Skew Handling in Aggregate Streaming Queries on GPUs*

Georgios Koutsoumpakis
Uppsala University
Sweden

Iakovos Koutsoumpakis
Uppsala University
Sweden

Anastasios Gounaris
Dept. of Informatics
Aristotle University
Thessaloniki, Hellas

ABSTRACT

Nowadays, the data to be processed by database systems has grown so large that any conventional, centralized technique is inadequate. At the same time, general purpose computation on GPU (GPGPU) recently has successfully drawn attention from the data management community due to its ability to achieve significant speed-ups at a small cost. Efficient skew handling is a well-known problem in parallel queries, independently of the execution environment. In this work, we investigate solutions to the problem of load imbalances in parallel aggregate queries on GPUs that are caused by skewed data. We present a generic load-balancing framework along with several instantiations, which we experimentally evaluate. To the best of our knowledge, this is the first attempt to present runtime load-balancing techniques for database operations on GPUs.

1. INTRODUCTION

As the amount of data stored and processed in databases increases dramatically, a lot of research effort is being put in the development of advanced techniques that would allow database management systems (DBMSs) to deal with extremely large data volumes more efficiently. Due to the sheer amount of data mentioned above, when conventional, centralized techniques are employed, the cost of processing is raising, thus leading to unacceptable processing times. In general, a DBMS query response time is affected by two factors, namely the I/O cost, which is the time spent in loading the data from the secondary storage into the main memory, and the computational cost, the time spent by the DBMS while processing the data. Traditionally, research on databases has mostly focused on reducing the I/O cost during query processing, since this is the major bottleneck in many operations. However, the benefits of increasing the system's throughput cannot be ignored as well. So, in an effort to get the most of the database systems, processing throughput should be maximized, and the most broadly established approach to this end is through parallelism. One of the most effective forms of parallelism in queries is partitioned parallelism; partitioned (or intra-operator) parallelism refers to the case where multiple query operators simultaneously process distinct partitions of the same dataset [7].

In this paper, we deal with partitioned queries but we diverge from the conventional DBMS, and we focus on Continuous Query (CQ) systems [15, 23]. In contrast to traditional DBMSs that answer streams of queries over a non-streaming database, CQ systems treat queries as fixed entities and stream the data over them. This means that the data are not known during the initialization of the query, but new data become available during the execution, so the data items are presented as a possibly infinite sequence of records. Meanwhile, by utilizing a fixed-size sliding window as it typically happens, we ensure that only the most recent data elements are considered when answering queries. In this way, the query results depend only on the latest and most up to date data. CQs are particularly applicable to streaming scenarios, where data is produced at such a fast rate that it is not practical first to store the data and then to process it; rather, data processing must be performed on the fly. More specifically, we investigate parallel aggregate queries over data streams, i.e., queries that split the dataset into groups, and for each group continuously update the value of an aggregate function leveraging the group-by database operator [9]. E.g., in a stream of stock market data, we can pose a query to continuously output the average price in the last one thousand transactions for each stock. Such queries are essential in challenging, real-time data-mining applications [11, 4].

An efficient way of increasing the computational capacity can be achieved by taking advantage of the parallelization capabilities and high computational power offered by modern graphics processing units (GPUs). This is commonly referred to as general purpose computing on GPUs (GPGPU). Since GPUs are especially designed for stream processing and provide free programmable processing cores, they are well suited to be used for several database operations, for example group-by query execution. Many-core technologies like NVidia's Compute Unified Device Architecture (CUDA) and the associated Fermi hardware architecture [17, 16] that have been built on top of GPUs simplify the development of highly parallel algorithms running on a single GPU. In general, a GPU can only execute algorithms for processing the

*Research conducted while G. Koutsoumpakis and I. Koutsoumpakis were with the Aristotle University of Thessaloniki.

data, called kernels, while the corresponding control logic is executed on the CPU. Previous research in the field has led to the implementation of database management systems that allocate data intensive tasks to the GPU. Examples of such systems are Sphyraena [1] and GPUQP [12], which are implementations of DBMSs for GPUs.

At the initialization of a CQ query, the query engine selects a default configuration, which provides the settings about how the query will execute among the graphics card's threads. Queries on a data stream will, by definition, run long enough to experience changes in data properties as well as system workload during their run. This implies that workload imbalances among the threads may occur, causing bottleneck to the system if the workload assignment is skewed, i.e., some processing units receive more work than the others. In parallel aggregate queries, the main cause of skewed execution is due to skewed data value distributions, because the assignment of data groups to processing units (which are GPU threads in our case) depends on data values, the distribution of which may be volatile. A continuous query engine should adapt gracefully to these changes or correct any bad initial workload assignments at runtime, in order to ensure efficient processing over time. To this end, runtime load balancing is employed, as data processing is dynamically reassigned among the card's threads.

In this paper, we investigate solutions to the problem of skewed execution in aggregate queries on GPUs, where the CPU and the GPU closely cooperate to achieve high performance. More specifically, we present a load balancing framework, where a load-balancing coordinator runs on the CPU to decide the allocation of groups to threads, whereas the execution of the query takes place on the GPU. Then, we introduce a family of load-balancing policies that instantiate the framework and we experimentally evaluate them. To the best of our knowledge, this is the first attempt to present runtime load-balancing techniques for database operations on GPUs.

The remainder of this article is structured as follows: the next section discusses the related work. Section 3 deals with the runtime load-balancing architecture. The detailed approaches to load-balancing are presented in Sec. 4. In Section 5, we evaluate the efficiency in skew handling for each of the proposed approaches. We conclude in Section 6.

2. RELATED WORK

An increasing number of researchers and practitioners use GPUs instead of CPU clusters for data- and computation-intensive problems. The proposals that are most closely related to our work include those that refer to the development of query processing techniques on GPUs. Although there are several early efforts towards this research direction (e.g., [2]), only recently fully-fledged query processing systems have appeared. The two most prominent examples are SphyraEna [1] and GPUQP [12, 13], which feature a fully functional DBMS with the capability to execute queries on the GPU, in order to benefit from its computing power. GPUQP provides a query engine where the queries (containing operators such as join, group-bys, and so on) run either entirely on the GPU or, in some cases, on both the CPU and the GPU. The database is not stored in the card's memory, but on the disk, as in conventional databases. When a query is to be executed, the system employs techniques to estimate the total cost, which includes data transfer to the GPU's memory and computational cost, and then to decide which parts of the query plan should be allocated to the GPU. Our work is different in the sense that we focus on a specific part of queries, namely aggregate queries, and we deal with the runtime load-balancing problem that is not considered by systems such as Sphyraena and GPUQP.

In addition, the MapReduce programming framework has emerged as an alternative environment for processing large amounts of data [5]. MapReduce inherently supports aggregate queries. The map phase splits data into groups, and the reduce phase is responsible for computing the aggregate function for each resulting group. An implementation of the MapReduce paradigm on GPUs has been proposed in [8]. Nevertheless, in MapReduce, no dynamic load balancing that modifies the allocation of reducers on the fly is supported.

A load balancing proposal for GPUs has appeared in [3]. The setting assumed by this work is quite different from the one in a streaming query though, since it relies on queues holding tasks. Each processor maintains a queue that contains the tasks to perform. When a task is carried out, it is popped from the queue and the processor deals with the next one. As a method of load balancing, when a processor is idle, it tries to steal tasks from the next processor's queue tail until all the tasks are completed. By contrast, in our work we do not employ queues for tasks, but each data group is assigned to a specific thread and decisions may be revised in each iteration, as explained in the following sections. Note also that our work is orthogonal to proposals that aim to fine tune GPUs in order to maximize performance (e.g., [24]).

Concerning stream processing and load balancing in CQ systems, a lot of available research material proposes methods for rebalancing data distribution to the processors [25, 22, 20, 21, 10, 23, 18]. Generally, the producer-consumer model is applied, where producers simply perform the data distribution and delegate the data processing to the consumers, who are responsible for the execution of the processing logic. Usually, the consumers do not have the same throughput, and the slowest of them acts as a bottleneck to the CQ system. Moreover, the data properties often change with time, causing uneven allocation and the need to redefine the query execution. In the Flux model [23], two methods of load balancing are proposed: the short term balancing method and the long term one. In the former case, a buffer for each consumer ensures that no producers will have to stay suspended until the consumer they want to deal with finishes his former work. As this method is inadequate for dealing with long term workload imbalances, Flux introduces a mechanism for long term rebalancing using state transfer. As data is divided in small partitions and distributed to the processors, when imbalance is detected, a partition is dynamically relocated to another processor on the fly. Flux techniques have been extended and improved upon in [10, 18]. Our work can be deemed as a proposal for techniques for long term imbalances in a GPU setting.

In general, techniques that modify the query execution at runtime are commonly referred to as adaptive query processing (AQP) ones [6]. AQP for CQs may have several additional flavors. For example, the proposals in [15, 26] try to effectively solve the load balancing problem by dynamically changing the query execution plan. More specifically, if multiple joins are being executed, the techniques devel-

oped change their order at runtime with a view to reducing the total processing cost. Other adaptive proposals that refer to CQs but do not deal with re-partitioning issues are discussed in [14, 19].

3. OUR LOAD-BALANCING FRAMEWORK

In this section we describe the architecture and the high-level approach to load balancing for aggregate queries on GPUs; the exact balancing algorithms are presented in Section 4. As the GPU is only suitable for executing kernels and the logic of the program is administered by the CPU, extensive collaboration between the two units is required. The CPU is responsible (i) for preparing the data so that coalesced memory access on the CUDA is enabled and (ii) for detecting and correcting imbalances. The GPU takes over the actual data processing. To this end, some auxiliary structures on both processing units are created in an effort to maximize the throughput of the system.

Throughout this paper, we consider a scenario where we execute a group-by query over data that consist of two fields, namely a group identifier *id* and an integer value *attr*. For each new tuple, the aggregate function needs to process the last attribute values received for each group identifier according to a fixed sized window; i.e., the aggregate value depends on the recent history of the stream. Although this is a specific scenario, it possesses all the characteristics so that it can easily be extended and generalized to further demanding streaming aggregate queries.

3.1 Operations on the CPU

As the data arrives at the CPU as a stream, the processing takes place in iterations. An example data and control flow is shown in Figure 1. In each iteration, a fixed size batch of tuples is processed, e.g., in the example in Figure 1 the batch size is 50K. The CPU maintains two auxiliary structures that (i) map groups to specific GPU threads that will process them and, (ii) map, in the reverse way, GPU threads to groups (these structures are not shown in the figure). Using the former, it organizes the data in a matrix, which will be later copied to the GPU's global memory, in a way that all the tuples of the groups allocated to the same thread will be in adjacent memory slots to allow for coalesced memory access; i.e., the matrix is not fully sorted. The *Reordered Data matrix* in Figure 1 illustrates an example. This matrix is produced in linear time, and more specifically, in two passes. In the first pass, for each thread, we count the occurrences of the data items that belong to groups assigned to that thread. This provides adequate information about the exact places in the matrix, where each data item should be placed, given that we know the mapping of groups to threads. The actual placement takes place in the second pass.

The CPU also utilizes an array that indicates where each GPU's thread should seek for its assigned data during the kernel launch; this is called *threadDataIndicator*.

Before copying the data to the GPU and calling the kernel, the skew handling techniques intervene, in order to investigate whether any imbalance occurs. The rebalancing technique used might then propose a new mapping of a group to a different thread. So, the group-to-thread mapping structures are continuously updated. However, since it is expensive to revise the ordered data matrix and its associated thread indicator array, the effects of the rebalancing take effect from the next iteration; in other words there is a delay of one iteration in our skew handling approach.

At the end of each iteration, the *Reordered Data matrix* and the *threadDataIndicator* array are copied to the GPU's global memory. At this point, the kernel is launched asynchronously as a single CUDA stream. With the help of these two data structures that are copied to the GPU, we ensure that every thread will be capable of instantly locating the data it is responsible for. Note that in GPGPU, each thread can be uniquely identified in a straightforward manner.

While waiting for the GPU to finish with the processing of the batch of tuples sent, the CPU prepares the next batch of streaming data. As explained above, for the next batch, the allocation of groups to threads will be based on the group-to-thread mappings that have resulted from the previous batch.

It is important to mention that, as verified also by our experiments, the preparation time on CPU overlaps with the more expensive GPU operations. When the grid size is small, the overlap is full and thus the load-balancing overhead is hidden. However, when the grid size increases, the overlap is partial and the overhead has an impact on the total running time.

3.2 Operations on the GPU

Upon receipt of a new batch of ordered tuples, each thread starts processing tuples. To do so, it accesses only the relevant data matrix cells (with the help of the *threadDataIndicator* array). For each tuple, the GPU thread adds it to a persistent data structure in the global memory that holds all the windows for all groups. If the window is full, the oldest tuple of the window has to be discarded. Overall, the auxiliary data structures on the GPU are: (i) a matrix structure that keeps the windows for all the groups, (ii) an array that maps each group to its window in the previous matrix, and (iii) an array structure that contains pointers to the oldest value of the window for each group (*nextPos*) (see Figure 2). When the oldest tuple is replaced, the pointer moves to the next cell in the same window.

To compute the aggregate function, each thread, for each allocated group, processes all the tuples within the window. When all the tuples have been processed, the kernel call terminates. Note that no new kernel can be called before termination. This implies that no data on the GPU are replaced with new data before their processing, thus ensuring correctness. As shown by the experiments in the next section, such an approach is not only capable of skew handling, but can also increase the system throughput significantly. However, there may still exist cases, where the data arrival rate exceeds the GPU capacity; in that case, our proposal needs to be complemented with load shedding and/or approximate techniques, which we leave for future work.

4. LOAD-BALANCING TECHNIQUES

As we have previously mentioned, the number of tuples each thread has to process is calculated on the CPU before the kernel call. At that stage, it is easy to identify imbalances. In all the re-balancing methods except the last one, we follow the same pattern: we keep two heaps, a min heap and a max heap, which contain information about the most and least loaded threads, respectively (in $O(1)$ time). Then, we match the most and the least loaded thread into

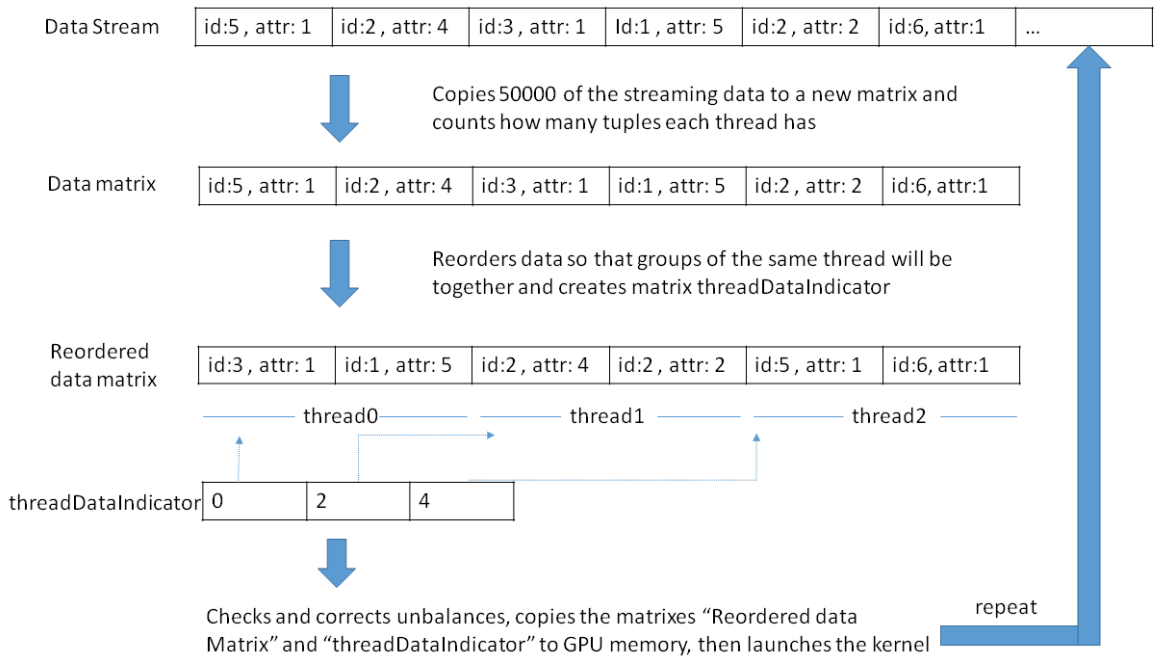

Figure 1: High level data flow and CPU operations.

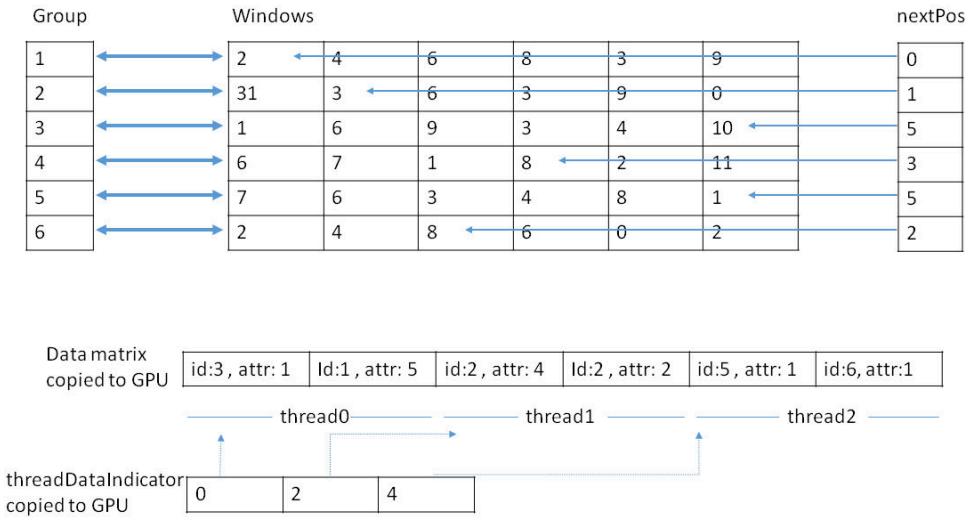

Figure 2: Auxiliary data structures on the GPU.

a pair. To indicate imbalance, we introduce a threshold of tuple count difference. If the difference is above the threshold, a group is moved from the most loaded thread to the least loaded one, depending on the technique selected, and the mapping structures are updated. Then, we execute the same process for the new most and least loaded threads. Since the number of this type of iterations may be high, the choice of heap data structures is beneficial. The methods for choosing the group that will be moved in case of imbalance are explained below.

getFirst is the least sophisticated technique and requires only small computation effort. The group to be moved is the first one in the thread-to-group mapping structure. The technique is given in Figure 3. In *checkAll* (see Figure 4), the group to be moved is the one with the most appearances in the thread. All the tuples of the thread in the current

Algorithm getFirst ($\vec{tpt}$, $threadThreshold$)
$\vec{tpt}$: a vector containing the number of tuples per thread,
$threadThreshold$: threshold to indicate imbalance

1. Find the thread with the highest number of tuples assigned $tmax$;
2. Find the thread with the lowest number of tuples assigned $tmin$;
3. **While** $tpt[tmax] - tpt[tmim] > threadThreshold$;
4. Assign the first group of thread $tmax$ to the thread $tmin$;
5. Find the new $tmax$,$tmin$;
6. **endWhile**;

Figure 3: Outline of the *getFirst* algorithm.

Algorithm checkAll ($\vec{tpt}$, $threadThreshold$)
$\vec{tpt}$: a vector containing the number of tuples per thread,
$threadThreshold$: threshold to indicate imbalance

1. Find the thread with the highest number of tuples assigned $tmax$;
2. Find the thread with the lowest number of tuples assigned $tmin$;
3. **While** $tpt[tmax] - tpt[tmim] > threadThreshold$;
4. Read the tuples of thread $tmax$ and find the group with the most occurrences;
5. Assign that group to the thread $tmin$;
6. Find the new $tmax$,$tmin$;
7. **endWhile**;

Figure 4: Outline of the *checkAll* algorithm.

Algorithm probCheck ($\vec{tpt}$, $threadThreshold$, pot)
$\vec{tpt}$: a vector containing the number of tuples per thread,
$threadThreshold$: threshold to indicate imbalance,
pot: thepercentage of thread's tuples that the to-be-reassigned group has to cover

1. Find the thread with the highest number of tuples assigned $tmax$;
2. Find the thread with the lowest number of tuples assigned $tmin$;
3. **While** $tpt[tmax] - tpt[tmim] > threadThreshold$;
4. $ngroups$ = number of groups assigned to $tmax$;
5. $limit = pot * tpt[max]/ngroups$;
6. Read the tuples of thread $tmax$ and stop when a group appears $limit$ times;
7. Assign that group to the thread $tmin$;
8. Find the new $tmax$,$tmin$;
9. **endWhile**;

Figure 5: Outline of the *probCheck* algorithm.

batch have to be scanned, in order to count the group appearances. Then, the most frequent group is selected and remapped with a view to correcting the imbalance faster than the previous technique.

probCheck is as an approximate version of *checkAll*, which tries to locate the most common group without having to scan all the tuples. To manage this, it first calculates the average number of tuples in the groups of the most loaded thread. Then, it selects the first group detected with frequency equal to *pot* times that average value ($0 < pot \leq 1$). The higher *pot* is, the higher the chance is the most common group will be selected at the expense of increased data scanning cost; however, the scanning cost is always less than the cost of scanning all the thread's tuples. *probCheck* is shown in Figure 5.

The algorithms thus far make simple choices as to which groups should be allocated to other threads. Neither random choices nor selecting the most frequent group can guarantee that the imbalance will be eliminated across all threads. *bestBalance* tries to address this limitation and detects the group that, if remapped, will achieve the best balance between the two corresponding threads. To achieve this, it scans all the tuples assigned to the most loaded thread, counting the appearances of each group. Then, it chooses the group that minimizes the difference in the workload, as shown in Figure 6.

Algorithm bestBalance ($\vec{tpt}$, $threadThreshold$)
$\vec{tpt}$: a vector containing the number of tuples per thread,
$threadThreshold$: threshold to indicate imbalance

```
1.  Find the thread with the highest number of tuples assigned tmax;
2.  Find the thread with the lowest number of tuples assigned tmin;
3.  While tpt[tmax] − tpt[tmim] > threadThreshold;
4.      Read the tuples of thread tmax and find the group that, if swapped, minimizes tpt[tmax] − tpt[tmin];
5.      Assign that group to the thread tmin;
6.      Find the new tmax,tmin;
7.  endWhile;
```

Figure 6: Outline of the *bestBalance* algorithm.

Algorithm shift ($\vec{tpt}$, $threadThreshold$)
$\vec{tpt}$: a vector containing the number of tuples per thread,
$threadThreshold$: threshold to indicate imbalance

```
1.  Find the thread with the highest number of tuples assigned tmax;
2.  Find the thread with the lowest number of tuples assigned tmin;
3.  While tpt[tmax] − tpt[tmim] > threadThreshold;
4.      if tmax > tmin
5.          foreach thread i ∈ (tmin, tmax]
6.              Move the first group from thread i to the thread i − 1;
7.      else
8.          foreach thread i ∈ [tmax, tmin)
9.              Move the last group from thread i to the thread i + 1;
10.     Find the new tmax,tmin;
11.     endif
12. endWhile;
```

Figure 7: Outline of the *shift* algorithm.

Algorithm shiftLocal ($\vec{tpt}$, $threadThreshold$)
$\vec{tpt}$: a vector containing the number of tuples per thread,
$threadThreshold$: threshold to indicate imbalance

```
1.  foreach thread i
2.      if tpt[i] − tpt[i + 1] > threadThreshold;
3.          Move the last group from thread i to the thread i + 1;
4.      else if tpt[i + 1] − tpt[i] > threadThreshold;
5.          Move the first group from thread i + 1 to the thread i;
```

Figure 8: Outline of the *shiftLocal* algorithm.

The last two methods aim to further benefit from coalesced memory access. More specifically, the *shift* method inserts a locality criterion in the skew handling according to which the groups are not moved directly from one thread to another, but only to the neighboring one. There are two cases: if the loaded thread has a smaller id number than the emptier one, then each thread in the range $[loaded, unloaded)$ has its last group assigned to the next thread. Consequently, the loaded thread will have to process one group less, while the least loaded one is now assigned with some extra load. The same happens if the loaded thread has a bigger id, only the other way around (see Figure 7).

shiftLocal does not rely on the detection of the most and least loaded threads, thereafter it does not require the two heaps. *shiftLocal* only fixes imbalances among neighboring threads. It compares each thread's load to the next one's, using an appropriate threshold factor, and properly moves the last or first group to the less loaded thread (see Figure 8).

5. EVALUATION

In this section, we evaluate the efficiency and the effectiveness of our approach to skew handling. We focus on

both performance improvements and the associated overheads. For completeness, we examine scenarios with no, low and high imbalance.

5.1 Experimental Setting

For the purpose of our experiments, two different system configurations supporting the Fermi architecture are used: the first system (referred to as *PC1*) has an Intel Core2 Duo E6750 CPU at 2.66GHz and an NVidia 460GTX (GF104) graphics processor at 810 Mhz on a PCIe v2.0 x16 slot (5GB/s transfer rate). *PC2* has an Intel P4 550 CPU, running at 3.4 Ghz. Also, it has an NVidia 550GTX Ti (GF116) at 910 MHz, which is installed on a PCIe v1.1 x16 (2.5GB/s transfer rate) slot. In both cases, our techniques have been developed using the NVidia Parallel NSight 2.1 platform. The two configurations are appropriately selected to favor the investigation of the relative performance of both (i) a system with slower CPU but more powerful GPU and (ii) a system with a faster CPU but slower GPU.

Also, we experiment with three datasets, namely *DS1, DS2* and *DS3*. *DS1* consists of unskewed data. It comprises 100M tuples, assigned to 40000 groups in a round robin way, so that the allocation of tuples to groups follows a uniform distribution. This dataset does not require any runtime balancing and is used for comparison purposes. *DS2* follows a zipf distribution. It consists of 100M tuples as well. In *DS2* the group ids are assigned in such a way that a group with id equal to y is more frequent than the groups with id z, if $z > y$. Finally, *DS3* is a randomly permuted version of the *DS2*, so that the group ids are not in decreasing order of frequency.

In each iteration, the batch of tuples consists of 50K tuples. As such, in our experiments, the processing finishes after 2,000 iterations for all datasets, but the results can be transferred to infinite streams as well. The size of the sliding window that needs to be maintained for each group is 100 tuples, and, initially each thread receives an equal number of groups with consecutive group ids. After every new tuple is copied to the appropriate window of a group, the complete window is scanned and its sum is calculated from scratch thus simulating a demanding data analysis task. When executing the kernel, we set the block size to 256 threads, but we vary the grid size. The *threadThreshold* value is set to 1000. For the *probCheck* method, the *pot* parameter is set to 0.5, which was experimentally found as the optimal value. All experiments were conducted three times and the average value is presented. The standard deviation is depicted when it is not negligible; in general it is very small.

5.2 Experiments

5.2.1 Performance degradation due to imbalance

We start our experiments by showing the detrimental effects of not performing load balancing when data is skewed. In Figure 9, the left column that corresponds to DS1 demonstrates the performance of our aggregation operator in the optimal case: dynamic balancing is neither needed nor performed (i.e., there is no overhead). However, for DS2, if no balancing technique is activated, the execution time increases by an order of magnitude, even if the total size to be processed remains the same. This is because only a few threads are burdened with the majority of the processing, while the others remain idle. In the case of DS3, although the allocation of tuples in groups is still skewed, the most common groups are randomly distributed among the threads, rather than being allocated to the first ones, so the overall increase in the execution time is smaller compared to DS2. In general, DS2 is regarded as a scenario, where the imbalance is high, and DS3 corresponds to a scenario with lower imbalance.

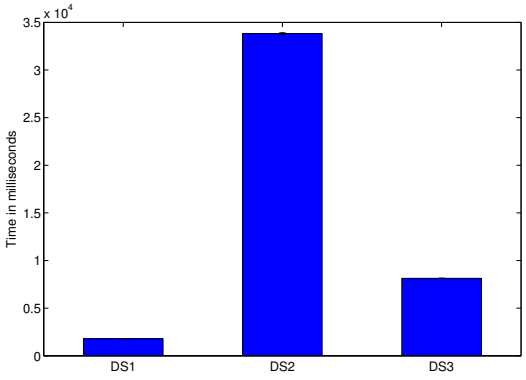

Figure 9: Comparison of execution without load-balancing in PC1 for grid size 4.

5.2.2 Performance improvements

In Figure 10, we present the evaluation results of the six methods of skew handling that were discussed in Section 4 for DS2 (high imbalance), and we compare them against the case where we do not perform load balancing. As expected, runtime skew handling results in considerable improvement in the execution of parallel aggregate queries execution, since the workload is more evenly distributed among the graphic card's processor units. However, as indicated in the figures, not all the methods can achieve the same improvement. More specifically, *shift* and *shiftLocal* are inferior to the other four methods for this dataset. This is due to the fact that, while these methods handle the rebalancing by moving groups to neighboring threads, in the DS2 dataset, the major amount of tuples is distributed among the first threads. Therefore, in order to maintain an acceptable level of balance, *shift* and *shiftLocal* require many iterations in order to fix the imbalance.

Comparing the other methods, we can draw the following observations: *probCheck* is slightly more effective than *checkAll*, as it chooses faster, albeit in a probabilistic manner, the optimal group to move, without having to consider all the tuples of the stream. On the other hand, the *bestBalance* method guarantees the optimal solution for the skew handling problem in terms of equalizing the workload among GPU threads. This implies that it requires the least group repartitioning effort in future batches. Nevertheless, as the balancing decisions are enforced after one round, where the conditions might have changed, and the demand for CPU processing time in order to compute the best solution increases, in most of the cases, the execution time of *bestBalance* is slightly higher than those of *getFirst, checkAll* and *probCheck*. Interestingly, as *getFirst* arbitrarily moves the first group of the most loaded thread, it cannot deal with severe imbalances successfully in a few rebalancing iterations,

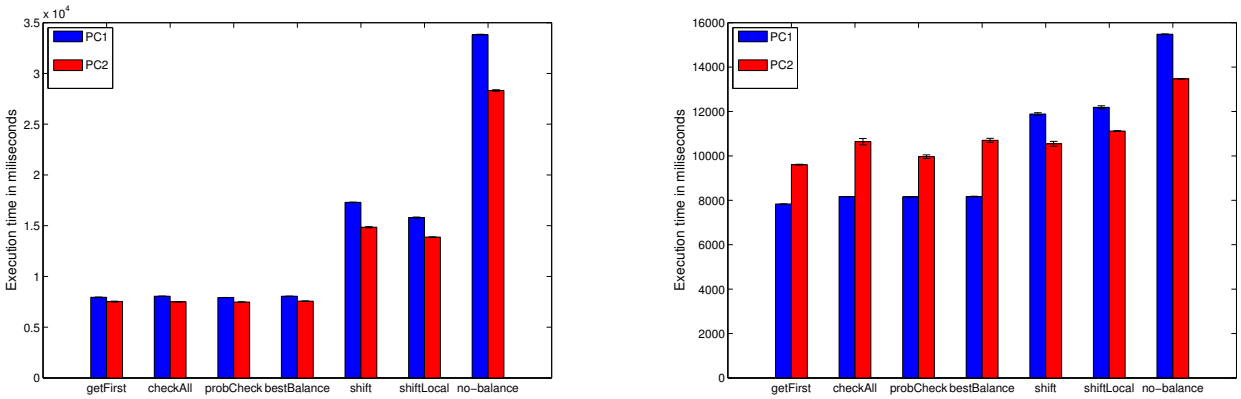

Figure 10: Comparison of execution time of load-balancing methods for DS2 (high imbalance) with grid size 4 (left) and 64 (right).

	High Imbalance (DS2)		Low Imbalance (DS3)	
Technique	PC1	PC2	PC1	PC2
getFirst	4.26	3.76	0.97	0.97
checkAll	4.20	3.78	1	0.89
probCheck	**4.27**	**3.79**	1	**1**
bestBalance	4.20	3.75	1	0.9
shift	1.96	1.91	**1.01**	0.99
shiftLocal	2.14	2.04	0.99	**1**
no balance	1	1	1	1

Table 1: Normalized throughput (tuples processed in time units) for grid size 4.

	High Imbalance (DS2)		Low Imbalance (DS3)	
Technique	PC1	PC2	PC1	PC2
getFirst	**1.98**	**1.4**	0.93	0.81
checkAll	1.9	1.27	0.93	0.76
probCheck	1.9	1.35	0.93	0.8
bestBalance	1.9	1.26	0.93	0.77
shift	1.3	1.28	0.97	0.82
shiftLocal	1.27	1.21	**1**	**0.99**
no balance	1	1	1	1

Table 2: Normalized throughput (tuples processed in time units) for grid size 64.

but because of its low overhead, it leads to competitive query execution time.

As mentioned earlier, DS2 corresponds to a scenario with high imbalance. In DS3, the imbalance is lower, and also, the loaded threads may have arbitrary ids. We repeat the same experiment for DS3, and the results appear in Figure 11. There are two main observations: firstly, no load balancing technique is actually effective, and secondly, *shift* and *shiftLocal* behave similarly if not better than the rest of the techniques.

The normalized results are summarized in Tables 1 and 2, where the value 1 corresponds to the throughput of the no balance technique in each setting. When the imbalance is high, *probCheck* dominates all the other rebalancing options in terms of performance for small grid sizes. In those cases, the speed up is more than 4 times. When the grid size is increased, the best performing technique is *getFirst*, which nearly doubles the throughput. So, we can deduce that less sophisticated and approximate load balancing techniques, which require less computational effort for the balancing itself, are more appropriate for GPGPU in highly skewed environments. When the imbalance is low, the maximum speedup is negligible for grid size set to 4, and *shift* and *shiftLocal* seem more appropriate. When we further increase the grid size, the throughput may degrade. We also discuss the impact of the grid size in more detail later.

Finally, as mentioned before, the time spent in load balancing on the CPU may be hidden by the time spent in GPU execution and data transfer. This holds for the case where the grid size is set to 4. When we set the grid size to 64, the CPU-based data preparation partially overlaps with the GPU-based aggregate computation, and, as a result, the preparation overhead affects the performance.

5.2.3 Overhead

To further investigate the overhead incurred by maintaining the auxiliary structures and performing the rebalancing computations, in Figure 12, we show the increase in the total time required when we let the skew handling techniques to unnecessarily investigate possible rebalancing actions; to this end we use DS1. From the figure, we can observe that simply enabling any of the methods has an impact on the performance, which is more evident with increased grid sizes. Actually, the overhead is negligible for the small grid size, where the CPU cost is fully hidden by the GPU processing. In addition, *shiftLocal* is the technique with almost negligible overhead for any combination of system and grid size; this is attributed to the fact that it does not utilize heap structures for finding the most and least loaded groups.

5.2.4 More on the effect of grid size

The grid size, as evidenced in all the performance figures thus far, is an important parameter that affects the performance. By increasing the grid size, we increase the number of threads produced in the card. Consequently, the number of groups to be processed by a thread is reduced. As a result, the odds of having a thread, which gathers many overloaded groups decrease; in other words, imbalance effects are indirectly mitigated through increased grid sizes. Thus, a larger grid size leads to shorter processing time, as much fewer

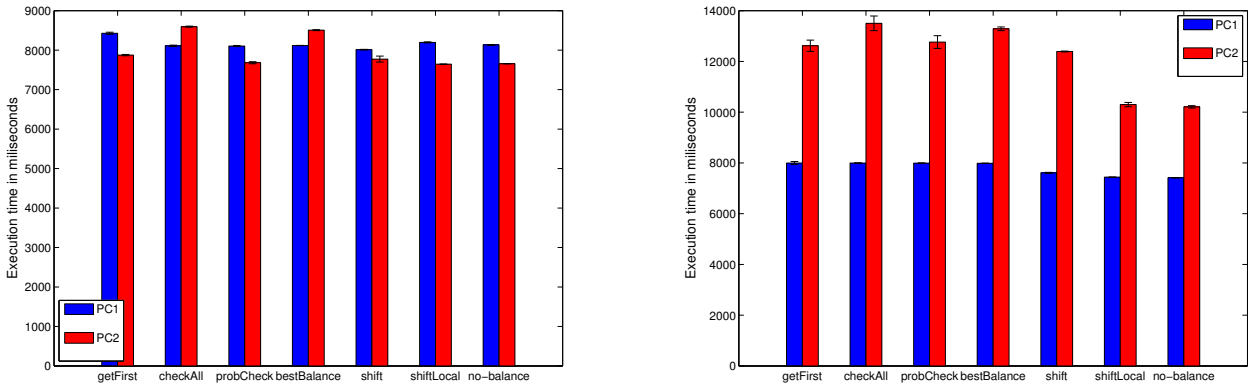

Figure 11: Comparison of execution time of load-balancing methods for DS3 (low imbalance) with grid size 4 (left) and 64 (right).

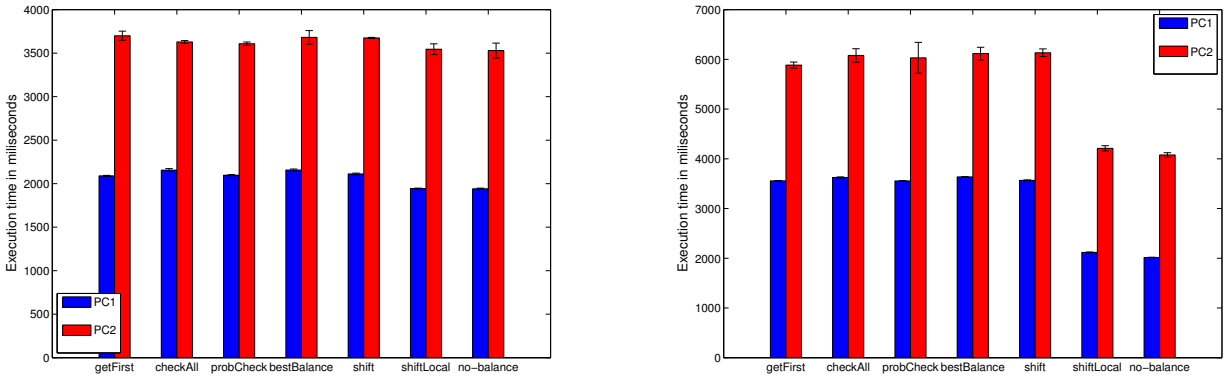

Figure 12: Comparison of execution time of load-balancing methods for DS1 (no imbalance) with grid size 4 (left) and 64 (right).

threads will run for very long time acting as a bottleneck. Taking advantage of the very low thread scheduling time offered by the Fermi architecture, we experience reduced total times, even without any of the skew handling techniques enabled. However, increasing the grid size makes sense only when there is at least one group in the aggregate query per thread. In our case, where we have 40K groups, this has proved to be not a problem, but in several other scenarios the total number of groups in the aggregate query may be a few hundreds, which necessitates smaller grid sizes because larger grid sizes are simply not applicable in those scenarios.

Moreover, if we increase the number of threads over a certain point, we reach a saturation point where there is no more benefit. This is due to the fact that the CPU, which maintains the structures that keep data for the thread load factor, now needs more time to process them. Consequently, the total execution time cannot be reduced under a limit, and in some cases the additional CPU effort might outweigh the benefit. Only in the case of *shiftLocal*, which does not need those extra structures and computations, the surcharge is small. Additionally, we notice that the *shift* method also benefits from the decrease of groups per thread due to increased grid size, as it has higher chances to achieve acceptable balance.

Figure 13 shows the impact of the grid size on the running time for DS2 being executed on PC1. We can observe that, by increasing the grid size, we can alleviate high imbalances, although we cannot eliminate them, since the difference between no balancing and balancing schemes decreases significantly but it never becomes negligible. For *shift* and *shiftLocal*, the saturation point is much earlier, whereas all the other techniques are insensitive to the grid size.

5.2.5 CPU vs GPU

Additionally, in order to demonstrate the power of the GPU processing, we implemented a simple CPU-based group by algorithm. There, as the data is not distributed among threads but handled serially, no skew handling is needed and the total throughput is the same in all datasets of equal size. The execution times are presented in the Figure 14. Compared with the running times in Figures 10-12, it is evident that employing the GPU is always beneficial. For example,

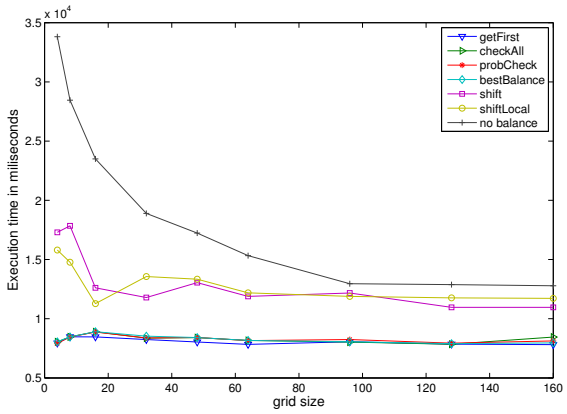

Figure 13: The effect of changing the gridsize in DS2 (PC1).

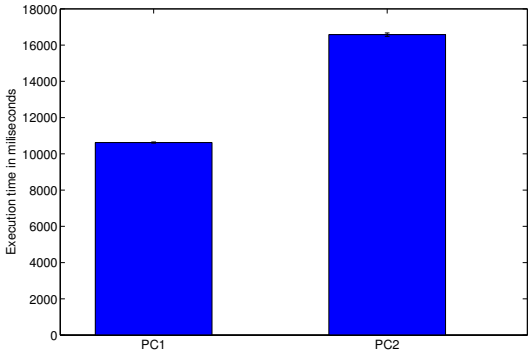

Figure 14: Execution time in CPU.

in DS2, the running time using the GPU can be as low as approximately 8 secs for both systems, whereas it is more than 10 and 16 secs when we use only the CPU of PC1 and PC2, respectively. The performance benefits are larger for DS1 and DS3.

5.2.6 *Further experiments and discussion*

In our last experiment, we investigate adding some extra load: we assume an aggregate function that checks the window elements 10 times instead of one per window update. Despite the fact that modern graphics cards are alleged to perform much better than the CPU on processing numeric and double precision data, rather than on accessing data from the card's global memory for a simple summing computation, their superiority and capability to yield improved performance is obvious in Figure 15.

Finally, as evidenced in the figures, there is no clear winner between the two systems. PC2, owing to the faster and more powerful graphics processing unit, proved to be faster in some cases were the majority of computation happens in the GPU. DS2 is such a case, where the imbalance among the threads acts as a bottleneck, and the CPU processing time is negligible compared to the GPU processing. Also,

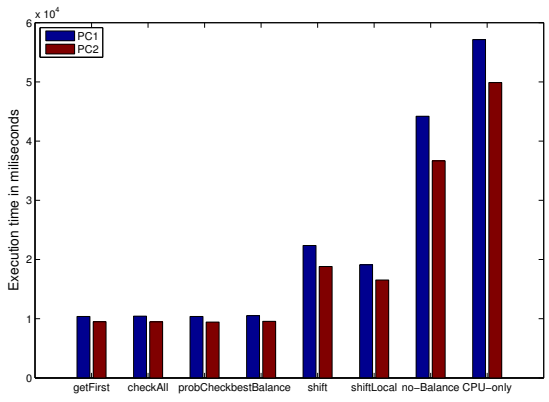

Figure 15: Performance comparison when there is a 10-fold increase in the window passes for DS2 (the grid size is 4).

PC2 benefits from the smaller grid sizes that require less CPU effort in order to handle the corresponding structures. Meanwhile, PC1 outperforms PC2 in the case of bigger grid sizes, where the need for skew handling is decreased, as explained earlier.

6. CONCLUSIONS

In this work, we dealt with the problem of skewed execution in aggregate queries on GPUs. We presented a generic load balancing framework along with specific balancing techniques. The lessons learnt can be summarized as follows: Firstly, we can significantly reduce execution time with the help of GPUs. In our experiments, we observed significant speed-ups up to 4 times and verified the fact that load imbalances can lead to serious performance degradations thus necessitating load balancing actions in order to avoid performance degradation. Secondly, the techniques we proposed are both efficient and effective; their efficiency is due to their low overhead, whereas their effectiveness is manifested through their capability to lead to throughput improvements in highly skewed scenarios. Interestingly, less sophisticated and approximate techniques exhibit superior performance, because the increased overhead of more sophisticated solutions outweighs any benefits and/or load balancing decisions are enforced with a delay of one round, where the exact conditions may have changed anyway. When small imbalances are experienced, no balancing technique behaves well. Finally, increasing the grid size mitigates the effect of skewed executions, but it is not always applicable in aggregate queries. Overall, this work aims to provide useful insights into the behaviour of rebalancing techniques for GPU-assisted data management.

Potential avenues for future work include the investigation of integrated techniques that both balance the load among the threads and vary the grid and the batch size, and of orthogonal issues, such as more efficient methods of data exchange between the GPU and the main memory. Finally, it is worth investigating the behaviour of our approach in the light of the recent dynamic parallelism extensions of the

CUDA programming model, introduced with the Kepler architecture.

7. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

8. REFERENCES

[1] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *GPGPU*, pages 94–103, 2010.

[2] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1021–1032. VLDB Endowment, 2004.

[3] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, 2008.

[4] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54:88–98, Aug. 2011.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[6] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[7] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[8] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Trans. Parallel Distrib. Syst.*, 22(4):608–620, 2011.

[9] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2 edition, 2008.

[10] A. Gounaris, C. A. Yfoulis, and N. W. Paton. Efficient load balancing in partitioned queries under random perturbations. *TAAS*, 7(1):5, 2012.

[11] J. Han and J. Gao. Research challenges for data mining in science and engineering. In H. K. et al, editor, *Next Generation of Data Mining*. Chapman & Hall, 2009.

[12] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), 2009.

[13] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *SIGMOD Conference*, pages 511–524, 2008.

[14] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In *ICDE Workshops*, pages 16–21, 2011.

[15] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 49–60. ACM, 2002.

[16] NVIDIA. Nvidia s next generation cuda compute architecture: Fermi(whitepaper), 2010.

[17] NVIDIA. Nvidia cuda programming guide, 2012.

[18] N. W. Paton, J. B. Chávez, M. Chen, V. Raman, G. Swart, I. Narang, D. M. Yellin, and A. A. A. Fernandes. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDB J.*, 18(1):119–140, 2009.

[19] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. In *ICDE Workshops*, pages 10–15, 2011.

[20] E. Rahm. Dynamic load balancing in parallel database systems. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, pages 37–52. Springer, 1996.

[21] E. Rahm and R. Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In R. Agrawal, S. Baker, and D. A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings.*, pages 182–193. Morgan Kaufmann, 1993.

[22] E. Rahm and R. Marek. Dynamic multi-resource load balancing in parallel database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, pages 395–406. Morgan Kaufmann, 1995.

[23] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 25–36. IEEE Computer Society, 2003.

[24] H. H. B. Sørensen. Auto-tuning dense vector and matrix-vector operations for fermi gpus. In *PPAM (1)*, pages 619–629, 2011.

[25] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A parallel hash join algorithm for managing data skew. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1355–1371, 1993.

[26] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 431–442. ACM, 2004.