PERFORMANCE EVALUATION AND EXTENSION OF CACHEJOIN IN A REAL-LIFE ENVIRONMENT

SOLOMON GEORGE GEORGEWASHINGTON

A thesis submitted to

Auckland University of Technology

In partial fulfilment of the requirements for the degree

Of

Master of Computer and Information Sciences (MCIS)

29th July 2015

School of Computer and Mathematical Sciences

Primary Supervisor: Dr Muhammad Asif Naeem

Secondary Supervisor: Shoba Tegginmath

Additional Supervisor: Dr Gerald Weber

Table of Contents

List of Figures

List of Tables

List of Algorithms

Attestation of Authorship

I hereby declare that this thesis submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Solomon George Georgewashington

Acknowledgements

“For God so loved the world that he gave his only begotten Son (Jesus Christ), that whosoever believeth in him should not perish, but have everlasting life.” – John 3:16.

Praise to my God Almighty, Jesus Christ, who gave me a wisdom, knowledge and the opportunity to receive this level of study in my life time, which I did not even think possible for me. All the honor, praise and glory are His alone. By His grace only I am able to attain this degree.

I dedicate this master degree to my dear, most loving Amma, P. Paul Rose who always longed to see me with this level of qualification. I also dedicate it to my most deeply beloved wife, Shiny Solomon, who has been with me at every step in my days. I would like to remember my sons, David Philip S. George and Daniel Jeshur S. George at this time, who have made my days of study more challenging and fun.

I wish to express my sincere gratitude to my supervisors: Dr. Muhammad Asif Naeem and Dr. Gerald Weber, who have abundantly provided me with invaluable assistance and guidance. They have supported me enormously to improve my research and thesis writing by guiding me with their combined knowledge and constructive criticism.

I would like to convey my sincere thanks to the MITRE 10 NZ, Support Centre, for providing me with both a scholarship and the opportunity to work with their dataset. My sincere thanks to my managers, Louise Acfield and Alex Augoustides, for encouraging me during this time.

Also, I would like to thank Catriona Carruthers for proof reading my thesis and suggesting necessary corrections.

Abstract

Active or real-time data warehousing is becoming very popular in business intelligence domain. In order to build a real-time or active data warehouse an online processing of stream of end users' transaction with disk-based master data is required. This is also called processing of semi-stream data. Fundamentally, this semi-stream processing is a process of joining an incoming stream data (transactional data) with the disk-based slow retrieving master data by using an effective join operator. Typically this join operator works with a limited amount of main memory which cannot hold the entire disk-based master data. Recently a number of semi-stream join algorithms have been proposed in the literature. Most of these algorithms have been tested using synthetic dataset while only a few using real-life dataset. It is always interesting to see how these algorithms behave in real environment. As each semi-stream join performs differently under the different characteristics of the stream data, it is important to select appropriate semi-stream join based on the characteristics of the stream data. Also these join algorithms use different strategies to access the disk-based master data e.g. index (clustered index or non-clustered index) or no index.

Based on an intensive literature review, in this thesis we select a well-known semi-stream join CACHEJOIN (Cache Join) and implement it in MITRE 10 NZ, one of the leading home improvement and hardware retail store. We study the behavior of the algorithm under two different datasets (synthetic dataset and MITRE 10 NZ dataset). We study the performance of the algorithm under both datasets. Our performance study shows that under MITRE 10 NZ dataset CACHEJOIN performs very closer to that of synthetic dataset.

As an extension of our work we find that MITRE 10 NZ incoming stream data (transactional data) needs to join with two tables in disk-based master data. First join is performed with product table (*sc*) using *stock_code* as a join attribute. While second join is performed with customer table (*cs_person*) using *account_code* as a join attribute. This gives us an opportunity to extend our existing CACHEJOIN for two-stage join. The stream tuples move to the second stage as soon as they complete the first stage. The performance of two-stage join is studied against normal CACHEJOIN using MITRE 10 NZ dataset. After analyzing the performance we are confident that extended CACHEJOIN performs reasonably well for MITRE 10 NZ real environment.

As a future work, we have a plan to explore more in two-stage join by trying different semi-stream joins and find out the best join combinations, and also explore more on parallelization of running 2 parallel nodes to handle the future growth of MITRE 10 NZ transactional data.

Chapter 1

1 Introduction

1.1 Real-time data warehousing

Near real-time or active data warehousing is becoming more and more emerging area of research due to demand of real-time business organizations. These real-time data warehouses are now required to fulfill the business organizations needs by providing them most up-to-date information about their businesses, e.g. their sales, their stock status, etc. This requires the end user's data being updated immediately in a data warehouse as it is received in operational data sources. Providing the latest information to business users will help business leaders in making the right decisions at the right time. Once active data warehouses are built, business intelligence tools then use these active data warehouses to provide real-time reporting in order to support businesses. Business intelligence software's are basically a collection of a number of decision support technologies that are aimed at providing information to senior managers, members of the board, managers, analysts, etc. (Surajit, Umeshwar & Vivek, 2011).

Traditional data warehouses do not have continuous update capability and normally these type of data warehouses are only updated once a day when there is little processing happening in the background. In this type of traditional data warehouses, tuples are buffered and joined based on the availability of resources (Annita & Peter, 1990), (Leonard & Shapiro, 1986). The downside of traditional data warehouses is that they are not up-to-date due to the lack of continuous update to the data warehouse.

To solve this, the concept of a real-time data warehouse has introduced (Burleson, 2004), (Alexandros, Panos & Evaggelia, 2005), (White Paper Oracle Corp., 2003). In order to build real-time data warehouses, semi-stream join algorithms are required. Basically semi-stream join algorithms are used to join the fast incoming stream data with the slowly changing master data which is normally to be found in relational databases. Such a join can be applied in real-time data warehousing (Asif, Gillian & Gerald, 2008). Extensive study has been undertaken on join algorithms since the initial days of database development. Initial works have hosted competent techniques for finite disk-based relational cases (Goetz, 1993). Of the many semi-stream join algorithms proposed, a few are selected, studied and explained in detail here.

The staging concept is used in many stream-based join algorithms to amortize expensive disk input/output costs over fast incoming stream data (Asif, Gerald, Gillian & Christof, 2013), (Abhirup & Ajit, 2009), (Asif, Gillian & Gerald, 2012), (Asif, Gillian & Gerald, 2011), (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008), (Muhammad, Gillian & Gerald, 2011), (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2007).

1.2 Motivation for the research

There have been many semi-stream join algorithms proposed in the real-time data warehousing domain. Few algorithms to name are the MESHJOIN algorithm, the R-MESHJOIN algorithm, CMESHJOIN algorithm, the HYBRIDJOIN algorithm, the CACHEJOIN algorithm, etc. which most of the algorithms have been explained in detail with data structural diagrams and with experimental results. Algorithms have been experimented on with synthetic datasets taken from authorized or well-known sources such as TCP-H datasets. However it is perhaps even more interesting to know that which semi-stream join would be suitable to a specific industry. Every industry will have different types of user/customer data. For example, the retail industry would have long-tailed distribution patterns in incoming stream data, whereas a weather forecasting dataset perhaps would have different characteristics. This provided the motivation to select an industry and an appropriate semi-stream join suited to that industry's dataset. Therefore there is a need to study these semi-stream join algorithms and to observe their performance using synthetic and real-life datasets. There is also an opportunity to extend the most suitable semi-stream join by considering the nature of processing of incoming stream data with disk-based master data in real environment. To accomplish our research, we select MITRE 10 NZ. MITRE 10 NZ is a local owner-operated chain consisting of nearly 150 hardware and home improvement stores in New Zealand.

1.3 Research Questions

The aim of this thesis is to address the following two research questions.

a) ***How does an appropriate semi-stream join perform differently on a real-life dataset when compared to a synthetic dataset?***

There are many semi-stream joins proposed with cost models and empirical studies. In most cases, semi-stream joins are tested using synthetic datasets while only a few using real-life datasets. It is always interesting to do a performance comparison study with both a synthetic dataset and a real-life dataset. Every company will have different incoming stream data patterns for example, retail industry stream data pattern would be different to a supply chain data pattern. This research had the opportunity to implement and test a well-known semi-stream join algorithm CACHEJOIN using a dataset produced by MITRE 10 NZ which is one of the leading home improvement and hardware retail companies in New Zealand. We conduct a wide literature review to identify a suitable semi-stream join for the given organization. We compare the performance of the algorithm under both synthetic dataset and the MITRE 10 NZ dataset and analyze how the join behaves with the different datasets. We expect to see a different performance behavior for the both datasets due to their different characteristics.

b) ***In MITRE 10 NZ the incoming transactional stream data need to join with two different master data tables, the question is how we can extend the normal CACHEJOIN algorithm to implement this two-stage join scenario?.***

Currently MITRE 10 NZ transactional data needs to join with two different master data tables product table and customer table. The research question here is how we can extend CACHEJOIN algorithm to implement this two-stage join scenario? In this thesis we extend the existing CACHEJOIN algorithm for the second stage join. Also we compare the performance of our extended two-stage join with the normal CACHEJOIN using MITRE 10 NZ transactional data.

1.4 Approach

A wide literature review is done in the semi-stream joins that uses different methodologies for joining the steam data with the master data. By understanding the workings, advantages and disadvantages of the semi-stream join algorithms an appropriate semi-stream join that suites to MITRE 10 NZ was chosen, as found in Section 2.2. The characteristics of the MITRE 10 NZ transactional data is studied and considered before selecting the algorithm for this implementation. The selected semi-stream join was implemented on synthetic and real-life datasets, and a performance study on service rate was done with both result sets. Next, an extension to the CACHEJOIN algorithm to handle a two-stage join was explored using the MITRE 10 NZ transactional tuples. In this extension, the first stage is normal CACHEJOIN algorithm which joins *stock_code* with product master table and the second stage joins *account_code* with customer master table which is in cache. Once this extension was coded, a performance evaluation was carried out on the semi-stream join and the two-stage semi-stream join using MITRE 10 NZ datasets. And the cost comparison on calculated analytical cost and empirical cost is done as a validation of our cost model.

1.5 Structure of the thesis

The thesis is structured as follows:

Chapter 1, gives the brief introduction about the near real-time data warehousing particularly the stream processing in the near real-time data warehousing. An introduction to the semi-stream joins and join algorithms are explained with listing few semi-stream join algorithms. It explains the reason and benefits for business to build an active or near real-time data warehouse. A small scenario explained why and how a real-time data warehouse helps a business attain good customer service. The motivation for this thesis, research questions and approach were also explained.

Chapter 2, provides a detailed literature review on semi-stream join algorithms. We consider a few well-known semi-stream joins and explain their working, advantages, and disadvantages. We also present their data structures and architectural design.

Chapter 3 presents the reasons for selecting the CACHEJOIN algorithm to implement on the MITRE 10 NZ transactional dataset. The chapter also presents CACHEJOIN in detail including its pseudo-code and data structure. Following the detailed explanation of the CACHEJOIN algorithm, the chapter presents an experimental setup including both synthetic and real-life datasets. Finally, the chapter describes the experimental results produced by the CACHEJOIN algorithm under the both datasets.

Chapter 4 presents our extension of the CACHEJOIN which is two-stage join. It includes the motivation behind this extension. The chapter presents the implementation and the pseudo-code for our two-stage. Finally, the performance of our two-stage join is compared with the normal CACHEJOIN algorithm.

Chapter 5 presents conclusions of this research and describes some future directions.

Chapter 2

2 Literature review

2.1 Introduction

To have a real-time data warehouse we need to have an effective semi-stream join operator which joins the fast coming stream tuple with the master data using limited resource in an efficient way. We did an extensive literature review on different semi-stream algorithms for this research work. Semi-stream join algorithms that uses different methodologies like the non-indexing method, the indexing method and the caching method for joining the stream tuples with slow moving relational master data are selected. These semi-stream join algorithms are studied in detail to understand their working, advantages and disadvantages and given below.

2.2 Existing Semi-Stream Joins

Figure. 2.1 shows different semi-stream joins studied in this thesis. For the simplicity we classify them into three types, non-index-based joins, index-based joins and cache-based joins.

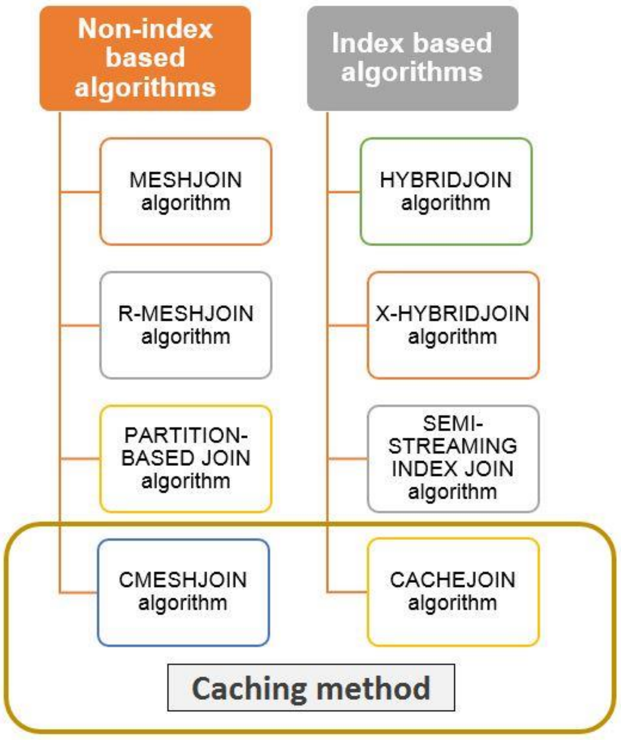


Figure 2.1 Classification of Semi-Stream Join algorithms.

2.2.1 Non-Index based Semi-Stream Joins

2.2.1.1 MESHJOIN

MESHJOIN was introduced to join a fast stream of data, S, of source updates, with a large disk-based relational master data, *R*. It was specifically designed to join a continuous stream data *S* with a slow moving disk-based master data *R* as in the scenario in an active or near real-time data warehouse (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2007). It performs a non-paused execution of the hash table which is built to load stream data more progressively.

(Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008) proposed the algorithm, which relies on two basic techniques to increase the efficiency of the necessary disk accesses. Firstly, it accesses the master data *R* solely through fast sequential scans and, secondly, it amortizes the cost of disk *I/O* operations over a large number of stream tuples. Figure. 2.2 shows a graphical depiction of this technique and illustrates the main data structures used in this algorithm. The mechanics of this diagram is, two inputs are accessed continuously and meshed together to generate the output of joining stream data *S* and disk-based relational master data from data warehouse *R* (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008). To be more specific, this performs a cyclic scan of relational data *R* and joins its tuple over the stream data *S*. The main idea is that the stream tuple enters the window when it arrives and it is expired from the window once it is probed with every tuple in master data *R*. As is shown in Figure. 2.2 it performs a continuous scan of relational data *R* with an input buffer of *b* pages. On other hand stream data *S* is accessed in batches of ω tuples that are inserted into the contents of the sliding window. When any tuple is inserted it causes the removal of the 'oldest' ω tuples from the window. To find the matching stream tuples more efficiently on each *R*-tuple, this algorithm synchronously maintains a hash table *H* in memory for the stream tuple based on their join key. Lastly the queue *Q* contains pointers to the tuples in hash table *H* and basically records the arrival order of the batches in the current window. This is used to remove the oldest ω tuples from the hash table *H* once they are expired from the window.

This algorithm is proposed making no assumptions about the physical characteristics of the stored relational disk-based master data *R* such as the existence of index or clustering properties, except that it is too large to fit in the main memory. Hence the developed solution

(MESHJOIN) is applicable in a wide range of settings. This opens an interesting venue for future work as it also opens the possibility of designing more effective join operators that takes those particular physical characteristics of disk-based relational master data *R* into account. (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008) gives the detailed experimental results of this algorithm which is a novel join operator that works under minimal assumptions about the stream data *S* and the relational master slow changing data *R*. Experiments were done with a real-life weather sensor data dataset which measures different parts of the globe (Carole, Stephen & Julius, 1996). This algorithm authors reported that it performs worse with skewed data. The performance of MESHJOIN algorithm is inversely proportional to the size of the disk-based relational master data *R*.

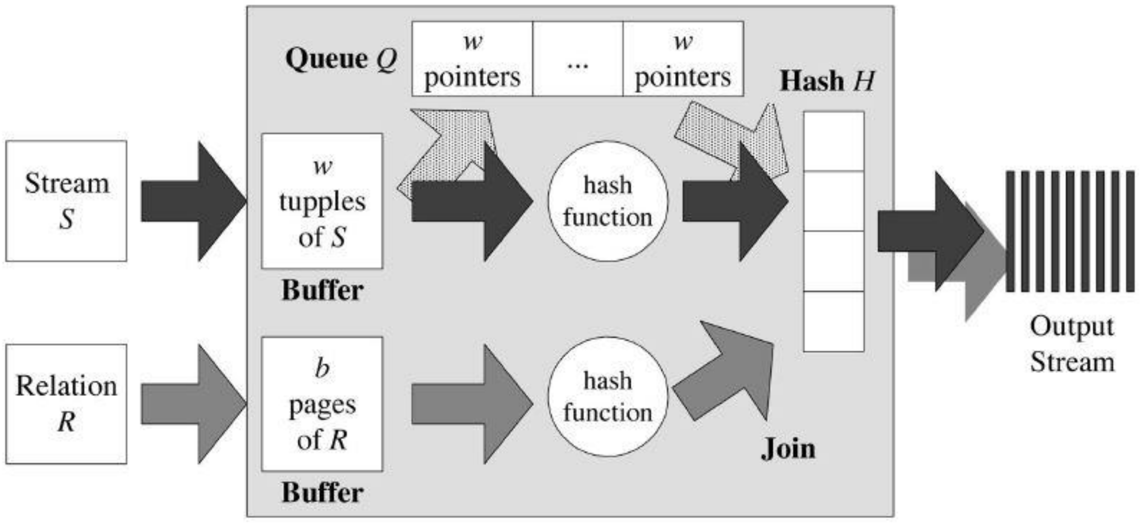


Figure 2.2 Data structure and architecture of MESHJOIN algorithm (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008)

2.2.1.2 R- MESHJOIN

(Asif, Gillian, Gerald & Shafiq, 2010) proposed a new improved version of the MESHJOIN algorithm called Reduced MESHJOIN (R-MESHJOIN). The MESHJOIN algorithm has a dependency between the partition size in an internal queue for incoming stream data, and the required iterations to bring the disk-based relational master data into the memory. This dependency hampers optimal memory distribution within the join components. The newly proposed, improved version of the MESHJOIN algorithm removes this dependency which enables it to distribute the available memory optimally within the join components. In R-MESHJOIN, the size of disk-buffer is not affected if the size of the disk-based relation is changed. An experimental study was conducted and the argument was validated (Asif, Gillian, Gerald & Shafiq, 2010). The study proved that R-MESHJOIN does slightly improve MESHJOIN and moreover helped to analyze the MESHJOIN algorithm theoretically and experimentally. The architecture of the R-MESHJOIN algorithm is shown in Figure. 2.3.

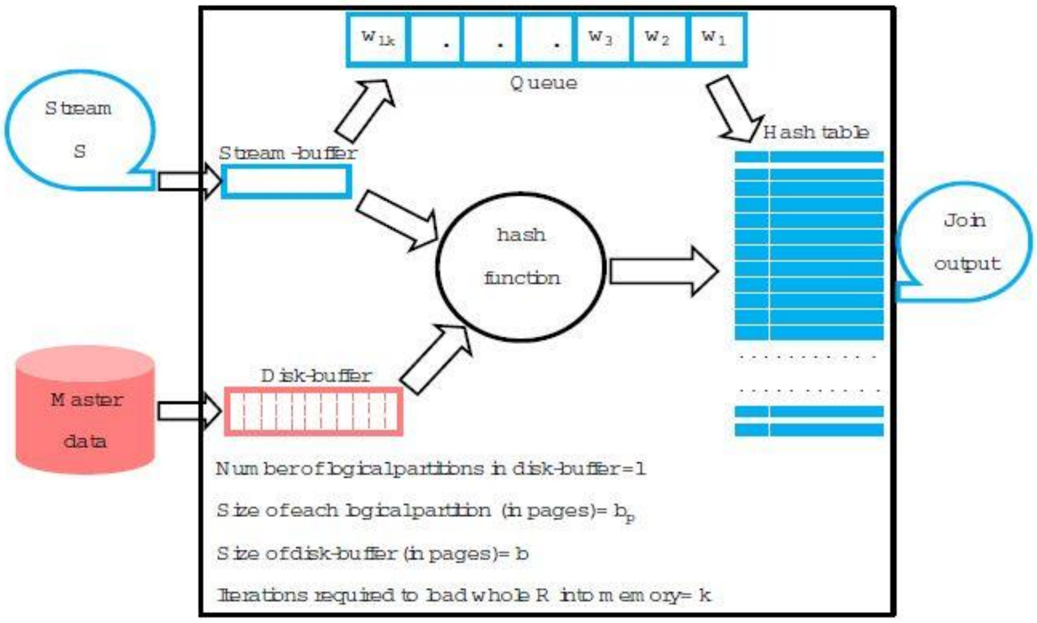

Stream S
Stream -buffer
Queue
w_{1k} . . . w_3 w_2 w_1
Hash table
hash function
Join output
Master data
Disk-buffer
Number of logical partitions in disk-buffer=1
Size of each logical partition (in pages)= b_p
Size of disk-buffer (in pages)= b
Iterations required to load whole R into memory= k

Figure 2.3 Data structure and architecture of R-MESHJOIN algorithm (Asif, Gillian, Gerald & Shafiq, 2010)

2.2.1.3 Partition Based Join

(Abhirup & Ajit, 2009) proposed a new algorithm called the partition-based semi-stream join algorithm which minimizes disk overhead, processing overhead and delays in output tuples. The previous sections outlined a novel state-of-art semi-stream join algorithm called MESHJOIN (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008), (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2007) which is the pioneer join in semi-stream join operators. Though MESHJOIN works better in a few scenarios, it has some limitations. The algorithm performs less well with a skewed data arrival pattern and the performance is inversely proportional to the size of the disk-based master data. Based on MESHJOIN performance and limitations authors (Abhirup & Ajit, 2009) proposed a new partition-based semi-stream join to join fastpaced incoming stream with determined relational data. This partition-based algorithm identifies the fast incoming stream tuple locality by joining often repeated incoming stream tuples inside the memory partitions. This is the difference between MESHJOIN and this newly proposed partition-join algorithm.

Unlike the MESHJOIN algorithm, the majority of incoming stream tuples are joined in a single disk read by which average processing time will be less. Disk access frequency is decreased by maintaining a wait-buffer which has all incoming stream tuples related to the partitions made on the disk. Once the wait-buffer is full, it invokes disk probing. When the pending tuples count corresponding to the specified partition is exceeds the invocation threshold (which is basically a user-defined limit), the disk probe is invoked. This means that the new approach does one disk read only to join the incoming stream tuple. In this partition-based join the disk-based relational table can be updated while join operations happen.

Figure. 2. 4 shows the join framework of this partition-based join. In this partition-based join, space a partitioning technique (hash-based or range-based) is applied to the disk relation in order to divide it into several segments which divides the series of joining attributes into various numbers of partitions. A cost-based caching method is applied to maintain the subset of segments in the memory which increase the in-memory service rate of incoming stream tuples. The incoming stream tuple is mapped to the respective partition in attribute space. Based on the availability of the respective disk segment in memory, the incoming stream tuple is joined with a disk segment or the stream tuple will be stored in the wait-buffer. Disk probing is invoked once the above mentioned conditions are met. Partitions are selected by the disk probe based on the order of sizes of tuples buffered in the wait-buffer and retrieves the disk segment and the buffered stream tuples are joined with the disk segment. Experimental results with cost calculation are given in (Abhirup & Ajit, 2009) and the performance appears better than the pioneer semi-stream join MESHJOIN.

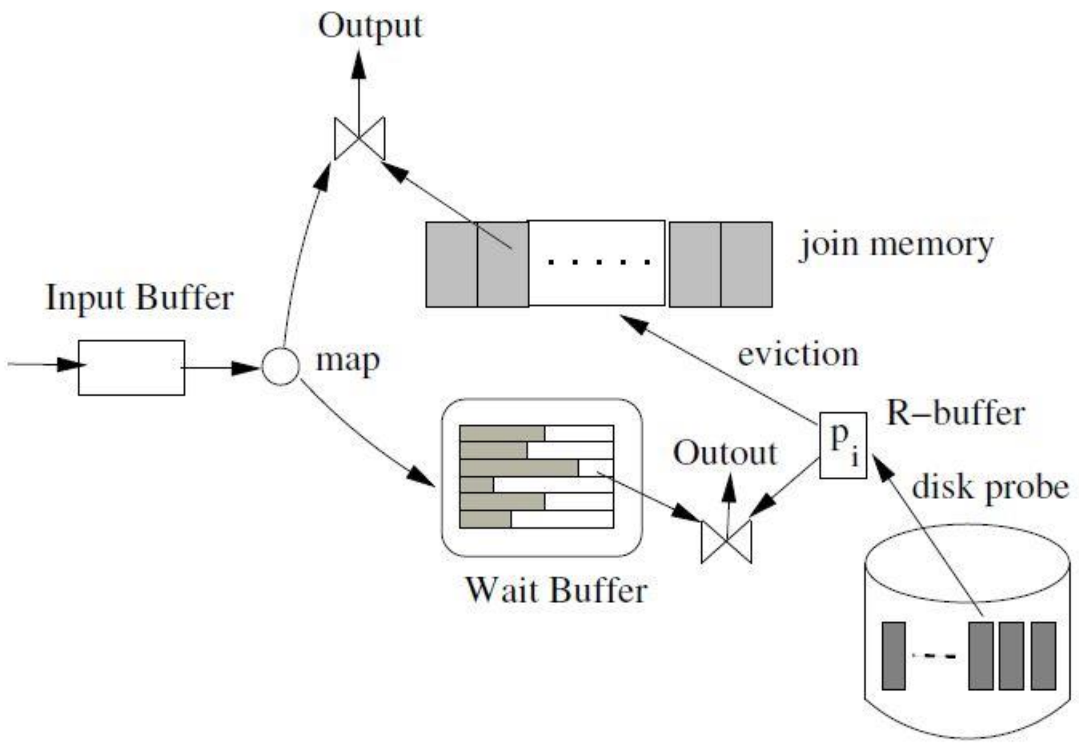


Figure 2.4 The Join Framework of the Partition-based Join (Abhirup & Ajit, 2009)

2.2.2 Index based Semi-Stream Joins

2.2.2.1 SEMI-STREAMING INDEX JOIN

The Extraction-Transformation-Loading (ETL) process plays a vital role in building efficient data warehouses. Traditionally this process is done during the business's quiet time, normally at night in batches, due to the time and resources involved. One of the most important steps in the ETL process is surrogate key replacement. This process is basically joining the tuples from each source with the metadata table which relates to the surrogate key and its related key. This process is called as *conforming* (Ralph & Joe, 2004). The traditional method of updating data warehouses in this off-line fashion, as illustrated in research and studies, is used as it enables efficient bulk loading techniques (Tom, Robert, Jim & Prakash, 1994), (Nick, Yannis & Mema, 1997) and the ETL process does not interfere with the query workload. However, in emerging applications, such as supply-chain monitoring, network monitoring, etc., we need to have efficient joins to perform the ETL process in an on-line fashion to build active data warehouses.

(Mihaela, Antonios, Yannis, Vasilis & Athens, 2011) proposed a novel algorithm called the Semi-Streaming Index Join (SSIJ) which maximizes the output of join by making use of effective index and caching frequently used pages into the memory. The main ideas of this algorithm are, the fast joining of stream tuples with matching the disk-based relational blocks, batch stream tuple processing which is batching the relational tuple required by stream tuples, batch index lookups, reading specific areas of the relational disk which are requested, more frequently requested pages are maintained in the memory so that they can easily be accessed, and adjusting the memory of the data structures dynamically. There are five components involved in the SSIJ algorithm and they are, index, cached relational blocks, input buffer, stream buffer and inverted index. Figure. 2.5 gives an overview of the Semi-Streaming Index Join.

This algorithm consists of three phases and they are the Pending phase, Online phase and Join phase. In the first phase (Pending phase) the algorithm waits for the minimum number of stream tuples in the input buffer to accommodate enough tuples to form a batch. The algorithm is moved to the next phase only when the number of required tuples is obtained in the input buffer component. This is mainly to take advantage of common access patterns and thus help in amortizing index and lookup costs. Once the tuples are filled in input buffer then the second phase kicks off.

The second phase is the online phase. In this phase the tuples are sorted based on used index characteristics. This sorting allows the algorithm to share the scans of index and of cached disk-based relational pages among several tuples. The join result is outputted immediately on all matching disk-based relational tuples that are in the cache. Any stream tuples that are not joined in this online phase needs to wait for the next phase, the join phase. This is the last phase and stream tuples will be matched in this phase when their matching disk-based relational tuple is located on the disk. Once the join phase is complete, the algorithm moves back into the first phase that is the pending phase. This Semi-Streaming Index Join is a state-of-the-art algorithm in index-based joining algorithms. The experimental results of this algorithm show that this algorithm supports very fast stream inputs and optimally exploits available memory.

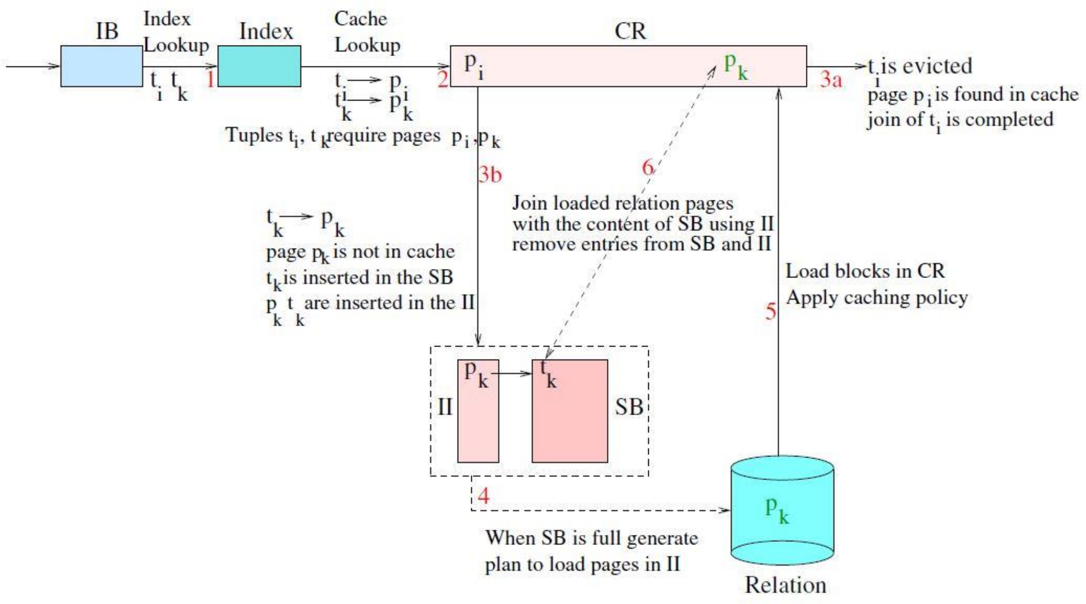

IB
Index Lookup
Index
Cache Lookup
CR
t_i t_k
1
$t_i \rightarrow p_i$
$t_k \rightarrow p_k$
2
p_i
p_k
3a
t_i is evicted
page p_i is found in cache
join of t_i is completed
Tuples t_i, t_k require pages p_i, p_k
3b
6
$t_k \rightarrow p_k$
page p_k is not in cache
t_k is inserted in the SB
p_k t_k are inserted in the II
Join loaded relation pages with the content of SB using II remove entries from SB and II
Load blocks in CR
Apply caching policy
5
II
SB
p_k
t_k
4
When SB is full generate plan to load pages in II
p_k
Relation

Figure 2.5 SSIJ Overview (Mihaela, Antonios, Yannis, Vasilis & Athens, 2011)

2.2.2.2 HYBRIDJOIN

In the section 2.2.1.1, we discussed the non-indexed semi-stream MESHJOIN. It reads the disk-based relational slowly-moving master data sequentially in partitions and then performs joining. The architecture and explanation of how it works is given in that section. This MESHJOIN algorithm successfully amortizes the fast arrival rate of the incoming stream data *S*, by executing the disk partition join with a large number of incoming stream tuples. However there are few issues found in this algorithm. Firstly, this algorithm reads unused or less used partitions of disk-based relational master data *R* by accessing it from the table sequentially, which increases the processing time of each stream tuple that is in the queue due to extra disk *I/O*. Secondly this algorithm cannot deal with burst incoming stream data effectively. If the stream input size is greater than or equal to the number of tuples in the stream buffer then disk invocation occurs. If the stream input data size has a lower arrival rate, then the existing tuples in the queue need to wait longer due to the delay in disk invocation. This waiting time also affects performance negatively. To overcome/handle these two issues, (Asif, Gillian & Gerald, 2011) proposed a new semi-stream join called the HYBRIDJOIN. Figure. 2. 6 shows the data structure and architecture of the HYBRIDJOIN algorithm. The components of HYBRIDJOIN algorithm are same as those of the MESHJOIN algorithm, which are; disk buffer, hash table, stream buffer and queue. In HYBRIDJOIN algorithm, it is assumed that relational master data

R contains sorted, unique and indexed values of join attribute. The disk buffer is used to store a portion of disk *R* data. The value of the join attribute is stored in the queue which also stores the address of its one-step neighbor nodes. There is a new feature of random deletion implemented in the HYBRIDJOIN algorithm queue which uses a doubly-linked-list. An important component in this algorithm is the hash table which stores the input stream tuples and the node addresses of the queue corresponding to the tuples. The benefit of this is the algorithm can start matching with all matching stream tuples from the queue once the disk partition is loaded into the memory using the join attribute value from the queue. This method helps to reduce the disk *I/O* cost of a fast arrival stream. Whenever a match is found, the algorithm generates an output of that tuple and then removes that node from queue and also the corresponding tuple from the hash table. Unmatched tuples are dealt with in a similar way to the MESHJOIN algorithm. Every disk input is bound to the stream input in the MESHJOIN algorithm whereas in the HYBRIDJOIN algorithm this constraint is removed by making independencies between each disk invocation from the stream input data. The cost model of this semi-stream join is explained in (Asif, Gillian & Gerald, 2011) along with a comparison analysis with the MESHJOIN algorithm and experimental results based on the Zipfian's distribution (Chris, 2006) pattern synthetic dataset. The theoretical results shows that this HYBRIDJOIN algorithm is significantly better than the MESHJOIN algorithm.

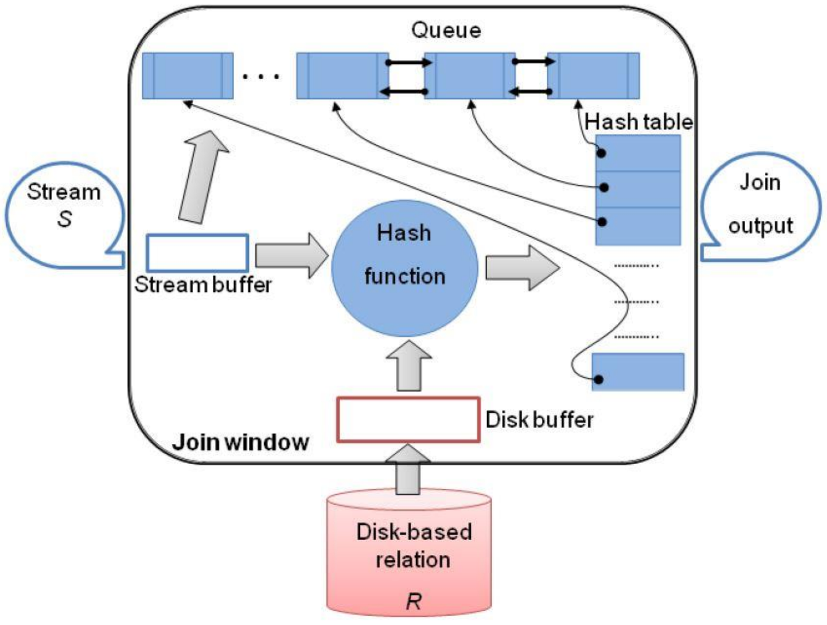


Figure 2.6 Data structure and architecture of HYBRIDJOIN algorithm (Asif, Gillian & Gerald, 2011)

2.2.2.3 X-HYBRIDJOIN

The HYBRIDJOIN algorithm join process described in section 2.2.2.2 uses an index. The MESHJOIN algorithm (section 2.2.1.1) does not take stream tuple frequency into account and does not need master data tuples to be indexed. In some circumstances this can be useful, but in many other cases in order to gain maximum performance, one obviously wants to have indexing. (Muhammad, Gillian & Gerald, 2011) proposed a new algorithm called the Extended HYBRIDJOIN algorithm (X-HYBRIDJOIN). The key feature of this algorithm is that it stores the most used portion of the master data *R*, which is disk-based relational data, which most often matches received items from stream data *S* in the memory. This reduces the disk *I/O* cost considerably and improves the performances of the join algorithm.

There are two major changes in the X-HYBRIDJOIN algorithm, when compared with the MESHJOIN algorithm. The first is that, hash join component is modified in the X-HYBRIDJOIN algorithm to make use of an index. The second is that the X-HYBRIDJOIN algorithm caches most frequently used relational disk-based master data *R*. In the HYBRIDJOIN algorithm only the first change was implemented.

Figure. 2.7 shows the working overview of X-HYBRIDJOIN. The difference between the HYBRIDJOIN and the X-HYBRIDJOIN algorithm is that the disk buffer component is divided into two parts. One is to store the most used pages of master data *R* permanently in memory called Non-Swappable in the diagram, and the other is to store the partitions of the remainder of the master data in the memory as is done in the HYBRIDJOIN algorithm. The algorithm becomes ready to be executed once available memory is distributed within the join components. The algorithm reads a particular portion of the master data *R* and loads it into the non-swappable part of the disk buffer before it starts the actual join execution. When the algorithm starts the hash table *H* slots are empty as incoming stream data *S* is to be assigned to it. Basically, the algorithm has two loops, an outer loop and an inner loop. The outer loop is an endless loop, its key role being to build the stream in the hash table. Within this loop there are two inner loops run by the algorithm. One of the inner loop performs the probing module in the non-swappable portion of the disk buffer and the other inner loop performs the probing module in the swappable portion of the disk buffer. When the outer loop is started, the algorithm observes the stream buffer status. If there is any stream found, it loads it into the hash table and also enqueue its attribute values in the queue. Now the algorithm executes the first inner loop. First inner loop reads every tuple from the non-swappable part one-by-one and performs lookup in the hash table. If there is any match found it generates an output and also deletes it from the hash table and the corresponding node in the queue. Then the algorithm increments the available vacated slots in the hash table. This is the end of the first inner loop. Before starting the second inner loop, the algorithm reads the oldest value from the queue and the swappable part of the disk buffer is loaded using the join attribute value as an index. Once this action is performed, a similar probing procedure to the first inner loop is performed here. If the first inner loop is switched-off, technically it becomes a HYBRIDJOIN algorithm. After various experiments, results shows that the X-HYBRIDJOIN algorithm performs better than other algorithms when relational master data *R* increases. Though the authors proposed a new algorithm by adding another component (the non-swappable portion of disk buffer) to the HYBRIDJOIN algorithm, they plan to improve this algorithm by tuning the X-HYBRIDJOIN algorithm to utilize the available memory resources optimally.

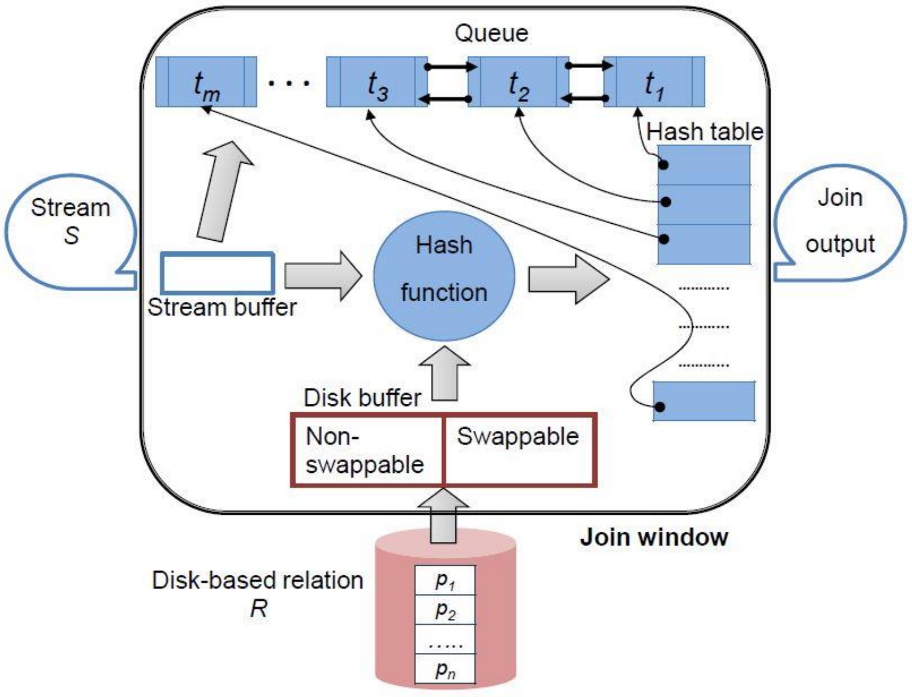


Figure 2.7 Data structure and architecture of X-HYBRIDJOIN algorithm (Muhammad, Gillian & Gerald, 2011)

The authors continued the work of improving the performance of the X-HYBRIDJOIN algorithm by better tuning the available memory resources (Muhammad, Gillian & Gerald, 2011). The existing cost model of the X-HYBRIDJOIN algorithm was revised and the join component was also tuned based on that cost model. As an outcome of this tuning, the available memory is distributed within all components properly and it has improved performance of the algorithm significantly. This new algorithm was presented by (Muhammad, Gillian, Gerald & Imran, 2012) with the name “Tuned X-HYBRIDJOIN Algorithm” with new cost model calculation and experimental results. The size of both the non-swappable and the swappable parts of the disk buffer are tuned to have memory distributed optimally to give better performance of the algorithm. The experimental outcome of Tuned X-HYBRIDJOIN algorithm was compared with the X-HYBRIDJOIN and proved that the revised cost model tuned algorithm performance is significantly better than that of the X-HYBRIDJOIN algorithm.

Optimized X-HYBRIDJOIN

The researchers (Muhammad, Gillian & Gerald, 2011), (Muhammad, Gillian, Gerald & Imran, 2012) investigated whether the performance of the X-HYBRIDJOIN algorithm could be further improved by handling frequently accessed data in a different way. They proposed a new join algorithm called “Optimized X-HYBRIDJOIN Algorithm” (Asif, Gillian & Gerald, 2012). This algorithm has two phases, one called the stream-probing phase and the other the disk-probing phase. The stream-probing phase deals with disk-based relational master data tuples *R* that are accessed frequently, and the disk-probing phase deals with the other parts of the relational disk-based master data *R*. Experimental outcomes for this Optimized X-HYBRIDJOIN algorithm are significantly better when compared with the performance of X-HYBRIDJOIN. The motivation for this Optimized X-HYBRIDJOIN algorithm was to minimize the bottleneck in the stream of updates.

Figure. 2.8 shows the memory architecture for the Optimized X-HYBRIDJOIN algorithm. This new optimized join algorithm decomposes it into two hash join phases called the disk probing phase and the steam-probing phase which can be executed separately. With respect to memory size, the largest component of this algorithm are two hash tables, one used to store steam tuples which is denoted as H_S and other to store tuples from disk-based relational master data R, which is denoted as H_R. The disk buffer, stream buffer and queue are the other main components of the algorithm.

The optimized X-HYBRIDJOIN algorithm alternates between the disk-probing phase and the stream-probing phase, parts of the update stream that are not matched in H_R are stored in hash table H_S. If the stream buffer is empty, or the hash table H_S is full, the stream-probing phase ends and the disk-probing phase becomes active. In this disk-probing phase, the loading partition of R is determined by the oldest tuple found in the queue for the single probing step. When an adequate number of stream tuples are removed from the hash table H_S, often after one probe iteration, the algorithm switches back to the other probing phase. One stream-probing phase with a subsequent disk-probing phase establish one outer iteration of the Optimized X-HYBRIDJOIN algorithm. The disk-probing phase is not dependent on the stream-probing phase and so it can work on its own. The stream-probing phase boosts performance by quickly matching the more often used relational master data tuples R.

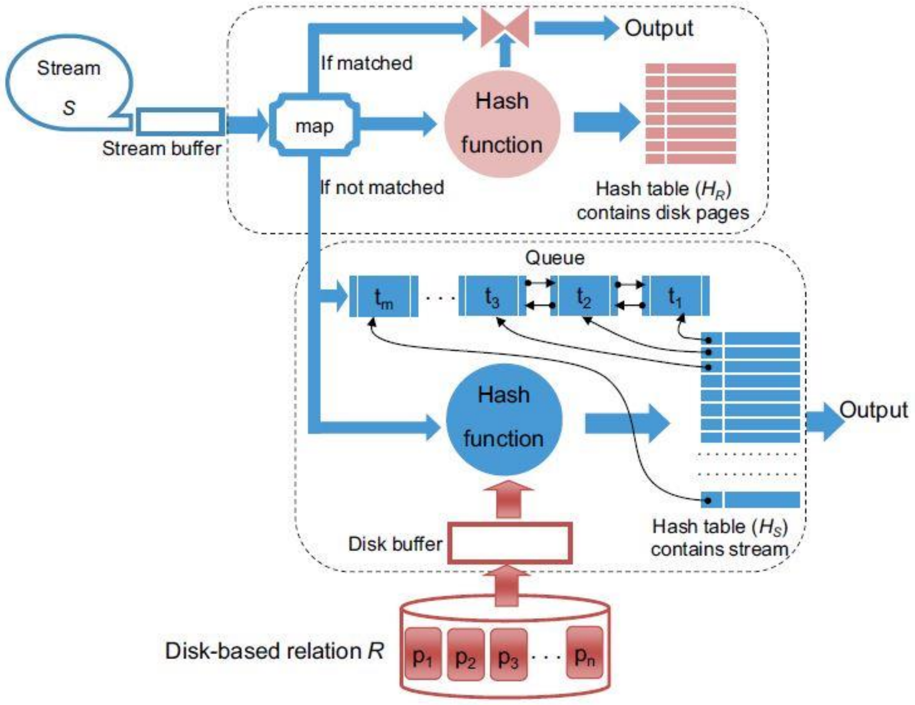


Figure 2.8 Memory architecture of Optimised X-HYBRIDJOIN algorithm (Asif, Gillian & Gerald, 2012)

2.2.3 Cache based Semi-Stream Joins

2.2.3.1 CMESHJOIN

The MESHJOIN algorithm is a novel semi-stream join algorithm that works under minimal assumptions about the stream data *S* and disk-based relational slow moving master data *R*, but does not perform well with skewed data distribution (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008). Hence a new algorithm called Cached MESHJOIN (CMESHJOIN) algorithm was proposed to improve the service rate by exploiting the skewed distributions of the stream data. (Asif, Gerald, Gillian & Christof, 2013) proposed a generic component called a cache which can be used as a front-stage for an arbitrary semi-stream join algorithm. Here the authors only considered one-to-many equijoins. This join occurs between a referenced primary key and foreign key. This join is an important class that occurs logically in online auction systems (Arvind, Shivnath & Jennifer, 2002), supply chain management (Eugene, Yanlei & Shariq, 2006) and in data warehousing (Lukasz, Theodore, Spencer & Vladislav, 2009). This algorithm is based on the MESHJOIN algorithm and it is extended with another phase called the Cache Front-stage to exploit skewed distributions.

By adding a front-stage to the MESHJOIN algorithm, a new algorithm called C-MESHJOIN algorithm as presented. Figure. 2.9 shows the execution architecture of this algorithm. In this algorithm both the MESHJOIN and the front stage are hash joins, hence this algorithm can be seen overall as holding two complementary hash join phases. In one phase, the MESHJOIN uses relational master data *R* stored in the tertiary memory. In the second join phase, the front-stage uses the incoming stream tuple *S* as the probing input and deals only with a small part of relational master data *R*. For each incoming stream *S* input, CMESHJOIN first uses the front-stage to find the join on frequent requests. If there is no join or match found in the front-stage, the stream tuple *S* is sent to the next phase called the MESHJOIN phase. This algorithm uses the original MESHJOIN architecture though there are alternative architectures such as using an order-preserving hash table data structure rather than a queue.

There are two main components in CMESHJOIN with respect to memory size and they are hash tables. One hash table is to store stream tuples, denoted as H_S and other hash table is to store the tuples from the disk based relational master table, denoted as H_R. This is relational master table hash H_R is the cache that stores most frequently accessed tuples from relational master data table *R*. Other main components are the disk buffer, frequency recorder, queue and stream buffer. The disk buffer is used to load parts of relational data R into memory using equally sized partitions. The frequency recorder is used to record the number of time tuples stored in H_R are accessed. The queue stores the pointers to the stream tuples that are saved in stream hash table H_S by keeping track of the tuples' order and enabling the removal of completely processed tuples. The stream buffer is a small buffer which holds part of stream for a while if necessary.

CMESHJOIN alternates between the front-stage phase and the MESHJOIN phase. Streams that do not match hash table H_R are stored in hash table H_S. The first front-stage phase will end if hash table H_S is completely full or the stream buffer becomes empty. Once the front-stage has ended, the second phase, MESHJOIN, becomes active. In every iteration of MESHJOIN, the algorithm loads a set of master data *R* tuples into memory to amortize the cost of disk access. Once disk pages are loaded into the disk buffer, the algorithm starts its probing and each tuple in the disk buffer in H_S is probed. Output is generated once the match is found. After each iteration the oldest chunk of stream data is removed from H_S. As the algorithm reads master data *R* sequentially, an index is not required on *R*. Here the front-stage phase is used to improve performance by quickly matching the most frequently accessed master data *R*. An experimental study was performed and discussed with a cost model (Asif, Gerald, Gillian & Christof, 2013), on one-to-many joins and provided better performance than the MESHJOIN algorithm.

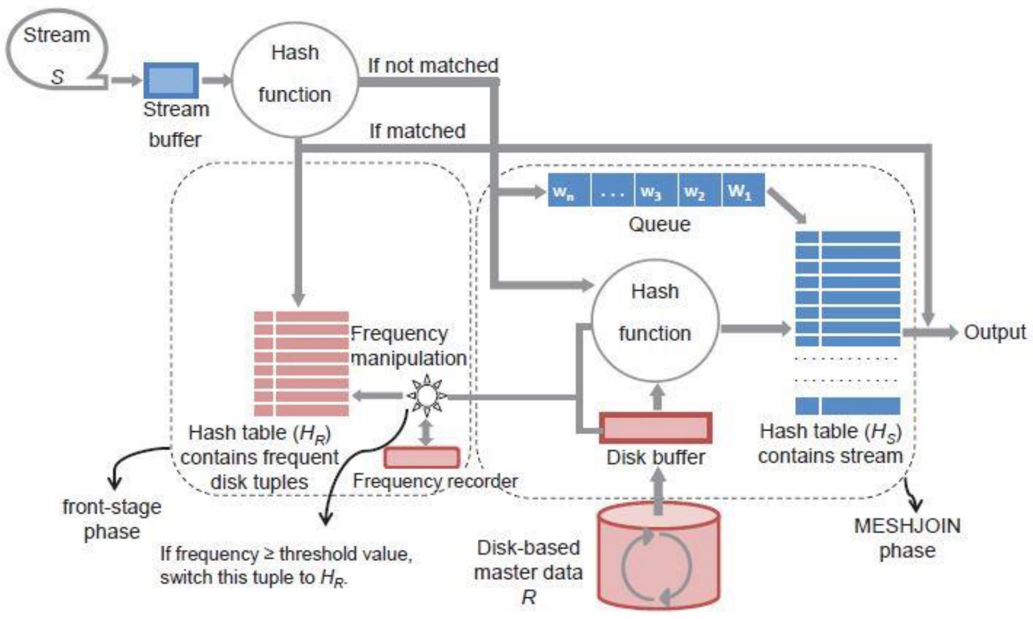


Figure 2.9 Data structure and architecture of CMESHJOIN algorithm (Asif, Gerald, Gillian & Christof, 2013)

2.2.3.2 CACHEJOIN

CACHEJOIN is another semi-stream join algorithm which is based on the caching method. This algorithm was proposed to give better performance in long-tailed skewed distribution patterns in incoming stream data (transactional data). Functionally, this semi-stream join is similar to HYBRIDJOIN which was explained in section 2.2.2.2. In CACHEJOIN a new logic is introduced which caches frequently-accessed relational database master data tuples in the static memory and this is called phase one in the CACHEJOIN algorithm (Figure 3.1). The second phase is basically a HYBRIDJOIN phase. So this CACHEJOIN has two phases, firstly the algorithm try to find the match from cached frequently accessed relational database master table tuple and if there is no match is found then it moves on to the second phase. There is a threshold factor involved to identify the frequently-accessed tuples. This threshold level is defined by the user at the beginning of this algorithm. At this stage there is no automatic tuning process done by the algorithm while it is running, but the authors (Asif, Gillian & Gerald, 2012) will develop future work in this area. As the algorithm does the joining with the most frequently cached tuples first, this boosts performance. As this was the algorithm chosen to implement with the MITRE 10 NZ dataset, it is explained in more detail with a cost model and architecture in the following.

Chapter 3

3 Implementation of CACHEJOIN in MITRE 10 NZ

3.1 MITRE 10 NZ current setup

Currently MITRE 10 doesn’t have an active or real-time data warehouse. They have a traditional data warehouse which always has one day old data. Normally a data backup of the live system to the data warehouse cannot be performed during business hours due to data integrity issues and because to do so has an effect on the live system giving rise to system slowness. When entering data into the data warehouse, a few processes such as data mapping are necessary. Normally, copying data from the live database to the data warehouse is done at night time. At MITRE 10 NZ, the live, point-of-sale system is connected to a UNIX server database called Powerflex. Added to this, the backup strategy is not optimal as there cannot be any JDBC installed in the system, so there is no way to easily establish a connection to the database from Java IDE. Instead all tables from the Powerflex database are copied across to the MySQL data warehouse at night and from there reports are created and published to users. This means users will only have a day-old data. If they want to know any current stock inventory position for an item then they need to log in to the AS400 system and run a report for that item to know the status. This cannot be undertaken on all items at one time and performing it on every item individually is also not feasible due to the range of products they have. Though there are many tables available in the data warehouse only three tables’ data is of interest here. They are; transactional stream input data, product master data and customer master data. These attributes and data types are explained in Tables 3.1, 3.2 and 4.2 respectively.

3.2 Suitability of CACHEJOIN to MITRE 10 NZ

In this work we implemented CACHEJOIN algorithm in MITRE 10 NZ, one of the largest companies in New Zealand, and tested using both synthetic and live MITRE 10 NZ transactional datasets. Performance of this algorithm on both datasets is observed and studied. MITRE 10 NZ has master data of nearly three hundred thousand unique product items indexed by product key. As this master data is uniquely indexed MESHJOIN, R-MESHJOIN and

CMESHJOIN algorithms were not considered for this implementation as these algorithms do not implement indexing. On the other hand HYBRIDJOIN, X-HYBRIDJOIN and CACHEJOIN algorithms use indexes to boost performance and work well with unique, indexed master data whereas the MITRE 10 NZ master data is also unique. Moreover, we analyzed MITRE 10 NZ and identified that it has a long-tail skew in it. In this case, around 15% of the master data being more frequently used and the other product items are rarely being purchased by customers hence not often used. This distribution pattern suits HYBRIDJOIN, X-HYBRIDJOIN and CACHEJOIN algorithms well (Asif, Gillian & Gerald, 2011), (Asif, Gillian & Gerald, 2012), (Muhammad, Gillian & Gerald, 2011), (Asif, Gillian, Gerald & Christof, 2012), (Asif, 2014).

Now the question was, which algorithm from the above list is optimal for MITRE 10 NZ? The HYBRIDJOIN algorithm does not use caching method whereas the CACHEJOIN algorithm use caching method in the first phase before it moves into the second phase which is technically a HYBRIDJOIN. Because the CACHEJOIN algorithm has advantage of caching method adding to the HYBRIDJOIN, the HYBRIDJOIN algorithm were not considered for this implementation. The X-HYBRIDJOIN algorithm uses caching method but the limitation for this algorithm was the need of sorting master table. This make a difference from the CACHEJOIN algorithm whereas there was no sorting needed. Also before making the selection the size of MITRE 10 NZ master data was studied. The size of the product master data was around only three hundred thousand. Because of the X-HYBRIDJOIN algorithm limitation we see that the CACHEJOIN algorithm was more suitable as it could cache almost every frequently used product items into the hash memory. As most of the frequently-used product items are cached in the hash memory, performance would be better than that of the HYBRIDJOIN algorithm. Most of the incoming stream tuples would be joined in the first phase of the CACHEJOIN algorithm itself before it moved into the second phase of join which is typically HYBRIDJOIN phase. In this way advantage is derived from cached memory. These observations motivate us to implement the CACHEJOIN algorithm to process MITRE 10 NZ transactional data.

3.2.1.1 Components of CACHEJOIN

Following are the key components involve in CACHEJOIN algorithm:

Disk Buffer: This is the component that loads the disk pages from the disk-based master data table (product) into the memory. The number of tuples to be retrieved from the master table to the disk pages can be controlled by the user and this needs to be set before executing this program.

Stream Buffer: The stream buffer is used to hold the fast, incoming stream if required. For example, if the rate of incoming stream tuples is faster than the service rate of the algorithm, then the stream tuples will overflow. In this case, this overflow part of the input stream can be stored in the stream buffer temporarily.

Hash Tables: There are two hash tables used in the CACHEJOIN algorithm unlike other algorithms such as MESHJOIN, R-MESHJOIN, and HYBRIDJOIN. They are called the H_R and H_S tables. H_R stores most frequently-used disk tuples from the product master data table. H_S stores the incoming stream tuples. As mentioned above, the Java hash table does not support the storing of multiple tuples containing the same key value at the same time. Therefore MultiHashMap provided by Apache is used to implement the algorithm. The fudge factor value that was considered in this implementation is 8.

Queue: The queue is used to store the pointer addresses of tuples in the hash table in order to keep a record of expired tuples. Each node in the queue is based on a double-linked list which contains the attribute value and the addresses of one-step neighbor nodes.

Frequency Recorder: This component is used to record the frequency of matching the master data tuples with stream tuples. This component is an important as this is the gate for the master data tuples to enter into the cache memory where frequent master data tuples are stored and used to process input stream tuples. There is a threshold value to decide the switching of master data tuple into the cache. This threshold is a flexible barrier and its value is controlled automatically by the algorithm.

Following are the two inputs that algorithm take in each of its iteration:

Relation *R*: This is the disk-based relational master data *R* that is stored in a MySQL database.

Stream *S*: This is the incoming stream data which is basically transactional data from the point of sale system in any retail store. Each stream tuple typically includes keys and a few attributes at the point of sale time such as quantities, time and date of sale, promotional offers, etc.

3.3 CACHEJOIN execution architecture

The CACHEJOIN (Cache Join) is a well-known algorithm (Asif, Gillian & Gerald, 2012) and was particularly designed to process skewed stream data with disk-based master data efficiently. In this algorithm, performance is not affected when a large number of unused or rarely used data is added to the disk-based relational master data *R*.

Figure. 3.1 shows the data structure and architecture of the CACHEJOIN algorithm. The pseudocode of the algorithm is given in Algorithm 3.1. An index is required for the CACHEJOIN algorithm to access the master data *R* selectively. This algorithm has two complementary hash join phases. One is disk-probing phase in which *R* is used as a probing input with the largest part stored in the tertiary memory. The other is called stream-probing phase in which the stream data is used as the probing input. This phase deals only with a small set of *R*. For every incoming stream tuple, CACHEJOIN algorithm first uses the stream-probing phase to quickly find matches on frequently-requested tuples from the master data *R* in cache. If there is no match found, the incoming stream tuple is forwarded into the next probing phase called the disk-probing phase. In this algorithm two hash tables H_S and H_R are the largest components with respect to the memory size. H_S is used to store stream tuples whereas H_R is used to store master data tuples. The other main memory components are the disk buffer, the stream buffer and the queue. The disk buffer loads the disk pages from the disk-based master data table (product) into the memory, the stream buffer is used to hold the fast incoming stream if required and the queue is used to store the pointer addresses of tuples in the hash table in order to keep a record of expired tuples. The CACHEJOIN algorithm alternates between the stream-based probing phase, and the disk-based probing phase.

Once the stream buffer is empty or the hash table H_S is completely full, the stream-probing phase ends and the disk-probing phase becomes active. In the disk-probing phase, the oldest tuple found in the queue is used to decide the master-data partition which is loaded for a single disk-probing phase into the disk buffer. This ensures that in every probe step process, the CACHEJOIN algorithm matches at least one tuple. After this probe step, and an adequate number of stream tuples are matched in hash table H_S, these are removed using the queue which supports this process of removing processed tuples from H_S. Once the disk-probing phase is complete, the algorithm is switched back to the stream-probing phase and this constitutes one outer iteration of the CACHEJOIN.

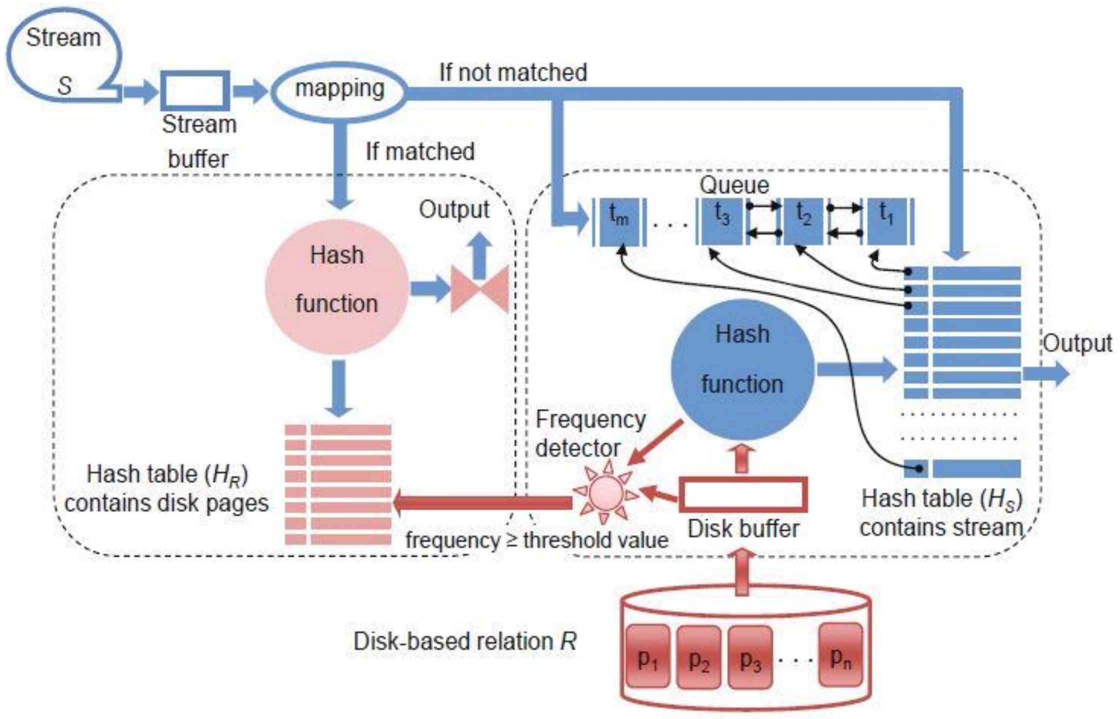
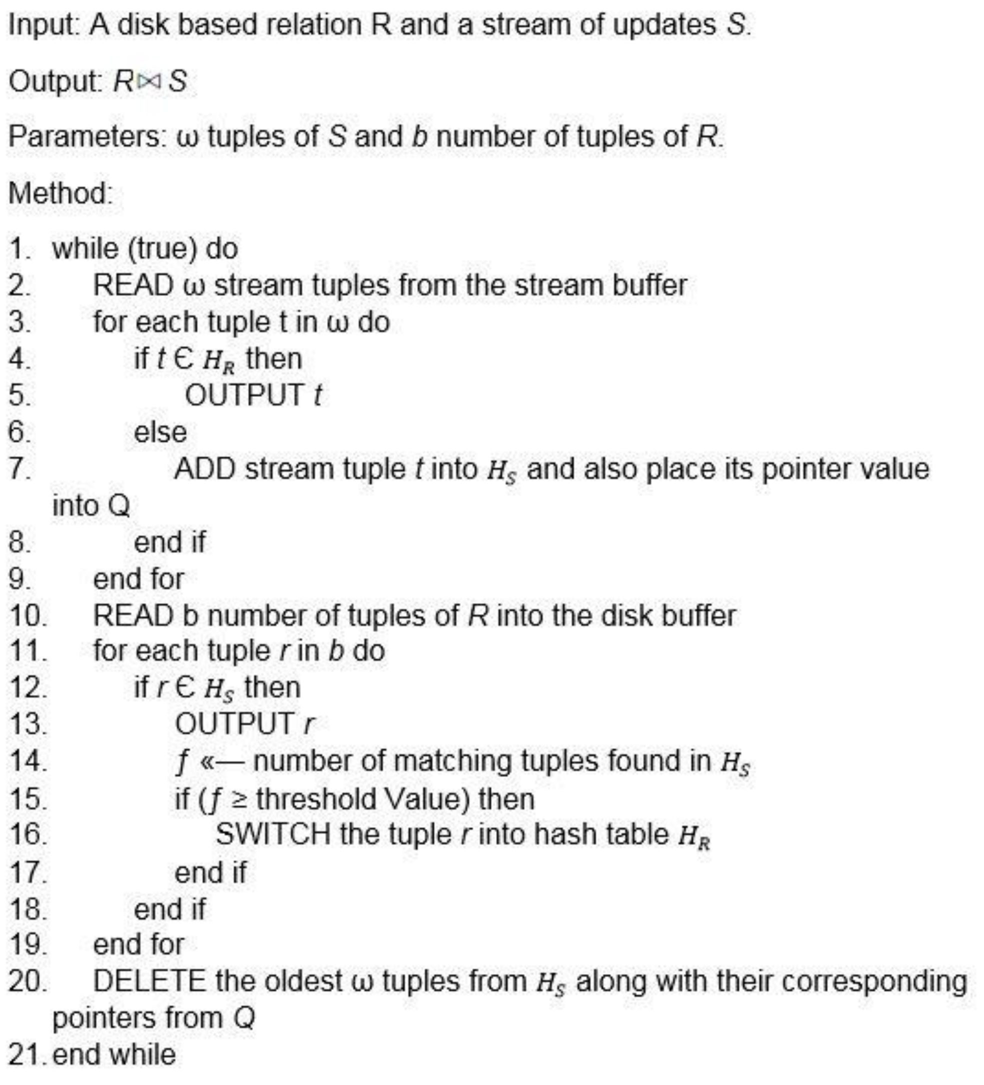


Figure 3.1 Data structure and architecture of CACHEJOIN algorithm (Asif, Gillian & Gerald, 2012)

As with the HYBRIDJOIN algorithm (Asif, Gillian & Gerald, 2011) the disk-probing phase can work independently from the stream-probing phase. The authors (Asif, Gillian & Gerald, 2012), (Asif, Gillian & Gerald & Chistof, 2013), (Asif, 2014) give a cost model of this algorithm and experimental study of the CACHEJOIN algorithm performance compared to the MESHJOIN algorithm (Neoklis, Spiros, Panos, Alkis & Nils-Erik, 2008), the R-MESHJOIN algorithm (Asif, Gillian, Gerald & Shafiq, 2010) and the HYBRIDJOIN algorithm (Asif, Gillian & Gerald, 2011).



```
Input: A disk based relation R and a stream of updates S.
Output: $R \bowtie S$
Parameters: ω tuples of S and b number of tuples of R.
Method:
1.  while (true) do
2.      READ ω stream tuples from the stream buffer
3.      for each tuple t in ω do
4.          if $t \in H_R$ then
5.              OUTPUT t
6.          else
7.              ADD stream tuple t into $H_S$ and also place its pointer value into Q
8.          end if
9.      end for
10.     READ b number of tuples of R into the disk buffer
11.     for each tuple r in b do
12.         if $r \in H_S$ then
13.             OUTPUT r
14.             $f$ «— number of matching tuples found in $H_S$
15.             if ($f \geq$ threshold Value) then
16.                 SWITCH the tuple r into hash table $H_R$
17.             end if
18.         end if
19.     end for
20.     DELETE the oldest ω tuples from $H_S$ along with their corresponding pointers from Q
21. end while
```

Algorithm 3.1 Pseudo-code for CACHEJOIN algorithm (Asif, 2014)

3.4 Experimental Set up

This section explains the system used for this implementation and the characteristics of the datasets.

3.4.1 System Setup

We implement the algorithm in Java language using Eclipse IDE version 4.4.0. We run our experiments on Intel Core i5 processor with 8GB main memory (RAM) and 700GB disk memory under the Windows 7 Professional Edition 64-Bit Operation System. The master data is stored on disk using MySQL database. To measure the memory cost of the algorithm we use external library “Sizeofag.Jar”. To measure the processing cost we use “System.NanoTime ()” method, provided by Java API. As Java hash tables do not support the storing of multiple tuples corresponding to the same key value, MultiHashMap provided by Apache Common Collections is used to store multiple tuples corresponding to the same key value at the same time. This is necessary as in this experiment this algorithm runs on a retail industry dataset where many customers can buy the same product within the same time period. For example, Customer A and Customer B can buy a cleaning towel at the same time.

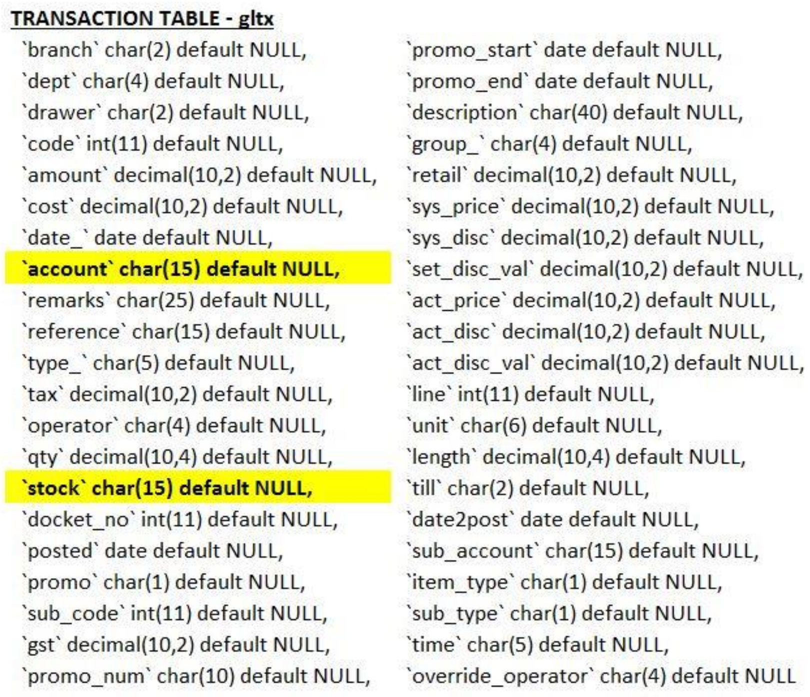
3.4.2 Datasets

3.4.2.1 Synthetic dataset

The stream dataset used in this algorithm is based on a Zipfian’s distribution which is found in a wide range of applications (Chris, 2006). There are 42 attributes in each tuple with size of 168 bytes. We consider each stream tuple equal in size to the MITRE 10 NZ transactional stream tuple. And the size of each master data tuple is also consider equal to the size of MITRE 10 NZ master data tuple. In our experiment the size of master data is three hundred thousand unique indexed tuples with each tuple having 86 attributes. Basically, we keep the structure of the synthetic dataset same as MITRE 10 NZ’s dataset, so that performance is compared fairly against the real-life dataset.

3.4.2.2 Real-life dataset

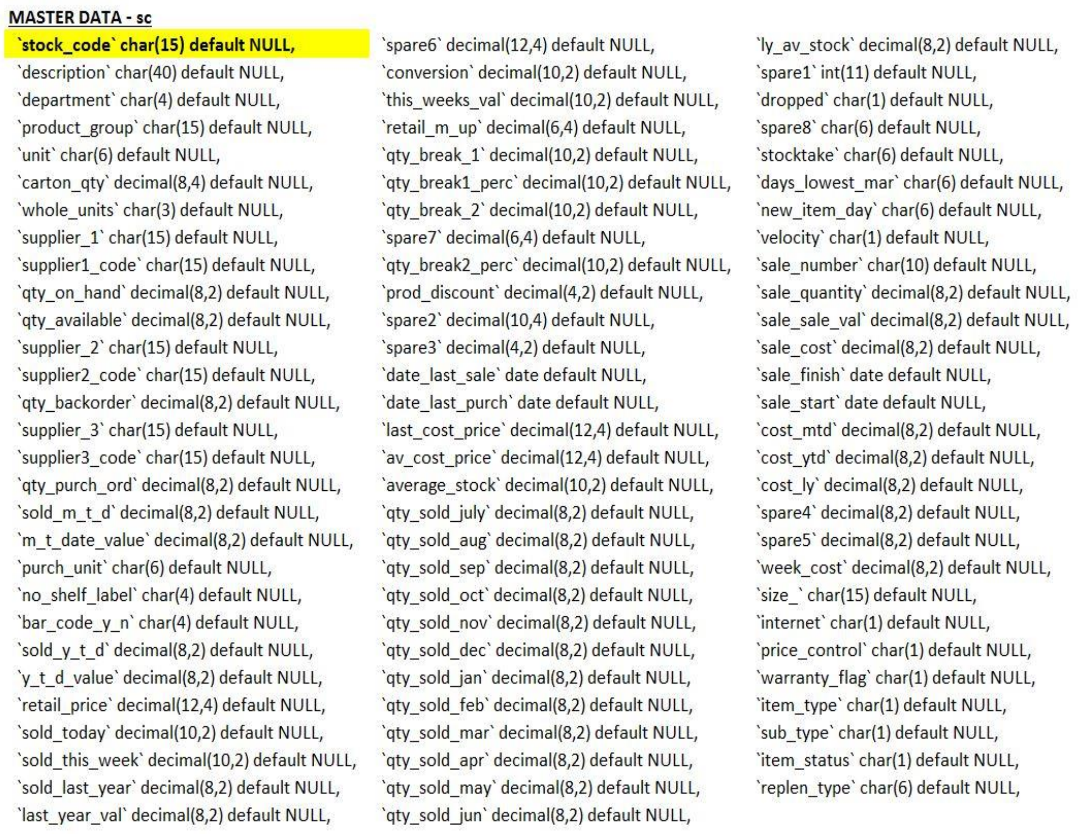
We use the real-life transactional dataset from MITRE 10 NZ. The incoming stream tuples have 42 attributes with different data types and they are explained below in Table. 3.1. The keys in this transactional data are *stock_code* and *account_code* (both are foreign key). In the CACHEJOIN algorithm we implement the join of only one attribute *stock_code*. *Stock_code* is the foreign key that joins with the primary key in master table called *sc*. This *sc* table has 86 attributes for every item/product as presented in Table. 3.2.



TRANSACTION TABLE - gltx

```
`branch` char(2) default NULL,
`dept` char(4) default NULL,
`drawer` char(2) default NULL,
`code` int(11) default NULL,
`amount` decimal(10,2) default NULL,
`cost` decimal(10,2) default NULL,
`date_` date default NULL,
`account` char(15) default NULL,
`remarks` char(25) default NULL,
`reference` char(15) default NULL,
`type_` char(5) default NULL,
`tax` decimal(10,2) default NULL,
`operator` char(4) default NULL,
`qty` decimal(10,4) default NULL,
`stock` char(15) default NULL,
`docket_no` int(11) default NULL,
`posted` date default NULL,
`promo` char(1) default NULL,
`sub_code` int(11) default NULL,
`gst` decimal(10,2) default NULL,
`promo_num` char(10) default NULL,
`promo_start` date default NULL,
`promo_end` date default NULL,
`description` char(40) default NULL,
`group_` char(4) default NULL,
`retail` decimal(10,2) default NULL,
`sys_price` decimal(10,2) default NULL,
`sys_disc` decimal(10,2) default NULL,
`set_disc_val` decimal(10,2) default NULL,
`act_price` decimal(10,2) default NULL,
`act_disc` decimal(10,2) default NULL,
`act_disc_val` decimal(10,2) default NULL,
`line` int(11) default NULL,
`unit` char(6) default NULL,
`length` decimal(10,4) default NULL,
`till` char(2) default NULL,
`date2post` date default NULL,
`sub_account` char(15) default NULL,
`item_type` char(1) default NULL,
`sub_type` char(1) default NULL,
`time` char(5) default NULL,
`override_operator` char(4) default NULL
```

Table 3.1 Data specifications of MITRE 10 NZ transactional stream dataset.



```
MASTER DATA - sc
`stock_code` char(15) default NULL,
`description` char(40) default NULL,
`department` char(4) default NULL,
`product_group` char(15) default NULL,
`unit` char(6) default NULL,
`carton_qty` decimal(8,4) default NULL,
`whole_units` char(3) default NULL,
`supplier_1` char(15) default NULL,
`supplier1_code` char(15) default NULL,
`qty_on_hand` decimal(8,2) default NULL,
`qty_available` decimal(8,2) default NULL,
`supplier_2` char(15) default NULL,
`supplier2_code` char(15) default NULL,
`qty_backorder` decimal(8,2) default NULL,
`supplier_3` char(15) default NULL,
`supplier3_code` char(15) default NULL,
`qty_purch_ord` decimal(8,2) default NULL,
`sold_m_t_d` decimal(8,2) default NULL,
`m_t_date_value` decimal(8,2) default NULL,
`purch_unit` char(6) default NULL,
`no_shelf_label` char(4) default NULL,
`bar_code_y_n` char(4) default NULL,
`sold_y_t_d` decimal(8,2) default NULL,
`y_t_d_value` decimal(8,2) default NULL,
`retail_price` decimal(12,4) default NULL,
`sold_today` decimal(10,2) default NULL,
`sold_this_week` decimal(10,2) default NULL,
`sold_last_year` decimal(8,2) default NULL,
`last_year_val` decimal(8,2) default NULL,
`spare6` decimal(12,4) default NULL,
`conversion` decimal(10,2) default NULL,
`this_weeks_val` decimal(10,2) default NULL,
`retail_m_up` decimal(6,4) default NULL,
`qty_break_1` decimal(10,2) default NULL,
`qty_break1_perc` decimal(10,2) default NULL,
`qty_break_2` decimal(10,2) default NULL,
`spare7` decimal(6,4) default NULL,
`qty_break2_perc` decimal(10,2) default NULL,
`prod_discount` decimal(4,2) default NULL,
`spare2` decimal(10,4) default NULL,
`spare3` decimal(4,2) default NULL,
`date_last_sale` date default NULL,
`date_last_purch` date default NULL,
`last_cost_price` decimal(12,4) default NULL,
`av_cost_price` decimal(12,4) default NULL,
`average_stock` decimal(10,2) default NULL,
`qty_sold_july` decimal(8,2) default NULL,
`qty_sold_aug` decimal(8,2) default NULL,
`qty_sold_sep` decimal(8,2) default NULL,
`qty_sold_oct` decimal(8,2) default NULL,
`qty_sold_nov` decimal(8,2) default NULL,
`qty_sold_dec` decimal(8,2) default NULL,
`qty_sold_jan` decimal(8,2) default NULL,
`qty_sold_feb` decimal(8,2) default NULL,
`qty_sold_mar` decimal(8,2) default NULL,
`qty_sold_apr` decimal(8,2) default NULL,
`qty_sold_may` decimal(8,2) default NULL,
`qty_sold_jun` decimal(8,2) default NULL,
`ly_av_stock` decimal(8,2) default NULL,
`spare1` int(11) default NULL,
`dropped` char(1) default NULL,
`spare8` char(6) default NULL,
`stocktake` char(6) default NULL,
`days_lowest_mar` char(6) default NULL,
`new_item_day` char(6) default NULL,
`velocity` char(1) default NULL,
`sale_number` char(10) default NULL,
`sale_quantity` decimal(8,2) default NULL,
`sale_sale_val` decimal(8,2) default NULL,
`sale_cost` decimal(8,2) default NULL,
`sale_finish` date default NULL,
`sale_start` date default NULL,
`cost_mtd` decimal(8,2) default NULL,
`cost_ytd` decimal(8,2) default NULL,
`cost_ly` decimal(8,2) default NULL,
`spare4` decimal(8,2) default NULL,
`spare5` decimal(8,2) default NULL,
`week_cost` decimal(8,2) default NULL,
`size_` char(15) default NULL,
`internet` char(1) default NULL,
`price_control` char(1) default NULL,
`warranty_flag` char(1) default NULL,
`item_type` char(1) default NULL,
`sub_type` char(1) default NULL,
`item_status` char(1) default NULL,
`replen_type` char(6) default NULL,
```

Table 3.2 Data specifications of MITRE 10 NZ product master table dataset.

Characteristics of Real-life transactional dataset

We studied the MITRE 10 NZ transaction (incoming stream) dataset to understand the tuple arrival pattern. This was necessary so suitable semi-stream joins could be selected according to the behavior or pattern of MITRE 10 dataset used for implementation. The study confirms that the incoming stream arrival pattern is identified as a long-tailed skewed pattern and Figure. 3.2 displays the distribution pattern of MITRE 10 NZ incoming stream data. This figure presents the pattern on two scales, the first graph showing the approach in normal scale whereas the second graph is based on a logarithmic scale. These graphs shows that the customers are more frequently buying a few items rather than every item being purchased at the same rate. This concludes that that there could be many items which are not purchased over a period.

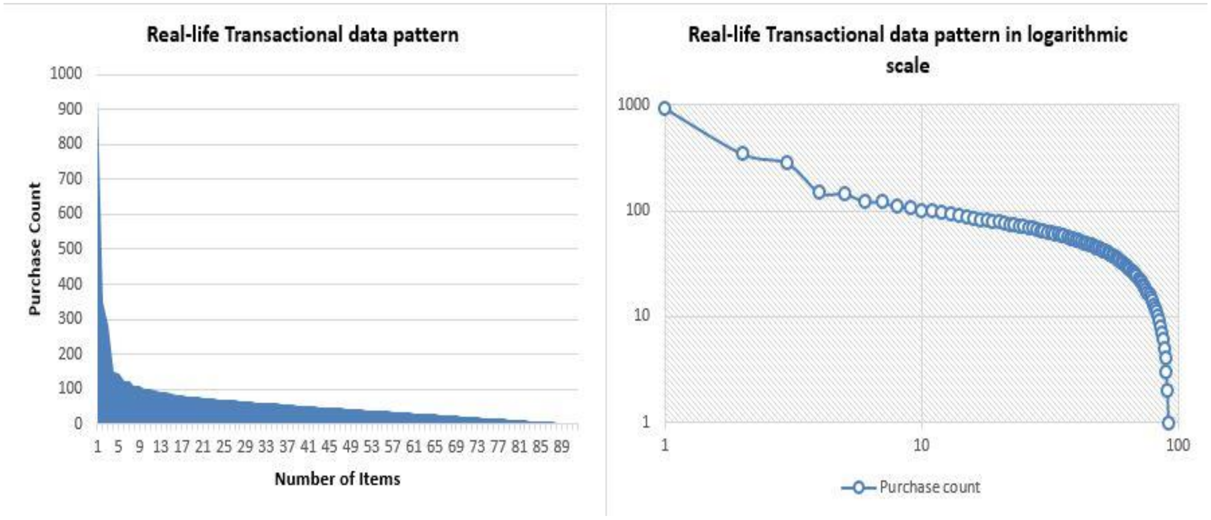


Figure 3.2 Incoming (transactional) dataset data distribution pattern of MITRE 10 NZ

3.5 Execution of the CACHEJOIN algorithm

To execute CACHEJOIN algorithm on the MITRE 10 setup a few things needed to be done. The aim of this study is to execute the CACHEJOIN algorithm on both a synthetic dataset and on the MITRE 10 dataset and carry out a performance comparison study on both data results. The algorithm was executed with different memory budgets as well. In order to do this successfully, we need to do some preparation work before running this algorithm. As explained in Section 3.1, we found few difficulties in implementing CACHEJOIN algorithm in live MITRE10 point-of-sale system. This is mainly because of current system architecture of MITRE10 NZ. There cannot be any JDBC installed in the live point-of-sale system. Hence the point-of-sale system cannot be connected to the master tables which is in MySQL database for joining streams with master data. To overcome this issue we took a copy of real transaction data from the point-of-sale system backend UNIX database and created a same replica table in MySQL database. This allowed us to run the CACHEJOIN algorithm with MITRE10 NZ real dataset. Also as explained in Section 3.4.1, to handle multiple tuples with the same key value MultiHashMap Jar files provided by Apache need to be added. Another jar file called "SizeofJar" is added to measure the processing cost. The System.Nanotime () method is used to measure processing time. A JDBC MySQL connector is added to the program to establish a connection between the MySQL master tables. Finally, output costs are written to the text file by the algorithm, the file name and location being controlled by the user.

To do a proper comparison study (apples-apples and oranges-oranges), the CACHEJOIN algorithm is used to run on a synthetic dataset amended to have the same number of attributes as the MITRE 10 dataset. In this study the size of the product master table does not change over all experiments. This is based on the MITRE 10 real-life scenario. For incoming transactional stream data, the entire December 2014 live data received from MITRE 10 NZ was used. The reason for using an entire month's volume is to have a larger dataset for experiment. All experiments are run during a quiet period where no other processes are running on the system at the same time. Once the execution is completed and the output is written to the output text file, then the CACHEJOIN cost calculation is applied to derive the service rate. From the output data, the first and last 15% of the data was not considered in our calculation. This is to make sure that we are not considering noisy scores in our service rate calculation. This is also enough time to give to the algorithm a warmup before starting to measure the costs. This is achieved by adding the FOR loop in execution process and only the last round costs are captured. In our experiments the algorithm is run for four times and on the fourth run, the costs are captured. This makes the comparison study more meaningful as the number of attributes and tuples considered are the same in both synthetic and real-life executions.

Experiments are completed with synthetic and real-life dataset as stated in the above section. The performance comparison study on service levels is done and explained in the following section.

3.6 Results

We want to understand the behavior of CACHEJOIN algorithm when it run with different memory sizes. The same algorithm is run with a synthetic dataset and with the MITRE 10 NZ dataset without making any changes to any user-defined parameter values as stated at the beginning of the CACHEJOIN.java file. And the system setup also remains the same when we run the algorithm with two different datasets. Service rate is calculated using the cost model by the authors (Asif, Gillian & Gerald, 2012).

Performance comparisons for different memory budgets: This experiment compares the performance of CACHEJOIN algorithm using both synthetic and real-life datasets. We run our experiment for different memory budgets varying from 50MB to 200MB. The size of the product master table (*sc* table) is set to a fixed size of 300,000 tuples in all our experiments. Figure. 3.3 shows the output of this experiment. From the figure we can see that the service rate (tuples processed in a second) of CACHEJOIN algorithm on MITRE 10 NZ dataset is very close to that of synthetic dataset. The reason for the slight variance could be due to the skew variation in real-life dataset. This also confirms that the implementation of CACHEJOIN algorithm on the MITRE 10 NZ dataset produced an acceptable output.

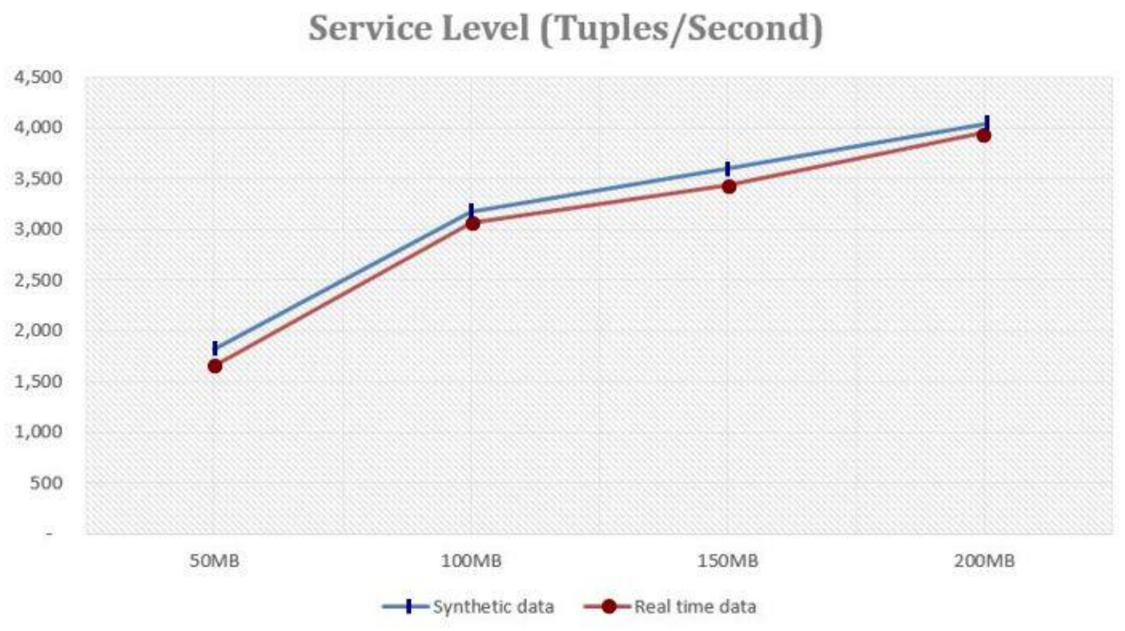


Figure 3.3 CACHEJOIN algorithm service rate comparison of synthetic and real-life dataset.

Chapter 4

4 Extension of CACHEJOIN in MITRE 10 NZ

4.1 Motivation

MITRE 10 NZ transactional data includes two key attributes (*stock_code* and the *account_code)* which need to be joined with the two different tables in disk-based master data. Product attributes can be identified by joining the *stock_code* key into product master table using simple CACHEJOIN algorithm. But to understand the type of transactions (whether retail customer or trade customer) we need to have two phase CACHEJOIN algorithm which *account_code* key from the same transaction is joined with another master table called *cs_person*. This second join cannot be done with current simple CACHEJOIN algorithm and hence there is a need to have two phase join CACHEJOIN algorithm. The *stock_code* attribute needs to join with the product table (*sc*) in master data to get information about each product item such as product name, product color, product length, product retail price, product cost price and etc. For example, hammer, Black color, 60cm length, $30, $25, etc. While, the attribute *account_code* needs to with the customer table in the master data to get information about the customer such as customer name, customer address, customer contact number, customer trading name etc. For example, David George, Auckland, 09-123456, Global Traders, etc. This joining of a stream tuple with two tables in the master data motivates us to extend our normal CACHEJOIN algorithm for implementing the join operation with two different tables in the master data. We call it Two-Stage join. Under the first join stage (which is normal CACHEJOIN) stream tuple is joined with the product table (*sc*) in the master data using *stock_code* as a join attribute while under the second join stage the stream tuple is joined with the customer table in the master data using *account_code* as a join attribute. In this chapter we implement our Two-Stage join algorithm and analyze its performance with normal CACHEJOIN using MITRE 10 NZ dataset.

4.2 Two-Stage Join architecture and algorithm

A simple architectural design of Two-Stage join algorithm is shown in Figure 4.1 while its pseudo-code is presented in Algorithm 4.1.

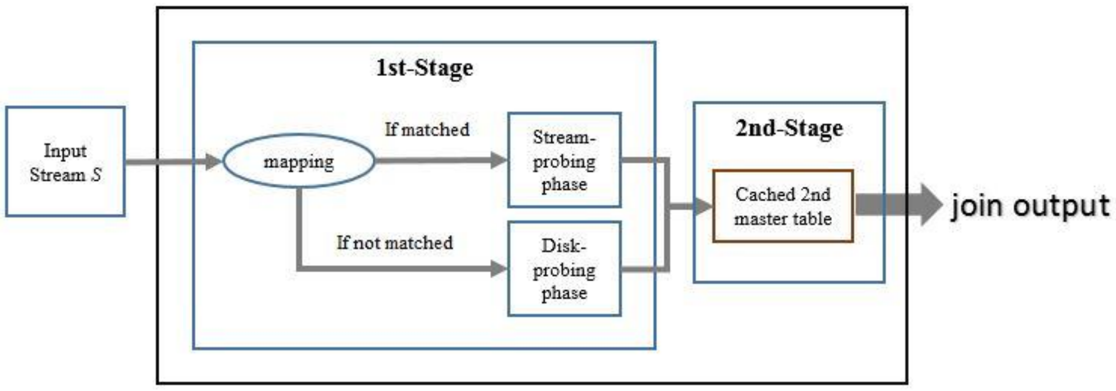


Figure 4.1 A simple architectural design for Two-Stage Join

From the figure the first stage implements the normal CACHEJOIN. As normal CACHEJOIN has further two phases – the stream-probing phase and the disk-probing phase – so on arriving of each stream tuple the CACHEJOIN algorithm maps it to the right phase for the processing. In CACHEJOIN the stream tuple is joined with the product table in the master data using *stock_code* as a join attribute. Each stream tuple after completing the Ist-stage is directed to the 2nd-stage. In 2nd-stage the stream joins with the customer table of the master data using *account_code* as a join attribute. Since the customer table is significantly smaller than the product table therefore, we load the whole table in the cache. Hence there is no disk-probing phase in the 2nd-stage of the algorithm. Once a stream tuple has completed these two joins it then moves to the output. We study the performance of our Two-Stage join algorithm using both real-time and a synthetic datasets.

```
Input: A disk based relation R and a stream of updates S.
Output: $R \bowtie S$
Parameters: ω tuples of S and b number of tuples of R.
Method:
1.  while (true) do
2.      READ ω stream tuples from the stream buffer
3.      for each tuple t in ω do
4.          if t is in $H_R$ then
5.              OUTPUT $t \bowtie H_{2R}$
6.          else
7.              ADD stream tuple t into $H_S$ and also place its pointer value
    into Q
8.          end if
9.      end for
10.     READ b number of tuples of R into the disk buffer
11.     for each tuple r in b do
12.         if r is in $H_S$ then
13.             OUTPUT $r \bowtie H_{2R}$
14.             f «— number of matching tuples found in $H_S$
15.             if ($f \geq$ threshold Value) then
16.                 SWITCH the tuple r into hash table $H_R$
17.             end if
18.         end if
19.     end for
20.     DELETE the oldest ω tuples from $H_S$ along with their corresponding
    pointers from Q
21. end while
```

Algorithm 4.1 Pseudo-code for Two-Stage join algorithm

4.3 Cost model for Two-Stage join

In this section we present the cost model for our Two-Stage join algorithm. Similar to the architecture and the algorithm we also extend the normal CACHEJOIN's cost model (Asif, Gillian & Gerald, 2012) for our Two-Stage join algorithm. Later we also use our cost model to calculate the analytical costs for our Two-Stage join algorithm.

Normally, under the cost model we calculate two costs - memory cost and the processing cost. Equation 4.1 and 4.2 present these costs respectively. Table 4.1 below describes the notations we used in deriving of our cost model.

For memory cost, considering the Ist-stage (normal CACHEJOIN) a main portion of the memory is assigned to the hash table H_S along with the queue. Whereas, a much smaller portion is assigned to hash table H_R and the disk buffer. Also compare to the 1st-stage relatively a smaller portion of memory is assigned to the 2nd-stage. We calculate memory for each component as below.

Memory for disk buffer (bytes) = $k.vp$

Memory for H_S (bytes) = $\alpha\ [M - (k + l)\ vp]$

Memory for H_R (bytes) = $l.vp$

Memory for the queue (bytes) = $(1 - \alpha)\ [M - (k + l)\ vp]$

Memory for 2nd-stage (bytes) $m_2 = k_2.vp_2$

The below equation (4.1) gives the total memory M cost for CACHEJOIN by aggregating the above calculations.

$$M = (k + l)\ vp + \alpha\ [M - (k + l)\ vp] + (1 - \alpha)\ [M - (k + l)\ vp] + m_2 \qquad (4.1)$$

Note, due to the negligible size of the stream buffer memory, 0.05 MB we do not include this in our calculation.

In order to make the processing cost calculation simple, the cost for every individual components is calculated first and then all costs are summed to calculate the processing cost for one iteration.

Cost to read k pages into the disk buffer = $C_{i/o}(k \cdot vp)$

Cost to look-up w_n tuples in $H_R = w_n \cdot C_H$

Cost to look-up w_n tuples in 2nd-stage = $w_n \cdot C_{2H}$

Cost to look-up w_s tuples in 2nd-stage=$w_s \cdot C_{2H}$

Cost to look-up disk buffer tuples in $H_S = d \cdot C_H$

Cost to compare all tuples frequency in disk buffer with the threshold value = $d \cdot C_F$

Cost to generate the output for w_n tuples in 2nd-stage = $w_n \cdot C_{2O}$

Cost to generate the output for w_s tuples in 2nd-stage = $w_s \cdot C_{2O}$

Cost to read the w_n tuples from the stream buffer = $w_n \cdot C_S$

Cost to read the w_s tuples from the stream buffer = $w_s \cdot C_S$

Cost to append w_s tuples into H_S and the queue = $w_s \cdot C_A$

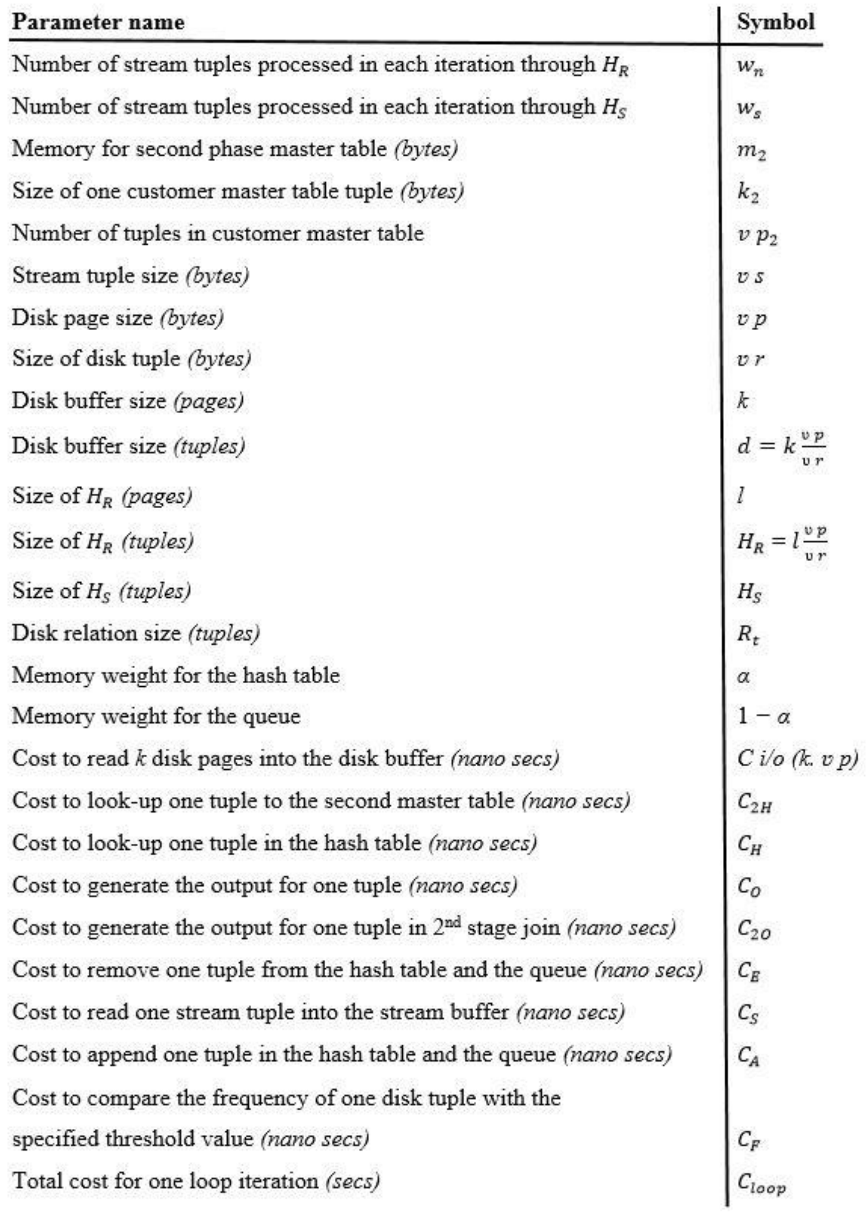
Cost to delete w_s tuples from H_S and the queue = $w_s \cdot C_E$

Total cost of the algorithm for one iteration can be calculated by using equation (4.2) by aggregating the above costs.

$$C_{loop} \text{ (secs)} = 10^{-9} \left[C\ i/o\ (k\ .\ vp) + d\ (\mathrm{C_H} + \mathrm{C_F}) + w_s\ (C_{2H} + \mathrm{C_{2O}} + \mathrm{C_E} + \mathrm{C_S} + \mathrm{C_A}) + w_n\ (\mathrm{C_H} + C_{2H} + \mathrm{C_{2O}} + \mathrm{C_S})\right] \quad (4.2)$$

Since in C_{loop} seconds the algorithm processes w_s and w_n tuples of the stream S, the service rate μ can be calculated using the below equation (4.3).

$$\mu = \frac{w_n + w_s}{\mathrm{C\ loop}} \quad (4.3)$$



Parameter name	**Symbol**
Number of stream tuples processed in each iteration through H_R	w_n
Number of stream tuples processed in each iteration through H_S	w_s
Memory for second phase master table *(bytes)*	m_2
Size of one customer master table tuple *(bytes)*	k_2
Number of tuples in customer master table	$v\ p_2$
Stream tuple size *(bytes)*	$v\ s$
Disk page size *(bytes)*	$v\ p$
Size of disk tuple *(bytes)*	$v\ r$
Disk buffer size *(pages)*	k
Disk buffer size *(tuples)*	$d = k\frac{v\ p}{v\ r}$
Size of H_R *(pages)*	l
Size of H_R *(tuples)*	$H_R = l\frac{v\ p}{v\ r}$
Size of H_S *(tuples)*	H_S
Disk relation size *(tuples)*	R_t
Memory weight for the hash table	α
Memory weight for the queue	$1 - \alpha$
Cost to read k disk pages into the disk buffer *(nano secs)*	$C\ i/o\ (k.\ v\ p)$
Cost to look-up one tuple to the second master table *(nano secs)*	C_{2H}
Cost to look-up one tuple in the hash table *(nano secs)*	C_H
Cost to generate the output for one tuple *(nano secs)*	C_O
Cost to generate the output for one tuple in 2nd stage join *(nano secs)*	C_{2O}
Cost to remove one tuple from the hash table and the queue *(nano secs)*	C_E
Cost to read one stream tuple into the stream buffer *(nano secs)*	C_S
Cost to append one tuple in the hash table and the queue *(nano secs)*	C_A
Cost to compare the frequency of one disk tuple with the specified threshold value *(nano secs)*	C_F
Total cost for one loop iteration *(secs)*	C_{loop}

Table 4.1 Notations used in cost calculations.

4.4 Experimental set-up

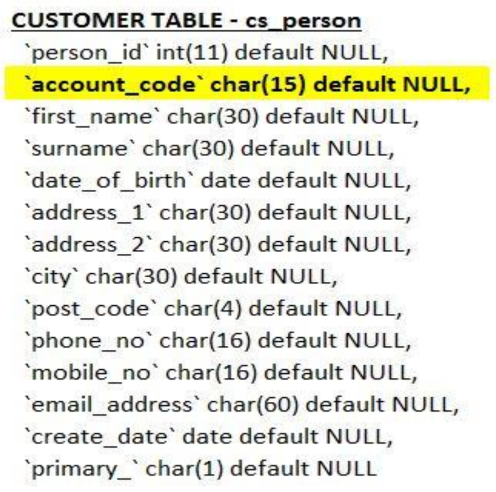
4.4.1 System set-up

We execute our Two-Stage join algorithm using the same platform specifications as in the normal CACHEJOIN algorithm.

In MITRE10 NZ, the customer master table is relatively smaller in size with 15 attributes and not more than 100,000 customer accounts. Because this implementation is mainly focused on MITRE 10 NZ which has smaller *cs_person* customer master table, we completely cached this table for second joining phase in Extended CACHEJOIN algorithm.

4.4.2 Real-life dataset

In this section we explain the datasets that we used to test our Two-Stage algorithm. As explained above incoming transactional stream data has two join attributes related to two different independent tables in the master data. The first table product is same as used in normal CACHEJOIN. The second table is called *cs_person* table. This table has 15 attributes for every customer/account from the table. The size of the product master table (*sc*) is set to 300,000 tuples and the size of the customer master table (*cs_person*) is set to 100,000 tuples. The customer master table attributes and data types are given in below Table 4.2. This table is used in 2nd-stage of our join algorithm



```
CUSTOMER TABLE - cs_person
  `person_id` int(11) default NULL,
  `account_code` char(15) default NULL,
  `first_name` char(30) default NULL,
  `surname` char(30) default NULL,
  `date_of_birth` date default NULL,
  `address_1` char(30) default NULL,
  `address_2` char(30) default NULL,
  `city` char(30) default NULL,
  `post_code` char(4) default NULL,
  `phone_no` char(16) default NULL,
  `mobile_no` char(16) default NULL,
  `email_address` char(60) default NULL,
  `create_date` date default NULL,
  `primary_` char(1) default NULL
```

Table 4.2 Data specification of MITRE 10 NZ customer table in master data.

4.5 Results

We run both the algorithms using the real-life dataset with two independent master tables called *sc* table and *cs_person* table holding the product key (*stock_code*) and the customer key (*account_code*) as primary keys respectively. We evaluated both the algorithms under different memory settings (50MB to 200MB). The algorithms are run without making any changes to any user-defined parameter values like disk_relation_size, page_size, threshold_value, etc as stated at the beginning of the CACHEJOIN.java file.

Figure 4.2 presents the performance (or service rate) of the both algorithms under all memory settings. . From the figure it can be observed that in case of Two-Stage join algorithm the service rate is slightly lower than that of normal CACHEJOIN which is plausible as in Two-Stage join algorithm stream tuples need to go through another join process before expiring them. Also from the figure we notice that there is a slight unusual behavior in the performance for the memory budget of 150MB. There is no difference on the experimental setup for this memory budget when compared to the other memory budgets. Hence it is interesting to see a slightly different behavior. This will lead us to investigate for the cause in our future research. The performance of our Two-Stage algorithm is still acceptable for the MITRE 10 NZ company according to their daily transactional volumes.

Also in Figure 4.2 we given the performance of CACHEJOIN using synthetic dataset. This gives us a visibility of how the attribute characteristics makes difference in the algorithm performance though size of the attributes are same as the real life dataset. Synthetic dataset is created using auto generated numeric values which are stored as a string whereas real life master table has a data with special characters and string with different sizes. This confirms that the performance of algorithm will vary in real life testing though the attribute sizes are made same in both synthetic and real life environment.

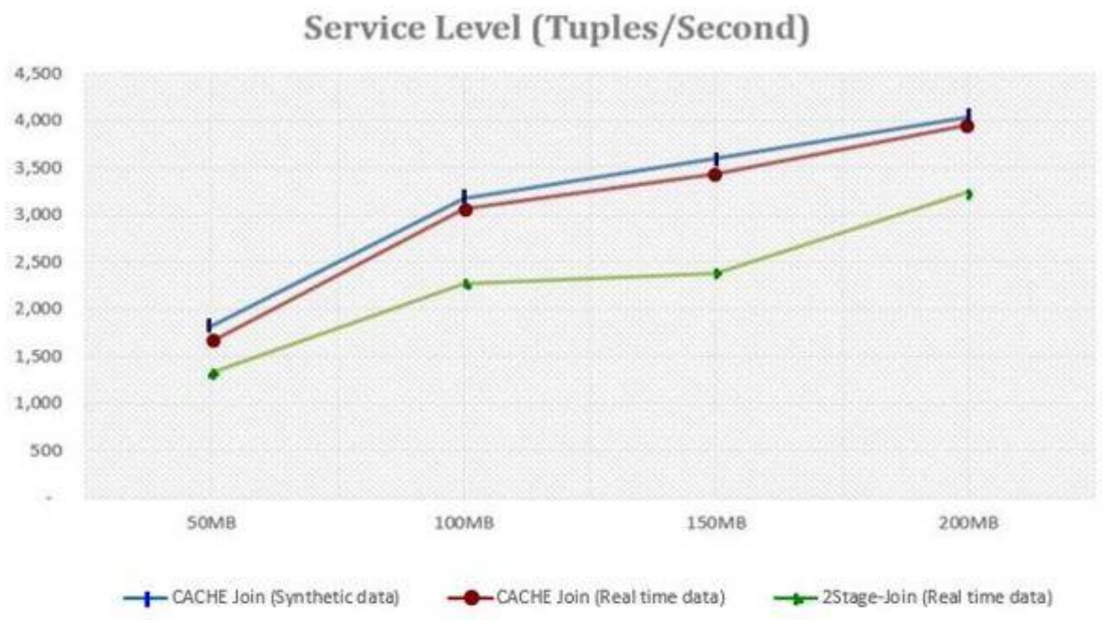


Figure 4.2 MITRE 10 NZ dataset service rate comparison between CACHEJOIN and two-stage CACHEJOIN semi-stream join.

4.6 Cost model validation

To add more value in the implementation, we carried out an experiment to validate our analytical cost with the empirical costs. Figure 4.3 presents the comparisons of both costs for each memory bucket of the algorithm. This graph is in a millisecond scale and shows that the calculated analytical cost is closely resembled with the empirical cost which validates our cost model. From the figure it is also observed that the total processing cost slightly increases by increasing the total allocated memory. The reason why there is a slight increase in the processing cost is because in all memory settings the I/O cost, which is the most dominant among all costs, doesn’t change due to the fixed size disk partitions.

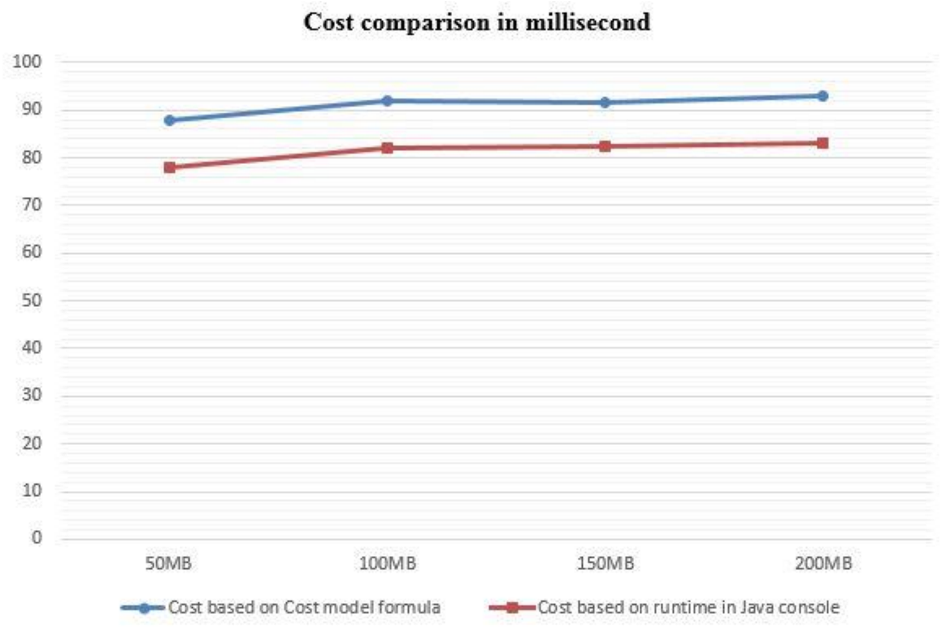


Figure 4.3 Cost comparison chart.

Chapter 5

5 Conclusions and Future work

5.1 Conclusions

In this thesis we implemented and evaluated a well-known semi-stream join algorithm called CACHEJOIN using a real-life transactional dataset by MITRE 10 NZ. An extensive literature review was done on a wide variety of semi-stream join algorithms to understand the characteristics of semi-stream join algorithms. And the MITRE 10 NZ transactional data distribution pattern is studied before choosing an appropriate algorithm for this research. The reason of choosing the CACHEJOIN algorithm was the long-tailed skew characteristics in MITRE 10 NZ transactional data. The thesis made following contributions.

Implementing of CACHEJOIN using MITRE 10 NZ dataset: CACHEJOIN was implemented and tested using MITRE 10 NZ dataset. The algorithm used *stock_code* as a join attribute in the both stream data and master data. CACHEJOIN consists of two phases: the stream-probing phase and the disk-probing-phase. A key feature in CACHEJOIN is a caching component. In stream-probing phase the algorithm uses a cache component to deal with this skew in stream data. Furthermore, the most frequently accessed master tuples were cached and were joined with the incoming stream data. In disk-probing phase the algorithm implements HYBRIDJOIN algorithm and deals with the rest of the master data on disk.

Extension of CACHEJOIN: Based on the nature of MITRE 10 NZ dataset we extended CACHEJOIN as a two-stage join. The stream data joined with two independent tables in the master data. The first table *sc* contains product information with *stock_code* as a primary key while the second table *cs_person* contains customers' accounts details with *account_code* as a primary key. In our two-stage join the first stage was normal CACHEJOIN while in second stage the algorithm only implements the stream-probing phase of the normal CACHEJOIN. According to our experimental evaluation the two-stage join worked well with the MITRE 10 NZ dataset. This provides a proof of concept to MITRE 10 NZ for their future implementation of the CACHEJOIN algorithm. In current state master data is well fitted into the memory but there can also be a possibility that master data never fits into the memory. However, we believe that our algorithm can handle this similar to a simple CACHEJOIN.

Deriving of cost model: We calculated analytical cost for our extended CACHEJOIN and validated this with our empirical cost. We observed that the empirical cost closely resembled the analytical cost which is the validation of our cost model.

Experimental study: We carried out the experimentations for our both CACHEJOIN and Extended CACHEJOIN algorithms and evaluated the performance using both synthetic and the real-life datasets. The experiments are performed using different memory budgets. Under all memory settings the CACHEJOIN algorithm and Extension of CACHEJOIN algorithm performed well. Hence we conclude that CACHEJOIN algorithm can handle the volume of MITRE10 NZ dataset with adequate performance.

5.2 Future work

The extension work in semi-stream CACHEJOIN which is a two-stage CACHEJOIN algorithm opens more interesting ideas for the future work in this area. In this work an index joining method for a second stage join implemented in CACHEJOIN was advanced. Taking this as a basis, using different semi-stream joins such as HYBRIDJOIN, CACHEJOIN, MESHJOIN, etc. could be trialed to perform a second stage join and monitored to see how these behave in regard to the performance of service rates. The size of the second master table could produce a difference in performance.

MITRE 10 NZ is a leading retail store in New Zealand and the amount of transactions occurring in a day is expected to grow in future. To handle the growing amount of transactions (incoming stream data) trying parallelization running two nodes, could be attempted to see how it reduces queue. It could be expected that with a higher number of incoming streams joining in the same time period, the same service rate as CACHEJOIN could be achieved as it will eventually perform two joins simultaneously (one on each node).

References

Anderson, C. (2006). *The long tail: Why the future of business is selling less of more.* New York: Hyperion.

Arasu, A., Shivnath, B. & Jennifer, W. (2002). *An abstract semantics and concrete language for continuous queries over streams and relations.* Stanford InfoLab.

Asif Naeem, M., Gillian, D. & Gerald, W. (2008). An event-based near real-time data integration architecture. *Enterprise Distributed Object Computing Conference Workshops, IEEE,* 401-404, doi:10.1109/EDOCW.2008.14.

Asif Naeem, M., Gillian, D., Gerald, W. & Shafiq, A. (2010). R-MESHJOIN for near-real-time data warehousing. *DOLAP '10 Proceedings of the ACM 13th intl. workshop on Data warehousing and OLAP, ACM,* 53-60, doi:10.1145/1871940.1871952.

Asif Naeem, M., Gillian, D. & Gerald, W. (2011a). HYBRIDJOIN for near-real-time data warehousing. *International Journal of Data Warehousing and Mining, vol. 7(4),* (21-42). IGI Global.

Asif Naeem, M., Gillian, D. & Gerald, W. (2011b). *X-HYBRIDJOIN for near-real-time data warehousing*. Berlin: Springer-Verlag, *vol-7051*, 33-47.

Asif Naeem, M., Gillian, D., Gerald, W. & Imran, S. B. (2012a). *Efficient usage of memory resources in near-real-time data warehousing*. Berlin: Springer-Verlag, *vol-281*, 326-337.

Asif Naeem, M., Gillian, D. & Gerald, W. (2012b). *A lightweight stream-based join with limited resource consumption*. Berlin: Springer-Verlag , *vol-7448,* 431-442.

Asif Naeem, M., Gillian, D. & Gerald, W. (2012c). Optimised X-HYBRIDJOIN for near-real-time data warehousing. *Proceedings of the Twenty-Third Australasian Database Conference, vol-124,* (21-30). Melbourne: Australian Computer Society, Inc.

Asif Naeem, M., Gillian, D., Gerald, W. (2013a). & Christof, L. A generic front-stage for semi-stream processing. *CIKM '13 Proceedings of the 22nd ACM international conference on Information & Knowledge Management, ACM*, 769-774, doi:10.1145/2505515.2505734.

Asif Naeem, M., Gerald, W., Gillian, D. & Christof, L. (2013b). SSCJ: A semi-stream cache join using a front-stage cache module. Berlin: Springer-Verlag, *vol-8057*, 236-247

Asif Naeem, M. (2014). A caching approach to process stream data in data warehouse. *Digital Information Management (ICDIM), 2014 Ninth International Conference, IEEE,* 162-167, doi:10.1109/ICDIM.2014.6991406.

Tom, B., Robert, B., Jim, G. & Prakash, S. (1994). Loading databases using dataflow parallelism. *ACM SIGMOD Record, vol-23,* 1 - 13, doi:10.1145/190627.190647.

Mihaela, A. B., Antonios, D. Yannis, K., Vasilis, V. & Athens, U. (2011). Semi-streamed index join for near-real time execution of etl transformations. *Data Engineering (ICDE), IEEE*, 159 - 170, doi:10.1109/ICDE.2011.5767906.

Burleson, D. (2004). *New developments in oracle data warehousing.* (City): Burleson Consulting.

Abhirup, C. & Ajit, S. (2009). A partition-based approach to support streaming updates over persistent data in an active aata warehouse. *Parallel & Distributed Processing, IEEE,* 1 - 11, doi:10.1109/IPDPS.2009.5161064.

Surajit, C., Umeshwar, D. & Vivek, N. (2011). An overview of business intelligence technology. *ACM, vol-54,* 88-98.

Lukasz, G., Theodore, J., Spencer, S. J. & Vladislav, S. (2009). Stream warehousing with datadepot. *SIGMOD '09 Proceedings of the 2009 ACM SIGMOD International Conference on Management of data , ACM,* 847 - 853, doi:10.1145/1559845.1559934.

Goetz, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR), vol-25,* 73 - 120, doi:10.1145/152610.152611.

Hahn, C. J., Warren, S. G. & London, J. (1996). *Edited synoptic cloud reports from ships and land stations over the globe, 1983 - 1991, Washington: United States*

Alexandros, K., Panos, V. & Evaggelia, P. (2005). ETL queues for active data warehousing. *IQIS '05 Proceedings of the 2nd international workshop on Information quality in information systems, ACM,* 28 - 39, doi:10.1145/1077501.1077509.

Ralph, K. & Joe, C. (2004). *The data warehouse ETL toolkit.* City: Wiley Publishing, Inc.

On-Time Data Warehousing with Oracle10g-Information at the Speed of Your Business. (2003, August). Oracle Corporation White Paper, 2003.

Neoklis, P., Spiros, S., Panos, V., Alkis, S. & Nils-Erik, F. (2007). Supporting streaming updates in an active data warehouse. *Data Engineering, IEEE,* 476-485, doi:10.1109/ICDE.2007.367893.

Neoklis, P., Spiros, S., Panos, V., Alkis, S. & Nils-Erik, F. (2008). Meshing streaming updates with persistent data in an active data warehouse. *Knowledge and Data Engineering, IEEE,* 976-991, doi:10.1109/TKDE.2008.27.

Roussopoulos, N., Kotidis, Y. & Roussopoulos, M. (1997). Cubetree: Organization of and bulk incremental updates on the data cube. *SIGMOD '97 Proceedings of the 1997 ACM SIGMOD international conference on Management of data, ACM, vol-26,* 89 - 99, doi:10.1145/253262.253276.

Leonard, D. S. (1986). Join processing in database systems with large main memories. *ACM, vol-11*, 239 - 264, doi:10.1145/6314.6315.

Annita, N. W. & Peter, M. G. A. (1990). Pipelining in query execution. *IEEE,* 562.

Eugene, W., Yanlei, D. & Shariq, R. (2006). High-performance complex event processing over streams. *SIGMOD '06 Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM,* 407 - 418. doi:10.1145/1142473.1142520.

Appendix A - CACHEJOIN Semi-Stream algorithm Java codes

In this implementation a few java classes are created to support running the algorithm successfully. Below are the java class files which are most important in this implementation.

CACHEJOIN.java code

```
package cacheJoinSource;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.HashSet;
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.DriverManager;
import org.apache.commons.collections15.MultiMap;
import org.apache.commons.collections15.multimap.MultiHashMap;
import sizeof.agent.SizeOfAgent;

public class CACHEJOIN {
      public static final int HASH_SIZE=35131;
      public static final int QUEUE_SIZE=HASH_SIZE;
      public static final int STREAM_SIZE=5000;
      public static final int DISK_RELATION_SIZE=300000;
      public static final int SWAP_DB=1000;
      public static final int MIN_KEY=1;
      public static final int MAX_KEY=DISK_RELATION_SIZE;
      public static final int THRESHOLD=3;
      public static int cs_person_match=0;
      public static final int MEASUREMENT_START=1;
      public static final int MEASUREMENT_STOP=DISK_RELATION_SIZE;

      static MultiMap<String,HybridJoinObject> mhm=new
MultiHashMap<String,HybridJoinObject>();
      static ArrayList <HybridJoinObject> list=new
ArrayList<HybridJoinObject>();
      static HashSet<String> accountCodeSet = new HashSet<String>(100000);
      static LinkedBlockingQueue<HybridJoinObject> streamBuffer=new
LinkedBlockingQueue<HybridJoinObject>();
      static String diskBuffervolatile[][]=new String[SWAP_DB][86];
      static int frequencyDetector[]=new int[SWAP_DB];

      Random myRandom=new Random();
      static Statement stmt=null;
      static ResultSet rs=null;
      Queue head,currentNode,deleteNodeAddress;
      static DiskHashTableManipulation dhtm=null;
```

```
	String streamRandomValue;
	int requiredTuplesCount=0,non_vola=0,vola=0;
	static int tuplesMatchedIntoDiskHash=0;
	static long CE[]= new long[DISK_RELATION_SIZE/100];
	static long CS[]= new long[DISK_RELATION_SIZE/100];
	static long CA[]= new long[DISK_RELATION_SIZE/100];
	static long CIO[]= new long[DISK_RELATION_SIZE/100];
	static long CH[]= new long[DISK_RELATION_SIZE/100];
	static long C2H[]= new long[DISK_RELATION_SIZE/100];
	static long CF[]= new long[DISK_RELATION_SIZE/100];
	static int streamInputSize[]=new int[DISK_RELATION_SIZE/100];
	static int StreamSizeMatchedInDiskHash[]=new
int[DISK_RELATION_SIZE/100];
	static int
accesed_page,CE_index=0,CS_index=0,CA_index=0,CIO_index=0,C2H_index=0,CH_in
dex=0,pt_index=0,input_index=0,queue_index=0,rt_index=0,bl_index=0,WT_index
=0,CF_index=0;
	float oneNodeSize=0,memoryForFiftyTuples=0;
	boolean measurementStart=false;
	double sumOfFrequency,random,rawFK,minimumLimit;

	int ir=52124,increment=12458,prime=2000003;

	String fileName = "Account_"+new
SimpleDateFormat("yyyyMMddhhmmss'.txt'").format(new Date());
	File accountFile = new File(fileName);
	FileWriter accountFileWriter;

	CACHEJOIN()throws java.io.IOException{
		for(int i=0; i<frequencyDetector.length; i++){
			frequencyDetector[i]=0;
		}
	}

	public Connection connectDB(){
		Connection conn=null;
		try{

			String userName = "root";
			String password = "root";
			String url = "jdbc:mysql://localhost/masterdata";
			Class.forName ("com.mysql.jdbc.Driver");
			conn = DriverManager.getConnection (url, userName,
password);
			System.out.println("Connected to Database");
		}
		catch (Exception e)
		{
			System.err.println (e);
		}
		return conn;
	}

	public void closeConnection(Connection conn){
		try{
			if(conn!=null){
				conn.close();
				System.out.println("Database connection closed");
			}
		}catch (SQLException e)
		{
```

```
                        System.err.println (e);
                }
        }

        public static double integral(double limit){
                return (Math.log(limit));           //Exponent x^1
        }

        public static double inverseIntegral(double x){
                return Math.exp(x);    //Inverse integral of exponent 1
        }


        public void fillHashTable(){

                int tuples=0;
                ArrayBlockingQueue<TransactionDAO> transactionList =
Transaction.fetchTransactionRecords();
                if(transactionList.isEmpty()){
                        transactionList = Transaction.fetchTransactionRecords();
                }
                TransactionDAO tDAO = transactionList.poll();
                streamRandomValue = tDAO.getStock();
                head=new Queue(streamRandomValue);
                currentNode=head;
                mhm.put(streamRandomValue,new
HybridJoinObject(tDAO.getBranch(),tDAO.getDept(),tDAO.getDrawer(),tDAO.getC
ode(),tDAO.getAmount(),tDAO.getCost(),tDAO.getDate_(),tDAO.getAccount(),tDA
O.getRemarks(),tDAO.getReference(),

        tDAO.getType_(),tDAO.getTax(),tDAO.getOperator(),tDAO.getQty(),tDAO.g
etStock(),tDAO.getDocket_no(),tDAO.getPosted(),tDAO.getPromo(),tDAO.getSub_
code(),tDAO.getGst(),

        tDAO.getPromo_num(),tDAO.getPromo_start(),tDAO.getPromo_end(),tDAO.ge
tDescription(),tDAO.getGroup_(),tDAO.getRetail(),tDAO.getSys_price(),tDAO.g
etSys_disc(),tDAO.getSet_disc_val(),tDAO.getAct_price(),

        tDAO.getAct_disc(),tDAO.getAct_disc_val(),tDAO.getLine(),tDAO.getUnit
(),tDAO.getLength(),tDAO.getTill(),tDAO.getDate2post(),tDAO.getSub_account(
),tDAO.getItem_type(),tDAO.getSub_type(),

                tDAO.getTime(),tDAO.getOverride_operator(),currentNode));
                oneNodeSize=SizeOfAgent.fullSizeOf(head);
                while(tuples<HASH_SIZE){
                        if(transactionList.isEmpty()){
                                transactionList =
Transaction.fetchTransactionRecords();
                        }
                         tDAO = transactionList.poll();
                         streamRandomValue = tDAO.getStock();
                         System.out.println(" Tuples Value " + tuples);
                                currentNode=currentNode.addNode(streamRandomValue);
                                mhm.put(streamRandomValue,new
HybridJoinObject(tDAO.getBranch(),tDAO.getDept(),tDAO.getDrawer(),tDAO.getC
ode(),tDAO.getAmount(),tDAO.getCost(),tDAO.getDate_(),tDAO.getAccount(),tDA
O.getRemarks(),tDAO.getReference(),

        tDAO.getType_(),tDAO.getTax(),tDAO.getOperator(),tDAO.getQty(),tDAO.g
etStock(),tDAO.getDocket_no(),tDAO.getPosted(),tDAO.getPromo(),tDAO.getSub_
code(),tDAO.getGst(),
	tDAO.getPromo_num(),tDAO.getPromo_start(),tDAO.getPromo_end(),tDAO.ge
tDescription(),tDAO.getGroup_(),tDAO.getRetail(),tDAO.getSys_price(),tDAO.g
etSys_disc(),tDAO.getSet_disc_val(),tDAO.getAct_price(),

	tDAO.getAct_disc(),tDAO.getAct_disc_val(),tDAO.getLine(),tDAO.getUnit
(),tDAO.getLength(),tDAO.getTill(),tDAO.getDate2post(),tDAO.getSub_account(
),tDAO.getItem_type(),tDAO.getSub_type(),


	tDAO.getTime(),tDAO.getOverride_operator(),currentNode));
				tuples++;
				if(tuples==49){

	memoryForFiftyTuples=SizeOfAgent.fullSizeOf(mhm);

				}
		}

	}

	public boolean probIntoHash(){

		long start=0,stop=0,
joinStart=0,joinStop=0,CH_per_Iteration=0,C2H_per_Iteration=0,CEH_per_Itera
tion=0,CEQ_per_Iteration=0,CF_per_iteration=0;
		boolean firstNode=false,lastNode=false,tupleInMD=true;
		int
processedTuplesCount=0,hashProbCount=0,detectedTupleCount=0;
		int index=new Double(head.popNode()).intValue();
		tupleInMD=readDBvolatilePage(index);

		if(tupleInMD){
			//Probing of disk buffer
			for(int row=0; row<SWAP_DB; row++){
				if(mhm.containsKey(diskBuffervolatile[row][0])){
					start=System.nanoTime();


	list=(ArrayList<HybridJoinObject>)mhm.get(diskBuffervolatile[row][0])
;
					joinStart=System.nanoTime();
					for(HybridJoinObject hjo : list){
						String account_code = hjo.attr8;

	if(!accountCodeSet.contains(account_code)){
							System.out.println(" Account Code
" + account_code + " Not Found");
						}
					}
					joinStop=System.nanoTime();

					stop=System.nanoTime();

					hashProbCount++;
					if(measurementStart){
						C2H_per_Iteration+=joinStop-joinStart;
						CH_per_Iteration+=stop-start;
					}
					start=System.nanoTime();
					mhm.remove(diskBuffervolatile[row][0]);
```

```
                                        stop=System.nanoTime();
                                        if(measurementStart){
                                                CEH_per_Iteration+=stop-start;
                                        }
                                        for(int listItem=0; listItem<list.size();
listItem++){
                                                firstNode=false;
                                                lastNode=false;

        deleteNodeAddress=list.get(listItem).nodeAddress;
                                                if(deleteNodeAddress==head){
                                                        head=deleteNodeAddress.getNext();
                                                        firstNode=true;
                                                }
                                                if(deleteNodeAddress==currentNode){

        currentNode=deleteNodeAddress.getPrecede();
                                                        lastNode=true;
                                                }
                                                start=System.nanoTime();

        deleteNodeAddress.deleteNode(firstNode,lastNode);
                                                stop=System.nanoTime();
                                                if(measurementStart){
                                                        CEQ_per_Iteration+=stop-start;
                                                        vola++;
                                                }
                                                frequencyDetector[row]++;
                                                requiredTuplesCount++;
                                        }

                                        if(measurementStart){

        CEH_per_Iteration+=CEQ_per_Iteration/list.size();
                                                CEQ_per_Iteration=0;
                                        }
                                        processedTuplesCount++;
                                }
                        }
                        start=System.nanoTime();
                        for(int row=0; row<SWAP_DB; row++){
                                if(frequencyDetector[row]>=THRESHOLD &&
DiskHashTableManipulation.dmhm.size()<DiskHashTableManipulation.NON_SWAP_DB
){


        DiskHashTableManipulation.dmhm.put(diskBuffervolatile[row][0], new
HybridJoinDiskObject(diskBuffervolatile[row][0],diskBuffervolatile[row][1],

        diskBuffervolatile[row][2],diskBuffervolatile[row][3],diskBuffervolat
ile[row][4],diskBuffervolatile[row][5],diskBuffervolatile[row][6],

        diskBuffervolatile[row][7],diskBuffervolatile[row][8],diskBuffervolat
ile[row][9],diskBuffervolatile[row][10],diskBuffervolatile[row][11],

        diskBuffervolatile[row][12],diskBuffervolatile[row][13],diskBuffervol
atile[row][14],diskBuffervolatile[row][15],diskBuffervolatile[row][16],

        diskBuffervolatile[row][17],diskBuffervolatile[row][18],diskBuffervol
atile[row][19],diskBuffervolatile[row][20],diskBuffervolatile[row][21],
	diskBuffervolatile[row][22],diskBuffervolatile[row][23],diskBuffervol
atile[row][24],diskBuffervolatile[row][25],diskBuffervolatile[row][26],

	diskBuffervolatile[row][27],diskBuffervolatile[row][28],diskBuffervol
atile[row][29],diskBuffervolatile[row][30],diskBuffervolatile[row][31],

	diskBuffervolatile[row][32],diskBuffervolatile[row][33],diskBuffervol
atile[row][34],diskBuffervolatile[row][35],diskBuffervolatile[row][36],

	diskBuffervolatile[row][37],diskBuffervolatile[row][38],diskBuffervol
atile[row][39],diskBuffervolatile[row][40],diskBuffervolatile[row][41],

	diskBuffervolatile[row][42],diskBuffervolatile[row][43],diskBuffervol
atile[row][44],diskBuffervolatile[row][45],diskBuffervolatile[row][46],

	diskBuffervolatile[row][47],diskBuffervolatile[row][48],diskBuffervol
atile[row][49],diskBuffervolatile[row][50],diskBuffervolatile[row][51],

	diskBuffervolatile[row][52],diskBuffervolatile[row][53],diskBuffervol
atile[row][54],diskBuffervolatile[row][55],diskBuffervolatile[row][56],

	diskBuffervolatile[row][57],diskBuffervolatile[row][58],diskBuffervol
atile[row][59],diskBuffervolatile[row][60],diskBuffervolatile[row][61],

	diskBuffervolatile[row][62],diskBuffervolatile[row][63],diskBuffervol
atile[row][64],diskBuffervolatile[row][65],diskBuffervolatile[row][66],

	diskBuffervolatile[row][67],diskBuffervolatile[row][68],diskBuffervol
atile[row][69],diskBuffervolatile[row][70],diskBuffervolatile[row][71],

	diskBuffervolatile[row][72],diskBuffervolatile[row][73],diskBuffervol
atile[row][74],diskBuffervolatile[row][75],diskBuffervolatile[row][76],

	diskBuffervolatile[row][77],diskBuffervolatile[row][78],diskBuffervol
atile[row][79],diskBuffervolatile[row][80],diskBuffervolatile[row][81],

	diskBuffervolatile[row][82],diskBuffervolatile[row][83],diskBuffervol
atile[row][84],diskBuffervolatile[row][85]));
					detectedTupleCount++;
				}
				frequencyDetector[row]=0;
			}
			stop=System.nanoTime();

			if(measurementStart){
				CF[CF_index++]=stop-start;
				CH[CH_index++]=CH_per_Iteration/hashProbCount;

	CE[CE_index++]=CEH_per_Iteration/processedTuplesCount;
				C2H[C2H_index++]=C2H_per_Iteration/hashProbCount;
			}
		}
		return tupleInMD;
	}

	public static void cacheAccountCode() {

			try{
						rs = stmt.executeQuery("select
account_code from cs_person");
```

```
                                while(rs.next()){

    accountCodeSet.add(rs.getString("account_code"));
                                }
                    }
                    catch(Exception e){
                          e.printStackTrace();
                    }
                    }


    public boolean readDBvolatilePage(int index){
          int row=0,PageStart;
          long start=0,stop=0;
          boolean firstNode=false,lastNode=false,tupleInMD=true;
          //Loading of disk buffer
          try{
                start=System.nanoTime();
                rs=stmt.executeQuery("Select stock_code FROM sc WHERE
stock_code='"+index+"'");

                if(!rs.next()){
                      list=(ArrayList<HybridJoinObject>)mhm.get(index);
                      mhm.remove(index);
                      for(int listItem=0; listItem<list.size();
listItem++){
                            firstNode=false;
                            lastNode=false;

    deleteNodeAddress=list.get(listItem).nodeAddress;
                            if(deleteNodeAddress==head){
                                  head=deleteNodeAddress.getNext();
                                  firstNode=true;
                            }
                            if(deleteNodeAddress==currentNode){

    currentNode=deleteNodeAddress.getPrecede();
                                  lastNode=true;
                            }

    deleteNodeAddress.deleteNode(firstNode,lastNode);
                      }
                      tupleInMD=false;
                }
                else{

                      PageStart=rs.getInt(1);
                      rs=stmt.executeQuery("SELECT * from sc where
stock_code >='" + PageStart +"' LIMIT " + SWAP_DB);
                      stop=System.nanoTime();
                      if(measurementStart){
                            CIO[CIO_index++]=stop-start;
                      }
                      while(rs.next()){
                            for(int col=1; col<=86; col++){
                                  diskBuffervolatile[row][col-
1]=rs.getString(col);
                            }
                            row++;
                      }
                }
```

```
            }catch(SQLException e){System.out.print(e);}
            return tupleInMD;
    }

    public void appendHash(){
            long start=0,stop=0,CA_per_Iteration=0;
            int eachInputSize=0;
            while(streamBuffer.size()<requiredTuplesCount*3);
            tuplesMatchedIntoDiskHash=0;
            while (requiredTuplesCount>0){
                    try {
                    if(dhtm.matchedIntoDiskHash(streamBuffer.peek().attr15)){
                            streamBuffer.poll();
                            tuplesMatchedIntoDiskHash++;
                    }
                    else{
                            start=System.nanoTime();

    currentNode=currentNode.addNode(streamBuffer.peek().attr15);
                            mhm.put(streamBuffer.peek().attr15,new
HybridJoinObject(streamBuffer.peek().attr1,streamBuffer.peek().attr2,stream
Buffer.peek().attr3,streamBuffer.peek().attr4,streamBuffer.peek().attr5,str
eamBuffer.peek().attr6,streamBuffer.peek().attr7,streamBuffer.peek().attr8,
streamBuffer.peek().attr9,streamBuffer.peek().attr10,

    streamBuffer.peek().attr11,streamBuffer.peek().attr12,streamBuffer.pe
ek().attr13,streamBuffer.peek().attr14,streamBuffer.peek().attr15,streamBuf
fer.peek().attr16,streamBuffer.peek().attr17,streamBuffer.peek().attr18,str
eamBuffer.peek().attr19,streamBuffer.peek().attr20,

    streamBuffer.peek().attr21,streamBuffer.peek().attr22,streamBuffer.pe
ek().attr23,streamBuffer.peek().attr24,streamBuffer.peek().attr25,streamBuf
fer.peek().attr26,streamBuffer.peek().attr27,streamBuffer.peek().attr28,str
eamBuffer.peek().attr29,streamBuffer.peek().attr30,

    streamBuffer.peek().attr31,streamBuffer.peek().attr32,streamBuffer.pe
ek().attr33,streamBuffer.peek().attr34,streamBuffer.peek().attr35,streamBuf
fer.peek().attr36,streamBuffer.peek().attr37,streamBuffer.peek().attr38,str
eamBuffer.peek().attr39,streamBuffer.peek().attr40,

    streamBuffer.peek().attr41,streamBuffer.peek().attr42,currentNode));
                            streamBuffer.poll();
                            stop=System.nanoTime();
                            if(measurementStart){
                                    CA_per_Iteration+=stop-start;
                            }
                            requiredTuplesCount--;
                            eachInputSize++;
                    }
                    }
                    catch(Exception e){
                            continue;
                    }
            }

            if(measurementStart){
                    CA[CA_index++]=CA_per_Iteration/eachInputSize;
                    streamInputSize[input_index]=eachInputSize;

    StreamSizeMatchedInDiskHash[input_index++]=tuplesMatchedIntoDiskHash;
            }
```

```
      }

      public static void main(String args[])throws java.io.IOException,
InterruptedException{

            CACHEJOIN hj=new CACHEJOIN();
            StartUpdatesStream stream=new StartUpdatesStream();
            dhtm=new DiskHashTableManipulation();
            boolean tupleInMD=true;
            System.out.println("Hybrid Join in execution mode...");
            Connection conn=hj.connectDB();
            hj.accountFile.createNewFile();
            hj.accountFileWriter = new FileWriter(hj.accountFile);
            try{
                  CACHEJOIN.stmt=conn.createStatement();
                  CACHEJOIN.stmt.setFetchSize(SWAP_DB);
                  System.out.println("Fetch Size:
"+CACHEJOIN.stmt.getFetchSize());
                  CACHEJOIN.cacheAccountCode();
            }catch(SQLException e){System.out.print(e);}

            hj.fillHashTable();
            stream.start();
            Thread.sleep(2000);
            for(int round=1; round<=4; round++){
                  if(round==1){
                        System.out.println("ROUND 1 Started...");
                        System.out.println("Disk hash tuple:
"+DiskHashTableManipulation.dmhm.size());
                        //Thread.sleep(700);
                  }
                  if(round==4){
                        System.out.println("ROUND 2 Started...");
                        System.out.println("Disk hash tuple:
"+DiskHashTableManipulation.dmhm.size());
                        //Thread.sleep(700);
                  }
                  for(int tuple=1; tuple<=DISK_RELATION_SIZE;
tuple+=SWAP_DB){
                        SimpleDateFormat smd=new SimpleDateFormat("dd-MM-
yyyy hh:mm:ss ");
                        Date currentDate=new Date();
                        System.out.println(" Iteration count " + tuple + "
out of " + DISK_RELATION_SIZE+"  Date "+smd.format(currentDate)+" Time
"+currentDate.getTime());
                        hj.measurementStart=false;
                        if((round==4)){
                              hj.measurementStart=true;
                        }
                        tupleInMD=hj.probIntoHash();
                        if(tupleInMD){
                              hj.appendHash();
                        }
                  }
            }
            stream.~~stop~~();
            hj.accountFileWriter.close();
            System.out.println("Hash tuples: "+mhm.size());
            System.out.println("Disk hash tuple:
"+DiskHashTableManipulation.dmhm.size());
            System.out.println("\nMEMORY COST");
            float Hash=(HASH_SIZE*(hj.memoryForFiftyTuples/50))/1048576f;
            float Queue=(hj.oneNodeSize*QUEUE_SIZE)/1048576f;
            float bufferW=SizeOfAgent.fullSizeOf(streamBuffer)/1048576f;
            float
bufferb=SizeOfAgent.fullSizeOf(diskBuffervolatile)/1048576f;
            float total=Hash+Queue+bufferW+bufferb;

            System.out.println(" Total Account Id Matches " +
cs_person_match);
            System.out.println("Memory used by Hash Table:  "+Hash+"  MB");
            System.out.println("Memory used by Queue:  "+Queue+" MB");
            System.out.println("Memory used by Stream buffer:  "+bufferW+"
MB");
            System.out.println("Memory used by buffer b :  "+bufferb+"
MB");
            System.out.println("Total Memory: "+total+" MB");
            System.out.println("pt_index"+hj.pt_index+" C2H_index:
"+hj.C2H_index+" CH_index: "+hj.CH_index+" CS_Index: "+hj.CS_index+"
CA_index: "+hj.CA_index+" CE_index: "+hj.CE_index+" CIO_index:
"+hj.CIO_index+"CF_index+ "+hj.CF_index);
            hj.closeConnection(conn);

            System.out.println("Queue status:"+hj.head.countNodes());
            System.out.println("Non Volatile: "+hj.non_vola);
            System.out.println("Volatile: "+hj.vola);
            long today = Calendar.getInstance().getTimeInMillis();
            BufferedWriter bw=new BufferedWriter(new
FileWriter("C://Users//Daniel//Desktop//vinod_cachejoin//Semi-Stream-
Joins"+today+".txt"));

            bw.write("Geralized CACHEJOIN PROCESSING COST");
            bw.newLine();

            bw.write("Total w\t w processed by disk Hash\t  w processed by
disk buffer\t     C2H(NSec)\t     CH(NSec)\t     CS(NSec)\t     CA(NSec)\t
CF\t   CE(NSec)\t     CIO(NSec)");
            bw.newLine();

            for(int i=0; i<CACHEJOIN.CIO_index; i++){

      bw.write((CACHEJOIN.StreamSizeMatchedInDiskHash[i]+CACHEJOIN.streamIn
putSize[i])+"\t\t");

      bw.write(CACHEJOIN.StreamSizeMatchedInDiskHash[i]+"\t\t");
                  bw.write(CACHEJOIN.streamInputSize[i]+"\t\t");
                  bw.write(CACHEJOIN.C2H[i]+"\t\t");
                  bw.write(CACHEJOIN.CH[i]+"\t\t");
                  bw.write(CACHEJOIN.CS[i]+"\t\t");
                  bw.write(CACHEJOIN.CA[i]+"\t\t");
                  bw.write(CACHEJOIN.CF[i]+"\t\t");
                  bw.write(CACHEJOIN.CE[i]+"\t\t");
                  bw.write(CACHEJOIN.CIO[i]+"");
                  bw.newLine();
            }

            bw.close();
            System.out.println("\nExecution has been completed");
      }
}
```

TRANSACTION.java code

```
package cacheJoinSource;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.concurrent.ArrayBlockingQueue;

public class Transaction {

      static int lastOffset=0;
      static int tuppleSize=1000;
      static int maxRowSize=50000;

      public static ArrayBlockingQueue<TransactionDAO>
fetchTransactionRecords() {
            ArrayBlockingQueue<TransactionDAO> transactionList = new
ArrayBlockingQueue<TransactionDAO>(tuppleSize);
            Connection conn = connectDB();
            // make sure autocommit is off
            try {
                  conn.setAutoCommit(false);

                  if(lastOffset==maxRowSize){
                        // Resetting Last Offset
                        lastOffset=0;
                  }

            Statement st = conn.createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM gltx LIMIT " +
Transaction.lastOffset + ", "+Transaction.tuppleSize);
            while(rs.next()){
                  TransactionDAO tDAO = new TransactionDAO();

                  tDAO.setBranch(rs.getString("branch"));
                  tDAO.setDept(rs.getString("dept"));
                  tDAO.setDrawer(rs.getString("drawer"));
                  tDAO.setCode(rs.getString("code"));
                  tDAO.setAmount(rs.getString("amount"));
                  tDAO.setCost(rs.getString("cost"));
                  tDAO.setDate_(rs.getString("date_"));
                  tDAO.setAccount(rs.getString("account"));
                  tDAO.setRemarks(rs.getString("remarks"));
                  tDAO.setReference(rs.getString("reference"));
                  tDAO.setType_(rs.getString("type_"));
                  tDAO.setTax(rs.getString("tax"));
                  tDAO.setOperator(rs.getString("operator"));
                  tDAO.setQty(rs.getString("qty"));
                  tDAO.setStock(rs.getString("stock"));
                  tDAO.setDocket_no(rs.getString("docket_no"));
                  tDAO.setPosted(rs.getString("posted"));
                  tDAO.setPromo(rs.getString("promo"));
                  tDAO.setSub_code(rs.getString("sub_code"));
                  tDAO.setGst(rs.getString("gst"));
                  tDAO.setPromo_num(rs.getString("promo_num"));
                  tDAO.setPromo_start(rs.getString("promo_start"));
                  tDAO.setPromo_end(rs.getString("promo_end"));
                  tDAO.setDescription(rs.getString("description"));
                  tDAO.setGroup_(rs.getString("group_"));
```

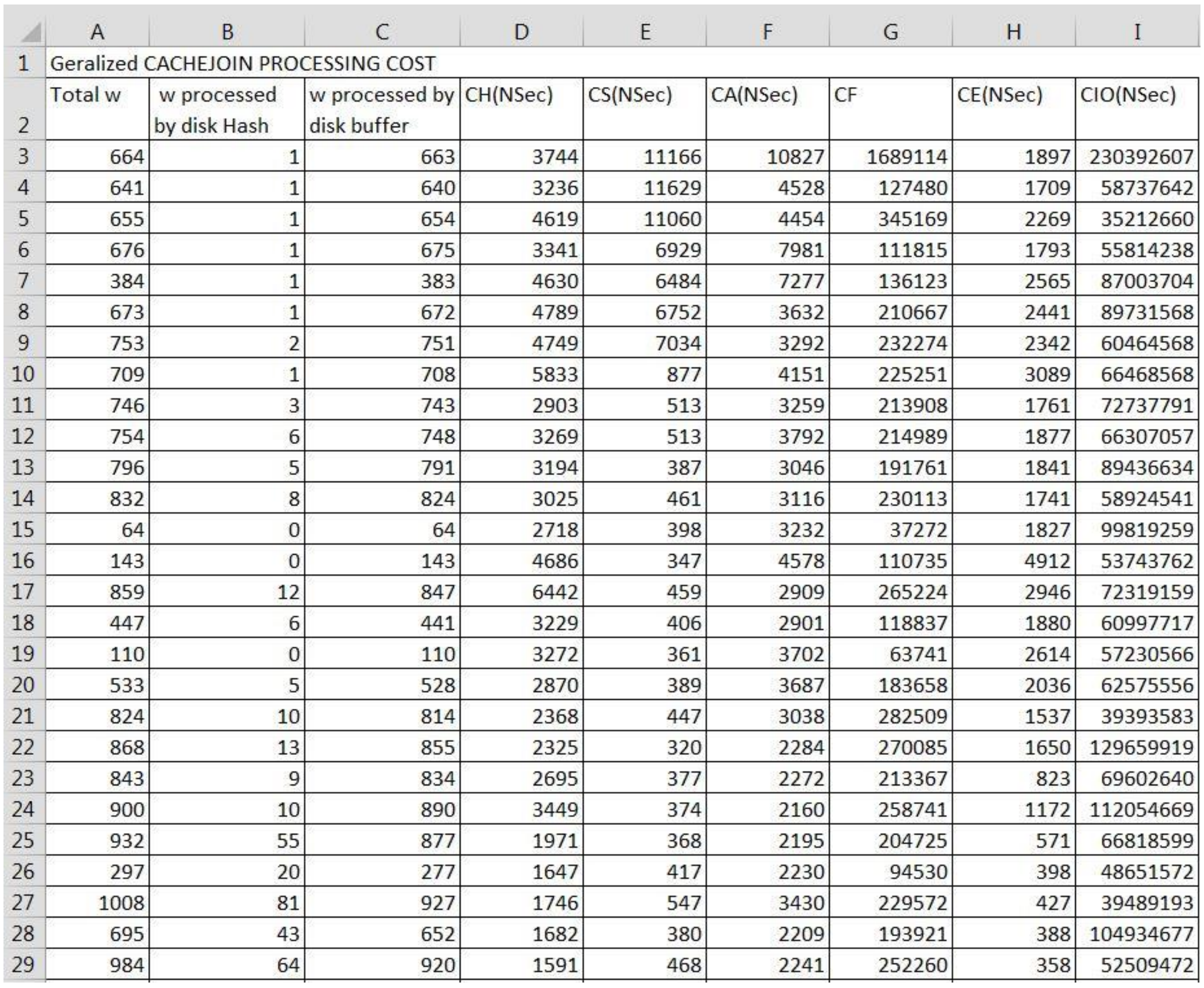
```
				tDAO.setRetail(rs.getString("retail"));
				tDAO.setSys_price(rs.getString("sys_price"));
				tDAO.setSys_disc(rs.getString("sys_disc"));
				tDAO.setSet_disc_val(rs.getString("set_disc_val"));
				tDAO.setAct_price(rs.getString("act_price"));
				tDAO.setAct_disc(rs.getString("act_disc"));
				tDAO.setAct_disc_val(rs.getString("act_disc_val"));
				tDAO.setLine(rs.getString("line"));
				tDAO.setUnit(rs.getString("unit"));
				tDAO.setLength(rs.getString("length"));
				tDAO.setTill(rs.getString("till"));
				tDAO.setDate2post(rs.getString("date2post"));
				tDAO.setSub_account(rs.getString("sub_account"));
				tDAO.setItem_type(rs.getString("item_type"));
				tDAO.setSub_type(rs.getString("sub_type"));
				tDAO.setTime(rs.getString("time"));
				tDAO.setOverride_operator(rs.getString("override_operator
		"));
				transactionList.add(tDAO);
					}
		} catch (SQLException e) {
				e.printStackTrace();
		}finally{
				closeConnection(conn);
		}
		lastOffset+=tuppleSize;
		return transactionList;

	}

	public static Connection connectDB(){
		Connection conn=null;
		try{
				String userName = "root";
				String password = "root";
				String url = "jdbc:mysql://localhost/masterdata";
				Class.forName ("com.mysql.jdbc.Driver");
				conn = DriverManager.getConnection (url, userName,
password);
				//System.out.println("Connected to Database");
		}
		catch (Exception e)
		{
				System.err.println (e);
		}
		return conn;
	}

	public static void closeConnection(Connection conn){
		try{
				if(conn!=null){
					conn.close();
					//System.out.println("Database connection closed");
				}
		}catch (SQLException e)
		{
				System.err.println (e);
		}
	}

}
```

STARTUPDATESTREAM.java code

```
package cacheJoinSource;
import java.util.PriorityQueue;
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.io.IOException;

public class StartUpdatesStream extends Thread implements
Comparable<Object>{
      public static TimeManager2 time;
      public boolean on=false;
      public double timeInChosenUnit;
      public DistributionClass distribution;
      public MyQueue2 ownQueue;
      public double bandwidth;
      Random myRandom=new Random();
      StartUpdatesStream(){

      }
      public int compareTo(Object o) {
            StartUpdatesStream y = (StartUpdatesStream) o;
               double diff = this.timeInChosenUnit - y.timeInChosenUnit;
               if(diff < 0.0) return -1;
               if(diff > 0.0) return 1;
               return 0;
       }
      public void run(){
            try{
                  startStream();
            }catch (InterruptedException ie){
              System.out.println(ie.getMessage());
           }catch (IOException io){
              System.out.println(io.getMessage());
           }
      }

      StartUpdatesStream(MyQueue2 ownQueue, DistributionClass distribution,
double bandwidth){
            this.distribution=distribution;
            this.ownQueue=ownQueue;
            this.bandwidth=bandwidth;
            timeInChosenUnit=System.nanoTime();
            swapStatus();
      }

      public void swapStatus(){


      timeInChosenUnit+=distribution.getNextDistributionValue()*TimeManager
2.STEP*bandwidth;

            if(on){
                  ownQueue.totalCurrentBandwidth-=bandwidth;
                  on=false;
            }

            else{
                  ownQueue.totalCurrentBandwidth+=bandwidth;
                  on=true;
            }
            ownQueue.offer(this);
      }
      public void startStream()throws InterruptedException,IOException{

            DistributionClass distribution=new DistributionClass();
            DistributionClass generator=new DistributionClass();
            TimeManager2 time=new TimeManager2();
            MyQueue2 myQueue=new MyQueue2();
            int tuple=0;
            int count=0;
            long CS_per_Iteration=0,start=0,stop=0;

            for(int i=0; i<6; i++){
                  new
StartUpdatesStream(myQueue,distribution,Math.pow(2,i));
            }
            StartUpdatesStream current=(StartUpdatesStream)myQueue.poll();
            ArrayBlockingQueue<TransactionDAO> transactionList =
Transaction.fetchTransactionRecords();

            while(true){
                  tuple=0;
                  time.waitOneStep();
                  while(time.now()>current.timeInChosenUnit){
                        current=(StartUpdatesStream)myQueue.poll();
                        current.swapStatus();
                  }
                  while(tuple<myQueue.totalCurrentBandwidth){

      //tupleValue=Integer.toString(generator.getNextDistributionValue());
                        if(transactionList.isEmpty()){
                              transactionList =
Transaction.fetchTransactionRecords();
                        }
                        TransactionDAO tDAO = transactionList.poll();
                        start=System.nanoTime();
                        CACHEJOIN.streamBuffer.put(new
HybridJoinObject(tDAO.getBranch(),tDAO.getDept(),tDAO.getDrawer(),tDAO.getC
ode(),tDAO.getAmount(),tDAO.getCost(),tDAO.getDate_(),tDAO.getAccount(),tDA
O.getRemarks(),tDAO.getReference(),tDAO.getType_(),tDAO.getTax(),tDAO.getOp
erator(),tDAO.getQty(),tDAO.getStock(),tDAO.getDocket_no(),tDAO.getPosted()
,tDAO.getPromo(),tDAO.getSub_code(),tDAO.getGst(),tDAO.getPromo_num(),tDAO.
getPromo_start(),tDAO.getPromo_end(),tDAO.getDescription(),tDAO.getGroup_()
,tDAO.getRetail(),tDAO.getSys_price(),tDAO.getSys_disc(),tDAO.getSet_disc_v
al(),tDAO.getAct_price(),tDAO.getAct_disc(),tDAO.getAct_disc_val(),tDAO.get
Line(),tDAO.getUnit(),tDAO.getLength(),tDAO.getTill(),tDAO.getDate2post(),t
DAO.getSub_account(),tDAO.getItem_type(),tDAO.getSub_type(),tDAO.getTime(),
tDAO.getOverride_operator(),null));
                        stop=System.nanoTime();
                        CS_per_Iteration+=stop-start;
                        count++;
                        if(count==1000){

      CACHEJOIN.CS[CACHEJOIN.CS_index++]=CS_per_Iteration/count;
                              CS_per_Iteration=0;
                              count=0;
                        }
                        tuple++;

                  }
            }
```

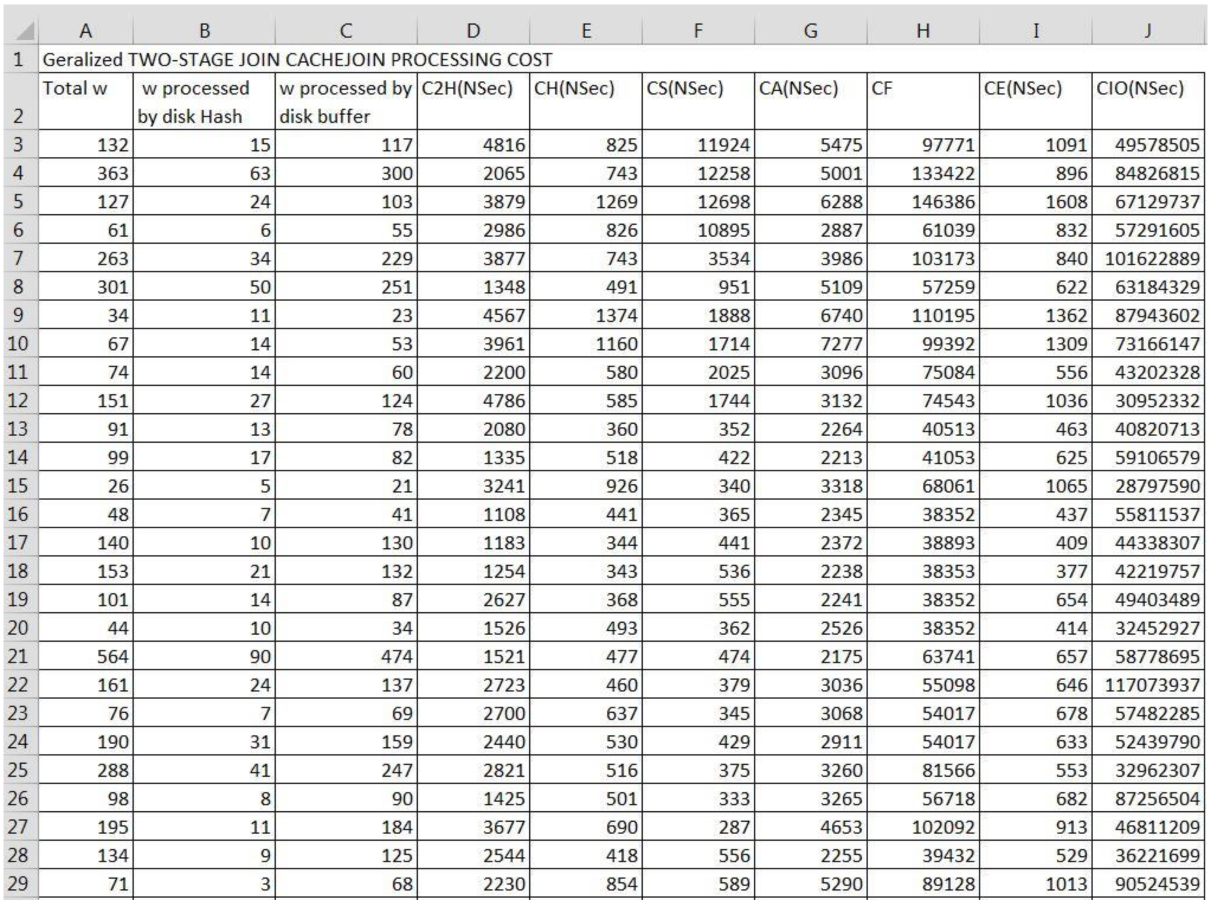
```
        }
}


class TimeManager2{
        public final static int STEP=15;
        public double now(){
                return(System.nanoTime());
        }
        public void waitOneStep(){
                try{
                        Thread.sleep(STEP);
                }catch (InterruptedException ie){
                        System.out.println(ie.getMessage());
                }
        }
}

class MyQueue2 extends PriorityQueue<StartUpdatesStream>{
        private static final long serialVersionUID = 1L;
        public long totalCurrentBandwidth=0;
}
```

Appendix B – Sample cost output file



	A	B	C	D	E	F	G	H	I
1	Geralized CACHEJOIN PROCESSING COST								
2	Total w	w processed by disk Hash	w processed by disk buffer	CH(NSec)	CS(NSec)	CA(NSec)	CF	CE(NSec)	CIO(NSec)
3	664	1	663	3744	11166	10827	1689114	1897	230392607
4	641	1	640	3236	11629	4528	127480	1709	58737642
5	655	1	654	4619	11060	4454	345169	2269	35212660
6	676	1	675	3341	6929	7981	111815	1793	55814238
7	384	1	383	4630	6484	7277	136123	2565	87003704
8	673	1	672	4789	6752	3632	210667	2441	89731568
9	753	2	751	4749	7034	3292	232274	2342	60464568
10	709	1	708	5833	877	4151	225251	3089	66468568
11	746	3	743	2903	513	3259	213908	1761	72737791
12	754	6	748	3269	513	3792	214989	1877	66307057
13	796	5	791	3194	387	3046	191761	1841	89436634
14	832	8	824	3025	461	3116	230113	1741	58924541
15	64	0	64	2718	398	3232	37272	1827	99819259
16	143	0	143	4686	347	4578	110735	4912	53743762
17	859	12	847	6442	459	2909	265224	2946	72319159
18	447	6	441	3229	406	2901	118837	1880	60997717
19	110	0	110	3272	361	3702	63741	2614	57230566
20	533	5	528	2870	389	3687	183658	2036	62575556
21	824	10	814	2368	447	3038	282509	1537	39393583
22	868	13	855	2325	320	2284	270085	1650	129659919
23	843	9	834	2695	377	2272	213367	823	69602640
24	900	10	890	3449	374	2160	258741	1172	112054669
25	932	55	877	1971	368	2195	204725	571	66818599
26	297	20	277	1647	417	2230	94530	398	48651572
27	1008	81	927	1746	547	3430	229572	427	39489193
28	695	43	652	1682	380	2209	193921	388	104934677
29	984	64	920	1591	468	2241	252260	358	52509472

Figure B.1 Sample copy of cost output file for CACHEJOIN algorithm written at the end of the algorithm in Java Eclipse.

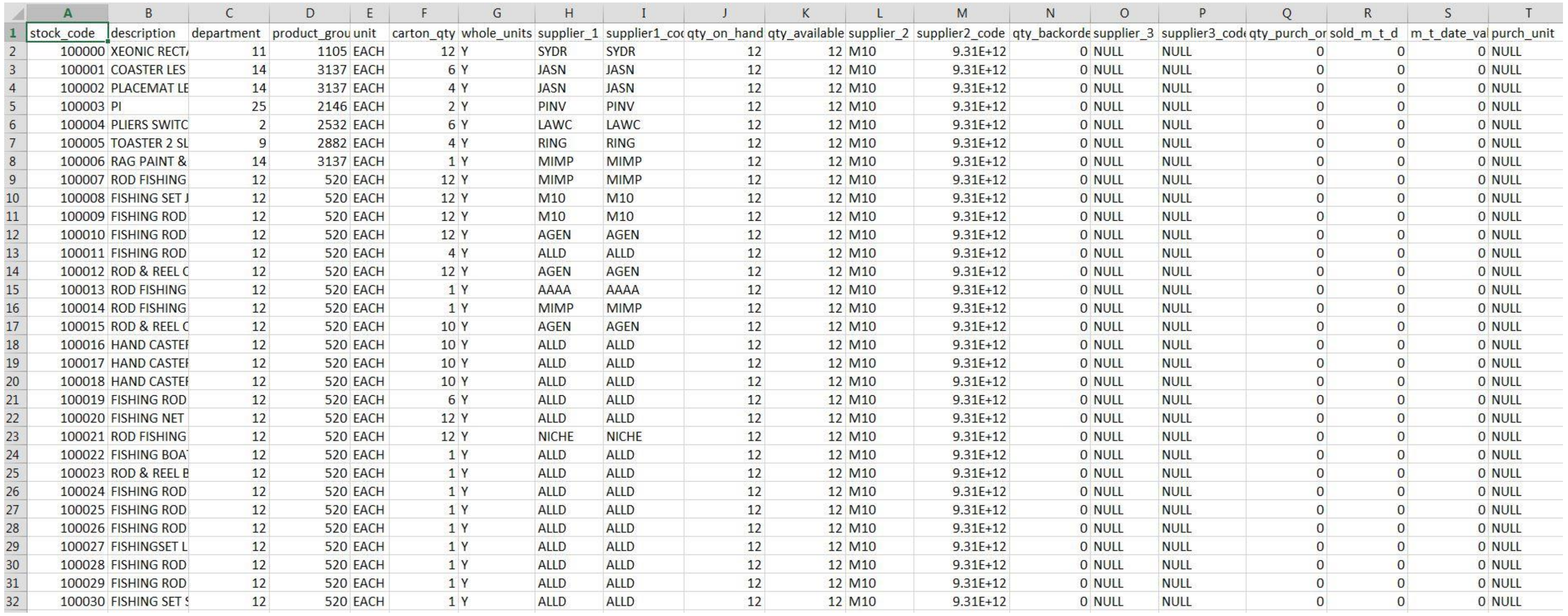


	A	B	C	D	E	F	G	H	I	J
1	Geralized TWO-STAGE JOIN CACHEJOIN PROCESSING COST									
2	Total w	w processed by disk Hash	w processed by disk buffer	C2H(NSec)	CH(NSec)	CS(NSec)	CA(NSec)	CF	CE(NSec)	CIO(NSec)
3	132	15	117	4816	825	11924	5475	97771	1091	49578505
4	363	63	300	2065	743	12258	5001	133422	896	84826815
5	127	24	103	3879	1269	12698	6288	146386	1608	67129737
6	61	6	55	2986	826	10895	2887	61039	832	57291605
7	263	34	229	3877	743	3534	3986	103173	840	101622889
8	301	50	251	1348	491	951	5109	57259	622	63184329
9	34	11	23	4567	1374	1888	6740	110195	1362	87943602
10	67	14	53	3961	1160	1714	7277	99392	1309	73166147
11	74	14	60	2200	580	2025	3096	75084	556	43202328
12	151	27	124	4786	585	1744	3132	74543	1036	30952332
13	91	13	78	2080	360	352	2264	40513	463	40820713
14	99	17	82	1335	518	422	2213	41053	625	59106579
15	26	5	21	3241	926	340	3318	68061	1065	28797590
16	48	7	41	1108	441	365	2345	38352	437	55811537
17	140	10	130	1183	344	441	2372	38893	409	44338307
18	153	21	132	1254	343	536	2238	38353	377	42219757
19	101	14	87	2627	368	555	2241	38352	654	49403489
20	44	10	34	1526	493	362	2526	38352	414	32452927
21	564	90	474	1521	477	474	2175	63741	657	58778695
22	161	24	137	2723	460	379	3036	55098	646	117073937
23	76	7	69	2700	637	345	3068	54017	678	57482285
24	190	31	159	2440	530	429	2911	54017	633	52439790
25	288	41	247	2821	516	375	3260	81566	553	32962307
26	98	8	90	1425	501	333	3265	56718	682	87256504
27	195	11	184	3677	690	287	4653	102092	913	46811209
28	134	9	125	2544	418	556	2255	39432	529	36221699
29	71	3	68	2230	854	589	5290	89128	1013	90524539

Figure B.2 Sample copy of cost output file for two-stage CACHEJOIN algorithm written at the end of the algorithm in Java Eclipse.

Appendix C - Sample real-life datasets

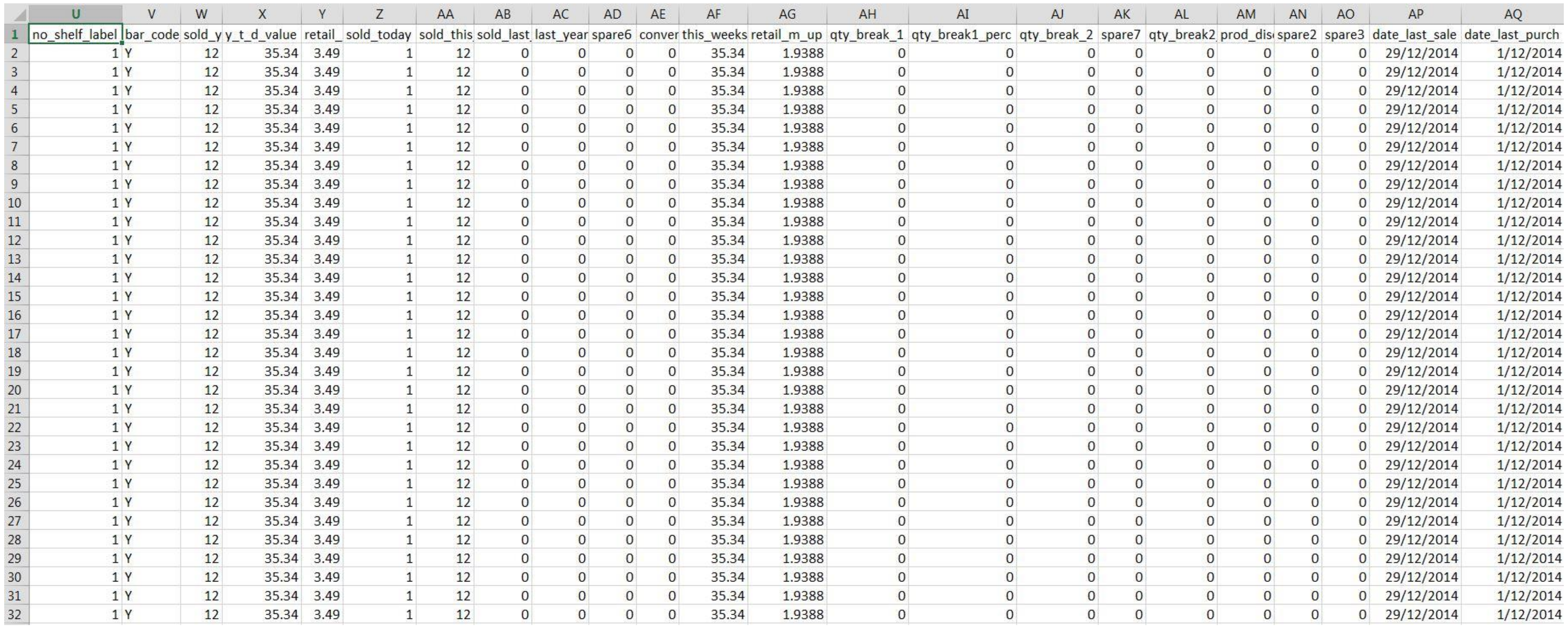
Below is the sample dataset of the MITRE 10 NZ product master table *sc*. This table has 86 attributes which cannot be shown in one screenshot. To make it more readable, it has been broken into four screenshots as below.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	stock_code	description	department	product_grou	unit	carton_qty	whole_units	supplier_1	supplier1_coo	qty_on_hand	qty_available	supplier_2	supplier2_code	qty_backorde	supplier_3	supplier3_code	qty_purch_or	sold_m_t_d	m_t_date_val	purch_unit
2	100000	XEONIC RECTA	11	1105	EACH	12	Y	SYDR	SYDR	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
3	100001	COASTER LES	14	3137	EACH	6	Y	JASN	JASN	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
4	100002	PLACEMAT LE	14	3137	EACH	4	Y	JASN	JASN	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
5	100003	PI	25	2146	EACH	2	Y	PINV	PINV	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
6	100004	PLIERS SWITC	2	2532	EACH	6	Y	LAWC	LAWC	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
7	100005	TOASTER 2 SL	9	2882	EACH	4	Y	RING	RING	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
8	100006	RAG PAINT &	14	3137	EACH	1	Y	MIMP	MIMP	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
9	100007	ROD FISHING	12	520	EACH	12	Y	MIMP	MIMP	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
10	100008	FISHING SET J	12	520	EACH	12	Y	M10	M10	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
11	100009	FISHING ROD	12	520	EACH	12	Y	M10	M10	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
12	100010	FISHING ROD	12	520	EACH	12	Y	AGEN	AGEN	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
13	100011	FISHING ROD	12	520	EACH	4	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
14	100012	ROD & REEL C	12	520	EACH	12	Y	AGEN	AGEN	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
15	100013	ROD FISHING	12	520	EACH	1	Y	AAAA	AAAA	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
16	100014	ROD FISHING	12	520	EACH	1	Y	MIMP	MIMP	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
17	100015	ROD & REEL C	12	520	EACH	10	Y	AGEN	AGEN	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
18	100016	HAND CASTER	12	520	EACH	10	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
19	100017	HAND CASTER	12	520	EACH	10	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
20	100018	HAND CASTER	12	520	EACH	10	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
21	100019	FISHING ROD	12	520	EACH	6	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
22	100020	FISHING NET	12	520	EACH	12	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
23	100021	ROD FISHING	12	520	EACH	12	Y	NICHE	NICHE	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
24	100022	FISHING BOAT	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
25	100023	ROD & REEL B	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
26	100024	FISHING ROD	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
27	100025	FISHING ROD	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
28	100026	FISHING ROD	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
29	100027	FISHINGSET L	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
30	100028	FISHING ROD	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
31	100029	FISHING ROD	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL
32	100030	FISHING SET S	12	520	EACH	1	Y	ALLD	ALLD	12	12	M10	9.31E+12	0	NULL	NULL	0	0	0	NULL

Figure C.1 Sample of MITRE 10 NZ product master table dataset R (sc table) which holds product item attributes (continued on next page)

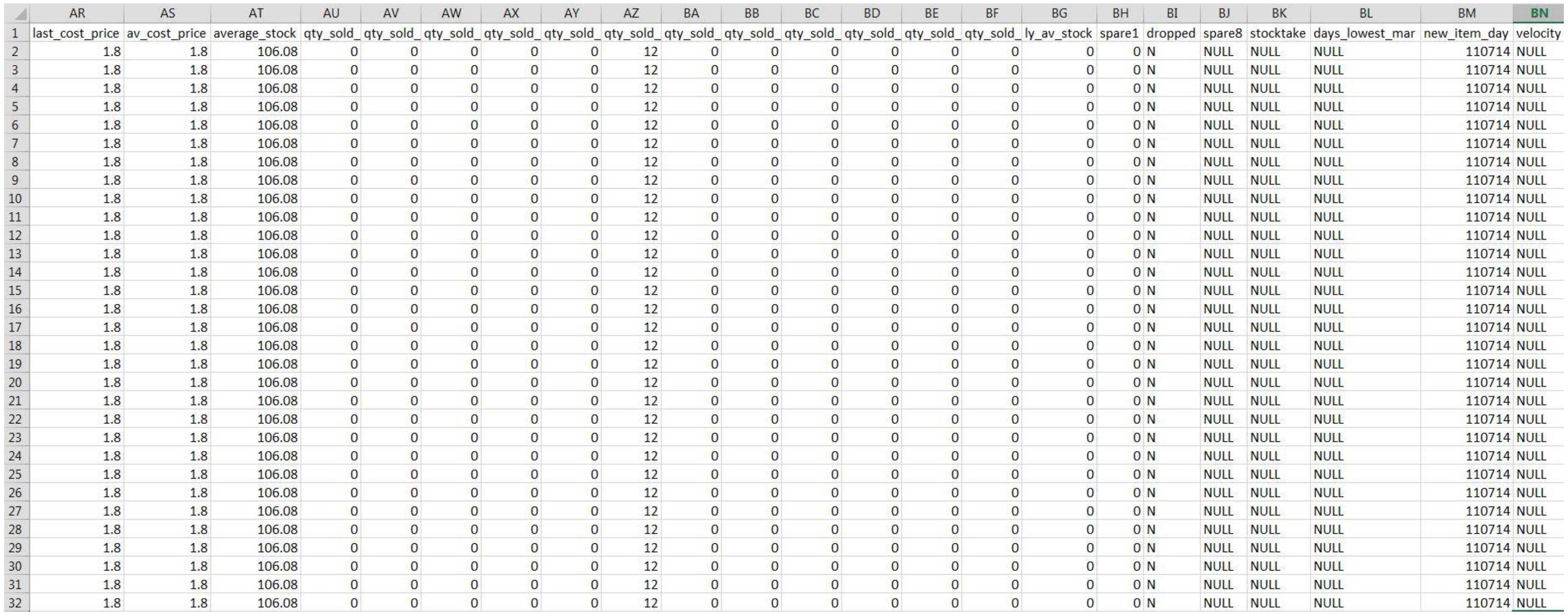
Figure C.1 – continued from previous page



	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ
1	no_shelf_label	bar_code	sold_y	y_t_d_value	retail_	sold_today	sold_this	sold_last	last_year	spare6	conver	this_weeks	retail_m_up	qty_break_1	qty_break1_perc	qty_break_2	spare7	qty_break2	prod_dis	spare2	spare3	date_last_sale	date_last_purch
2	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
3	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
4	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
5	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
6	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
7	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
8	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
9	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
10	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
11	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
12	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
13	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
14	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
15	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
16	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
17	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
18	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
19	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
20	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
21	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
22	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
23	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
24	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
25	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
26	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
27	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
28	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
29	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
30	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
31	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014
32	1	Y	12	35.34	3.49	1	12	0	0	0	0	35.34	1.9388	0	0	0	0	0	0	0	0	29/12/2014	1/12/2014

(Continued on next page)

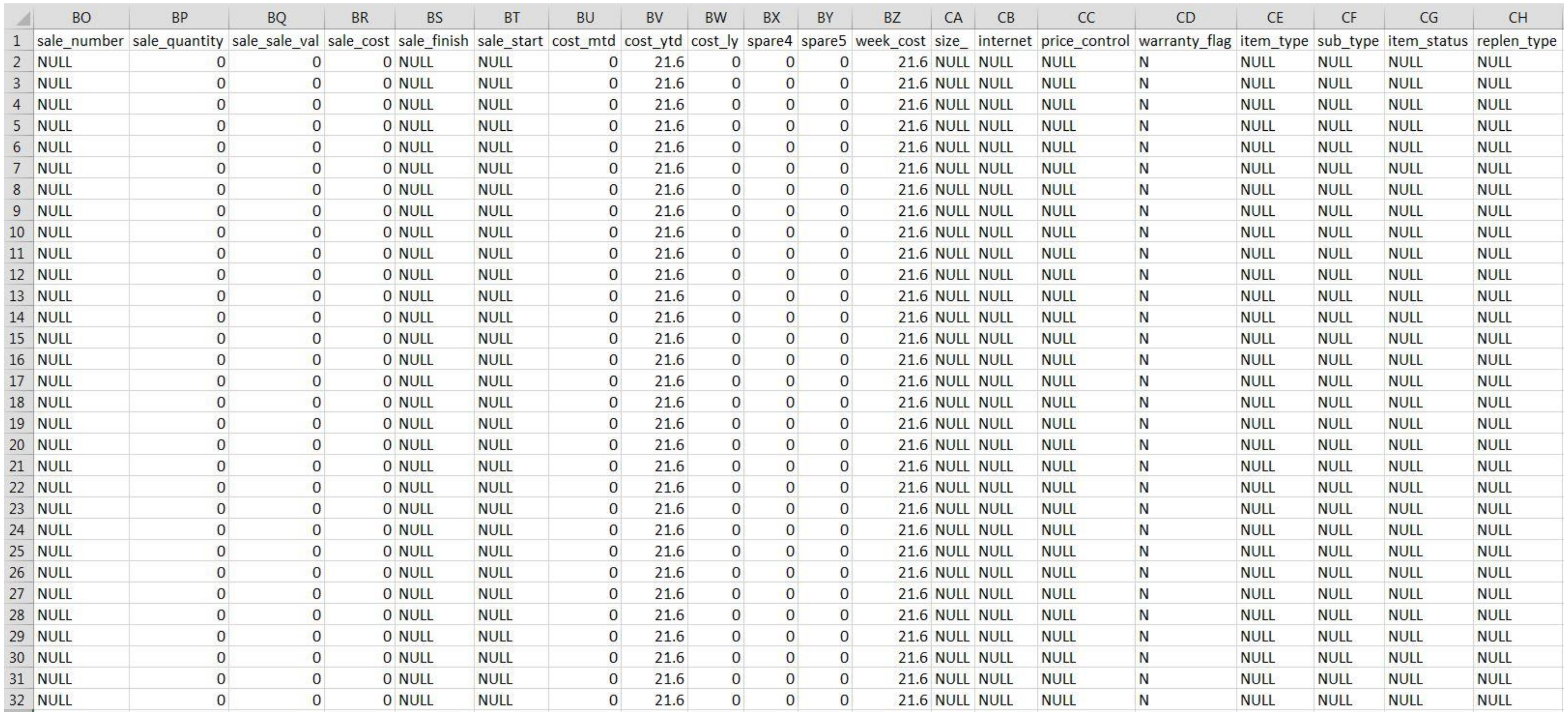
Figure C.1 – continued from previous page



	AR	AS	AT	AU	AV	AW	AX	AY	AZ	BA	BB	BC	BD	BE	BF	BG	BH	BI	BJ	BK	BL	BM	BN
1	last_cost_price	av_cost_price	average_stock	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	qty_sold_	ly_av_stock	spare1	dropped	spare8	stocktake	days_lowest_mar	new_item_day	velocity
2	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
3	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
4	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
5	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
6	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
7	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
8	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
9	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
10	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
11	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
12	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
13	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
14	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
15	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
16	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
17	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
18	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
19	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
20	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
21	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
22	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
23	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
24	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
25	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
26	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
27	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
28	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
29	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
30	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
31	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL
32	1.8	1.8	106.08	0	0	0	0	0	12	0	0	0	0	0	0	0	0	N	NULL	NULL	NULL	110714	NULL

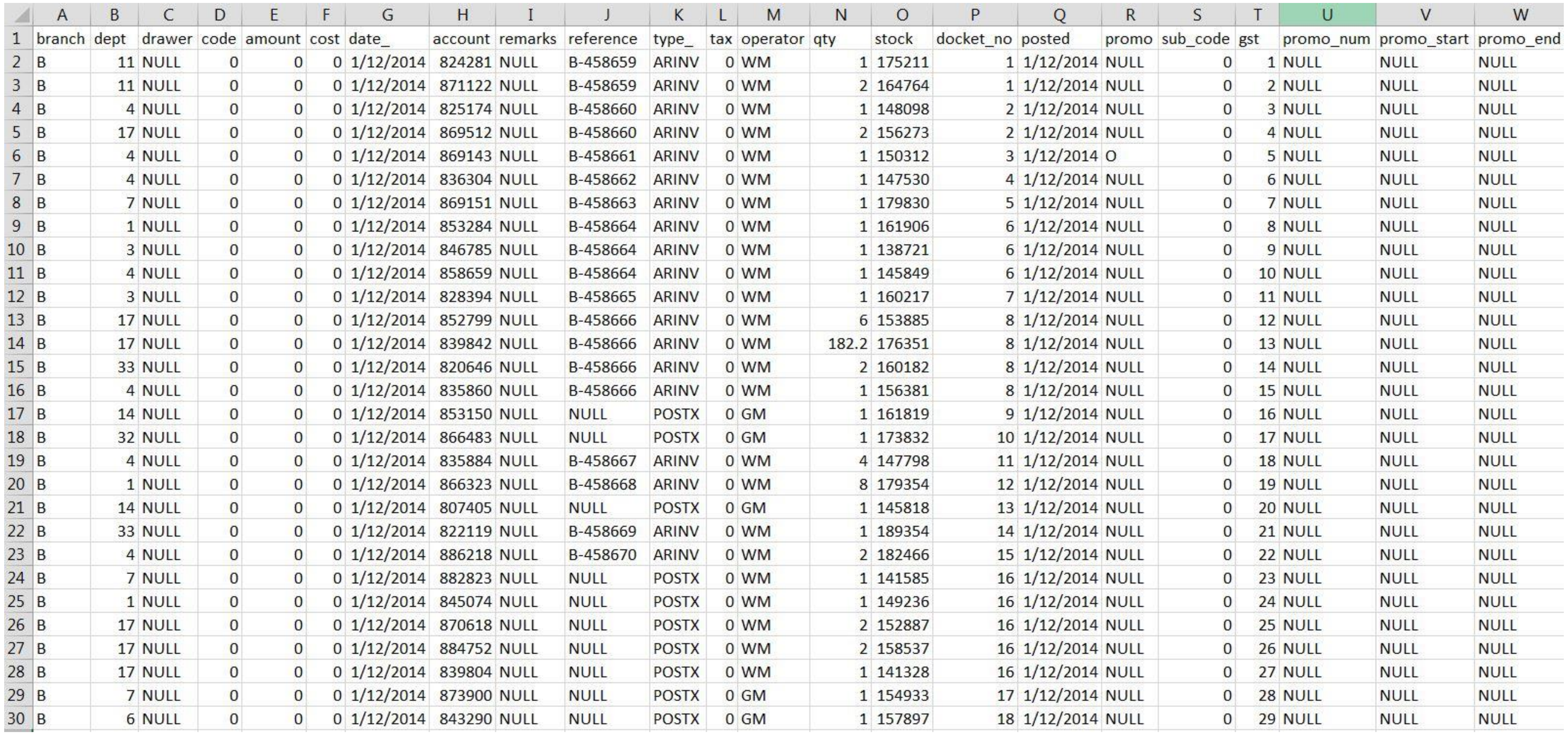
(Continued on next page)

Figure C.1 – continued from previous page



	BO	BP	BQ	BR	BS	BT	BU	BV	BW	BX	BY	BZ	CA	CB	CC	CD	CE	CF	CG	CH
1	sale_number	sale_quantity	sale_sale_val	sale_cost	sale_finish	sale_start	cost_mtd	cost_ytd	cost_ly	spare4	spare5	week_cost	size_	internet	price_control	warranty_flag	item_type	sub_type	item_status	replen_type
2	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
3	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
4	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
5	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
6	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
7	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
8	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
9	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
10	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
11	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
12	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
13	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
14	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
15	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
16	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
17	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
18	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
19	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
20	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
21	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
22	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
23	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
24	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
25	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
26	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
27	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
28	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
29	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
30	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
31	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL
32	NULL	0	0	0	NULL	NULL	0	21.6	0	0	0	21.6	NULL	NULL	NULL	N	NULL	NULL	NULL	NULL

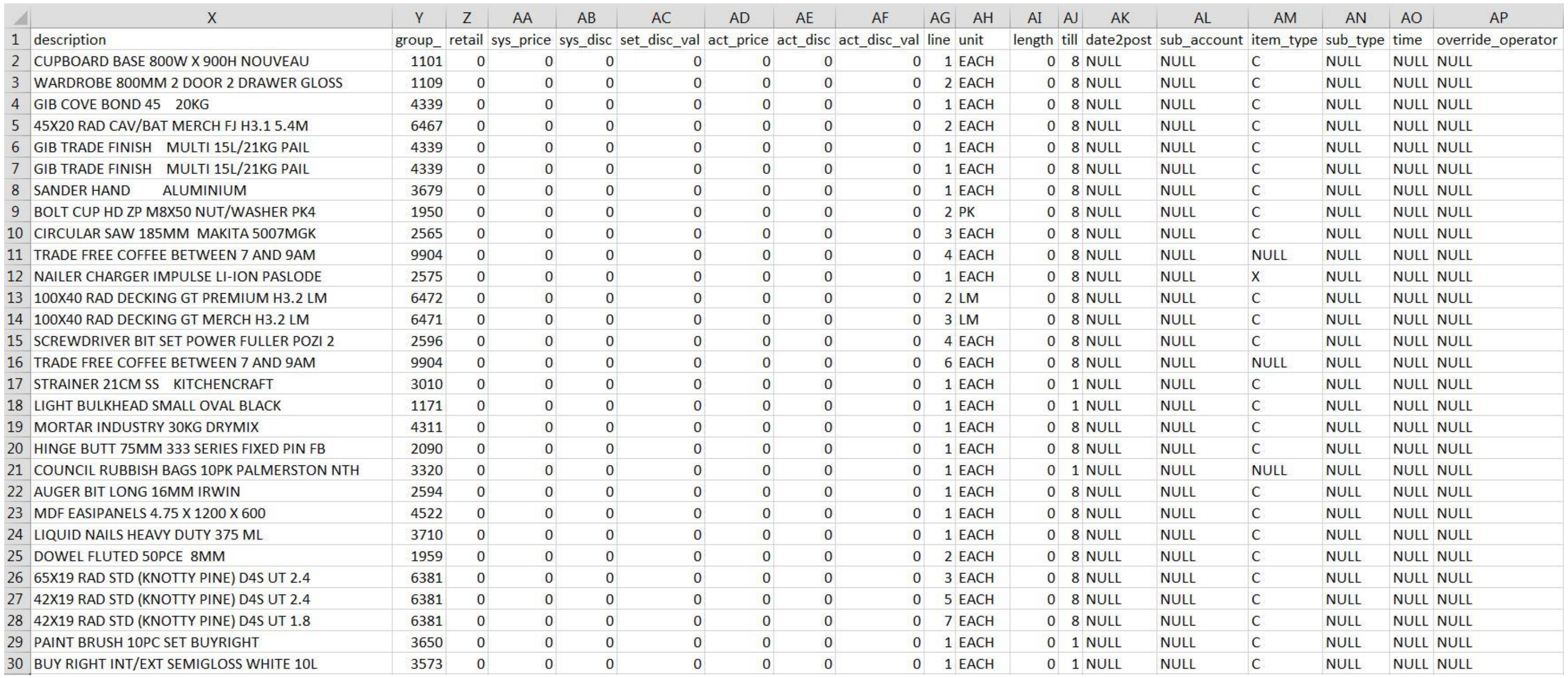
Below is the sample dataset of the MITRE 10 NZ transactional table *gltx*. This table has 42 attributes which cannot be shown in one screenshot. To make it more readable, it has been broken into two screenshots as below.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1	branch	dept	drawer	code	amount	cost	date_	account	remarks	reference	type_	tax	operator	qty	stock	docket_no	posted	promo	sub_code	gst	promo_num	promo_start	promo_end
2	B	11	NULL	0	0	0	1/12/2014	824281	NULL	B-458659	ARINV	0	WM	1	175211	1	1/12/2014	NULL	0	1	NULL	NULL	NULL
3	B	11	NULL	0	0	0	1/12/2014	871122	NULL	B-458659	ARINV	0	WM	2	164764	1	1/12/2014	NULL	0	2	NULL	NULL	NULL
4	B	4	NULL	0	0	0	1/12/2014	825174	NULL	B-458660	ARINV	0	WM	1	148098	2	1/12/2014	NULL	0	3	NULL	NULL	NULL
5	B	17	NULL	0	0	0	1/12/2014	869512	NULL	B-458660	ARINV	0	WM	2	156273	2	1/12/2014	NULL	0	4	NULL	NULL	NULL
6	B	4	NULL	0	0	0	1/12/2014	869143	NULL	B-458661	ARINV	0	WM	1	150312	3	1/12/2014	O	0	5	NULL	NULL	NULL
7	B	4	NULL	0	0	0	1/12/2014	836304	NULL	B-458662	ARINV	0	WM	1	147530	4	1/12/2014	NULL	0	6	NULL	NULL	NULL
8	B	7	NULL	0	0	0	1/12/2014	869151	NULL	B-458663	ARINV	0	WM	1	179830	5	1/12/2014	NULL	0	7	NULL	NULL	NULL
9	B	1	NULL	0	0	0	1/12/2014	853284	NULL	B-458664	ARINV	0	WM	1	161906	6	1/12/2014	NULL	0	8	NULL	NULL	NULL
10	B	3	NULL	0	0	0	1/12/2014	846785	NULL	B-458664	ARINV	0	WM	1	138721	6	1/12/2014	NULL	0	9	NULL	NULL	NULL
11	B	4	NULL	0	0	0	1/12/2014	858659	NULL	B-458664	ARINV	0	WM	1	145849	6	1/12/2014	NULL	0	10	NULL	NULL	NULL
12	B	3	NULL	0	0	0	1/12/2014	828394	NULL	B-458665	ARINV	0	WM	1	160217	7	1/12/2014	NULL	0	11	NULL	NULL	NULL
13	B	17	NULL	0	0	0	1/12/2014	852799	NULL	B-458666	ARINV	0	WM	6	153885	8	1/12/2014	NULL	0	12	NULL	NULL	NULL
14	B	17	NULL	0	0	0	1/12/2014	839842	NULL	B-458666	ARINV	0	WM	182.2	176351	8	1/12/2014	NULL	0	13	NULL	NULL	NULL
15	B	33	NULL	0	0	0	1/12/2014	820646	NULL	B-458666	ARINV	0	WM	2	160182	8	1/12/2014	NULL	0	14	NULL	NULL	NULL
16	B	4	NULL	0	0	0	1/12/2014	835860	NULL	B-458666	ARINV	0	WM	1	156381	8	1/12/2014	NULL	0	15	NULL	NULL	NULL
17	B	14	NULL	0	0	0	1/12/2014	853150	NULL	NULL	POSTX	0	GM	1	161819	9	1/12/2014	NULL	0	16	NULL	NULL	NULL
18	B	32	NULL	0	0	0	1/12/2014	866483	NULL	NULL	POSTX	0	GM	1	173832	10	1/12/2014	NULL	0	17	NULL	NULL	NULL
19	B	4	NULL	0	0	0	1/12/2014	835884	NULL	B-458667	ARINV	0	WM	4	147798	11	1/12/2014	NULL	0	18	NULL	NULL	NULL
20	B	1	NULL	0	0	0	1/12/2014	866323	NULL	B-458668	ARINV	0	WM	8	179354	12	1/12/2014	NULL	0	19	NULL	NULL	NULL
21	B	14	NULL	0	0	0	1/12/2014	807405	NULL	NULL	POSTX	0	GM	1	145818	13	1/12/2014	NULL	0	20	NULL	NULL	NULL
22	B	33	NULL	0	0	0	1/12/2014	822119	NULL	B-458669	ARINV	0	WM	1	189354	14	1/12/2014	NULL	0	21	NULL	NULL	NULL
23	B	4	NULL	0	0	0	1/12/2014	886218	NULL	B-458670	ARINV	0	WM	2	182466	15	1/12/2014	NULL	0	22	NULL	NULL	NULL
24	B	7	NULL	0	0	0	1/12/2014	882823	NULL	NULL	POSTX	0	WM	1	141585	16	1/12/2014	NULL	0	23	NULL	NULL	NULL
25	B	1	NULL	0	0	0	1/12/2014	845074	NULL	NULL	POSTX	0	WM	1	149236	16	1/12/2014	NULL	0	24	NULL	NULL	NULL
26	B	17	NULL	0	0	0	1/12/2014	870618	NULL	NULL	POSTX	0	WM	2	152887	16	1/12/2014	NULL	0	25	NULL	NULL	NULL
27	B	17	NULL	0	0	0	1/12/2014	884752	NULL	NULL	POSTX	0	WM	2	158537	16	1/12/2014	NULL	0	26	NULL	NULL	NULL
28	B	17	NULL	0	0	0	1/12/2014	839804	NULL	NULL	POSTX	0	WM	1	141328	16	1/12/2014	NULL	0	27	NULL	NULL	NULL
29	B	7	NULL	0	0	0	1/12/2014	873900	NULL	NULL	POSTX	0	GM	1	154933	17	1/12/2014	NULL	0	28	NULL	NULL	NULL
30	B	6	NULL	0	0	0	1/12/2014	843290	NULL	NULL	POSTX	0	GM	1	157897	18	1/12/2014	NULL	0	29	NULL	NULL	NULL

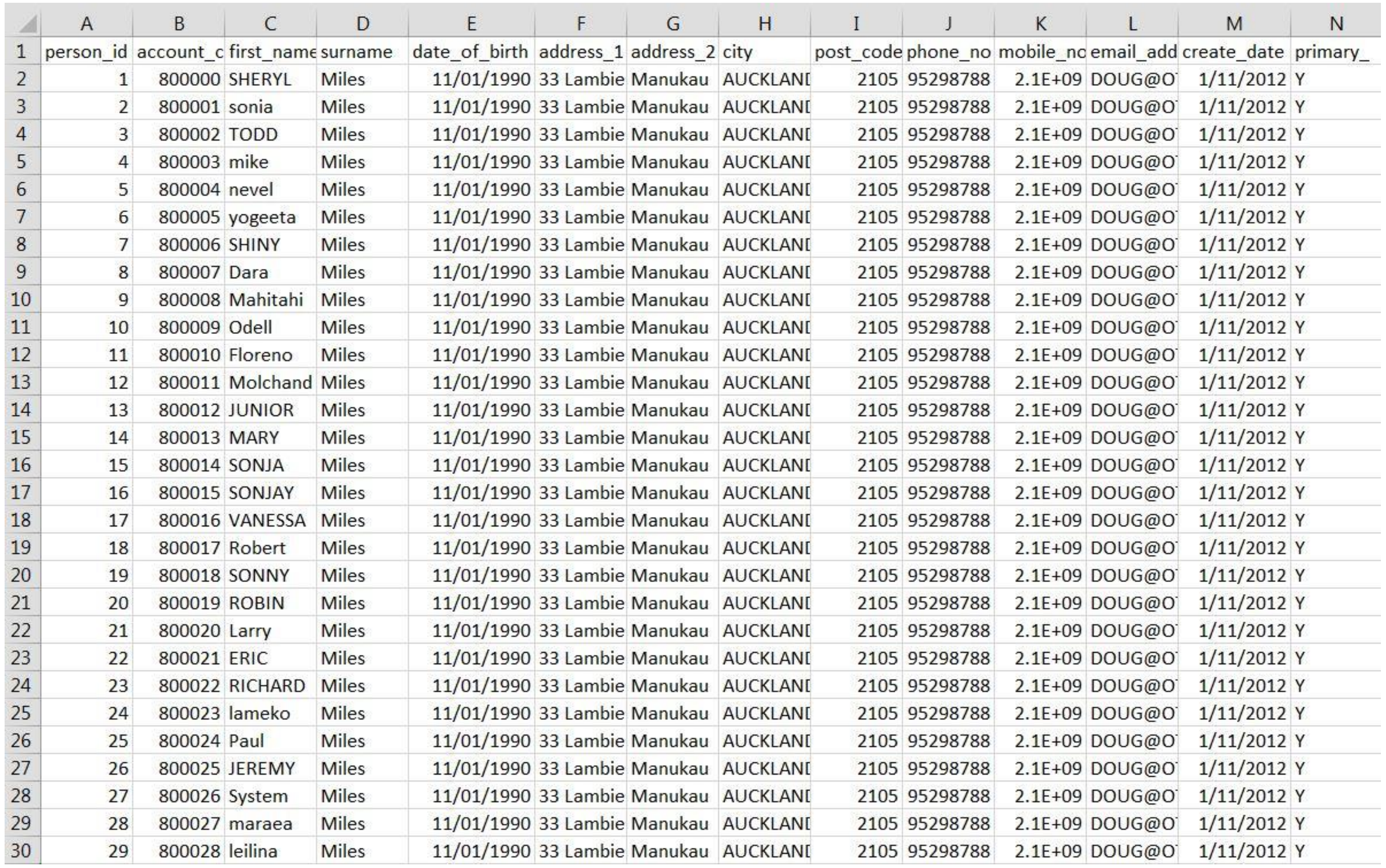
Figure C.2 Sample of MITRE 10 NZ transactional dataset (gltx table) which is incoming stream data S (continued on next page)

Figure C.2 – continued from previous page



	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP
1	description	group_	retail	sys_price	sys_disc	set_disc_val	act_price	act_disc	act_disc_val	line	unit	length	till	date2post	sub_account	item_type	sub_type	time	override_operator
2	CUPBOARD BASE 800W X 900H NOUVEAU	1101	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
3	WARDROBE 800MM 2 DOOR 2 DRAWER GLOSS	1109	0	0	0	0	0	0	0	2	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
4	GIB COVE BOND 45 20KG	4339	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
5	45X20 RAD CAV/BAT MERCH FJ H3.1 5.4M	6467	0	0	0	0	0	0	0	2	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
6	GIB TRADE FINISH MULTI 15L/21KG PAIL	4339	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
7	GIB TRADE FINISH MULTI 15L/21KG PAIL	4339	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
8	SANDER HAND ALUMINIUM	3679	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
9	BOLT CUP HD ZP M8X50 NUT/WASHER PK4	1950	0	0	0	0	0	0	0	2	PK	0	8	NULL	NULL	C	NULL	NULL	NULL
10	CIRCULAR SAW 185MM MAKITA 5007MGK	2565	0	0	0	0	0	0	0	3	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
11	TRADE FREE COFFEE BETWEEN 7 AND 9AM	9904	0	0	0	0	0	0	0	4	EACH	0	8	NULL	NULL	NULL	NULL	NULL	NULL
12	NAILER CHARGER IMPULSE LI-ION PASLODE	2575	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	X	NULL	NULL	NULL
13	100X40 RAD DECKING GT PREMIUM H3.2 LM	6472	0	0	0	0	0	0	0	2	LM	0	8	NULL	NULL	C	NULL	NULL	NULL
14	100X40 RAD DECKING GT MERCH H3.2 LM	6471	0	0	0	0	0	0	0	3	LM	0	8	NULL	NULL	C	NULL	NULL	NULL
15	SCREWDRIVER BIT SET POWER FULLER POZI 2	2596	0	0	0	0	0	0	0	4	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
16	TRADE FREE COFFEE BETWEEN 7 AND 9AM	9904	0	0	0	0	0	0	0	6	EACH	0	8	NULL	NULL	NULL	NULL	NULL	NULL
17	STRAINER 21CM SS KITCHENCRAFT	3010	0	0	0	0	0	0	0	1	EACH	0	1	NULL	NULL	C	NULL	NULL	NULL
18	LIGHT BULKHEAD SMALL OVAL BLACK	1171	0	0	0	0	0	0	0	1	EACH	0	1	NULL	NULL	C	NULL	NULL	NULL
19	MORTAR INDUSTRY 30KG DRYMIX	4311	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
20	HINGE BUTT 75MM 333 SERIES FIXED PIN FB	2090	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
21	COUNCIL RUBBISH BAGS 10PK PALMERSTON NTH	3320	0	0	0	0	0	0	0	1	EACH	0	1	NULL	NULL	NULL	NULL	NULL	NULL
22	AUGER BIT LONG 16MM IRWIN	2594	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
23	MDF EASIPANELS 4.75 X 1200 X 600	4522	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
24	LIQUID NAILS HEAVY DUTY 375 ML	3710	0	0	0	0	0	0	0	1	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
25	DOWEL FLUTED 50PCE 8MM	1959	0	0	0	0	0	0	0	2	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
26	65X19 RAD STD (KNOTTY PINE) D4S UT 2.4	6381	0	0	0	0	0	0	0	3	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
27	42X19 RAD STD (KNOTTY PINE) D4S UT 2.4	6381	0	0	0	0	0	0	0	5	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
28	42X19 RAD STD (KNOTTY PINE) D4S UT 1.8	6381	0	0	0	0	0	0	0	7	EACH	0	8	NULL	NULL	C	NULL	NULL	NULL
29	PAINT BRUSH 10PC SET BUYRIGHT	3650	0	0	0	0	0	0	0	1	EACH	0	1	NULL	NULL	C	NULL	NULL	NULL
30	BUY RIGHT INT/EXT SEMIGLOSS WHITE 10L	3573	0	0	0	0	0	0	0	1	EACH	0	1	NULL	NULL	C	NULL	NULL	NULL

Below is the sample dataset of the MITRE 10 NZ customer master table *cs_person*. This table has 15 attributes.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	person_id	account_c	first_name	surname	date_of_birth	address_1	address_2	city	post_code	phone_no	mobile_nc	email_add	create_date	primary_
2	1	800000	SHERYL	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
3	2	800001	sonia	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
4	3	800002	TODD	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
5	4	800003	mike	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
6	5	800004	nevel	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
7	6	800005	yogeeta	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
8	7	800006	SHINY	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
9	8	800007	Dara	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
10	9	800008	Mahitahi	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
11	10	800009	Odell	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
12	11	800010	Floreno	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
13	12	800011	Molchand	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
14	13	800012	JUNIOR	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
15	14	800013	MARY	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
16	15	800014	SONJA	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
17	16	800015	SONJAY	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
18	17	800016	VANESSA	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
19	18	800017	Robert	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
20	19	800018	SONNY	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
21	20	800019	ROBIN	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
22	21	800020	Larry	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
23	22	800021	ERIC	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
24	23	800022	RICHARD	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
25	24	800023	lameko	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
26	25	800024	Paul	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
27	26	800025	JEREMY	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
28	27	800026	System	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
29	28	800027	maraea	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y
30	29	800028	leilina	Miles	11/01/1990	33 Lambie	Manukau	AUCKLANI	2105	95298788	2.1E+09	DOUG@O	1/11/2012	Y

Figure C.3 Sample of MITRE 10 NZ customer master table dataset R (cs_person table) which holds customer account attributes.