저작자표시-비영리-변경금지 2.0 대한민국

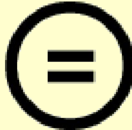
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

SIMD-Conscious Optimization of Star Schema Query Processing

스타-스키마 쿼리 처리의 SIMD 최적화

2015년 2월

서울대학교 대학원

전기 컴퓨터 공학부

송 상 협

SIMD-Conscious Optimization of Star Schema Query Processing

지도교수 차 상 균

이 논문을 공학석사 학위논문으로 제출함

2015년 2월

서울대학교 대학원

전기 컴퓨터 공학부

송 상 협

송상협의 공학석사 학위논문을 인준함

2015년 2월

위 원 장 이 재 진 (인)

부위원장 차 상 균 (인)

위 원 정 교 민 (인)

Abstract

Most modern CPUs today come equipped with SIMD (Single Instruction, Multiple Data) registers and instructions, which allow for data-level parallelism by offering the ability to execute a given instruction on multiple elements of data. With its wide availability and compiler support, lack of need for hardware changes and potential for boosting performance, exploiting SIMD instructions in database query processing has been the subject of some attention in literature.

Star schemas are a popular method of data mart modeling, and with the sharp rise in the need for efficient big data analysis, star schemas serve as an important case study for OLAP performance optimization. Whilst literature on SIMD optimization of star schema queries exists for the GPGPU domain - where the GPGPU method of execution is synonymous with the SIMD paradigm - none has explored the topic using SIMD instructions on CPUs.

In this paper, we show that by optimizing star schema query processing for SIMD instructions, speedup in excess of four times can be achieved in performance. Instead of relying on the traditional operator-based query processing model, we focus on the so-called invisible join; an algorithm specialized for star schema joins. We describe the steps and procedures involved in the SIMD-conscious optimization of the invisible join algorithm, and demonstrate that our SIMD optimization

methods achieve up to 4.8x overall speedup over its scalar equivalent, and up to 6.4x speedup for specific operations.

Keywords : SIMD query processing, star schema, in-memory column-store
Student Number : 2012-23953

Table of Contents

Abstract I

Table of Contents III

List of Figures V

Chapter 1. Introduction 1

Chapter 2. Related Work 5

Chapter 3. Star Schema and Invisible Join 7

- 2.1 The Star Schema 7
- 2.2 The Invisible Join 8

Chapter 4. SIMDification of Invisible Join 13

- 4.1 Extending the Invisible Join 13
- 4.2 SIMDification of the Invisible Join 15

Chapter 5. Experimental Results 21

- 5.1 Experimental Setup 21
- 5.2 Overall Results 22
- 5.3 Breakdown of Results 23

Chapter 6. Conclusion and Future Work ·············· 30

References ·· 32
국문 초록 ·· 35
Acknowledgements ·· 37

List of Figures

Figure 1: Layout of a typical star schema ········· 8
Figure 2: A typical star schema query ········· 9
Figure 3: Step 1 of the invisible join ········· 10
Figure 4: Step 2 of the invisible join ········· 11
Figure 5: Step 3 of the invisible join ········· 12
Figure 6: Implicitly performed join ········· 14
Figure 7: SIMDification of the invisible join ········· 15
Figure 8: Fetching join key values ········· 16
Figure 9: Overall execution time ········· 22
Figure 10: Distribution of overall execution time ········· 23
Figure 11: Performance of join key fetch ········· 24
Figure 12: Performance of filter processing ········· 25
Figure 13: Performance of group by ········· 27
Figure 14: Performance of aggregation ········· 28

Chapter 1. Introduction

SIMD (Single Instruction, Multiple Data) refers to a form of vector processing which enables the processing of multiple elements of data with a single instruction. SIMD-enabled systems typically function by having a set of dedicated SIMD registers - which are larger in capacity than normal registers - and instructions: the extra-large SIMD registers have data elements of fixed size loaded onto them as vectors, after which a given SIMD instruction executions its operation with each of the vector elements. This allows for a theoretical degree of parallelism equal to the number of data elements which can fit into the SIMD register.

Initially added as extensions in 1997 to the x86 instruction set for the purpose of accelerating multimedia processing, SIMD has evolved over time to become a fully functional, general purpose instruction set for vector processing. With SIMD optimization support from major compiler vendors and ubiquitous compatible hardware, software can easily and safely be optimized to exploit SIMD architecture. In addition, SIMD hardware and technology continues to evolve: AMD has dropped its own SIMD implementation (3DNow) in favor of Intel's implementation (SSE/AVX) for a more unified landscape; Intel plans to increase the width of its SIMD registers to 512 bits by 2015 (128-bit registers were the most common at the time of writing); the general purpose GPU (GPGPU) vendors - whose products essentially follow the SIMD paradigm - continue to improve their hardware (4,992 cores with 24 GB of memory, as of writing).

There has been extensive work covering database operations for SIMD architectures, spread across three different hardware types: CPU, GPU, and integrated CPU/GPU. The GPU platforms have been covered extensively in [3] [5] [6] [7] [8], addressing the main bottlenecks of that platform – lack of GPU memory capacity and slow data transfer rates between GPU and main memory. Although impressive results can be achieved with GPUs, the aforementioned bottlenecks have prevented GPUs from having impact in the commercial DBMS market. Integrated CPU/GPU hardware – which essentially eliminate the memory capacity and data transfer bottlenecks – has also been covered to some extent in [9], which focused on the effective utilization of the available hardware by distributing operators across the devices. It is notable, however, that only a relative few have focused on SIMD instructions for CPU. The work in [19] one of the first to put focus to SIMD acceleration of database operators, covering the SIMDification process of some basic operator logic. The work in [16] and [17] focused purely on the scan operation, describing the SIMDification process of column-store predicate handling with SIMD instructions. [2] and [10] studied the performance of hash and sort-merge joins – a widely discussed topic in academia – with each producing highly optimized versions of the joins; while SIMD instructions were used throughout the optimization process, SIMDification was not the main focus.

It is also important to note that all the research in this field so far have based their work on the traditional operator-based query processing model – the so-called Iterator model [12]. The iterator model consists of a number of discrete operators, each independently performing a specific database function – such as filtering, joining and grouping – based on their set of inputs. The incoming query is

analyzed to produce an optimized query plan consisting of the required operators, based on the costs associated with each operator. While this model of query processing is elegant and widely used throughout the industry, it is very much a general-purpose query processing model, with all the overheads associated with generalization. Indeed, the work presented in literature so far is all operator-specific, with performance benefits limited to within operator boundaries. There has not been any work to date which explored SIMD optimization outside the bounds of the iterator model, e.g. for specialized database use-cases. As specific scenarios call for specialized optimization, such a process can offer valuable optimization opportunities.

For this study, we concentrate on the SIMD optimization of star schema query processing. Star schemas are a popular method of organization in a data warehousing system. Specialized for fast and interactive analysis of data, star schemas are arranged in a heavily denormalized manner - consisting of a single large fact table containing data recorded at very atomic levels, and multiple dimension tables of smaller size containing attributes which describe the fact table data. As dimension tables in a star schema are arranged in a single hierarchy, the amount of joins required to process a given query is reduced, thereby resulting in simpler queries and faster response times. Owing to their simplicity and performance, star schemas are popular, supported by a large number of business intelligence applications.

Although star schemas have been the subject of many studies such as [13] and [15], to the best of our knowledge, the only work covering the SIMD optimization of star schemas was in [18] - on GPU hardware, and again, based on the iterator model. We base our

work on the so-called Invisible Join described in [1] - an extremely streamlined join algorithm designed for processing of star schemas, taking advantage of column-store data structures. We describe the SIMDification process of the invisible join algorithm, including the SIMDification process of the different stages of the algorithm. We demonstrate the efficiency of our SIMDified algorithm, achieving over four times speedup over its scalar equivalent. We show that the algorithm is flexible enough to handle all the queries in the Star Schema Benchmark [14], and that the algorithm can easily be ported to other SIMD-based hardware platforms, such as GPUs.

The rest of this paper is structured as follows: in Chapter 2, we present an overview of the previous related work. In Chapter 3, we briefly describe the star schema and the invisible join technique, while Chapter 4 details our SIMDification of the invisible join. Experimental results are presented in Chapter 5, and we make our concluding remarks and outline future work in Chapter 6.

Chapter 2. Related Work

The use of SIMD architecture for DBMS operations have been studied to some extend in literature, with work generally spread across three different hardware platforms: CPU, GPU, and integrated CPU/GPU.

The fundamental operators and first examples of join operators on GPUs were presented in [5] and [8], which identified the lack of on-board memory and poor data transfer rates between it and main-memory as key bottlenecks for the GPU platform; follow-up research mostly focused on addressing these bottlenecks - data compression for efficient PCI-e transfers [3], high throughput transactions [7], and CPU/GPU co-processing [6], where workload was distributed between CPU and GPU for effective utilization of the hardware. The idea of co-processing was further explored in [9], utilizing pre-calculated cost metrics to determine the placement of operators on devices.

CPU-based SIMD optimization of database operations has also been explored to some degree in literature. [19] presented one of the first examples of optimizing query processing operator logic explicitly using SIMD instructions, covering the basic scan, nested loop join and aggregation operations, as well as index tree search operations. In [16] and [17], the authors focused on optimizing for the scan operator, and described SIMD techniques for evaluating predicates with compressed column-store data. The authors of [10] described their technique for loading CSV data from disk to memory, utilizing SIMD string operations for parsing of the data.

[10] addressed the much-discussed comparison of performance between hash and sort-merge joins. The authors attempted to utilize SIMD instructions to achieve optimal performance for both join types, and through experimentation, predicted that once larger SIMD width became available, sort-merge joins would outperform hash joins. The authors of [2] followed up this work, and through extensive experimentation and optimization using SIMD instructions in addition to hardware-conscious techniques, concluded that radix hash joins outperformed sort-merge joins. It is worthy of note that the objective of the two hash vs. sort papers was purely to compare the overall optimized performance of the two types of joins; the usage of SIMD instructions was a side-effect of the optimization process.

The star schema is not a new concept, and consequently, numerous studies exist addressing techniques related to it, such as in [13] and [15]. In [1], the authors presented a number of column-store query processing techniques, one of which is the invisible join - a streamlined join algorithm targeted specifically for star schemas on column-stores; our SIMDification work is based around the invisible join.

At the time of writing, work related to the processing of star schema queries and SIMD has only recently been explored in [18], with GPUs. In this work, the authors presented their attempts at accelerating the Star Schema Benchmark queries using the traditional operator-based query processing model. The authors explored performance characteristics under various conditions and on different platforms, and presented an analytical model for predicting the performance of query processing on GPUs.

Chapter 3. Star Schema and Invisible Join

In this Chapter, we describe a typical star schema and its role in data warehousing applications, and we describe the invisible join [1], the column-store algorithm streamlined for processing star schema queries.

3.1 The Star Schema

Commonly used as the schema of the data mart layer of data warehousing environments, star schemas are a simple but efficient style of data schema design. As an extension to the snowflake schema, star schemas typically consist of two main factors: a large fact table consisting of facts, measures, and foreign keys to dimension tables; and a number of dimension tables which categorizes the aforementioned facts and measures in the fact table. Fact tables typically contain information relating to specific events at various granularities, whereas the dimension tables which the foreign keys of the fact table points to contain the descriptive information about the events.

An example of star schema usage is that of a store chain. The sales recording system of a store chain would typically require a table containing the invoices of all sales; the fact table. This table would record the date (foreign key), the store identifier (foreign key), the product identifier (foreign key), and the quantity of the product (value). The dimension tables for such a scenario would be a table for dates (which describes dates - the day, day of week, quarter, etc.), a table for stores (containing information on each of the stores

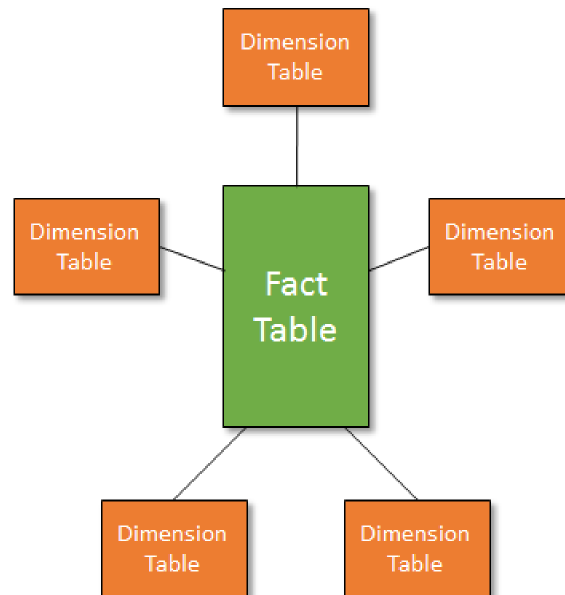


Figure 1: Layout of a typical star schema

– the number, location, etc.), and a table for products (containing details on the products – name, brand, category etc.). With such a schema, it would only require a trivial SQL query to report on, for example, the number of product X sold, grouped by product category and the location, in a given year.

Owing to the simplicity and efficiency, star schemas are widely used, and supported by a wide variety of business intelligence applications in the commercial market.

3.2 The Invisible Join

In [1], as part of the description of columns-store database, the authors presented the so-called Invisible Join algorithm. The invisible join is a join algorithm specialized for star schemas, taking advantage of the column-wise nature of the data structures involved in column-store databases. Where typical DBMS would process star

schema queries like any other query, that is, by parsing the incoming query and building and executing an optimal plan based on the costs involved with the required operators (typically it would involve multiple hash joins between the fact table, and the dimension tables), the invisible join performs the join by rewriting the joins as foreign key predicates. The steps involved in the invisible join algorithm for the query in figure 2 are detailed below.

```
SELECT c_nation, s_nation, d_year,
   SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
   AND lo_suppkey = s_suppkey
   AND lo_orderdate = d_datekey
   AND c_region = 'ASIA'
   AND s_region = 'ASIA'
   AND d_year >= 1992 AND d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year ASC, revenue DESC;
```

Figure 2: A typical star schema query

Step 1: Build the dimension hash tables

The first step involves building a hash table for the dimension table predicates. The goal behind this step is to retrieve a set of dimension record IDs which correspond to dimension table records matching the predicates given in the query.

For this scenario, the predicates for the Customer, Supplier and Date tables are evaluated, and the primary keys of the matching records are inserted into a hash table for each dimension table, as shown in figure 3.

A hash table - where the key is the primary key of the dimension table - is built, which allows for easier probing with fact tables values, covered in the next step.

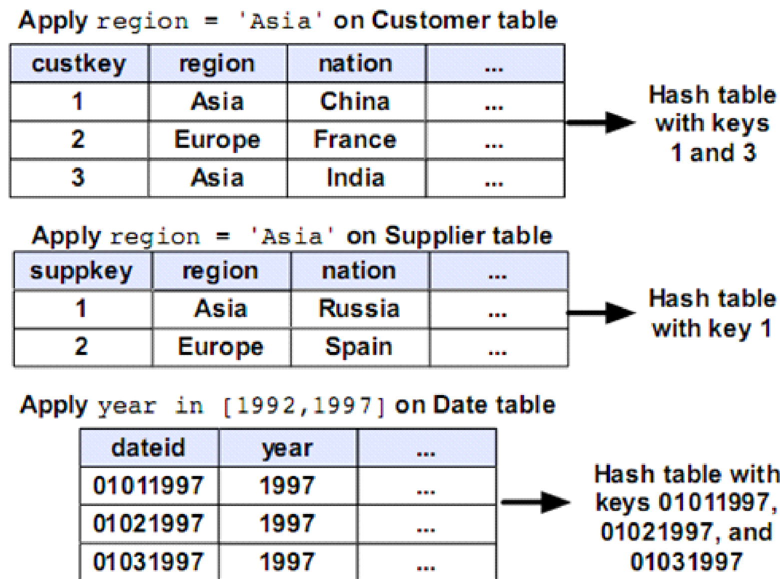


Figure 3 (from [1]): Step 1 of the invisible join

Step 2: Generate filter mask for the fact table

Once the hash tables have been built, the next step of the algorithm is to generate a filtering mask, which represents the records of the fact table which satisfy all the join predicates. For this, the hash tables are probed with the values of each of the foreign key columns in the fact table. Since the hash table keys correspond to the primary key of the corresponding dimension table, a match in the probing indicates that the particular fact table column value satisfies the predicates for that particular dimension table. A vector containing 1s (indicating a match) or 0s (indicating a no-match) is generated as output (figure 4).

Once all the foreign key columns have generated their resulting vectors, they are then merged together using a bitwise AND, to generate the final filtering mask representing the fact table tuples which satisfy all join predicates, to be used in step 3.

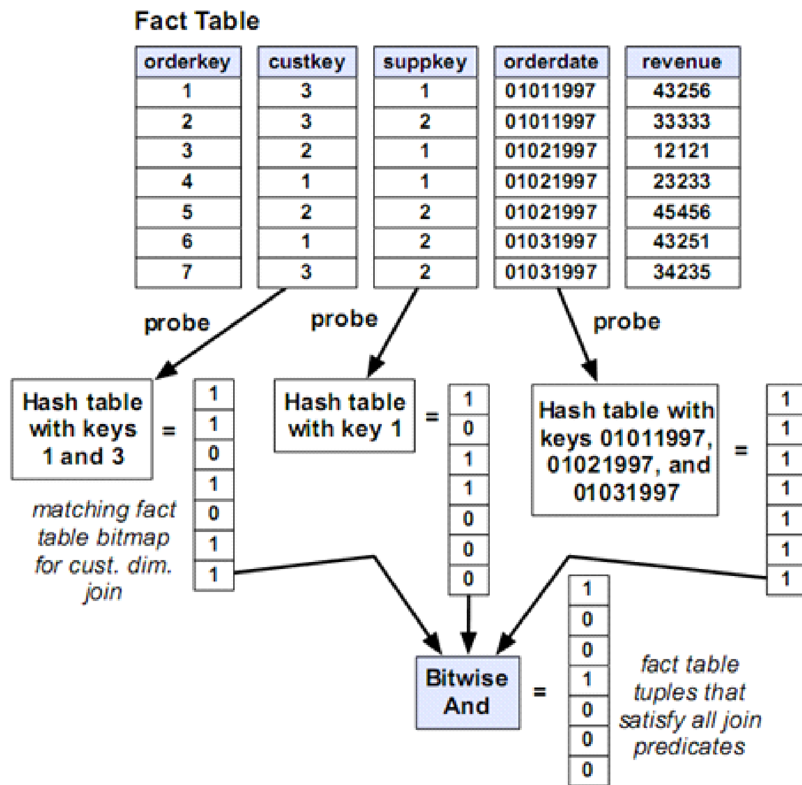


Figure 4 (from [1]): Step 2 of the invisible join

Step 3: Extract the matching records

The final step of the invisible join algorithm involves filtering out the matching tuples from the fact table, and proceeding to materialization.

The steps involved in the final filtering process is trivial – a bitwise AND is performed to retrieve only the matching tuples. The join is complete at this point; materialization can be performed as normal by looking up the dimension table columns using the filtered foreign key columns of the fact table.

The invisible join algorithm is very much a specialized algorithm, exploiting the column-wise nature column-store databases and late materialization techniques. Whilst perhaps unsuited for general purpose query processing, this technique is highly efficient for

star schemas and can offer tremendous performance benefits in that specialized area. In the next Chapter, we detail our steps towards SIMDifying the invisible join algorithm.

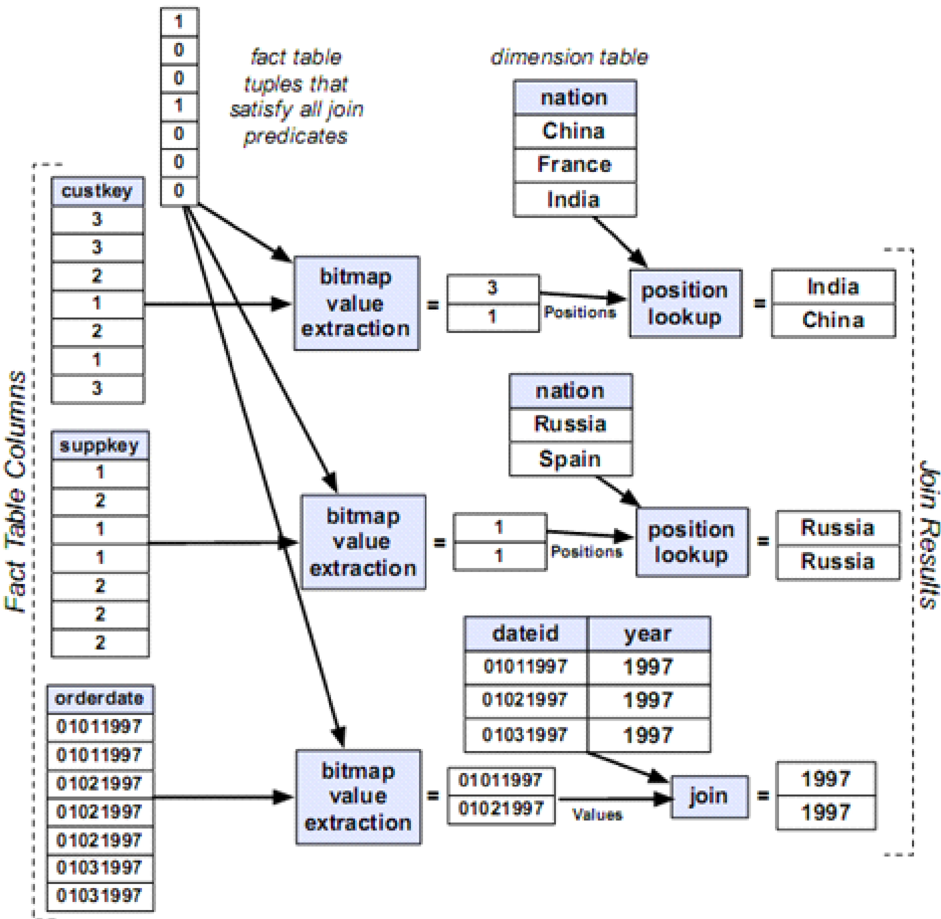


Figure 5 (from [1]): Step 3 of the invisible join

Chapter 4. SIMDification of Invisible Join

In this Chapter, we describe the process of SIMDifying the invisible join algorithm described in Chapter 3. There are two main aspects of the process: extending the invisible join algorithm to remove the hash table operations during predicate evaluation, and the actual SIMDification itself.

4.1 Extending the Invisible Join

Hash table operations are not SIMD friendly. Depending on the algorithm, the hash function itself can be implemented with SIMD instructions; however, since the result of said hash functions typically return random values, that is, values of un-sequential nature, SIMDification is difficult for working with the actual hash table.

As described in the previous Chapter, hash tables are utilized as part of the predicate rewrite stages of the invisible join. In order to facilitate a more efficient SIMDification of the invisible join, we apply a simplified version of the extensions to the invisible join as described in [4], to remove the hash table predicate operations, described below.

The key to the extensions are two assumptions: a) the columns are dictionary encoded, and b) the dimension tables are sorted by their primary keys. One of the main benefits that column-store databases offer is the opportunity to apply dictionary encoding to the columns, thereby enabling huge savings in the storage space requirements; it is therefore reasonable to assume that column data are dictionary encoded. The sorting of the dimension

tables can also be deemed a reasonable assumption: due to the much smaller size of the dimension tables in relation to the fact table, the cost involved in sorting the dimension tables is minimal.

Given the above assumptions, the key aspect of the extension to the invisible join algorithm is working with the encoded values of the foreign key columns of the fact table. The encoded values of the foreign key column are used to look up the dictionary for that column. As the dictionary of the foreign key column would be identical to the dictionary of the primary key column of the dimension table, the encoded value from the foreign key column can be used directly to probe the dimension table dictionary. In addition, since primary keys are unique, the encoded value of the foreign key column can effectively be considered the row ID of the dimension table.

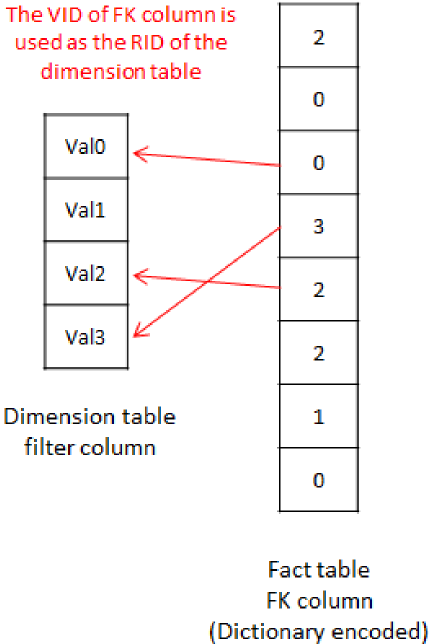


Figure 6: Implicitly performed join

Assuming that an interface exists whereby column value can be retrieved with a row ID given as input, the join between the dimension table and the fact table becomes implicit as part of, for example, retrieving the filtering value for a given row ID from the dimension table, as we iterate through the tuples of the fact table. For our prototype implementation, we have used an array-based data structure for our column-store; performing the join is as simple as accessing an array element at a given index.

4.2 SIMDification of the Invisible Join

We now describe our SIMDification of the invisible join algorithm. Our implementations are based on 128-bit SIMD registers, using Intel's SSE 4.2 intrinsics. As we iterate through the tuples of the fact table, we perform a number of steps designed to SIMDify the access and processing of the required columns, as is visualized in figure 7.

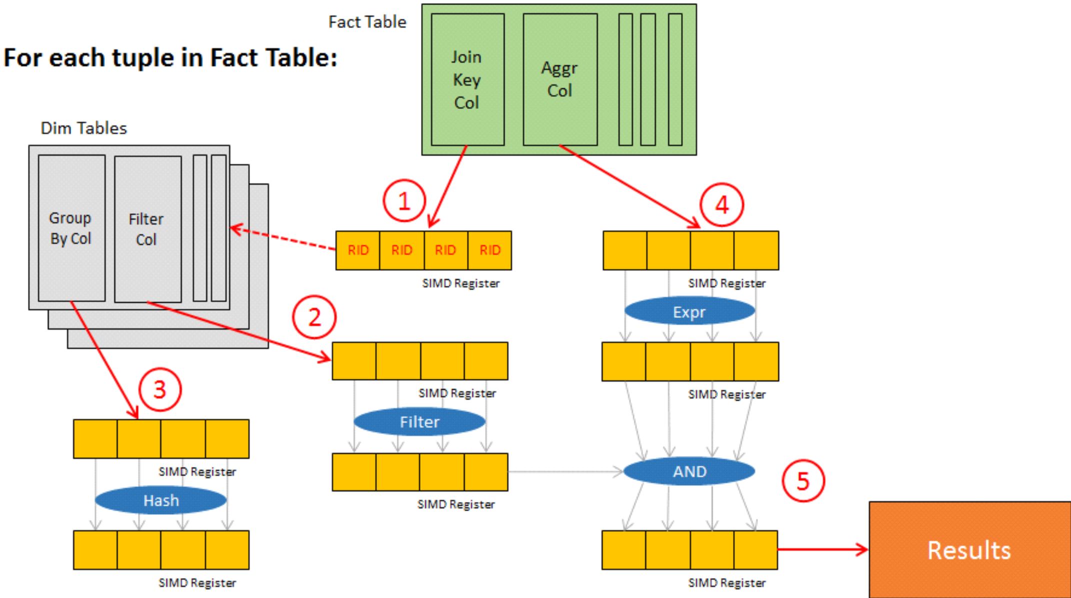


Figure 7: SIMDification of the invisible join

Step 1: Fetch the join key column

The first step of our algorithm involves fetching the join key columns from the fact table. As the columns are dictionary encoded, that is, compressed, we first work to decompress those values.

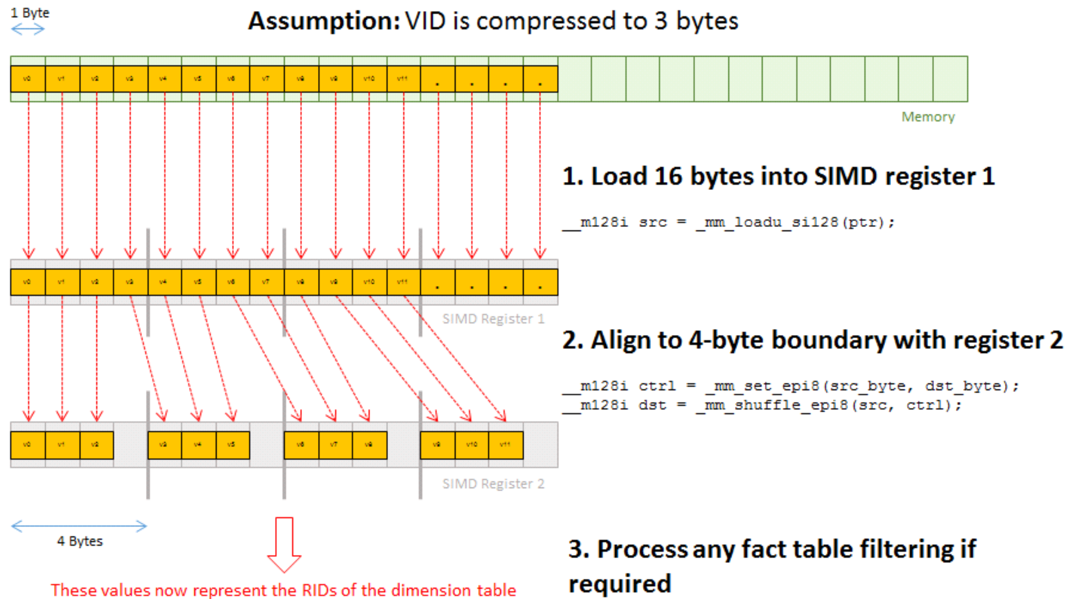


Figure 8: Fetching the join key values

Assuming that the values are 3-byte-compressed, we use SIMD instructions to first load the maximum 16 bytes from memory into a SIMD register. As these 16 bytes contain compressed data, the next required step is to decompress the values, by aligning them to 4-byte boundaries using a second SIMD register. The alignment is performed with the SSE shuffle intrinsic, which allows programmable shuffle control – that is, designating which byte of a particular SIMD element goes into which byte in the destination. The destination SIMD register will then contain 4 elements (12 bytes total) from the original 16 byte foreign key data, decompressed and

ready to be used. Furthermore, as described previously, the value of the elements in the SIMD register represent the row ID of the corresponding dimension table, and will be used to access values from that table.

If the query contains predicates on a column of the fact table, they are processed as part of this step; the details of predicate handling are detailed in step 2.

Step 2: Process the filtering columns

The second step of our algorithm involves processing the predicates on the dimension table columns. Whilst accessing the filter column values is trivial as mentioned above, SIMDifying the access is not. As mentioned previously, SIMD instructions require that the data elements they access be sequential. However, the nature of joins inherently means that access to dimension table values is random; indeed, this is the case every time that columns need to be accessed from any of the dimension tables.

There are two options to address this issue: one is to simply fetch the dimension table values in a serial manner (serialization), and the other is to rely on the GATHER instruction, supported by Intel's AVX2. Whilst the GATHER instruction provides a convenient intrinsic interface from which to build the codebase on, its performance on the Haswell architecture has not been documented by Intel or other authorities at the time of writing. Furthermore, we have determined from experiments that its performance is also affected by a few other factors, including the choice of compilers, and the cache: the effect of cache was demonstrated by the fact that the GATHER instruction outperformed serialization when the data to be fetched both in the cache (a cache hit), and in limited number of cache lines (close data locality). With the uncertainties surrounding

the performance of GATHER on Haswell, we use the serialization method throughout our implementation for simplicity – we leave the performance analysis of the GATHER instruction for future work.

Using the serialization method in conjunction with the row ID fetched from step 1, the values from the dimension table columns are loaded into a SIMD register for evaluation. Predicates in SQL can range widely, from simple numeric comparisons to complex evaluations of string values. However, as column values are dictionary encoded, all operations can be simplified down a combination of integer operations between two operands. For simplicity, we limit our predicate evaluation implementation to comparisons between two SIMD integer values, using corresponding binary comparison intrinsics provided. The results of the binary comparison operation are stored in a SIMD register in the form of bit masks, each element indicating the outcome of the comparison. An element with all its bits set to zero indicates a negative result, whereas an element with all its bits set to one indicates a positive result. The results are retained for filtering out unmatched tuples, as will be described in step 5 below.

As mentioned above, the predicate handling method described here is also employed if predicate processing is required on a column of the fact table.

Step 3: Fetch the group by columns

As mentioned in step 2, accessing any data from the dimension tables require random access to the memory locations, due to the nature of joined data. This problem applies to the fetching of the group by columns, as star schema queries most often perform group by operations on columns from the dimension tables. Again, for simplicity purposes, we use the aforementioned serialization method.

Once all the group by columns have been fetched and loaded into SIMD registers, we use the values to produce a hash key: each of the group by values are put through a combination of SIMD arithmetic operators together to produce a unique key. This in turn is then used to index the result table (a pre-allocated array in our implementation) for the actual aggregation, described in step 4.

Step 4: Fetch the aggregation columns

As the values to be aggregated are stored in the fact table for star schemas, the fetching of their values to be loaded onto SIMD registers is trivial, as is the evaluation of the aggregate expressions using the corresponding arithmetic SIMD intrinsics.

Step 5: Apply the bit mask from steps 1 and 2

The final step of our SIMDification of the invisible join algorithm is to filter out the tuples which do not satisfy all the predicates, by combining the bit mask results from steps 1 and 2, with the aggregate values from step 4. This is achieved using the SIMD equivalent of the binary AND bitwise operation, with the first operand being the aggregates from step 4, and the second operand being the bit mask results from step 1 and 2. For aggregates using floating point data types, we emulate the AND operation by first shifting the bit mask results to the right by 31 bits in order to convert the elements of the mask into either a numeric 0 or 1; we then perform a SIMD multiplication of these values with the floating point aggregate values. The result of these operations is the aggregate value itself if all predicates have been satisfied, or zero if one or more of the predicates were not satisfied. These results are then incremented into the result table using the hash keys from step 3.

The incrementation process is the opposite of the random

access problem with dimension tables, in that instead of reading non-sequential values, we now need to write to non-sequential memory locations. Again, two solutions exist to address this problem: to write the values in a serialized manner as before, or to use the SCATTER instruction, provided by Intel's AVX512. As of writing, AVX512 has not yet had an official release from Intel, and for simplicity purposes, we implement the serialization option in this paper.

Although the SIMDification steps described above have focused on SIMD instructions for CPUs, it can very easily be adapted for other hardware platforms supporting the SIMD paradigm, such as GPUs; with threads executing on the available cores, this would result in SIMD widths numbering in the thousands.

Chapter 5. Experimental Results

5.1 Experimental Setup

For evaluating the performance of our implementation, we make use of the Star Schema Benchmark (SSB) toolset [14]. Derived from the TPC-H benchmarking tool, the SSB is a domain-specific benchmarking system, designed specifically to support classical data warehousing applications. The SSB converts the TPC-H schema into a star schema (by e.g. combining the LINEITEM and ORDERS tables into the LINEORDER table), and its query set departs from TPC-H by attempting to provide the Functional Coverage and Selectivity Coverage [14].

Our experiments are conducted on the Intel i3 4160 chip, with 4 GB of total RAM available. The processor provides support for Intel's AVX up to version 2; our implementation, however, is designed for SSE 4.2, i.e. 128-bit SIMD width, or four 32-bit data elements processed in parallel. Our implementation was coded with C++ using SSE Intrinsics. Our server runs Ubuntu Server 14.04.1 LTS, and we compiled our code using GCC version 4.8.2.

Our experiments were designed to demonstrate the direct effects of our SIMDification efforts. We achieve this by comparing the performance of our SIMDified algorithm with the scalar equivalent - the only difference between the two algorithms is the SIMDification steps described in Chapter 4. Thus, for accurate measurements, we compiled the scalar version of our algorithm with flags specified to exclude SSE optimization. Both binaries were compiled with optimization level 3.

We first present the overall performance results from start to finish, followed by the breakdown of the results to the main components of the invisible join algorithm.

5.2 Overall Results

We present the overall results of our SIMDified invisible join algorithm. Our results exclude pre-loading, pre-processing and all other unrelated functions, and instead measure pure operator performance.

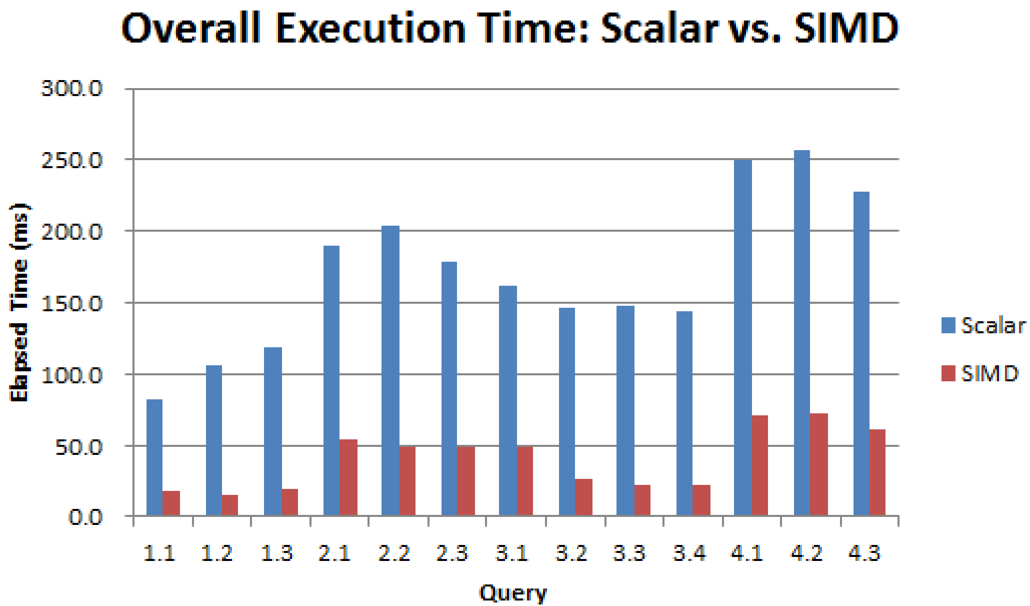


Figure 9: Overall execution time

As shown in figure 9, our SIMDification of the invisible join algorithm achieved 4.8x speedup on average over its scalar equivalent, exceeding the 4x potential speedup offered by 128-bit SIMD operations. The overall speedup is highly affected by the proportion of overall time taken up by the individual stages of our algorithm.

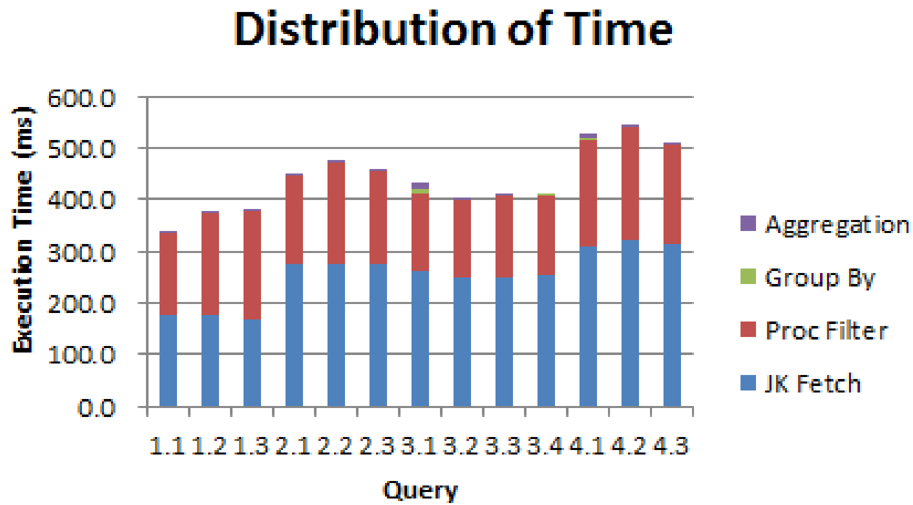


Figure 10: The distribution of overall execution time

From figure 10, we observe that the first two stages of our invisible join algorithm – namely the join key fetch and the filter processing stages – accounts for the overwhelming majority of the total execution time. The group by and aggregation stages of the algorithm are highly dependent on the selectivity of the incoming queries, and account for only a very small proportion of the overall running time. Consequently, performance speedups achieved by the first two stages largely determine the overall performance speedup; we show that this is indeed the case in the following breakdown of the overall results.

5.3 Breakdown of Results

We now break down the overall performance results to the four main stages of our SIMDified invisible join algorithm: join key fetch, filter processing, group by, and aggregation. It is important to note that the code used to measure the individual stages of the algorithm

affects the performance – relative to the overall start-to-finish performance – and the figures presented below do not reflect the actual run time of the said stages.

Join Key Fetch

The performance of our SIMDified join key fetch achieved 4.9x speedup on average over its scalar counterpart, exceeding the logical potential speedup of 4x for 128-bit SIMD; in fact, a peak speedup of 6.4x was observed for Q3.4, far exceeding the potential, as can be seen in figure 11.

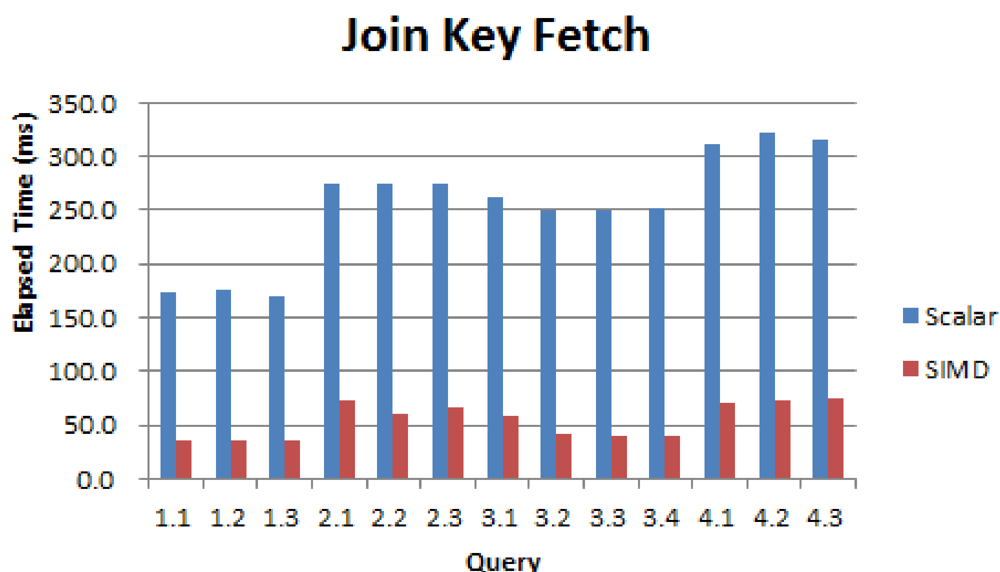


Figure 11: Performance of join key fetch

The extra performance can be attributed to the difference in the quality of the compiled code between the two algorithms. The join key fetch stage is, in essence, simply the copying of a particular section of memory from address A to address B. For the scalar version of our algorithm, the most efficient method of achieving this was to use the memcpy() function provided by C++, which is a

function call and incurs all the costs associated with such an operation. For our SIMDified version, this involved the loading of the data to the SIMD register, followed by a shuffle operation for 4-byte alignment. Due to the use of intrinsics, our SIMD code translated directly into their SIMD assembly counterparts, and avoided the expensive function call overheads incurred by the scalar memcpy() implementation, accounting for speedup in excess of the logical potential.

Process Filter

The filter processing stage of our SIMDified invisible join algorithm achieved 3.6x speedup on average over its scalar equivalent, as is shown in figure 12.

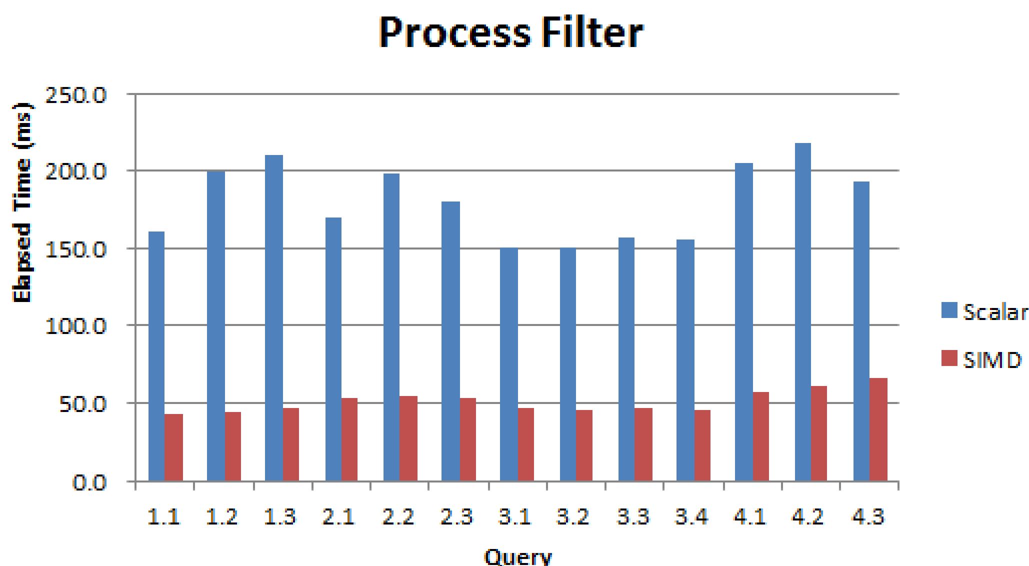


Figure 12: Performance of filter processing

The potential 4x speedup was not achieved for this stage of our algorithm, primarily due to the dimension table accesses required. As described in the previous Chapter, reading values from the dimension table in our algorithm requires random memory access,

which remains serialized in the SIMDified version. While the evaluation of the binary comparison operators can easily be SIMDified, the random memory access involved in reading the filtering column values from the dimension tables affects our algorithm from achieving the full 4x speedup potential. The GATHER instruction in AVX2, along with Intel's Xeon Phi architecture which fully supports the instruction set with hardware, is hypothesized to address the effect of random memory access. We plan to further analyze the effect of that combination in future work.

Group By

The selectivity of the SSB query set was very high – that is, the percentage of the fact table tuples which satisfied all the query predicates was extremely small; only 0.7% of the fact table tuples satisfied all the predicates, with the maximum being 4.1% for Q3.1. As the group by stage of the algorithm is performed only after all predicates have been satisfied for a given tuple, the high selectivity had the effect of reducing the run time of the group by stage to only a few clock cycles, making effective comparisons difficult. For the purpose of measuring the effect of SIMD on our algorithm, we adjusted the selectivity to 0%, i.e. the group by stage was processed for all the tuples in the fact table, regardless of predicates. The results are shown in figure 13.

We observed our algorithm achieving 3.5x speedup on average over the scalar version of the algorithm. Queries 1.1, 1.2, and 1.3 do not contain group by statements, and were excluded from the results. Again, the potential 4x speedup was not achieved for this stage of our algorithm, due to the aforementioned dimension table accesses required.

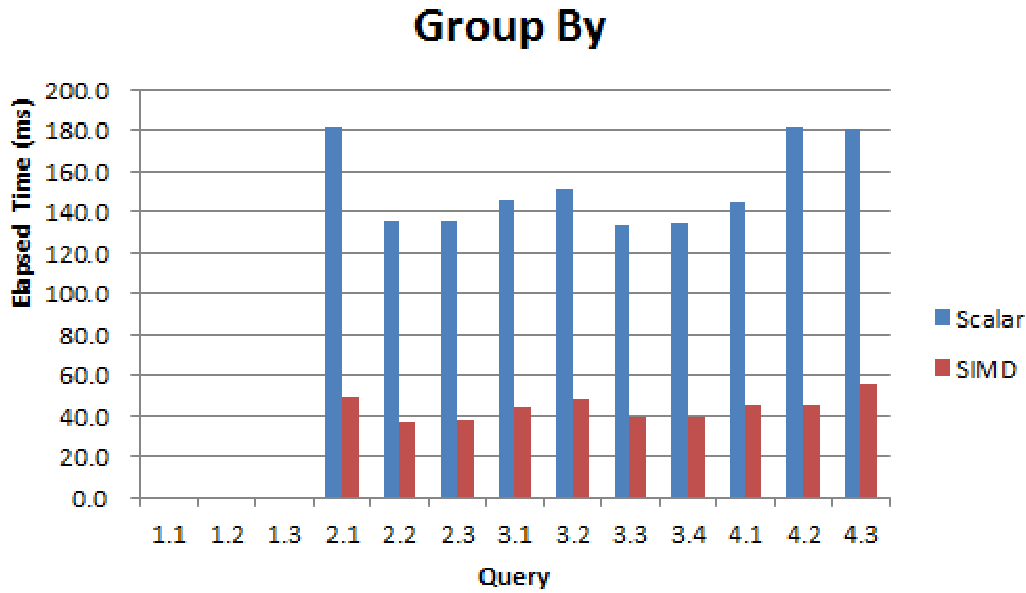


Figure 13: Performance of group by

Aggregation

As with the group by stage, the aggregation stage is also affected by the aforementioned high selectivity of the SSB queries, resulting in performance which could not be reliably measured. Again, we adjusted the selectivity to 0% to demonstrate the effectiveness of our SIMDification on the aggregation stage.

Unlike the previous stages of our algorithm, the SIMDification of the aggregation stage only achieved 1.8x speedup on average over its scalar counterpart, far less than the potential 4x speedup. As with the group by stage, the evaluation of the aggregation expressions consists of arithmetic operations, which are readily SIMDified using the appropriate intrinsics. Similarly, applying the bit mask to filter out the fact table tuples which do not satisfy all the predicates is also translated directly into SIMD intrinsics. However, as was described in Chapter 4, the aggregation stage requires write

operations to memory locations corresponding to the hash keys generated by the group by stage, i.e. random write access. While our implementation serialized this section of the algorithm, the effect of the serialized write is much larger than that of the serialized read, due to the complexities in serialization. Code-wise, the serialization of the random read access is implemented simply as four separate read operations. The serialization of the random write access is restricted by the limitations of the SSE intrinsics library: for floating point data types (all the SSB aggregation columns are floating point data types), the most efficient method of accessing the individual elements of a given SIMD register is to write the contents back into memory, and then access them individually. Given this restriction, the serialized random writes of our aggregation stage suffers much more overhead than that of the serialized read, resulting in a heavier penalty on the performance.

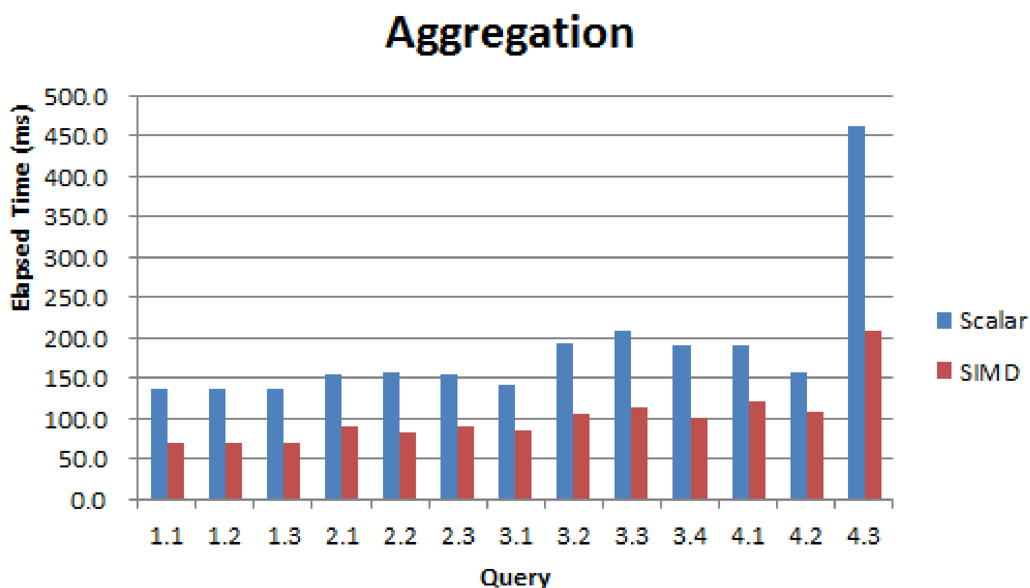


Figure 14: Performance of aggregation

Again, whilst the group by and aggregation stages of our algorithm do show performance improvements with SIMDification, they only account for a very small percentage of the overall running time, due to the high selectivity of the incoming queries. The improvements achieved in the join key fetch and filter processing stages of our algorithm have the biggest impact to the overall performance.

Chapter 6. Conclusion and Future Work

Hardware is constantly evolving, and at a rapid pace, to meet the ever-increasing demands of database systems – a trend which will only increase with the prevalence of the so-called Big Data. And although the development of bigger and better hardware is beneficial, attention must be given to making use of the hardware which is available today. Indeed, SIMD processing on the CPU is a prime example of this: although widely available on a large number of platforms and with a history of continued development, focus on making explicit use of SIMD instructions for database operations has been limited. With further SIMD hardware development planned by vendors, focusing on optimizing database systems for SIMD is essential.

Though some work has covered the usage of SIMD instructions for database operations, the large majority of the work was based around the iterator model. In order to showcase the full potential of the SIMD hardware, we must move outside the bounds of the traditional, general purpose models used today, and focus on specific scenarios which offer more opportunities for optimization. In this paper, we have described the SIMDification process of one such scenario that is widely used in the data warehousing field, the processing of star schema queries. Through the SIMDification of the invisible join algorithm, we have demonstrated that can be achieved which match, and sometimes exceed, the logical hardware potential. Although the focus on specific scenarios sacrifices flexibility for performance, the techniques described can serve as a base for further

optimization toward a more generalized framework.

As future work, we plan to perform an in-depth analysis of the performance speedups we have achieved with our SIMDification. We also plan to explore the optimization opportunities available with newer hardware, namely Intel's AVX512 and the full set of instructions that it supports.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 967-980, New York, NY, USA 2008. ACM.

[2] C. Balkesen, G. Alonso, J. Teubner, and M. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. Proceedings of the VLDB Endowment, 7(10): 85-96, 2014.

[3] W. Fang, B. He, and Q. Luo. Database Compression on Graphics Processors. Proceedings of the VLDB Endowment, 3(1-2): 670-680, 2010.

[4] F. Färber, S. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. ACM SIGMOD Record, 40(4): 45-51, 2011.

[5] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient Gather and Scatter Operations on Graphics Processors. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07, article no. 46, New York, NY, USA 2007. ACM.

[6] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. Sander. Relational Query Co-Processing on Graphics Processors. ACM Transactions on Database Systems (TODS), 34(4): 21, 2009.

[7] B. He, and J. X. Yu. High-Throughput Transaction Executions on Graphics Processors. Proceedings of the VLDB Endowment, 4(5): 314-325, 2011.

[8] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Joins on Graphics Processors. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 511-524, New York, NY, USA 2008. ACM.

[9] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. Proceedings of the VLDB Endowment, 6(9): 709-720, 2013.

[10] C. Kim, T. Kaldewey, V. Lee, E. Sedlar, A. Nguyen, N. Satish, J. Chhugani, A. Di blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. Proceedings of the VLDB Endowment, 2(2): 1378-1389, 2009.

[11] T. Mühlbauer, A. Reiser, W. Rödiger, A. Kemper, R. Seilbeck, and T. Neumann. Instant Loading for Main Memory Databases. Proceedings of the VLDB Endowment, 6(14): 1702-1713, 2013.

[12] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. Proceedings of the VLDB Endowment, 4(9): 539-550, 2011.

[13] P. O'Neil, G. Graefe. Multi-table joins Through Bitmapped Join Indices. ACM SIGMOD Record, 24(3): 8-11, 1995.

[14] P. O'Neil, B. O'Neil, and X. Chen. Star Schema Benchmark revision 3, 2009. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[15] A. Weininger. Efficient Execution of Joins in a Star Schema. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, SIGMOD '02, pages

542-545, New York, NY, USA 2002. ACM.

[16] T. Willhalm, I. Oukid, F. Faerber, and I. Müller. Vectorizing Database Column Scans with Complex Predicates. In Proceedings of the 4th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS '13, pages 1-12, Riva del Garda, Italy 2014.

[17] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. Proceedings of the VLDB Endowment, 2(1): 385-394, 2009.

[18] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. Proceedings of the VLDB Endowment, 6(10): 817-828, 2013.

[19] J. Zhou, K. Ross. Implementing Database Operations Using SIMD Instructions. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, SIGMOD '02, pages 145-156, New York, NY, USA 2002. ACM.

국문 초록

스타-스키마 쿼리 처리의 SIMD 최적화

서울대학교 대학원
전기 컴퓨터 공학부
송 상 협

대부분의 최신 CPU들은 SIMD (Single Instruction, Multiple Data) 레지스터와 명령어를 갖추어서, 하나의 명령어로 여러 데이터를 동시에 처리하는 것이 가능하다. 높은 활용성과 컴파일러의 지원, 하드웨어를 변경할 필요가 없는 점, 그리고 성능 향상의 가능성이 크다는 점에서, SIMD 명령어를 데이터베이스 질의 처리에 활용하는 것은 몇몇 연구의 주제로 다뤄지기도 하였다.

스타 스키마는 데이터마트 모델링에서 많이 사용되는 방법이고, 빅데이터 분석의 필요성이 급격하게 늘어나면서, 온라인 분석 처리 성능 최적화의 주요 용례로 떠오르고 있다. GPGPU 분야에서 스타 스키마 질의 처리를 SIMD 최적화하는 연구가 이미 진행된 바가 있지만, GPGPU와 SIMD의 작동방식의 유사성에도 불구하고 CPU에 장착된 SIMD 명령어를 활용하는 연구는 아직 진행된 적이 없다.

본 논문에서 우리는 스타 스키마 질의 처리를 SIMD 최적화함으로써, 4배 이상의 속도향상을 얻을 수 있음을 확인하였다. 전통적인 연산자-기반 질의 처리 방식 대신, 스타 스키마에 특화된 소위 invisible join알고리즘에 주목한다. Invisible join 알고리즘의 각 단계와 그 수행 절차 및 SIMD를 고려한 최적화 과정을 설명한 후, 그 결과물이 대응되

는 기존의 스칼라 구현과 비교하여 전체적으로는 최대 4.8배, 특정 연산자에서는 최대 6.4배의 속도 향상을 얻을 수 있음을 보인다.

주요어 : SIMD 질의 처리, 스타 스키마, In-memory column store
학 번 : 2012-23953