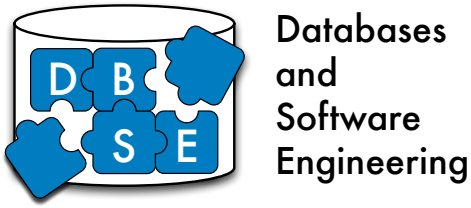
University of Magdeburg

School of Computer Science



Master's Thesis

Analysis of Hashing Techniques for Efficient Group-Based Aggregation

Author:

Balasubramanian Gurumurthy

September 19, 2017

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. David Broneske

Department of Computer Science

Prof. Dr.-Ing. Thilo Pionteck

Department of Electrical Engineering and Information Technology

Gurumurthy, Balasubramanian:
Analysis of Hashing Techniques for Efficient Group-Based Aggregation
Master's Thesis, University of Magdeburg, 2017.

Abstract

Hash-based aggregation strategy is used for performing grouped aggregation queries. However, this strategy has high processing time for result computation. In this work, we explore the use of code optimization strategies to reduce the execution time. we introduce a way to perform insertion along with probing in the underlying hashing techniques. We also exploit the SIMD capability of modern hardwares to speed up the hashing technique operations. In our work, we evaluate the performance of scalar and vectorized execution of hashing-based aggregation with cuckoo hashing and linear probing as underlying hashing techniques. We use different dataset distributions for our evaluation. We also evaluate the impact of group size on runtime for computing grouped aggregation. Finally, we provide our inferences on the evaluation results obtained.

Acknowledgement

First, I would like to thank Prof. Gunter Saake for providing me this opportunity of writing this thesis under his supervision. I extend my deepest gratitude to my advisor David Broneske, whose door were always open whenever I ran into trouble. His guidances and expertise steered me in the right direction. I thank him once again for his patience and constant availability during the whole tenure of this thesis.

Further, I would like to thank my parents, my brother, my family for their continuous support and trust on me during my year of studies.

I would also like to take this opportunity to thank my friends and colleagues for their continuous encouragement. Special mention to my TCS and Deutsche Telekom teammates, who were ready to help me whenever I needed help.

Finally, special thanks to all my Magdeburg friends who were always there for me and provided me constant care. I would also like to thank my roommates: Sreeram Jagannathan and Nivin Joseph for their tolerance of my shenanigans. Finally, I thank everyone who guided me in writing this paper. This work would not have been complete without you all . Thank you.

To my grandparents, for providing a colorful childhood

Contents

List of Figures

List of Algorithms

List of Code Listings

1. Introduction

Faster query processing is an important part of database management system (DBMS). However, faster execution of the query depends on the type of query being executed and various other factors [ADHW99]. For instance, grouped aggregation queries are computed using a two-pass processing [GMUW00]. This process consumes a lot of CPU time. However, this time delay can be improved using code optimization strategies. It is shown that, code optimization strategies can be used to speed-up selection query processing [BMS17]. Similar strategies can be used optimize the group-aggregate queries. In this thesis, we explore the different code optimization strategies that can be applied to optimize execution of grouped aggregation queries.

Grouped aggregation in query processing

In general, a given query is passed through various translation and optimization steps and then it is converted into executable code. These operations are fragmented into pipelines with multiple operations within each pipeline for execution. These pipelines are executed using a stream-based processing model [DK97]. This model executes the operations inside a single pipeline in one step forwarding intermediate results from the operation to another without storing them in memory. This avoids the memory access for intermediate operations and the data flow between operators reduces processing time. However, the flow in pipeline is obstructed by functions known as pipeline breakers [Neu11]. These functions materialize their results only after all intermediate data are collected. This obstructs the data flow in the pipeline. Typical example for pipeline breaker is aggregate function where the resultant aggregate is not forwarded until all the intermediate results are processed. This stops the flow of data in pipeline thereby creates stalls [1]. These stalls due to aggregate functions decreases the efficiency of query processing and increases execution time [Neu11].

[1] Stall - Delay in the data flow

Resolving stall of data while aggregation improves the efficiency of grouped aggregation queries. Furthermore, the number of stalls in process pipeline increases with the amount aggregate functions used in a query. Hence, the complexity of a query increases with number of aggregates in it. Moreover, aggregate functions are often coupled with grouping of data when using a `GROUP BY` clause in a query. This subjects the intermediate results in the data flow to be grouped based on the given criteria, and aggregates are found for these groups. Grouping creates further stalls in the data flow. Thus, grouped aggregation queries are delayed due to multiple stalls created due to grouping and aggregation of data. This increases the overall query processing overhead and reduces efficiency. These stalls can be reduced by optimizing the executable code used to materialize the results. Thus, optimizing the executable code impact the efficiency of performing grouped aggregation queries.

Hash limitation

From the technique given in [SZ96], aggregation is performed along with insertion of value into their respective group. Since, aggregation is done along with grouping the stall due to aggregation is replaced by stall due to grouping. Grouping in grouped aggregation queries is performed using hashing techniques. A hashing technique has one or more hash table with a specific number of buckets. Each bucket may have multiple slots in which a value is stored. A value to be inserted is hashed to get a key. This key points to the bucket in which the value hs to be stored. There can be multiple slots within the bucket. In some cases two data items are given the same slot in hash table. This is called collision. These collisions are responsible for stalls due to grouping. Collision forces the hashing techniques to find an alternative position to store the collided value before processing the next data item. This is done by a collision resolution function. This function takes considerable time in finding the right position to store value in the hash table [MC86] stopping the data flow until the current data item is stored in its slot.

Moreover, collisions also impact probing for a search value in the hash tables. In an ideal hashing technique, all the values are stored in their slot and are directly accessed. But, collisions displace the value to another slot. This leads to searching of value in the hash table increasing processing time. Hence, faster execution of grouped aggregation queries depends on efficiency of underlying hashing techniques and their collision resolution mechanism. Thus, the stalls created by grouping can be reduced by optimizing execution of the underlying hashing technique.

Collision resolution

Hashing techniques are differentiated based on the collision resolution mechanism used. In chained hashing technique, overflow values[2] are stored in a linked list in the same bucket. In the worst case, probing may lead to linear search of all stored values in single bucket increasing processing time. Open addressing techniques are another way to

[2]overflow value - value whose slot is already occupied

resolve collisions. In these methods the overflow values are inserted in alternative locations of the same hash table. These techniques promise constant size of values to search as the hash table size is constant. Linear probing [Lit80] and cuckoo hashing [Dil14] are part of the open addressing techniques. It is shown that these methods are efficient in insertion and probing of write once read many(WORM) workloads. As analytical data are mostly read only [RAD15], we examine in this thesis the impact of using code optimization strategies to these two hashing techniques for processing grouped aggregation queries: linear probing and cuckoo hashing.

Linear probing technique stores values in a list. If the slot pointed to by the key of a value is occupied, the value is stored in the next free location available in the same hash table. Now, probe for this value is then done by performing linear search from the slot previously pointed to by the key until the search value is found. In case an empty slot is encountered during linear search, the value is not found in the table. Thus, the stall in this technique is due to performing linear search while searching an empty slot during insertion and while probing the search value. In case of cuckoo hashing, the values are stored in multiple hash tables. If the slot pointed to by the key of insert value is occupied in the first table, the value already present in the slot is swapped with the insert value. This swapped value is then hashed to get the key that points to a slot in the next table. This guarantees constant probing time for cuckoo hashing. However, probing of a search value is done in sequence for all hash table slots. In worst case, the slots are probed until the last hash table. Hence, this method also has an overhead of searching linearly for the search value. Thus, these methods resolve collision by adding overhead for probing a value [Ros07]. We investigate in the thesis, the use of code optimizations to improve these probing limitations available in the hashing techniques. We also argue, that improving execution of these techniques can speed-up the process of grouping, further leading to faster execution of the grouped aggregation query itself.

Contribution of the thesis

- In the thesis, we investigate the code optimization Single Instruction Multiple Data (SIMD) acceleration on the above mentioned hashing techniques. SIMD enables data parallelism by exploiting machine hardware. This feature can be used to perform probing in cuckoo hashing and linear probing faster [JA17]. In [Ros07], the author presents the way to use SIMD in cuckoo hashing to perform probing. Similarly, SIMD can be adapted to execute grouping along with aggregation in the technique. Similar method can be used to perform aggregation using SIMD on linear probing as well.

- In the thesis, we also present a way to reuse probing codes for insertion of a value in the hashing techniques. This reduces the two step process of hashing insertion and probing to one linear process. We introduce the technique called insertion while probing where the insertion in the hashing technique is combined with probing. During insertion of a value, the value is hashed and the slot pointed by the key is compared for empty space. If the slot is empty, the value is inserted.

Similarly while probing, the search value is hashed and the key pointed slot is probed for the search value. Thus in both the insertion and probing functions comparison of values is performed. Hence, these two can be combined into a single process. Insertion is performed if the search value is not available in the hash table while probing for the value. We detail the code strategy for the above mentioned hash techniques in this thesis.

- As mentioned earlier, aggregation can be performed along with insertion [SZ96]. There are few aggregate functions such as count, min, max etc., that can be computed on the-fly [GMUW00]. Each value inside the hash tables are stored as value-payload pairs with the group identifier as value and its corresponding aggregate as payload. This also removes the overhead of storing multiple copies of same value in the bucket. We introduce in the thesis, the way to incorporate direct aggregation along with probing by insertion technique for both the hashing techniques.

Overall, in this thesis we examine the efficiency and reduction of stall in pipeline by applying code optimization techniques for hashing techniques. We investigate usage of SIMD to reduce the data stall created while probing in hashing mechanisms. We also explore the possibility of using code optimization strategies such as, direct aggregation and insertion using probing to speed-up grouped aggregation. Finally, we compare and present results on efficiency of these optimization strategies applied on both hashing techniques.

Structure of the Thesis

In Chapter 2, we discuss the basics of query compilation and later explains the usage of hashing techniques to perform DBMS operations. We also detail the functional details of the hashing techniques used. The chapter also provides the necessary SIMD functions used. In Chapter 3, we explain the conceptual details of adapting the hashing techniques for aggregate-grouping. We also detail on the use of SIMD for accelerating the execution. We present the implementation details for incorporating SIMD codes into the hashing techniques in Chapter 4. The comparison results of execution of scalar and SIMD accelerated hashing are presented in Chapter 5. We provide details of related work in Chapter 6 and our conclusion in Chapter 7. Finally, we detail the future work that can be done in Chapter 8.

2. Background

The query given by an user is processed by various components to finally produce the results. These components are broadly divided into two systems known as the query compiler and the query executor [GMUW00]. We discuss in this chapter, the of processing group-aggregation queries in these two systems. We also provide an overview of the hashing techniques used to perform grouping and the ways to adapt them for aggregation. The fundamental concepts discussed in this chapter are referred from [DK97].

The chapter is divided as follows. We detail the two systems mentioned above in Section 2.1 and Section 2.2. These strategies for executing query are introduced in Section 2.3. In Section 2.4, we explore the group-aggregation query processing in general. The hashing techniques used for grouping data is discussed in Section 2.5. Finally, in Section 2.6, we explain the necessary SIMD functions used to optimize the hashing techniques.

2.1 Query Compiler

The query compiler consumes a high-level user query and provides the executable codes necessary to perform the query. It has two main components, the parser and the optimizer. The parser converts the given query into an internal format and the optimizer produces an efficient execution plan for the given query.

Figure 2.1 shows the general processing steps of the query compiler. The detailed description of the different components are given below,

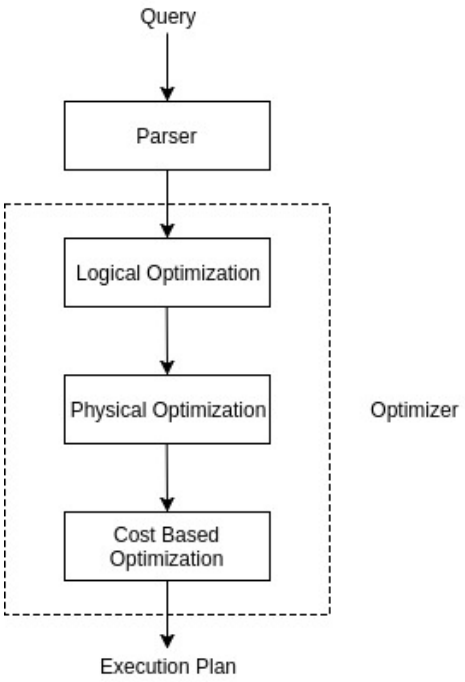


Figure 2.1: Query compiler

2.1.1 Parser

The parser reads the query and checks for correctness of syntax and also for the equality of relation and column names. This evaluated query is then converted into an internal representation called the parser tree [GMUW00]. This tree is a combination of relational algebra operators. The details of these operators are discussed below.

2.1.1.1 Relational Algebra

Relational algebra is used to denote the various operators used in a query. The given query is broken into simple query bocks and converted into a relational algebra tree. This simple query block consists of SELECT-FROM-WHERE (SFW) statement, with a GROUP BY clause if available [DK97]. Queries to a DBMS have some arbitrary level of nesting. These queries are split into simpler blocks from the outer nested to the inner and are recursively converted into their corresponding tree structure. The basic relational algebra operators matching operations in SFW block are [Cod70],

- Projection
- Selection
- Join
- Union and Difference
- Aggregate and Group By

Projection - π

The projection operator is used to return attributes values defined in a query. The operator is noted as,$\pi_{Attribute1,Attribute2,Attribute3...}(Relation)$. Attributes are the column name given in the query to retrieve the values from tables.

Consider, the relations STUDENT and FACULTY storing student and faculty information respectively. The structure of the relations are given as,

$$STUDENT(\underline{ID}, NAME, AGE, GRADE\_POINT, FACULT\_ID \rightarrow FAULTY\_ID)$$

$$FACULTY(\underline{FACULTY\_ID}, FACULTY\_NAME)$$

Consider, the query to select the name of all the students. In SQL, it is written as

SELECT NAME
FROM STUDENT;

is translated as,

$$\pi_{NAME}(STUDENT)$$

Selection - σ

Selection operator is used to select the tuples that satisfy a particular predicate. The operator is noted as, $\sigma_{Predicate}(Relation)$. `Predicate` is the conditional logic given in the query to retrieve the desired tuples and the `Relation` is the table mentioned in the query.

For example, the query to select all the students having grade point below 2.0 is written as,

SELECT *
FROM STUDENT
WHERE GRADE_POINT $<$ 2.0;

and is translated in relational algebra as,

$$\sigma_{GRADE\_POINT<2.0}(STUDENT)$$

Join - $\bowtie$

Join is used to combine two relations based on a common attribute from both the relations. This operator is denoted as, Table1 $\bowtie$ Table2.

Considering the query to get name of a student along with the faculty name of the student. Then the query is,

SELECT NAME, FACULTY_NAME
FROM STUDENT
NATURAL JOIN FACULTY;

This is translated as,

$$STUDENT \bowtie DEPARTMENT$$

Union and Difference - $\cup$ and $-$

Union and difference are binary operators. They are used to combine two relational predicates. These merge results of two queries into single result. These operators are written as, Table1 $\cup$ or $-$ Table2.

For example, the query to list out all the student name along with all the faculty name is,

(**SELECT** NAME
FROM STUDENT)
UNION
(**SELECT** FACULTY_NAME
FROM FACULTY)

and is translated as,

$$\pi_{NAME}(STUDENT) \cup \pi_{FACULTY\_NAME}(FACULTY)$$

Group By and Aggregation - γ

Aggregation functions occur in `SELECT` clause and are usually coupled with a `GROUP BY` clause. These two operations are interpreted and implemented together. These two are represented together as, $\gamma_{Col;Agg}(Relation)$ where `Col` is the column by which the values are grouped and `Agg` is the aggregation function to be performed.

Consider the query to retrieve the average grade point of all students in each department,
SELECT FACULTY_ID, AVG(GRADE_POINT)
FROM STUDENT
GROUP BY FACULTY_ID;

This is translated as,

$$\gamma_{DEPT\_ID;AVG(GRADE\_POINT)}(STUDENT)$$

2.1.1.2 Relational Algebra Tree

After each operation in the query is translated into their respective relational algebra operators, these are combined to form the relational algebra tree. The leaf nodes of the tree comprises of the relation names and the internal nodes have the algebra operations. Branches are added to the tree in case of binary operators.

For example, consider the selection of all the student with their respective department. The query is written as,

SELECT S.NAME, F.FACULTY_NAME
FROM STUDENT
NATURAL JOIN FACULTY;

The query has projection along with a join operation. This is represented in tree structure as shown in the Figure 2.2,

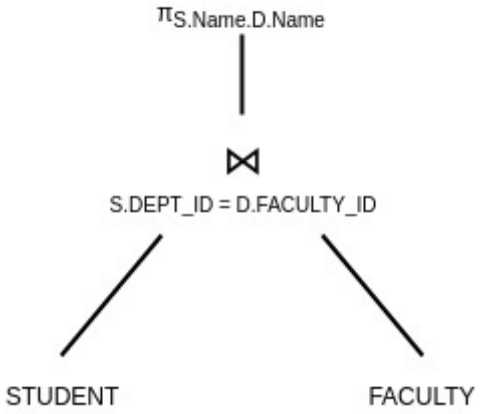


Figure 2.2: Relational algebra tree

In figure, leaf nodes are different relations used in the query- STUDENT and FACULTY. All the internal nodes represent the operations to be performed- join and projection.

2.1.2 Optimizer

The query given by user is optimized for faster execution using optimizer. There are various optimization steps carried out in this component. First, the given structure is simplified into an equivalent one using set of rules(i.e., logical optimization). Each node in the reduced tree is then replaced with their corresponding physical algorithm. For example, join operation is replaced with the hash join or the nested loop join algorithm. These optimization strategies results in availability of various equivalent query structures and ways to execute the resultant operators. Out of these different strategies, the best possible strategy is selected (cost-based selection). This final execution path is then processed to produce desired result.

2.1.2.1 Logical Optimization

Logical optimization reduces number of operations to be performed in a query. A expression in the relational algebra tree can be converted into an equivalent one with less operations using a set of algebraic rules reducing redundancy. This reduces query execution time. The rules re-write the given query to provide a logical plan with faster execution time. The most commonly used operations in DBMS are SELECTION and PROJECTION. Below listed are the possible optimization techniques used to provide logical optimal path for selection and projection queries.

- **Cascade of** σ: Conjunctive selection conditions can be broken into cascade of individual selections.
  $\sigma_{(p1 \wedge p2 \wedge p3... \wedge pn)}(R) \equiv \sigma_{p1}(\sigma_{p2}(...(\sigma_{pn}(R)...))$

- **Commutativity of** σ : The order of selection conditions can be changed.
  $\sigma_{p1}(\sigma_{p2}(R)) \equiv \sigma_{p2}(\sigma_{p1}(R))$

- **Cascade of** π : In case the set of attributes are same or the set of the outer projection are a subset of the inner one, everything except the last one can be ignored.

  $\pi_{a1}(\pi_{a2}(...(\pi_{an}(R)...)) \equiv \pi_{a1}(R)$

- **Commutative** σ *and* π : The operators are commutative given that the predicate in selection has the same attribute given in projection.
  $\pi_{(}\sigma_{c}(R)) \equiv \sigma_{c}(\pi_{a1,a2,a3...an}(R))$

The resultant tree from logical optimization reduces the execution time by having less operators to execute. However, the efficiency of processing itself depends on the use of right algorithm for the operators. This selection of right method to perform operation is done in physical optimization.

2.1.2.2 Physical Optimization

Each logical operator in relational algebra tree is executed using various algorithms. Each of these operator in the tree implements one step in the query execution. There are physical operators for operations not available in the relational algebra tree(eg: scanning).Also, different algorithms implement a single logical operator. The algorithms operate faster depending on the data. For example, Merge-sort join algorithm combines two tables faster, if the values in table are already sorted. Hence, selection of right algorithm based on data present is essential for faster processing. The possible implementations for few basic operations [DK97] are presented below:

Selection

Selection is the process of searching values from the given set of tuples that satisfies the given criteria. Below are some of the common selection algorithms [DK97] used,

- **Linear Search** - This algorithm retrieves all the records from memory and tests if the given selection criteria is satisfied. This is also known as brute force algorithm(refer book - fundamental of db). In this method all the data is subjected to selection criteria linearly and the tuples satisfying the predicate are forwarded.
- **Binary Search** - This algorithm is used if the underlying data are sorted and the selection condition contains equality condition. This is more efficient than linear search as each iteration reduces the search space into half. Every step in the iteration, half of the data are only searched.
- **Using a hash key** - This is used if equality condition is given to a key value attribute that is stored as hash value. This method directly computes the position of the key using hashing mechanisms rather than searching for the value. This algorithm retrieves at most one record.

Projection

Projection operation is easy to implement if the tuples are tagged with a key. In this case, "the result of the operation is equal to the number of tuples in the relation" [DK97]. if duplicates are available in the given relation the tuples are sorted first and then duplicates are removed. The tuples can also be hashed to eliminate the duplicates. This duplicates are removed only if the DISTINCT clause is present as by default SQL returns duplicates for projection.

Aggregation

In [DK97, GMUW00] Elmasri and Garcia-Molina explain the methods to compute aggregates over set of values. Based on the underlying data structure and the query used, different ways are used to calculate results. For example, if data are stored in B^+-tree structure, then maximum value can be found by following the right most pointer to reach the right most leaf of the tree.

2.1.2.3 Cost Based Estimation

The availability of different approaches to execute a query leads to explosion of paths that can be taken for execution. Hence, the best execution path has to be decided before start of the execution. The cost of executing the different approaches may differ from each other based on various properties.For example, as mentioned in the previous section, there are different algorithms that can be executed to perform join. The strategy with lowest cost is defined as the best plan to be executed. Also, finding the right execution will itself requires considerable CPU time. Also, there can be lot of

ways to execute the same query and estimating costs for all the paths is not feasible. For example, in case of join order optimization, the number of paths to join results increases with number of tables present in the join statement. Evaluating cost for all the paths consume more time than producing actual result. Hence, cost estimation must to restricted to only few feasible approaches. The basic optimization technique is to search for a solution in the search space that minimizes the cost function.Finding the best optimal path is not feasible due to its complexity. Different aspects are considered for computing costs. Below are the components in case of disk based DBMS [DK97],

- **Access cost to secondary storage** - The transfer cost of data blocks from disk to memory. This is also known as disk I/O cost(refer book - fundamental DBMS)
- **Disk storage cost** - The cost of storing intermediate files generated during execution.

  Cost based on execution of operations,

- **Computation cost** - The cost of performing in-memory operations. This is also known as CPU cost.
- **Memory usage cost** - Usage cost of memory buffers in main memory.
- **Communication cost** - Transfer cost of moving the results from database terminal to client.

These costs are estimated using predefined cost functions. Based on the data available in the database, cost estimate on the data are prepared. These estimates are provided as input to cost function to get the cost values. The estimates can be improved by,

- **Histogram** - Provides approximation of real distribution
- **Parameterized function** - Function that parameterizes the real distribution of data.
- **Sample estimates** - Selectivity is estimated from randomly selected sample.

Based estimates and cost functions, best possible path is selected. Once all the translation and optimization of query is done, the generated executable code is executed using the query executor.

2.2 Query Executor

Query executor processes the executable codes based on the path given by the query compiler. The processing model selected by the query executor is defined by the DBMS engine. Typically, there are two processing models normally used for query execution,

- Tuple at a time [Gra90]
- Operator at a time [MKB09]

These processing models are explained in detail below.

2.2.1 Tuple at a Time

This is also known as the volcano model. In this model, each operator processes one tuple at a time and forwards the results to the next operator available. Figure 2.3 shows execution on the tuple-at-a-time processing model. Each operator requests for a tuple to process using the next() call. The intermediate results from the current operator are placed in a buffer in order to be processed by the next operator. This method exploits the pipeline level parallelism and it yields partial results with remaining tuples still in process. Also, this model does not require all the records in memory to start processing.

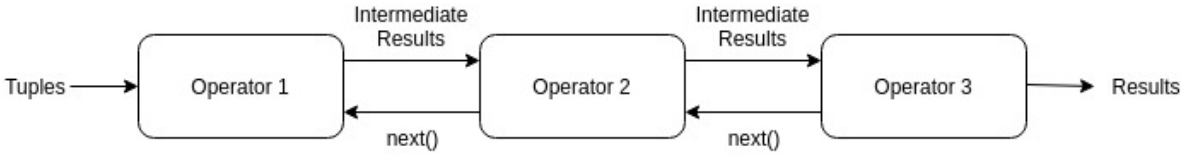


Figure 2.3: Tuple-at-a-time processing

Multiple operations are interleaved in this model for processing. The combined operations footprint is very large and may not fit into instruction cache. This leads to cache misses. The operators also call multiple next() increasing the functional call overhead. Also in case of functions like hashing, large intermediate results are generated and the results might not fit the data cache leading to data cache miss [AMDM07].

2.2.2 Operator at a Time

In operator at a time processing, each operation processes all the input tuples before forwarding the results to the next operator. Figure 2.4 represents operator at a time processing model. In each step, the intermediate results are fully processed. Hence, no partial results are returned during the execution. The results are not returned to the user until all the results are materialized. Therefore, this model has sequences of statements rather than interleaving multiple operations for execution. The advantage of this model is that each operator is executed once per execution cycle leading to

less instruction cache miss. This is because one operation resides in the cache at any instant. Inter-operator and intra-operator parallelism is used to speed the processing of this model.

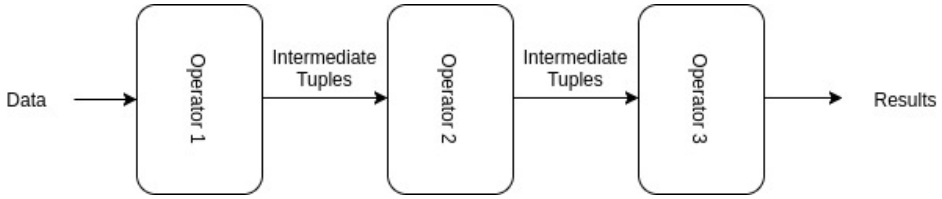


Figure 2.4: Operator-at-a-time processing

Processing of multiple tuples by single operator creates tight loops. This is optimized using features of modern compilers such as, loop unrolling, vectorization (using SIMD). However, the complete input data might not fit into the cache. This leads to the multiple access of memory for further processing. This requires repeated scan of data from memory. Also, the intermediate results may not fit into cache leading to further increasing the overhead of memory access. The data cache miss problem is reduced by vectorized processing model.

2.2.3 Vectorized Processing [ZB12]

Vectorized processing model combines the above mentioned models. This model processes the data in volcano-style iteration with bulk of data and forwarding this set of intermediate results during next() call. The vector [1] is selected so that it is big enough to perform parallel processing and also small in order to fit in the cache. Figure 2.5 is an example for vectorized processing model.

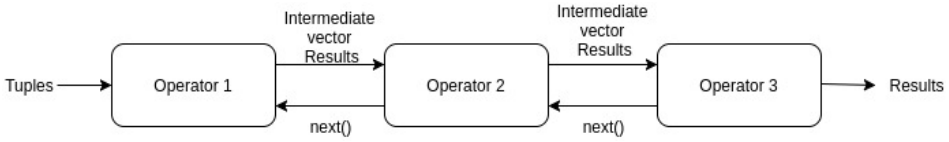


Figure 2.5: Vectorized processing model

This model exploits both the pipeline level parallelism and the data level parallelism. The volcano-style processing provides partial outputs from the different operations. This execution of operations over vector of data can be accelerated using modern hardware features such as SIMD. But, interleaving of operators while execution still leads to instruction cache miss reducing the efficiency of processing. This instruction cache miss is reduced by processing input data in pipelines. This processing model is explained in detail below.

[1] vector - Size of data to be processed

2.3 Pipeline Processing

Executing each database operators linearly produce lot of intermediate results from each operation. This creates an overhead of storing these values and retrieving them for the next operation in sequence. This overhead is reduced by grouping the operations. This reduces the number of partial results being stored. These grouped operations are executed as a single instruction. This process model is called pipelining processing [DK97].

Different set of operations can be combined to provide results without changing the nature of the operations. For example, selection operation can be combined with projection and the results are not altered. The result from selection are directly propagated to projection. This flow of values in the pipeline is not always constant. There are few functions that blocks this flow of values. They are called pipeline breakers.

2.3.1 Pipeline Breakers

In [Neu11], Neumann and et al, refers an operator as a pipeline breaker if the function needs all the intermediate results from the previous operator to perform its operation. In other words, these operators requires the results from previous operation to be stored into memory before starting their process. An operator is called a full pipeline breaker if it materializes the complete tuples before forwarding to next operators [Neu11].

For example, consider querying for grade point average students from different faculties with more than 100 students,

The query is written as,

SELECT F.FACULTY_NAME, **AVG**(S.GRADE_POINT)
FROM STUDENT **AS** S **NATURAL JOIN** (**SELECT COUNT**(STUD_ID) **AS** STUDENT_COUNT, FACULTY_NAME, FACULTY_ID
FROM FACULTY
GROUP BY FACULTY_NAME, FACULTY_ID) **AS** F
WHERE F.STUDENT_COUNT $>$ 100 **AND** S.DEPT_ID = F.FACULTY_ID;

The translated query looks and the corresponding pipeline boundaries are shown in Figure 2.6.

The pipeline-1 in Figure 2.6 has its flow obstructed by the aggregate grouping function. The function, COUNT(STUD_ID) materializes its result and stores it in memory before providing to next operator. Similarly the data has to be materialized for calculating group average for GRADE_POINT attribute.

The data flow in a pipeline is always obstructed by the pipeline breakers in it. These pipeline breakers can be optimized for faster execution by processing values quicker and propagating results to next operator. The algorithms for group-aggregation queries can also be optimized to perform their operations faster.

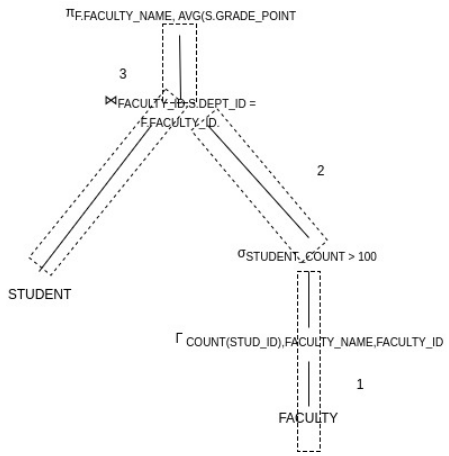


Figure 2.6: Query pipeline boundaries

2.4 Group Based Aggregation

SQL provides the ability to perform aggregation over a set of attributes in the given relation. Five aggregate operations are performed in SQL,

- Min
- Max
- Avg
- Count
- Sum

These aggregate functions are commonly combined with a `GROUP BY` clause to perform the aggregate for each set of values within group. Grouping and aggregation functions are implemented together for faster processing [GMUW00]. Group-aggregation is computed using two strategies: sort based and hash based aggregation [DK97].

The sort-based aggregation uses various sorting strategies available to first sort the input tuples. The tuples are sorted by the attribute given in the GROUP BY clause. Then, the aggregation function is executed over these sorted tuples. It takes two passes to calculate aggregation for groups with one pass for sorting with and another for computing aggregates for the groups.

Another way of performing group-aggregation is hash based aggregation. In this paper, we use the hash based grouping strategies to efficiently execute group-aggregation.

2.4.1 Hash Based Aggregation

Basic idea of the hash based aggregation strategy is to use the different hashing techniques available to group input tuples and perform aggregates for the groups. First, all the tuples are hashed into their respective buckets. This can be performed by a hashing function chosen based on the grouping attribute given in the `GROUP BY` clause. The function groups the input tuples of same group in a single bucket. There can be multiple groups present in a bucket. Once the tuples are grouped, aggregation is performed.

1 represents the basic operation of hash based aggregation. First, the process(lines 1 to 7) perform hashing over all the tuples using the group by attribute as hashing variable. Then, the process (lines 8 through 11) performs the aggregation for each groups within every buckets in the hash table.

Algorithm 1: Hash based aggregation

```
Data: Tuples
Result: Aggregates
1  while Tuple are available do
2      read current tuple;
3      key = hash(tuple);
4      if bucket[key] is free then
5          bucket[key] = value;
6      else
7          CollisionResolutionMechanism();
8  for each bucket do
9      for each group do
10         Perform aggregate;
11         return value;
```

As we can see, the hashing mechanisms are one of the main factors affecting the efficiency of the group-aggregation functions. In the below section, we discuss in detail the different hashing techniques used in performing group-aggregation.

2.5 Hashing Techniques

Hashing techniques are used to perform different applications in DBMS [DG85]. These techniques accelerate searching of a value and reduces the search space. These techniques use hashing functions to produce a key based on the given value. This key represents a bucket to store the inserted value. In some case, there can be two values hashing to provide a same key resulting in pointing to the same bucket. This is called collision. The hashing techniques resolve collision in different ways. The different hashing techniques and their collision resolution mechanisms are discussed in below sections.

2.5.1 Chained Hashing

Chained hashing [CLRS91] uses linked lists to store values. Each bucket has the head of a lined list and the values are stored into this node. If there is an overflow, the value is inserted by appending a new node to the linked list. This provides dynamic memory allocation for buckets. The basic structure of chained hashing is given in Figure 2.7. For overflow value, a new node is created to store the value in the corresponding linked list.

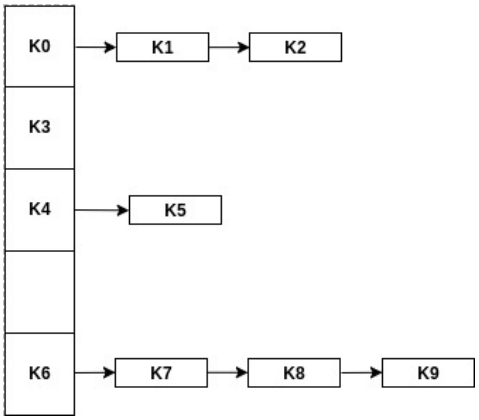


Figure 2.7: Chained hashing

Insertion

The insertion algorithm for chained hashing is given in Algorithm 2. First, the insert value is hashed using the hash function (line 1). The key returned from the hash function is then used to select the bucket. If no value is present in the bucket, the value is directly added to the head of the linked list (lines 2 to 5). If value is available in the head, the current value is appended to the linked list (line 7).

Algorithm 2: Chained hashing - insert

```
1 key ← hash(Data);
2 if bucket[key] = NULL then
3 |   head ← CreateLinkedList();
4 |   head ← data;
5 |   bucket[key] ← &head;
6 else
7 |   append(bucket[key], value);
```

For example, let us consider the hash table shown in Figure 2.8(a). The hash function used is $h(x) = x\%N$, where N is size of the hash table (in the example N=5). To insert 7, the value is hashed to the key 2. The position 2 is free and the value is stored. In case of 12, the key is again 2. Hence, the value is stored in a new memory location and the address is linked with the previous value as represented in Figure 2.8(b).

a) Chained Hashing - Before Insertion b) Chained Hashing - After Insertion

Figure 2.8: Chained hashing - insertion

Probing

Probing in chained hashing has two steps. The pseudo-code in algorithm 3 shows the probing process of chained hashing. First, the value to be probed is hashed to find the corresponding key. This key represents the bucket on which the value might reside. The values in the bucket are probed for the expected value using linear search. Search provides the value if present, else returns null.

Algorithm 3: Chained hashing - probe

```
1 key ← hash(Data);
2 if bucket[key] != NULL then
3 |   head ← bucket[key];
4 |   result ← LinearSearch(head, Data);
5 |   return result;
6 else
7 |   return NULL;
```

For example, To probe the value 12 from hash table available in Figure 2.8(b), the value is hashed first. The resultant key is 2. Then linear search is performed in the bucket 2 until the value is found. In this case, two memory spaces are probed in order to find the value.

The simplicity of the technique comes with several disadvantages. In worst case all the values reside in a single bucket. This leads to performing linear search for each probe. In fact, linear search of value has the time complexity of $\mathcal{O}(n)$. This increases the time to perform probing.

Dynamic memory allocation in linked list also introduces overhead to access the memory location. Each value is stored in different locations in a linked list, leading to additional

memory accesses for searching a value in it. This issue is addressed by open addressing techniques [MC86]. These methods store one value in each bucket location. Overflow values are stored in the same hash table by searching for different location in same table. Since, memory access degrades the efficiency of further operations open addressing techniques are more suitable for DBMSs [Lit80]. In this thesis, we discuss the open addressing techniques cuckoo hashing and linear probing and the ways to adapt them for faster aggregation.

2.5.2 Linear Probing

The data structure of linear probing allows to store one value per slot [Knu97]. There can be multiple slots in a bucket. In case of overflow, the insertion value is inserted in the next empty slot available in the data structure. The slot for the value is found by performing linear search until an empty position is found. The hash table for this techniques is implemented using an array data structure to store values. In Figure 2.9, we show an example of a partially filled data structure used in linear probing. The empty spaces in the table can be used to store collided values.

Figure 2.9: Linear probing

Insertion

The algorithm for linear probing insertion is given in Algorithm 4. Similar to chained hashing method, the insertion value is first hashed and the corresponding position in hash table is found (line 3). If the slot is already occupied, linear probing is done (lines 7 to 9) on the hash table to find the next empty slot and the value is stored in the slot.

Algorithm 4: Linear probing - insert

```
1 if Hashtable full then
2     return false;
3 key ← hash(Data);
4 if bucket[key]isfree then
5     bucket[key] ← Data;
6 else
7     while current slot not empty do
8         Goto next slot;
9     insert;
```

For example, consider inserting the value 12 into the partially filled table given in Figure 2.10(a). The hash function result for the values is given as 2. Since the location is already occupied by the value 7, linear probing is performed to find the next free slot to insert the value. In this case, the value 12 is stored in slot 3 as given in Figure 2.10(b).

a) Before insert

b) After insert

Figure 2.10: Linear probing- insertion

Probing

Probing follows the same function as insertion. The algorithm to probe is explained in Algorithm 5. Search value to be probed is hashed to get the key and the corresponding bucket is probed for value (lines 1 to 3). If the value is not found in the slot, linear search is performed until the value is found (lines 5 to 7). The algorithm assumes that, value is present in table.

Algorithm 5: Linear probing - probe

```
1 key ← hash(Data);
2 if bucket[key] = Data then
3 |   return true;
4 else
5 |   while value[current_slot] != key do
6 |   |   current_slot++;
7 |   return value[current_slot];
```

If all the spaces are occupied, the array size is increased and all the values are rehashed. The technique also suffers the same disadvantage of chained hashing. In worst case, the whole table is probed for the search value. Using the example in Figure 2.10(b), to probe the value 12, linear search is performed from slot 3 (given by hash function).

2.5.3 Cuckoo Hashing

Cuckoo hashing uses multiple hash tables to store values [DK12]. This function uses N hash functions and each corresponds to N hash tables. This function uses multiple hash tables as an alternative to store the overflow values. This technique also provides constant look-up time while probing. The general structure of cuckoo hashing is shown in Figure 2.11

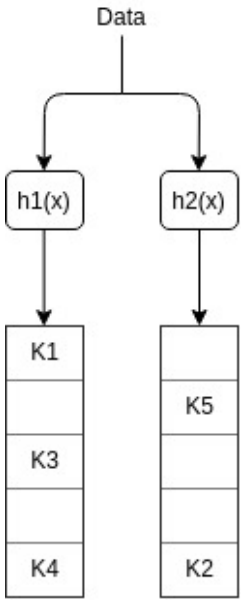


Figure 2.11: Cuckoo hashing

All the hash tables can be of same size or of different sizes. Also, the bucket in each table can hold multiple entries. Each table has its own hashing function.

Insertion

The general algorithm to insert a value is given in Algorithm 6. The insertion value is hashed with the first hash function to store the value in first hash table (line 2). If the key slot is occupied, then the value already present in the slot is evicted and the insert value is stored in the slot. Afterwards, this evicted value is hashed by the next hash function to store it in the next hash table (if present) (line 1 to 4). This process of eviction and storing of value is done iteratively until all the values are stored.

Algorithm 6: Cuckoo hashing - insert

```
1 while value!=null do
2     key ← hash(i,Data);
3     swap(Data, table(i, key));
4     i++;
```

For example, consider inserting a value into the cuckoo hashing tables shown in Figure 2.12(a). The hash functions used in the example are h1(x) = x%5 and h2(x) = floor(x/5)%5 respectively. To insert the value 3, it is first hashed using hash function h1(x) providing the key as 3. However, the slot 3 is already occupied by the value 43. The value 43 is swapped with the value 3. Now, the value 43 is rehashed with function h2(x) and the value is stored into the slot in hash table 2 as given in Figure 2.12(b).

a) Before insertion b) After insertion

Figure 2.12: Cuckoo hashing- insertion

Probing

Probing in cuckoo hashing has near constant look up time. The general algorithm to probe a search value in cuckoo hashing is given in Algorithm 7. To probe a value, the search value is hashed by all the hash functions(line 1 and 2). The slots of the corresponding keys provided by hash functions are probed in their respective tables. The search value is present only in one of these probe positions and result is returned in case value is found(line 3 and 4). Else, the probed value is not found.

Algorithm 7: Cuckoo hashing - probe

```
1 while i != Total no. of hash functions do
2     key ← hash(i,Data);
3     if search_value = table(i, key) then
4         return true;
5 return false;
```

For example, to probe the value 11 in the hash tables available in Figure 2.12(b), the value is hashed by both functions h1(x) and h2(x) providing the keys (1,1) respectively. These slots are probed in the tables respectively to get the result.

The overall performance of these two techniques can be improved by using modern processors, mainly using SIMD. For example, SIMD can be used to perform the same hashing function over multiple data. The efficiency of the hashing techniques can be improved using SIMD [JA17] [MSL+15]. In Section 2.6, we detail the SIMD operations that can be adapted to perform the hashing operations.

2.6 Single Instruction Multiple Data (SIMD)

SIMD enables processing of several data items in parallel by exploiting data level parallelism. In this thesis, we use the SSE instruction set for x86 architecture to perform the SIMD operations. This instruction set uses 128- bit registers to perform operations

on data thereby, processing 128 bits in one clock cycle. This functionality of SIMD can be used to accelerate various DBMS operations [ZR02] [PRR15].

In below sections, we explain in detail the SIMD functions[2] necessary for performing the hashing techniques detailed in above sections. These functions are categorized as,

- Arithmetic operation
- Bit manipulation
- Bit comparison

Also, we limit ourself to data stored as type **__m128i**. This type holds four 32-bit integers to be processed in parallel.

2.6.1 Arithmetic Operation

Hash calculation in the hashing techniques requires performing arithmetic operations over given input. The SSE instructions to perform these operations are,

_mm_add_epi32(m128i a, m128i b) adds four 32-bit values in one cycle. This packs the registers with four 32-bit values adding to 128-bit.

_mm_sub_epi32(m128i a, m128i b) is used to subtract four 32-bit values in one cycle.

_mm_mul_epu32(m128i a, m128i b) multiplies two lower 32-bit values from two 64-bit values. This provides a 64-bit value as output.

2.6.2 Bit Manipulation

SIMD can be used to perform bit manipulation by combining multiple values. For example, a set of values can be collectively shifted or rotated. Below are the functions that are useful in performing the hashing techniques.

_mm_shuffle_epi32(__m128i a, int imm) shuffles four signed or unsigned 32-bit integers based on given `imm` value. Here, `imm` serves as the mask using which the shuffling is performed.

_mm_andnot_si128(__m128i a, __m128i b) is a bit manipulation operation that performs (Ã)&B on two 128-bit values and produces the result.

_mm_and_si128(__m128i a, __m128i b) performs bitwise AND operation on the given two 128-bit values. This is useful for executing bitwise AND over multiple values. For example, two sets of four integers packs can be performed bitwise AND operation.

[2] referred from https://software.intel.com/sites/landingpage/IntrinsicsGuide/

_mm_or_si128(__m128i a, __m128i b) executes bitwise OR operation on given 128-bit values. This function can be exploited the same way as bitwise AND operation.

_mm_movemask_epi8(__m128i a) returns an integer by selecting the most significant 8-bits from the values in `a`.

_mm_extract_epi16(__m128i a, int imm8) this function extracts 16-bit integer from `a` using the integer mask given in imm8.

2.6.3 Comparison Operation

Comparison of values is performed for both insertion of a value and for search of a value. The function to compare values is,

_mm_cmpeq_epi32(__m128i a, __m128i b) compares four signed or unsigned integers and provide the result as 1 for equality of bit and 0 for inequality. The end result of the operation is a series of 0s and 1s.

3. Scalar and Vectorized Hash-Based Aggregation

Hashing techniques group data based on a hashing function and this can be easily extended to perform grouped aggregation [SZ96]. Also, Kenneth Ross in [Ros07] explains several ways to exploit the modern hardware feature-SIMD to probe a value in the cuckoo hashing technique. This SIMD search in cuckoo hashing can be extended to perform also the aggregation functions. SIMD can also be used to probe a value in the linear probing. When grouping is done in parallel using SIMD, it is advisable to directly aggregate values while grouping [SZ96]. In this chapter, we explore the possibility to perform direct aggregation while grouping using cuckoo hashing and linear probing methods. We also attempt to accelerate the grouped aggregation process using SIMD. We adapt the work of Kenneth Ross in [Ros07] to probe values in cuckoo hash tables using SIMD to perform aggregation directly while grouping.

3.1 Cuckoo Hashing

In this section, we discuss the conceptual design of cuckoo hashing technique adapted for SIMD probing and direct aggregation. We explain the cuckoo hashing tables structure modified for SIMD in Section 3.1.1. We explain in Section 3.1.2 the mechanism to probe value in the modified table structure. Also, we detail the way to insert a new value into the hash table bucket in Section 3.1.3. Finally, we explain the extension of scalar probing and insertion using SIMD acceleration in Section 3.1.4 and Section 3.1.5.

3.1.1 Table Structure

Cuckoo hashing technique store values in multiple tables. Hence, the technique needs at least two hash tables to perform insertion of a new value. The basic structure of the table is given in Figure 3.1. Each table holds two value-payload pairs per bucket.

This is due to the use of SSE2 instruction set for SIMD. SSE2 has 128 bit registers and can accommodate at most 4 integer values as we are restricting to integers (32*4 = 128 bits). Depending on the SIMD instruction set and the data type used, the table bucket size can be varied. To use SIMD efficiently, the values are packed together followed by packed set of payloads. The advantage of this method is explained in Section 3.1.4. This data structure is used in both scalar and vectorized version of cuckoo hashing techniques.

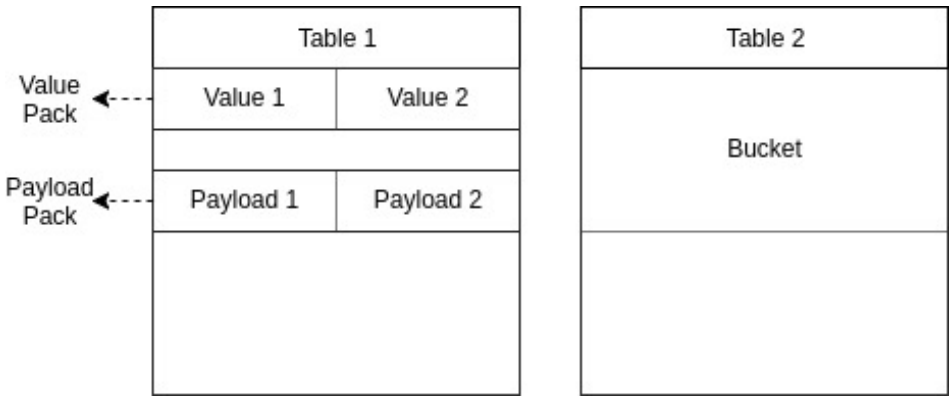


Figure 3.1: Table structure - cuckoo hashing based on [Ros07]

3.1.2 Scalar Probing

The probing module for cuckoo hashing is implemented along with the aggregation mechanism. This technique branches the execution either to insertion of the new value or perform aggregation on the existing value. In this thesis, we consider count as our aggregation function to be performed over the groups. While probing for a values, each value in the selected bucket is checked for equality with the search value. If they are equal, the corresponding aggregate payload is incremented.

The general steps of cuckoo probing is given in Algorithm 8. The value to be probed is hashed by all the hashing functions. Each slot identified by the hash result is probed in linear fashion until the result is found as mentioned in Figure 3.2. Once the value is found, resultant of comparison is used to determine the aggregate. Insertion is executed if the value is not present in any of the slots.

Algorithm 8: Cuckoo hashing - scalar probe

```
1 for i in no. of tables do
2     key ← hashFunction(i,Data);
3     Bucket = Table[i,key];
4     for j in BucketSize do
5         if Bucket[j].key == Data then
6             Bucket[j].value++; return;
7     insert(Data);
```

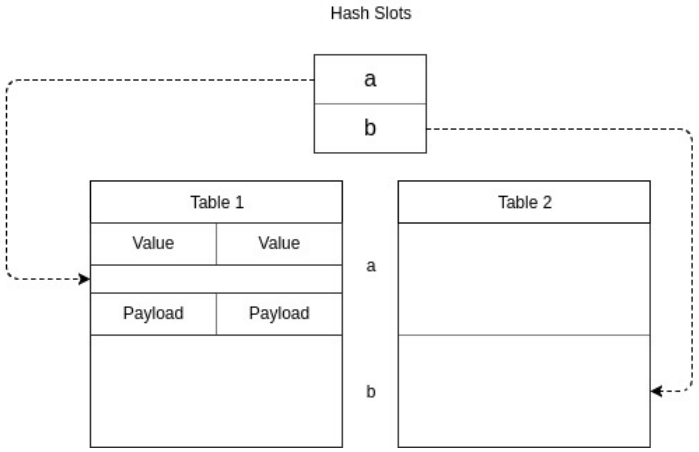


Figure 3.2: Cuckoo hashing - probing

3.1.3 Scalar Insertion

Faster probing in cuckoo hashing is traded for high insertion time. Once the search value is not found in the tables, the value is inserted along with its corresponding aggregate payload. The mechanism to insert a value-payload pair in the table is given in Algorithm 9

Algorithm 9: Cuckoo hashing - scalar insert

```
1 Aggregate ← InitialValue;
2 i ← 0;
3 while value ≠ NULL do
4     slot ← Hash[i](value);
5     value ← shift(hashtable[i][slot].key,value);
6     aggregate ← shift(hashtable[i][slot].value,aggregate);
7     i ← (i++)%2;
```

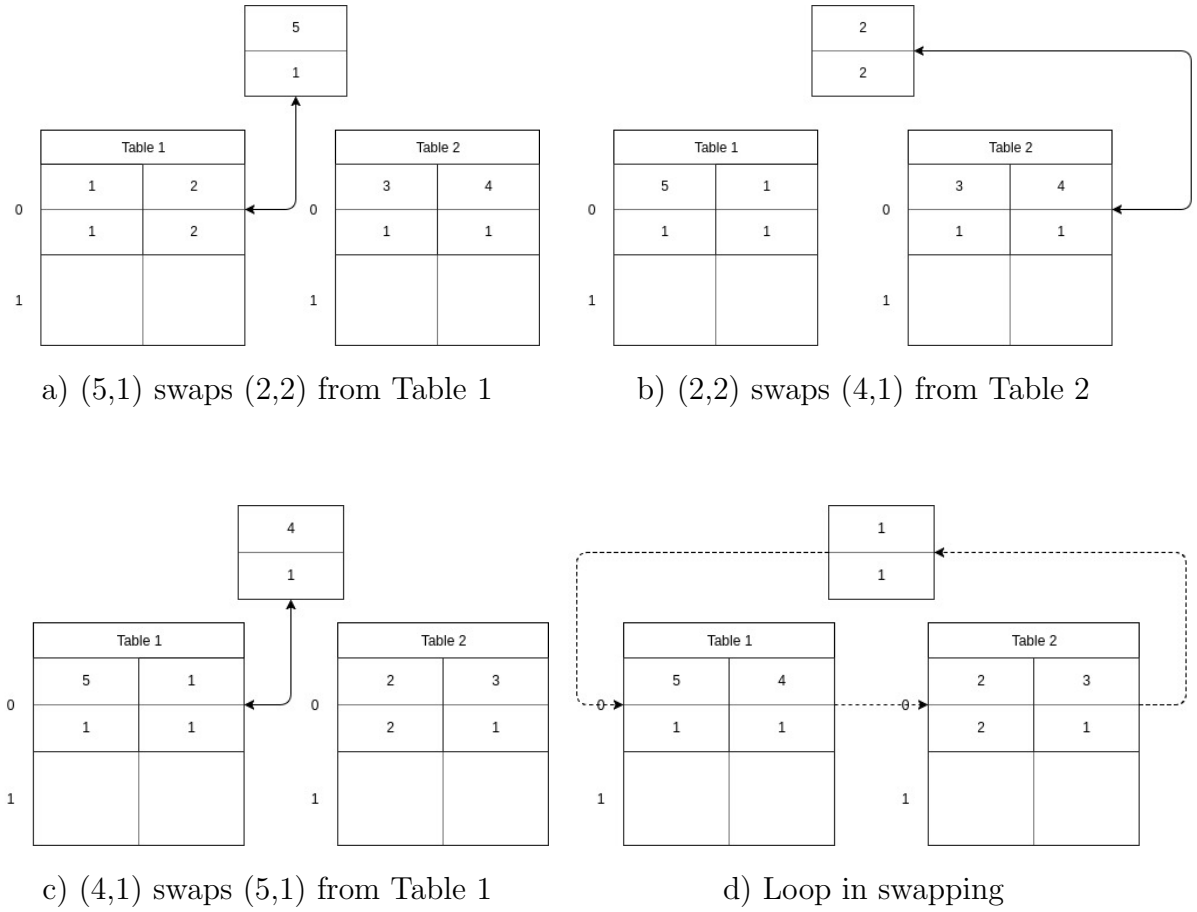
The value and aggregate payload to be inserted are initialized into a temporary variable and the process to find the correct position for the pair is started. Using the first hash function's result, the position in table 1 is determined. The pair is inserted into the bucket available in the location. If the slot has no free space, the oldest value-payload pair is evicted and the current pair is stored. This can be done by shifting the values inside the bucket so that the oldest value is evicted. This evicted pair is inserted in its corresponding position in next table. This process is looped until no value-payload pair is available for insertion.

3.1.3.1 Insertion Cycle

The eviction of a value-payload pair from a bucket may lead to constant eviction of pairs between the tables. This may form an insertion cycle. In the case of two tables, the oldest pair from table 1 is evicted and stored in temp space. This pair in the temp

space might evict another pair in table 2. Now, This second pair might again evict a pair from the same bucket in table 1 in which the first pair was stored forming a cycle.

For example, Consider the table as below with hash functions h(x) = x %10. Suppose, if pair (5,1) is to be inserted, it will be given the slot 0. This evicts (2,2) from the table 1. Now, value pair (2,2) is inserted into the second table. The slot given for the pair is 0 again, which evicts (4,1). Again, the value pair (4,1) hashes to slot 0 in table 1 forming a cycle. This creates endless swap of pairs between tables.



a) (5,1) swaps (2,2) from Table 1

b) (2,2) swaps (4,1) from Table 2

c) (4,1) swaps (5,1) from Table 1

d) Loop in swapping

Figure 3.3: Cuckoo hashing - insertion cycle example

The insertion cycle can be suppressed by limiting the number of swaps for a single insertion. A counter is set to keep track of number of swaps being carried on. After a given set of swaps are made, the iteration is stopped and cycle resolution is performed. Insertion cycle can be resolved in two ways. When the cycle is detected, all the table structures can be changed and values are rehashed. The tables can be increased of their bucket size, more buckets can be added and finally, number of tables can be increased along with hashing function. Although this method increases execution time effective cardinality estimators can limit the number insertion swaps there can be.

3.1.4 SIMD Probing

Using SIMD, all the values in a bucket can be probed instantly. This is done by modifying the table structure to suit SIMD execution. We use the splash tables structure [Ros07] for our SIMD implementation. A row in this data structure store buckets of same key from two different tables. The array positions 0,1 stores bucket from table 1 and positions 2 and 3 holds table 2 bucket of the same slot. During SIMD probing, each probe is done on buckets in same row from table 1 and 2.

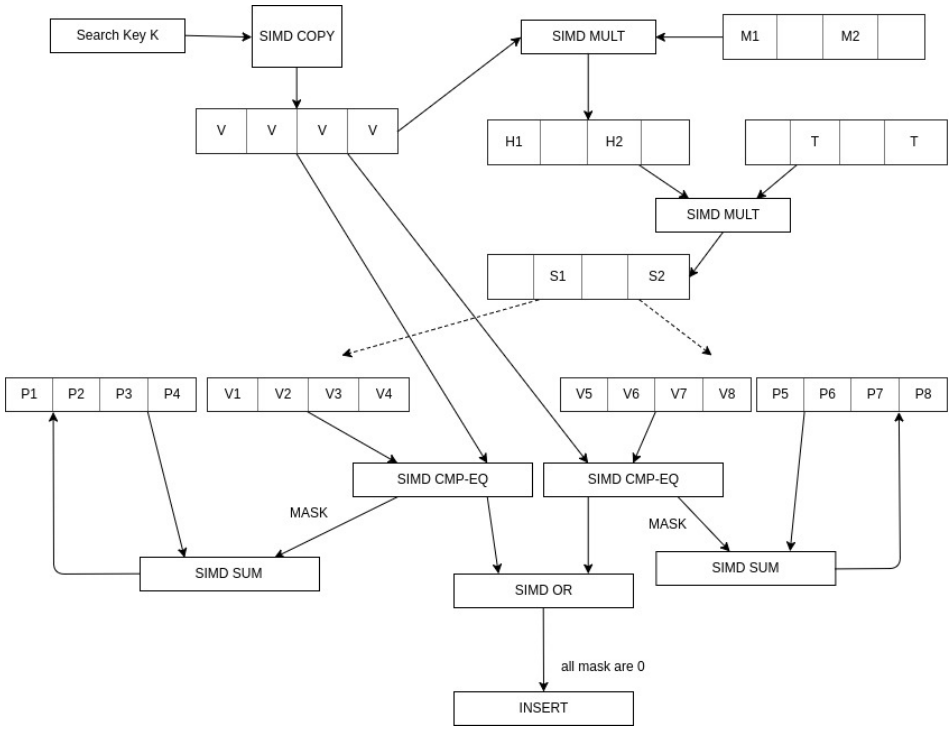


Figure 3.4: Aggregation over SIMD cuckoo probing [Ros07]

The mechanism of SIMD probing in cuckoo hashing is given in [Ros07]. This process flow is tweaked to adapt for aggregate computation. Figure 3.4 shows the process flow for aggregate computation while hash probing. In cuckoo probing implementation [Ros07], search value is first duplicated into SIMD registers.Then the hash multipliers are multiplied with the values to get keys. This resultant value is again multiplied with table size to get the slot locations. SIMD MULT provides the values in slots mentioned in the Figure 3.4. The values available in the slots are compared for equality with the search value and set of mask values is returned. Since the comparison masks are series of 0s and 1s representing equality or inequality of values the masks from comparison is added to the payloads to perform aggregation. If all the values in the mask are zero, insertion is performed.

3.1.5 SIMD Insertion

SIMD insertion is similar to scalar insertion. The hash function is executed using SIMD and provides the slots for both the tables in one clock cycle. Since two buckets from the two different tables are present in one slot, extra execution is done to select the values from a single table to be used for insertion. These extracted bucket is then used to swap values to accommodate the new entry. This is done using controlled shuffling in SIMD.

3.2 Linear Probing

we detail in this section the design of SIMD accelerated linear probing and direct aggregation on the hashing technique. In Section 3.2.1 we detail the model of the hash table and its bucket structure used for performing linear probing. in Section 3.2.2, we explain the primary grouping problem present in linear probing and the ways to prevent it. Scalar probing in the given table structure is explained in Section 3.2.3 and insertion of data in Section 3.2.4. Finally, we detail the SIMD accelerated linear probing and insertion mechanisms in Section 3.2.5 and Section 3.2.6.

3.2.1 Table Structure

The value-payload pairs are stored in linear arrays for linear probing mechanism. SIMD can be used to search multiple values in sequence in the arrays thereby providing faster results. Probing of data is the crucial part in this method and insertion also follows the same process flow. So, this method is made as an extension for probing in this technique. We use structure of arrays to store the value-payload pairs in the hash tables. The values and payloads are stored in a separate arrays of SIMD data type. The arrays hold packed values and payloads respectively, since multiple values can be stored in one SIMD data type variable. The value and its corresponding payload are stored in the same index in the two arrays for easier access.

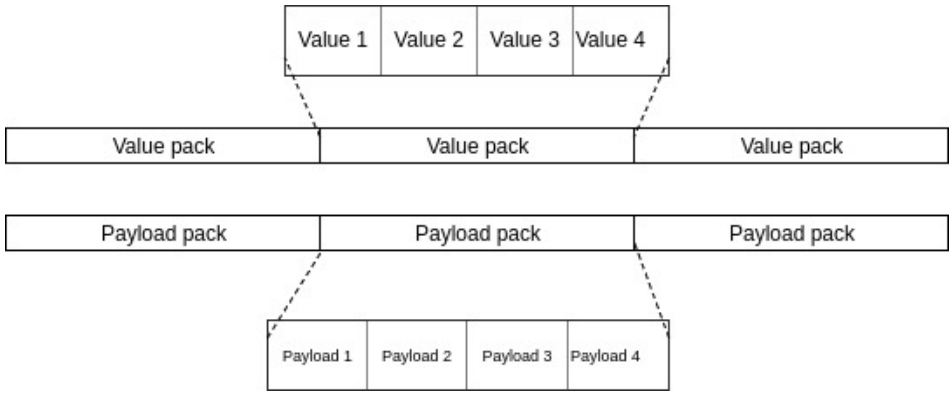


Figure 3.5: Linear probing - table structure

3.2.2 Primary Grouping

The efficiency of linear probing degrades with the formation of primary clusters [Smi04]. These clusters are formed if more values hash to the same slot. This hashing of values to the same slot increases the distance from the desired slot forming a cluster of values for the same location. For example, consider a hash table of size 6. Using the hashing function h(x) = x % TABLE_SIZE, the values 1,11,21,31 and 41 will hash to same location: 1. For the value 1, the value is stored in the desired location. But, in case of the other values the distance between the desired value increases linearly forming a cluster as shown in Figure 3.6. These clusters are removed by using Knuth's hashing function [Knu97] which provides good dispersion of values within a hash table.

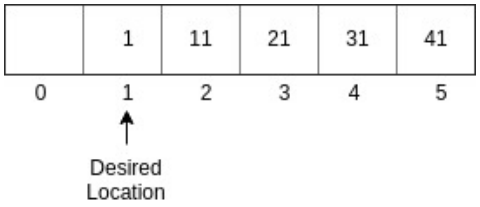


Figure 3.6: Linear probing - primary clusters

3.2.3 Scalar Probing

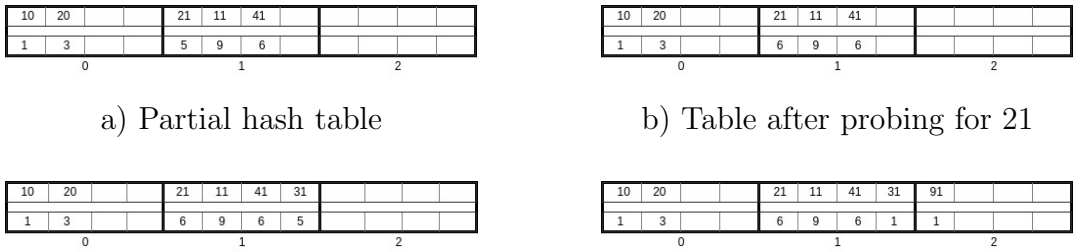
First, the corresponding slot is computed by hashing the given search value. Then, the slot provided is probed i.e. the search value is compared with each value in the bucket linearly for equality. In case all values in the bucket are unequal, the next slot is probed. Iteratively all slots in the table are probed until search value is found or an empty location is found.

If the search value is found, the necessary aggregation is performed. The corresponding payload of the value in the bucket is updated with the necessary aggregate value. For example, the resultant from comparison of equality of bucket values and the search value can be used to sum up the count as the test yields either 0 or 1.

In case probing pointer reaches the end of the table, probing is started from slot 0 of the hash table. The hash table is full when the search again probes the hashed slot. In this case, the bucket and table sizes can be increased and the value-payload pairs are re-hashed to accommodate new pairs. Similar to cuckoo hashing the cardinality estimators from DBMS can be used to define the table size so that no overflow of value occurs.

3.2.4 Scalar Insert

Insertion in linear probing is done when an empty location is found while searching for a value. The empty slot represents unavailability of the search value in the table. Since an empty location is the nearest to the hashed slot, the new value along with its corresponding aggregate payload is stored in this location.



a) Partial hash table

b) Table after probing for 21

c) Table after insertion of 31

d) Table after insertion of 91

Figure 3.7: Linear probing - insertion example

Consider performing sum aggregate on series of values using the partially filled table given in Figure 3.7(a). The hash function used is h(x) = x % 3. If 21 is the next value in the sequence to be aggregated, the value is hashed and the slot is given as 1. Linear probing is done in this bucket and the corresponding value is incremented as the key is already present in the table. The updated table is given in Figure 3.7(b). If the value 31 comes next, hashing functions gives the slot 1 again. Linear probing hits an empty space in the bucket, the value is stored in the location with its corresponding aggregate payload initialized as in Figure 3.7(c). Finally, in case of the value 91, the hash key is 1. The value is again searched in bucket 1 and now it is full. The next bucket is probed and the first location is found to be empty. The value is stored in the location and aggregate is initialized. The final hash table is given in Figure 3.7(d)

3.2.5 SIMD Probing

Similar to cuckoo hashing, the bucket with four values can be processed faster with SIMD acceleration. The mechanism is given in Figure 3.8. The value to be probed is duplicated and loaded into a SIMD variable. Using the hash function the slot to be probed is determined. Instead of comparing equality for keys value-wise in the probe bucket, the values inside whole bucket is loaded into a SIMD variable. These variables can be used to compare four values at one clock cycle. The two SIMD variables are compared and the resultant value is checked for equality of search value.

Similar to scalar process, the comparison value can be used directly to perform aggregation. The result has a set of 0 or 1 masks given by the comparison operator. The packed payload set is loaded into another SIMD variable and the resultant mask from comparison is subtracted to it to perform the aggregation. The mask value is subtracted because the resultant value is either 0x0000 or 0xFFFF. The process is stopped if value is found in the probed bucket.

3.2.6 SIMD Insert

If the value is not available in the current probed bucket and also an empty space is encountered, insertion is performed. This function follows the same procedure as scalar insertion. The new value can be either inserted by directly finding the position using

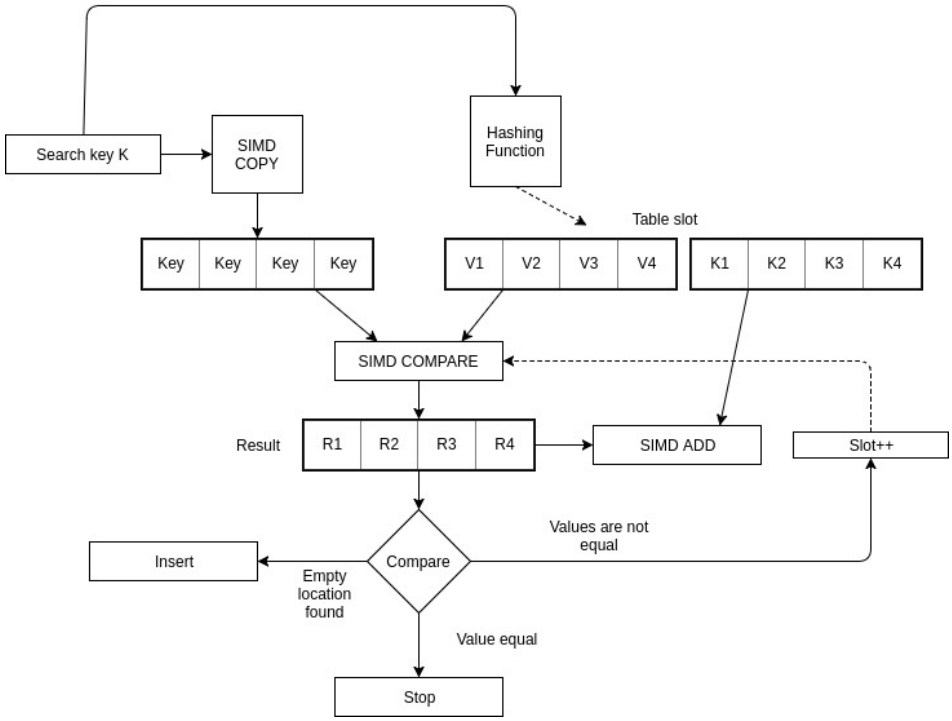


Figure 3.8: Linear probing - SIMD probing with count aggregation

the comparison vector or the value can also be inserted by using shifting operations in SIMD. This is done by shifting all the values in bucket by one slot in the table. No preexisting value in the bucket is evicted due to shifting operation as the new value replaces an empty location available in the bucket.

4. Vectorized Hash Based Aggregation - Implementation

Based on the design details of hashing techniques, the hashing system is implemented in C++. The implementation is done for GCC compiler version 4.9.2 and the SIMD features are implemented for SSE2 instruction set. We use count as the aggregate function to be performed.

4.1 Cuckoo Hashing

Based on the table structure in Section 3.1.1, a SIMD variable can hold multiple values. This is implemented by using a union of different basic data types as given in Listing 4.1. Using this, any one of the primitive data type can be used to store values within the hash table.

```
typedef union {
__m128i v;
unsigned int ui[4];
signed int si [4];
unsigned short us[8];
signed short ss[8];
} vec __attribute__ ((aligned (16)));
```

Listing 4.1: Data definition

Based on the size of bucket various information can be stored. The basic structure of a bucket and the hash table is given in Listing 4.2. This hash table bucket holds a single pack of values and their respective payloads. The hash table is constructed with user defined number of buckets.

```
typedef struct {
__m128i keys;
__m128i payloads;
} entry;

entry hashtable[HSIZE] __attribute__ ((aligned (128))); \\HISIZE − hash table size
```

Listing 4.2: Bucket and table structure

4.1.1 Hashing Function

The SIMD-based hashing function is implemented as below. Knuth's values for the hashing function are stored in a SIMD variable and SIMD multiplication is called. Once the process is started, the function `setM()` function is executed. This sets the initial values for the hashing function. Now, once the `hash()` function is called, the hash values in the variable `m0` and the search value, `k` are multiplied and the result is again multiplied with size of the table. Since, SIMD multiplies low unsigned 32 bits from packed 64 bit integers, the modulo results of two values are available in the lower 32 bits of the results. These lower 32 bits are then extracted using `kextract()` function representing the slots.

```
#if (HSIZE <= 65536)
#define kextract(hvec,size) _mm_extract_epi16(hvec,size)
#else
#define kextract(hvec,size) (_mm_extract_epi16(hvec,size) | (_mm_extract_epi16(
    hvec,size+1)<<16))
#endif
void setM(){
m.ui[0] = 1300000077;
m.ui[2] = 1145678917;
m0 = m.v;
tbsize = _mm_set_epi32(HSIZE,HSIZE,HSIZE,HSIZE);
}

__inline __m128i hash(__m128i k)
{
__m128i h;
h = _mm_mul_epu32(m0,k);
h = _mm_mul_epu32(h,tbsize);
return h;
}
```

Listing 4.3: SIMD hashing function

4.1.2 SIMD Probing

For probing, the search value must by converted from basic type to SIMD variable type i.e. the given value is replicated and stored into the SIMD array. Listing 4.4 shows an excerpt of the SIMD probing technique. The replicated search value is hashed using the SIMD hashing function and the slots to probe in the hash tables are determined. Each slot is extracted using the `kextract()` function, and the value in the slot is compared with the search value. The resultant mask from the comparison is then added with the payloads available in the slot. The SIMD function `__mm__cmpeq__epi32()` provides -1 in case of a match. Hence, the payload values are subtracted with the resultant masks.

```
__inline int SIMDprobe(unsigned int key){
        ...
        ...
        ...
        k=_mm_cvtsi32_si128(key);
        k=_mm_shuffle_epi32(k,0);
        ...
        ...
        ...
        foffset0  = kextract(h,2);
        slot0  = hashtable[ foffset0 ]. keys;
        tmp0 = _mm_cmpeq_epi32(k,slot0);
        hashtable[ foffset0 ]. payloads = _mm_sub_epi32(hashtable[foffset0].payloads,
            tmp0);

        foffset1  = kextract(h,6);
        ...
        ...
        ...
        tmp1 = _mm_andnot_si128(tmp0,tmp1);
        hashtable[ foffset1 ]. payloads = _mm_sub_epi32(hashtable[foffset1].payloads,
            tmp1);
        ...
        ...
        ...
        int maskval = !((_mm_movemask_epi8((__m128i)tmp0)) || (
            _mm_movemask_epi8((__m128i)tmp1)));
        return maskval;
}
```

Listing 4.4: SIMD probing

Since we use splash tables for our implementation of cuckoo hashing, the probing of a value is done for values available in same slot for both the tables. Hence, there might arise a situation where the hash function might provide the same slot for both tables.

In this scenario, the aggregate value would be added twice. To eliminate this error, we use the `_mm_andnot_si128()` function. This function performs B AND ($\tilde{A}$). This function ensures that value of aggregate is not changed by the second slot's probing if the value was updated during the probe of the first slot. The maskval returned by the probe function is used to determine if an insertion has to be performed.

4.1.3 SIMD Insertion

The insertion in cuckoo hashing has indeterministic time. This is due to the insertion cycle. The Listing 4.5 shows the part of insertion mechanism used. The insertion is done within a FOR loop to make sure the insertion cycle is cut-off. In our implementation, the limit is set to 1000. Apart from that, the insertion FOR loop is also stopped if no other value, payload pairs are evicted. Eviction of values is done using the shuffling function available in SIMD. Insertion of value-payload pair alternates between table 1 and 2. This is indicated by count value. If *count*%2 is 0, table-1 is selected, else table-2. First, the values and payloads present in the slot given by hash function is pointed by integer pointers to select each value-payload within the bucket. Depending on the table we are currently inserting the shifting of values must be done. Since this function shifts all four values in the SIMD register, the shuffling function-`_mm_shuffle_epi32(value,mask)` is used with respective mask to shift only the desired values. For shifting values in table 2, mask value 225 is used and mask is set as 180 to shift values in tablecount 1. Finally, We insert the valuer and payload pair.

```
int SIMDinsert(unsigned int searchKey){
   for(count=0;((count<1000)&&(key!=0));count++){
          ...
          ...
          ...
          h = hash(k);
          ...
          ...
          ...
          int *valkey = (int*) &hashtable[foffset ].keys;
          int *val = (int*) &hashtable[foffset ].payloads;
          ...
          ...
          ...
          hashtable[ foffset ].payloads =_mm_shuffle_epi32(hashtable[foffset ].payloads
              ,225);
          hashtable[ foffset ].keys =_mm_shuffle_epi32(hashtable[foffset ].keys,225);
          ...
          ...
          ...
          valkey[(2*countPercent)] = key;
```

```
        val[(2*countPercent)] = payload;
        key = tmpk;
        payload = tmpp;
    }
    if(count >= 1000){
        //printf("count : %d\n", count);
        return -1;
    }
    return count;
}
```

Listing 4.5: SIMD insertion

4.2 Linear Probing

Linear probing utilizes a similar physical structure for table as cuckoo hashing table structure. Hence, the code in Listing 4.1 and Listing 4.2 are reused for data definition and table structure. But, the access to the buckets and insertion of values into them are performed differently.

4.2.1 SIMD Probing

In Section 3.2.5, it is given that insertion in linear probing is done along with probing itself. Hence, insertion is included within the implementation of probing for the technique. The code excerpt for linear probing using SIMD is given in Listing 4.6.

```
int VectorProbe(unsigned int key){

unsigned int foffset = hash(key);
...
...
...

slot  = hashtable[ foffset ].keys;

k = _mm_set_epi32(key,key,key,key);
tmp = _mm_cmpeq_epi32(k,slot);


if(_mm_movemask_epi8(tmp)){
        hashtable[ foffset ].payloads = _mm_sub_epi32(hashtable[foffset].payloads,
            tmp);
        return 1;
}
```

```
__m128i zeroPos = _mm_cmpeq_epi32(mask0,slot);
int resMove = _mm_movemask_epi8(zeroPos);
if(resMove){

        hashtable[ foffset ].payloads = _mm_bslli_si128(hashtable[ foffset ].payloads
            ,4);
        hashtable[ foffset ].keys = _mm_bslli_si128(hashtable[ foffset ].keys,4);

        //Place the key in first position
        int *valkey = (int*) &hashtable[foffset ].keys;
        int *val = (int*) &hashtable[foffset ].payloads;
        val[0] = 1;
        valkey[0] = key;
        return 2;
}

 foffset =foffset++;
return -1;

}
```

Listing 4.6: Linear probing - SIMD probing

Probing starts with executing the hashing function to get the key. The bucket in the position is probed for the search value. There are three outcomes for this process.

- The value is found in the bucket. Then, the comparison mask is added with payloads of the bucket and the process is stopped.
- The value is not in the bucket and is full. In this case, the slot value is incremented and probing is done again with the new bucket.
- At least one null value in the bucket. The value is then inserted into the first empty location available in the bucket by moving all the values and payloads one slot towards right.

5. Runtime Analysis of the Hashing Techniques

Faster processing of hashing technique can be determined by their runtime. This estimate of the hashing technique runtime is done with different dataset distributions. The hashing techniques are tested of their efficiency for both scalar and vectorized implementation. This is executed to determine the speed up gained by SIMD acceleration. We use count as the aggregation function for all the evaluation scenarios. In Section 5.1, we detail the execution environment and the setup necessary for evaluating the hashing techniques. We also explain the different dataset distributions used for evaluation. We provide our analysis about the cuckoo hashing evaluation results in Section 5.2. Then, we present the results for linear probing in Section 5.3. We provide a comparative analysis of both techniques based on the results in Section 5.4. We also provided a comprehensive discussion on the impact of group size in execution of grouped aggregation in Section 5.5. Finally, we provide our conclusion in Section 5.6.

5.1 Evaluation Setup

The datasets given as input to the hashing techniques ranges from one to fifty million records with step size of five million. These dataset are evaluated for twenty iterations and the average of the results are used. We determine the execution time taken for the techniques in our evaluation. We explain the different distributions used for evaluation in Section 5.1.1. The details of the execution environment used for our evaluation is discussed in Section 5.1.2.

5.1.1 Dataset Distributions

The hashing techniques are tested with three different data distribution sets. Two of these distribution sets have only unique values to be grouped and aggregated. These datasets are used to test the insertion mechanism of both the techniques. The distributions used are,

Dense unique random distribution

As the name suggests, the data in the dense unique random value distribution are all unique and are densely populated. This random distribution of unique values leads to scattering of values. The values in this distribution are generated using the generator explained in [GSE+94]. For generating a value in the distribution, We select a prime number(P) greater than the dataset size(n) and a random generator(G)[1] . New value is generated by performing $value(i) = G^i \ mod \ P$.

Sequential distribution

Sequential distribution generates series of values, creating a sequence of given order. In our evaluation, we used a arithmetic progression generator with step size(S). A value in the series is generated by, $value(i) = i * S$.

Uniform random values

All data in uniform random distribution have the same probability. Generation of each value in this series is random and does not depend on the previous value generated. The values are generated using, $value(i) = rand()$.

5.1.2 Evaluation Environment

All the evaluations are ran on a machine running CentOS Linux version-7.1.1503 with Octa core Intel Xeon E5-2630 v3s- 2014. The machine has 1024GB RAM for processing and memory.

5.2 Cuckoo Hashing Evaluation

Cuckoo hashing has an indeterminate insertion time. Hence, we have split the execution time for cuckoo hashing into insertion time and probing time. The analysis is performed to determine the general impact of insertion and probing in cuckoo hashing. We also compare the execution time needed for scalar and vectorized cuckoo hashing for both insertion and probing. In Section 5.2.1 we detail the results for cuckoo hashing insertion and in Section 5.2.2 we discuss the results for probing.

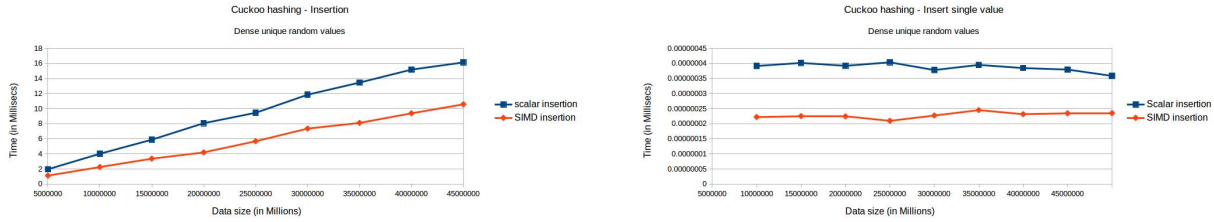
5.2.1 Insertion

The insertion time taken for scalar and vectorized cuckoo hashing for the given distribution sets are plotted in the charts. Since cuckoo hashing is prone to insertion cycles, there could be potential outliers forming the insertion loop and deteriorating the execution time. Hence, our test script record only the runtime for successful insertion.

[1] For dispersion of values in the distribution

Dense unique random values

For dense unique random distribution of values, the insertion time increases with increase in unique values as shown in Figure 5.1(a). Figure 5.1(b) shows that insertion is nearly constant for both scalar and vectorized cuckoo hashing. Also, we see SIMD acceleration is nearly 1.5 times faster than scalar insertion. The insertion time impacts the overall execution for this distribution as the values are unique and insertion is executed for every values.

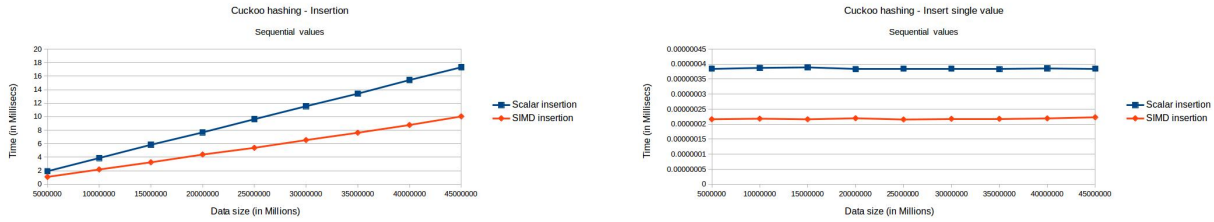


a) Total insertion time b) Insert time - individual value

Figure 5.1: Cuckoo hashing insertion - dense unique random values

Sequential values

Figure 5.2(a) details the insertion of sequential values. The chart shows a similar runtime as dense unique random distribution. Figure 5.2(b) plots the insertion time taken to insert single value. This time is a constant with scalar insertion time as 0.2 microseconds and SIMD based insert is 0.4 microseconds. Hence, the speed up of SIMD acceleration is factor of 2 over scalar insertion. The hashing function disperses the insert values and stores them without many swaps.



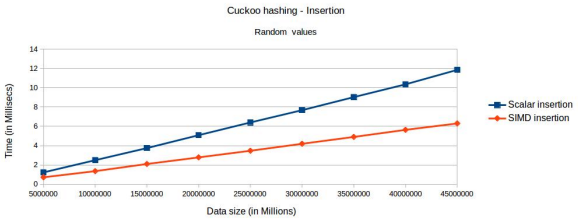
a) Total insertion time b) Insert time - individual value

Figure 5.2: Cuckoo hashing insertion - sequential values

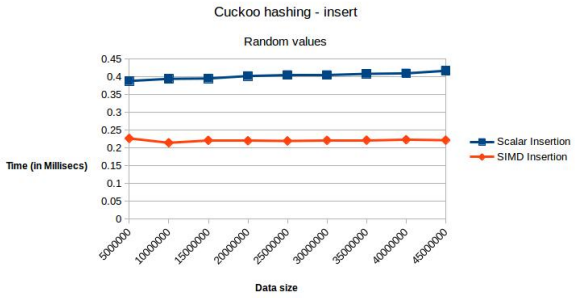
Random values

In case of random distribution of values with constant number of groups the insertion time is given in Figure 5.3. The total time taken to insert all the values (Figure 5.3(a)) shows linear growth. As the number of groups is given as a constant, the values within

each group increases with increase of total data size leading to the linear growth of time. Also, the insert time for single value (Figure 5.3(b)) is showing a minute increase with data size. As the values are randomly generated, the possibility of collision is higher. Thus the insertion runtime increases with unique data.



Total insertion time

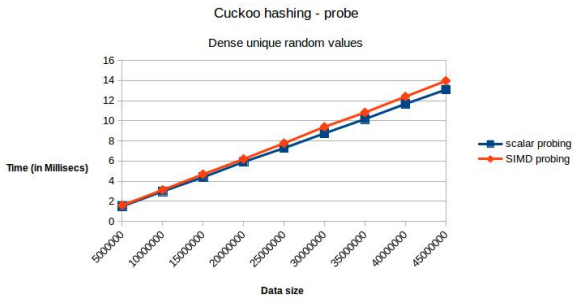


Insert time - individual value

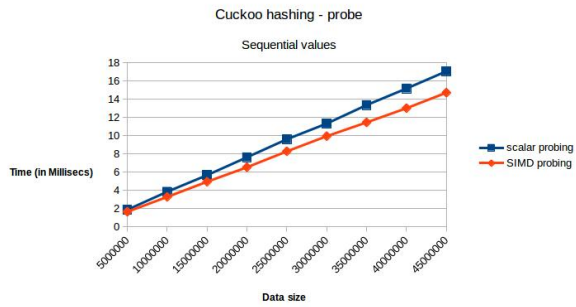
Figure 5.3: Cuckoo hashing insertion - uniform random values

5.2.2 Probing

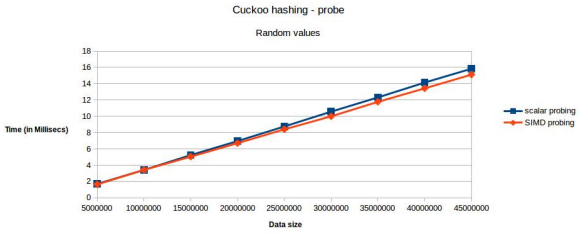
We use probing for path selection in the hashing techniques. Hence, the probing mechanism is done for all the values to be processed. The graph in Figure 5.4 shows a linear growth of runtime with increase in values and is constant for all the distributions.



a) Dense unique random distribution



b) Sequential distribution



c) Uniform random distribution

Figure 5.4: Cuckoo hashing probe - total runtime

We also see from Figure 5.5, the SIMD has nearly the same speed as scalar probing mechanism. This is the maximum speed up that can be achieved as we probe two values in SIMD based probing.

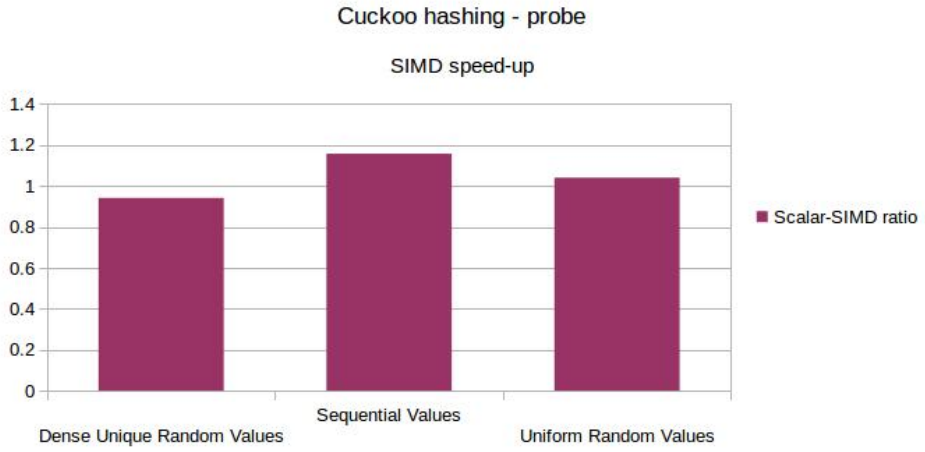


Figure 5.5: Cuckoo hashing probe - SIMD v scalar processing time

Thus, we conclude in this section that SIMD acceleration provides faster results in cuckoo hashing with speed-up in both insertion and probing components. In the following, we analyze the execution results for linear probing mechanism.

5.3 Linear Probing Evaluation

Linear probing has no special mode for insertion. Insertion in linear probing takes constant time as we insert the value in the empty location found by probing. The results plotted for linear probing is shown in Figure 5.6.

We found that SIMD provides very less acceleration compared to scalar linear probing. The SIMD speed-up is nearly the same as its scalar counterpart. This is shown in Figure 5.7.

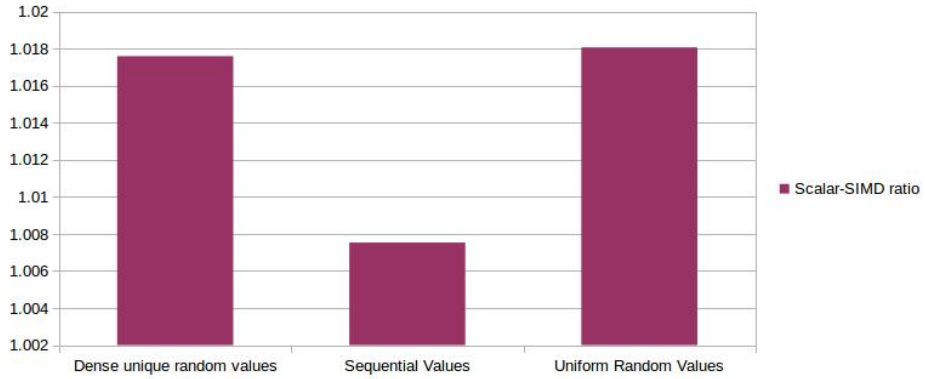
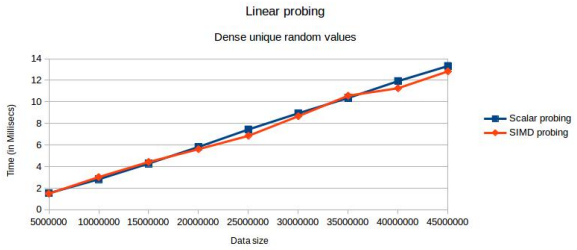
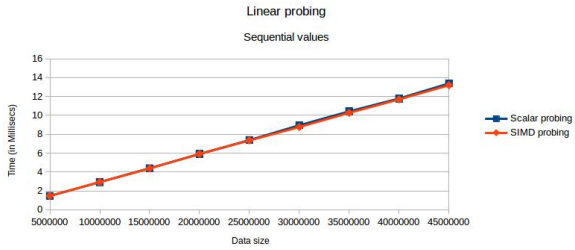


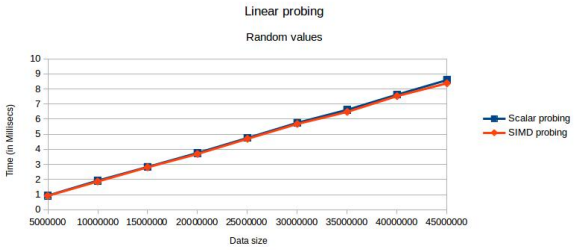
Figure 5.7: Linear probe - SIMD v scalar processing time



a) Dense unique random distribution



b) Sequential distribution



c) Uniform random distribution

Figure 5.6: Linear probe - total runtime

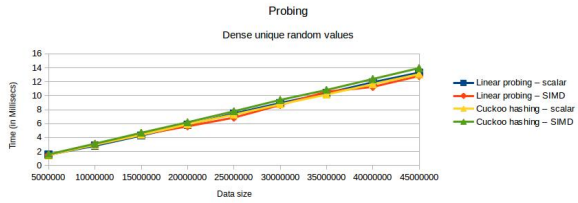
The SIMD version of linear probing suffers from the overhead of preparing the values for SIMD processing. In scalar version, the values are directly processed without any preprocessing step. This leads to faster processing in scalar probing technique than vectorized technique.

5.4 Comparison of Probing in Cuckoo Hashing and Linear Probing

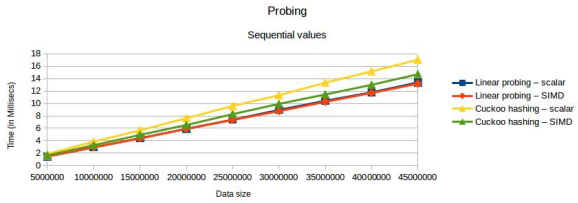
Figure 5.8 depicts the combined graph of linear and cuckoo hash probing. We see probing in both the techniques are nearly having same execution time for distributions with unique values.

In case of distribution with multiple duplicate values, the performance of cuckoo hashing drops. Also, the cuckoo hashing insertion is ignored in the comparison and still it works slower than linear probing. Based on the results, it is clear that linear probing is faster than cuckoo hashing for both scalar and SIMD accelerated executions.

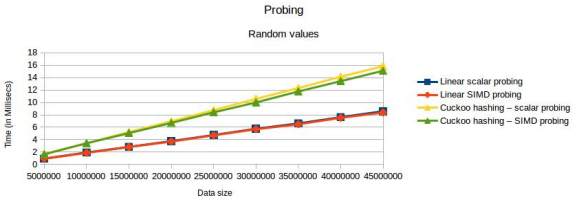
Cuckoo hashing suffers high processing time due to its insertion. While linear probing can store values until the whole hash table is full, cuckoo hashing can form insertion loops with free spaces available in the tables. Also, from the evaluation results it is found that SIMD has good speed-up in cuckoo hashing but SIMD does not affect the execution time in linear probing due to other overheads.



a) Dense unique random distribution



b) Sequential distribution

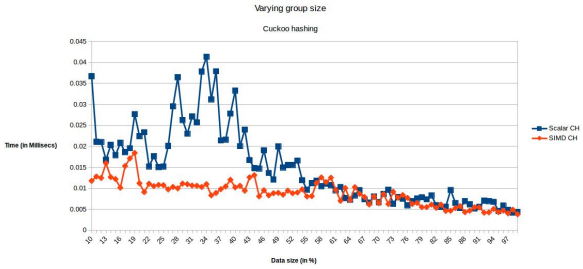


c) Uniform random distribution

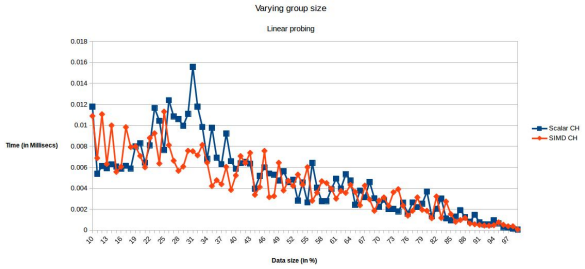
Figure 5.8: Probe time for all techniques

5.5 Impact of Group Size

The hashing techniques are fed datasets with varying group sizes. Increase in number of values in a group decreases the number of inserts being carried out. Figure 5.9 depicts the impact of group size in the hashing techniques.



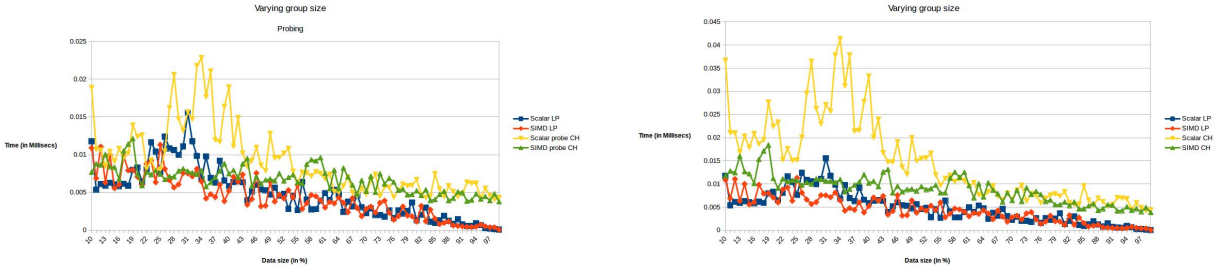
a) Cuckoo hashing



b) Linear probing

Figure 5.9: Impact of group size

From the results, we find that the cuckoo hashing implementation works faster than linear probing for datasets having number of groups less than 50% of total values. The high runtime within this range is due to the high insertion rate in these datasets. In case of linear probing, we find a near constant results for scalar and SIMD versions.



a) Probing b) Probing and insertion

Figure 5.10: Group size with combined hashing techniques

On analyzing the combined results of both the hashing techniques, we find that in case of only probing cuckoo hashing works faster than linear probing for group size less than 54%. This is given in Figure 5.10. The slower linear probing speed is due to the high insertion rate in this range. Comparing the total runtime for both the techniques, we see that the SIMD cuckoo hashing and both scalar and SIMD versions are having same runtime until group size of nearly 55%. The scalar cuckoo hashing takes more time for execution for group size less than 50% as there are many values to be inserted.

5.6 Evaluation Result

In this section, we provide our overall inference on the results. Based on the results, it is evident that the insertion in cuckoo hashing is an overhead. Although the SIMD accelerated approach for insertion provides a considerable speed-up in cuckoo hashing, probing in cuckoo hashing is not providing any significant impact in the execution time. This is due to the extra overheads available in preprocessing the values to fit the SIMD execution.

From the results for linear probing, the execution is auto-vectorized by the compiler for efficient execution. Also, the SIMD version of linear probing also has penalty in adapting the values for SIMD processing. Hence, the linear probing does not have high impact due to SIMD acceleration.

Finally from our analysis of varying group size, we found that the SIMD approach for both the hashing techniques have less impact to varying group size. They provide near constant execution time for various sizes. Also, we inferred that scalar cuckoo hashing is highly influenced by group sizes. We found scalar cuckoo hashing has more insertion time for datasets with less groups. We also found that SIMD version of cuckoo hashing has near constant runtime and SIMD of linear probing decreases linearly with increase in number of values in group.

6. Related Work

In this chapter, we compare our work with others that are similar in performing aggregation operation in databases. Their approaches also use SIMD and other hardware related optimizations over group based aggregation function.

Jiang et al [JA17] use a modified bucket chaining hashing technique to group data and aggregate them on the fly while insertion. In order to eliminate conflicts arising due to SIMD parallelization while insertion and to accommodate values within main memory, they have proposed to add a distinctive offset for each of the SIMD lanes and manipulate data separately [JHL+15] [PRR15]. This approach is also extended for MIMD execution. They propose to use a local hash tables for cache efficient processing [MSL+15] and merging them later [JA17]. They conclude by saying that distinct SIMD lanes help in reducing the conflicts and the efficiency increases linearly with an increasing number of cores. However, merging of local hash tables reduces the overall efficiency of the process. In our approach, we use a centralized hashing table and we use open addressing technique for hashing. We also make use of the advantage of cardinality estimators in knowing the memory size that will be allocated.

Broneske et al. use SIMD to accelerate selection operations [BMS17]. The authors argue that SIMD acceleration influences the operation of aggregation operations. They also provide an evaluation on the impact of SIMD acceleration on the overall performance. In our approach we have gained similar improvement in SIMD acceleration of grouping with aggregation operation. The SIMD accelerated selection can also be extended with our implementation of SIMD accelerated grouped aggregation.

Various SIMD-based aggregation operations are explained in the paper by Zhou and Ross [ZR02]. They detail the performance impact of SIMD on different aggregation operations. These features has to be modified to be included into our grouping based approach as they detail the ways to perform scalar SIMD over input data.

Absalyamov et al. explore the usage of FPGAs for performing group-aggregation queries [ABW+16]. They present an implementation of in-memory hash-based aggrega-

tion accelerated by FPGA. They use hardware multi threading to hide the data transfer latency. If the data exceeds memory, data is chunked and agggregation is performed on individual chunks. Finally, the results are merged to get the total result. However, the system has significant processing overhead to perform merging. In our scenario, single hash table is used, so that no merging is performed later.

Ritcher et al. provided an extensive analysis of different hashing techniques and related functions [RAD15]. They performed evaluation on different hashing techniques and presented their results on the criteria to use different hashing techniques and the functions. Our analysis of the hashing techniques are supporting the results from their analysis.

7. Conclusion

In this thesis, we explored the ways to perform grouped aggregation queries faster. We detailed about the delays present in the pipelines for grouped aggregation. To reduce these delays, we explored the use of code optimizations strategies in the execution of grouped aggregation processing.

In the work, we provided an overview of general query execution in a DBMS engine. We detailed the different modules used in query processing. The pipeline processing for computing results are also detailed along with the different processing models. We also provided an overview about pipeline breakers and their impact in execution time. Pipeline breakers require all the data in the pipeline to compute results. This potentially stops the flow of data in the pipeline. The pipeline breakers present in the hash based aggregation strategy are due to collision of data while hashing. We argued in this work, that the stalls in the pipeline can be reduced using SIMD acceleration as a code optimization.

To perform SIMD optimization on hash based aggregation strategy, we used cuckoo hashing and linear probing. We detailed the insertion and probing mechanism of cuckoo hashing and also the method's indeterministic runtime for inserting a value. We also explained about the insertion cycle problem available in cuckoo hashing. Then, we provided details about the insertion and probing mechanism of linear probing.

These hashing techniques are modified to perform SIMD accelerated execution. The results in this thesis are as follows:

- We argued that insertion is done by probing. The process to insert and probe a value in the hash table are mainly the same. We provided in the work a novel way to combine probing and insertion into one step. We explained that based on the availability of the probe value either insertion or aggregation can be performed.

- We provide in this work, the process flow while using SIMD in probing a value. We explained the process of performing probing multiple values using SIMD for the two techniques.
- We also provide ways to perform aggregation over the values using SIMD. We explain the ways to adapt the scalar execution of aggregation technique to vectorized execution.

For our experimentation, we used three different distributions. The dense unique random and sequential data distributions have distinct values and they had significant impact in insertion time for cuckoo hashing. In case of linear probing, all the three distributions had the same impact for both SIMD and scalar execution. Overall, the SIMD acceleration provided 2x speed up for probing in cuckoo hashing but, linear probing had not much acceleration from SIMD. Also, based on our evaluation with groups we found that the probing in cuckoo hashing is faster with groups less than 50% of total size. Moreover, we inferred from our results that SIMD cuckoo hashing along with scalar and vectorized cuckoo hashing have nearly the same execution time.

8. Future Work

This work can be extended by using different hashing techniques, for example those shown in [RAD15]. Based on the characteristics of the data available in the database, the hashing techniques can be selected. The SIMD acceleration of these hashing techniques can also be analyzed for these techniques.

This work can also be extended to perform other aggregation functions. The implementation of various DBMS functions is explained in [ZR02]. By adapting these instructions, different aggregation functions can be performed in the hash tables. Also, the overhead for SIMD preprocessing can be reduced by modifying the data structure and reducing the number of instructions used.

The hash based aggregation approach can be performed using other hardware related accelerations. This approach can be modified to be executed in GPU and FPGAs and can be optimized for efficiency by exploiting the hardware related instructions.

The SIMD accelerated approach to perform grouped aggregation can be coupled with selection predicate strategy given in [BMS17]. This predicate selection approach can be present below grouped aggregation in the query pipeline.

Bibliography

[ABW+16] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J. Halstead, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-accelerated group-by aggregation using synchronizing caches. *Proceedings of the 12th International Workshop on Data Management on New Hardware - (DaMoN '16)*, pages 1–9, 2016. (cited on Page 51)

[ADHW99] Anastassia Ailamaki, David J. Dewitt, Mark D. Hill, and David A. Wood. DBMSs On A Modern Processor : Where Does Time Go ? *Proceedings of the VLDB Endowment*, 1394:266–277, 1999. (cited on Page 1)

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. *Proceedings - International Conference on Data Engineering*, pages 466–475, 2007. (cited on Page 13)

[BMS17] David Broneske, Andreas Meister, and Gunter Saake. Hardware-Sensitive Scan Operator Variants for Compiled Selection Pipelines. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2017. (cited on Page 1, 51, and 55)

[CLRS91] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. *The Journal of the Operational Research Society*, 42(9):816, 1991. (cited on Page 18)

[Cod70] Edgar. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. (cited on Page 6)

[DG85] David J. Dewitt and Robert H Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the VLDB Endowment*, VLDB '85, pages 151–164. VLDB Endowment, 1985. (cited on Page 17)

[Dil14] Michael B Dillencourt. Cuckoo Hashing. (1):1–17, 2014. (cited on Page 3)

[DK97] Slobodanka Djordjević-Kajan. *Fundamentals of database systems*, volume 28. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997. (cited on Page 1, 5, 6, 10, 11, 12, 15, and 16)

[DK12] Michael Drmota and Reinhard Kutzelnigg. A Precise Analysis of Cuckoo Hashing. *ACM Trans. Algorithms*, 8(2):11:1—-11:36, apr 2012. (cited on Page 21)

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database system implementation*, volume 653. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2000. (cited on Page 1, 4, 5, 6, 11, and 16)

[Gra90] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *ACM SIGMOD Record*, volume 19 of *SIGMOD '90*, pages 102–111, New York, NY, USA, 1990. ACM. (cited on Page 13)

[GSE+94] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. *ACM SIGMOD Record*, 23(2):243–252, 1994. (cited on Page 44)

[JA17] Peng Jiang and Gagan Agrawal. Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation. *Proceedings of the International Conference on Supercomputing*, pages 24:1—-24:11, 2017. (cited on Page 3, 23, and 51)

[JHL+15] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proceedings of the VLDB Endowment*, 8(6):642–653, 2015. (cited on Page 51)

[Knu97] Donald E. Knuth. *The art of computer programming, volume 1: Fundamental algorithms. Redwood City.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. (cited on Page 20 and 33)

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. *Proceedings of the VLDB Endowment*, pages 212–223, 1980. (cited on Page 3 and 20)

[MC86] Ian J. Munro and Pedro Celis. Techniques for Collision with Resolution in Hash Tables Open Addressing. *Proceedings of ACM Joint Computer Conference*, pages 601–610, 1986. (cited on Page 2 and 20)

[MKB09] Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database architecture evolution. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, 2009. (cited on Page 13)

[MSL+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-Efficient Aggregation. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 1123–1136, 2015. (cited on Page 23 and 51)

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011. (cited on Page 1 and 15)

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. *Sigmod*, pages 1493–1508, 2015. (cited on Page 24 and 51)

[RAD15] Stefan Richter, Victor Alvarez, and Jens Dittrich. (P1) A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment*, 9(3):96–107, 2015. (cited on Page 3, 52, and 55)

[Ros07] Kenneth A. Ross. Efficient hash probes on modern processors. In *Proceedings - International Conference on Data Engineering*, pages 1297–1301, 2007. (cited on Page xi, 3, 27, 28, and 31)

[Smi04] Peter Smith. *Applied Data Structures with {C++}*. Jones and Bartlett Publishers, Inc., USA, 2004. (cited on Page 33)

[SZ96] Anoop Sharma and Hansjorg Zeller. Hash-based database grouping system and method, apr 1996. (cited on Page 2, 4, and 27)

[ZB12] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012. (cited on Page ix and 14)

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. *International Conference on Management of Data*, page 145, 2002. (cited on Page 24, 51, and 55)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 19. September 2017