MCMASTER UNIVERSITY

A data synchronization solution with data quality support

Useful, friendly and reliable cleaning to improve data quality

Project report for CAS 749: Advanced topics in data management

Erik, Yu Wang
erik.wang@mcmaster.ca
12/12/2013

Supervise by Dr. Chiang, Fei

Contents

Change History

The following Change History log contains a record of changes made to this document:

Published / Revised	Version #	Author	Section / Nature of Change
16 Sep 2013	1.0	Erik Wang	Initial Release, missing test result and abstract.
12 Dec 2013	1.1	Erik Wang	For 749 project report

Abstract

With increasingly demands of data in industries, recently research towards to find and apply new technique to improve data quality. Meanwhile, people become focus on data audit in terms to ensure they are able to be advised of any data change for the sake of data security purpose. But with significantly data explosion, efficiency and accuracy turn to the major challenges of the industry. In this research, we purpose a ready-to-go solution, as an implementation of common data quality technique that been modularized to our data synchronization engine. We purpose a parallel-and-partition algorithm to expedite the comparison process within acceptable time consumption. Our engine helps to reduces database workload, find data changes and dirty data during synchronization process. In addition, it suggests cleaning operations and fixes them automatically with friendly user interaction interface. We evaluated scalability and accuracy of our engine on million level data with data cleaning modules. The results show synchronization time consumption has limit increasing, the constraints matching is correct as expected. We believe it is feasible to add data cleaning modules to data synchronization tool in terms to improve data quality.

Introduction

Nowadays, RDBMS productions still act as major data carrier. Due to the fact of heterogeneous application systems are designed for different business, more and more requirements are raised regarding to transform information among systems, for the sake of business integration. With increasing quantity, people never stop researching on how to do the job faster and more precisely. Meanwhile, people have higher standard to data quality. One research direction is towards to find technique so that we can perform data cleaning and audit during data transmission process.

Data consistent is very important to today's research and business. The main reason causes inconsistency is that different data source come from different subject, different geography and managed by domain experts. Like financial service, the branches over the world, with different data format due to the nation's custom. And it is too expensive to connect to the center database instance in real-time like a star topologic. In this case, currently the popular solution is to use ETL tool as connector to transform data. But so far, typical ETL solutions don't have particular mechanism to audit data change and ensure data quality.

On the other hand, the essential requirements to data transform are fast and accuracy. Suppose work on the quantity at million level, it is impractical to do a synchronization process if it takes many hours. Regarding to accuracy, like financial data need one hundred percent correct otherwise it damages the business and customer. They also need to know any unexpected data change like malicious purpose. The situation shows like **Fig 1**.

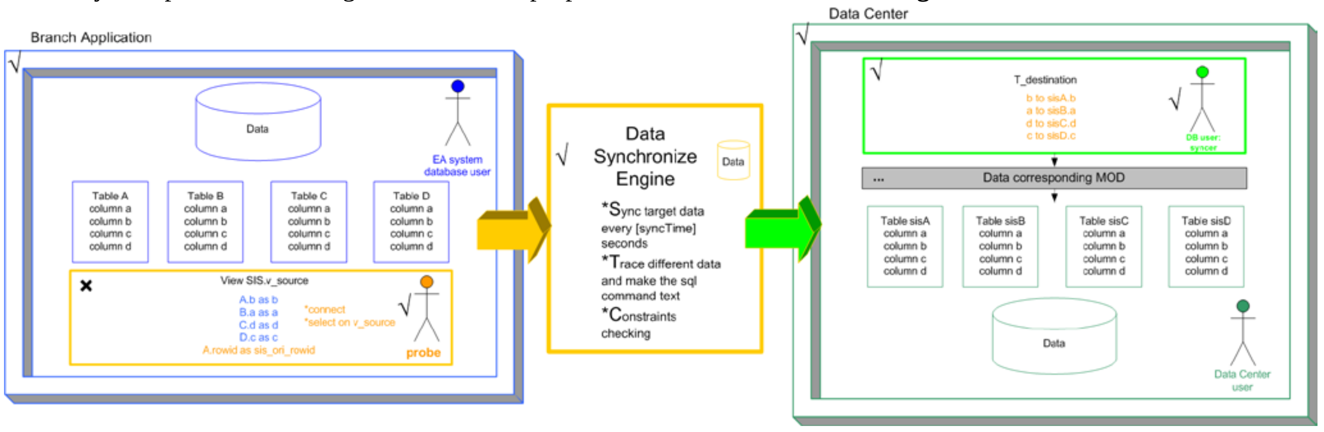

Branch Application
√
Data
EA system database user
Table A column a column b column c column d
Table B column a column b column c column d
Table C column a column b column c column d
Table D column a column b column c column d
×
View SIS.v_source
A.b as b
B.a as a
C.d as d
D.c as c
A.rowid as sis_ori_rowid
*connect
*select on v_source
√
probe
√
Data Synchronize Engine
Data
*Sync target data every [syncTime] seconds
*Trace different data and make the sql command text
*Constraints checking
Data Center
√
√
T_destination
b to sisA.b
a to sisB.a
d to sisC.d
c to sisD.c
√
DB user: syncer
...
Data corresponding MOD
Table sisA column a column b column c column d
Table sisB column a column b column c column d
Table sisC column a column b column c column d
Table sisD column a column b column c column d
Data
Data Center user

Fig 1. An example of data engine work for data synchronization

I completed the engine phase-one before start this project. To continue it, in this project my research focuses on how to apply for data clean technique in terms of improves data quality during synchronization. The main challenges in the research are:

First, consider common scenario, there are sort of data quality technique could be involved. But what kind of quality technique is suitable to data synchronization? Second, the running performance concern, since the target data quantity at million level, any piece design detail would consume extra time, it will eventually accumulate to significant slow. So how to implement the quality technique but still keep the engine with good performance? Third, as a tool, user expect a flexibility interface/method to define their own synchronization strategies, how to satisfy this requirement? Fourth, to observe and prove the research result is correct, how to design the experiment with insufficient hardware/software condition?

Our solution includes a design of efficient algorithm to find data change, then involve data filter and conditional functional dependency as data cleaning module. Next, we enable user define his own configuration file to indicate the transform strategy. Finally, we design a practicable experiment to validate scalability and accuracy.

Contributions

Our contribution includes: First, we built a ready-to-go tool which can be used at common circumstance of data synchronization, to find data change and dirty data, then auto fix them, at million-level within acceptable time of running. Second, we gave two implementation of data quality technique, they can be applied and improve data quality as a module of data ETL tool. As modularization design, it gives flexibility to developers and users. Third, we demonstrated an experiment to show the scalability of the engine and the impact of data quality module to the engine.

Background and related work

Personally, I engaged this problem in 2005 when I worked in university. The university has several of business systems owned by different departments respectively. Before 2002, there are no connections among the systems. The owners use file or spreadsheet to transform data. With increasingly demands, some requirements reflect to improve quality. In industry, to solve this kind of issue, the most common approach is to do by use database query manually. Database engineers and application owners are involved to figure out the strategy. The result is that data quality is always low but costly, with high inconsistence rate, sometimes even incomplete.

More precisely, there are three popular techniques in industry: The first one is full manually, query by query. It suits for the small and not too complex data source. The problem is low efficiency, particularly with many data sources and complex data mapping. The second method is to write all the strategy by SQL scripts (e.g. PL/SQL) then deploy database schedule to perform it automatically. It is easy to run but the problem is that those scripts are very hard to write when dealing complex process, and also hard to maintain. When add a new subject to the strategy, it is quite costly to modify the script. Both of the two methods have following several common challenges: the query could be very slow when deal with huge quantity data. Normally, to find data content change, we will perform a cross-check like **Fig 2.** In the worst case, given ***n*** tuples, it could be ***n*** times ***n*** queries. The increasing is at exponential level. Moreover, it is hard to find data change and make it traceable, logged and give feed back to users friendly. Finally, the running workload is on the database server, it leads reductions to the database performance.

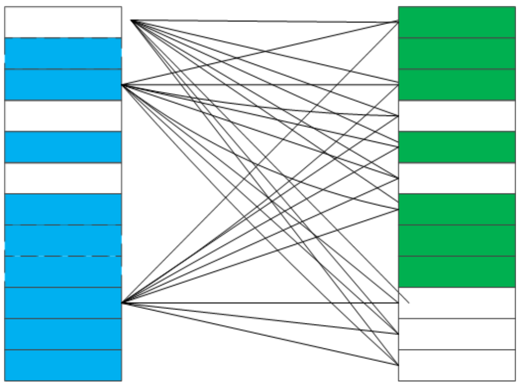


Fig 2. A worst case of cross check

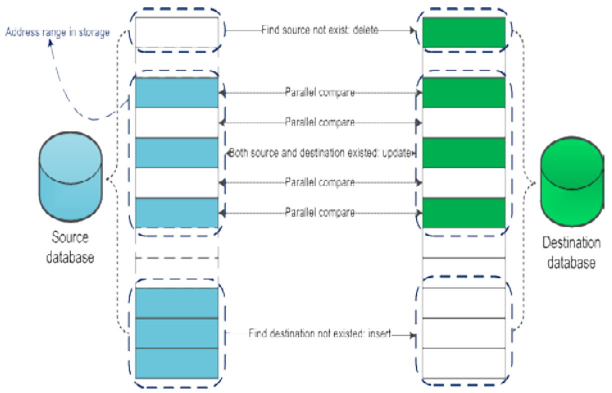


Fig 3. Parallel + Partition (P&P)

The third method is to use commercial or opens source ETL tools. Even this, as my knowledge, so far, all the three methods can't support data quality checking well. Also, it needs extra expensive budget and study curve to master the tools. A summary of challenges shows as **Table 1**.

Due to the scenario, our solution is to build a data synchronization engine with the features of efficiently, friendly, and support popular data clean techniques.

Challenges	**Past approach**	**Problems**	**Improved method**
Data transfer tool issue	No particular tool. Data sent by spreadsheet via email, or do import to database. Data change can't be found. Hard to trace and log data operations. Many people work and repeated works.	Low efficiency. Hard to maintained. Hard to trace and log. Run on database server. No quality technique	Need a tool. Tool can support trace and log. Tool is friendly. Tool can support data quality technique.
Efficiency issue	Optimize SQL query Use PL/SQL script	SQL queries are much costly and slow while on huge data quantity. Workload on database's hardware	Reduce query times. Run outside database server.
Data quality issue	Manually check or do cleaning after transfer	Couldn't support data quality technique well.	Support data quality technique during synchronization.

Table 1. Challenges to data transformation tasks

Dbsync engine

We named our solution "dbsync engine". The design follows the idea of modularzation. It includes five modules as **Fig 4**. Beside foundational support modules, developers can impement their own modules in terms of work for various circumstances (e.g. for SQL server or other database). In our case, based on particular requirements, we built and tested modules for Oracle database.

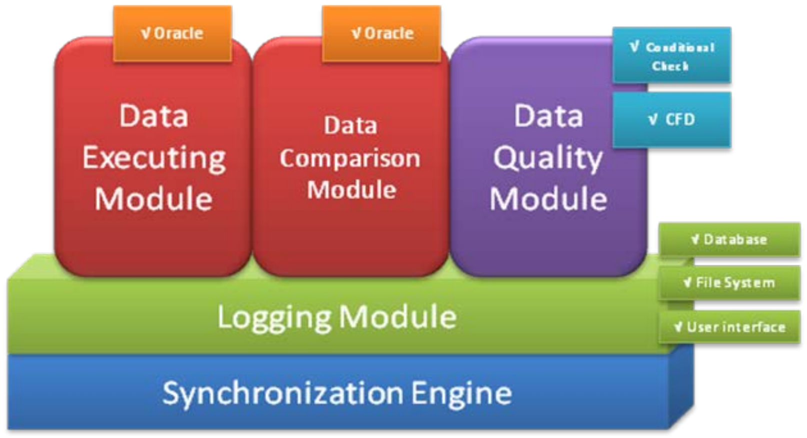


Fig 4. Modules of dbsync engine

P&P synchronization algorithm

Our solution to ensure efficient synchronization is the algorithm of parallel and partition (P&P) generally shows as **algorithm 1**. It uses space to trade for time. This is due to hardware is cheaper while time requirement is higher, it is worthy to trade-off. More specifically, we use extra space in memory to map storage address for the sake of make cross-check to paraller-check, detail see **Appendix 1**. When face huge quantity data, to solve the insurfficient memory issue, our solution uses parition technique, to divide data into blocks. When a whole block needs do sychronize, our engine will call the bunch data operation instead of do many single on- by-one query, while brunch operation is much faster. The comlexity of paraller and partition is: given ***n*** tuple and let ***k*** be the size of each block by partition. It is $\left(\frac{n}{k}\right)^2 + k$ as increasing at sub-exponential level. More functional advantages show in **Table 2**.

```
load user defined strategy from configuration file
for each synchronization do
        calculate number of tuples n
        get settings k as block size
        if  k < n then
            get lower bound (ad_low) and upper bound (ad_up) by method GetBound();
            call parallel compare pc(n, ad_up, ad_low);
            do next iteration;
        else
            compute upper and lower boundary for each partition
            get address boundary for each partition par_ad_groups by method Vector<String, String>
        for each unit in par_ad_groups
                call pc(k, ad_up, ad_low);
        end if
do next iteration
```

Algorithm 1. P&P synchronization algorithm

Factor	Traditional SQL	dbsync engine
Comparison method	Cross-check	Partition + Parallel
Residential	Run on one of the databases	Either side of databases, or a 3rd party box
Workload to database instance	Heavy	Lighter (select from single side)
Compare each attribute	No, or very complex PL/SQL	Yes, user define
Generate support SQL	No	Generate Insert/Delete/Update, and repairing suggestions
Support data quality check	No, or very complex PL/SQL	Yes, conditional check, CFD
Traceable / Logging	Yes, by DBMSs level logging	Yes, logs to file system, database, user interface
Schedule run / Batch run	Yes, implement on DBMS	Yes, user define
Expansibility	Bad	Good

Table 2. Advantages in functions of our engine

Data quality policy

The next module, as the main work in the project is data quality module. Currently, our engine supports two data quality techniques. The theory of its working includes following steps: The engine will load user defined policies into memory, during tuple comparison, it will apply the policy to tuple, and then catch those tuple which satisfied the policy, give feedback to user, finally do corresponding actions/operations automatically as user setting. To ensure our friendly user interaction principle, all the quality policies are built as a class from programming aspect. By understand the meaning of configuration files and interfaces, developers can implement their own policies by their own designs. The configuration of the quality policy was described as XML format. With well-designed logging system, the entire data quality log either display on user screen instantly, or is written to a local log file, or save to database.

General Data Filter - GDF

As the name, GDF is a general filter which applies condition check during data synchronization process. User can give some general conditions to the strategy. For example, some explicit errors which against ground truth are no need to figure out what are the possibility of their correction. In additional, user can also use this filter to enhance his synchronization strategy. For example, if several tuple for the same object but with different timestamp, the engine will read the latest one.

In our engine, user can define his filter condition by XML format. An example likes **code. 1**:

```
<FD>
            <FID>1</FID>
            <FATTR>VALUE</FATTR>
            <FOPER>great</FOPER>
            <FVALUE>2000.05</FVALUE>
</FD>
```

Code 1: GDF user define example

The filter supports user defined operators, e.g. “greater”, “lesser”, “not equals”, “between” and so forth. Developer could implement the definition by their own strategy.

From program design view, we built a class for GDF. It includes units which indicate the attribute (name or LHS or the condition), operators and values (or RHS of the condition) for database entity. In this case, the filter could catch those tuple which satisfy the condition. GDF can be widely used in all kinds of synchronization scenarios. For example, to find abnormal value which than expected scope, in case of liquid water temperature greater than 100 degree Celsius, or credit card transaction than limit, such kind of dirty tuples.

Since all of the data is loaded to memory, we can implement more complex computing at program level, on the contrast, to do the same computing by SQL script is limited and costly. User can define complex operator, or bring a group of attribute values to the GFD class. This is another advantage of GDF.

Conditional Functional Dependency

The next technique is CFD. We know it is common problem that data format in different system is heterogeneous. By using CFD, user can define constraints specifically to different conditions. It is widely used to find inconsistence data. Not like FD or other repair decision based on probability, CFD describes data relations in a certain way (even wildcard can be considered as have a value). It is very suitable for high demand on data accuracy.

Like GDF, CFD is another implementation of data quality policy by implement the interface. A user-define CFD likes C**ode 2.** We use key words "LHS" and "RHS" to indicate attribute name and value as string format. Users can also define whether they wish to use clean suggestion feedback or enable auto cleaning during the process that can be written in the configuration file. While the module is running, our engine will read the user defined configuration file then interpret CFD policies to new memory space, as **Fig 5.** Those memory spaces will existing until entire related job completed. During the synchronization, engine enters CFD module while loading a tuple. More specific process as **Algorithm 2**.

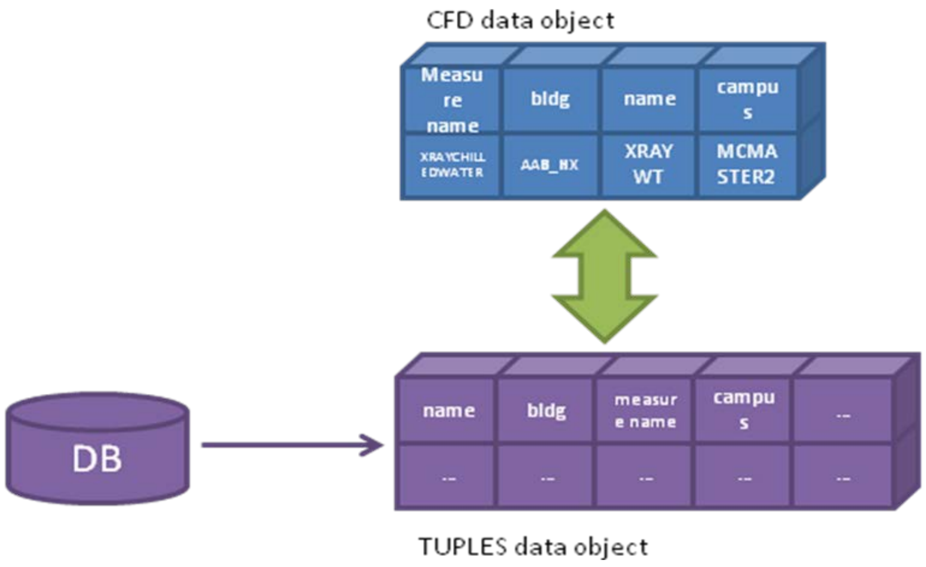


Fig 5. CFD checking in memory objects

A worthy highlight here is that considering improve efficiency, our CFD computing is in memory, similar to find tuple change. If LHS doesn't match CFD, the algorithm will not do future RHS computing. Meanwhile, the suggestion clean operation can be generated by read both tuple and CFD from memory at the same time, so that no addition query is required to database. Moreover, it won't do the clean right away, one-by-one. The engine will put all suggestion operations to user interface, logs and a ready-to-execute SQL queue. The queue execution is managed by the engine with optimized SQL execute module. Those features ensure don't reduce running time.

```
<CFD>
            <CFDSUGGESTSQL>YES</CFDSUGGESTSQL>
            <CFDAUTOCLEAN>NO</CFDAUTOCLEAN>
            <CFDID>1</CFDID>
            <CLHS>
                  <CLATTR>MEASURENAME</CLATTR>
                  <CLATTR>BLDG</CLATTR>
                 <CLVALUE>XRAY CHILLED WATER</CLVALUE>
                 <CLVALUE>ABB_HX</CLVALUE>
       </CLHS>
       <CRHS>
                 <CRATTR>NAME</CRATTR>
                 <CRATTR>CAMPUS</CRATTR>
                 <CRVALUE>XRAYRWT</CRVALUE>
                 <CRVALUE>MCMASTER2</CRVALUE>
       </CRHS>
 </CFD>
```

Code 2: CFD user define example

```
load CFD from user configuration file
assign memory to each CFD object – cfd(LHS, RHS, suggest, autoclean)
while engine do any operator to tuple
            if (cfd.CheckLHS(cfd.LHS, tuple.getAttribute(LHS, attributeGroupToCompareWithCFDLHS)))
              if(cfd.CheckRHS(cfd.RHS, tuple.getAttribute(RHS, attributeGroupToCompareWithCFDRHS)))
                        countMatch ++;
            else{countNotMatch++;
                  if(suggest){
                    call GiveSuggestSQL(cfd,tuple);
                        if(execute){
                            call  SaveToExecuteSQLList();
                        }
                  }
            }else{
                  countNotMatch++;
            }
```

Algorithm 2. CFD checking algorithm

Experiments

The experiment was designed to test two assumptions: First, as an engine, we need to ensure his reliability and stabilization of running on million level data. Moreover, it is also important to check the scalability. Second, we need to confirm the data quality module can work efficiently and accuracy.

Experiment preparation

To test scalability of our engine, we need at least three groups of tuple with increasing quantity. Test data based on McMaster University's building monitoring database. It includes timely measuring data, like temperature from different buildings. The original size is 200K. We prepared another two data group as 1.6M and 3.2M based on the first 200K. To test data quality module, we defined two policies corresponding to GDF and CFD.

The tested GDF policy indicates "whether there is a value greater than 2000.5", if catch one, give feedback to user (see **Table 3**). Regarding to CFD for test, by observe and understand the meaning of the data set, we first found

CFD candidate group by run SQL query with GROUP BY clause, including LHS attributes and RHS values. In this case it is easy to find the number of tuple which satisfied potential CFD relations. We built a CFD policy with four attributes, a given tableau value example as **Table 4**.

We designed three variables which can be changed in terms of demonstrate different synchronization strategies. The first variable is ***n*** for test data size, as 200K, 1.6M and 3.2M. The second variable is the block size ***k*** (how many tuples in the one partition) of partition, from 1M to 3M, increasingly. The last variable is a switch to enable or disable data quality policy to the synchronization. The experiment hardware is a laptop with Inter i7 processor, 16GB memory and 1T 5400 rpm hard driver. Software were used list in **Table 5**. The result shows as **Fig 6**.

Experiment conclusion

From test result, as our expectations, we get following points: First, the block size will dramatically increase running time. The reason is that each time the database do a bunch tuple query, it costs time to create query statements and transactions. Due to the size of each block, if it is too small to database memory buffer size, the database needs extra operations for create another query transaction, the time of create and communication is costly, so that it reduces the overall performance. Second, data quality module leads minor increasing to the running time, however, even that, accumulate to million level, at the end the total increasing is obvious. Third, the scalability of the engine is linear increasing, rather than exponential, for with or without data quality module. Finally, the number of filter and CFD match is accurate as we designed, all of the feedback either on user interface or log files are easy read and traceable.

ATTRIBUTE	OPERATOR	VALUE
VALUE	≥ (greater)	2000.5

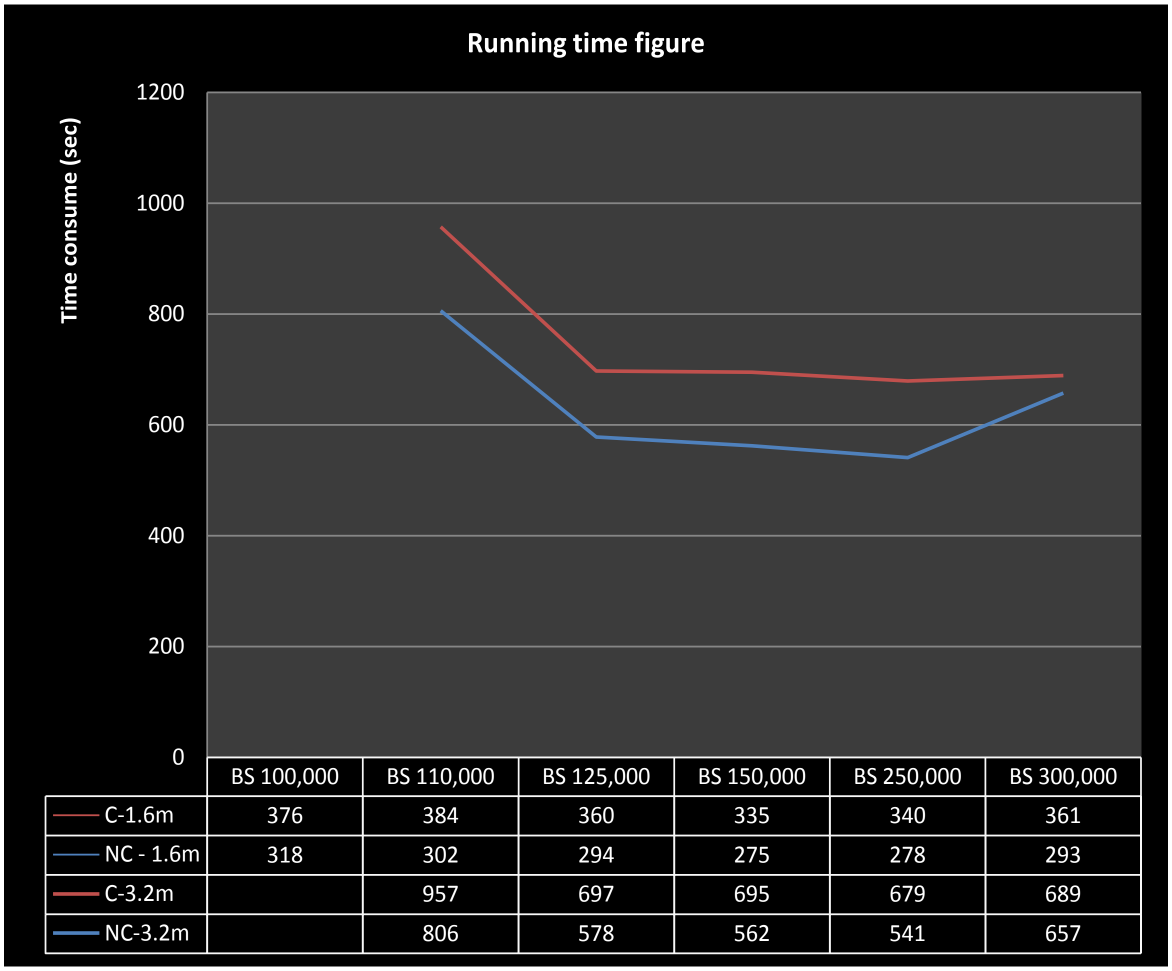
Table 3. Filter policy for experiment

φ1 = ([MEASURENAME, BLDG] → [NAME, CAMPUS], T1)	
LHS	**RHS**
MEASURENAME, BLDG	NAME, CAMPUS
“XRAY CHILLED WATER”, “ABB_HX”	“XRAYWT”, “MCMASTER2”

Table 4. CFD for experiment

Item	Describe
Database	Oracle 11G R2 32 bit
OS	Oracle enterprise Linux
Host	VMWARE work station 9
VM host	Windows 8.1 64 bit
Java	JDK 7

Table 5. Experiment software environment



	BS 100,000	BS 110,000	BS 125,000	BS 150,000	BS 250,000	BS 300,000
C-1.6m	376	384	360	335	340	361
NC - 1.6m	318	302	294	275	278	293
C-3.2m		957	697	695	679	689
NC-3.2m		806	578	562	541	657

Fig 6. Experiment result, c = with cleaning module, nc = without cleaning module, bs = block size

We learned following experience from the experiments: Firstly, we need to focus on every detail for the sake of improve the speed, because this effort will accumulate to benefit the finally result while deal with huge quantity data. Secondly, add data quality policy module in ETL process is feasible, because all computing in the third party memory which is very faster than do it based on DBMS, in our case, the extra workload for data checking is not on database server. Thirdly, by using GDF and CFD, users can essentially improve productivity to data transformation jobs, particularly make good use of CFD module, otherwise it is complex to use traditional SQL script to check and clean (modify) dirty data, even that it without good logging and statistics. Finally, as a practical tool, friendly user interaction and sound logging system is very important to ensure data transformation, either in test purpose or actual works.

Applications

Our engine is ready-to-go, it helps many situations especially using Oracle database. For example, in the financial industry, branches can use our engine to do daily data synchronization to center bank. Meanwhile, if user enables

GDF and CFD, it could potentially find dirty data and fix them automatically. The database engineer can define all of the synchronization strategy, policies and schedules. All the operations will be logged and traced. The next application example is when inside an organization, to do data audit in terms to find any data change. As we know, it is costly by use Oracle archived data log to find data change, while our engine can help to run periodicity to catch them. Finally, people can use our engine as a data cleaning platform to deal with data cleaning tasks.

But we need to point out that this engine doesn't has good automatically verification mechanism to ensure job's completeness. And many optimizations are expected to improve the performance. So far, from efficient concern, our engine is not suitable to those very large (10 million full synchronization with log and cleaning module will take more than two hours) data quantity due to comparison algorithm is still essentially based on string match.

Conclusions

Based on the research result, we believe that it is feasible to involve data quality policy to data transformation jobs. Comparing with past method (do the job based on database SQL query), the individual program/engine brings following improvement to the process: First, it reduces database workload, the save the investment on expensive database hardware. Second, it is much faster than cross-check query. Third, developer can implement various data quality policy as modules which can be loaded to the engine so that it has flexibility to the users. Finally, due to work on million or even bigger quantity, it is very important to optimize every detail to reduce execute time, either choose right technique or appropriate configuration setting. Any minor difference will eventually reflect to significant spending.

In future, we will focus on implement more data quality technique like MDM, ranking data repairing mechanism. Another effort could be on performance aspact by improve data execute and data comparison module. We also think data privacy technique could be modularized to our engine as a Trusted Third Party solution.

References

- Oracle online documentation http://www.oracle.com/technetwork/indexes/documentation/index.html
- Oracle Java development documentation http://www.oracle.com/technetwork/java/javase/documentation/index.html
- Erhard Rahm, Hong Hai Do, *Data Cleaning: Problems and Current Approaches*
- Philip Bohannon, Wenfei Fan *Conditional Functional Dependencies for Data Cleaning*
- Nilothpal Talukder, Mourad Ouzzani, *Detecting Inconsistencies in Private Data with Secure Function Evaluation*

Appendix 1: Founation of P&P algorithm

Regarding to data synchronization, if data quantity is not so big, manual upload and do full rewrite is somehow acceptable (i.e. you drop whole table and re-create and import data), but this brings two negative results: First, it will cost more hardware resource (CPU/Hard disk) to do a full update. Expensive disks need to do more write actions that dramatically reduce the server performance. Oracle achieve mode this makes additional meaningless redo logs. The second thing is that it takes time to deal with spreadsheet, for example files in different character sets, version, typo, wrong sheet layout etc, moreover, it is not a option to transfer credential data in email attachment, it takes more threat like virus and hack. Therefore, we intended to make a tool to help us:

Transfer data	Aim to Oracle/MSSQL database Do data flush (synchronize) automatically
Find data change	Find and report changed tuples, trace SQL DDL
Minimize time cost to do synchronize	As short as possible

Table 6. Some initial requirement for this tool

Normally, when does comparison in two tables, the SQL query could be looked like: "SELECT ATT1 FROM TABLE1 WHERE NOT EXIST IN SELECT ATT1 FROM TABLE2". Without any optimize, it will do a cross comparison, and the database will query for "count(*) from table1 times count(*) from table2" times. Consider if we work with a million level data source, this kind of query will consume all the hardware resources.
The key idea to expedite the process is to make logical comparison to "physical" level comparison. One possible solution is to move all of the data in physical memory units where to do the comparison. It is much faster than do the same process in virtual memory – for example pages on hard disks. Moreover, fortunately, the method of Oracle database storage its data give us one possible solution to shorten the time cost is to change cross comparison to parallel comparison.

Currently, Rowid has four segments which can be used to divide and group tuples. The first two segments are "Data Object ID" and "Data File ID" which won't be changed from time to time, if no DBA actions happend. The third segment is "Data Block ID" which indicates the location of this block which contains this particular tuple in the data file. "Data Block ID" is relative to their data file, not their tablespace. Thus, two rows with identical block numbers could reside in different data files of the same tablespace. The last segment is a serial number to indentify the row in the Data Block.
Some important assumptions here:

1. ORACLE won't change its ROWID meaning and data storage method in the future
2. Entity attributes can't change – otherwise we will do a full duplication (drop destination table and do a full copy)
3. So far it doesn't support blob or other binary data type – it is possible to implement but this need to redesign the engine's data structure.
4. We won't do database level operations to change data blocks. For example to shirk table, truncate table, actions for partition table change by DBA **are not allowed**. Any kind of this operation shall be treated as critical change in a production environment and it needs a comprehensive change plan and change window to perform. Only SQL level modify which trigger a SQL to be executed in original data source is allowed

Functions	Regular cross comparison by execute SQL	Algorithm based on Oracle ROWID
Do full copy	NORMAL	NORMAL
Bulk data add / remove	SLOW	FASTER
Single data add / remove	Very slow	FASTER
Check data content change	Very slow	FASTER
Generate change DDL	Not available	YES, can choose execute or not execute

Table 6. Efficiency of comparisons by logical SQL query and ROWID based SQL query

During a ROWID comparison, the first step is to divide tuples in groups, divided by Block Number. Then check Block Number to find bulk change first. If find a difference, do a bulk add/remove for that Block Number regardless to check each tuples inside it. For example, some new tuples are created in the original data source, and then they were located in a new Data Block area by the database process. So a new block number should be detected but not be found in destination data source. Therefore a bunch of simple INSERT SQL will be generated.
If found the same Block Number, do a parallel compare in short of range of tuples. So it just needs to do this comparison in the Block Number scope, which is much faster than do the comparison in the whole data source.

Process to do comparison
1. Do parallel compare for B1 and B1'
2. Identify B2 tuples were created
3. Do parallel compare for B3 and B3'
4. Identify B5' tuples were removed

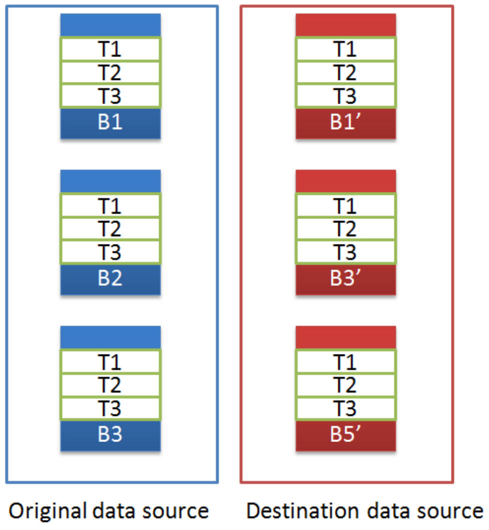


Figure 7. Process of comparison

Appendix 2: Reference of Oracle ROWID

An extended rowid is displayed in a four-piece format, **OOOOOOFFFBBBBBBRRR**, with the format divided into the following components:

OOOOOO
The data object number identifies the segment. A data object number is assigned to every database segment. Schema objects in the same segment, such as a table cluster, have the same data object number.
FFF
The tablespace-relative data file number identifies the data file that contains the row.
BBBBBB
The data block number identifies the block that contains the row. Block numbers are relative to their data file, not their tablespace. Thus, two rows with identical block numbers could reside in different data files of the same tablespace.
RRR
The row number identifies the row in the block.

Appendix 3: Check the content of each tuple

To check tuple contents, it is very difficult do write SQL or storage procedure to check each attribute (column) of the entity.

One simple and brutal solution is to make a composite string includes all the attributes in every single tuple. For example, here is a tuple as a mark record for Tom in his physics course:
STID = '2003061302', FNAME = 'Tom', YEAR = '2004-2005', COURSE = 'Physics’, MARK = 'A'
To make the tuple into a string, most simply is to make them together, so it gets “2003061302Tom2004-2005PhysicsA”. So this composited string is also unique.

Consider if the string is very long, it is not a good idea to do a string comparison because this needs more memory and CPU time to handle. One good method is to get it MD5 hash value of the string which is unique for every single string. Therefore fewer resources are required to do an integer comparison rather than a long string.