Proc. The First International Workshop on Scalable Social Event Processing and Management (SSEPM 2015), In conjunction with the ICDE 2015 Conf., April 13, 2015, SEOUL, Korea.

# SINCA: Scalable in-memory event aggregation using clustered operators

Mahesh Kumar Behera [*1], Kalyan S [*2], Prasanna Venkatesh [*3], Antoni Wolski [#4]

[*] *Huawei Technologies India Private Limited*
*Divyashree Techno Park, Bangalore, India*
[1] mahesh.behera@huawei.com
[2] kalyan.s@huawei.com
[3] prasanna.venkatesh@huawei.com
[#] *AWO Consulting, Helsinki, Finland*
[4] awoc@wolski.fi

*Abstract* — **Analytical processing of various information created in the operation of social media requires queries involving grouping and aggregating of large volumes of detail data. Any advanced query processing method should take into account two dominating hardware trends: increasing main memory capacities and increasing parallel processing capacity exposed as growing number of cores per processor chip. We introduce a scalable in-memory method for data aggregation (SINCA), using clustered operators, which profits from the hardware trends. The method uses a concept of a microengine being a set of resources that can be utilized in parallel, with great efficiency. The resulting parallelized aggregation algorithm is characterized by a low overhead and high volume, and is suitable to both real-time and extract-transform-load scenarios. The core idea of the method is to use real-time histograms to partition the data for grouping. As the data is already grouped during the partitioning phase, the group aggregation can be done very efficiently. Additionally, some of the grouped data can be cached for re-use in subsequent queries.**

## I. Introduction

With the pervasive usage of social networking sites, the events getting generated come in larger and larger sets. A new industry termed "big [social] data analytics" has emerged around the existence of such large data. These large, unstructured, heterogeneous data sets have to be churned to derive "values" contained in it. The derived value helps people to gain better insights, which can be used to define and derive new business models. There are two important aspects that must be taken care of while deriving values from such events. Firstly it is hard to ignore the fact that the pace at which these events get generated make the older data outdated and stale very soon. Thus "freshness" is an important attribute of the extracted information in today's digital era. Secondly because of the abundance of data generated from various streams in social media, summarization is needed. The data has to be grouped and aggregated to bring out a bigger picture. For example, role analysis (which analyses the percentages of different roles that are assumed by community users over time[1]), behavioral analysis[2], market segmentation, viral event prediction and CDN (Content Delivery Network) data prefetch [3], etc. depend directly or indirectly on grouped data analysis.

The industry has responded to this massive amount of data by creating methods of pipeline-based processing. Multiple mechanisms ranging from stream processing to massive history data processing have emerged. Variations like real-time processing, massive batch processing is also prevalent.

There have been significant, fast paced advancements in the hardware front over the past few years. A major trend, though not new, is to increase the processing capability of the system by adding more and more cores into a single system. NUMA (Non-Uniform Memory Access) based system is an evolution of such a trend. Apart from increasing the processing power, large RAM capacities and faster, larger caches, SIMD (Single-Instruction, Multiple-Data stream) execution, etc. have been forcing analytical database designers to relook the designs. These trends are fostering in-memory solutions to be cheaper and more viable options than disk-based solutions. This would effectively remove the bottlenecks associated with I/O and make the data available for operation in memory.

However, to tackle large volumes of data in real time, along with the hardware capabilities, it is required to have improved algorithms to harness them. We focus on meeting the challenges of large hierarchical memories and NUMA architecture in the context of grouped aggregation. Grouped aggregation is in the center of analytical data processing by producing highly summarized results. In SQL, grouped aggregation is exposed by the use of aggregate functions like AVG() and the GROUP BY clause.

In this paper we propose a two-phased, intra-query parallelized group aggregation algorithm. We call this new radix-cluster-based algorithm *Scalable In-memory NUMA aware Clustered Aggregation* algorithm (SINCA algorithm). We introduce an intra-query parallelization engine, referred to as *microengine* (ME). Microengine is a thread operating on local memory. By way of pre-defined core affinity, the thread can execute a parallel part of the algorithm on a data partition residing in local memory. In SINCA, the data to be grouped is partitioned (clustered) to minimize the synchronization overhead. The algorithm can make use of vectorized code execution too and thus is suitable for exploiting both the SIMD instructions and thread level parallelism available in today's multi-core processors. In the first phase, the microengine based intra-query parallelization is used to reduce the impact of NUMA and multi core synchronization overhead. In the second part, a histogram-based radix cluster is used to group the input data into smaller subsets to maximize the use of pipelining, SIMD and local memory caches.

We implemented and evaluated the SINCA algorithm on an in-memory column store database, and we measured the time taken for group aggregation query. By using our method, the time taken to execute a query and fetch all the records is decreased at least by 1.4 times compared to normal parallel group-by algorithms. The improvement increases with the increase in number of groups and it reaches 5 times with 1.44 million groups. By using our new method, the time taken to execute the query and fetch the first set of records (query response time) is decreased at least by 4 times, compared to normal parallel group-by algorithms. The improvements increases with increase in number of groups and it reaches 32 times with 1.44 million groups.

This paper is arranged as follows. In Section II, the related work is presented. The detail implementation and motivation for the clustered group aggregation algorithm is described in Section III. In Section IV, experimental results are reported. Section V concludes the paper.

## II. Related Work

Grouped aggregates are resource-consuming operations. Their best case complexity lies in O(n. log n). Hence it is beneficial to parallelize their execution. Grouped aggregates are by nature *materialized*. That is to say that all the input data for the grouping have to be materialized and only then the aggregate results can be produced. This materialized nature of the grouped aggregates makes parallelization of grouped aggregates difficult. The following diagram (Fig. 1) shows the prior art. In this implementation the last step *"FINAL AVERAGE"* is an obstacle:

1. It makes the execution wait until the final step is over, thereby introducing the latency
2. The final step is generally executed on a single thread, thereby making this step the bottleneck.

Substantial work has been done to improve the performance of group aggregation. The key trend has been towards parallelizing this building block, as in intra operator parallelism. Be it commercial disk database, e.g. Microsoft SQL server [9] or a patented idea [10], there has been substantial development and interest in parallelizing the grouping operation by splitting the data to number of threads.

However there is no known work done to tune the group aggregation query for in-memory execution and to make use of current hardware improvements.

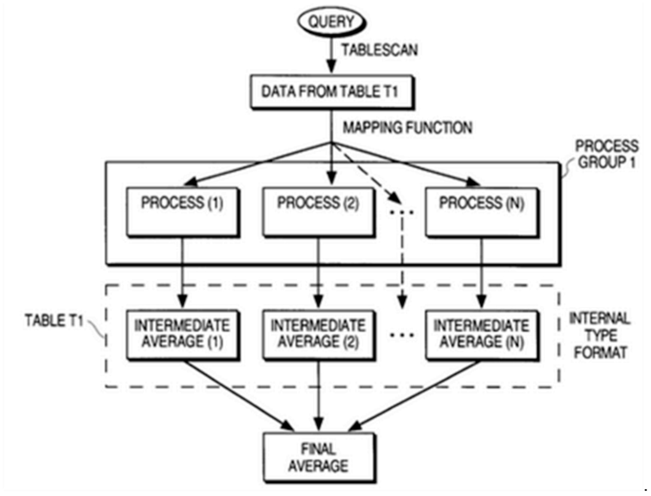

Fig. 1. Query execution in prior art.

Processors are equipped with SIMD hardware allowing to performing so-called vectorized processing that is, executing the same operation on a series of closely adjacent data. Current parallel grouped aggregate methods use SIMD in a limited way to compute the aggregates. This paper suggests using SIMD more widely in the query processing.

Columnar stores store each column separately. A notional row is bound by row-ids shared by the column constructs. Traditional repartition techniques involve copying of the columns on which grouping is done and also the columns which need to be aggregated. Such a model does not leverage the storage layout of columnar databases.

As hardware trends involve putting more and more processors and cores into a single chip, the processing capacity of the system has been growing. As suggested in [6], the trend is shifting towards multi-core utilization rather than on speeding up individual CPUs. In [6] and [7], authors emphasize data placements and data movement to gain scalability and high performance on NUMA based systems. The tests reported in [7] ascertain the claims.

There are also improvements in the capabilities of other hardware components, like larger caches, increased RAM memory and larger TLB, and huge page support. As per the experiments and study put forth in [5] and the case studied further in [4], it is necessary to tune the software to tame the hardware power. In [4] and [5], an approach is taken to perform grouping by splitting and repartitioning the data using radix clustering, so that the data fits within the TLB better, and avoiding cache misses while using the data. The crux of the performance lies in choosing suitable radix bit for the radix clustering [8].

In this work, we built upon existing algorithms [9] [10] and increased the emphasis on better accommodation of NUMA, SIMD, TLB, and cache consciousness.

## III. SINCA Algorithm

We have designed a multi-threaded NUMA aware method utilizing parallel processing units called microengines (MEs). A microengine executes one of many parallel threads on a data partitioin that is local to the core the microengine is running on.

The algorithm starts with creating a histogram on the data to be grouped and aggregated. This histogram's output lets the query planner to decide the applicability of the SINCA algorithm. SINCA is suitable for calculating grouped aggregate on groups where cardinality of each group is roughly the same. Next we use a radix clustering method to cluster the data based on a particular column's value. The number of radix bits (b) to be used for the clustering is chosen in such a way that it can exploit the cache and TLB as suggested in [5]. Also, as part of clustering, the data (or records) from various NUMA memory banks are moved to a common memory bank based on the radix cluster number. This step is critical to exploit the advantages of NUMA memory access and achieve maximum scalability. Since the operation in one group will not interfere with the one in another group, contention is completely eliminated. Next, these clusters are fed to ME threads, which are bound to the core on which the data is local and execute the aggregation algorithm. This step is completely contention-free as the data on which each thread works is already clustered and the aggregation calculation on one cluster is totally independent of the data in other group. Also the cache and TLB usages will be superior because of the size of the cluster chosen.

In Fig. 2, the flow of data in the SINCA algorithm is shown. The input data can be table data or can be data streamed from real time event sources. In case of table data, the clustering phase creates a separate copy of NUMA-aware clustered data required for the group aggregation. In case of streamed data, the data can be partitioned to different clustered during insertion. The figure depicts the data storage in a 2 socket NUMA machine having 4 processors. As the chosen cluster size is based on the system information, the data for hash table, aggregated value and the final result can be accommodated inside processor cache to improve the query execution speed. The final output can be streamed to the next phase when the results from each processor are ready and the processors can start working on the next cluster.

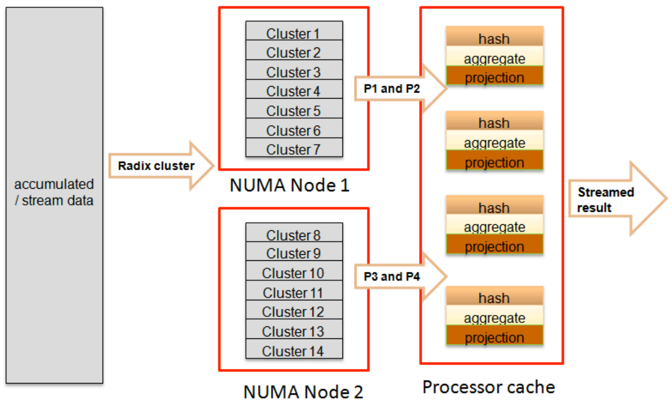

Fig. 2. SINCA query execution

In Fig. 3, the notional query plan generated during the execution of the SINCA algorithm is shown. The SINCA distributor node parallelizes the execution of the aggregation. The Scan nodes in the bottom of the query tree operate on the clustered data from each memory bank and feed the results to the parent nodes for processing.

In the rest of the section we will explain, in detail, the algorithm.

## A. *Microengine based Intra Query Parallelisation*

Most database systems are constructed in such a way that each query is executed in a separate thread. This is called a *thread-per-query model*. In an analytical system, the queries can be rare but very complex. The query response time becomes an important performance indicator. If the thread-per-query model is used in a modern system having a large number of processor cores, the cores can be left unutilized and the response time can remain poor. To cope with this problem, we introduce operation-level parallelism: each query is split to a number of subthreads (called microthreads) that are executed concurrently in different cores.

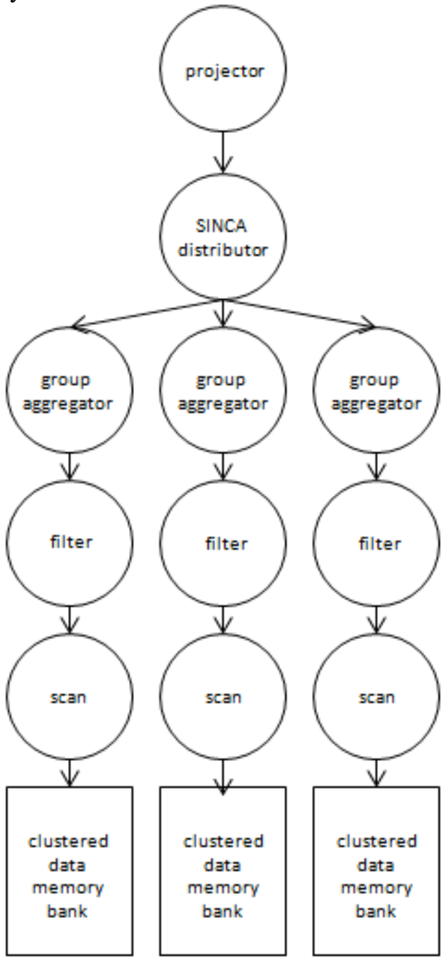

Fig. 3. SINCA query execution

This microthread model is illustrated in Fig. 4, where it is contrasted with the thread-per-query model.

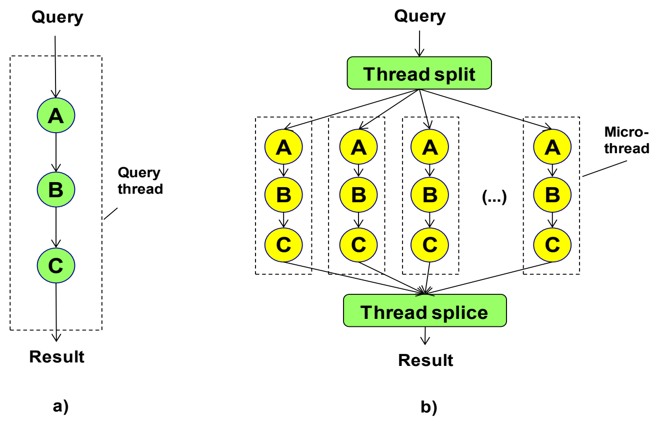

Fig. 4. Query execution in a) the traditional thread-per-query model and in b) the microthread model.

It is not sufficient to just make the operation parallel. With increasing size of physical memory, the time to access the memory has increased drastically. With new systems having non-uniform memory partition, it is now more and more important to keep the memory access local.

The problems with caches are that they do not end with cache misses. If the same memory location is copied to more than one cache, writing to that location is subject to cache coherency protocol. That protocol, executed by the on-chip hardware, has the purpose of ensuring that no concurrent conflicting writes (to two different copies of a memory location) can happen. The protocol, typically, invalidates the old copies of the same memory location, so that other threads cannot use them. It is only when all invalidation messages are acknowledged, that the data item can be written into the cache. The cache protocol has to be executed across all the sockets in the system, and thus, it has to use the socket interconnect mechanism (In Intel, it is called QPI – QuickPath Interconnect) (Fig. 5).

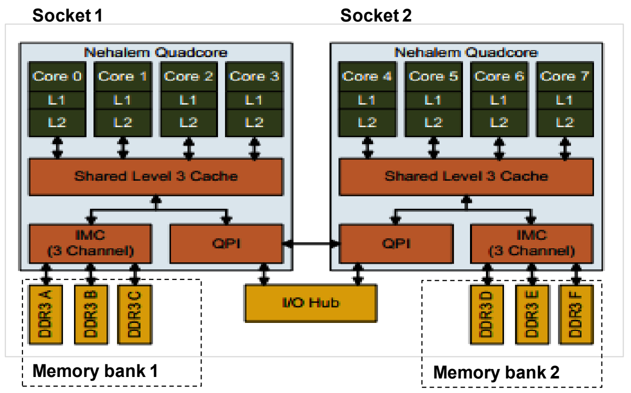

Fig. 5. Illustration of the memory hierarchy and inter-socket communication components in a processor (Intel Nehalem).

All of this can increase the duration of a write operation by one or two orders of magnitude, depending of the system size and load situation. The protocol affects also the memory read operations: the next access to the invalidated data will cause a cache miss. The best way to avoid the intervention of the cache coherency protocol is to make the cached data *private* — when there are no other cached copies at all. This is what all cache-sensitive algorithms try to do.

The problems with the hierarchical storage do not even end here. As can be seen in Fig. 5, different parts of main memory (memory banks) are connected to different sockets. The time to access the socket's *local* memory bank is much shorter (about a half) compared to accessing a *remote* memory bank. The solution is called NUMA (Non-Uniform Memory Architecture) and was introduced to avoid a bottleneck caused earlier by a single memory bus. With NUMA, on-chip memory controllers allow for fast access to the local memory bank. The solution works well if the threads running on a chip predominantly use local memory. However, if the remote memory is accessed, the *NUMA penalty* has to be paid. It is not only in the form of longer access times. Because the data requested from remote memory banks flows through the same socket interconnect mechanism that is used by the cache coherency protocol, that mechanism can become the single most damaging bottleneck in the system.

The microengine-based system comes handy in these scenarios. As the data is partitioned at run time, only adjacent data are feed to each microengine. This ensures that each microengine accesses only local data and do not touch data accessed by other microengines. The number of microengines can be configured to be dependent on the amount of data and load on the system. With no shared access to data, the microengine based system reduces the system cache contention drastically. This makes the query execution faster.

## B. *Radix Clustered Group Aggregation*

The major problem in parallel algorithm is synchronisation overhead. The first kind of synchronisation overhead is related to data dependency. The problem with data dependency can be avoided using data partitioning, as we have discussed in the previous section. But there is a bigger problem with respect to data dependency when the final output depends on the whole of the data, like in grouped aggregation. Because of the interlining of the result with the whole of data, most of the parallel grouped aggregation algorithms operate in a two-phase manner as discussed above. These two-phase algorithms make use of the multiple cores in the first phase but merge back to a single-thread operation in the second phase and thus make the system underutilised. In SINCA algorithm we have removed the second phase of normal group-by algorithm by partitioning the input data based on group-by column. The detailed algorithm has three steps. The first step is distributing the input data based on radix bits. The next step is preparing the data for each microengine. This step makes a copy of the input data which will be used for aggregation and grouping. The final step is the actual grouped aggregation operation by each microengine and projection of the result.

We are applying radix clustering methods to cluster the input data using a fixed number of bits, called the radix bits, to create clusters that are groups of adjacent values having the same radix bits. If the number of radix bits is *n*, then the number of clusters generated is $2^n$. In the example, two least significant bits are used, resulting in four clusters (Fig. 6).

Each cluster carries the GROUP BY columns values and the corresponding row_IDs. The radix bits are chosen using the single scan analysis of the input data to make the clusters small enough to fit into the system cache. Radix clustering makes clusters independent for group aggregation calculation. With the clustered data fitting into cache size, all the group aggregation processing can be done inside cache and thus improve the query execution speed.

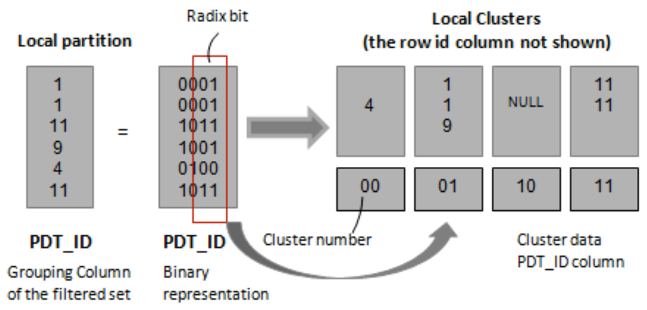
Local partition
Radix bit
Local Clusters (the row id column not shown)
PDT_ID
Grouping Column of the filtered set
PDT_ID
Binary representation
Cluster number
Cluster data PDT_ID column

Fig. 6. Grouping of records based on radix cluster.

Each thread continues on a cluster by expanding it with aggregate columns. Since each thread operates on data existing in its NUMA node, the projection operation is NUMA-friendly. The following diagram (Fig. 7) shows the projection performed by three threads on three local clusters.

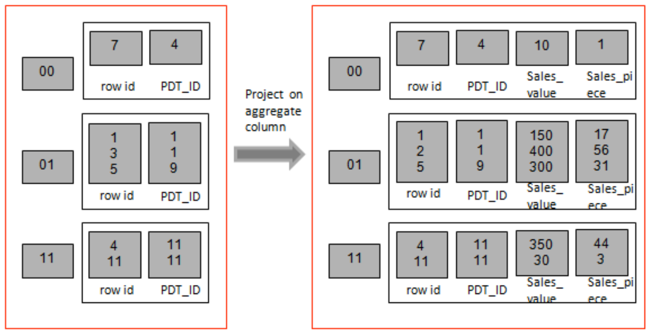

Fig. 7. NUMA friendly storage of cluster data.

A single total cluster (a combination of local clusters) is spread across discontinuous memory. This is because each thread individually performed the clustering. A thread which will perform the grouping on a single total cluster will need to gather all these cluster data. The gathering is done by the thread obtaining the starting address and the number of items in that address (Fig. 8). The table data is not copied. Even if the piece of a total cluster is present in the remote node, the access is going to be sequential and, thus, very efficient.

The grouping can be done by any method (Hash, Sorted or nested loop). The group aggregation can be done in parallel, independently for each total cluster.

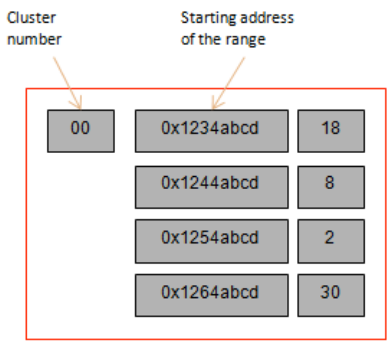

Fig. 8. Projected data from cluster with radix bits 00.

The aggregation is done by looking up the aggregation column values present in the clustered data. The grouped aggregate for one cluster (of cluster no. 01) is shown here (Fig 9).

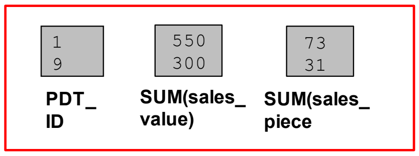

Fig. 9. Group aggregate

## IV. Experimental Results

The experiment has been performed on an in-memory column store database. The database table has 5 columns of integer type. The dataset includes 5 million rows. The experiment is done to measure the total execution time and the response time of the first result row. We have done the experiments on three different group sizes. The group size depends on the number of distinct values in the input data.

TABLE I

| Number of groups | Execution Time (in milli seconds) | | |
|---|---|---|---|
| | Prior Art | SINCA | Improvement |
| 50001 (1%) | 410.902 | 293.141 | 1.4 |
| 499979 (10%) | 2467.583 | 511.594 | 4.8 |
| 1446523 (30%) | 4386.6 | 849.652 | 5.1 |

In Table 1, the execution time for prior art parallel group aggregation is compared against the result of SINCA. Here, the time is measured from start of query execution to the time it finishes fetching the last record. In the first row, the number of resultant rows is 1% of the total number of records in the table. In the second row the result includes 10% of the number of records in the table, and in row 3 the number is 30%. We see in the Improvement column that, by using the SINCA algorithm, we can get better results when there are more of resultant rows. The improvement is over 5 times when the number of result records is 30% of the input records.

TABLE 2

| **Number of groups** | **Response Time (in milli seconds)** | | |
|---|---|---|---|
| | **Prior Art** | **SINCA** | **Improvement** |
| 50001 (1%) | 410.902 | 49.274 | 4.1 |
| 499979 (10%) | 2467.583 | 116.853 | 21.1 |
| 1446523 (30%) | 4386.6 | 134.202 | 32.6 |

In Table 2, the response time for normal parallel aggregation algorithm is compared against the corresponding SINCA result. The response time is the time taken by the query execution from the start of the query execution to the output of the first record. As in SINCA a single-phase query execution is used, as compared to two phases in the prior art algorithm, the response time improvement is more than 32 times than the normal parallel algorithm.

## V. CONCLUSIONS

We have demonstrated that new hardware capabilities of contemporary processor chips can be put to good use while performing group-by aggregated queries on massive data. We presented a method of parallelized query execution, that benefits from large memory size, can utilize multiple processor cores and local memory banks efficiently, and yields well to vectorized processing with SIMD hardware. We reported on performance experiments showing that the response time can be improved by as much as 32 times, compared to traditional methods, if there are a large number of groups. The throughput is increased significantly also.

## REFERENCES

[1] J. Chan, C. Hayes, and E. M. Daly, "Decomposing discussion forums and boards using user roles". in *Proc. ICWSM, 2010*.

[2] Marcel Karnstedt, Scalable Social Analytics for Online Communities, [online] http://stcsn.ieee.net/e-letter/vol-1-no-2/scalable-social-analytics-for-online-communities.

[3] Puneet Jain, Justin Manweiler, Arup Acharya, and Romit Roy Choudhury. "Scalable Social Analytics for Live Viral Event Prediction", Association for the Advancement of Artificial Intelligence (2014).

[4] Cagri Balkesen, and Jens Teubner, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware", Technical Report Nr. 779, Systems Group, Department of Computer Science, ETH Zurich, November 30, 2012.

[5] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten, "Optimizing database architecture for the new bottleneck: memory access", in The *International Journal on Very Large Data Bases* 9:3 (2000): 231-246.

[6] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann, "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems", Proc. VLDB Endow. 5:10 (June 2012):1064-1075, 2012.

[7] T. Kiefer, B. Schlegel, and W. Lehner, "Experimental evaluation of NUMA effects on database management systems", in *BTW*, 2013

[8] D. E. Knuth, *The Art of Computer Programming*, Vol.III: Sorting and Searching. Addison-Wesley, page 80, Ex. 13, 1973.

[9] Craig Freedman, "Introduction to Parallel Query Execution", [online] http://blogs.msdn.com/b/craigfr/archive/2006/10/11/introduction-to-parallel-query-execution.aspx

[10] William H Waddington and Jeffrey I Cohen, “Method and apparatus for parallel processing aggregates using intermediate aggregate values,” U.S. Patent 5,850,547, December 15, 1998.

[11] D.E Ott, “Optimizing Software Applications for NUMA”, Whitepaper (Intel), 2009