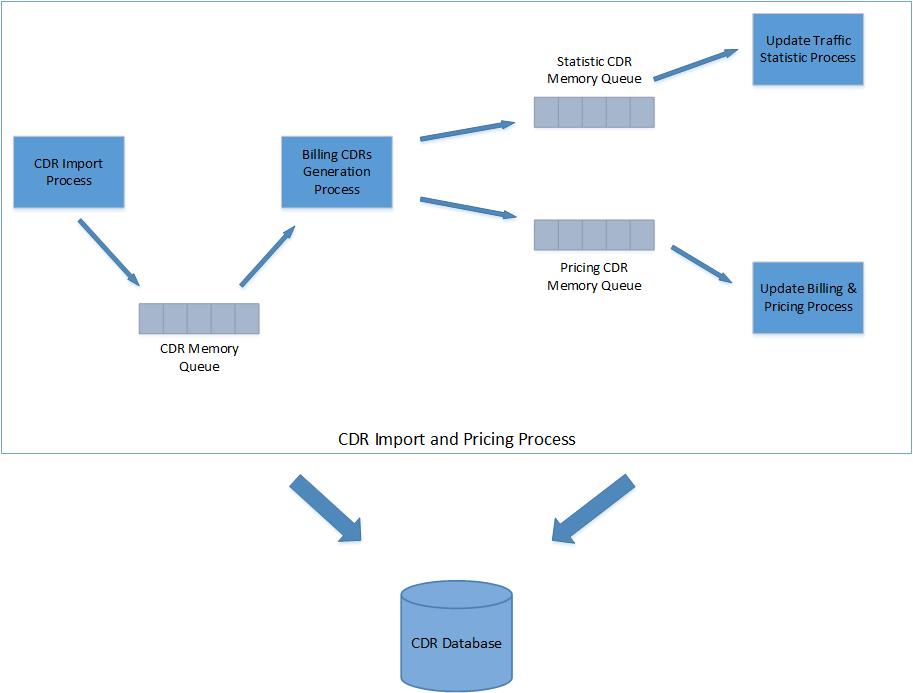
The following sections discusses the new processes and their main characteristics:

### CDR Import and Pricing Processes

The CDR Import and Pricing would be divided into 4 processes:

* CDR Import Process
* Billing CDRs Generation Process
* Update Traffic Statistics Process
* Update Billing Process

The following diagram shows the interaction between these processes:



The following tables show summary about each process

#### CDR Import Process

|  |  |
| --- | --- |
| CDR Import Process | |
| Trigger | Time Schedule |
| Input | SwitchID |
| Output | CDR Memory Queue |
| Parallel Restriction | One instance per switch |
| Job Summary | It fetches the new CDR records from that switch which has the input Switch ID. It then updates the CDR Memory Queue and TOne database with new CDRs |
| Tables Affected | * CDR: insert |
| Sequence of Execution | 1. Get CDRs from the Switch 2. Store CDRs in the CDR table 3. Insert CDRs to memory Queue |

#### Billing CDRs Generation Process

|  |  |
| --- | --- |
| Billing CDRs Generation Process | |
| Trigger | Time Schedule |
| Input | SwitchID , CDR Memory Queue |
| Output | Pricing CDR Memory Queue, Statistic CDR Memory Queue |
| Parallel Restriction | One instance per switch |
| Job Summary | it retrieve CDRs from the CDR Memory Queue and generate billing CDR records |
| Tables Affected | None |
| Sequence of Execution | 1. Get CDR record from the CDR memory queue 2. Convert the CDR to billing object (Main, Invalid, Failed, or Incorrect) 3. Insert the billing CDR object to Statistic CDR Memory Queue 4. Insert the billing CDR object to Pricing CDR Memory Queue |

#### Update Traffic Statistic Process

|  |  |
| --- | --- |
| Update Traffic Statistic Process | |
| Trigger | Time Schedule |
| Input | SwitchID, Statistic CDR Memory Queue |
| Output | None |
| Parallel Restriction | One instance per switch |
| Job Summary | it retrieve billing CDRs from the Statistic CDR Memory Queue and update traffic statistic tables in the database |
| Tables Affected | * TrafficStats: insert/update * TrafficStatsDaily: insert/update |
| Sequence of Execution | 1. Get billing CDRs from the Statistic CDR Memory Queue 2. Build new/updated traffic statistic records 3. Insert/update traffic statistic tables |

#### Update Billing & Pricing Process

|  |  |
| --- | --- |
| Update Billing & Pricing Process | |
| Trigger | Time Schedule |
| Input | SwitchID, Pricing CDR Memory Queue |
| Output | None |
| Parallel Restriction | One instance per switch |
| Job Summary | it retrieves billing CDRs from the Pricing CDR Memory Queue, generate pricing values, and update billing tables in the database |
| Tables Affected | * Billing\_CDR\_Main: insert * Billing\_CDR\_Invalid: insert * Billing\_CDR\_Failed: insert * Billing\_CDR\_Incorrect: insert |
| Sequence of Execution | 1. Get billing CDRs from the Billing CDR Memory Queue 2. update the billing CDRs of type Main with pricing values (Cost & Sale) 3. Insert billing records to billing tables |

### Repricing Process

The following table summarizes the proposed Repricing process

|  |  |
| --- | --- |
| Update Billing & Pricing Process | |
| Trigger | User Request |
| Input | Repricing Day |
| Output | None |
| Parallel Restriction | None |
| Job Summary | it regenerates the billing and statistics records for the requested day starting from the CDR table and it updates the tables in the database for that day |
| Tables Affected | * Billing\_CDR\_Main: delete and insert * Billing\_CDR\_Invalid: delete and insert * Billing\_CDR\_Failed: delete and insert * Billing\_CDR\_Incorrect: delete and insert * TrafficStats: delete and insert * TrafficStatsDaily: delete and insert * Billing\_Stats: delete and insert |
| Sequence of Execution | 1. It divides the Repricing day into time ranges based on the traffic statistics time parameter (e.g. a time range each 15 minute in the requested day) 2. It initiates Repricing sub process for each time range. Each sub process runs on another thread (in parallel) 3. In each time range it:    * It deletes the billing traffic statistic records from the database for that time range    * It reads the CDR records from the CDR table    * It builds the billing objects with pricing and it builds the statistic records in memory    * It updates daily traffic statistics in memory (shared dictionary between all time range sub processes)    * It update billing and traffic statistics tables in the database 4. After all time range sub processes are completed, the main process continues 5. It updates Billing statistics in the database |

## Loading Big Configuration Data

The approach of loading and processing large data (e.g. CodeMap, PricingGenerator) would be changed to give more appropriate and non-blocking solution. Currently, the entire object is loaded in-memory.

The proposed solution consists of getting the required data upon request from the database based on a predefined key. And putting the retrieved data in a cache store instance available for the lifetime of the workflow instance

The key of the CodeMap used to retrieve data from the database and put in the cache is based on the:

* SupplierID
* First digit of the code to which a code match is requested
* EffectiveDate

The key of the PricingGenerator is based on the:

* CustomerID
* ZoneID
* EffectiveDate

## Impact of Proposed Solution

The followings would be affected by the proposed solution:

* Web pages
* Reports
* Stored procedures (e.g. update billing statistics)

This section is not yet ready

## Problems Solved

The following table shows the problems that would be solved by the proposed solution:

|  |  |  |
| --- | --- | --- |
| Problem | Category | Solved By |
| Database Backup | Data Storage | Database partitioning into multiple databases |
| Database Archiving | Data Storage | Database partitioning into multiple databases |
| Deleting Data | Data Storage | Daily Tables. The entire table would be replaced in case of Repricing |
| Inserting Data | Data Storage | Daily Tables |
| Database Design | Data Storage | Fixed by new Database design |
| Slow Queries | Data Storage | Fixed by Database partitioning, Daily Tables, and new Database design |
| Table with Huge Data | Data Storage | Fixed by Database partitioning and Daily Tables |
| Hard to Maintain Processes | Business Process | Fixed by new Business Process approach |
| Process Dependency | Business Process | Fixed by new Business Process approach – Business Process framework |
| High Memory | Business Process | Fixed by loading on demand |
| Sequential Execution | Business Process | Fixed by new Business Process approach |
| Data Access | Business Process | Fixed by new Data Access Layer |

# Appendix I – Business Process Approach

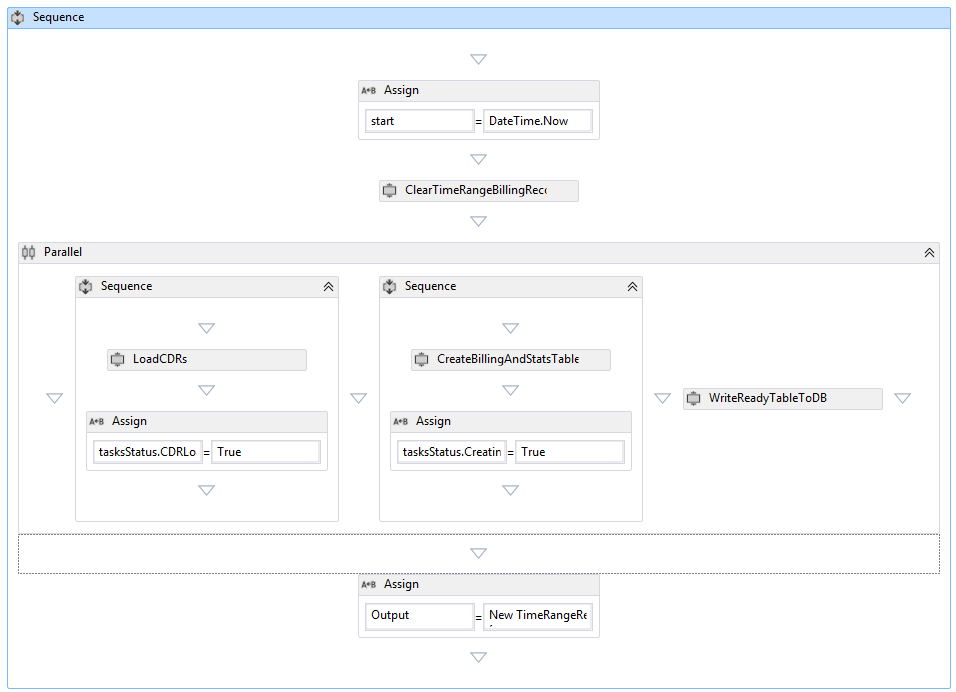
The following approach would be followed for all business processes that are to be built or rebuilt:

* We would have a common Business Process Framework based on Microsoft Windows Workflow Foundation (WF4). All business processes will be built on top of this framework
* Each business process would be fragmented into independent parts (called activities). Each activity would have a very specific job with input and output arguments
* Parallel execution of independent parts would be implemented when appropriate

### Business Process Framework

The Business Process Framework is based on Microsoft Windows Workflow Foundation (WF4) and gives the following features:

1 – Visual designer to draw the main execution of the business process:



2 – Common execution manager which would be responsible of:

* Controlling the number of instances that can run in parallel of each process
* Retry instance execution on failure if needed
* Communication between different processes. Event-based communication
* Ability to implement long running processes. For example, a process reaches a state when it waits an external event before continuing execution (e.g. time schedule, user approval, event triggered by another process)
* Ability to have process instance hierarchy. In this scenario, a process instance can run another process and receive notification once it is done. A parent-child relation would be defined for these two process instances
* Resolve dependency between different processes and different instances. For example, it is not allowed to run multiple instances of the Repricing process for the same day
* State persistence when needed (i.e. long running processes)

3 – Common Process State management and tracking:

* Common database for all business processes that have a record for each created instance which its final state
* Common database that store the tracking messages and status change history of the processes
* Common UI to browse those databases

### Business Process Activities

Each business process would be split into multiple portion, each of which would be responsible of a very specific job. This portion is called Activity and has the following characteristics:

* The business process will be built using a set of Activities and will define the sequence of execution of those activities
* The Activity is a black box independent from other activities and even from the business process that is part of
* Each Activity defines its input and output arguments

This approach would give the following benefits:

* Separation of concerns
  + Each activity address a very specific business need (e.g. load new CDRs, build billing records, update statistic table…)
  + The business process defines the execution of the activities. In the business process, we would have full concentration on the execution pattern we want to use (sequential, parallel, multiple instances, waiting external events…)
* Testability and maintainability
* Each activity would be reusable within the same process and in multiple processes
* Ability to configure retry on failure on the activity level

### Parallel Execution

As described in the previous section, there is separation between the business implementation and the execution logic. The business implementation is the handled in the Activity implementation. The execution logic is handled by the business process. This gives us more abilities to define Parallel execution of a specific logic and address most of the parallelism challenges (e.g. synchronization, bottleneck...)

Two main options are available to obtain parallel execution:

* By running multiple instances of the same business process. Each of which with different input arguments
* By defining parallel execution of the activities within a single business process

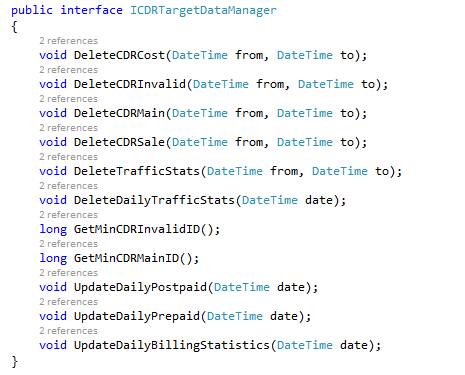
# Appendix II - Data Access Layer

Data Access is a big concern especially in systems similar to TOne. Currently, the data access is merged with the business logic component. The new proposed approach would be as follows:

* All transactions to database, read and write, will be delegated to a separate layer called the Data Access Layer. This layer would only be responsible for providing and manipulating data and would not perform any business logic or any other functions

The Data Access Layer would be divided into two parts:

1. Abstract definition of all operations needed by the system to access/manipulate data. This relies on defining OOP interfaces:



1. Multiple implementations of the data access interfaces against multiple database platform as needed:
   * SQL server database implementation
   * Oracle database implementation
   * MySQL implementation

## SQL Server Data Access Component

For each database platform, there would be a unified approach to access/manipulate data in the data store. For example for SQL server database, we would a have a base class which gives all types of database operations that would be performed on the database. The implementation of this class is based on Microsoft patterns & practices Enterprise Library

