IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 21, 2020, accepted October 20, 2020, date of publication October 23, 2020, date of current version November 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033464

Big Data Velocity Management–From Stream to Warehouse via High Performance Memory Optimized Index Join

M. ASIF NAEEM[1,2], (Member, IEEE), FARHAAN MIRZA[2], (Member, IEEE), HABIB ULLAH KHAN[3], (Member, IEEE), DAVID SUNDARAM[4], NOREEN JAMIL[1], AND GERALD WEBER[5]

[1]IKMA Laboratory, Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad 44000, Pakistan
[2]School of Engineering, Computer, and Mathematical Sciences, Auckland University of Technology, Auckland 1010, New Zealand
[3]Department of Accounting and Information Systems, College of Business and Economics, Qatar University, Doha, Qatar
[4]Department of Information Systems and Operations Management, The University of Auckland, Auckland 1010, New Zealand
[5]Department of Computer Science, The University of Auckland, Auckland 1010, New Zealand

Corresponding author: M. Asif Naeem (mnaeem@aut.ac.nz)

ABSTRACT Efficient resource optimization is critical to manage the velocity and volume of real-time streaming data in near-real-time data warehousing and business intelligence. This article presents a memory optimisation algorithm for rapidly joining streaming data with persistent master data in order to reduce data latency. Typically during the transformation phase of ETL (Extraction, Transformation, and Loading) a stream of transactional data needs to be joined with master data stored on disk. To implement this process, a *semi-stream join* operator is commonly used. Most semi-stream join operators cache frequent parts of the master data to improve their performance, this process requires careful distribution of allocated memory among the components of the *join* operator. This article presents a *cache inequality* approach to optimise cache size and memory. To test this approach, we present a novel Memory Optimal Index-based Join (MOIJ) algorithm. MOIJ supports many-to-many types of joins and adapts to dynamic streaming data. We also present a cost model for MOIJ and compare the performance with existing algorithms empirically as well as analytically. We envisage the enhanced ability of processing near-real-time streaming data using minimal memory will reduce latency in processing big data and will contribute to the development of high-performance real-time business intelligence systems.

INDEX TERMS Big data, near-real-time data warehouse, memory optimisation, performance optimisation, index-based join, cache inequality, high volume semi-stream data.

I. INTRODUCTION

Business intelligence people use data warehousing to capture relevant data (sense), analyze data (interpret), and produce valuable information that enables quick and effective decisions (respond) [1] for their business. The fundamental purpose of business intelligence systems [2], technologies, and knowledge discovery processes [3] is to minimize the time taken to capture all relevant data (data latency), analyze the data (analysis latency) and take an informed effective decision (decision latency) Figure 1. Data warehouse is an integrated time-variant pool of data used to support business intelligence. Data warehousing technology has enabled companies to organize and store large volumes of business data in a form that can be analyzed [4]. For example Walmart implemented the data warehouse and the initial Return on Investment (ROI) analysis viewed the investment as a strategic advantage over their competitors, however, the warehouse faced several problems including performance in the initial stages [5]. This is a common problem in large enterprises. Researchers have contributed various solutions to deal with the performance and optimisation challenges of voluminous high velocity data.

The concept of near-real-time data warehousing is a step towards minimizing the data latency in making user's data available in the data warehouse for analysis [7]. The tools and techniques for minimizing this data latency and increasing the

The associate editor coordinating the review of this manuscript and approving it for publication was Shajulin Benedict.

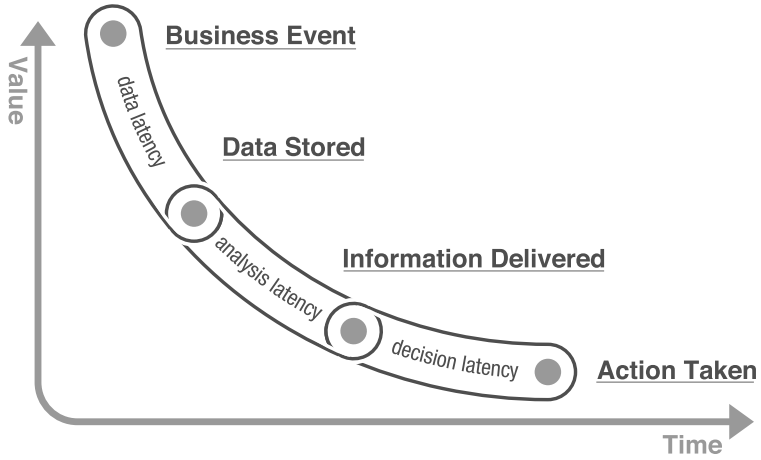


FIGURE 1. Data (Sense), Analysis (Interpret), and Decision (Respond) Latencies (adapted from [6].

level of data freshness in the data warehouse are evolving at a fast pace [8]–[12].

Initially, most data warehouses followed *batch-oriented* concepts where a complete reload of data was performed in each periodic cycle. Furthermore there used to be data consistency issues because database applications applied extensive caching and replication to boost performance [13]. This coupled with mobile or ubiquitous devices (increasing the data collection/access) posed performance challenges [14] and query correctness issues in databases [15]. As a result due to the high demand of up-to-date information, those batch-oriented data warehouses were unfit to fulfill growing and competitive business requirements. The most important factor that influences an organization's information need is its dynamic competitiveness, and consequently, companies with a dynamic supply chain [13] would need a faster transaction and operations data system [11]. Therefore the mechanism of loading data into warehouses was upgraded from a *full load* to an *incremental load*, in which only new updates are loaded to the warehouse [16]. While this approach is better than the previous one, it is still only periodic as the new data is reflected in the warehouse only after a certain period of time.

To meet the high demand of data freshness by businesses, batch-oriented and incremental *refresh approaches* are being replaced with *continuous data loading strategies* [17]–[22]. According to these strategies data is being captured, transformed, and loaded into the data warehouse on a continuous basis. In the following sections we contextualize this problem using an example, and outline our solution and contributions.

A. PROBLEM: NEAR REAL-TIME STREAM DATA PROCESSING

The data warehouse often uses a different format for storing data compared to the operational data sources. The source data therefore needs to be transformed into the format required by the data warehouse. Replacing of data *source key* to *surrogate key* or enriching of master data to the source data – also called content enrichment [23] is a typical scenario of such transformation. To explain this further we consider an example of a retail system where sales transactions need to be transformed with the master data before loading these to the data warehouse as shown in Figure 2. The sales transactions extracted from the data sources contain attributes *productid*(p_id), *storeid*(s_id) *quantity*(qty) and *Date*. Prior to loading these transactions to the data warehouse they require enrichment of attributes *surrogatekey*(s_key), *vendorid*(v_id), and *price* from the master data. Therefore a join operator is used to perform this enrichment process under the transformation layer of ETL (Extraction-Transformation-Loading).

In the context of near-real-time data warehousing, in which a stream of transactions needs to be joined with the disk-based master data, a semi-stream join operator is required. The challenge for such semi-stream join operator is to deal with the inputs coming from different sources at different arrival rates. The sales transactions input is generated in the form of a high volume stream with bursty nature while the master data input is disk-based.

B. SOLUTION: MEMORY OPTIMISED INDEX JOINS

A possible solution for the above problem is using a semi-stream join operator which is not trivial like simple static tables join or full stream join operators. The reason for this non-trivial nature is that, the access of disk-based master data is significantly slower than the stream input because of the disk I/O cost. This creates a bottleneck at the time of *join operation*. The challenge for this scenario is to use the available memory resources optimally among the join components to eliminate this bottleneck.

An exiting SSBJ algorithm [24] mentioned in the related work section below proposed a cache equation for the optimal distribution of available memory among the join components. However, the algorithm is limited to the scenario where the

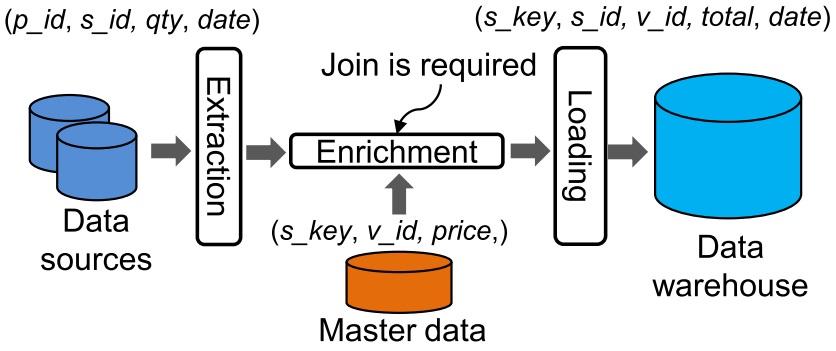


FIGURE 2. An example of stream-based join.

master data does not have index. In some scenario master data can have index therefore, to address this type of research challenge is important.

In this article we present a *cache inequality* approach for deciding the optimal memory among the cache and the other joins components particularly when the master data has index. To implement this cache inequality we present a novel semi-stream join algorithm called Memory Optimal Index-based Join (MOIJ). The proposed algorithm is presented to minimize the latency for processing streams data with disk-based master data. By using the new memory equation we can guarantee that the algorithm uses minimum amount of memory for producing the optimal service rate. In other words to produce a given service rate an algorithm can determine the minimum amount of memory. Also the algorithm uses tuple-level cache to store the master data records appeared in the stream most frequently. In addition to that MOIJ supports many-to-many type of join between the stream and the index-based master data which is useful for data cleaning operation in data warehouse. Another advantage that MOIJ has over SSBJ is guaranteed output against each disk access of the master data. Since MOIJ access the master data using a join attribute value in the stream as an index, at least one stream tuple match in the master data is guaranteed. While SSBJ reads the master data sequentially without an index and it is possible to have no stream tuple match against a disk read of the master data. This also means that SSBJ loads the unmatched partitions of master data into memory with equal frequency of loading the matched partitions which generates an unnecessary I/O cost without any join output.

Our main contributions contrary to existing Cache Join (CJ) [25] can be summarized as follows:

1) **Cache equation:** we present a cache equation (similar to SSBJ) for index-based semi-stream joins for optimal memory distribution among the join components. The equation ensures that the algorithm consumes minimum amount of memory to achieve the required value of service rate.
2) **A novel algorithm:** we present a novel algorithm called MOIJ, which implements the following features:
   a) MOIJ operates with minimum memory. The algorithm implements the new cache equation and therefore, acquires a minimum amount of memory for a given service rate.
   b) MOIJ uses *tuple level cache* meaning every single master data tuple stored in the cache is frequent. This facilitates continuous transition into main memory join to increase service rate.
   c) Contrary to SSBJ, MOIJ guaranties producing of at least one stream tuple as an output against each disk access of the master data.
   d) After the tuning phase, MOIJ adapts to online changes in stream data as well as the master data.
3) **Performance improvement:** our experimental data shows that MOIJ performs significantly better than other join algorithms for skewed stream data which is an important characteristic in sales data.
4) **Cost model:** we develop a cost model for our algorithm and evaluate this empirically.

The rest of the paper is structured as follows. Section II presents related work in this area. The methodology in Section III articulates how we conducted the research. In Section IV the paper presents theorems, algorithms, and architectures. Our approach towards the systems development and experimentation is presented in Section V, followed by mathematical evaluation of memory and cost models in Section VI. The empirical evaluation of memory and performance analysis is discussed in Section VII. Finally, Section VIII concludes the paper.

II. RELATED WORK

Processing data immediately after a business event will cause reduction in data latency and provide analysis of data to occur soon after as shown in 1. Semi-stream joins are a mechanism that can help us in reducing the data latency and even the analysis latency. Typically considering master data these semi-stream joins are of two types (a) index-based semi-stream joins (b) non-index-based semi-stream joins. Index-based semi-stream joins assume index on the master data while non-index-based semi-stream joins don't have this requirement. In the following we present our literature review against each type however, we restrict it to only hash-based semi-stream joins as this is directly related to our work.

A. NON-INDEX-BASED SEMI-STREAM JOIN

MESHJOIN (Mesh Join) [26] is a popular algorithm in the area of semi-stream processing that is designed especially for joining a stream of incoming data with a disk-based master data. This is a typical scenario in real-time data warehouses. MESHJOIN basically implements a hash join where the master data uses as the probe input while the stream data uses as the build input. One feature of MESHJOIN is that it builds the hash table in stages by loading the stream data in chunks. A limitation of the algorithm is that it does not make any assumptions about the distribution of the streaming data and the organization of the master data. The results show that the algorithm performs worse with skewed or non-uniform data [26]. Also MESHJOIN suffers with unnecessary dependencies among its components.

A variant version of MESHJOIN called R-MESHJOIN (reduced Mesh Join) [27] was proposed to remove these dependencies. As a result the performance improved slightly in R-MESHJOIN as compared to MESHJOIN. However, R-MESHJOIN also does not consider the aspect of skew in the streaming data.

Recently a cache based algorithm called Semi-Stream Balanced Join (SSBJ) has been presented in the literature [24]. The algorithm extends existing MESHJOIN by adding a cache module to it. Also the new algorithm implements a cache equation to optimally distribute the available memory between the cache module and the other join components. The cache equation ensures that the algorithm consumes minimum amount of memory in order to achieve a required service rate. SSBJ is a best memory efficient algorithm among the non-index types of joins. However, there can be a scenario where the master data has index and in this case the algorithm due to its nature of accessing master data sequentially can not be applied. The focus of this work is to develop a join for this type of the master data.

B. INDEX-BASED SEMI-STREAM JOIN

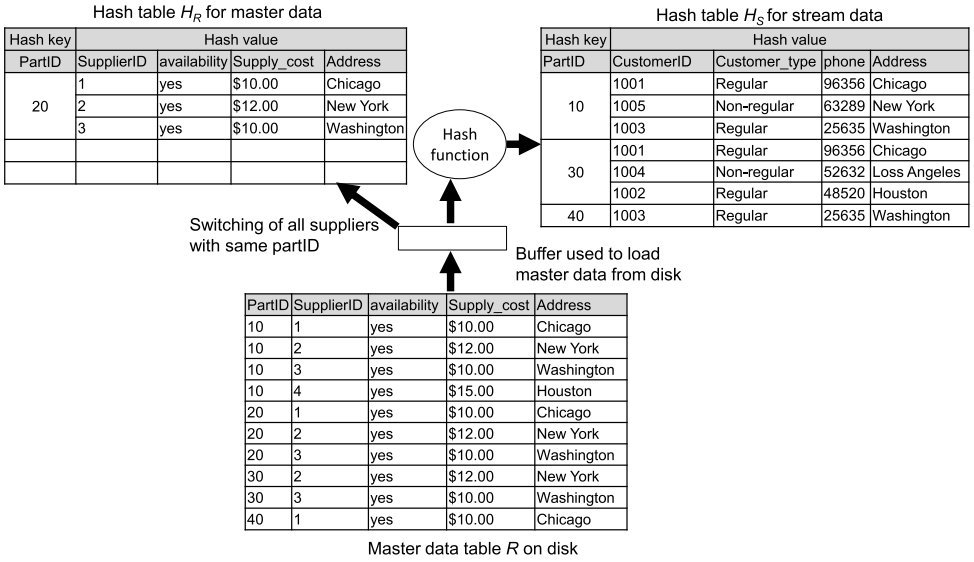
The Index Nested Loop Join (INLJ) [28] is an algorithm that can also be used to join the streaming data with disk-based master data. However, INLJ is based on non-clustered with respect to the join attribute in the streaming data. E.g., if the join is processed over *product_id* between the the streaming data and the master data then the master data is indexed by *product_id* and in this case it will be non-clustered index. Because of this non-clustered index access INLJ is known to be inefficient as the algorithm cannot amortize the expensive disk I/O cost over a rapid incoming stream of data. Eventually the algorithm produces a low service rate.

Another algorithm that attempts to improve MESHJOIN is the partition-based join [29]. This algorithm introduces a two-level hash table. In first level the algorithm attempts to join incoming stream tuples as they arrive. In case of no join, the algorithm executes the second level where the stream tuples are moved to a partition-based waiting area. However, the waiting of a tuple till execution is unbounded. Moreover, the algorithm uses cache at the page-level, which means there is no guarantee that every tuple on the page is frequent that makes the use of the cache memory suboptimal.

Another algorithm called the Semi-Streamed Index Join (SSIJ) [30] was tried to join streaming data with disk-based master data. The algorithm consists of three phases. The first phase called *pending phase* is when the stream input needs to wait in a waiting buffer unless the buffer size crosses the predefined threshold limit or the stream terminates. When the size of input buffer is greater than the predefined threshold limit, the algorithm advances to the second phase called *online phase*, in which the algorithm reads stream tuples from the waiting buffer and finds the matching disk tuples stored in cache blocks. In case if the required disk tuple is found in the cache, the algorithm executes join and produces the tuple as an output. In case if the 'required tuple' is not available in the cache, the algorithm moves the stream tuple to a stream buffer where it awaits for the next phase called join phase. The algorithm induces a preset threshold on the stream buffer. During the *join phase* a disk block is loaded to the cache if the size of the stream buffer crosses the threshold value. Then the algorithm performs join between the tuples in the stream buffer and the tuples in the loaded disk block. The algorithm implements a priority counter for each disk block loaded into the cache and based in that the algorithm decides about the disk blocks that need to be kept in memory. However similar to partition-based join, SSIJ uses page-level cache which means there is no guarantee that every tuple on the page is frequent that makes the use of the cache memory suboptimal. Also none of the above algorithms can deal the bursty nature of the streaming data.

The algorithm for joining stream data with a disk-based relation by Derakhshan *et al.* [31] uses a cache to store frequent master data tuples and a waiting queue for stream tuples that are not joined through the cache. The algorithm processes this waiting queue in batches. The primary focus of the algorithm is to preserve the arrival order of the stream tuples in the output. While useful for some applications, this order is not important in most applications such as the ones we consider. Preserving this order can cause delays in producing outputs for some stream tuples, e.g. a stream tuple already processed through the cache cannot be output if its predecessor tuple is still in the waiting queue. Derakhshan and others also demonstrated the role of stream-relation joins in a federated stream processing system called MaxStream [32]. Apache Flink is another architecture for processing stream and batch data [33]. The key focus of this architecture is processing stream and batch data independently rather than a join operation between stream and disk data.

Another algorithm called HYBRIDJOIN (Hybrid Join) [34] was proposed to deal with the bursty and non-uniform characteristics of the streaming data. The algorithm joins the streaming data with a historical master under limited available memory resources. The algorithm uses a clustered index on the join attribute in the master data. Due to this the algorithm loads only those pages of the master data into the memory which at least have one matching tuple



Hash key	Hash value			
PartID	SupplierID	availability	Supply_cost	Address
20	1	yes	$10.00	Chicago
	2	yes	$12.00	New York
	3	yes	$10.00	Washington

Hash key	Hash value				
PartID	CustomerID	Customer_type	phone	Address	
10	1001	Regular	96356	Chicago	
	1005	Non-regular	63289	New York	
	1003	Regular	25635	Washington	
30	1001	Regular	96356	Chicago	
	1004	Non-regular	52632	Loss Angeles	
	1002	Regular	48520	Houston	
40	1003	Regular	25635	Washington	

PartID	SupplierID	availability	Supply_cost	Address
10	1	yes	$10.00	Chicago
10	2	yes	$12.00	New York
10	3	yes	$10.00	Washington
10	4	yes	$15.00	Houston
20	1	yes	$10.00	Chicago
20	2	yes	$12.00	New York
20	3	yes	$10.00	Washington
30	2	yes	$12.00	New York
30	3	yes	$10.00	Washington
40	1	yes	$10.00	Chicago

FIGURE 3. Many-to-many join example.

in the stream data whereas there is no such guarantee in MESHJOIN and R-MESHJOIN. The other issue that the algorithm has addressed was to deal with the intermittency in the streaming data. Although by using the index on the master data HYBRIDJOIN amortizes the expensive disk I/O cost over the high speed input stream however, there is a possibility to scale up the performance if the aspect of skew in the stream data is to be considered. Also the algorithm does not apply for many-to-many type of joins.

Cache Join (CJ) [25] is an optimised version of HYBRIDJOIN that presents an additional cache component to deal with skewed stream data optimally. In this section we present CJ in details and highlight our observations that the memory distribution among the components of CJ algorithm is suboptimal.

The CJ algorithm includes two join phases, disk-probing phase and stream-probing phase. In the *disk-probing phase* the algorithm uses the disk-based master data R as the probe input whereas in the *stream-probing phase* the algorithm uses the stream as the probe input. For each tuple from the stream data, the algorithm first executes the stream-probing phase where the algorithm looks for a match to the stream tuple, and if match is not found, the algorithm forwards the tuple to the second phase i.e. disk-probing phase.

In terms of memory usage the key components for the algorithm are two hash tables, H_S storing stream tuples and H_R storing data from disk-based relation. The other components of the algorithm are disk buffer denoted by D_B, a queue denoted by Q and a tiny stream buffer denoted by S_B. Master data R and stream of sales transactions S are the input sources for the algorithm. H_R in the stream-probing phase retains the popular tuples of R in memory permanently. While H_S stores the stream tuples which do not find matched tuples in H_R.

The algorithm moves back and forth between disk-probing and the stream-probing phases. As mentioned above. For each stream tuple the algorithm first finds the matching tuple from H_R. If matched tuple is identified the algorithm it produces the output for that stream tuple otherwise loads that stream tuple to H_S. A stream-probing phase terminates if H_S gets filled or if no stream tuple in S_B is waiting. Once the stream-probing phase terminates the disk-probing phase begins. In the disk-probing phase the algorithm reads the last element from the queue and loads the relevant partition of R into D_B by using this element as an index.

Unlike Partitioned Join and SSIJ, CJ uses a tuple-level cache rather than a page-level cache to utilize the cache effectively. We identified two primary limitations in CJ. **First**, the algorithm doesn't support many-to-many relationship due to a clustered index on R. This is a common drawback in most of the semi-stream joins which use index on R [25], [30], [34], [35]. **Second**, the memory distribution between the stream-probing phase and the disk-probing phase can be suboptimal. Consider an example of join execution between a stream of customers' transaction and suppliers as shown in Figure 3. In the example, R contains a list of suppliers with their *partID*(s) and some other information. H_R keeps the frequent tuples of R in cache while H_S stores stream data which includes a list of customers' sales transactions. A join is needed to enrich some supplier details from R to the stream data required in the data warehouse. In the example we assume that the tuples in R and the stream are equal in size. We also assume the threshold value that the algorithm uses to determine if a tuple is frequent is equal to 3. By observing H_S from the figure we consider two cases.

In the first case where *partID* is 10, H_S contains three customer requests against that *partID*. According to the threshold value, this *partID* is frequent in the stream, therefore

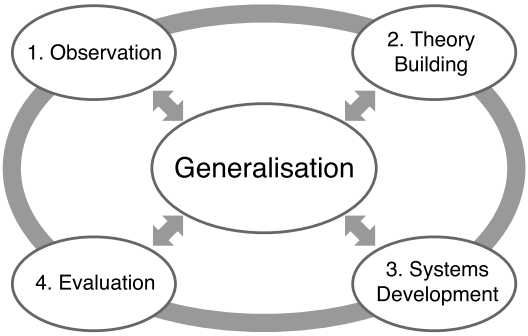


FIGURE 4. Phases of the research methodology.

according to CJ all suppliers in R related to this *partID* will be switched into H_R. However, the total number of suppliers in R who produce this *partID* are 4, so switching this *partID* into H_R (cache) is not an optimal option with respect to the memory consumption. In other words, H_S is a right place to process this *partID*. In the second case where *partID* is 30, we also have three customers' transaction against that *partID*. However, there are two suppliers who produce that *partID*. Contrary to the first case, switching of partID 30 into H_R is a right option as it will consume less memory. Contrarily CJ and other related approaches [30], [35] do not consider this thinking in their implementations. Consequently, the distribution of memory among all join components is not optimal and it negatively affects the performance. We present a solution to the above issues by proposing the MOIJ algorithm. The following section presents the research methodology that we adopted in our approach.

III. RESEARCH METHODOLOGY

The core objective of this work is to design, implement, test, and evaluate the proposed MOIJ algorithm. For this study, we adapted the design science research methodology presented by Nunamaker *et al.* [36] and Von Alan *et al.* [37]. This multi-method approach is an effective way of designing and implementing a system. It includes following areas of work - observation, theory building, systems development and experimentation as shown in Figure 4. The unordered phases are interconnected to facilitate creation and validation of a system with several iterations. As research progressed through each phase, and each iteration, the artifacts, theory, and results produced would be generalised as shown in the figure.

The research began with the *observation* phase. The observation involved reviewing existing literature to define the problem and technical motivation of the research (section I-A). The observation also involved technical articulation of similar algorithms with respect to the problem including CJ, MESHJOIN, R-MESHJOIN, INLJ, SSIJ, and HYBRIDJOIN (Section II). The learning from *observation* phase led to the *theory building* phase of the research. This phase involved creation of conceptual theorems, as shown in Section IV, these consider cache component to store frequent tuples in the stream to minimize the memory required for a given service rate. The architecture of MOIJ was introduced and this is articulated in a graphical representation (Figure 5), including pseudocode presented in Algorithm 1.

The *Systems development* phase of this research considered *design as a search process*. The authors in [37] state that design science is inherently iterative, the *search* for the best or optimal design is often intractable for realistic problems. Therefore we adapt an approach where we are able to generate, test and iterate. The architecture shown in figure 5 was setup as a system for conducting iterative design improvements and experimentation activities, presented in section V. The system developed allowed for rigorous experimentation activities.

The final phase dealt with *evaluation*, the evaluation was intended to test and enhance the MOIJ algorithm to an extent it is generalised. The generalisation was an ongoing process which was achieved through multiple iterations of design science phases described above. The evaluation process included mathematical evaluation (sectionVI) and empirical evaluation (section VII). The following sections articulate each research phase in further detail.

IV. THEORY BUILDING - THEOREMS, ALGORITHMS AND ARCHITECTURES

In this section we build our theorems, algorithms and based on our observation on hash based stream joins (Section II). Due to the same nature of cache inequality to SSBJ the proof of Theorems 1 and 2 are similar to those are presented in SSBJ. However the implementation of the equation as presented in Algorithm 1 is different due to the index on the master data.

A. CACHE INEQUALITY

CJ uses cache component to store frequent tuples from R in order to improve the service rate. To decide which R tuple is frequent the algorithm counts the number of matches in H_S against the R tuple. If the number of matches is greater than a preset threshold, the algorithm considers this tuple as a frequent and switches it into H_R (i.e. cache). However, as stated in the example above the switching of R tuples to H_R by considering their frequency only can be suboptimal with respect to memory consumption.

The purpose of the cache is therefore not to increase the service rate, but to minimize the memory required for a given service rate. In our cache inequality solution we follow a natural strategy, namely deciding for every join value individually whether it should be joined in the cache or the disk phase, based on how much memory either option would take. If a join value consumes less memory when joined in the cache, the required amount of memory is added to the cache. The memory used by a join value is the memory that all tuples with that join attribute consume in the respective hash table (H_S or H_R), plus some negligible overheads. We assume that the memory consumption of the hash table is strictly linear in the number of tuples stored.

Algorithm 1 Pseudo Code for MOIJ

Input: Incoming stream S and the master data R on the disk.
Output: $R \bowtie S$
Parameters: w (where $w = w_S + w_N$) input tuples of S and a segment of R.
Method:

1: $hSavailable := h_S, f\_count := 0$
2: **while** (true) **do**
3: **while** ($hSavailable > 0$ AND the stream available) **do**
4: READ a stream tuple t from S_B
5: **if** $H_R.lookUp(t.partID)$ **then**
6: OUTPUT all matches with H_R
7: $H_R(t.partID).f\_count := H_R(t.partID).f\_count + 1$
8: **else**
9: ADD t into H_S and join attribute $t.partID$ into Q
10: $hSavailable := hSavailable - 1$
11: **end if**
12: **end while**
13: READ the oldest $t.partID$ from Q
14: **if** $t.partID$ does not match in R **then**
15: DELETE $t.partID$ from Q
16: Go to line 13
17: **end if**
18: READ a segment of R into D_B using $t.partID$ as an index
19: **for** each tuple r in D_B **do**
20: **if** $H_S.lookUp(r.partID)$ **then**
21: OUTPUT all matches tuples from H_S
22: $m_{disk} := H_S(r.partID).size() \times v_S$
23: $m_{stream} := D_B(r.partID).size() \times v_R$
24: **if** $m_{disk} > m_{stream}$ **then**
25: $hSavailable := hSavailable - m_{stream}/v_S$
26: MOVE all tuples with $partIDs$equal to $r.partID$ from D_B to H_R
27: **end if**
28: $hSavailable := hSavailable + H_S(r.partID).size()$
29: DELETE all matched tuples from H_S with their pointers from Q
30: **end if**
31: **end for**
32: **if** $CacheEvictionProcess$ **then**
33: **for** each join value j in H_R **do**
34: $m_{stream} := H_R(j).size() \times v_R$
35: $m_{disk} := H_R(j).f\_count/10 \times v_S$
36: **if** $m_{stream} > m_{disk}$ **then**
37: DELETE $H_R(j)$ with all tuples against j
38: $hSavailable := hSavailable + m_{stream}/v_S$
39: **end if**
40: RESET $H_R(j).f\_count$ to 0
41: **end for**
42: **end if**
43: **end while**

This strategy can guarantee overall best memory usage. We capture this as Theorem 1. However, we need to generalise the considerations to cases, where stream tuples and R tuples have different sizes, and this will yield to introduction of the actual cache inequality in Theorem 2.

Theorem 1 (Minimal Memory Consumption): If every join value j is placed in the hash table where it uses less memory, then the overall memory consumption is minimal.

Proof: The proof is straightforward based on the linear memory consumption of hash tables. Let $m_{stream}(j)$ and $m_{disk}(j)$ be the memory needed for join value j in the stream-probing phase and the disk-probing phase, respectively. With J the set of all join values, the memory consumption of both hash tables is:

$$\text{Memory for } H_R \text{ and } H_S = \sum_{j \in J} \min(m_{stream}(j), m_{disk}(j))$$

All join values would use more memory if switched, hence the memory consumption is minimal. □

The footprint of other components (such as Q in MESHJOIN and additional frequency counters in both phases) is negligible, as they are dominated by the tuple sizes. Now we derive what we call *Cache Inequality* in Theorem 2, which builds upon the criterion used in Theorem 1.

Theorem 2 (Cache Inequality): Assume a join value j is used in m records of R and is appearing in an expected number of n stream tuples in the whole Q length. v_S and v_R is the size in bytes of a stream tuple and a R tuple, respectively. Then j can be processed with less memory consumption in the stream-probing phase than in the disk-probing phase if:

$$m \times v_R < n \times v_S$$

Proof: We prove that the sides of the inequality reflect the memory consumption of j in either of the two hash tables. To process j in the stream-probing phase, all R records matching that join attribute have to be loaded into H_R, i.e. $m \times v_R$. Since all stream tuples remain in H_S for the whole Q length, the expected number of tuples in H_S matching j at any point in time is n. Hence the expected memory consumption of j when processed in the disk-probing phase is $n \times v_S$. The strategy given in the theorem chooses the option with less memory. □

B. EXECUTION ARCHITECTURE FOR MOIJ

Figure 5 presents an execution architecture for MOIJ. The algorithm implements the cache inequality to decide moving data into and out the cache component. Similar to CJ, MOIJ also has two complementary hash join modules called *stream-probing phase* and *disk-probing phase*.

Each tuple from the stream first enters the stream-probing phase, which uses selected tuples of R as the build input and the stream as the probe input of a hash join. On arrival of the new stream tuple it is probed in the stream-probing phase first. If the tuple is matched, the join output is created. In case of unmatched the tuple is loaded to the disk-probing phase.

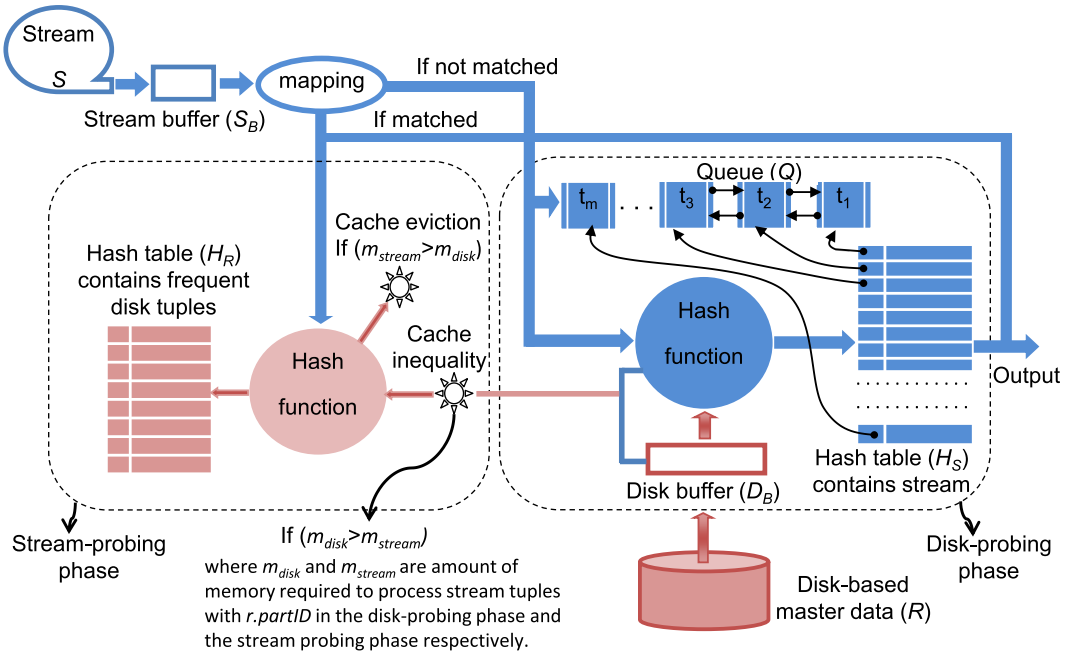


FIGURE 5. Execution architecture for MOIJ.

MOIJ switches between the stream-probing and the disk-probing phases, which includes one main loop step of CJ [25]. In further work, the two phases can be parallelized and executed in different threads, but this is not the aim of this article. Note that the stream input for the disk-probing phase is filtered by the stream-probing phase, i.e. many stream tuples are processed without going into the disk-probing phase. Master data R and stream of sales transactions S are two external inputs for the join algorithm. Similar to CJ in MOIJ a major part of memory is consumed by two hash tables: the disk-probing hash table H_S storing stream tuples, and the stream-probing has table H_R storing the most frequently accessed tuples from R. The queue in the disk-probing phase stores pointers to the stream tuples in H_S enabling the loading of segments from R to disk buffer D_B using index and ensures a stream tuple that enters into the disk-probing phase must be processed. It is growing with H_S and hence only adds a constant factor to the memory consumption per tuple in H_S. The other associated components of MOIJ are D_B and a stream buffer S_B. D_B is used to load the master data into memory in segments. S_B is comparably small, holding part of the stream when necessary.

C. PSEUDOCODE FOR MOIJ

Algorithm 1 presents the pseudo code for MOIJ. We use *partID* for the join value, aligned with the example in Figure 3 and consider index on it in R. The first loop in the algorithm runs infinitely, which is normal in these type of algorithms (line 2). The loop executes both phases - the stream-probing phase (lines 3 to 12) and the disk-probing phase (lines 13 to 31) - and cache eviction process which runs intermittently, after every ten iterations in our case (lines 32 to 42). These two phases alternate in the outer loop.

For every input the algorithm loads the number of stream tuples equal to the number slots vacant in H_S from the previous iteration. We use variable *hSavailable* to count this number. At the beginning of the algorithm, all slots in H_S are vacant (line 1).

In the stream-probing phase the algorithm continues reading stream input until stream tuples in S_B and vacant slots in H_S are available (line 3). For each of these stream tuples t the algorithm probes it in the disk-built hash table H_R (line 5). If t matches in H_R, the algorithm enriches the master data attributes to t, generates the join output(s), and increases the stream tuple frequency f_count for join value $t.partID$. Since H_R is a multi-hashmap, there can be more than one match (lines 6 and 7). In case if t does not find any matched, the algorithm forwards t to the disk-probing phase, i.e. loads t into H_S and enqueues its pointer $t.partID$ (also called join attribute value) in Q. Also the algorithm decreases counter *hSavailable* by one (lines 8 to 11). Once the stream probing phase reaches to its end the algorithm switches to the disk probing phase. This switching is the matter of switching a program thread from one process to other, so it takes very minimal time. The value of this switching time is too little that it is almost invisible in cost processing. Consequently, its influence on the algorithm is ignorable.

Lines 13 to 31 comprise the disk-probing phase. The algorithm reads the oldest join attribute value $t.partID$ from Q and search it in the value(s) in R. If no match found, the algorithm deletes $t.partID$ from Q and moves back to line 13. This is the point where we can say the algorithm also supports

$0 - to - many$ type of join (lines 13 to 17). Otherwise, the algorithm reads a segment of R using $t.partID$ as an index and load this into D_B (line 18). One point to clarify here is that because of the join accuracy the algorithm reads all *partIDs* provided by one or more suppliers in one segment which means the algorithm doesn't read *partIDs* of one supplier in two different segments. Therefore, the size of D_B is slightly flexible with respect to the variation in the segment size.

In an inner loop (lines 19 to 31) the algorithm reads one-by-one each tuple r from D_B and looks it up in H_S. If r matches in H_S, the algorithm enriches the master data attributes to the matched tuple and generates the join output(s); there can be more than one match (lines 20 and 21). Lines 22 and 27 deal with the memory calculations for processing matching stream tuples against disk tuples r in D_B in both of the phases and implementing cache inequality. In line 22 variable m_{disk} computes the memory for processing all matching stream tuples in H_S against $r.partID$ in D_B. As there can be more than one same *partIDs* in D_B provided by different suppliers, this is the case where we need to use many-to-many type of join. While in line 23 variable m_{stream} computes the memory required if these stream tuples are processed through stream-probing phase. Based on these two memory calculations the cache inequality is applied (line 24). If the cache inequality recommends switching, in this case the algorithm first releases the equal amount of memory from the stream-probing phase by reducing the size of next input, *hSavailable*, and then loads all R records for that join attribute $r.partID$ into H_R (line 25 and 26).

At the end of the disk-probing phase the algorithm deletes all the matching tuples from H_S with their corresponding nodes from Q. Since H_S is a multi-hash-map, there can be multiple matches against one $r.partID$. This creates empty slots in H_S and therefore *hSavailable* has to increase (lines 28 and 29).

Lines 32 to 42 describe the *cache eviction process*. This can happen if a join value in disk-probing phase (i.e. in cache) gets less frequently used over time. The execution of this feature is not required in every outer loop iteration. In order to increase the quality of the frequency estimate, we consider the execution of this process after every 10th outer loop iteration. Again, the memory for the both phases needs to be calculated and based on this the cache inequality has to be applied (lines 34 to 36). For computing of m_{disk} the frequency count has to be divided by 10 since the count runs for 10 outer loop iterations. In line 36 where this comparison is true, the join value j has become infrequent and should be evicted. The algorithm simply deletes join value j from H_R with all its related tuples (line 37). In order to add the released memory to the stream-probing phase the algorithm increases *hSvariable* accordingly. If any stream tuple with j appears in the stream data it will be processed in the disk-probing phase. Finally, the algorithm resets $f\_count(s)$ for all join values in H_R so that the algorithm can start counting again for the next ten iterations (line 40).

V. SYSTEMS DEVELOPMENT AND EXPERIMENTAL SETUP

This section presents an experimental study where we compare the memory and service rate of MOIJ with CJ using synthetic and real-life data. We use different parameters as independent variables in order to obtain a range of service rate results that can be analyzed in different conditions. Through the experiments we also validate the cost models for both MOIJ and CJ. The following text describes the experimental setup.

A. MEASUREMENT STRATEGY

In each experiment, the both algorithms go under their warm-up phase which is normal for these type of algorithms that run for indefinite time. The warm-up phase is a phase where an algorithm tunes all its components with respect to the available resources e.g. available memory. We take the measurements after the warm-up phase ends, so that no transient effects from startup influence our measurements. In our experiment where required we calculated the 95% confidence interval based on the mean on at least 1000 runs for one setting. In some results presented here, the confidence interval is too small to show.

B. DATA SPECIFICATIONS

We used three data sets that we in the following refer to as synthetic, TPC-H, and real-life data. The characteristics of each data set are described below.

C. SYNTHETIC DATA

A dataset that we used was providing synthetic data with a configurable Zipfian distribution exponent, this was useful for tests where the zipfian exponent was an independent variable, since it was not possible to get e.g. real-life data for a given Zipfian exponent on demand, without changing that data. We used this data in experiments varying the exponent from 0 to 1 - at 0 the data is fully uniform while at 1 the data is highly skewed. In the data generator we needed a simple and natural way to ensure that the master data had duplicates e.g. in case MOIJ duplicate *partIDs*. The specifications of synthetic dataset that we used in our experiments are presented in Table 1.

D. TPC-H

As a second dataset we adapted data from the TPC-H benchmark. We used scale factor 100 for generating the benchmark. We generated two tables - *tpch_partsupp*[1] that we used as master data while *tpch_lineitem*[2] that we used as stream data. The join attribute was *partID* between both inputs. Both tables contained 20 million tuples. The size of each tuple in *tpch_partsupp* was 223 bytes and in *tpch_lineitem* was 138 bytes. The motivation of using this dataset was to show the performance of the algorithm on a standardized workload that was not designed or chosen by us. A possible

[1] contained parts details provided by each supplier.
[2] contained orders details against the available parts.

TABLE 1. Synthetic dataset specifications.

Parameter	Value
Total memory available M for the algorithm	1% of R (0.11GB) to 10% of R (1.12GB)
Size of R	100 million tuples (≈11.18GB)
Disk tuple size	120 *bytes*
Stream size	infinite
Stream tuple size	20 *bytes*
Size of each node in the queue	4 *bytes*
Dataset	based on Zipf's law (skew value from 0 to 1)

case for such a join is to enrich supplier details related to the *partIDs* found in an order, before loading that order into a data warehouse. Again for CJ we removed the duplicate *partIDs* from R because it has no support for many-to-many type of joins.

E. REAL-LIFE DATA

We also compared the service rate of MOIJ with CJ using a real-life dataset.[3] The dataset is a collection of weather reports for the entire globe. The original CJ [25] also used the same dataset to evaluate its performance. More specifically the master data table contained meteorological data for the months of April and August. The stream data was built based on the reports generated in December. The size of the master data 20 million tuples and the of stream data table was 6 million tuples. The tuple size was 128 bytes in the both master data and the stream data. The attribute used for joining the both tables was *longitude*. The join type was many-to-many and again for CJ we filtered out the duplicates from the longitude attribute.

F. COMPUTER SPECIFICATIONS

We used Pentium-i5 with 8GB main memory and APPLE SSD AP0512J 500GB hard drive with cache 256 MB in all our experiments. Both algorithms were implemented in Java. MOIJ needs both hash tables that can store several elements against the same key, but this feature is not supported by Java hash table. Therefore in order to implement the above feature we used the Apache library class Multi-Hash-Map. We used MySQL database with buffer pool size 1 MB to store the master data table while the table has an index on the join attribute e.g. *partID*. The table had a composite primary key e.g. on the attributes *partID* and *supplierID*. However in case of CJ we had one *partID* against each supplier as the algorithm dose not support many-to-many type of join.

VI. MATHEMATICAL EVALUATION - MEMORY AND PROCESSING COST MODELS

We calculated cost model in terms of memory and process time for our algorithm. The main motivation for developing a cost model is to interrelate the key parameters of the algorithm, such as the input size $w = w_N + w_S$ tuples, the processing cost c_{loop} seconds required to process these w tuples, the allocated memory M in bytes, and the service rate μ in tuples/second. We developed the cost model on the same styles used in original CACHEJOIN [25] and MESHJOIN [26]. Equation 1 represents the total memory (except S_B) consumed by the algorithm while Equation 2 represents the processing cost for each iteration of the algorithm. The symbols that we used in developing of the cost model are described in Table 2.

A. MEMORY COST

As described above two hash tables H_R and H_S are the main components of the algorithm, the major portion of the total memory is assigned to these two has tables together with Q. While a significantly smaller portion is assigned to D_B. We assigned 0.1 MB to D_B in all our experiments. The stream buffer is another tiny component (0.05 MB was sufficient in our experiments) which we included in the architecture diagram but is not included in the cost model.

An important point with respect to memory distribution in MOIJ is that the algorithm assigns memory to both hash tables (H_R and H_S) dynamically. We start the algorithm by assigning a certain amount of memory to the each hash table. The algorithm then optimises this using cache inequality as described in the algorithm section. This is the reason that the algorithm is adaptive for any change in the stream or the master data. In general the size of H_R depends upon the skew in stream data. For greater skew in stream the size of H_R will be comparatively bigger while in case of completely uniform stream data the size of H_R will be zero as it does not play any role in the performance of the algorithm. This shows MOIJ is an optimal algorithm for uniform stream data. In the following we calculate the memory for each component when total memory available to assign is M.

$$\begin{aligned} H_R \text{ (bytes)} &= h_R \cdot v_R \\ D_B \text{ (bytes)} &= b \cdot v_R \\ H_S \text{ (bytes)} &= \alpha(M - (b + h_R)v_R) \\ \text{Queue (bytes)} &= (1 - \alpha)(M - (b + h_R)v_R) \end{aligned}$$

Now after having memory for each component total memory M can be calculated as shown in Equation 1.

$$M = h_R \cdot v_R + b \cdot v_R + \alpha(M - (b + h_R)v_R) + (1 - \alpha) \times (M - (b + h_R)v_R) \quad (1)$$

[3]This dataset is available at: http://cdiac.ornl.gov/ftp/ndp026b/

TABLE 2. Symbols used in the cost model of MOIJ.

Parameter	Symbol
Total memory used by the algorithm *(bytes)*	M
Service rate *(processed tuples/sec)*	μ
Stream arrival rate *(tuples/sec)*	λ
#stream tuples matched in the stream-probing phase in each iteration	w_N
#stream tuples matched in the disk-probing phase in each iteration	w_S
Stream tuple size *(bytes)*	v_S
Disk tuple size *(bytes)*	v_R
D_B size=one segment of R *(tuples)*	b
H_R size *(tuples)*	h_R
Memory weight for H_S	α
Memory weight for Q	$1-\alpha$
Cost for reading b number of disk tuples into D_B *(nanosecs)*	$c_{I/O}(b)$
Cost for probing one tuple either in H_R or H_S *(nanosecs)*	c_H
Cost for generating the output for one stream tuple *(nanosecs)*	c_O
Cost for executing cache inequity and switching a disk tuple to H_R *(nanosecs)*	c_I
Cost for deleting one tuple from H_S and Q *(nanosecs)*	c_E
Cost for reading one stream tuple into S_B *(nanosecs)*	c_S
Cost for adding one stream tuple in H_S and Q *(nanosecs)*	c_A
Cost for executing the eviction process for one tuple in H_R *(nanosecs)*	c_V
Total cost required for one loop iteration *(secs)*	c_{loop}

B. PROCESSING COST

Processing cost refers to the time taken by the algorithm for each iteration. For the simplicity we first calculate the cost for each component separately and then aggregate these costs to get the total cost.

Cost to load b number of tuples from disk to D_B (nanosecs) $= c_{I/O}(b)$

Cost to read the w_N tuples from S_B (nanosecs) $= w_N \cdot c_S$

Cost to read the w_S tuples from S_B (nanosecs) $= w_S \cdot c_S$

Cost to look-up w_N tuples in H_R (nanosecs) $= w_N \cdot c_H$

Cost to append w_S tuples into H_S and Q (nanosecs) $= w_S \cdot c_A$

Cost to look-up D_B tuples in H_S (nanosecs) $= b \cdot c_H$

Cost to execute the cache inequality and switching of disk tuples to H_R if necessary (nanosecs) $= b \cdot c_I$

Cost to generate the output for w_N tuples (nanosecs) $= w_N \cdot c_O$

Cost to generate the output for w_S tuples (nanosecs) $= w_S \cdot c_O$

Cost to delete w_S tuples from H_S and Q (nanosecs) $= w_S \cdot c_E$

Cost to execute the cache eviction process - after every ten outer loop iteration (nanosecs) $= \frac{h_R \cdot c_V}{10}$

Now Equation 2 sums up the above costs to determine the total cost that the algorithm required to complete its one loop iteration .

$$\begin{aligned} c_{loop}(secs) = 10^{-9}[&c_{I/O}(b) + w_N(c_S + c_H + c_O) \\ &+ w_S(c_S + c_A + c_O + c_E) + b(c_H + c_I) \\ &+ \frac{h_R \cdot c_V}{10}] \end{aligned} \tag{2}$$

The term 10^{-9} is used to convert nanoseconds to seconds. Since the algorithm processes w_N and w_S tuples in c_{loop} seconds, the service rate μ can be determined as in Equation 3.

$$\mu = \frac{w_N + w_S}{c_{loop}} \tag{3}$$

The proposed algorithm computes the exact join between a stream and master data provided that $\lambda \leq \mu$. By substituting the value of μ from Equation (3):

$$\lambda \leq \frac{w_N + w_S}{c_{loop}} \tag{4}$$

The minimum values of w_N and w_S are specified by Equation 4 as follows:

$$\lambda = \frac{w_N + w_S}{c_{loop}} \tag{5}$$

From Equation (1) the memory required to process w_N and w_S tuples can be calculated as follows:

$$\begin{aligned} \text{Memory for } w_N &= h_R \cdot v_R \\ \text{Memory for } w_S &= b \cdot v_R + \alpha(M - (b + h_R)v_R) \\ &\quad + (1-\alpha)(M - (b + h_R)v_R) \end{aligned}$$

By substituting the amount of memory taken by w_N and w_S in Equation 5, λ can be written as follows:

$$\begin{aligned} \lambda = \frac{1}{c_{loop}}[&h_R \cdot v_R + b \cdot v_R + \alpha(M - (b + h_R)v_R) \\ &+ (1-\alpha)(M - (b + h_R)v_R)] \end{aligned} \tag{6}$$

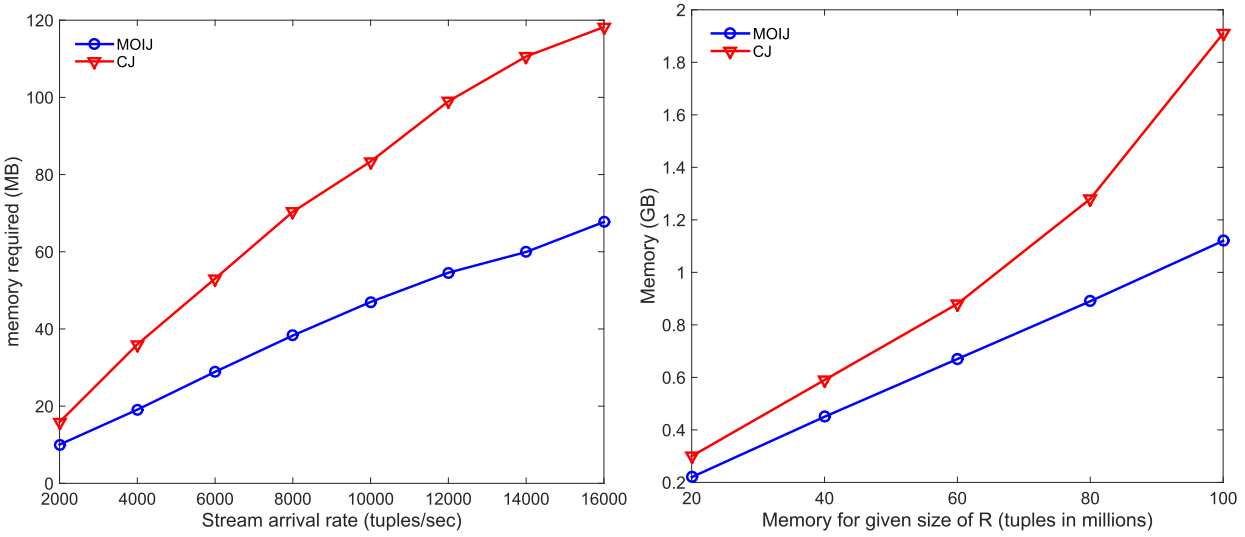
This cost model can be used to make later comparisons with the measured behavior of the algorithm.

VII. EMPIRICAL EVALUATION

We compare MOIJ with CJ based on two parameters, memory and service rate. This section presents our analysis against both parameters.

A. MEMORY ANALYSIS

The new algorithm adapts its memory consumption to the stream arrival rate, therefore in our first experiment we observed how much memory MOIJ can save compared to CJ



(a) Memory for given stream arrival rate

(b) Memory for given size of R

FIGURE 6. Memory consumption measurements.

while producing the same service rate. We also compared the memory consumption by MOIJ with CJ considering the same sizes of the master data. In this experiment we used the synthetic data, since we wanted to achieve a clear understanding of the effect of a well-defined Zipfian exponent (set to 1) on the algorithm behavior.

The results of our first experiment by varying the stream arrival rate are presented in Figure 6(a). In this experiment we fixed the size of R (denoted by $|R|$) to 20 million tuples, and the value of Zipfian exponent is equal to 1. The stream arrival rate was controlled by varying the parameters of our data generator. The results in the figure show that CJ needs about 73% more memory than MOIJ to produce the same service rate under all stream arrival rate settings. The results in Figure 6(b) are obtained by varying $|R|$, while keeping a fixed service rate of 50000 tuples/sec. In this case CJ needs about 36% more memory than MOIJ for $|R| = 20$ million tuples and about 70% more for $|R| = 100$ million tuples. This is due to the fact of the Zipfian distribution, a larger master data table contains more frequent tuples while they all are not cached in case of CJ while they are managed to cache in MOIJ due to memory adaptation.

B. SERVICE RATE ANALYSIS

We evaluated the service rate (which is the dependent variable) by varying three key parameters - available memory, $|R|$, and the intensity of skew in the streaming data. To see the effect of individual parameter on the service rate we varied one parameter at a time. In experiments presented in Figures 7(a)–(c) and (f) we used the synthetic data.

1) MEMORY SIZE VARIES

In this experiment we evaluated the service rate for different settings of memory. We varied the available memory size from 1% to 10% of $|R|$ While fixed $|R|$ at 100 million tuples ($\approx$11.18GB) and the skew in the streaming data is equal to 1.

Figure 7(a) depicts the results of our experiment. From the figure we observed that MOIJ was $\approx$1.5 times faster than CJ with very limited memory (1% of $|R|$) and $\approx$ 2 times faster for 10% of $|R|$. This shows that the impact of the cache increases slightly with larger memory.

2) $|R|$ VARIES

In this experiment we evaluated the service rate by varying $|R|$ from 20 million tuples to 100 million tuples. We kept the other two parameters fixed, memory size is equal to $\approx$1.12GB and the value of the skew is equal to 1. The results presented in Figure 7(b) show that MOIJ performed $\approx$ 2 times better than CJ for $|R| = 20$ million tuples and $\approx$1.7 times better for $|R| = 100$ million tuples.

3) SKEW VARIES

Finally we varied the value of skew in the streaming data between 0 to 1. At the skew value is equal to 0 the input stream S was fully uniform, while at the skew value is equal to 1 the stream was skewed (non-uniform). The other two parameters $|R|$ and the available memory were fixed to 100 million tuples ($\approx$11.18GB) and to 10% of $|R|$ ($\approx$1.12GB) respectively. Figure 7(c) depicts the results of the experiment. It is clear from the figure that MOIJ starts performing better as soon as the skew appears in the stream data and this improvement becomes more evident and visible as the value of skew increases. For the high values of skew (i.e. equal to 1) MOIJ performs $\approx$2 times better than CJ. Although CJ also exploits the feature of skew in the stream data but due to the suboptimal distribution of the memory between the two join phases the algorithm can not perform at its maximum. We did not evaluate the performance for skew value is higher

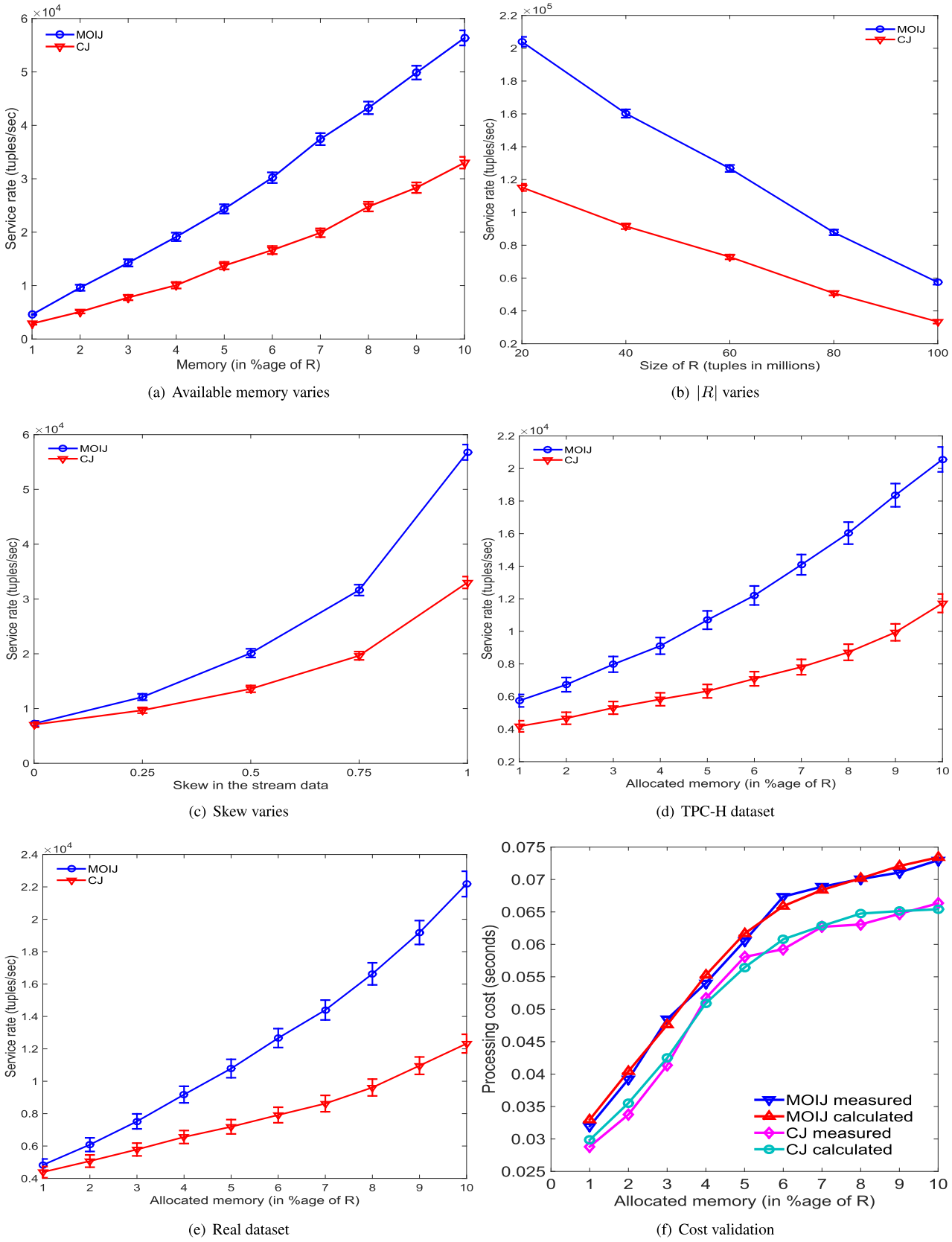


FIGURE 7. Service rate analysis and cost validation.

than 1, as the higher skew values would imply comparatively short tails. However, we presume that the improvement in the service rate will continue for such short tails.

4) TPC-H AND REAL-LIFE DATASETS

We also compared the service rates of the both algorithms using the TPC-H and the real-life weather datasets (see Section V). For both algorithms we compared the service rate under different memory settings, varied from 1% of $|R|$ to 10% of $|R|$. Figures 7(d) and (e) present the service rate generated by the both algorithms using TPC-H and real-life datasets respectively. From Figure 7(d) it can be observed that, under memory size 10% of $|R|$, MOIJ performs about $\approx$1.8 times better than CJ. Similarly in case of the real-life weather dataset, from Figure 7(e), MOIJ outperforms CJ under all memory settings. An interesting aspect of these measurements is that the TPC-H data as well as the real-life data produce very similar behavior to our synthetic dataset. This gives us confidence that the use of synthetic data is in itself justified and relevant. Moreover the behavior for both the datasets matches quite closely to the behavior for the skew is equal to one in the synthetic data. This is not unexpected, since such a moderately strong skew is, as we stated at the very beginning, is a good rule of thumb for many real life situations.

C. COST VALIDATIONS

Finally, we validated the cost models for both algorithms where we compared the calculated cost for a central cost parameter, c_{loop}, with measurements of c_{loop}. Figure 7(f) depicts the results of the experiment. From the figure it is clear that in case of both of the algorithms the calculated cost is very close to the measured cost. This is a proof for the correctness of our cost models.

VIII. CONCLUSION

Maximizing the value of an event in streaming data scenarios is closely correlated with minimizing the data analysis and decision latencies. The growing amount of streaming big data applications and the processing time of storing this streaming data into the data warehouse poses a performance challenge for *minimizing data latency*. Traditional approaches have used a variety of hash based stream joins during the ETL transformation phase. During this phase a semi-stream join operator is commonly used to join streaming data with master data. Most semi-stream join operators cache frequent parts of master data to improve their performance, this process requires careful allocation of memory to each component of the join operator.

In this article we developed a new ''cache inequality'' for semi-stream index-based joins. The new cache inequality can be applied to any cache-based semi-stream indexed join for the optimal caching. To test the cache inequality we presented a novel algorithm called MOIJ which supports many-to-many type of join and to the best of our knowledge no semi-stream indexed join in the literature provides this support. The new algorithm adapts to online changes in both stream as well as the master data. We tested this by varying the intensity of the skew in the stream data and the total size of the master data. We evaluated the service rate of the new algorithm using synthetic, TPC-H, and real data and our results show that the new algorithm significantly outperforms existing algorithm called CJ under all three data sets. We also derived and validated the cost model for our algorithm.

In future we have a plan to apply our cache inequality to other semi-stream joins. We are also aiming to parallelize our MOIJ algorithm for processing of multiple inputs of stream data with distributed master data placed at multiple nodes.

REFERENCES

[1] L. Fink, N. Yogev, and A. Even, ''Business intelligence and organizational learning: An empirical investigation of value creation processes,'' *Inf. Manage.*, vol. 54, no. 1, pp. 38–56, Jan. 2017.

[2] S. Piramuthu and M. J. Shaw, ''Learning-enhanced adaptive DSS: A design science perspective,'' *Inf. Technol. Manage.*, vol. 10, no. 1, pp. 41–54, Mar. 2009.

[3] I. Bose and R. Pal, ''Predicting the survival or failure of click-and-mortar corporations: A knowledge discovery approach,'' *Eur. J. Oper. Res.*, vol. 174, no. 2, pp. 959–982, Oct. 2006.

[4] I. Bose and R. K. Mahapatra, ''Business data mining—A machine learning perspective,'' *Inf. Manage.*, vol. 39, no. 3, pp. 211–225, 2001.

[5] I. Bose, L. A. K. Chun, L. V. W. Yue, L. H. W. Ines, and W. O. L. Helen, ''Business data warehouse,'' in *Proc. Data Mining Appl. Empowering Knowl. Societies*, 2009.

[6] R. Hackathorn, ''Real-time to real-value,'' *Inf. Manage.*, vol. 14, no. 1, p. 24, 2004.

[7] Y.-T. Park, ''An empirical investigation of the effects of data warehousing on decision performance,'' *Inf. Manage.*, vol. 43, no. 1, pp. 51–61, Jan. 2006.

[8] A. Karakasidis, P. Vassiliadis, and E. Pitoura, ''ETL queues for active data warehousing,'' in *Proc. 2nd Int. Workshop Inf. Qual. Inf. Syst. (IQIS)*, New York, NY, USA, 2005, pp. 28–39.

[9] M. A. Naeem, G. Dobbie, and G. Weber, ''An event-based near real-time data integration architecture,'' in *Proc. 12th Enterprise Distrib. Object Comput. Conf. Workshops (EDOCW)*, Washington, DC, USA, 2008, pp. 401–404.

[10] X. Han, J. Li, and D. Yang, ''PI-join: Efficiently processing join queries on massive data,'' *Knowl. Inf. Syst.*, vol. 32, no. 3, pp. 527–557, Sep. 2012.

[11] M. Townsend, T. Le Quoc, G. Kapoor, H. Hu, W. Zhou, and S. Piramuthu, ''Real-time business data acquisition: How frequent is frequent enough?'' *Inf. Manage.*, vol. 55, no. 4, pp. 422–429, Jun. 2018.

[12] R.-U.-D. Faizal, R. Doss, and W. Zhou, ''String matching query verification on cloud-hosted databases,'' in *Proc. 17th Int. Conf. Distrib. Comput. Netw. (ICDCN)*, New York, NY, USA, 2016, pp. 17:1–17:10.

[13] S. Piramuthu, ''Knowledge-based Web-enabled agents and intelligent tutoring systems,'' *IEEE Trans. Educ.*, vol. 48, no. 4, pp. 750–756, Nov. 2005.

[14] I. Bose, W. Ping, M. Shan, W. Shing, Y. Shing, C. Tin, and S. Wai, *Databases for Mobile Applications*. Hershey, PA, USA: IGI Global, 2005.

[15] F. Riaz-ud-Din, W. Zhou, and R. Doss, ''Query verification schemes for cloud-hosted databases: A brief survey,'' *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 31, no. 6, pp. 543–561, Nov. 2016.

[16] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik, ''Efficient resumption of interrupted warehouse loads,'' *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 46–57, Jun. 2000.

[17] X. Zhang and E. A. Rundensteiner, ''Integrating the maintenance and synchronization of data warehouses using a cooperative framework,'' *Inf. Syst.*, vol. 27, no. 4, pp. 219–243, Jun. 2002.

[18] R. M. Bruckner, B. List, and J. Schiefer, ''Striving towards near real-time data integration for data warehouses,'' in *Proc. Int. Conf. Data Warehousing Knowl. Discovery*. Berlin, Germany: Springer, 2002, pp. 317–326.

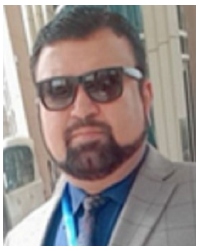
[19] M. N. Tho and A. M. Tjoa, ''Zero-latency data warehousing for heterogeneous data sources and continuous data streams,'' in *Proc. 5th Int. Conf. Inf. Integr. Web-Based Appl. Services*, 2003, pp. 55–64.

[20] F. Araque, ''Real-time data warehousing with temporal requirements,'' in *Proc. CAiSE Workshops*, 2003.

[21] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk, "Stream warehousing with datadepot," in *Proc. 35th SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2009, pp. 847–854.
[22] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica, "Online updates on data warehouses via judicious use of solid-state storage," *ACM Trans. Database Syst.*, vol. 40, no. 1, p. 6, 2015.
[23] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Reading, MA, USA: Addison-Wesley, 2004.
[24] M. A. Naeem, G. Weber, and C. Lutteroth, "A memory-optimal many-to-many semi-stream join," *Distrib. Parallel Databases*, vol. 37, pp. 623–649, Aug. 2018.
[25] M. A. Naeem, G. Dobbie, and G. Weber, "A lightweight stream-based join with limited resource consumption," in *Proc. Data Warehousing Knowl. Discovery (DaWaK)*. Berlin, Germany: Springer, 2012, pp. 431–442.
[26] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell, "Meshing streaming updates with persistent data in an active data warehouse," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 7, pp. 976–991, Jul. 2008.
[27] M. A. Naeem, G. Dobbie, G. Weber, and S. Alam, "R-MESHJOIN for near-real-time data warehousing," in *Proc. ACM 13th Int. Workshop Data Warehousing (DOLAP)*, Toronto, ON, Canada, 2010, pp. 53–60.
[28] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. New York, NY, USA: McGraw-Hill, 2000.
[29] A. Chakraborty and A. Singh, "A partition-based approach to support streaming updates over persistent data in an active datawarehouse," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Washington, DC, USA, May 2009, pp. 1–11.
[30] M. A. Bornea, A. Deligiannakis, Y. Kotidis, and V. Vassalos, "Semi-streamed index join for near-real time execution of ETL transformations," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 159–170.
[31] R. Derakhshan, A. Sattar, and B. Stantic, "A new operator for efficient stream-relation join processing in data streaming engines," in *Proc. 22nd ACM Int. Conf. Conf. Inf. Knowl. Manage.*, 2013, pp. 793–798.
[32] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, A. Gupta, L. Haas, K. Kim, C. Lee, G. Mundada, M.-C. Shan, N. Tatbul, Y. Yan, B. Yun, and J. Zhang, "A demonstration of the MaxStream federated stream processing system," in *Proc. IEEE 26th Int. Conf. Data Eng. (ICDE)*, 2010, pp. 1093–1096.
[33] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, 2015.
[34] M. A. Naeem, G. Dobbie, and G. Weber, "HYBRIDJOIN for near-real-time data warehousing," *Int. J. Data Warehousing Mining*, vol. 7, no. 4, pp. 21–42, Oct. 2011.
[35] M. A. Naeem, G. Weber, C. Lutteroth, and G. Dobbie, "Optimizing queue-based semi-stream joins with indexed master data," in *Data Warehousing and Knowledge Discovery*. Berlin, Germany: Springer, 2014, pp. 171–182.
[36] J. F. Nunamaker, M. Chen, and T. D. M. Purdin, "Systems development in information systems research," *J. Manage. Inf. Syst.*, vol. 7, no. 3, pp. 89–106, Dec. 1990.
[37] R. H. Von Alan, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quart.*, vol. 28, no. 1, pp. 75–105, 2004.

M. ASIF NAEEM (Member, IEEE) received the master's degree (Hons.) in computer science, and the Ph.D. degree in computer science from The University of Auckland, New Zealand. He is currently the Founder of the Data Science Research Group (DSRG), Auckland University of Technology, and the Co-Director of the Intelligent Knowledge Mining and Analytics (IKMA) Laboratory, National University of Computer and Emerging Sciences (NUCES), Pakistan. He is also a Professor with the School of Computing, NUCES. He has more than 15 years research, industrial, and teaching experience. He has published over 80 research papers in high-repute journals, conferences, and workshops, including IEEE, ACM, and VLDB. His research interests include data stream processing, real-time data warehousing, data science, big data management, and knowledge engineering. He has been awarded the Best Ph.D. Thesis Award from The University of Auckland.

FARHAAN MIRZA (Member, IEEE) is currently a Researcher and a Lecturer with the Department of IT and Software Engineering, Auckland University of Technology. His passion is to use the Internet of Things (IoT), mobile apps, and web technologies along with big data to develop applications for public sector development, specifically towards domains of healthcare, transportation, education, and telecommunications.



HABIB ULLAH KHAN (Member, IEEE) received the Ph.D. degree in management information systems from Leeds Becket University, U.K. He is currently an Associate Professor of MIS with the Department of Accounting and Information Systems, College of Business and Economics, Qatar University, Qatar. He has more than 20 years of industry, teaching, and research experience. He is an Active Researcher and his research work has published in leading journals of the MIS field. His research interests include IT security, online behavior, IT adoption in supply chain management, Internet addiction, mobile commerce, computer mediated communication, IT outsourcing, big data, cloud computing, and e-learning. He is a member of leading professional organizations like DSI, SWDSI, ABIS, FBD, and EFMD. He is a reviewer of leading journals of his field and also working as an editor for some journals.

DAVID SUNDARAM is an Engineer by background, a Teacher, a Researcher, a Consultant by profession, and a lifelong Student. He is passionate about the modeling, design, and implementation of flexible and evolvable information, decision, knowledge, and social systems. Exploration and application of these to the architecting and design of learning, adaptive, agile, and sustainable enterprises and societies is close to his heart.

NOREEN JAMIL received the Ph.D. degree in computer science from The University of Auckland, New Zealand. She is currently an Associate Professor with the Department of Computer Science, National University of Computer and Emerging Sciences (NUCES). As a part of her Ph.D. Programme, she was a Visiting Research Fellow with the Department of Mathematics, University of Maryland, USA. She has more than ten years academic and research experience at university level. She has published more than 25 research papers in well-reputed conferences and journals, including IEEE and Elsevier. She has published two articles in *Journal of Computational and Applied Mathematics*, one of the world leading Computational Mathematics Journals. Her research interests include computational mathematics, human–computer interaction, numerical computation, and constraint programming. She has received the Best Paper Award in IEEE-ICDIM 2013 and the Best Student Paper Award at The University of Auckland, in 2013.

GERALD WEBER received the Ph.D. degree from FU Berlin. He is currently a Senior Lecturer with the Department of Computer Science, The University of Auckland. He joined The University of Auckland, in 2003. He is also the Information Director of the Proceedings of the VLDB Endowment. He is a coauthor of the book *Form-Oriented Analysis*, and of over 40 peer-reviewed publications. His research interests include databases and data models, human–computer interaction, and theory of computation. He has been the Program Chair of several conferences.