PAPER

Efficient Parallel Join Processing Exploiting SIMD in Multi-Thread Environments*

Gilseok HONG[†a)], **Seonghyeon KANG**[†b)], **Chang soo KIM**[††c)], ***Nonmembers***, ***and*** **Jun-Ki MIN**[†d)], ***Member***

SUMMARY In this paper, we study parallel join processing to improve the performance of the merge phase of sort-merge join by integrating all parallelism provided by mainstream CPUs. Modern CPUs support SIMD instruction sets with wider SIMD registers which allows to process multiple data items per each instruction. Thus, we devise an efficient parallel join algorithm, called *Parallel Merge Join with SIMD instructions* (*PMJS*). In our proposed algorithm, we utilize data parallelism by exploiting SIMD instructions. And we also accelerate the performance by avoiding the usage of conditional branch instructions. Furthermore, to take advantage of the multiple cores, our proposed algorithm is threaded in multi-thread environments. In our multi-thread algorithm, to distribute workload evenly to each thread, we devise an efficient workload balancing algorithm based on the kernel density estimator which allows to estimate the workload of each thread accurately.

key words: *sort-merge join, SIMD, kernel density estimator, multi-thread*

1. Introduction

For the past decades, the technology of microprocessors has been progressed tremendously. Of particular, the single-instruction-multiple-data (*SIMD*) technology was introduced [7]. On this technology, by using comprehensive SIMD instruction sets, the computational performance of a system can be improved since SIMD instructions with wider CPU registers, called SIMD registers, process more data items per an instruction in parallel.

The advent of such microprocessor technologies is making a profound impact on software development. Today's microprocessors provide three sources of parallelism: *thread parallelism*, *instruction level parallelism* and *data parallelism* [22]. Thread parallelism is achieved by executing multiple threads on CPUs. The pipeline architecture [16] provides instruction level parallelism in which an instruction is divided into several stages and multiple instructions with different stages each other are executed simultaneously. Data parallelism is achieved by adapting SIMD instructions to the basic operations of a system. Note that, for utilizing all parallelism to achieve the optimal performance, the traditional algorithms should be tailored to the processing devices. That is a non-trivial and challenging task.

In this work, we study parallel join processing to improve the performance of a join operation by integrating all parallelism. Modern database queries and data mining applications are very data-intensive and thus demand high computing power due to the rapid growth of data volume. Among diverse operations in database systems such as selection, projection and aggregation, join is an expensive and key operation that facilitates the combination of two relations based on a pair of join attributes. Thus, an efficient implementation of a join operation will improve the performance of database systems and its diverse applications. In [15], the authors showed that sort-merge join benefits greatly by exploiting the SIMD technology and its performance will continue to improve with the trend of wider SIMD registers. We thus devise an efficient parallel join algorithm, called *Parallel Merge Join with SIMD instructions* (abbreviated by *PMJS*), based on the sort-merge join algorithm.

The sort-merge join algorithm consists of *sort phase* and *merge phase*. In the sort phase, the tuples of each relation participated in a join operation are sorted according to their join attribute. In the merge phase, every pair of tuples, each of which is coming from each relation, satisfying a join condition is generated as a join result. In this work, we only focus on parallel processing of the merge phase of the sort-merge join algorithm since, although there are several effective sorting algorithms [10], [14], [23] by using SIMD instructions, there is still room for the performance improvements in the merge phase. The contributions of our work are summarized as follows:

Integrating all parallelism: In our proposed algorithm, we utilize data parallelism by exploiting SIMD instructions handling up to 256-bit sized SIMD registers. On the pipeline architecture providing instruction level parallelism, conditional branch instructions are problematic since, if the branch prediction is wrong, the instruction pipeline should be flushed and various bookkeepings are required to ensure consistent operations [28]. Thus, this misprediction seriously degrades the overall performance of the merge phase. We thus accelerate the performance of the merge phase by avoiding the usage of conditional branch instructions. Furthermore, to take advantage of the multiple cores, our proposed algorithm is threaded in multi-thread environments.

Manuscript received September 21, 2017.
Manuscript publicized December 14, 2017.
[†]The authors are with Korea Univ. of Tech. & Edu., Korea.
[††]The author is with ETRI, Korea.
*This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea goverment (MSIT) (No. R0113-15-005, Development of an Unified Data Engineering Technology for Large-scale Transaction Processing and Real-time Complex Analytics).
a) E-mail: remocon33@koreatech.ac.kr
b) E-mail: overs2002@koreatech.ac.kr
c) E-mail: cskim7@etri.re.kr
d) E-mail: jkmin@koreatech.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2017EDP7300

Effective Workload Balancing: Skewed workload distribution can reduce parallelism in some naive approaches. Thus, for the best performance, we devise the workload balancing algorithm to make the workloads of all threads to be similar. Since our workload balancing problem is the same as the multiprocessor scheduling problem [13] which is known as NP-Hard, we devise an effective approximation algorithm. In addition, we use the *kernal density estimator* [26] to estimate the workload of each thread.

Extensive Performance Study: To demonstrate the efficiency of our *PMJS*, we implemented *PMJS*, its variants and scalar merge-join algorithm as well as conducted extensive performance study. Our experimental results confirm that *PMJS* is very efficient compared to the others.

2. Related Work

Over the past few decades, significant efforts have been made to develop efficient join algorithms [8], [18], [19] since join is an important but expensive operation in databases. Among the diverse join algorithms, sort-merge join [4] and hash join [6] are the representative algorithms for computing equi-join for a pair of relations. For a long time, choosing one of them has been a point of discussion in a database field.

In early relational database systems, the sort-merge join algorithm was dominantly used [20]. Later, the invention of the hash join algorithm changed the balance since hash join outperforms sort-merge join in many situations. With regards to the choice of a join algorithm, Graefe et al. [12] compared sort-merge join and hash join and recommended that both algorithms be included in a DBMS and be chosen by a query optimizer based on the statistics of relations. The hash join algorithm is a natural choice when the size of two relations differ markedly. They also showed that the skewed data distribution degrades the performance of hash join.

Progression of microprocessor, memory and network technologies as well as the rapid growth of data volume have prompted researchers to debate the *sort or hash* question again over the years [2], [21]. Schneider et al. [25] compared the performances of both algorithms in distributed environments and concluded that the hash join algorithm is superior unless memory was limited. Hash join was also the main choice in most of the early parallel database systems [9]. In addition, as the capacity of main memory has increased over the years, researchers have focused on main-memory join operations [3], [5]. Recently, researchers have explored new architectures to improve join performance. Gedik et al. [11] proposed a join algorithm running on the Cell processors. The above work tries to exploit the parallel nature of these devices with associated high compute density and bandwidth as well as shows significant performance benefits over optimized CPU-based counterparts.

Current trend in general purpose CPUs is in the direction of increasing parallelism, both in terms of the number of cores on a chip and the width of SIMD registers on each core. In [15], parallel hash join and sort-merge join exploiting both SIMD instructions and multiple threads were evaluated. They concluded that wider SIMD registers will soon make sort-merge join a better choice. However, in [15], efficient sorting with SIMD instructions and multi-threading are only considered in sort-merge join. In other words, efficient merge join and workload balancing techniques for each thread are not introduced in [15].

To compute an intersection of two sorted sets, some parallel algorithms utilizing SIMD instructions were proposed in [17], [24]. The authors in [24] insist that the proposed algorithms can be applied to a join operation. As a result of a join operation, each pair of tuples (or a pair of tuple id) satisfying a join condition should be generated. However, since each value appeared in both sets is generated as a result of intersection, the above algorithms cannot be used directly for a join operation. Moreover, since the set type does not allow duplications, we cannot use the above algorithms for the general join operation in which join attribute values of each relation could be duplicated.

The most related work of ours is *P-MPSM* [1] in which the authors showed that sort-merge join is faster than hash join in multi-core processors. In *P-MPSM*, to perform a join of relations R and S, relation S is split into equi-sized t chunks $S_1, \ldots, S_t$ where t is the number of threads and each chunk is sorted locally. For workload balancing, equi-sized chunk of R is also distributed to each thread and each thread builds a histogram from its assigned chunk. Then, by merging all generated histograms into a single histogram, the distribution of relation R can be identified. With respect to the consolidated histogram, *P-MPSM* computes disjoint t ranges of join attribute such that every workload of each thread becomes similar each other. According to the computed join attribute ranges, *P-MPSM* partitions the relation R into $R_1, \ldots, R_t$ as well as splits each chunk S_i into $S_{i1}, \ldots, S_{it}$ according to the same key ranges for workload balancing. Subsequently, each partition R_j of R is broadcast to every thread and each thread T_i performs the merge join of R_j and S_{ij}. Since every tuple in R_j is compared with that of S_{ij} rather than that of S_i, the performance merge join can be improved. However, in *P-MPSM*, to generate join results, each thread has to receive all partitions of R and retrieve every tuple in R. Furthermore, in the work of *P-MPSM*, data parallelism and instruction level parallelism are not considered. In contrast, we integrate all parallelism as mentioned in Introduction as well as each thread of our proposed algorithms requires a pair of disjoint partitions from R and S, respectively, to generate join results.

3. Preliminary

Our algorithm PMJS is based on the sort-merge join algorithm and extensively uses SIMD instructions to achieve data parallelism. Thus, in this section, we briefly explain the sort-merge join algorithm and SIMD technology.

```
Procedure sort-merge-join(R, S)
begin
//Sort phase
1.  if(R is not sorted on R.a) sort R on R.a
2.  if(S is not sorted on S.b) sort S on S.b
//Merge phase
3.  i=1, j=1
4.  while(i ≤ |R| and j ≤ |S|) {
5.    if(R[i].a == S[j].b) {
6.      jj = j
7.      while(R[i].a == S[jj].b and jj ≤ |S|) {
8.        output(R[i], S[jj])
9.        jj = jj + 1
10.     }
11.     i = i + 1
12.   } else if(R[i].a > S[j].b) {
13.     j=j + 1
14.   } else i=i + 1          //(R[i].C < S[j].C)
15. }
end
```

Fig. 1 A scalar sort-merge join algorithm

3.1 Scalar Sort-Merge Join

The basic idea of the sort-merge join algorithm is *sorting* the relations subjected to a join with respect to their join attributes and then *merging* the sorted relations by scanning them sequentially to generate qualified tuples. Figure 1 shows the pseudo-code of a scalar sort-merge join algorithm which takes a pair of relations R and S where the join attributes of R and S are a and b, respectively. The sort-merge join consists of two phases: sort phase and merge phase.

In the sort phase, scalar sort-merge join sorts R and S according to the join attributes $R.a$ and $S.b$ if they are not sorted (lines 1–2). In the merge phase, by sequential scanning the relation R and S (lines 3–15), every pair of tuples having the same join attribute value is generated. To do so, if the join attribute value of R's i-th tuple is equal to that of S's j-th tuple (line 5), the scalar sort-merge join algorithm bookmarks j as jj (line 6) since there may exist the tuples having the same join attribute value of the j-th tuple in the rest of relation S. Then, the pair of R's i-th tuple and S's jj-th tuple are generated by increasing jj until the join condition is satisfied (lines 7–10) and i is increased by one. If the join attribute value of R's i-th tuple is greater than that of S's j-th tuple, we increase j by one (lines 12–13). Otherwise, i is increased by one (line 14).

As shown in Fig. 1, the scalar sort-merge join algorithm contains several conditional statements. Such conditional statements lead to performance degradation resulting from pipeline stall by misprediction. Thus, we propose an efficient merge join algorithm by using SIMD instructions minimizing misprediction.

3.2 SIMD (Single Instruction Multiple Data)

Data parallelism is achieved by utilizing SIMD instructions each of which operates multiple data items at the same time. Most modern microprocessors provide SIMD instruction sets such as MMX, SSE 1/2/3/4.1/4.2, AVX 1/2/-512 in Intel CPUs and 3D Now, Enhanced 3D Now in AMD CPUs. By SIMD instructions, several data items are vectorized into a SIMD register. The size of SIMD registers becomes larger (e.g., 64 bits on MMX, 128 bits on SSE 1/2/3/4.1/4.2, 256 bits on AVX 1/2, and 512 bits on AVX-512 in Intel CPUs). Thus, more data items can be handled simultaneously by only one instruction. For instance, four 32-bit values can be loaded on a 128-bit SIMD register whereas eight 32-bit values can be loaded on a 256-bit SIMD register.

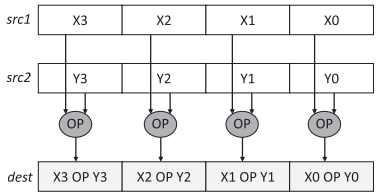


Fig. 2 A typical behavior of SIMD instructions

To process the input operands efficiently, SIMD instructions contains sufficient operations such as arithmetic operations (e.g., add, sum), comparison operations (e.g., less, string comparison), logic operations (e.g., and, or, shift), data movement (e.g., load, store) and miscellaneous operations (e.g., shuffling, type conversion). We illustrate a typical behavior of SIMD instructions in Fig. 2. Let us assume that a SIMD register can keep four data values. Then, a SIMD instruction takes two SIMD registers $src1$ and $src2$ as input, conducts element-wise operation OP in parallel and writes the results into an output register $dest$.

A direct way to use SIMD instructions is to inline assembly code. But, this way is tedious and error-prone. Another way is utilizing features of compilers which can partially transform scalar instructions to SIMD instructions. However, utilizing SIMD instructions fully is difficult for a compiler. The other approach is using SIMD intrinsics which allow us to use the syntax of C functions taking SIMD registers as inputs. In general, although direct implementation with assembly code outperforms that with SIMD intrinsics, we utilize SIMD intrinsics due to its convenience.

4. PMJS

In this section, we present our proposed join algorithm *PMJS* which utilizes SIMD instruction and multi-thread. In our implementation, we use conditional branches minimally to improve the performance of our algorithm. Furthermore, to maximize the performance of our algorithm in multi-thread environments, we devise the workload balancing technique which distributes the workload evenly to each thread. To estimate the workload assigned to each thread, we adopt the kernel density estimator [26].

4.1 Merge Join with SIMD

In our work, since we only focus on the merge phase in sort-merge join, we assume that a pair of relations R and S subject to a join operation are sorted with respect to their join attributes, respectively. In the scalar sort-merge join algorithm presented in Sect. 3, the join attribute value of R's i-th

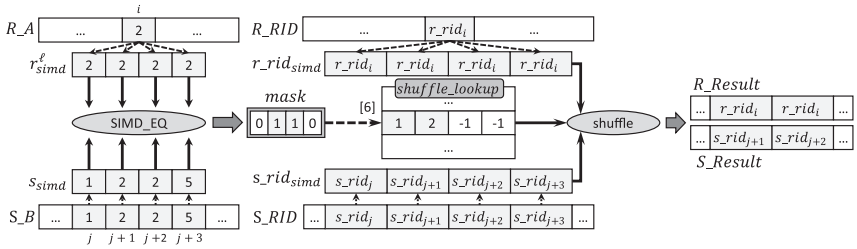


Fig. 3 The behavior of *PMJS*

tuple and that of S's j-th tuple are evaluated iteratively. On the contrary, by utilizing SIMD instructions, we can evaluate multiple pairs of join attribute values to generate join results in parallel.

Let R_A and S_B be the sorted lists of the join attribute values of R and S, respectively, as well as R_RID and S_RID be the lists of R's record ids and S's record ids whose orders are followed by R_A and S_B, respectively. We denote the i-th entry of a list L as $L[i]$. Assume that a CPU provides SIMD registers which can contain ℓ data values as well as the i-th entry of R_A (i.e., $R\_A[i]$) and j-th entry of S_B (i.e., $S\_B[j]$) will be evaluated. To identify the same values for $R\_A[i]$ among S_B's j-th to $(j+\ell-1)$-th entries (i.e., $S\_B[j..(j+\ell-1)]$) simultaneously, we keep $R\_A[i]$ in a SIMD register r^{ℓ}_{simd} by duplicating ℓ times and $S\_B[j..(j+\ell-1)]$ in another SIMD register s_{simd}. We next evaluate the value pairs of SIMD registers r^{ℓ}_{simd} and s_{simd}. To find the pairs of the same values, we use a comparison SIMD instruction. If the value pair of the p-th lane with $0 \le p \le \ell$ are the same, the SIMD instruction sets 1 in the p-th bit in a register *mask*. Otherwise, the p-th bit of *mask* is set to 0 by the SIMD instruction.

By using the register *mask*, now we can generate the pairs of record ids in R_RID and S_RID as join results whose corresponding tuples have the same join attribute values. However, if we access R_RID and S_RID directly to extract the record id pairs, we have to use conditional branches. To minimize the usage of conditional branch, we need a SIMD instruction that shuffles the data values in a register using lane indexes from another register. Fortunately, such a shuffle instruction is provided by all SIMD instruction sets. Moreover, to use a shuffle instruction, we need the permutation mask whose i-th value x indicates that the $(x+1)$-th value of a SIMD register moves to i-th lane of another register. Similar to the work in [27], we build a table *shuffle_lookup* to get the permutation mask for each bit sequence of *mask*. The table *shuffle_lookup* consists of multiple entries each of which has four digits as a permutation mask. To find the entry of *shuffle_lookup* with respect to the result of the comparison SIMD instruction (i.e., *mask*) quickly, we use the value of the register *mask* as an index of *shuffle_lookup*. By regarding the bit sequence of *mask* as an integer value k, we get *shuffle_lookup*$[k]$ as a permutation mask. According to the obtained permutation mask, we can extract the record ids from r_rid_{simd} and s_rid_{simd} as join results without conditional branches and store them in R_Result and S_Result, respectively. Figure 3 illustrates the behavior of *PMJS*. Let us assume that ℓ be 4 as well as $R\_A[i]$ be 2 and $S\_B[j..(j+3)]$ be 1, 2, 2, 5. Then, the result of the comparison SIMD instruction SIMD_EQ is 0110. We next get *shuffle_lookup*$[6 = 0110]$ $(= [1, 2, -1, -1])$ as a permutation mask where -1 means do nothing. By utilizing the permutation mask, we store r_rid_i, r_rid_i in R_Result and s_rid_{j+1}, s_rid_{j+2} in S_Result which denote that (r_rid_i, s_rid_{j+1}) and (r_rid_i, s_rid_{j+2}) are the join results.

After evaluating $S\_B[j..(j+\ell-1)]$ with the SIMD register s_{simd}, we check whether the last lanes of s_{simd} and r^{ℓ}_{simd} are the same. If it is true (i.e., $S\_B[j+\ell-1] = R\_A[i]$), there may exist an entry in $S\_B[(j+\ell)..(j+\ell\cdot 2-1)]$ which is equal to $R\_A[i]$. Thus, if both values of last lanes of s_{simd} and r^{ℓ}_{simd} are the same (i.e., the last bit of *mask* is 1), we let m be 2 and load $S\_B[(j+\ell\cdot(m-1))..(j+\ell\cdot m-1)]$ to s_{simd} and $S\_RID[(j+\ell\cdot(m-1))..(j+\ell\cdot m-1)]$ to s_rid_{simd}. Then, we conduct the above steps to generate the join results with i-th tuple of R and $(j+\ell\cdot(m-1))$-th to $(j+\ell\cdot m-1)$-th tuples of S. We repeat this by increasing m by 1 until the last lanes of s_{simd} and r^{ℓ}_{simd} are the same. When the last entries of s_{simd} and r^{ℓ}_{simd} are different (i.e., the last bit of *mask* is 0), we check whether $R\_A[i]$ is less than $S\_B[j+\ell-1]$. If $R\_A[i] < S\_B[j+\ell-1]$, we increase i by 1. Otherwise, j is increased by ℓ. We repeat the above process until all entries of R_A or S_B are evaluated.

We present the pseudo-code of our proposed algorithm *PMJS* with intrinsic functions provided by Intel[†] in Fig. 4. The procedure *PMJS* takes R_A, S_B, R_RID, S_RID as inputs where R_A and S_B are the sorted lists of join attributes of the relations subject to a join operation as well as R_RID and S_RID keep the record ids of the relations. As the results of *PMJS*, we generate R_Result and S_Result which store the record ids of R_RID and S_RID, respectively. A pair of record ids with the same location in R_Result and S_Result represents a join result. Since we already present the overview of *PMJS*, we now explain the behavior of *PMJS* based on the Intel intrinsic functions presented in the pseudo-code as shown in Fig. 4 briefly. Note that, since we use AVX 2 instructions with 256-bit SIMD registers in the pseudo-code of *PMJS*, eight 32-bit values can be loaded in a 256-bit SIMD register (i.e., ℓ is 8).

By scanning R_A and S_B, we find every pair of the join attribute values in R_A and S_B as well as generate

[†]https://software.intel.com/sites/landingpage/IntrinsicsGuide/

```
Procedure PMJS(R_A, S_B, R_RID, S_RID)
begin
1.  i=0, j=0, pos = 0
2.  while(i < |R_A| and j < |S_B|){
3.    r^ℓ_simd = _mm256_set1_epi32(R_A[i])
4.    r_rid_simd = _mm256_set1_epi32(R_RID[i])
5.    jj = j
6.    do{
7.      s_simd = _mm256_load_si256(& S_B[jj])
8.      s_rid_simd = _mm256_load_si256(& S_RID[jj])
9.      res_v = _mm256_cmpeq_epi32(r^ℓ_simd, s_simd)
10.     res_f = _mm256_castsi256_ps(res_v)
11.     mask = _mm256_movemask_ps(res_f)
12.     if(mask == 0) break
13.     p_r = _mm256_shuffle_epi8(s_rid_simd, shuffle_lookup[mask])
14.     _mm256_storeu_si256(&S_Result + pos, p_r)
15.     _mm256_storeu_si256(&R_Result + pos, r_rid_simd)
16.     pos += _mm_popcnt_u32(mask)
17.     jj = ((R_A[i] == S_B[jj + 7]) ? jj+8 : |S_B|)
18.   } while(jj < |S_B|)
19.   b = S_B[j + 7]
20.   j += (R_A[i] > b) * 8
21.   i += (R_A[i] ≤ b)
22. }
23. output(R_Result, S_Result)
end
```

Fig. 4 The pseudo-code of *PMJS*

every pair of the record ids whose corresponding tuples in relation R and S, respectively, have the same join attribute value (lines 2–22). To load $R\_A[i]$ and $R\_RID[i]$ to r^{ℓ}_{simd} and r_rid_{simd} by duplicating 8 times since ℓ is 8, we use the SIMD intrinsic function *_mm256_set1_epi32* (lines 3–4). Since we have to find the entries of S_B from j having the same join attribute value with $R\_A[i]$, we first set jj to j (line 5). We next use the SIMD intrinsic function *_mm256_load_si256* to load $S\_B[jj..(jj + 7)]$ and $S\_RID[jj..(jj + 7)]$ into s_{simd} and s_rid_{simd}, respectively (lines 7–8). By using the SIMD intrinsic function *_mm256_cmpeq_epi32*, we can evaluate whether each value pair of the each lane of r^{ℓ}_{simd} and s_{simd} is the same and generate a SIMD register res_v in which the p-th 32-bit integer value is -1 (or 0) if the pair of p-th lane of r^{ℓ}_{simd} and s_{simd} are the same (are different) (line 9). To generate a sequence of 8 bits *mask*, we use a type conversion function *_mm256_castsi256_ps* and a bit extraction function *_mm256_movemask_ps* (lines 10–11). When the value of *mask* is 0, there is no join result in $S\_RID[jj..(jj+7)]$. In this case, we code a conditional branch (line 12) to skip the remaining steps in the do-while loop. Note that, although our algorithm works correctly without this conditional branch, we use the conditional branch for efficiency since we find that the benefit of using the conditional branch to avoid invoking useless instructions is greater than the loss resulting from pipeline stall by misprediction.

With the permutation mask in the *shuffle_lookup* table accessed with *mask*, we shuffle s_rid_{simd} to locate the join results in s_rid_{simd} to the front of a SIMD register p_r by invoking *_mm256_shuffle_epi8* (line 13). Then, we simply copy eight values in p_r and r_rid_{simd}, respectively, to the main memory whose address starting from $S\_Result + pos$ and $R\_Result + pos$ by executing *_mm256_storeu_si256* (lines 14–15) where *pos* keeps the position of S_Result and R_Result to be stored next. We next update *pos* with increasing by the number of 1s in *mask* obtained by *_mm_popcnt_u32* since the number of 1s in *mask* is the number of the join pairs of $R\_RID[i]$ and $S\_RID[jj..(jj + 7)]$ (line 16). We next increase jj by 8 if $R\_A[i]$ and $S\_B[jj+7]$ are the same since there may be the same values of $R\_A[i]$ in $S\_B[(jj + 8)..(jj + 15)]$ and we set jj to $|S\_B|$ otherwise (line 17). If jj is equal to $|S\_B|$, we can evaluate the next pairs (line 18). When $R\_A[i]$ is greater than $S\_B[j + 7]$, we increase j by 8, otherwise we increase i by 1 (lines 19–21).

4.2 Exploiting Thread Parallelism

In this section, we present how to apply *PMJS* to multi-thread environments in order to exploit thread level parallelism. We refer to *PMJS* running in multi-thread environments as *PMJS_M* whereas *PMJS* with a single thread as *PMJS_S*. One key issue of *PMJS_M* is to achieve good load-balancing since the execution time of *PMJS_M* mainly determined by the longest execution time of threads.

Without loss of generality, we assume that the domains of join attributes of both relations R and S are the same and there are t threads $T_1, T_2, \ldots, T_t$. To evenly distribute the workload to each thread, we first split the domain of a join attribute $[s, e]$ into k *disjoint* partitions $p_1 = [s_1, e_1]$, $p_2 = [s_2, e_2]$, $\ldots$, $p_k = [s_k, e_k]$ where, for each partition p_i, $e_i - s_i = (e - s)/k$ and, for every pair of partitions p_i and p_j, $e_i \leq s_j$ with $1 \leq i < j \leq k$. We next assign partitions to each thread. Suppose that a set of partitions $P_i = \{p_{i_1}, p_{i_2}, \ldots, p_{i_{|P_i|}}\}$ be assigned to the i-th thread T_i. Let the workload of a thread T_i be $J(T_i)$ and the number of tuples of both relations R and S whose join attribute value is in a partition p_j be $n(p_j)$. Then, we have $J(T_i) = \sum_{p_j \in P_i} n(p_j)$ since the merge phase of sort-merge join with a pair of relation R and S runs in $O(|R| + |S|)$ on the average.

Our goal is that the longest workload of threads is minimize. This is equivalent to the well known *multiprocessor scheduling* problem [13]. Since the *multiprocessor scheduling* problem is NP-hard, we devise a greedy algorithm *BF* (Best-Fit) whose time complexity is $O(\log t \cdot k)$. In the algorithm *BF*, the initial workload of each thread is zero and we assign each partition iteratively to the thread with the smallest workload. Due to its simplicity, we can efficiently distribute the workload to each thread. The other advantage of *BF* is that we do need to re-sort the join attribute values (and record ids) assigned to each thread when we evaluate partitions p_j with increasing j one by one.

To utilize the algorithm *BF* in *PMJS*, the number of tuples $n(p)$ associated with each partition p is required. To calculate $n(p)$, we utilize the *kernal density estimator* [26] since it effectively approximates an unknown data distribution. The kernel density estimator is a generalized form of sampling, whose initial step is to produce a uniform random sample. In kernel estimation, each point distributes its weight in the space around it. A kernel function $K(x)$ describes the form of this weight distribution, generally distributing most of the weight in the area near the point. Summing up all the kernel functions, we obtain a density func-

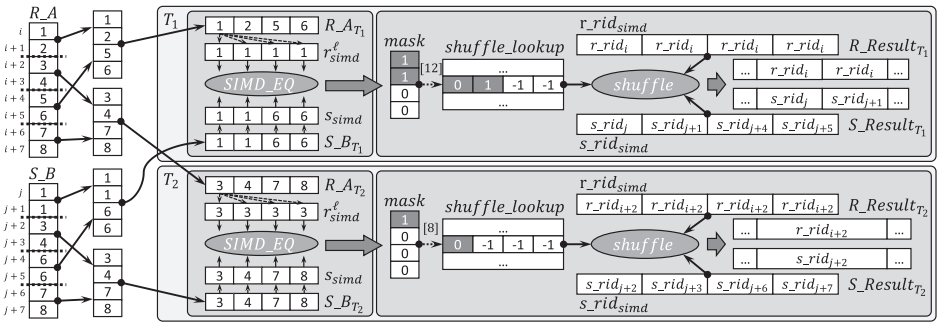


Fig. 5 The behavior of *PMJS_M*

tion for the dataset. Suppose that we samples a set of values S from a relation R's attribute. We can approximate the distribution of the attribute values using the distribution function $f(x)$ as follows:

$$f(x) = \frac{1}{|S|}\sum_{x_i \in S} K(x - x_i) \tag{1}$$

Then, the number of values within a range $[s_i, e_i]$ can be calculated as $\int_{[s_i,e_i]} f(x)$. The selection of the kernel function does not significantly affect the accuracy of the approximation [26]. Thus, we use the *Gaussian kernel* function since it does not need to normalize the input range into $[-1, 1]$ which causes additional cost:

$$K(x) = \frac{1}{B}\left(\frac{1}{2\pi}e^{-\frac{1}{2}(\frac{x}{B})^2}\right) \tag{2}$$

where B is the bandwidth of the kernel function $K(x)$.

According to scott's rule [26], B is set to as follows:

$$B = \left(\frac{4\hat{\sigma}^5}{3|S|}\right)^{\frac{1}{5}} \approx 1.06\hat{\sigma}|S|^{-\frac{1}{5}} \tag{3}$$

where $\hat{\sigma}$ is the standard derivation of the sample S.

We now estimate the number of tuples $n(p_j)$ associated with a partition p_j. Let $n\hat{}_{R_j}$ and $n\hat{}_{S_j}$ be the number of tuples of relations R and S whose join attribute values belong to the partition p_j, respectively, calculated by Eqs. (1), (2) and (3). Then, we get the estimate value $n(\hat{p}_j)$ of $n(p_j)$ as $n\hat{}_{R_j} + n\hat{}_{S_j}$. We thus calculate the workload $J(T_i)$ of each thread T_i using every $n(\hat{p}_j)$.

Figure 5 shows the behavior of *PMJS_M* when the number of threads t is 2 and the number of partitions k is 4. Suppose that the partitions $p_1 = [1, 2]$ and $p_3 = [5, 6]$ are assigned to the thread T_1 as well as $p_2 = [3, 4]$ and $p_4 = [7, 8]$ to T_2 by the workload balancing algorithm *BF*. Then, the join attribute values and record ids are allocated to proper threads with respect to the partition assignment to threads. Finally, the procedure *PMJS* presented in Sect. 4.1 is performed in each thread independently with its own join attribute values and record ids. Note that, since all partitions of join attribute domain, we do not need to compare the values allocated to a thread with those of the other threads. After each thread T_i generates its results on its own buffer called $R\_Result_{T_i}$ and $S\_Result_{S_i}$ as illustrated in Fig. 5, PMJS merges the results generated by all threads into a list.

Table 1 Implemented sort-merge join algorithms

Algorithms	Description
scalar_S	The traditional sort-merge join algorithm
$PMJS\_S_{128}$	*PMJS_S* with 128-bit SIMD registers
$PMJS\_S_{256}$	*PMJS_S* with 256-bit SIMD registers
scalar_M	The multi-thread version of the *scalar_S*
$PMJS\_M_{128}$	*PMJS_M* with 128-bit SIMD registers
$PMJS\_M_{256}$	*PMJS_M* with 256-bit SIMD registers
P-MPSM	The range Partitioned Massively Parallel Sort-Merge join [1]

Table 2 Parameters

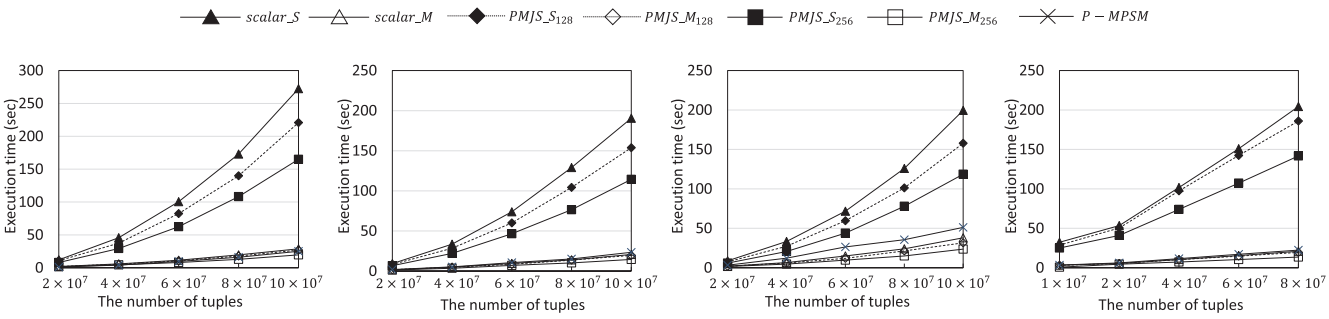
Parameter	Default	Range
No. of tuples (n)	8×10^8	$2 \times 10^8 \sim 8 \times 10^8$
No. of threads (t)	16	$1 \sim 128$
No. of partitions (k)	512	$16 \sim 4096$
Data distribution	normal	uniform, normal, zipf

5. Performance Study

5.1 Experimental Setup

All experiments were conducted on a machine with AMD RYZEN 7 1700 8-core 3.00GHz CPU supporting AVX 1 and AVX 2 instruction sets with 128-bit and 256-bit SIMD registers, respectively, and 16GB of DDR4 main memory running MS Windows 10. Note that, AMD RYZEN 7 1700 CPU can activate 16 thread contexts concurrently by SMT (Simultaneous MultiThreading). The implementations of all algorithms in Table 1 written in C/C++ were compiled by Intel icc 17.0 compiler with the highest optimization option -O3. In our experiments, we only measured the performance of the merge phase of each implemented algorithm including *P-MPSM* since our work only focuses on parallel processing of the merge phase.

Datasets: We empirically evaluated the performance of the implemented algorithms on the synthetic and real-life data sets. The synthetic data sets were generated following the uniform distribution, normal distribution and zipf distribution within the domain of data values $[0, 2^{31} - 1]$. In the zipf distribution, the skewness factor is 0.5. In our experiments, we set the sample size to 128 in order to build the kernel density estimator. The parameters used in our experiments are summarized in Table 2. To measure the perfor-



(a) uniform distribution (b) normal distribution (c) zipf distribution (d) Stack Overflow

Fig. 6 Varying n

mance of the algorithms in diverse environments, we varied the number of tuples n, the number of threads t and the number of partitions k.

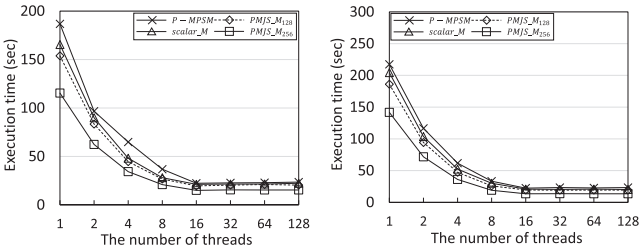
As a real-life data set, we obtained Stack Overflow from Google BigQuery[†] which contains contents of an online community for programmers to learn, share their knowledge, and advance their careers. The real-life data set Stack Overflow consists of 16 relations. As the relations subjected to the join operation, we used *post_history* and *posts_answers* each of which contains 93,449,204 and 22,046,899 tuples, respectively. As a join attribute, we selected *post_id* which is a common attribute of both relations. To show the scalability of our algorithm with the real-life data set, we randomly selected 10, 20, 40, 60 and 80 million tuples from relation *post_history* whereas we used whole tuples in relation *posts_answers*.

5.2 Experimental Results

The execution time of each algorithm is the average execution time measured by executing five times.

Varying the data size (n): In this experiment, we varied the number of tuples of both relations subjected to the join operation. The execution time of each algorithm on the data sets generated by uniform, normal and zipf distributions as well as real-life data set, respectively, is plotted in Fig. 6 (a), (b), (c) and (d), respectively.

As shown in Fig. 6, the performance of every algorithm gradually degrades as the number of tuples n increases. In general, as shown in Fig. 6 (a), (b), (c) and (d), the algorithms on multi-thread (i.e. *P-MPSM*, *scalar_M*, $PMJS\_M_{128}$ and $PMJS\_M_{256}$) are faster than the single thread algorithm *scalar_S* over all data sets. It indicates that thread parallelism affects the performance of each algorithm larger than the other parallelism. *P-MPSM* is 7.41 times faster than *scalar_S* due to its workload balancing feature and thread parallelism. However, due to the limitation of *P-MPSM* such that each thread has to retrieve every tuple of a relation participating in a join operation as mentioned in Sect. 2, *P-MPSM* is slower than *scalar_M* exploiting our workload balancing technique. In contrast to *P-MPSM*, by exploiting our workload balancing algorithm *BF*, both relations subjected to a join operation are split into disjoint partitions and each partition pair is allocated to each individual thread to generate join results. Thus, whole tuples of one of both relations are not required in each thread in *scalar_M* and *PMJS_M*. Therefore, *scalar_M* is at most 8.42 times faster than *scalar_S* due to its thread parallelism and 1.2 times faster than *P-MPSM* owing to our workload balancing algorithm. Among the algorithms on multi-thread, the performances of the algorithms with SIMD registers (i.e., $PMJS\_M_{128}$ and $PMJS\_M_{256}$) are better than that of *scalar_M* due to instruction level parallelism and data parallelism. Furthermore, $PMJS\_M_{256}$ shows the best performance over all data sets, as we expected, owing to wider SIMD registers. Meanwhile, regardless of varying n, the traditional sort-merge join algorithm *scalar_S* is the worst performer since it does not exploit any parallelism. When n is 2×10^7 with synthetic data sets, $PMJS\_M_{256}$ executes a join operation 8.1 times faster than *scalar_S* on the average. Furthermore, when n is 10×10^7, $PMJS\_M_{256}$ executes a join operation 12.62 times faster than *scalar_S* on the average. As shown in Fig. 6 (d), the experimental result with the real-life data set Stack Overflow confirms that $PMJS\_M_{256}$ is the best performer among implemented algorithms thanks to thread parallelism by multiple threads, instruction level parallelism utilizing minimal conditional branch and data parallelism exploiting wider SIMD registers.

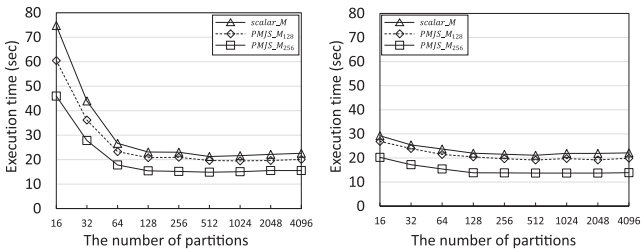


(a) normal distribution (b) Stack Overflow

Fig. 7 Varying t

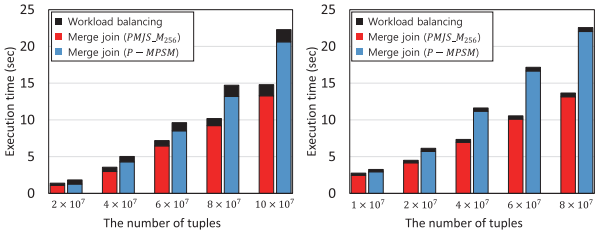
Varying the number of threads (t): To observe the effect of the number of threads (t), we varied t from 1 to 128 and plotted the performance of each algorithm in Fig. 7. As shown in Fig. 7 (a) and (b), our proposed algorithm with

[†]https://bigquery.cloud.google.com/dataset



(a) normal distribution (b) Stack Overflow

Fig. 8 Varying k



(a) normal distribution (b) Stack Overflow

Fig. 9 The performance breakdown of $PMJS\_M_{256}$

Table 3 Effect of workload balancing

Alg.	n	2×10^7	4×10^7	8×10^7
scalar_M	BF	1.504	4.89	13.95
	NONE	3.48	10.12	35.93
$PMJS\_M_{128}$	BF	1.75	4.68	13.65
	NONE	3.84	11.27	39.78
$PMJS\_M_{256}$	BF	1.37	3.51	10.22
	NONE	3.27	9.02	30.388

wider SIMD registers $PMJS\_M_{256}$ is the best performer regardless of changing t. Since each algorithm does not fully utilize the cores of CPU, each algorithm does not show the best performance when t is less than the number of concurrently activated thread contexts (i.e., 16). When t is greater than or equal to 16, the performance of each multi-thread algorithm becomes stable since the resources of the CPU cores are fully utilized and the overhead of executing multiple threads in a core is very low. Furthermore, since each thread has disjoint workloads, the performance of each multi-thread algorithm with more than 16 threads is not degraded and not improved as shown in Fig. 7.

Varying the number of partitions (k): To distribute the workload evenly to each thread, we first split the domain of a join attribute into k partitions by utilizing the workload balancing algorithm *BF* as presented in Sect. 4.2. In this experiment, we show the effect of k in our algorithms. Since *P-MPSM* splits the domain of a join attribute into t partitions where t is the number of threads, we do not report the performance of *P-MPSM* in this experiment. As shown in Fig. 8, when k is extremely small (i.e., $k = 16$), each algorithm takes the longest execution time since each thread is assigned to the equi-width partition and thus the workload cannot be distributed evenly. As k becomes larger, the performance of each algorithm becomes improved. When k is 512, each algorithm shows the best performance. Meanwhile, when k becomes greater than 512, the execution times of our algorithms are not reduced but very slightly increase since the benefit of small sized partitions for workload balancing becomes marginal but the overhead of allocating large number of partitions to threads a little bit increases.

Effects of workload balancing: To show the effect of our workload balancing technique, we reported the execution time of each algorithm on multi-thread utilizing the workload balancing algorithm BF and that without workload balancing (NONE) in Table 3. When n is very small, the gap of execution times between BF and NONE is very small since all algorithms are terminated within a few seconds. However, as n increases, the algorithms with BF are superior to those without BF.

To show the overheads of our workload balancing algorithm BF exploiting the kernel density estimator and that of *P-MPSM*, we present the execution times of workload balancing and merge join in Fig. 9 (a) and (b). As shown in Fig. 9 (a) and (b), as n increases, both overheads of workload balancing and merge join increase. However, while the overhead of merge join rapidly increases with increasing n, that of workload balancing gradually increases. As shown in Fig. 9 (a) and (b), since the workload balancing technique of *P-MPSM* scans a table in parallel to build a histogram, the execution time of the workload balancing of *P-MPSM* is longer than BF which utilizes a random sample to calculate a kernel density estimator. Furthermore, as reported in Table 3, the algorithms utilizing BF based on kernel density estimation show the better performance than those without BF. It is indicated that our proposed workload balancing algorithm is effective. Consequently, since *PMJS_M* distributes the workload evenly to each thread utilizing workload balancing algorithm and exploits all parallelism, it is faster than the other algorithms.

6. Conclusion

In this paper, we presented an efficient merge join algorithm *PMJS* with SIMD instructions. The existing scalar sort-merge join algorithm includes conditional branches that degrade the performance in the merge phase. To solve this problem, we carefully design our algorithm minimizing the use of conditional branches. In addition, in order to fully utilize the multi-core architecture of modern microprocessors, we extend *PMJS* to multi-thread environments. Moreover, we develop an effective workload balancing algorithm in which we adopt the *kernel density estimator* to estimate accurately the workload assigning to each tread. In our experiments, we shows the superiority of our algorithm *PMJS* over diverse environments.

References

[1] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," PVLDB, vol.5, no.10, pp.1064–1075, 2012.
[2] C. Balkesen, G. Alonso, J. Teubner, and M.T. Özsu, "Multi-core,

main-memory joins: Sort vs. hash revisited," PVLDB, vol.7, no.1, pp.85–96, 2013.
[3] C. Balkesen, J. Teubner, G. Alonso, and M.T. Özsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," In IEEE ICDE, pp.362–373, 2013.
[4] M.W. Blasgen and K.P. Eswaran, "Storage and access in relational data bases," IBM Syst. J., vol.16, no.4, pp.363–377, 1977.
[5] P.A. Boncz, S. Manegold, M.L. Kersten, et al., "Database architecture optimized for the new bottleneck: Memory access," VLDB, pp.54–65, 1999.
[6] K. Bratbergsengen, "Hashing methods and relational algebra operations," VLDB, pp.323–333, 1984.
[7] R. Cypher and J.L. Sanz, The SIMD model of parallel computation, Springer Science & Business Media, 2012.
[8] D.J. DeWitt and R. Gerber, "Multiprocessor hash-based join algorithms," VLDB, pp.151–164, 1985.
[9] D.J. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "A High Performance Dataflow Database Machine," Computer Science Department, University of Wisconsin, 1986.
[10] B. Gedik, R.R. Bordawekar, and P.S. Yu, "Cellsort: high performance sorting on the cell processor," VLDB, pp.1286–1297, 2007.
[11] B. Gedik, P.S. Yu, and R.R. Bordawekar, "Executing stream joins on the cell processor," VLDB, pp.363–374, 2007.
[12] G. Graefe, A. Linville, and L.D. Shapiro, "Sort vs. hash revisited," IEEE Trans. Knowl. Data Eng., vol.6, no.6, pp.934–944, 1994.
[13] R.L. Graham, "Bounds on multiprocessing timing anomalies," SIAM journal on Applied Mathematics, vol.17, no.2, pp.416–429, 1969.
[14] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-sort: A new parallel sorting algorithm for multi-core SIMD processors," In IEEE International Conference on Parallel Architecture and Compilation Techniques, pp.189–198, 2007.
[15] C. Kim, T. Kaldewey, V.W. Lee, E. Sedlar, A.D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: fast join implementation on modern multi-core cpus," PVLDB, vol.2, no.2, pp.1378–1389, 2009.
[16] P.M. Kogge, The architecture of pipelined computers, CRC Press, 1981.
[17] D. Lemire, L. Boytsov, and N. Kurz, Simd compression and the intersection of sorted integers, arXiv, 2014.
[18] H. Lu, K.-L. Tan, and M.-C. Shan, "Hash-based join algorithms for multiprocessor computers," VLDB, pp.198–209, 1990.
[19] R. Martin, A vectorized hash-join, unpublished course report, University of California at Berkeley, May, 1996.
[20] T.H. Merrett, "Why sort-merge gives the best implementation of the natural join," ACM SIGMOD Record, vol.13, no.2, pp.39–51, 1983.
[21] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, Sort vs. hash join revisited for near-memory execution, ASBD, 2015.
[22] O. Polychroniou, A. Raghavan, and K.A. Ross, "Rethinking simd vectorization for in-memory databases," In ACM SIGMOD, pp.1493–1508, 2015.
[23] N. Satish, C. Kim, J. Chhugani, A.D. Nguyen, V.W. Lee, D. Kim, and P. Dubey, "Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort," In ACM SIGMOD, pp.351–362, 2010.
[24] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using simd instructions," ADMS@ VLDB, pp.1–8, 2011.
[25] D.A. Schneider and D.J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," In ACM SIGMOD, vol.18, no.2, pp.110–121, 1989.
[26] D.W. Scott, Multivariate density estimation: theory, practice, and visualization, John Wiley & Sons, 2015.
[27] A.A. Stepanov, A.R. Gangolli, D.E. Rose, R.J. Ernst, and P.S. Oberoi, "Simd-based decoding of posting lists," In CIKM, pp.317–326, ACM, 2011.
[28] J. Zhou and K.A. Ross, "Implementing database operations using simd instructions," In ACM SIGMOD, pp.145–156, 2002.

Gilseok Hong was born in 1992. He is currently pursuing the M.S. degree in computer science and engineering from Korea University of Technology and Education, Republic of korea. He received the bachelor's degree from the School of Computer Science and Engineering, Korea University of Technology and Education, in 2016. His main research interests include database, parallel processing and data mining.

Seonghyeon Kang was born in 1992. He is currently pursuing the M.S. degree in computer science and engineering from Korea University of Technology and Education, Republic of korea. He received the bachelor's degree from the School of Computer Science and Engineering, Korea University of Technology and Education, in 2016. His main research interests include database, parallel processing and data mining.

Chang soo Kim received his M.S. degree in computer science from Sogang University, Seoul, Republic of Korea, in 1995 and his Ph.D. degree in information and communication engineering from Chungbuk National University, Cheongju, Republic of Korea, in 2006. He is a principal researcher working at the Software Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea. His research interests include big data management and processing systems, database systems, cloud computing and storage systems.

Jun-Ki Min was born in 1972. He is a Professor with the School of Computer Science and Engineering, Korea University of Technology and Education, Republic of Korea. He received the Ph.D. degree from KAIST, Republic of Korea, in 2002. Then, he continued his research work as a Post-Doctoral Researcher with the School of Computing, KAIST, from 2003 to 2004. In 2004, he worked as a Senior Researcher in ETRI, Republic of Korea. His main research interests include XML, spatial-temporal DB, stream data, sensor data, Big data, and MapReduce.