BANK RECONCILIATION STATEMENT VALIDATION USING QUADRATICALLY PROBED HASHING

A Seminar Report

Submitted By

ADITYA

(1605004)

In partial fulfilment for the award of the degree of

BACHELOR IN TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

At

SCHOOL OF COMPUTER ENGINEERING

KIIT Deemed to be University

BHUBANESWAR

MARCH, 2020

School of Computer Engineering

KIIT Deemed to be University, Bhubaneswar

Odisha-751024, India.

Certificate

This is to certify that the **SEMINAR REPORT** entitled **Bank Reconciliation Statement Validation Using Quadratically Probed Hashing**, is a bona fide work done by Aditya (1605004) in partial fulfilment for the requirement for the award of the degree of Bachelor of **Computer Science and Engineering**.

Mr. Dev Kant

Seminar Supervisor

Seminar Report

1. Abstract

In this day and age, the manual work by accounting and book-keeping are being managed by the automating softwares like RPA, etc. But still there exists a ton of companies where the employees do this task manually as a part of their job description. Such an incident was rather intriguing. This seminar report is entitled **BRS Validation Using Quadratically Probed Hashing** (or ***BRSV***), themes the enterprise software designed by the author for insurance companies which do not have an automated solution for the BRS Validation, except for doing it manually. Let's elucidate the problem and assume that there exists a certain insurance company **XYZ** whose insurances policies are sold to ***N*** aged policy-holders by ***M*** insurance agents. Let's say over a period of one month these ***N*** policy-holders are entitled to submit ₹ β_1, β_2, β_3... β_N respectively as premiums to their corresponding insurance-agents. These people give their policy-premiums as either cash or cheques, which are later deposited on the **cash-counters** of **XYZ** Company. Cash is managed properly, but the problem occurs when the payments are in **cheque-form**. When these ***M*** insurance–agents submit these cheques on **cash-counters**, the cheques are stamped and send off to the bank where the bank encashes those cheques into the account of Company **XYZ**. The bank might not encash those cheques directly, but might do those in bundles. So, let's say three cheques C_1, C_2 and C_3 are submitted as premiums of amounts **₹ 1000**, **₹ 3000** and **₹ 4520**, totalling to **₹ 8520**. Since the bank processes huge amounts of cheques at singular, it might either encash these as C_1, C_2 and C_3, or might encash C_1, C_8 and C_4 of amounts **₹ 1000**, **₹ 2500** and **₹ 3529**, amounting to **₹ 7529** only. The bank although does know about this error but since they have automated systems for encashing and reprocessing of cheques, they prepare a BRS in Excel and hand it over to the employees of **XYZ** Company where they **spend first week of every month** to match more **170k+ records** between **4 different spreadsheets** manually deleting them when a match is found and **parallel calculations**. This tedious task was semi-automated by **BRSV** which implements **Quadratic Probed Hashed** searching, being load-factor 0.8 behind **ReactJS-Flask Client-Server** application setup.

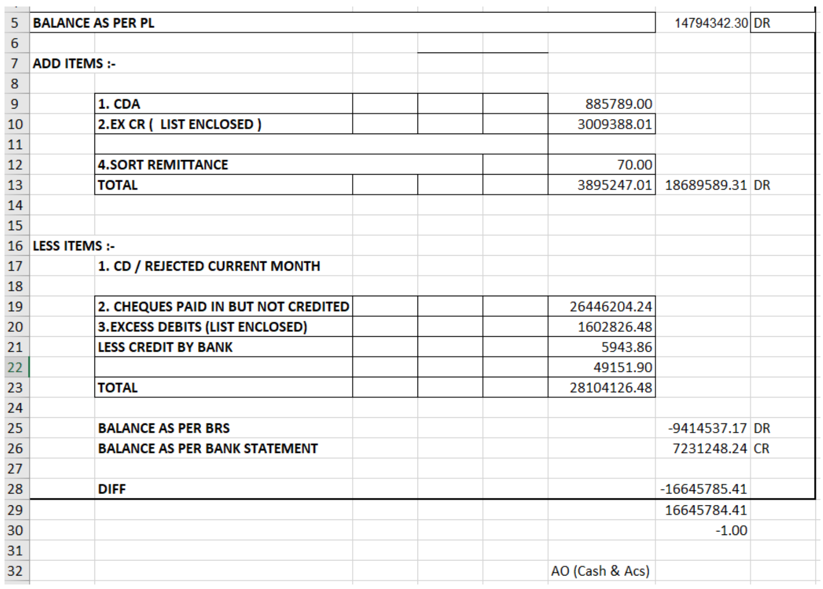
2. Table of Contents

3. Introduction of BRSV

This section explains what actually BRSV does in its core.

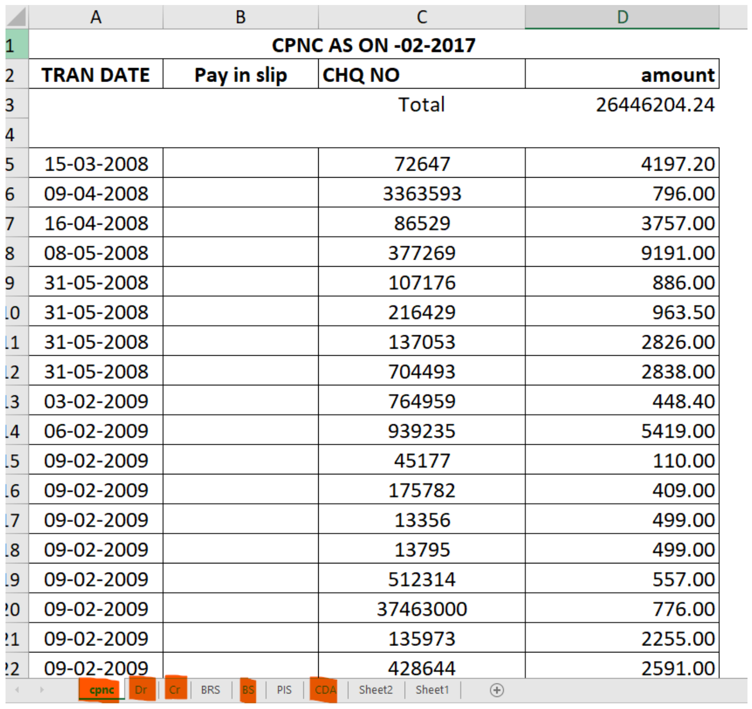
Let’s start by explaining what BRS or Bank Reconciliation Statement is.

The BRS sheets in its core looks somewhat like this:



5	BALANCE AS PER PL			14794342.30	DR
6					
7	ADD ITEMS :-				
8					
9		1. CDA	885789.00		
10		2.EX CR (LIST ENCLOSED)	3009388.01		
11					
12		4.SORT REMITTANCE	70.00		
13		TOTAL	3895247.01	18689589.31	DR
14					
15					
16	LESS ITEMS :-				
17		1. CD / REJECTED CURRENT MONTH			
18					
19		2. CHEQUES PAID IN BUT NOT CREDITED	26446204.24		
20		3.EXCESS DEBITS (LIST ENCLOSED)	1602826.48		
21		LESS CREDIT BY BANK	5943.86		
22			49151.90		
23		TOTAL	28104126.48		
24					
25		BALANCE AS PER BRS		-9414537.17	DR
26		BALANCE AS PER BANK STATEMENT		7231248.24	CR
27					
28		DIFF		-16645785.41	
29				16645784.41	
30				-1.00	
31					
32			AO (Cash & Acs)		

Fig 3.1 BRS calculation sheet



A	B	C	D
	CPNC AS ON -02-2017		
TRAN DATE	Pay in slip	CHQ NO	amount
		Total	26446204.24
15-03-2008		72647	4197.20
09-04-2008		3363593	796.00
16-04-2008		86529	3757.00
08-05-2008		377269	9191.00
31-05-2008		107176	886.00
31-05-2008		216429	963.50
31-05-2008		137053	2826.00
31-05-2008		704493	2838.00
03-02-2009		764959	448.40
06-02-2009		939235	5419.00
09-02-2009		45177	110.00
09-02-2009		175782	409.00
09-02-2009		13356	499.00
09-02-2009		13795	499.00
09-02-2009		512314	557.00
09-02-2009		37463000	776.00
09-02-2009		135973	2255.00
09-02-2009		428644	2591.00

Fig 3.2 CPNC sheet (The highlighted tabs are the 4 other sheets similar to this one, all totalling to five)

The match flow is something like this:

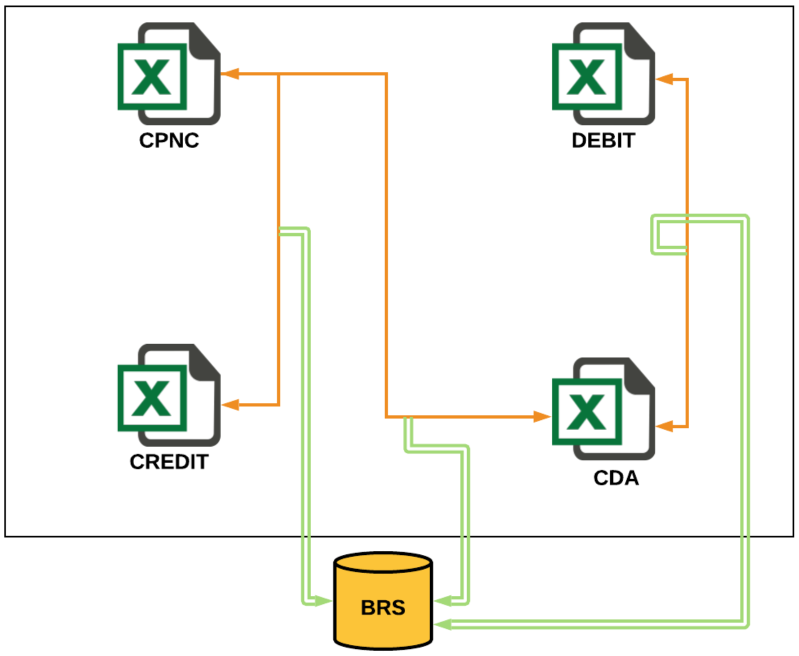


Fig 3.3 Match flow of the BRS Validation procedure

As it is visible from the diagram, there are four Excel sheets in BRS –

I. Cheques Paid but Not Credited (CPNC)
II. Credits
III. Debits
IV. Cheques Dishonour Action (CDA)

The match-flow transaction works in the following ways:

(i) The transactions in CPNC sheet is matched in Credit sheet as well as CDA sheet.
(ii) The transactions in CDA sheet is matched in Debit sheet.
(iii) The transactions stay in their respective sheets if the match is not found.
(iv) If the match is found, the records are simultaneously removed from the matching sheets and the amount is deduced from the gross amount in BRS such that it gets tallied in the end.

The software displays its frontend and makes up data frames by inputting the various columns of the excel sheet. These data-frames is sent to the Flask api-server for processing. After the processing of the data-frames occurs, fresh Excel sheets are created under the names of the respective sheets. Then a spatial vector graph is generated to represent the undone/baseless transactions.

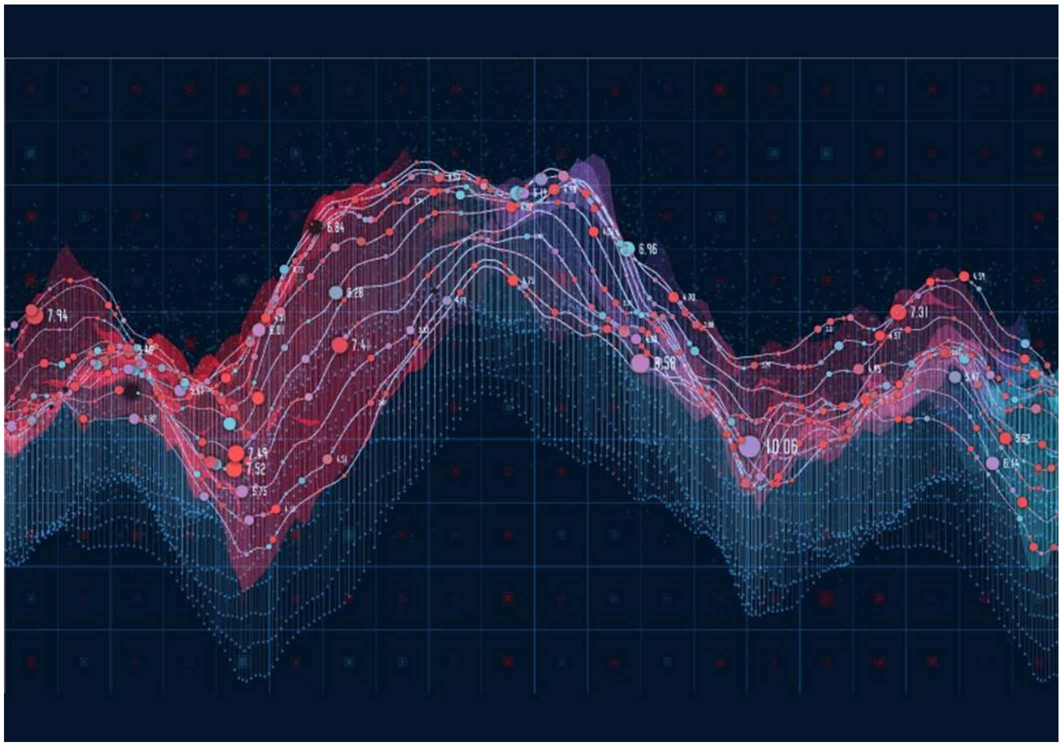


Fig 3.3 Example of a spatial vector visualization generated (Post-Processing)

4. Detailed Review

Quadratic Probing and Hashing – Definition and Reason of Implementation

The key idea behind building BRSV is fastening the access of records and making searching as fast as possible.

Possibilities could be:

(i) The first and the foremost choice could be implementing as a sorted array. Implementing this easy but the insertion and searching takes O(log(n)).

(ii) We could similarly implement a linked list, but this worsens the time complexity by O(n).

(iii) Next, could be implementing a Balanced Binary Search Tree, but the time complexity for insertion and searching does not improve, still being O(log(n))..

(iv) A drastic solution can be Direct Access Table implementation, but the time complexity for insertion and searching becomes O(1), but here there is a limitation for the size of the table which if exceeds $m*10^n$ the time complexity starts to resemble O(log(n) [1].

(v) An improvement on DAT can be Hash Tables, where the time complexity for all operations is O(1) [2].

That is why, Hashing became the obvious choice.

Hashing

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted [3].

Hashing is implemented in two steps:

An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

The element is stored in the hash table where it can be quickly retrieved using hashed key.

```
hash = hashfunc(key)

index = hash % array_size
```

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using modulo operator (%).

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes [4].

To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

(i) Easy to compute: It should be easy to compute and must not become an algorithm in itself.

(ii) Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.

(iii) Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques [5].

There are different ways to resolve collisions while making a hash table:

(i) Separate Chaining: In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list [6].

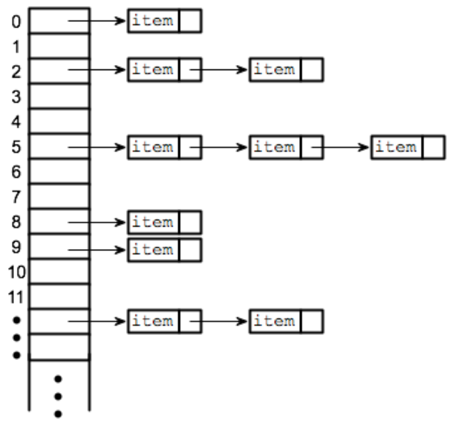


Fig 4.1 Example of a Separate Chaining

But cache performance is not so good and there is wastage of space as well, and if we consider

$$\alpha = \frac{n(number\ of\ transactions)}{m(number\ of\ table\ slots)}$$

So the time complexity here comes to be $O\left({}^{1}/_{(1+\alpha)}\right)$

(ii) If we can quadratic probing, we can improve this to $O\left({}^{1}/_{(1-\alpha)}\right)$ where the Hashing formula is

$$h_i = (hash(x) + i^2)\%\ HashTableSize(\psi)$$

Since, with the variation of load factor α (<1) the quadratic probing yield better results of 1.12 [7].

Hence, <u>hashing was selected for implementation with quadratic probing for insertion and searching</u>.

Software Workflow

This section explains how Bank Reconciliation Statement Validation System works as an enterprise software as well as how the updates are pushed.

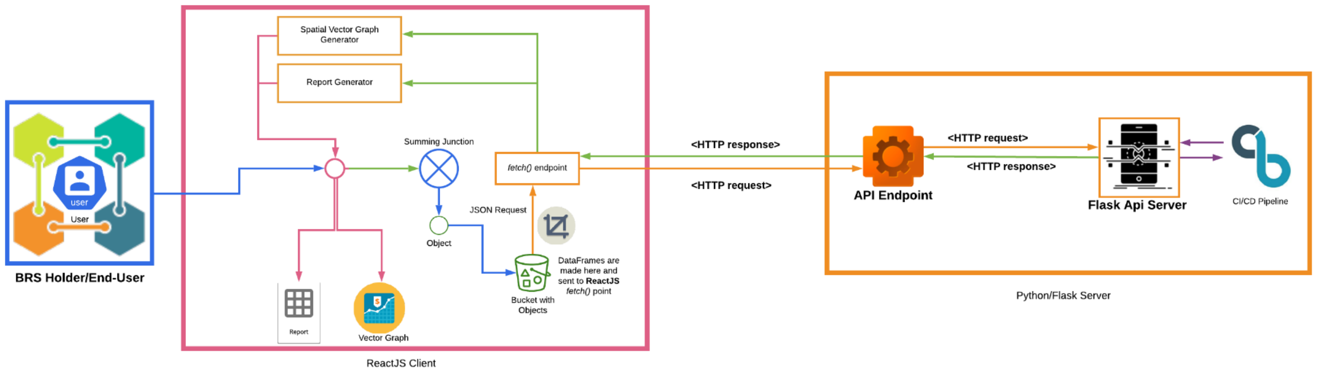


Fig 4.1 Work Flow of BRSV

(i) The user who is the holder of the BRS compilation carefully inputs the three columns of the all four sheets one by one, to obtain four individual data frames on **npm** server on the defined **summing junction**.

(ii) The individual data frames are the compiled into one **bucket**, and then transformed into a **JSON request** at ***fetch ()*** request endpoint. This frontend server now sends out an HTTP request to the **Flask API Engine** at **server endpoint**.

(iii) The server loads up data and starts hashing the records on data frames onto their respective **hash tables**.

(iv) User then select all the required match-ups, enters the matching restrictions and then triggers the matching. The search happens very quickly the entire matching is done by single iteration of the matching hash tables of both participating members in the match-up.

(v) The remaining data-frames are then written on fresh excel files under their corresponding sheet names and the same along with a report object is sent back to the frontend where the report and spatial vector graph is generated.

(vi) For pushing updates, the software runs an **internal CI pipeline** to install rolling updates from **GitHub**, where it periodically fetches or clones the repository from the **repository upstream**.

This sums up the working of the **BRSV system**.

5. Summary

All in all, this seminar report described how the enterprise software was implemented. Also, this report covers the evidence supporting how a basic Data Structure Fundamental topic solves a very tedious manual task which a whole bunch of people spending almost an entire week of time into quick and easy job of bare 5-7 minutes(depending on the amount of transactions in data sheets). Although this software was deployed on premise, yet there could be further improvements like **Amortized Hashing using Doubled Probing** could be explored for same solution. Further, as rolling updates for bugs and enhancements, the server could be made into a **gRPC platform** and similar enhancements.

6. References

[1] B. V. Rompay, "Analysis and Design of Cryptographic Hash functions, MAC algorithms and Block Ciphers", Ph.D. thesis, Electrical Engineering Department, Katholieke Universiteit, Leuven, Belgium, 2004.

[2] FIPS 180, Secure Hash Standard (SHS), National Institute of Standards and Technology, US Department of Commerce, Washington D. C., 1993.

[3] FIPS 180-1, Secure Hash Standard (SHS), National Institute of Standards and Technology, US Department of Commerce, Washington D. C.,1995.

[4] FIPS 180-2, Secure Hash Standard (SHS), National Institute of Standards and Technology, US Department of Commerce, Washington D. C., 2002.

[5] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton, "Seven-Properties-Preserving Iterated Hashing: The RMC Construction", ECRYPT document STVL4-KUL15-RMC-1.0, private communications, 2006.

[6] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton, "Seven-Property-Preserving Iterated Hashing: ROX", IACR Cryptology ePrint Archive, 2007, pp.176.

[7] M. Bellare, and T. Ristenpart, "Multi-Property-Preserving Hash Domain Extension and the EMD Transform", in ASIACRYPT, 2006, pp.299-314