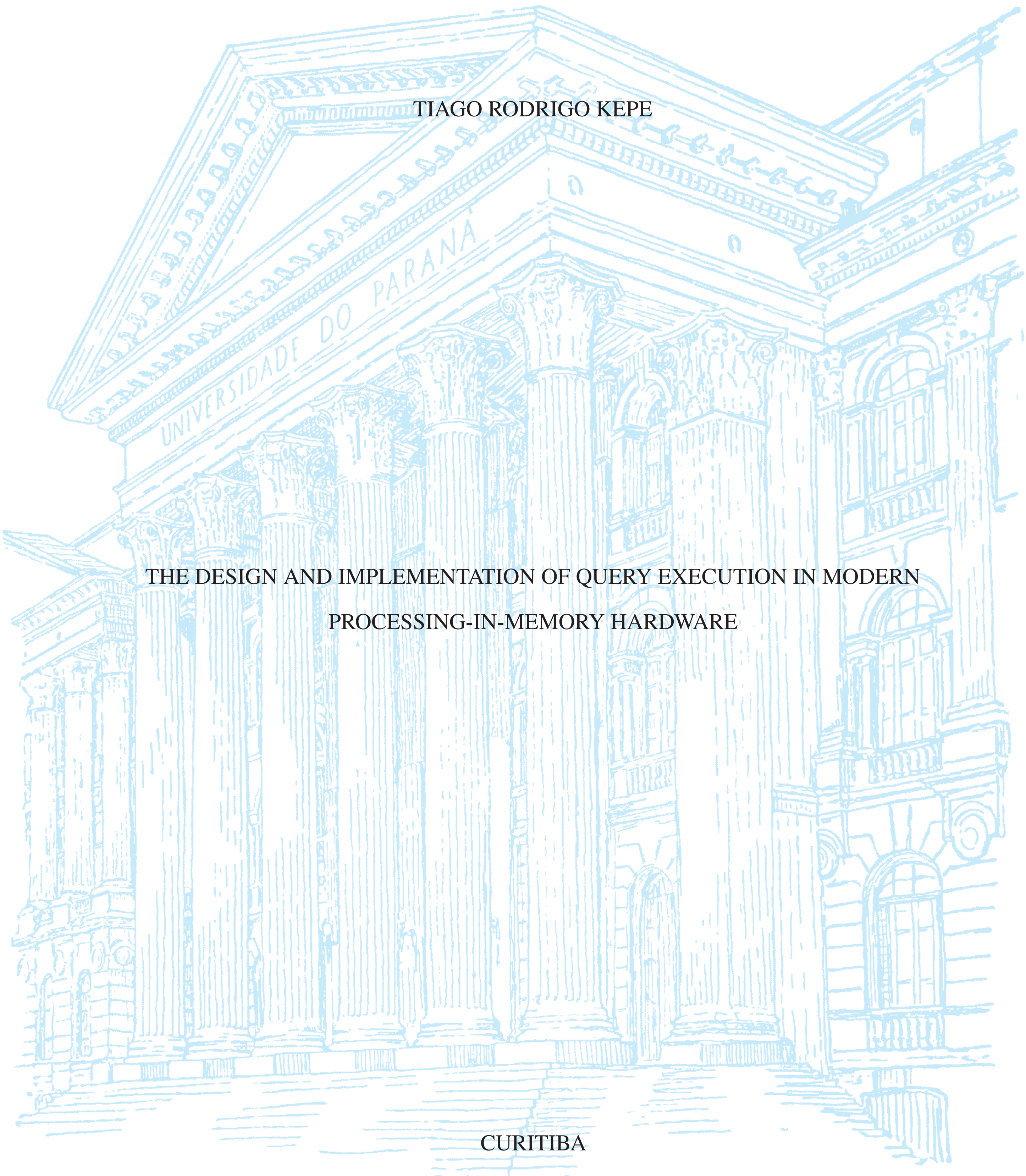
UNIVERSIDADE FEDERAL DO PARANÁ



TIAGO RODRIGO KEPE

THE DESIGN AND IMPLEMENTATION OF QUERY EXECUTION IN MODERN PROCESSING-IN-MEMORY HARDWARE

CURITIBA

2019

TIAGO RODRIGO KEPE

THE DESIGN AND IMPLEMENTATION OF QUERY EXECUTION IN MODERN PROCESSING-IN-MEMORY HARDWARE

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Eduardo Cunha de Almeida.

Coorientador: Prof. Dr. Marco Antonio Zanata Alves.

CURITIBA

2019

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

K38d Kepe, Tiago Rodrigo
The design and implementation of query execution in modern processing-in-memory hardware [recurso eletrônico] / Tiago Rodrigo Kepe. – Curitiba, 2019.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2019.

Orientador: Eduardo Cunha de Almeida. Coorientador: Marco Antonio Zanata Alves.

1. Gerenciamento de memória (Computação). 2. Big data. 3. Banco de dados. I. Universidade Federal do Paraná. II. Almeida, Eduardo Cunha de . III. Alves, Marco Antonio Zanata. IV. Título.

CDD: 005.435

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **TIAGO RODRIGO KEPE** intitulada: **The Design and Implementation of Query Execution in Modern Processing-in-Memory Hardware**, sob orientação do Prof. Dr. EDUARDO CUNHA DE ALMEIDA, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.
A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 20 de Dezembro de 2019.

[illegible]
EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora

[illegible]
PHILIPPE OLIVIER ALEXANDRE NAVAUX
Avaliador Externo (UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL)

[illegible]
MARCO ANTONIO ZANATA ALVES
Coorientador (UNIVERSIDADE FEDERAL DO PARANÁ)

Daniel Oliveira
DANIEL ALFONSO GONCALVES DE OLIVEIRA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

[illegible]
FABIO ANDRE MACHADO PORTO
Avaliador Externo (LABORATÓRIO NACIONAL DE COMPUTAÇÃO CIÊNTÍFICA)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

To my Lord who has gave me strength during all my years of study, especially during my doctoral.

"...
LORD has helped 'me' to now. "
1 Samuel 7:12

ACKNOWLEDGEMENTS

Firstly, I express my gratitude to Prof. Marco Zanata for insightful research conversations about computer architecture. Above all, my most sincere gratitude to Prof. Eduardo C. de Almeida for the guidance, patience, and continuous support over the last decade; my very thank you, I could not imagine a better advisor and mentor for my Ph.D., Master's and undergrad studies.

I would also like to thank all the professors and lab colleagues (LBD and HyPES) at UFPR, in special to Prof. Marcos Castilho and Daniel Weingaertner for the opportunity to join C3SL, one of the best places that I have worked and researched. I also remember the good talks and conference trips with Edson Ramiro, Simone Dominico, Eduardo Pena, and the remote ones in the Netherlands (CWI).

A special thanks to all my family. I would like to express my gratitude to my parents Rose and Julio for the unconditional support and wise advice, my brothers, and also my wife's family to the prayers and to believe in me.

Last but not least, I would like to express my gratitude to my beloved wife Bruna for standing beside me throughout my career and studies. You always believed in me from the early days of my baccalaureate, when I was still working at dawn, then came my master's degree and now doctorate; thank you very much, Bruna D. Kepe. Words are not enough to express how much I love you! Also, to my lovely son Paulo Vinícius, who was conceived at the end of my master's degree, thank you "Daddy's Didi" for being a beam of light in my life, you are one of the biggest inspirations to propel me beyond the edges.

RESUMO

Os sistemas modernos de processamento de consultas foram projetados com base em modelos de arquitetura centrados na computação. No entanto, o rápido crescimento de "*big data*" intensificou o problema de movimentação de dados ao realizar o processamento analítco de consultas: grandes quantidades de dados precisam passar pela memória até a CPU antes que qualquer computação ocorra. Portanto, esses sistemas são afetados pela movimentação de dados que degrada severamente o desempenho e exige muita energia durante a transferência de dados. Estudos recentes sobre a carga de trabalho do Google mostraram que cerca de 63% de energia é gasta em média na movimentação de dados. Para resolver esse problema oneroso, propomos explorar as arquiteturas Processamento-em-Memória (PIM) que invertem o processamento de dados tradicional, enviando a computação para a memória repercutindo no desempenho e na eficiência energética.

Nesta tese, demonstramos empiricamente que a movimentação de dados exerce grande influência nos sistemas de banco de dados atuais e identificamos os principais operadores de consulta que são afetados. Apresentamos um estudo experimental sobre o processamento de operadores de consulta SIMD (*Single Instruction Multiple Data*) em hardware PIM em comparação com processadores x86 modernos (ou seja, usando as instruções AVX512). Discutimos o tempo de execução e a diferença de eficiência energética entre essas arquiteturas. Este é o primeiro estudo experimental, na comunidade de bancos de dados, a discutir as compensações entre tempo de execução e consumo de energia entre PIM e x86 nos sistemas atuais de execução de consultas: materializado, vetorizado e *pipelined*. Como resultado, nós introduzimos um novo sistema híbrido de processamento de consultas PIM-x86 SIMD que incita novos desafios e oportunidades. Além disso, também discutimos os resultados de um escalonador de consultas híbridas ao intercalar a execução dos operadores de consultas SIMD entre o hardware de processamento PIM e x86. Em nossos resultados, o plano de consulta híbrido reduziu o tempo de execução em 45%. Também reduziu drasticamente o consumo de energia em mais de 2 vezes em comparação com os planos de consulta específicos para cada *hardware*.

Palavras-chave: Execução de Consulta 1. Processamento em Memória 2. Escalonador de Consulta Híbrido 3. Eficiência Energética 4.

ABSTRACT

Modern query execution systems have been designing upon compute-centric architecture models. However, the rapid growth of “big-data” intensified the problems of data movement, especially for processing analytic applications: Large amounts of data need to move through the memory up to the CPU before any computation takes place. Therefore, analytic database systems still pay for the data movement drawbacks that severely degrades performance and requires much energy during data transferring. Recent studies on Google’s workload have shown that almost 63% of energy, on average, is spent in data movement. To tackle this costly problem, we propose to exploit the up-to-date Processing-in-Memory (PIM) architectures that invert the traditional data processing by pushing computation to memory with an impact on performance and energy efficiency.

In this thesis, we empirically demonstrate that data movement has an impact on today’s database systems yet, and we identify the foremost query operators that undergo it. Therefore, we present an experimental study on processing query Single Instruction Multiple Data (SIMD) operators in PIM compared to the modern x86 processor (i.e., using AVX512 instructions). We discuss the execution time and energy efficiency gap between those architectures. However, this is the first experimental study, in the database community, to discuss the trade-offs of execution time and energy consumption between PIM and x86 in the current query execution models: materialized, vectorized, and pipelined. As a result, a new hybrid PIM-x86 SIMD query execution system is introduced, bringing new challenges and opportunities. Besides, we also discuss the results of a hybrid query scheduler when interleaving the execution of the SIMD query operators between PIM and x86 processing hardware. In our results, the hybrid query plan reduced the execution time by 45%. It also drastically reduced energy consumption by more than 2$\times$ compared to hardware-specific query plans.

Keywords: Query Execution 1. Processing-in-Memory 2. Hybrid Query Scheduler 3. Energy Efficiency 4.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ALU	Arithmetic Logic Unit
AVX	Advanced Vector Extension
AD	Active Disks
API	Application Programming Interface
BAT	Binary Association Table
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DBMS	Database Management System
DDR	Double Data Rate
DSS	Decision Support System
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
ECC	Correcting Code Memory
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HDD	Hard Disk Driver
HIVE	HMC Instruction Vector Extensions
HPC	High-Performance Computing
HMC	Hybrid Memory Cube
HT	Hash Table
IPB	Instruction-Per-Byte
IPC	Instruction-Per-Cycle
ISA	Instruction Set Architecture
JIT	Just-In-Time
LLC	Last Level Cache
LLVM	Low Level Virtual Machine
MAL	MonetDB Assembly Language
MMU	Memory Management Unit
MR	MapReduce
MSHR	Miss Status Handler Register
NDP	Near-Data Processing
NLJ	Nested Loop Join
nvm	Non-Volatile Memory
OoO	Out-of-Order
OLAP	Online Analytical Processing

OLTP	Online Transaction Processing
PCM	Phase Change Memory
PIM	Processing-in-Memory
RVU	Reconfigurable Vector Unit
SIMD	Single Instruction Multiple Data
SiNUCA	Simulator of Non-Uniform Cache Architectures
SSD	Solid-State Drive
SQL	Structured Query Language
SSD	Solid-State Drive
SSE	Streaming SIMD Extensions
TSV	Through-Silicon Via
VC	Vault Controller

LIST OF SYMBOLS

σ	denotes a selection operator
π, Π	denotes a projection operator
Γ	denotes a group by operator
$\bowtie$	denotes a join operator
$\bowtie_{a=b}$	denotes a hash table of a hash join, building from *a* and probing from *b*

CONTENTS

1 INTRODUCTION

Applications based on data analysis need to move large amounts of data between memory and processing units to look for patterns. Computers have relied on this traditional computing-centric processing since the introduction of the Von Neumann model, which detaches the processor core from the main memory. In this model, data movement severely affects performance and energy consumption. Recent studies show that data movement accounts for around 63%, on average, of the total energy consumption and imposes high latencies [1, 2].

The relational Database Management System (DBMS) is the essential component in modern computing environment to support the applications of data analysis. As defined by Silberschatz et al. [3]: *The DBMS is a collection of interrelated data and a set of programs to access those data.* In a DBMS, the query execution subsystem is the fundamental set of programs to support the applications of data analysis. This subsystem interacts with almost every component of a DBMS, like command compiler, concurrency control and recovery manager, but essentially it fetches data from the database and moves this data into memory buffers for processing. Modern DBMSs implement one of the following query execution models: materialized, vectorized and pipelined. However, these query execution models have been implemented only on computing-centric models [4]. The materialization query execution model generates lots of intermediate data that move along the memory hierarchy to process all the operators implemented by users in a query program [5, 6]. The vectorized query execution model tries to exploit the caching mechanism and the CPU processing with a high interpretation overhead of the query program [7, 8]. The pipelined query execution model uses the Just-In-Time (JIT) compilation to fuse query operators of the same pipeline into a monolithic code fragment. Although the authors of [9] call JIT as a data-centric compilation, the query execution is still computing-centric by moving data to the CPU with many adaptations to make better use of the CPU pipeline. In this thesis, we investigate the data-centric model to tackle the data movement problem in query execution systems with logical units integrated closer to the data (inside memory devices), which is called Processing-in-Memory (PIM) [10].

Database engineers have been evaluating PIM approaches with processing components installed in magnetic disks [11, 12, 13], RAM [14], and more recently in flash disks [15, 16, 17]. However, commercial products have not been adopting those approaches for three main reasons: 1) Limitations of the hardware technology; 2) The continuous growth in CPU performance complied to the Moore's Law and Dennard scaling[1]; 3) The lack of a general programming interface that leads to low abstraction level when handling hardware errors.

[1]The Dennard scaling has a parallel to Moore's law in terms of energy consumption. In 1974, Robert H. Dennard et al. postulated that transistor areas get smaller as their power density stays constant.

Recently, PIM architectures came back to the spotlight due to the introduction of Through-Silicon Vias (TSVs). The TSV enables the integration of DRAM dies and logic cells in the same chip area, forming a 3D-stacked memory. Current commercial GPUs already embed the emerging 3D-stacked memories, such as the Hybrid Memory Cube (HMC) [18] and the High Bandwidth Memory (HBM) [19]. However, there has not been any in-depth study of query execution on PIM with SIMD support.

1.1 THE PROBLEM

Modern query execution systems have relied on computing-centric architecture to process database operators. They were designed to extract the best features of modern CPUs (e.g., CPU pipeline and out-of-order instruction execution) and of the caching mechanism. The major downside of those designs is the data movement throughout the memory hierarchy. Data still need to be transferred from memory to the processor within the CPU, which severely degrades performance and requires much energy during data transferring. For instance, in Online Analytical Processing (OLAP), the data movement to validate filters in long-running queries accounts for $40-80\%$ of the execution time in resource stalls, memory stalls, and branch mispredictions [20]. A recent study of the Google's workloads [1] demonstrated that 62.7% of the total system energy spent is due to data movement. This study also presents the potential of PIM reducing around 55% of energy and execution time, on average.

In this thesis, we investigate the impact of data movement of query execution in modern relational DBMS. Figure 1.1 presents the execution time and memory usage when executing the 100 GB TPC-H data analysis benchmark [21] in the MonetDB [22] DBMS[2]. This result highlights the data movement problem that we face in today's DBMS.

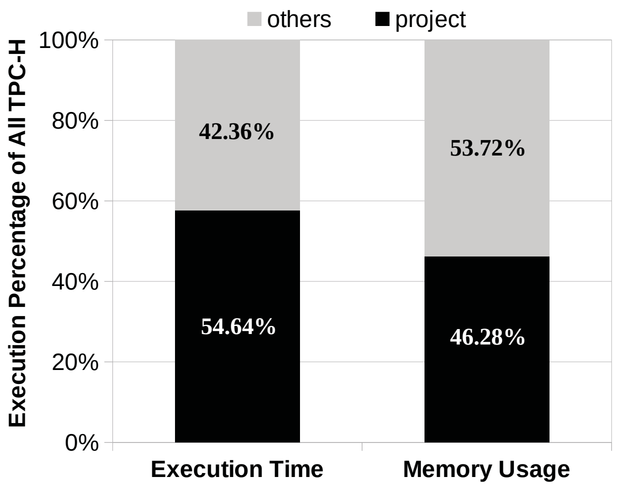


Figure 1.1: The impact of data movement on MonetDB.

[2]The TPC-H is the standard query execution benchmark for data analysis. Further details of the experiment design are presented in Chapter 5.

We highlight the projection operator and include the other operators into the category "others". Just the projection operator is responsible for almost 55% of execution time. The projection operator takes the burden of tuple reconstruction and the materialization of intermediate results with direct impact in data movement with more than 46% of the memory usage compared to all the other query operators. These results are mainly caused by moving data throughout the memory hierarchy. Notice that the other operators also move data around the memory hierarchy as a direct impact of the computing-centric architecture design of the materialization query execution system of MonetDB (also implemented in VoltDB). As we will show in this thesis, this impact is not restricted to the materialized query execution as we also observed in all the other query execution models: vectorized (e.g., SQLServer, VectorWise, Hyrise, DB2) and pipelined (e.g., PostgreSQL, DB2, Oracle, MySQL, SQLite).

1.2 MOTIVATION

The recent trend of PIM promises to tackle the memory and energy wall problems lurking in the data movement around the memory hierarchy. Naturally, we investigate the performance of query operators running on modern PIM as a kickoff to unravel potential advantages and drawbacks. For instance, we evaluated the projection operator and a naive implementation of the join operator (i.e., the Nested Loop Join (NLJ)) in Chapter 7. The preliminary results show distinct performances for both operators. While the projection operator reaches a significant improvement on PIM (almost 6× faster than the x86 processor), the NLJ has an intricate result with better performances on x86 processing.

Those preliminary results point out that the decision to execute database operators on PIM is not trivial. The traditional x86 computing-centric model still has beneficial properties, and there are several opportunities for energy savings with PIM execution. Also, the query execution systems have many operators with a myriad of algorithm implementations. Besides, dataset characteristics (e.g., size and cardinality) shall interfere with the performance of the database operators on each architecture. In this thesis, we investigate *how modern DBMS can embrace PIM in query execution.*

1.3 HYPOTHESIS

To alleviate the data movement problem on modern database systems, we formulate the following hypothesis:

New PIM devices shall mitigate the data movement in Query Execution System, and thus reduce execution time and energy consumption.

1.4 CONTRIBUTIONS

This thesis reveals the importance of rethinking the query execution systems for emergent PIM architectures. We rely on our hypothesis to investigate the following research questions with the specific contributions listed:

1. What is the potential reduction in data movement that PIM devices can provide on query execution systems?

    - We present an investigation of near-data processing approaches on memory devices (e.g., hard disks, flash disks, and main memory). Our goal is to extract some learning of how query execution systems were evaluated over the years (Chapter 3).
    - We explore the selection operator (for its simplicity and relevance) using current PIM capabilities (Chapter 4).

    This work is completed and was published in ADMS@VLDB$_{2018}$ [23].

2. Which is the most relevant group of query operators to evaluate on PIM?

    - We present a study of the most time and memory consuming database operators on MonetDB (Chapter 5).
    - We detail the implementation of query operators on PIM hardware with SIMD support (Chapter 6).
    - We design a new SIMD sorting algorithm to take advantage of SIMD capabilities on x86 (using AVX512 extensions) and PIM devices (Chapter 6).

    This work is completed and was published in PVLDB$_{2019}$ [24].

3. What are the relevant characteristics that impact the decision between PIM and x86 processing?

    - We present a comprehensive performance analysis of the query operators on modern PIM hardware and the traditional x86-style processing regarding the effect of data movement around memory (Chapter 7).
    - We distinguish the trade-offs to process each SIMD operator on top of the materialized, vectorized, and pipelined query execution models (Chapter 7).

    This work is completed and was published in PVLDB$_{2019}$ [24].

4. How can DBMSs coordinate intra-query execution between the CPU and PIM to exploit the potential benefits of each architecture?

    - We propose a classification method based on operator profiles to decide in which architecture to process each operator (Chapter 8).

- We design a PIM-aware query scheduler to exploit the potential of hybrid scheduling of database operators, which interleaves the execution between PIM and x86 processing (Chapter 8).
- We provide heuristics to build a hybrid query plan and discuss the experimental results. Our hybrid scheduler reduced execution time by 35% and 45% when compared to PIM and x86 hardware-specific query plans, respectively. The hybrid scheduler reduced energy consumption by more than 2× compared to the traditional x86 processor (Chapter 8).
- We introduce a new hybrid PIM-x86 SIMD query execution system that brings new challenges and opportunities (Chapter 8).

This work is completed and was published in PhD@VLDB$_{2018}$ [25], SBBD$_{2019}$ [26], and DEXA$_{2019}$ [27].

1.5 ORGANIZATION

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of the architecture of DBMSs and the main query execution models. Chapter 3 presents related work on near-data processing in memory devices. Chapter 4 gives an overview of the new PIM architectures. Chapter 5 delivers our experiment design and introduces the simulator used to evaluate the baseline PIM architecture. Chapter 6 details the SIMD instructions used in the implementation of the database operators. Chapter 7 presents results and analysis of the execution of operators on PIM and x86 architectures. We also evaluate the performance and energy consumption of the distinct query execution model. Chapter 8 introduces the hybrid PIM-x86 SIMD query execution system with results for a hybrid PIM-x86 query scheduler. Chapter 9 discusses the main conclusions of this thesis.

2 DATABASE FOUNDATIONS

In this chapter, we present the foundations of relational databases and detail the modules in which this thesis takes place. A relational database is a collection of related data stored in tabular data structures corresponding to real-world entities (or rows) and their attributes (or columns). There is a distinction between the logical and physical representation of database tables, called "Data Independence"[1]. At a logical level, we are concerned with the conceptual schema of an application and user views. At a physical level, database tables need to be mapped onto one dimensional structures before being stored: rows-by-rows (or row-store) and/or column-by-column (or column-store). Although this thesis focuses on the physical level, more specifically on how to retrieve relational data from modern memory hardware, we need to discuss the fundamental relational operations that come from the model designed at the logical level.

The DBMS is a collection of programs that enables the distinction between logical and physical levels, and provides a way to store and retrieve database information [28]. We begin with an overview of the "Three-schema arquiteture" of DBMS that separates the logical and physical database levels, depicted by Figure 2.1. On the left, there is a schema in three layers: the external, the conceptual, and the internal levels. On the right, the diagram shows the main components of a DBMS. The external interface (e.g., Structured Query Language (SQL)) receives commands for data recovery and the SQL Interpreter transforms them to an internal representation. The *Query Evaluator* generates candidate query plans and chooses one to be executed. During query execution, the *Data Access* module manages the data access recovering data from internal structures in the *Database* layer.

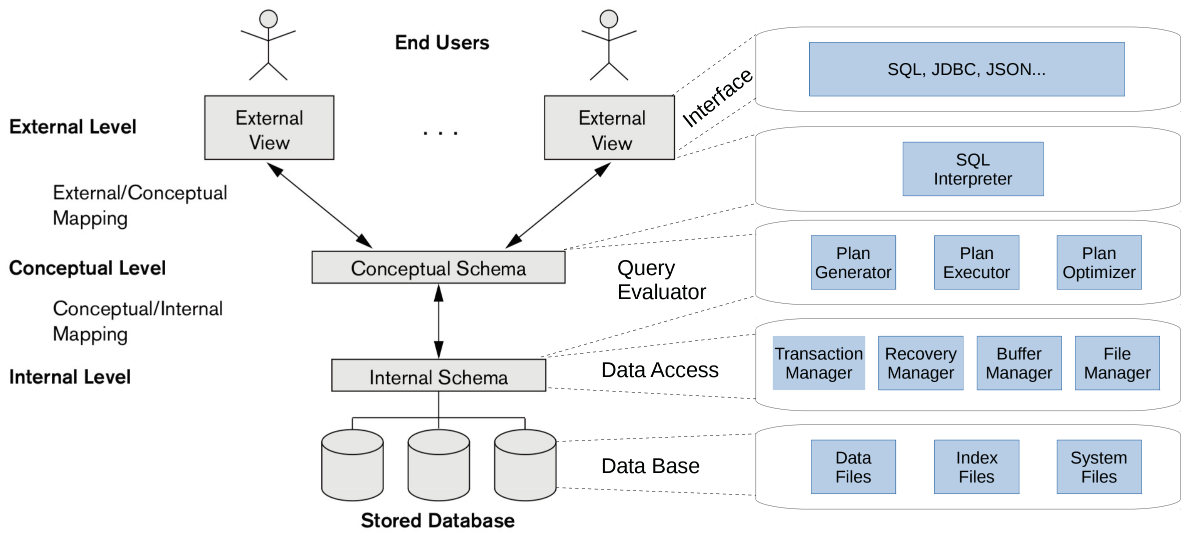


Figure 2.1: A general architecture of a DBMS [28, 29].

[1]"Data Independence" is also defined by [28] *as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level.*

Now, we briefly describe the execution of SQL queries in the DBMS. Further details in the execution of queries are left to the sections of query execution models. Query processing starts with the submission of an SQL query to the DBMS. The *SQL Interpreter* validates the query in lexical, syntactic, and semantic aspects. After the query validation, the *Query Evaluator* generates logical query plans based on the relational algebra. The *Plan Optimizer* chooses the best-estimated plan to be executed and transforms the logical plan into a physical execution plan. The physical plan consists of primitives (functions and algorithms) and dataset information (e.g., indexes, files, and columns) needed to run the plan. Figure 2.2 presents an SQL query and its corresponding query plan from the relational algebra.

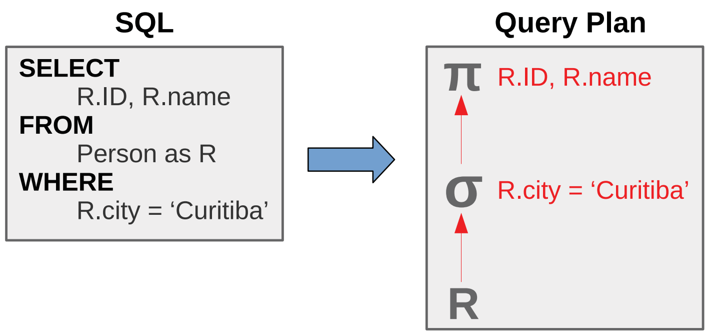


Figure 2.2: An SQL query and its corresponding query plan.

The *Plan Executor* processes the query from the physical plan. The query scheduler (within the Plan Executor) orchestrates the execution of primitives, ordering them, selecting the input dataset, storing intermediate data, and synchronizing the execution among primitives. During the physical plan execution, the scheduler communicates with the *Data Access* layer, consulting the *Buffer Manager* and *File Manager* components. The Buffer Manager contains information about the pages in memory and their addresses. When needed to access data in secondary memory, the File Manager is responsible for that extraction.

As our thesis investigate the execution of queries to retrieve relational data stored in modern PIM hardware, in the next sections, we next expose the relational model, the operators of the relational algebra, and the distinct storage layouts. We finish with the query execution models.

2.1 RELATIONAL MODEL

The relational model proposed by Edgar F. Codd [30] represents databases as a set of relations, which is the conceptual schema of Figure 2.1 and the internal level are the data structures to store the relations. A relation is abstracted to a table and the columns to attributes of the relation. Thereby, a row in the table is a colection of values that can represent an instance of a relation or a relationship.

Formally in the relational model, a row is called a tuple (t), a column is an attribute (A), and a table is defined as a relation (R). Each attribute belongs to an specific domain (D), commonly assigned to a data type.

A schema of the relation R is denoted by $R(A_1,A_2,...,A_n)$, such that R is the name of the relation followed by a list of attributes $A_1,A_2,...,A_n$. Each attribute A_i has a defined domain as $dom(A_i)$.

An instance r from the schema $R(A_1,A_2,...,A_n)$ is denoted by $r(R)$, which is a state of the schema. $r(R)$ is comprised by a set of tuples $r = \{t_1,t_2,...,t_m\}$ and the quantity of tuples is defined as $|R|$. A tuple consists of a sequential list of n values $t = \{< v_1,v_2,...,v_n >\}$, each value v_i belongs to $dom(A_i)$. The i-th value (A_i) of a tuple is denoted as $t.A_i$ or $t[i]$.

For example, let us consider the entity Person with four attributes: ID, Name, Phone, and City. The relational schema for that entity is $Person(ID,Name,Phone,City)$, which contains the name of the relation (Person) and the four attributes. An instance of this schema with four tuples is represented bellow:

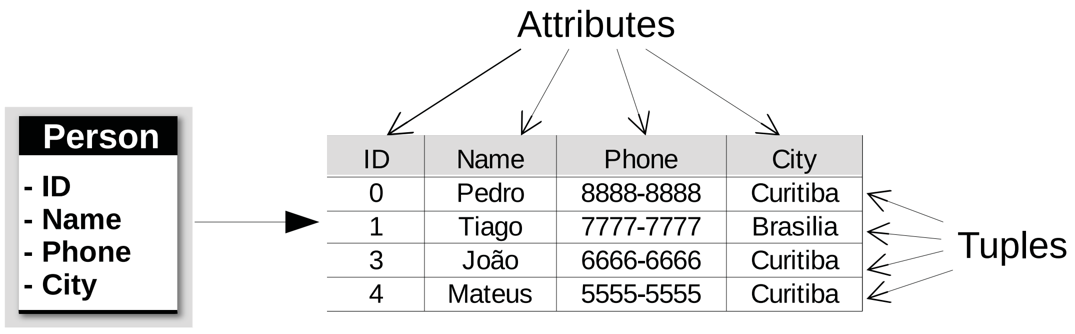
$$r = \{< 0,\ Pedro,\ 88888888,\ Curitiba >, \quad (2.1)$$

$$< 1,\ Tiago,\ 77777777,\ Brasilia >, \quad (2.2)$$

$$< 3,\ Joo,\ 66666666,\ Curitiba >, \quad (2.3)$$

$$< 4,\ Mateus,\ 55555555,\ Curitiba >\} \quad (2.4)$$

Figure 2.3 illustrates that relation schema. Considering the tuple $t_1 =< 0,\ Pedro,\ 8888\text{-}8888,\ Curitiba >$, to access the fourth value (A_4): $t_1.A_4 \equiv t_1[4] = Curitiba$.



ID	Name	Phone	City
0	Pedro	8888-8888	Curitiba
1	Tiago	7777-7777	Brasilia
3	João	6666-6666	Curitiba
4	Mateus	5555-5555	Curitiba

Figure 2.3: The entity Person and an instance of its respective relational schema.

2.2 RELATIONAL ALGEBRA

The relational algebra is a formal language for retrieving data from the relational model, which consists of a basic set of operations that allows the user to specify queries using expressions. An operation produces a new relation as an entry to another one. A chain of operations in relational algebra forms an expression in which the output is also a new relation. Although the expression is unique, other ones can generate the same resulting relation, i.e., there are many options to reach the same result. Based on this principle, DBMSs create different query plans to optimize data recovery.

The relational algebra is based on the set theory, since each relation is defined as a set of tuples ($r = \{t_1, t_2, ..., t_m\}$). Consequently, it has the following operations inherited from set theory: *union*, *intersection*, *difference set*, and *cartesian product*. Other operations were also added to relational databases, such as *selection*, *projection*, and *join*. These operations are associated with operators in SQL queries.

2.2.1 Selection Operation

In relational algebra, the selection operation ($\sigma_{<p>}\ (R)$) selects a subset of the original relation (R) according to a condition. The resulting subset is a new relationship with the number of tuples $|\sigma_{<p>}\ (R)|\ \leq\ |R|$. The selection condition is also known as a predicate ($_{<p>}$) that is a boolean expression [28] composed by clauses of the form:

- <*attribute name*> <*operator of comparison*> <*constant value*>, or
- <*attribute name*> <*operator of comparison*> <*attribute name*>

Such that <*attribute name*> is an attribute of *R*, <*operator of comparison*> is one of the operators:$\{=, <, \leq, >, \geq, \neq\}$ and <*constant value*> is a constant belonging to the domain of the attribute. Boolean operators (e.g., AND, OR, and NOT) connect multiple predicates to form a generic selection condition.

The following is an example of applying the selection operation on the relational schema of Figure 2.3 and the selection condition filters the PERSON whose name is Tiago. Code 2.1 presents that selection operation, which corresponds to the WHERE clause of the SQL in Code 2.2.

$$\sigma_{Name=Tiago}\ (Person)$$

Listing 2.1: Example of Selection.

```
SELECT *
FROM   Person
WHERE Name = "Tiago"
```

Listing 2.2: Selection in a SQL query.

It is important to note that selection is a commutative operation, i.e.:

$$\sigma_{<p_1>}\ (\sigma_{<p_2>}\ (R))\ =\ \sigma_{<p_2>}\ (\sigma_{<p_1>}\ (R))$$

Therefore, multiple selections on the same relation shall be grouped into a single operation and the predicates combined through the conjunction operator (AND), such that:

$$\sigma_{<p_1>}\ (\sigma_{<p_2>}\ (...\ (\sigma_{<p_n>}\ (R)...)\ = \sigma_{<p_1>AND<p_2>AND...<p_n>}\ (R)$$

2.2.2 Projection Operation

The projection operation of the relational algebra picks a subset of attributes from a relation (R), forming a new relation with only those attributes and their values. This operation corresponds to the following general form:

$$\Pi_{<attribute\ list>}\ (R)$$

Π is the symbol to denote the projection operator, <attribute list> is a subset of attributes contained in R. Code 2.3 presents an example of the projection operation, which corresponds to the SELECT command in SQL, as exemplified by Code 2.4. This is an example of applying the projection operation to the relational schema of Figure 2.3, where the names and cities are projected from the relation Person.

```

$\Pi_{Name,City}\ (Person)$
```

Listing 2.3: Example of Projectin.

```
SELECT Name, City
FROM   Person
```

Listing 2.4: Projection in a SQL query.

Multiple projection operations on the same relation can be abbreviated, once the attribute list of one projection is contained in another one, i.e.:

$$\Pi_{<list1>}\ (\Pi_{<list2>}\ (R))\ =\ \Pi_{<list1>}\ (R), \text{ if and only if, } {}_{<list2>}\ \subseteq\ {}_{<list1>}$$

Usually, the selection and projection operations occur together in expressions of the relational algebra and SQL queries, as exemplified in Codes 2.5 and 2.6. In addition, it is possible to perform some aggregations together with projections, such as MIN, MAX, SUM, AVG, COUNT, arithmetic operations $(+,-,*,/)$ etc.

```

$\Pi_{Name,City}\ (\sigma_{Name=Tiago}\ (Person))$
```

Listing 2.5: Example of Projection and Seletion together.

```
SELECT Name, City
FROM   Person
WHERE name = "Tiago"
```

Listing 2.6: Projection and Selection in a SQL query.

2.2.3 Join Operation

Join is a binary operation that combines tuples of two relations into a single one. It establishes the relationship between relations by comparing the join attributes and generating a set of tuples that match these attributes.

The general formula to join two relations $R(A_1,A_2,...,A_n)$ and $S(B_1,B_2,...,B_m)$ is:

$$R\ \bowtie_{<join\ condition>}\ S$$

The result of the join operation is a new relation Q with the tuples that satisfy the join condition. The tuples of this relation are formed by the concatenation of tuples in R and S, $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, in this order, such that Q contains $n + m$ attributes. When performing the join operation, as the join condition is satisfied, the tuple of R is matched with the corresponding tuple in S. Multiple join conditions are connected through conjunctions: $<condition1>$ AND $<condition2>$ AND ... AND $<conditionN>$, a condition is a predicate of the form $A_i\ \theta\ B_j$, such that A_i is an attribute of R and B_j is an attribute of S, and θ is an of the comparison operators: $\{<, \leq, >, \geq, =, \neq\}$.

To exemplify this operation, let us consider the relation Employee (Person_ID, Salary) that contains employee information for a company, which relates to the Person relation. Code 2.7 and Code 2.8 present the join expression and the SQL query, respectively. Table 2.1 exemplifies the result of the operation.

$$Person\ \bowtie_{ID\ =\ ID\_Person}\ Employee$$

Listing 2.7: Example of Join.

```
SELECT *
FROM Person JOIN Employee
ON ID=ID_Person
```

Listing 2.8: Join in a SQL query.

ID	Name	Phone	City	ID_Person	Salary
0	Pedro	8888-8888	Curitiba	0	$2.000,00
1	Tiago	7777-7777	Brasília	1	$15.000,00
2	João	6666-6666	Curitiba	2	$5.000,00
3	Mateus	5555-5555	Curitiba	3	$3.500,00

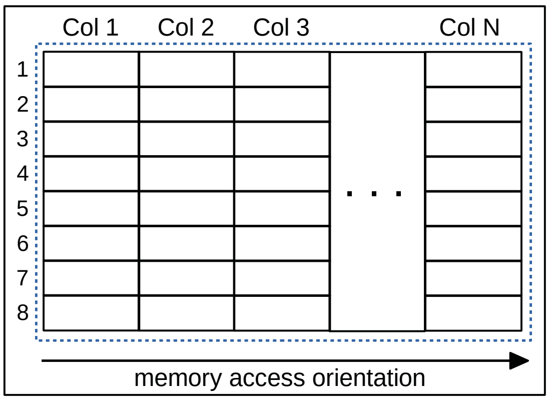
Table 2.1: The join of the relations Person and Employee.

2.3 STORAGE LAYOUT

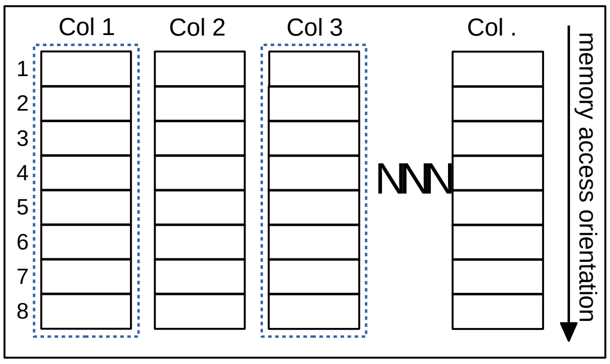
The storage layout is a fundamental component of a DBMS (within the DataBase layer, see Figure 2.1). Studies [31] have pointed out that the way data are arranged directly influences the query processing. In a nutshell, a table can be stored by its rows or columns. The row-store layout (also called linear or row-by-row layout) stores rows contiguously in the data files. This means that a row is entirely stored in a file and read from a file. The column-store layout (also called columnar or column-by-column layout) stores columns contiguously in the data files. This means that a row is partitioned by its columns that are entirely stored in a file and read from a file.

Figure 2.4 depicts the row-store and column-store layouts. To access two columns, e.g., columns $Col1$ and $Col3$, a column-store system straight load the required columns from memory (the dashed lines in Figure 2.4(b) may represent a file or a data structure with each column stored separately). At the same time, a row-oriented system has to load the full row with many useless columns (the dashed lines in Figure 2.4(a)), wasting memory throughput. As observed in

Figure 2.4 and demonstrated by Abadi D. et al. [32] the row-store layout is suitable for Online Analytical Processing (OLTP) workloads, because transactions are written to disk without any row partitioning with low input/output operation overhead. In contrast, the column-store needs to partition each column from a row to different data structures with high input/output operation overhead. However, the columnar layout is suitable for OLAP, because complex queries fetch from disk only the required columns without wasting memory throughput. Although we briefly describe both storage layouts in the following sections, this thesis focuses on the column-store that is the layout of choice of large-scale OLAP applications of contemporary BigData systems.



(a) Row layout.



(b) Column layout.

Figure 2.4: Row vs. Columnar layout. The arrows show the data access orientation, and the dashed blue lines depict the access to two columns on both layouts.

2.3.1 Row-Store Layout

The row-store layout in the relational model is defined as follow:

Linear $\rightarrow r = (t_1, t_2, ..., t_m)$, r is an instance of a relation and t_i is a tuple with n values: $t_i =< v_1 \Rightarrow v_2 \Rightarrow ... \Rightarrow v_n >$, such that $v_j \Rightarrow v_{j+1}$ means that the values of a tuple are stored continuously (adjacent), $\forall\ j \mid 1 < j < n$. Therefore, the j-th value of a tuple is accessed by $r.t_i[j]$ and, consequently, the next value $r.t_i[j+1]$ will be adjacent in memory.

The DBMS applies the *N-ary Storage Model* (NSM) [33] for physical storage, which divides the relation (table) into blocks, storing row-by-row. Thus each row contains all its columns (values) contiguously arranged in memory.

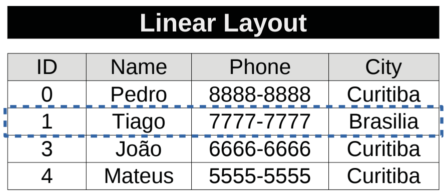
This arrangement requires a few memory operations to read an entire tuple from memory, case $|t_i|$ is less than or equal to one cache line, all values of a tuple will be on the same cache line. As the values of a tuple (see Section 2.1) represent an instance of an entity, the row-store layout has proven conducive to workloads that require information at the entity granularity. Therefore, row-oriented systems can fetch all information of an entity, update and even remove it efficiently using a few read and/or write memory operations. This behavior is characteristic of OLTP workloads.

Let us consider the SQL Code 2.9 that selects the information of the Person which *ID* is 1. In the row-store layout, just a single memory read operation is enough to obtain all information to that person, as Figure 2.5 depicts on the dashed line.

```

SELECT * FROM Person WHERE ID = 1
```

Listing 2.9: A OLTP query.



Linear Layout

ID	Name	Phone	City
0	Pedro	8888-8888	Curitiba
1	Tiago	7777-7777	Brasilia
3	João	6666-6666	Curitiba
4	Mateus	5555-5555	Curitiba

Figure 2.5: OLTP query on the row-store layout.

On the other hand, the row-store layout is not suitable for OLAP workloads. Analytics queries require to access a few columns. Using that layout, however, many memory operations are wasted when accessing useless columns for the query, augmenting data movement. Considering the OLAP query in the left side of Figure 2.6, it selects the tuples in which the *City* column is equal to "Curitiba" and computes the total. This query reads all the columns of the table (see Figure 2.6, dashed lines), loading into the memory hierarchy the unneeded columns: *ID*, *Name*, and *Phone*, but the query just requires the *City* column.

2.3.2 Colum-Store Layout

The column-store layout in the relational model is defined as follow:

Columnar $\rightarrow r = (t_1, t_2, ..., t_m)$, r is an instance of a relation and t_i is a tuple with n values: $t_i =< v_1 \iff v_2 \iff ... \iff v_n >$, such that $v_j \iff v_{j+1}$ means that the values of a tuple are **not** stored continuously (adjacent) in memory, $\forall\ j \mid 1 < j < n$. In contrast, values from the same column are stored contiguously, i.e., $t_i[j] \Rightarrow t_{i+1}[j]$, $\forall\ t_i \mid 1 < i < m$.

Therefore, we define a relation instance in the columnar layout as follow:

Columnar $\rightarrow r = (C_1, C_2, ..., C_n)$, n is the degree of the relation, a C_i is a column that contains all values for the attribute A_i. The columns are stored separately, values within the same column are adjacent and arrange according with the order of its tuples. The i-th value of a tuple t_j is accessed by $C_i[j]$, such that $C_i[j] \Rightarrow C_i[j+1]$ are coalescing values, $\forall\ j \mid 1 < j < m$. Thus, the j-th tuple is defined as $t_j = (C_1[j], C_2[j], ..., C_n[j])$.

The columnar layout is inspired on the *Decomposition Storage Model* (DSM) [34], which vertically partitions the tables, then each column is stored separately, as depicted in

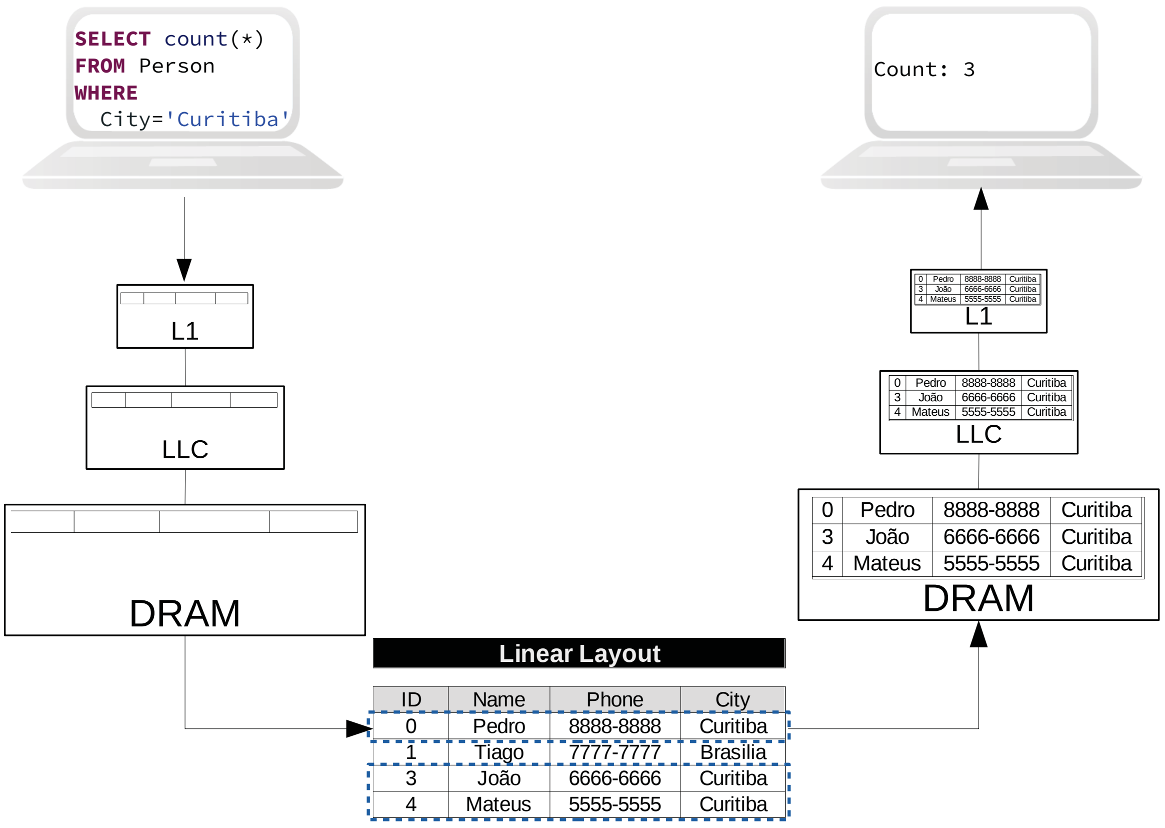


Figure 2.6: OLAP query on the *row-store* layout.

Figure 2.4(b). Thefore, OLAP queries take advantages of this layout by loading from memory only the required columns. Figure 2.7 illustrates the behavior of OLAP queries on the columnar layout, then just the column *City* is load from memory, saving memory throughput and energy.

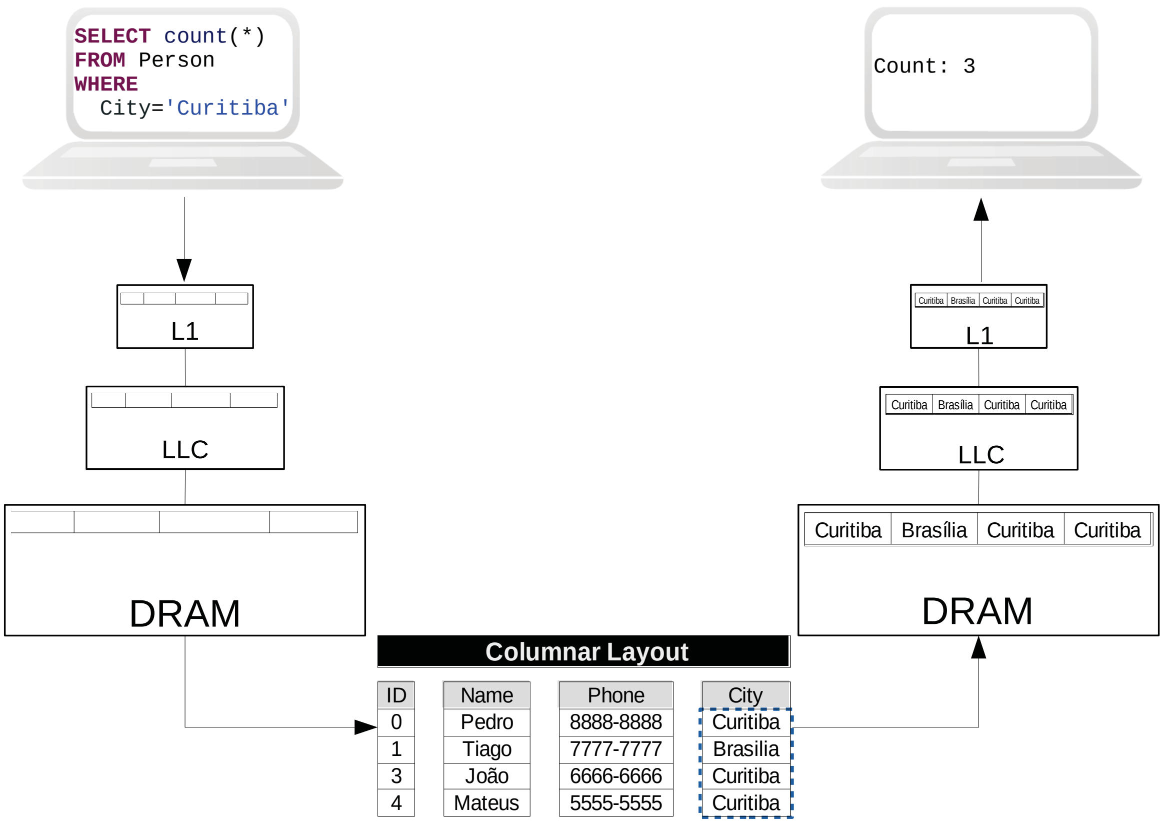


Figure 2.7: OLAP query on the *columnar* layout.

2.4 QUERY EXECUTION MODELS

This section introduces the current query execution models implemented as the query execution subsystem in modern DBMSs. In [35], Daniel J. Abadi describes four main properties of OLAP workloads that can be used to guide the implementation of a query execution system. They are: (1) **unpredictable**, OLAP queries tend to be more exploratory in nature rather than defined to specific business tasks, as transactions; (2) **longer lasting**, OLAP queries may execute in hours moving large volumes of data in contrast to transactions that are expected to execute in milliseconds with small amount of data; (3) **read-mostly**, OLAP queries are more Read-Oriented than Write-Oriented. Typically, only batch writes are executed in OLAP databases; and (4) **attribute focused**, OLAP queries are meant to process summaries of lots of entities (or rows). In this case, summaries aggregate records of a few number of columns.

Next, we describe the Volcano query execution model, followed by the materialization, vectorized, and pipelined model.

2.4.1 The Volcano Query Execution Model

Most of the query execution systems have relied on the iterator model implemented in the Volcano system [36], like in PostgreSQL, MySQL and SQLite DBMSs. In this model, operators in a query plan exchange tuples through a standard iterator interface with the *open*, *next*, and *close* procedures in a producer and consumer design way, also called tuple-at-a-time processing. This interface isolates the operators, and thus each one manages all issues of control, internally.

Figure 2.8 shows the iteration model in the execution of a query (see Listing 2.10). Each operator in a query tree emits tuples to the next operator, which consumes a tuple through the invocation of the *next* procedure. Coalescing operators in a query tree exchange data (tuples) through a shared buffer (queue), the producer operator pushes data into the buffer for a pull consumer. The DBMS uses a semaphore to synchronize such exchanging. In OLAP, the Volcano-like tuple-at-a-time execution incurs in a high runtime interpretation overhead and waste of memory throughput by loading the entire tuple, even useless columns [37, 32].

Listing 2.10: Simple SQL query statement with two projections and one selection operations.

```sql
SELECT
   R.ID, R.name
FROM
   R
WHERE
   R.city = 'Curitiba';
```

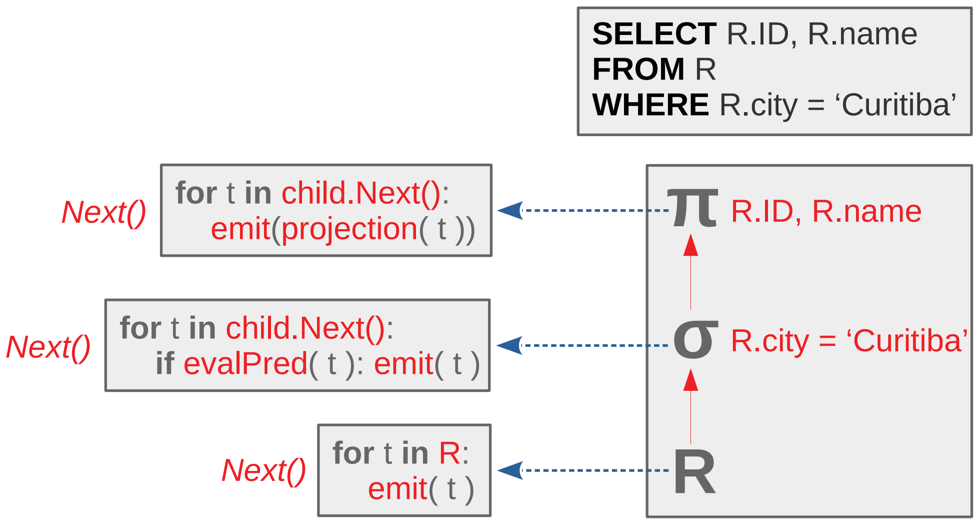


Figure 2.8: Volcano iterator model to the left and the query execution tree to the right (i.e. query plan).

2.4.2 The Materialization Query Execution Model

The volcano-style systems have many downsides for query processing when modern DBMS implement the column-store layout for OLAP (see [32] for a detailed discussion). An alternative is the materialization query execution model implemented by the MonetDB [38] and C-Store [39, 40] DBMSs. The materialization query execution is designed to benefit from the column-store layout and consequently favor OLAP workloads. This model performs column-at-a-time processing as it only moves the required columns around memory to execute a query saving a great deal of memory throughput.

Although modern column-oriented DBMS improve the execution of analytic queries, they undergo the tuple reconstruction problem [41]. Every time a given column is fetched, these systems must perform a tuple reconstruction action based on a temporary data structure (which is the result of a previous operator in a query plan). The $projection$ and $projectionpath$ physical operators take the burden of the tuple reconstruction or the materialization of intermediate results. They are invoked multiple times within a query plan, e.g., case a query requires N columns, at least $N-1$ projections have to be performed. Therefore, they implement the following temporary data structures to hold intermediate results: selection vector, position vector, and bitmap.

The execution model processes each query operator to completion over its entire input data in a column-at-a-time manner. One operator finishes before any invocation of a subsequent data-dependent operator. Also, it demands the late materialization strategy to delay the reconstruction of tuples until it is necessary to continue executing the query [6].

To demonstrate how the materialization query execution model operates, we run a small piece of the TPC-H Query 03 in MonetDB, see Listing 2.11. This piece implements five filter predicates ($WHERE$ clause) from three different tables ($customer$, $orders$, and $lineitem$) to project the $l\_discount$ column.

Listing 2.11: Small piece of the TPC-H Query 03.

```
SELECT
   l_discount
FROM
   customer,
   orders,
   lineitem
WHERE
   c_mktsegment = 'BUILDING'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date '1995-03-15'
   and l_shipdate > date '1995-03-15'
```

We enabled the SQL *trace* statement of MonetDB on the query to record every primitive of MonetDB Assembly Language (MAL). Figure 2.9 presents a top-down diagram of the execution plan from that query and highlights the projection operators (*projection* and *projectionpath*). The query plan optimizer pushes down the selections to filter tree columns: *c_mktsegment*, *o_orderdate*, and *l_shipdate*. Each selection operator emits as output a Binary Association Table (BAT) [42] of OIDs[2], i.e., a selection vector with the positions that matched the filter predicates as exemplified by Listing 2.12, which applies the branch boolean calculation technique to avoid branch misprediction [43].

Listing 2.12: Selection code example.

```
void select(int n, int *col, int filter, int *res) {
 for (size_t i,j=0; i < n; ++i) {
   /* boolean calculation, branchless code */
   bool b = test_filter(col[i], filter);
   j += b;
   res[j] = i; }
}
```

Projections use those intermediate selection vectors to materialize the join columns: *c_custkey*, *o_custkey*, and *l_orderkey*, i.e., they filter an input column based on the selection vector, for instance, the Listing 2.13.

Listing 2.13: Projection code example.

```
void projection(int n, int *sel_vec, int *col, int *res) {
 for (int i=0; i < n; ++i) {
   int pos = sel_vec[i];
   res[i] = col[pos]; }
}
```

[2]OID is the object ID, i.e., it is an ID number assigned to a certain position in the table.

The query plan, in some points, requires to materialize columns that depend on two or more position vectors. In these points, the *projectionpath* primitive is invoked to glue distinct position vectors and then to project a target column. For instance, in Figure 2.9, two *projectionpath* primitives join position vectors generated by selection and join operators. The first one glues two position vectors: the one generated by the selection on column *o_orderdate* and another from the join of the *custkey*s (*c_custkey* and *o_custkey*). It traverses the vectors to project the mapped values from the column *o_orderkey*. Listing 2.14 exemplifies as the *projectionpath* behaves. Firstly, it traverses the small position vector, e.g., *pos_vec*1, and uses it to filter the second one (*pos_vec*2), then only the filtered values are materialized from the target column.

Listing 2.14: Projectionpath code example.

```
void projectionpath(int n, int *col, int *pos_vec1,
            int *pos_vec2, int *res) {
 for (size_t i=0; i < n; ++i) {
   int pos1 = pos_vec1[i];
   int filter_pos = pos_vec2[pos1];
   res[i] = col[filter_pos]; }
}
```

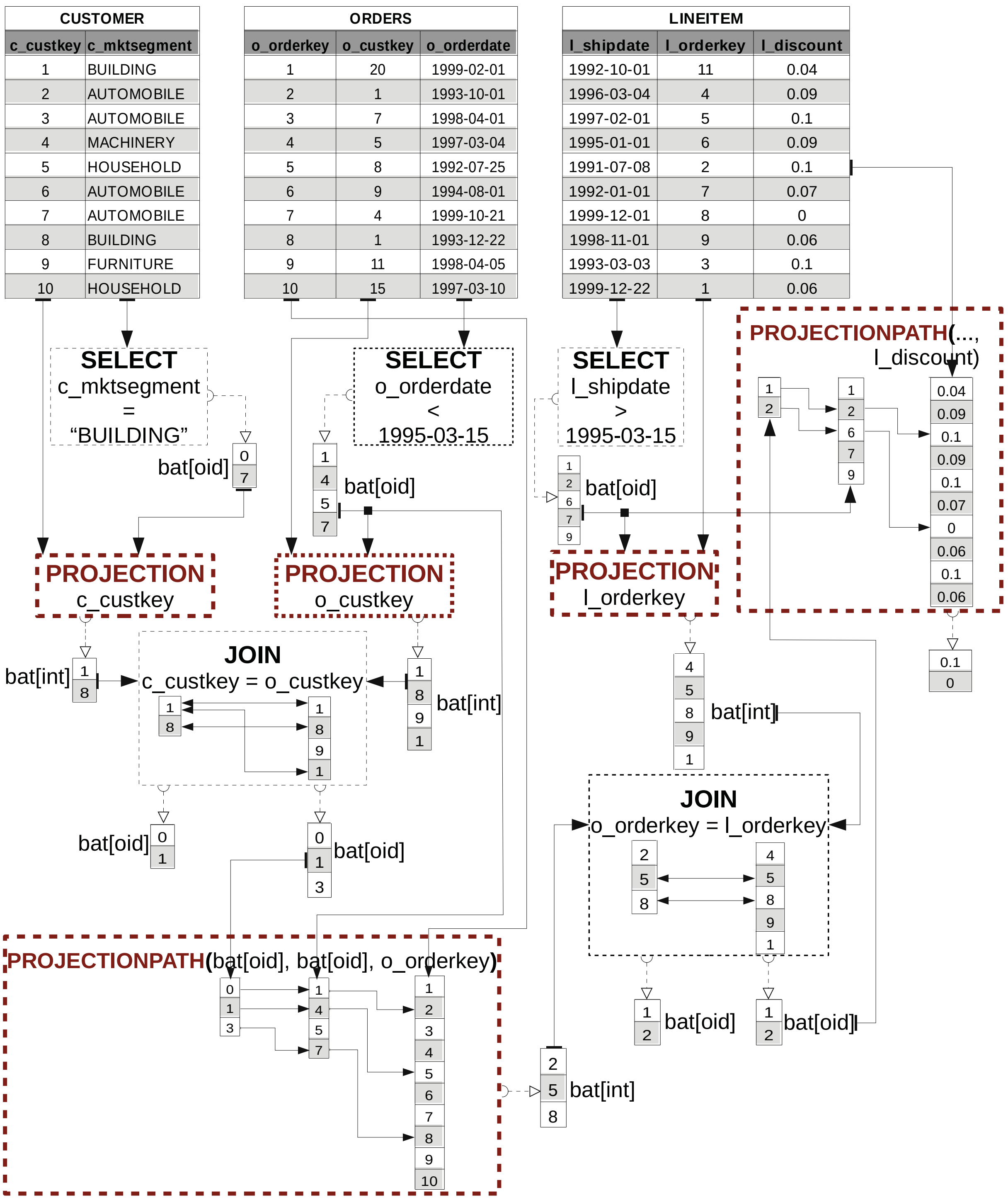
The diagram in Figure 2.9 illustrates the interaction of database operators in a query plan. Furthermore, it discloses details of how tuple reconstruction materializes within a materialization query execution model, and thus the relevance of the projection operations on it. Although these systems mitigate data movement by processing column-at-a-time and avoiding memory throughput waste such as in row-oriented systems, they have to maintain several intermediate data structures (e.g., selection vector or bitmap) in memory. Moreover, they materialize many intermediate columns during query execution.

2.4.3 The Vectorized Query Execution Model

The vectorized query execution model tries to avoid some pitfalls of the Volcano-like iterator model. The tuple-at-a-time execution of Volcano-style causes high interpretation overhead and inhibits compilers to exploit CPU parallelism, leading DBMSs to low Instructions per Cycle (IPC), e.g., IPC of 0.7 [8].

The Volcano iterator model allows high cohesion and low coupling by encapsulating database operators. However, it performs pipelined processing among database operations using the tuple-at-a-time execution: Every database primitive[3] is called multiple times during a query execution at a tuple granularity. It means that one singular operation (e.g., an arithmetic operation) is invoked for every singular value that it should process.

[3]A database primitive is one physical compiled function/routine that the DBMS calls during the query execution.



CUSTOMER	
c_custkey	c_mktsegment
1	BUILDING
2	AUTOMOBILE
3	AUTOMOBILE
4	MACHINERY
5	HOUSEHOLD
6	AUTOMOBILE
7	AUTOMOBILE
8	BUILDING
9	FURNITURE
10	HOUSEHOLD

ORDERS		
o_orderkey	o_custkey	o_orderdate
1	20	1999-02-01
2	1	1993-10-01
3	7	1998-04-01
4	5	1997-03-04
5	8	1992-07-25
6	9	1994-08-01
7	4	1999-10-21
8	1	1993-12-22
9	11	1998-04-05
10	15	1997-03-10

LINEITEM		
l_shipdate	l_orderkey	l_discount
1992-10-01	11	0.04
1996-03-04	4	0.09
1997-02-01	5	0.1
1995-01-01	6	0.09
1991-07-08	2	0.1
1992-01-01	7	0.07
1999-12-01	8	0
1998-11-01	9	0.06
1993-03-03	3	0.1
1999-12-22	1	0.06

SELECT c_mktsegment = “BUILDING”
bat[oid]
SELECT o_orderdate < 1995-03-15
bat[oid]
SELECT l_shipdate > 1995-03-15
bat[oid]
PROJECTIONPATH(..., l_discount)
PROJECTION c_custkey
PROJECTION o_custkey
PROJECTION l_orderkey
bat[int]
JOIN c_custkey = o_custkey
bat[int]
bat[int]
bat[oid]
bat[oid]
JOIN o_orderkey = l_orderkey
bat[oid]
bat[oid]
PROJECTIONPATH(bat[oid], bat[oid], o_orderkey)
bat[int]

Figure 2.9: Top-down diagram of a partial execution of the TPC-H Query 03 on MonetDB.

Considering, for instance, the sum expression: **+(double src1, double src1) : double**, which sum two double values, in which **src1** could be a value from one column and **src2** either a value from another column or one constant value. The assembly-like code would look like:

Listing 2.15: Sum of two double values in an assembly-like code.

```
LOAD  src1, reg1
LOAD  src2, reg2
ADD   reg1, reg2, reg3
STORE dst,  reg3
```

The pitfalls of this code are the inherent data hazards (data dependence): one between the LOADs and the ADD instruction and another between the ADD and STORE. In the first one, the ADD instruction must wait for the completion of the LOADs to continue the CPU pipeline. The STORE instruction depends on the result of the addition to save it into the memory. Without those dependencies, a MIPS [44] processor with only one pipeline should have an IPC of 1 (one instruction per cycle). However, the CPU must add at least two "STALL"[4] instructions into the pipeline to delay the execution of the ADD and another before the STORE instruction, leading to an IPC of 0.66 ($0.66 = 4/6$: 4 useful instructions (LOADs, ADD, and STORE) divided by the total of instructions (useful instructions plus STALLs)). This is a clear decline in CPU performance that applications must be aware.

At the same time, compilers easily solve data dependencies by putting more instructions into the pipeline, i.e., they apply the *loop pipelining/loop unrolling* technique. However, primitives in legacy DBMS compute only one operation per call, which prevents compilers to perform the loop pipelining.

The consequences of tuple-at-a-time processing are twofold: 1) Database primitives perform one operation per call, precluding compilers to make a pipelined loop; 2) The high cost of primitive call for every single operation.

Those drawbacks lead to the vector processing of database primitives introduced in MonetDB/X100 [7], which later let to the Vectorwise spin-off [45, 46] and many other DBMSs, including SQLServer [47] and DB2 BLU [48]. These DBMSs conceive the vectorized processing query system. They yet employ the Volcano-like pipelined processing, but instead of tuple-at-a-time execution, they operate at the granularity of vectors. These vectors have a small size (e.g., 1024 values) to fit into the caches, and thus database primitives should be cache-conscious by processing large datasets in cache-chunk fragments. Vectorized primitives allow compilers to generate efficient loop-pipelined code.

Let us consider the simplified version of TPC-H Query 01, in Listing 2.16. This query filters the column $l\_shipdate$ and generates a selection vector, then it applies a subtraction on column $l\_discount$, and right after a multiplication with column $l\_extendedprice$, the result (sum_disc_price) is aggregated based on the $l\_returnflag$ column.

[4]STALL is an useless instruction to delay the execution of the pipeline until a certain value is ready, it is also called a **bubble** or **NOP** ("no operation") instruction.

Listing 2.16: Simplified version of TPC-H Query 01.

```sql
SELECT
   l_returnflag,
   sum(l_extendedprice * (1 - l_discount)) as sum_disc_price
FROM
   lineitem
WHERE
   l_shipdate < date('1998-09-03')
GROUP BY
   l_returnflag
```

The diagram in Figure 2.10 presents the plan execution of Query 01. Every database primitives process a vector-at-a-time. The scan operator, at the bottom of Figure 2.10, emits vector chunks to be processed by the selection operator, which generates a selection vector. Then, there are two projections with arithmetic expressions that apply the selection vector to filter the target columns and perform the subtraction or multiplication operations. The aggregation operator reads the group column ($l\_returnflag$), filtering it with the selection vector, calculates the hash address to load the Hash Table (HT) entries, and updates the aggregation values with the results of the projection expressions. Note that only the target columns are scanned when strictly needed, during query execution intermediate data (e.g., selection vector and the result of projections) are kept as long as possible in caches.

Listing 2.17 brings the projection expression to multiple float vectors from two columns (e.g., $col1$ and $col2$) in a loop-fashion. It filters the columns using a selection vector and stores the result into a separate vector, which shall be processed for the next operators in a query plan.

Listing 2.17: A map database primitive that projects two columns based on a selection vector and multiples those columns, it also stores the result in another vector [8].

```c
void map_mult_float_col_float_col(
   int n,
   float*__restrict__ res,
   float*__restrict__ col1,
   float*__restrict__ col2,
   int*__restrict__ sel_vec
)
{
  for(int j=0; j<n; ++j) {
   int i = sel_vec[j];
   res[i] = col1[i] * col2[i];
  }
}
```

Vectorized primitives, like the one presented in Listing 2.17, enable the compiler to unroll the loop and thus find instructions to go into different pipelines, and also allow the *out-of-order* execution technique. With loop pipelining is also possible to solve data dependencies among instructions like the ones exposed in Listing 2.15.

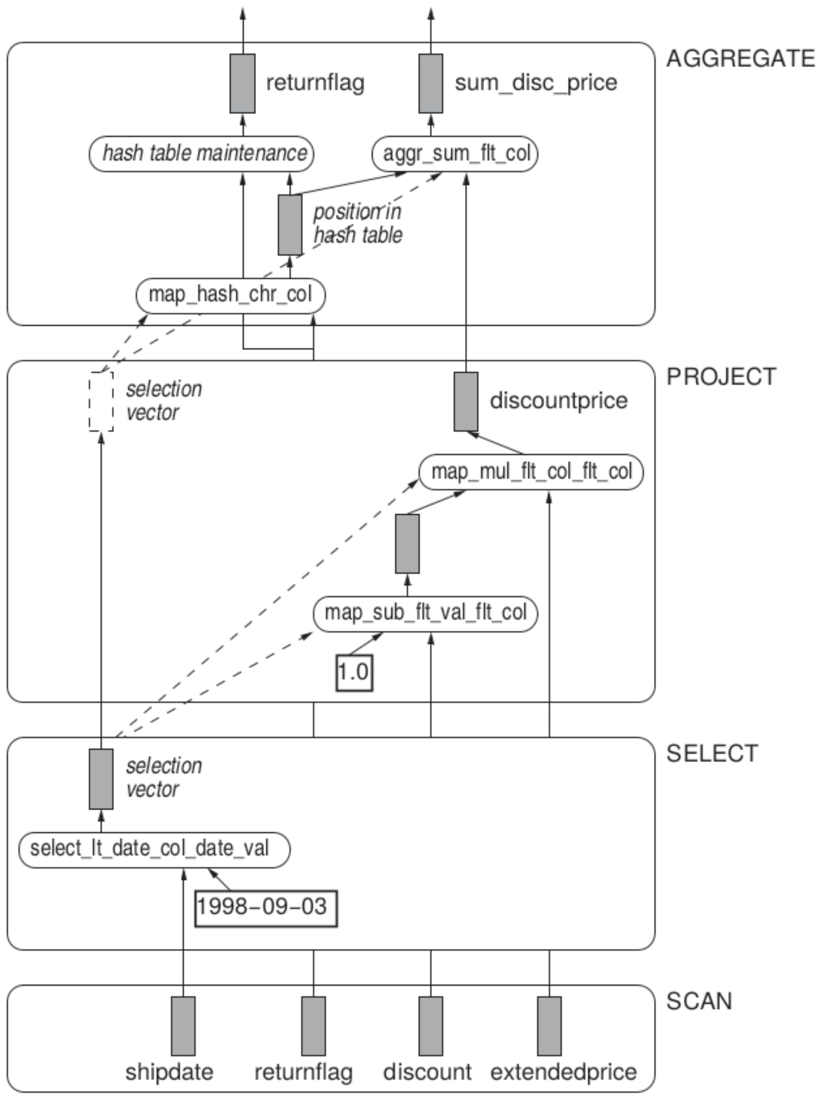


Figure 2.10: Top-down diagram of a simplified version of TPC-H Query 01 on MonetDB/X100 [8].

To demonstrate how the loop pipelining would be applied, we present Listing 2.18. This listing depicts the sum expression of Listing 2.15 unrolled 4× by the compiler using loop pipelining. Note that we considered the input sources as vectors of doubles. The compiler successively unrolls the LOAD instructions that are followed by four ADD instructions. The first ADD (line 11) occurs six instructions after starting the LOADs (lines 2 and 3) of its two operands, and the corresponding STORE (line 16) starts three instructions later. The intervals of instructions among the dependent ones circumvent data hazards and boost CPU performance.

Listing 2.18: Loop pipelining with loop unroll depth of 4x applied to the sum expression: **+(double src1, double src1) : double**.

```
1 // unrolling the LOAD instructions
2 LOAD  src1 +0, reg1
3 LOAD  src2 +0, reg2
4 LOAD  src1 +4, reg4
5 LOAD  src2 +4, reg5
6 LOAD  src1 +8, reg7
```

```
LOAD  src2 +8, reg8
LOAD  src1 +12, reg10
LOAD  src2 +12, reg11
// unrolling the ADD instructions
ADD   reg1, reg2, reg3
ADD   reg4, reg5, reg6
ADD   reg7, reg8, reg9
ADD   reg10, reg11, reg12
// unrolling the STORE instructions
STORE dst +0, reg3
STORE dst +4, reg6
STORE dst +8, reg9
STORE dst +12, reg12
```

The main proposal of vectorized query execution model is to exploit CPU parallelism by allowing compilers to apply critical optimization techniques. The vector-at-a-time execution provides many instructions for loop pipelining that can solve data hazards, take advantage of multiple pipelines, and the out-of-order execution technique that modern processors support. Moreover, another indirect beneficial effect is a better usage of the caching mechanism, as the vectors are cache chunks that inhibit the wasting of memory throughput. Although all these effects mitigate the memory movement problem, it arises again when vectors start to exceed CPU caches, causing extra memory traffic.

2.4.4 The Pipelined Query Execution Model

While processing a query, the DBMS translates it into an expression of the relational algebra, and then evaluates every piece (i.e., the operators) of such expression to produce the query answer, such as in the Volcano-like iterator model. The major benefit of that is *pipeline data*, i.e., an operator can emit data to the next one without copying or materialize the data. The vectorized execution, albeit mitigates the high cost of function calls and favors CPU parallelism by enabling loop pipelining, disables the *pipeline data* of the iterator model: It breaks the operations into pre-compiled primitives that have to materialize the output at a vector-at-a-time fashion.

On the other hand, the pipelined query execution model proposes a new paradigm for data-centric processing [49] (the other approaches are operator-centric). The main goal is to keep data into CPU registers as long as possible, handling operator boundaries. A pipelined system operates on a push-based processing model, which means that data is pushed towards operators instead of pulling into them, resulting in a better data locality. To that end, JIT compilation is applied to generate optimized code at runtime, thanks to the Low Level Virtual Machine (LLVM) compiler framework [50]. Then, a single tuple can go through the entire query pipeline without materializing the intermediate results.

Thomas Neumann [49] introduced the definition of *pipeline-breaker*: In the query execution plan, a *pipeline breaker* is a whenever point where an algebraic operator has to evict an incoming tuple out the CPU registers. The pipeline-breaker concept is quite close to the definition of blocking input/output edge [51], and later detailed by Luc Bouganim et al. [52]. Therefore, an execution plan might have many pipeline breakers each pipeline fuses operators into a loop and processes tuple-at-a-time.

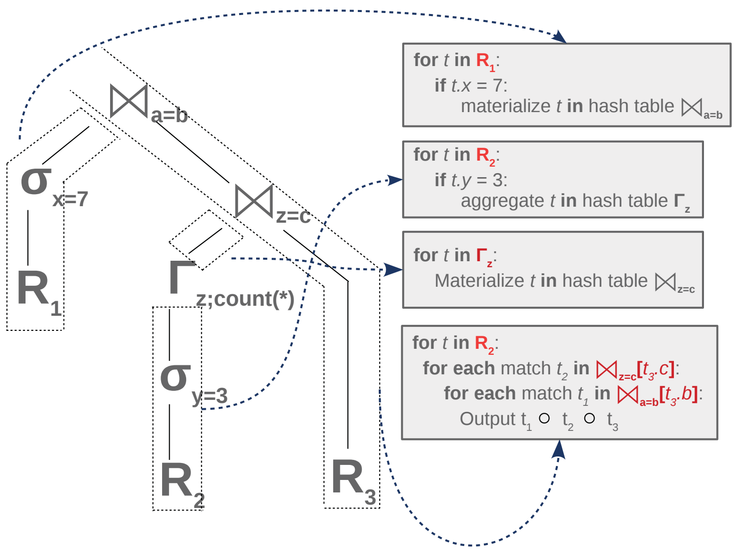
Listing 2.19: Query example for the pipelined query execution model [49].

```
SELECT
   *
FROM
   R1,R3,
   (SELECT
      R2.z,
      count(*)
   FROM
      R2
   WHERE
      R2.y = 3
   GROUP BY R2.z) AS R2
WHERE
   R1.x = 7
   and R1.a = R3.b
   and R2.z = R3.c
```

Listing 2.19 presents a query that we use as an example to detail the pipelined query execution model (based on the related work [49]). The corresponding execution plan is shown in Figure 2.11, in the left, and the compiled query with fragmented code, in the right. The query originates four pipelines (four tight loops) with operators fused into them. The left-most pipeline selects tuples from R1 ($\sigma_{x=7}$) and materializes them into the hash table $\bowtie_{a=b}$, i.e., builds the hash table from column $R1.a$. In the branch on the left, one pipeline selects some tuples from table R2 ($\sigma_{y=3}$), groups them by $R2.z$ building a hash table (Γ_z) with many distinct occurrences of $R2.z$, and aggregates them ($count(*)$). The third pipeline joins the hash table of Γ_z with $R3.c$ ($\bowtie_{z=c}$). Then, the last pipeline joins the tuples from the right branch within hash table $\bowtie_{z=c}$ with the tuples from the left branch within the hash table $\bowtie_{a=b}$.

The interesting remark on the fragmented code in Figure 2.11 is the fusion of operators into tight loops, generating a data-centric query execution, i.e., each code fragment executes every possible instruction in a CPU pipeline execution, before evicting (materializing) any result out of CPU registers to the next pipeline breaker, augmenting data locality in the register bank of the processor.

[9] investigates the trade-offs between vectorized and pipelined processing systems. The former favors parallel data access with some advantage in memory-bound queries, e.g., the ones with aggregation and join on large hash tables. The latter supports well compute-intensive



```
for t in R1:
  if t.x = 7:
    materialize t in hash table ⋈a=b
for t in R2:
  if t.y = 3:
    aggregate t in hash table Γz
for t in Γz:
  Materialize t in hash table ⋈z=c
for t in R3:
  for each match t2 in ⋈z=c[t3.c]:
    for each match t1 in ⋈a=b[t3.b]:
      Output t1 ∘ t2 ∘ t3
```

Figure 2.11: Example execution plan with pipeline breakers [49] and the respective compiled code.

queries by keeping data into CPU registers as long as possible. Another related work [53] proposed to exploit the advantages of both systems through a relaxed operator fusion to glue compilation, vectorization, and prefetching. The pipelined query execution model is implemented by Hyper [54], and the latest version of PostgreSQL V.12 [55].

2.5 SUMMARY

In this chapter, we presented the database operations and how the current query execution models run them into modern DBMSs. In the last decades, database engineers have been conceiving new query execution model to manage the issues of the Volcano-like processing, since its introduction.

The materialization query execution model implemented in MonetDB [5] shifted the paradigm of tuple-at-a-time to column-at-a-time execution. It benefits from the columnar storage layout to avoid wasting memory throughput. However, it requires the full materialization of intermediate results allover the query execution.

The vectorized query execution model of MonetDB/X100 [7] proposes the vector-at-a-time execution. The main goal is to exploit modern CPU capabilities, such as pipeline and out-of-order instruction execution. The vector-at-a-time allows the compiler to apply the loop pipelining technique to leverage computing parallelism that modern superscalar processors provide. On the other hand, it also needs to materialize intermediate results in a vector granularity.

The pipelined query execution model, also known as a data-centric compilation, aims at keeping tuples as much as possible into the CPU registers without materializing or copying. It introduces the *pipeline-breaker* [49] to define the boundaries of a pipelined query execution. Modern DBMS have applied this technique for compute-intensive applications, such as Hyper [54], and the latest version of PostgreSQL [55] enables JIT by default.

However, those query execution models were thought for a computing-centric hardware architecture. Although they alleviate some performance pitfalls related to the Volcano-style iterator model, all of them are still based on the von Neumann architecture design that still requires to transfer data from the memory to the CPU registers, paying all cost related to the data movement problem. At the same time, one big research concern of today's computer architects is to shift the design of computer architectures to data-centric architectures: Process data where it resides (where it makes sense) [56]. In the database field, we should share the same thought for query execution systems. The next chapter brings related work on processing database operations in memory hardware processors.

3 RELATED WORK

In this Chapter, we analyze related work for Near-Data Processing (NDP) that follows the memory technology over the years. In Figure 3.1, we present a top-down overview of a modern memory hierarchy. At the top of the hierarchy is the CPU with a register bank and logical units. Close to the CPU, there is a detached cache hierarchy that caches data from DRAM, such organization is from the legacy von Neumann model. Recent research work [57, 58] have suggested to include non-volatile memories between the DRAM layer and memory disks that are placed at the bottom of the memory hierarchy. In all memory levels, we include the approximated latency (Cycles Per Instruction (CPI)) and size in bytes, and some related work.

Along with this chapter, we present related work for NDP through a bottom-up perspective based on Figure 3.1, i.e., we start with related work on memory disks and visit all levels of the memory hierarchy, sequentially. In some sections, we start with generic approaches for NDP, then it follows with specific related work for database operations. Typically, database operations (e.g., selection, join, aggregations, etc) are translated into near-data operations and offload to functional units within memory hardware, and the data from the relational model are processed in chunks, for example, modern PIM hardware process data using chunks of 256B.

The rest of this chapter is organized as follows: We begin with related work for NDP on magnetic disks. In Section 3.2, we introduce some peculiarities of flash disks and point related work. As there are many efforts directed to Non-Volatile Memory, we briefly introduce current open issues and directions for that in Section 3.3. In section 3.4, we bring NDP work on DRAM and the PIM for database operators and data analytics. We summarize related work in Section 3.5 with a table containing only the related work for NDP on database operators.

3.1 NEAR-DATA PROCESSING IN MAGNETIC DISKS (HDD)

Naturally, many efforts have been spending to design NDP architectures for secondary memory once it is at the base of the hierarchy with the biggest storage capacity (more than 2^{40} bytes), but it has the highest latency for memory operations between 10^5 and 10^6 cycles of CPU. Many works tried to design smart storage devices with the addition of logical components inside magnetic disks. However, they were proposed in an epoch when Moore's law scaled, and processing capabilities were limited.

In magnetic disks, NPD approaches explored the sequential access to maximize disk bandwidth and to optimize the use of disk arms [59]. At the same time, random memory access causes arm movement that increases seek time, leading applications to have a high CPI due to the waiting of I/O operations. Thus, in DBMSs, random disk accesses are reduced by caching hot pages in the main memory [59].

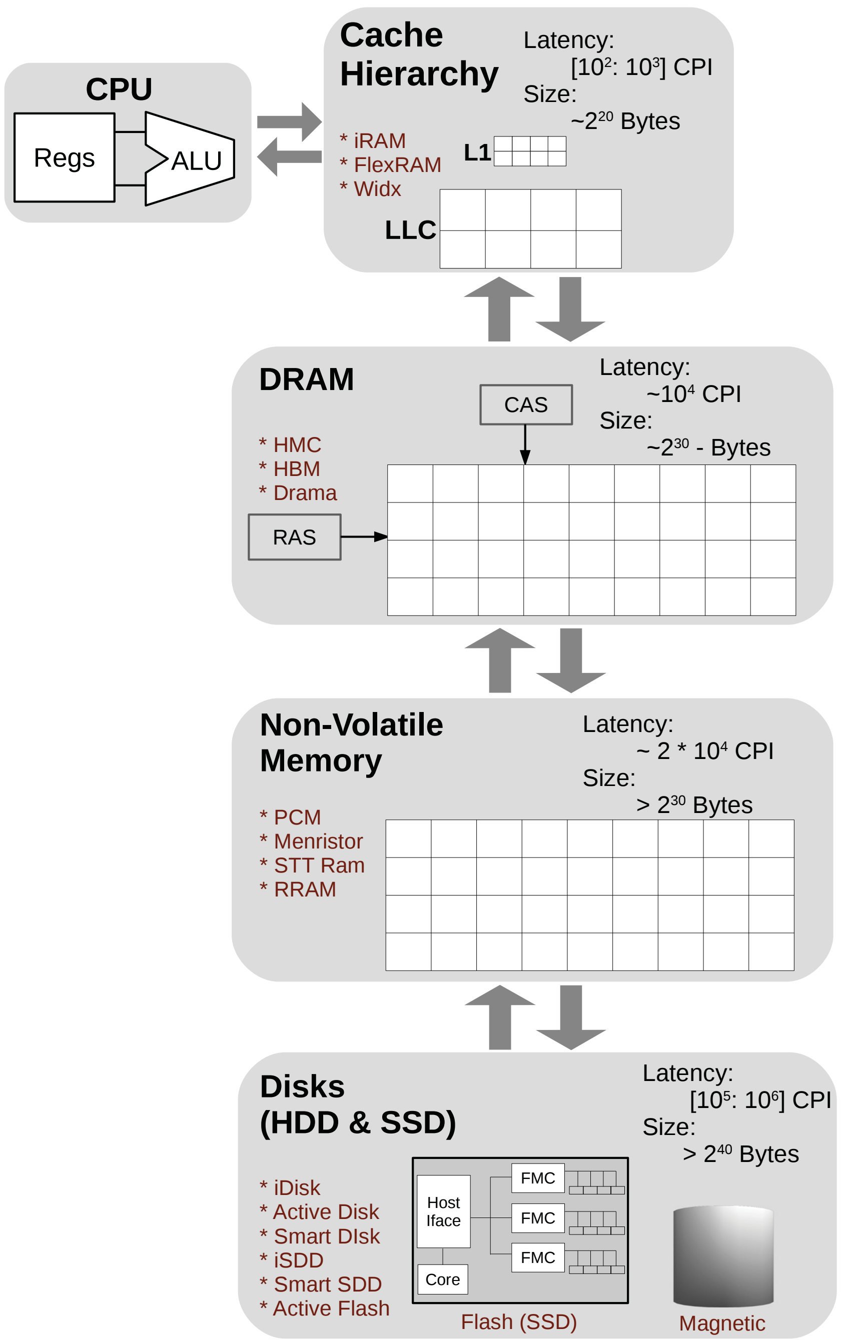


Figure 3.1: Memory hierarchy with the latency and size of all levels, and some corresponding related work.

Pioneer works in database machines, the ’70s and ’80s, added specialized processing components to perform particular database algorithms in hard disks. In [11], David DeWitt and Paula Hawthorn classified those architectures according where the data is placed on disk to attach the new processing components: processor-per-track, processor-per-head, and processor-per-disk. Inspired on the database machines, Figure 3.2 presents NPD architectures designed for magnetic disks.

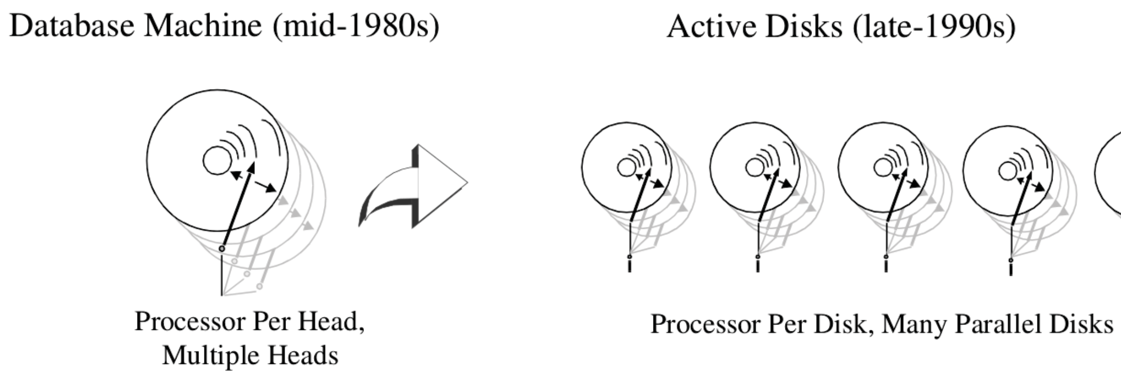

Database Machine (mid-1980s)
Active Disks (late-1990s)
Processor Per Head, Multiple Heads
Processor Per Disk, Many Parallel Disks

Figure 3.2: Commom processor-per-head in database machines to processor-per-disk in Active Disks with disks in parallel [60].

3.1.1 Database Operators Processing in Magnetic Disks

Intelligent Disks (IDISKs) [12] was proposed in 1998 to support the increased demand of I/O and associated processing requirements for data warehousing and decision support systems. It emerged as a replacement alternative of the shared-nothing cluster-based database, in which each IDISK consists of a hard disk with an embedded processor, an internal memory limited to 64MB, and a gigabit network serial link. The links connect IDISKs via non-blocking crossbar switches to communicate with each other. From the perspective of the DBMS architecture four proposals were presented for IDISKs: 1) Incorporate a shared-nothing database server and operation system on each IDISK; 2) Adjust applications to interact with IDISK through a library of functions provided by the disk manufacturer; 3) Add the database storage manager and a reduced operating system on each IDISK; 4) Run basic database operators and a reduced operating system on IDISKs. The IDISKs study just presented a guideline for future implementations without supplying empirical results.

In the literature, we found two works called Active Disks (AD), both based on an architecture with a server-host responsible for coordinating and scheduling a cluster of many ADs in parallel with NDP support. Acharya, A. et al. [13] proposed AD as a stream-based programming model to process disk-resident code, called disklets. Conventional I/O requests are sent to disk, but instead of returning data to the host, they are put in internal buffers of the AD and processed by a disklet. This requires an operation-system layer divided in DiskOS and the Host-level OS support. DiskOS is responsible for memory management, stream communication, and disklet scheduling. It allocates the disklet streams in internal disk buffers and schedules the disklets as the buffers are ready. The Host-level is responsible for the management of the host-resident stream and the initialization of disklets. The performance and scalability of that architecture were evaluated through simulation using the following database applications: *selection* operator with predicates, SQL *group-by* with aggregation functions (MIN, MAX, SUM, AVG, and COUNT), *sort* operator and *datacube* [61]. Those AD applications outperformed conventional-disk applications by between 1.0 and 3.2 times of execution time in 4-disk and

32-disk configurations, but a drawback is that they must be adapted to Active Disks architecture due the parallelism and coded as disklet app.

Erik Riedel et al. [62, 60, 63] designed another approach also called Active Disk grounded on three principles: 1) Leveraging parallelism in data-centric environments; 2) Processing data as a stream with a small amount of state; 3) Execute a few instructions per byte. Similar to the work of Acharya, A. et al., the host processor is responsible for parameter initialization, coordination, and merging results, but Erik Riedel et al. conducted a testbed with ten real prototypes of AD, i.e., a non-simulated environment. ADs improved throughput and execution time for the operators: *selection*, *join*, and *aggregation*, extracted from queries 1, 5, 6, and 9 of TPC-D. The results of selection and aggregation are more salient, reaching up to 100% of improvement in both metrics. But the improvement of the join operator is marginal between 11% and 17%, using the Bloom Join algorithm.

Gokhan M. et al. [64] evaluated Smart Disk (a general term for disks with computational power, which comprises Active Disks and IDISKs) through representative queries from Decision Support System (DSS). Frequently occurring database operators were extracted from those queries and bundled in the same operation to be executed together in a single invocation. The following three architectures were designed and tested using the bundles of operators: a single host-based, cluster-based, and smart disk-based. The smart disk architecture outperformed the host and cluster-based ones with similar configurations.

Steve C. et al. [65, 66] extended the smart disk system to a distributed architecture by devising Smart Disk Group (SDG). Within an SDG exchanging, combination and redistribution of data can occur directly between smart disks, no need of communication with the host. The authors evaluated that new distributed architecture with queries 1, 6, and 12 of TPC-H, data clustering, and fast Fourier transform. To run those TPC-H queries the following database primitives were implemented and offloaded to the Smart Disks (SDs): scan, join, sort, group-by, and aggregate. The scan works as a filter in local SD, the sort is performed in parallel using buckets and global communication, also group-by and aggregate use the sort operator. The join primitive hashes data of SDs, then re-distributes them among SDs to perform local joins.

Piernas, J. and Nieplocha, J. [67, 68] designed an active storage architecture on the LUSTRE parallel file system [69]. Different from previous works, the active storage nodes are uncoupled of computing (host) nodes, which means that both are clients of the distributed file system and can get access to all files stored in it, regardless they are local or not. One of the clients runs as master of the Active Storage program which has all the information to do the job, i.e., the program run path, parameters, and the input file paths to process. That work focuses on scientific, parallel and supercomputing workloads.

Many other approaches have exploited active storage in magnetic disks. We highlighted the work of V. Stoumpos and A. Delis [70] due to the implementation of the Grace Join algorithm on active storage for the join database operator. S. W. Son et al. [71] proposed an energy-saving system that dynamically identifies, at compile-time, data filtering portion of an application and delegates to smart disk. IBM also proposed an NDP architecture on disks, the idea was to provide code extensions to applications to run on disks, such as searching, sorting, and indexing [72]. In Storage Fusion [73] the storage system was adapted to dynamically prefetch I/O requests from database query plans through two prefetching levels, additionally, the authors proposed an autonomic database reorganization into the storage to tackle the problem of structural deterioration.

3.2 FLASH DISKS (SSD)

As Jim Gray foresaw: “Tape is dead, disk is tape, flash is disk, ram locality is king.” [74]. Flash disks have emerged as natural substitutes for magnetic disks, because flash devices have no mechanical components and data are electronically stored. This eliminates the movement of the disk arm to position the head into the correct track (*seek time*) and, after, wait the rotation of disk platter to get the target block (*rotation time*). Consequently, this shift in disk technology decreases latency time (10^5 CPU cycles) and saves energy.

A flash-based SSD has several arrays of flash chips. Each one is wired to its Flash Controller via one flash channel (bus), as depicted in Figure 3.3. Additionally, flash chips are composed of logical units (LUN). I/O operations can occur in parallel between distinct LUNs and flash channels, but, in the same LUN, operations are executed serially [75]. Such parallelism leads the flash random access to be faster than sequential ones. However, legacy DBMSs based on the fast sequential access of hard disks, which plays a big role in how NDP approaches for databases should work on flash disks.

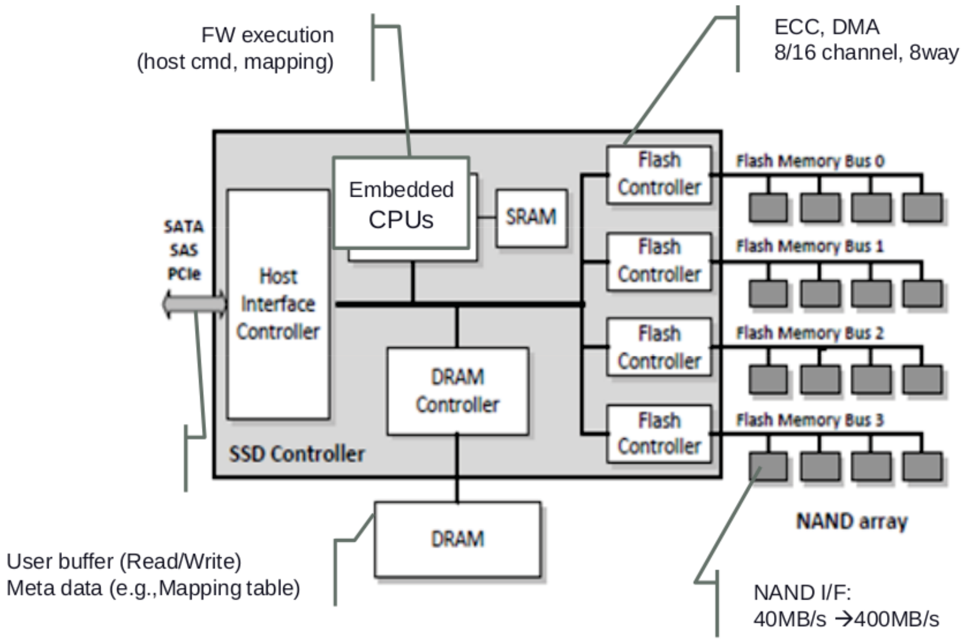


Figure 3.3: SSD general architecture [76].

However, flash memories have some constraints to deal with [77, 78]: 1) In cell flash, write operations must be performed at a page-granularity. 2) Before the writing of a page, the flash disk must perform an erase operation at the flash block-granularity (typically 64 flash pages). 3) Sequential writes within a block. 4) The limited number of erase operations in a block (lifetime: 10^6 up to 5×10^4 erases per block). Thus, according to P. Bonnet and L. Bouganim [77], a bimodal flash can expose internal SSD features to DBMS to share the responsibility to attend those constraints and improve the I/O performance.

3.2.1 Near-Data Processing In Flash Disks

Goetz Graefe et al. [79, 80] proposed two processing operators adjusted to SSDs: FlashScan and FlashJoin. They used the PAX storage model [81], considering that SSDs work with data transfer units (512 KB up to 2 KB) lesser than database pages (8KB up to 128KB), then the FlashScan reads only the PAX-minipages inside one page, and fetches the needed attributes to run the query. FlashScan explores the quick random access of SSDs by interleaving the reading of minipages in the same PAX page. Likewise, the FlashScan reads only the join attributes and transfer them to the JoinKernel, then the FetchKernel transfers the attributes of interest to the next operators.

The scan operator was also boosted by Kim, S. et al. [76] in terms of performance and energy consumption. They argued that SSDs are bounded by the embedded CPU and DRAM that cannot explore the maximum bandwidth of the storage media. Hence, they suggested incorporating a dedicated scan-processor into the Flash Controller (see Figure 3.3), which fetches data from flash memory bus to the scan-processor that works as a proxy by transferring only the filtered data to the SSD internal DRAM. Each Flash Controller has a dedicated scan-processor that scale-out according with the flash arrays.

For High-Performance Computing (HPC) workloads Active Flash [82, 15] enabled *in situ* processing, in which the compute node (host CPU) runs scientific applications to simulate natural phenomenons that generate a large volume of data for post-processing data analysis. Active Flash runs out-of-core data analysis inside the SDDs, avoiding multiple rounds of data movement in the memory hierarchy that HPC workloads perform. They added into the SSD controller four common kernel functions of scientific workloads: max, mean, standard deviation, and linear regression. A command protocol manages the invocation of those kernel functions at an SSD idle time when an I/O request arrives, the kernel function is interrupted, reassuming after the I/O finish.

In the database context, Smart SSD [16] was proposed for data processing using an embedded ARM processor into the SSD through a firmware composed of a communication protocol and an API for command management, threads, memory, and data storage. Smart SSD was evaluated with the selection operation, selection with aggregation functions, and the join operator [83]. The experiments were conducted in four synthetic data sets (tables) with 4, 16, and 64 integers stored in a single tuple and the LineItem TPC-H table [21]. The authors verified that tables with fewer tuples per block (e.g., 64 integers per tuple) reach better performance because they require less processing power of the embedded ARM processor, i.e., they are I/O-bound.

On the other hand, tables with short tuples imply in more tuples per block while processing they saturate the ARM processor because those data sets are CPU-bound, and should be processed in the x86 processor.

The authors of Intelligent SSDs (iSSD) [17] added a reconfigurable stream processor inside the SSD that acts as dedicated hardware to reach high processing performance with energy savings. The bandwidth of the internal shared DRAM limits such a processor. However, iSSD was tested through a mixed workload composed of programs with high instructions per byte (IPB), i.e., compute-intensive programs, and others with low IPB (I/O-intensive). For example, programs like query 6 of TPC-H with low IPB reached 2.6 times of improvement than the x86 processing. Intelligent SSDs was also applied in data mining applications for Big Data [84], the K-means and PageRank algorithms were evaluated.

The Ibex project [85] deployed an engine of processing in an FPGA connected to the SSD via the SATA bus. That engine was integrated into the MySQL to offload selection with predicates and aggregation functions (COUNT, SUM, MIN, MAX, AVG).

Swanson, S et al. [86] focused on the processing of lists intersection, a common operation in DBMSs to merge intermediate results. They implemented the list intersection inside a real prototype of SSD (Samsung Smart SSD). Such a device has a smart SSD firmware that iterates with the application via the SSDlet component. A SSDlet is an execution program inside the SSD that is event-driven by the component *Smart SSD runtime system*. The application communicates with the SSD firmware via an API with the commands: OPEN, CLOSE, GET and PUT. The execution result of a SSDlet is added into an output buffer, then the application gets back them through the GET command, according to the polling (heart-beat) strategy.

3.3 NON-VOLATILE MEMORY

Non-Volatile Memory (NVM) emerged as a promise to fill the gap between main memory and secondary storage by being byte-addressable and due to its memory access latency lower than the latter. But NVM has shortcomings, such as lower memory bandwidth than DRAM, high write latency/power, and low endurance. Therefore, the integration of DRAM and NVMes [58] arises as a potential solution towards the limitations of NVM and, also, to reduce the memory power consumption of the memory subsystem. DRAM serves as a buffer cache to the NVM to mask its write latency and lower bandwidth [57].

Consequently, NDP approaches in NVMes consider that integration. For example, in a cooperative heterogeneous memory system, Liu Z. et al. [87] studied when to run applications either on DRAM or NVM in a NDP fashion. Their insights suggest that applications in which datasets can fit into DRAM and have lower reuse distance of blocks [88] than DRAM capacity are amenable to process near DRAM. On the contrast, applications with streaming behavior, reuse distance larger than the DRAM capacity, large datasets, and high read-to-write ratios are propitious to process near to NVM.

Many research and industrial projects are investigating emerging NVM technology such as Resistive RAM (RRAM), Spin-Transfer Torque Magnetoresistance RAM (STT-MRAM), Phase Change Memory (PCM), 3D Xpoint, Memristors and more. Also, on those technologies, computer architects share the same desire to make in-memory processing feasible [89].

3.4 MAIN MEMORY (DRAM)

The introduction of 3D-stacking memory technologies using a TSV [90] boosted the conception of new memory architectures, such as HMC [18], HBM [19], and DRAMA [91]. Those emerging memories devices have 4, 8, or 16 stacked-DRAM dies grounded on a logical layer and connected via TSV. That memory organization aims to tackle the scaling limit of DRAM (high capacity), improving the DRAM bandwidth (high bandwidth up to 320 GB/s) and reducing energy consumption. They are named Processing-in-Memory (PIM), and thus NDP approaches have benefited from their logical layer by processing applications on-chip memory to reduce data movement throughout the cache memory hierarchy and, consequently, saving energy.

Although, many works designed custom-hardware near to DRAM to accelerate specialized NDP applications, such as sparse matrix-matrix multiplication [92] and Fast Fourier Transform [93]. However, those approaches depend on custom-circuitry and support just specific applications.

Vermij E. et al. [94, 95] designed a NDP architecture on FPGA. On that board, they implemented a specialized component, called NDP-Manager, which manages the virtual memory for NDP, global address translation, data access, and ensure the cache coherence. Again, such an approach is custom-circuitry dependent and optimized for a specific workload. The authors used the MergeSort algorithm as a use case to evaluate empirically the NDP-Manager.

PIM-enabled-instructions, designed by Onur Mutlu et al. [96], is a new PIM architecture that monitors the locality of data accessed by applications through dedicated hardware and decides at runtime where to run instructions, i.e., in the host processor or the HMC.

NDCores [97] arranged a chain of four HMC devices and embedded 512 processing cores in each device. That architecture aims to explore the HMC parallelism and to saturate the high bandwidth (up to 320 GB/s) provided by the vaults in order to execute MapReduce applications.

For the processing of graphs in Big Data, Teseract [98] was proposed to maximize the usage of HMC bandwidth. Two specialized prefetchers hardware were incorporated to detect memory access patterns when processing graph applications.

3.4.1 Processing-In-Memory For Database Operators

Select Scan. In DBMS, vector processing is recognized as an outstanding technique to turn database operations highly efficient by availing of the cache data locality [8]. HIVE [99] emerged as an architecture to perform vector operations inside the HMC, which adds new vectorized instructions to the HMC-ISA that operate over continuous data at the granularity of 8KB (large registers). Santos P. et al. [100] proposed a Reconfigurable Vector Unit (RVU) over the HIVE that allows vector processing in units of variables sizes between 8KB and 256 bytes.

JAFAR [101] adds an off-chip memory dedicated hardware as a proxy between the CPU and DRAM, which is connected to the memory I/O buffer. When the DBMS pushes down the *Select Scan* operator to JAFAR, it directly requests data to DRAM, after JAFAR filters the DRAM output data by applying the predicates: $=, <, >, \leq, \geq$, then it just returns to CPU the filtered data. Avoiding data movement, JAFAR can provide up to $9\times$ improvements for the *Select Scan*. However, the DRAM bandwidth is a bottleneck, Jafar cannot benefit from inter-chip nor intra-chip memory parallelism, therefore.

Join Operator. Mirzadeh et al. [102] studied the impact of the hash and sort join algorithms (the state-of-the-art join algorithms: radix-hash join [103] and parallel sort-merge join [104]) and how to adapt them into the HMC. The authors proposed a join logic unit inside the HMC. They modeled the latency and energy consumption of the HMC, the CPU, and the new join logic unit. From results obtained through a first-order analytical model, they inferred that the random memory accesses of the hash join is the main cause of its poor performance and energy-efficiency in the HMC. In contrast, the sort join algorithm presented more suitable to run in the HMC because it leverages DRAM row locality.

Onur Kocberber et al. [105] built Widx, an on-chip accelerator for database hash index look-ups, which plays a big role in hash-join operations. A Widx unit decoupled the hash key generation in a dedicated component (the dispatcher) that serves the key hashing for the walkers. A walker is specialized hardware for pointer chasing in a hash bucket placed into the TLB or L1 D-cache. Widx is bound by four walkers running in parallel due to the off-chip bandwidth limitations. In the best scenery, i.e., four walkers and large hash buckets, Widx achieves a speedup of $4\times$ to process a hash-join kernel and an average speedup of $3.1\times$ in the TPC-H and TPC-DS benchmarks compared to an Out-of-Order (OoO) processor, reducing in 83% the energy consumption. However, Widex degrades case the hash table index entries fit in L1, and another trouble is that the dispatcher can be a bottleneck as it serves concurrently up to 4 walkers. For small hash buckets, the walkers can complete early and must wait while the keys are hashing.

However, all of those previous works provide a one-dimensional picture of the query processing systems. They are a one-sided approach that neglects the potential of CPU-PIM co-processing with caching and energy-saving benefits.

3.4.2 Processing-In-Memory For Data Analytics

Pioneer works have proposed PIM-based architecture for data analytics based on simulation. MapReduce (MR) applications with high spatial locality were adapted to PIM [97, 106], leading memory cores to reduce the latency up to 93.2%. This work is orthogonal to our results because MR jobs have a resembling access pattern of the selection and projection, while our study tested pipelined operators. Another work [107] relies on an analytic model to estimate the latency of 3D-stacked memories through scan-aggregate queries. They presented improvements in latency and energy consumption against traditional CPU processing and big-memory servers.

However, they only consider the dataset size variation in their analysis. In our work, we evaluate and argue more intricate factors, such as memory access patterns in caches. Mondrian [108, 109] implements an algorithm-hardware co-design for near-memory processing of data analytics operators. It is built upon a partitioning phase to turn random accesses to sequential ones, enabling a memory streaming hardware to exploit PIM capabilities. The presented results are complementary to ours. They ratify that sequential access favors PIM and show that random access is an obstacle to use the whole bandwidth. Mondrian considers algorithms with a PIM-tuned partitioning and probe phase. Instead, we evaluate pure database operators leading to the conclusion that they shall be optimized to benefit from PIM.

3.5 SUMMARY

We presented many related work for Near-Data Processing in memory devices since secondary memory until the primary memory (DRAM). We summarize the main related work in Table 3.1. While building this table, we only included related work for NDP on database operators, and thus excluded others that focused on specialized applications.

We also grouped the related work according to the memory devices: magnetic disks, flash disks, and DRAM. We emphasize the database operators that each work evaluated in someway. The interest group of database operators is disclosed in Section 5.1 (select, project, join, aggregation, and sort operators).

According to our study, at the end of the 1990s and during the decade of 2000, the industry and research community directed the attention to magnetic disks. They focused on design devices for memory processing using as use case many applications, including database operations. The advent of the flash memory technology changed the focus to flash disks (SDDs), and many NDP approaches arose from that.

Now, the NDP wave is rolling to the shore of in-memory processing devices. Nowadays, it is known as Processing-In-Memory. There are several proposal architectures for on-chip processing to address a myriad of applications, including scientific and mathematic workloads, machine learning algorithms, image processing, etc.

In Table 3.1 we summarize related work for database operators on PIM.Note that no approach encompasses all the database operators that we are investigating. In the next chapter, we detail the emerging PIM architectures, also we expose their internals, highlighting the prominent on-chip capabilities. The next chapter presents the details of the current PIM hardware and its potential to leverage database operations through a preliminary experiment.

Table 3.1: Related work summary of near-data processing for database operators.

Memory Device	**Related Work**	**Database Operators**				
		Select	Project	Join	Aggr	Sort
Magnetic Disks	Active Disks (AD) [13], 1998	X			X	X
	Riedel E. et al. [62, 63], 1998 to 2001	X		X	X	
	Smart Disk [64], 2000	X		X	X	X
	Steve C. el al. [65, 66], 2003; 2006	X		X	X	X
	Stoumpos V. el at. [70], 2006			X		
	Son S. W. et al. [71], 2006	X				
	David D. Chambliss et al. [72], 2008	X				X
Flash Disks	FlashScan; FlashJoin [79, 80], 2010	X		X		
	Kim, S. et al. [76], 2011	X				
	Smart SSDs [16], 2013	X			X	
	Intelligent SSDs (iSSD) [17], 2013	X				
	Ibex (MySQL) [85], 2014	X			X	
	SSD in-storage computing [86], 2016			X		
DRAM (PIM)	Widx [105], 2013	X		X	X	X
	NDCores [97, 106], 2014	X		X	X	
	Mirzadeh N. et al. [102], 2015			X		X
	JAFAR [101], 2015	X				
	NDP-Manager [94, 95], 2016; 2017					X
	HIPE [110], 2017	X				
	RVU [100], 2017	X				
	Mondrian [108, 109], 2017; 2018	X		X	X	X
	Tomé D. G. et al. [23], 2018	X	X	X		

4 PROCESSING-IN-MEMORY ARCHITECTURE

Supporting computational capabilities in-memory has been proposing since the end of the ’60s [111, 112, 113], i.e., in the beginnings of the main memory conceiving. At that time, the central idea was to add small logical circuitry into cells of cellular arrays to perform a desired logical behavior.

Over the years, according to the progress of memory technology, many approaches have been designed to incorporate computation into memory devices. For example, Active Disks [13], Active Storage [62], and Intelligent Disks [12] emerged at the end of the ’90s, they tried to add logical functions into the Hard Disk Driver (HDD). At the same time, Intelligent RAM [14, 114] revisited the first idea presented in cellular arrays by adding “intelligence” inside the DRAM device to support specific computations. Also, the Smart SSDs [16], Active Flash [15], and Intelligent SSDs [17] tried to embed logic units inside flash disks.

Nevertheless, those approaches were not adopted in commercial products in that time, because of the continuous growth in CPU performance complied the Moore’s Law and Dennard scaling. However, as the CPU performance has increased the main-memory access became the bottleneck for many applications, a problem known as the “memory wall” [115]. Today, the assumptions of Moore and Dennard come to an end, but the memory wall is still an issue in data-centric systems as well another recent problem: the “energy wall” [116, 117]. In data-centric systems, both walls are critical as large amounts of data move around the memory hierarchy from disks, main memory, and caches to the CPU. Moreover, the amount of data to store and to process is growing up tremendous and have accentuated both walls: a major challenge in the agenda of database researchers [118].

Nowadays, PIM architectures have emerged as a solution to tackle the performance aftereffects of data movement, which head applications to waste 62.7% of the total system energy, on average [1], and impose high latencies.

The rest of this chapter is organized as follows: We begin with an introduction of the internals of emerging 3D-Stacked Memory in Section 4.1. The state-of-the-art extensions for 3D-Stacked memory are introduced in Section 4.2. Further details of our PIM architecture baseline are presented in Section 4.3. Section 4.4 and 4.5 brings the potential of PIM on the selection query operator exposing the benefits of data access parallelism, high memory bandwidth, and on-chip processing capabilities. We summarize the PIM architectures in Section 4.6.

4.1 3D-STACKED MEMORY

The 3D-stacked memories are emerging PIM devices proposed to leverage the DDR technology as CPU’s main memory, they are also available on commercial graphic cards as GPU’s main memory [119, 120]. A typical 3D-stacked memory consists of up to 8 layers of DRAM dies interconnected by the TSV to the logic die at the base. The 3D memory devices logically split the DRAM dies into 32 independent vaults. Each vault contains up to 8 independent DRAM banks, where each DRAM bank provides as much as 256-bytes of data per row access. This 3D design achieves 512 parallel requests and can deliver a maximum bandwidth of 320 GB/s, which is about 4x higher than a traditional DDR-3 design. The logic layer of 3D-stacked memories supports the implementation of traditional logic, similar to those present inside processors. In the case of the HMC proposal, it implements update operations performing arithmetic, logical, and bit-wise atomic instructions over scalars of up to 16 bytes size.

Figure 4.1 presents such an architecture with the memory and logic layers, and the external links to receive memory requests and on-chip instructions. The memory layer comprises of 4 DRAM dies in a stacked design. The DRAM dies are split into 32 vertical memory partitions (**vaults**), each one with 8 memory banks (*B*0 to *B*7). The memory banks *B*0 and *B*1 belong to the die at the bottom of the stack, banks *B*2 and *B*3 belong to the die above and so on. A single bank has several memory rows of 256-bytes, as depicted on top of Figure 4.1 for *B*7. The TSV technology connects banks from the memory layer to the corresponding vault logic unit within the logic layer. The Vault Controller (VC), inside the logic unit, manages data accesses to the memory layer using read and write buffers. A data request encapsulates a physical memory address then the VC uses it to fetch a memory row of 256-bytes. The VCs are independent by accessing distinct memory banks. Thus, applications can emit 32 parallel memory requests of 256-bytes reaching the high bandwidth up to 320 GB/s. Also, in the logic layer, the on-chip logic instructions are executed on operands up to 16 bytes.

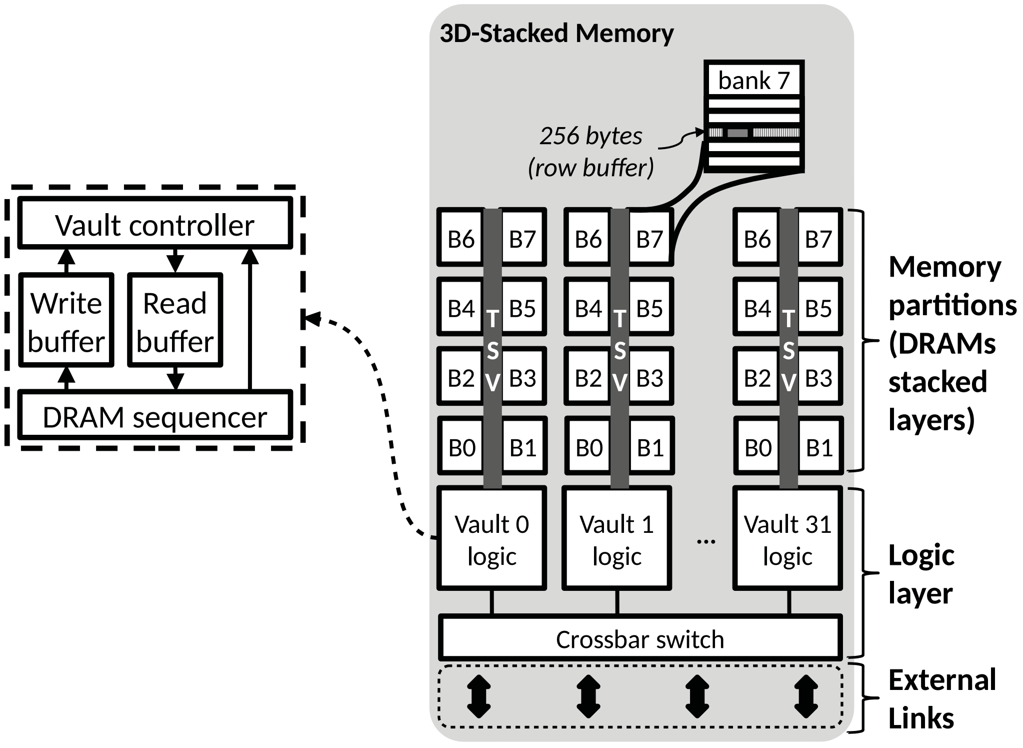


Figure 4.1: A 3D-stacked memory architecture comprised of memory and logic layers, and external links to receive memory requests and PIM instructions [18].

4.2 3D-STACKED MEMORY EXTENSIONS

The 16-bytes granularity of the on-chip instructions on current 3D-stacked memories [18, 19] inhibits the benefits of PIM. Therefore, research work extended the logic layer of 3D-stacked memories. In this section, we briefly expose the main extensions from the state-of-the-art. Figure 4.2 depicts the three architecture extensions: **HIVE** [99], **RVU** [100], and **HIPE** [110].

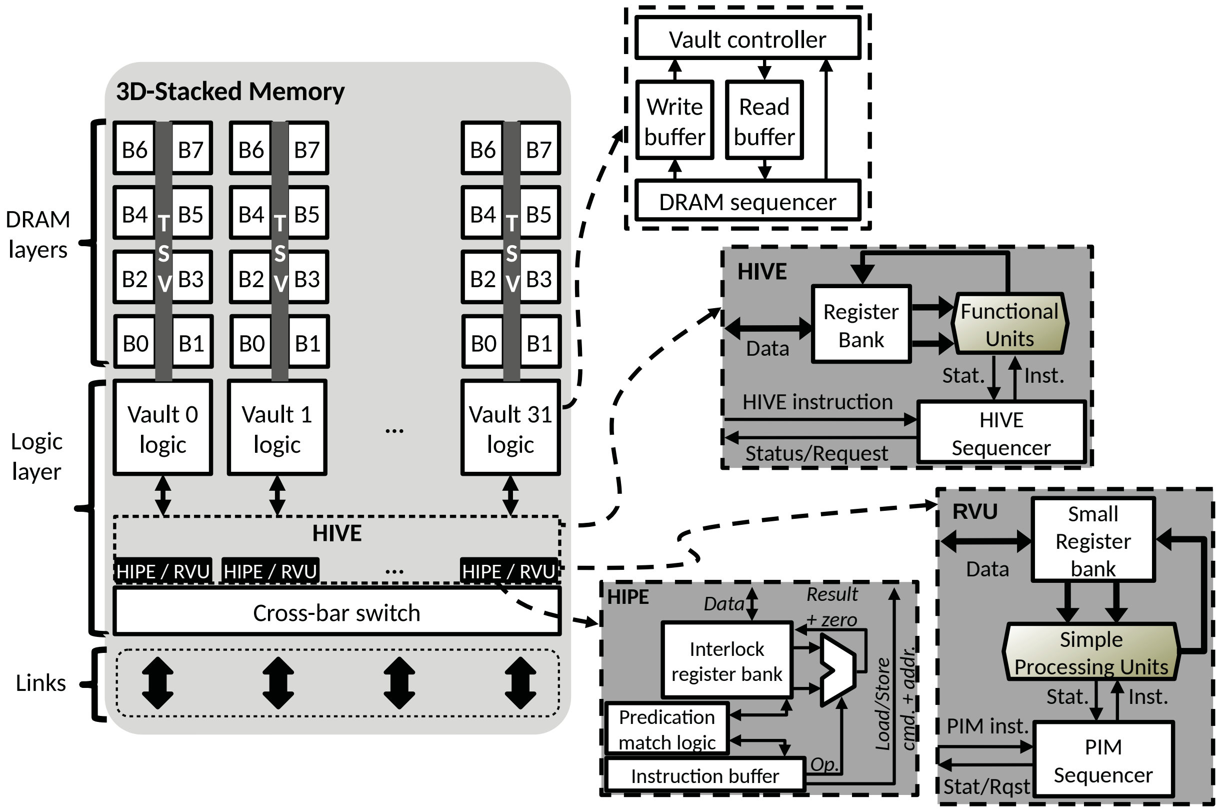


Figure 4.2: State-of-the-art PIM architectures [99, 100, 23, 110].

HIVE. [99] incorporated into the logic layer a vector processing hardware to provide vectorized instructions that operate over coalescing memory data at granularity from 256-bytes up to 8 KB (very large registers). Therefore, HIVE allies the maximum data access parallelism of 3D memories with on-chip processing. In next Section 4.3, we present more details of the HIVE's architecture.

RVU. [100] proposed a Reconfigurable Vector Unit (RVU) over the HIVE that allows vector processing in small units of variables sizes. RVU is compelling to run database operators that access intermediate data of different granularities during query execution. In column-oriented DBMSs, the *select scan* with a chain of predicates on different columns can take advantage of the RVU, because each predicate may produce intermediate results with different sizes according to the query selectivity.

HIPE. [110] designed HIPE as an HMC extension for instruction predication. HIPE incorporates functional units to support predicated execution inside the memory, thus transforming control-flow dependencies into data-flow ones. Applications with many branch instructions (*if-then-else* instructions) might take advantage of HIPE.

4.3 HIVE: INSTRUCTION VECTOR EXTENSIONS

Our implementations of database operators and experiments are based on HIVE. We chose HIVE because it has a more simplistic logic layer that can support SIMD instructions already implemented in the Intel AVX512 technology. Such SIMD intructions were evaluated on in-memory database systems [121]. Thus, in this section, we further detail the HIVE's architecture with its on-chip SIMD processing capabilities.

HIVE is a different logic layer design to propose larger registers with SIMD parallelism inside the 3D-staked memory [99]. Although HIVE foresees the feasibility of SIMD instructions operating over SIMD registers from 256 B up to 8 KB, we will use a modest size of 256 B per operation. Thus, our PIM-256B architecture works with registers of 256 bytes wide that shall store multiple operands. For simplicity, we call each operand position inside a SIMD register as a lane (e.g., when using 32-bit operands, a single 256 B register may contain up to 64 valid lanes). Similar to the HMC proposal, HIVE also relies on the CPU to trigger instructions to be executed inside the memory. We use HIVE in our experiments due to its simplicity and low energy consumption (not requiring a full processor inside the memory), its high performance (with previous work showing more than 10$\times$ gains), and its acceptance as it was used to implement derived architectures [100, 110].

Figure 4.3 describes the PIM architecture with SIMD support from HIVE (to the right side) and the traditional x86 architecture inspired by the von Neumann model (to the left): formed by the processor core and a detached cache hierarchy. At the top of Figure 4.3, the processor dispatches PIM instructions (dashed line) directly to the PIM device bypassing the caches while maintaining the coherence with the Last-Level Cache (LLC) directory. The instructions to access memory (load/store) might require accessing up to 256 bytes each. The 32 independent vaults allow 32 PIM SIMD-like load instructions of 256-bytes at a time. However, this high level of parallelism depends on the memory access pattern from the application. During a memory load from PIM logic, the data request goes to a specific DRAM bank inside a designated vault (using the load address to indicate the correct device). Once data is available, the Vault Controller transfers it to the PIM SIMD register bank in which every register implements a ready bit (interlock mechanism), and thus the operations only continue whenever case that bit is set. At the end of the execution of each instruction, the PIM device only returns the instruction status to the CPU. This data-centric design is the main advantage compared to current database systems that filter data in hardware before passing to the CPU, like Netezza and Exasol. They have to deal with packing qualifying tuples into condensed pages to avoid unnecessary bufferpool pollution, which is expensive and error-prone. Therefore, the significant benefits to be explored in the

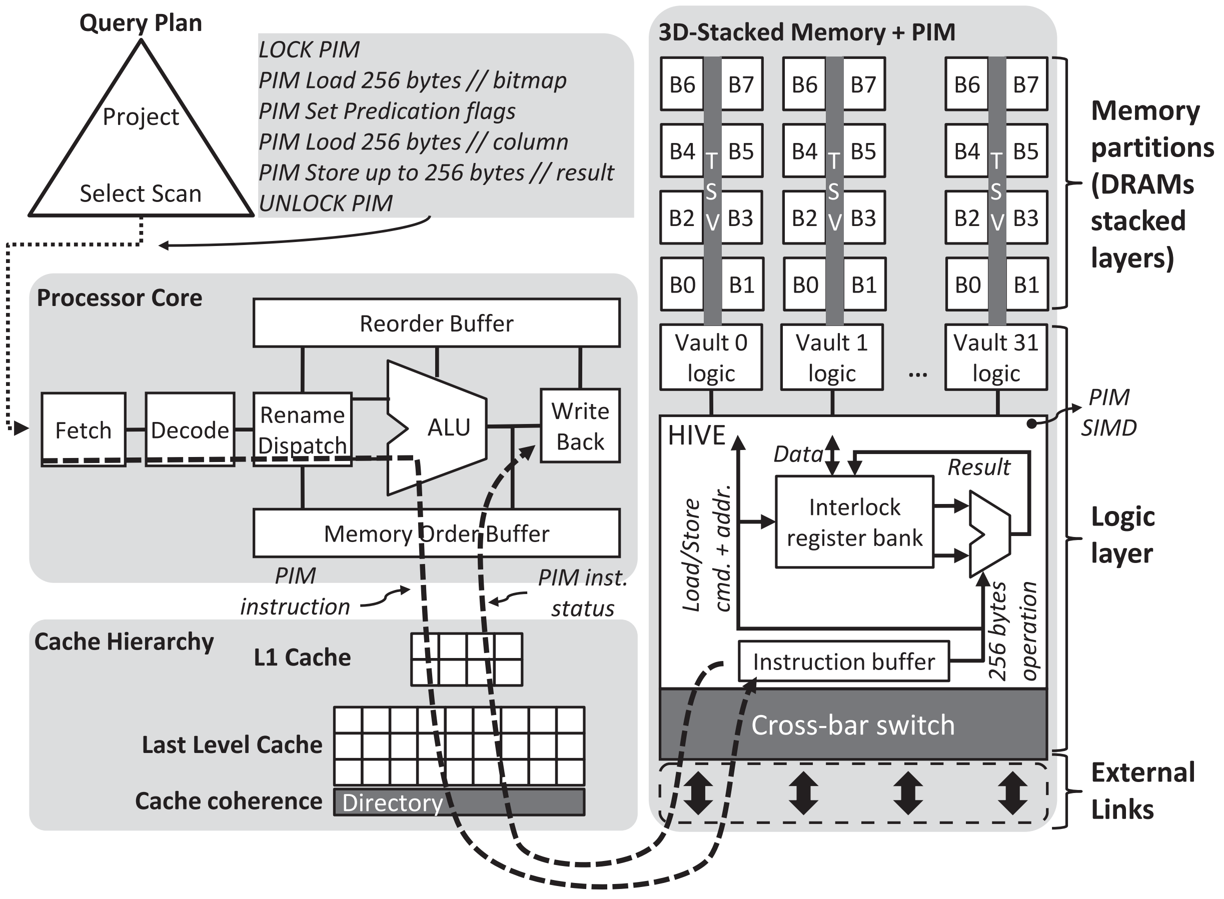


Figure 4.3: A query datapath movement in a traditional von Neumann architecture plus a modern 3D-stacked memory with PIM and SIMD support [99].

Database-PIM co-design are the drastic reduction in energy consumption and the internal high memory bandwidth due to the high levels of data access parallelism and on-chip processing.

Other capabilities of PIM include memory protocols to support all the idiosyncrasies of PIM instructions, such as cache coherence, Memory Management Unit (MMU), Error-Correcting Code Memory (ECC) and Direct Memory Access (DMA). The execution flow works at instruction-granularity as the traditional CPU processing (e.g., AVX/SSE x86-extensions), i.e., programmers insert intrinsics PIM instructions into the code, like Intel Intrinsics, and the compiler flags them as special memory PIM instructions.

4.4 UNDERSTANDING THE PIM SELECTION

To understand the impact of PIM with SIMD on query processing and to simplify our analysis, we initially focus on the execution of the selection operator, instead of more complex operations (like join). In the traditional selection operator, the memory requests start from the CPU to the main memory reaching all levels of cache, moving data up and down through the memory hierarchy.

As the traditional von Neumann architecture detaches the processor from the main memory, the data movement is an inherited side effect that memory caches try to alleviate. The caching mechanism is particularly efficient for applications with data reuse. However, this is not the case of the selection operator because it streams datasets polluting the hierarchy of memory caches with dead-on-arrival cache lines. Even the selection with an index (select-index) has the same streaming behavior. The selection stays restrict within the indexed-data portions while streaming the data.

The selection operator appears as a good fit for PIM because the SIMD logical units of on-chip processing better exploit the high internal bandwidth of 3D stacked memories. Figure 4.4(a) depicts the execution of the selection using PIM with SIMD support. A dataset can be either an entire table, a column, or even a chunk of an indexed-data portion. The selection operator performs sequential memory access to process fragment-at-a-time of 256 B (S_0 to S_n in Figure 4.4(a)). Figure 4.4(b) presents the C language code of the selection operator and the respective translation to the PIM Assembly-like code. As a simplistic use case, we generate the output of the selection as a bitmap, although it is also possible to emit a selection vector as output.

The PIM instructions used in that code snippet are (see Figure 4.4(b)):

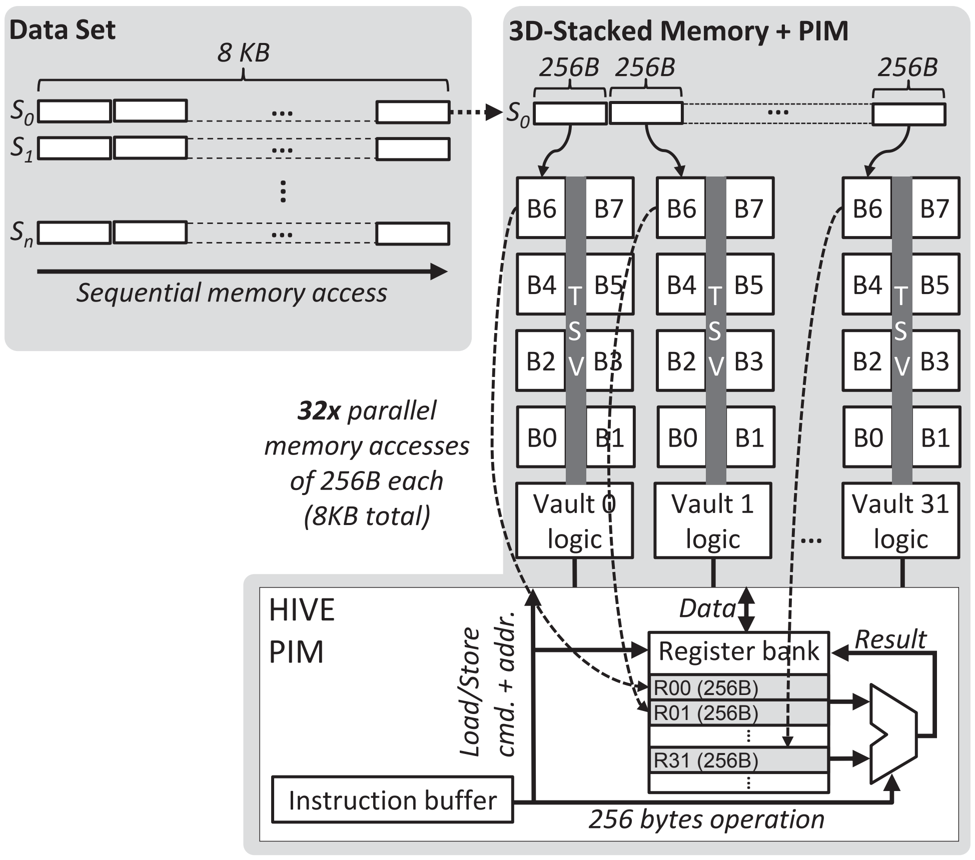
1. **PIM_LD**: instruction to load data from a DRAM bank into a specific register (e.g., *V0*).
2. **PIM+SIMD_CMP**: instruction to compare two PIM-SIMD registers (e.g., *V0* and *VF*).
3. **PIM_ST**: instruction to store data from one specific register (e.g., *V0*) into a DRAM bank of a vault.

During the execution of the selection operator in HIVE, the CPU sends the PIM instructions for on-chip processing. Inside the logic layer, HIVE interprets and executes each instruction. During bursts of memory loads, up to 32 parallel reads can be performed by HIVE, using all the throughput of the memory vaults (up to 320 GB/s) [18, 19]. Although it is possible to issue multiple loads in parallel, the execution follows strict in-order fashion. We observe that all the registers can receive data from any memory vault, during memory loads, since HIVE is coupled with the interconnection of the vaults. For the first instruction, HIVE loads 256 bytes of data ($data[i]$) from one specific memory vault into the SIMD register bank. Then, the **PIM+SIMD_CMP** instruction compares the PIM+SIMD loaded register and the SIMD register of filter (*VF*): a pre-load PIM+SIMD register that has replicas of the filtering value. In the end, the **PIM_ST** instruction writes the resulting bitmap into a given memory vault.

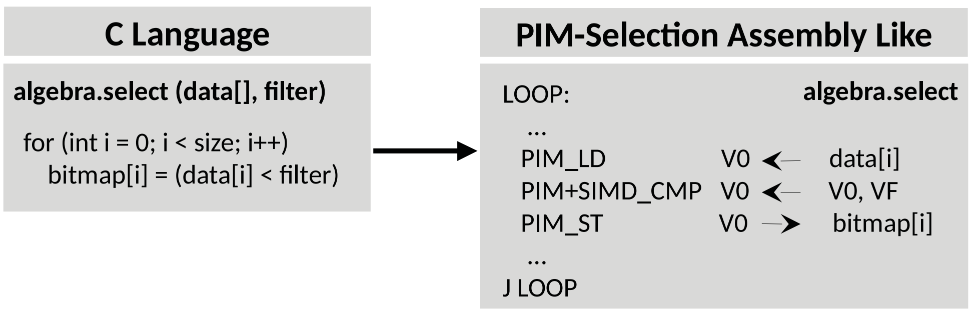
4.5 THE POTENTIAL PARALLELISM OF PIM-256B VS. X86 AVX512-64B

In this section, we briefly highlight the potential parallelism of PIM compared to the x86 processor for processing selections (Chapter 5 presents the details of the experiments).

The x86 version of the selection operator uses AVX512 extensions with SIMD registers of 64 bytes (AVX512-64B), and the PIM version uses SIMD registers of 256 bytes (PIM-256B). Notice that AVX512-64B uses the largest SIMD registers available for the traditional



(a) Selecting data using PIM + SIMD support.


C Language

```
algebra.select (data[], filter)

 for (int i = 0; i < size; i++)
    bitmap[i] = (data[i] < filter)
```

PIM-Selection Assembly Like

```
LOOP:                          algebra.select
  ...
  PIM_LD          V0 <-    data[i]
  PIM+SIMD_CMP    V0 <-    V0, VF
  PIM_ST          V0 ->    bitmap[i]
  ...
J LOOP
```

(b) The selection operator code in C and PIM Assembly-like.

Figure 4.4: The selection operator in PIM.

x86 processor. We also applied the loop unrolling technique to push the architectures to the maximum degree of parallelism available, i.e., the AVX512[1] processing up to unroll depth of 8× and PIM up to 32× to take advantage of the 32 independent vaults.

This experiment measures the latency (execution time) of the operator varying the size of the input dataset to fit into the L1 or L2 caches. Figure 4.5 shows an appealing case for the x86 processing due to a small dataset processing with a low ratio of cache misses. Our goal is to show the potential of PIM even in unfavorable cases. In datasets bigger than cache sizes, the cache misses degrades the performance of the x86 processing.

[1]Generally, 8× is the deepest unroll implemented by compilers due to the reduced number of general purpose registers.

The three foremost benefits in here for PIM processing are: 1) Only a single load inside PIM-256B shall retrieve up to 256 B whereas the AVX512-64 requires 8 operations to access the same amount of data; 2) Considering 4 B operands (e.g., integer variable) the PIM-256B shall operate over 64 elements (lanes), while AVX512-64 operates over only 16 by the same time; 3) It is possible to considerably reduce the number of data transfers between CPU and main memory operating directly inside the memory for streaming data patterns. Based on those benefits, the PIM execution is 3× faster than AVX512 for both datasets when using all the memory vaults. In Chapter 7, we provide an in-depth analysis of the results for many query operators.

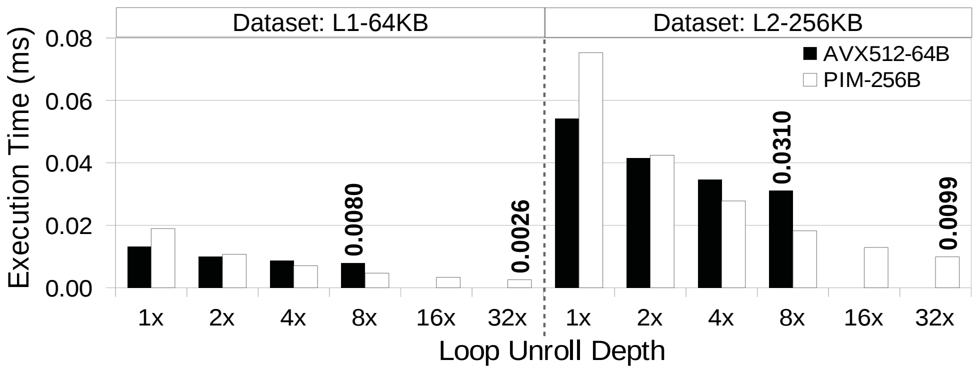


Figure 4.5: The execution time of the selection operator for AVX512-64B and PIM-256B. The dashed line separates the X-axis in two data sets: one fits in cache L1 and the other in cache L2. Also, we ranged the loop unroll depth from 1× up to 32×, which implies in varying the degree of parallelism.

Although x86 ISAs provide load instructions that bypass the cache memories, it is important to notice that off-chip communication is still present, consuming time and energy. Processors usually can only perform 10 parallel requests per processing cores (due to MSHR - miss status handler register - limitations), resulting in total parallelism of 10× 64 bytes (640B), which is smaller than the parallelism of PIM, i.e., 32× 256-bytes (8KB).

4.6 SUMMARY

In this chapter, we presented the current PIM architectures, introducing the particularities of emerging 3D-stacked memories and the state-of-the-art design extensions, i.e., HIVE, RVU, and HIPE. Our experiments use the 3D-stacked memory, presented in Section 4.1, as the main memory for both architectures x86 and PIM, making a strong case for our comparisons. Also, we further detailed the on-chip SIMD processing capabilities of HIVE that we applied to our experiments.

We disclosed the implementation of the selection operator on PIM and its potential to exploit the data access parallelism and on-chip processing. In a preliminary performance comparison, the selection operator was 3x faster on PIM over the AVX512 execution even against unfavorable instances, i.e., for input datasets that fit into the L1 and L2 caches. Next chapter brings our experiment design to evaluate the query exeuction systems on the ground of PIM support.

5 EXPERIMENT DESIGN

In this chapter, we detail the design of our experiments. In Section 5.1, we describe how we choose the query operators for evaluation in this study. Then, in Section 5.2, we present the workload data distribution. The simulation environment used allover this thesis are presented in Section 5.3. Also, Section 5.3.1 brings an empirical demonstration of the simulation stability of our simulator. We summarize the experiment design in Section 5.4.

Our evaluation metrics take into consideration the operator execution time and energy consumption. In the micro-benchmark analysis, each operator is evaluated in isolation, with no interactions among them, except in the pipelined execution that requires such interaction. For the operator latency, we record the execution time. For energy consumption, we measure the memory read, write, and data transfer operations. We compute the memory energy estimation of the DRAM values considering the architecture of the current 3D-stacked memories [18, 19]. In the macro-benchmark analysis, we evaluate the whole query execution.

5.1 CHOOSING THE GROUP OF OPERATORS

In this section, we investigate the most time and memory consuming database operators to justify a relevant group of operators in our study. First, we investigate the response time breakdown of the TPC-H queries with 100 GB using the column-wise database MonetDB v11.33.11 (available at [22]). We carried out the experiments on a real machine using an Intel quad-core i7-5500U processor running at 2.40 GHz with 16 GB of RAM (DDR-3L 1333/1600) and 4 MB LLC running OpenSuse Leap 42.3 on Linux kernel 4.4.76-1-default. We added the TRACE statement modifier of MonetDB on each query to collect statistics and performance traces.

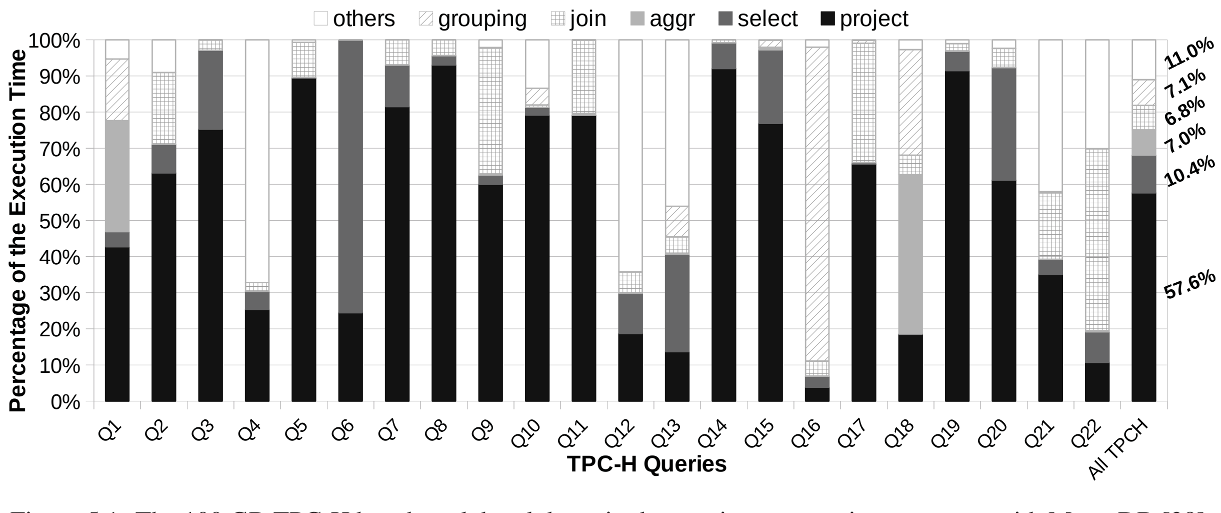


Figure 5.1: The 100 GB TPC-H benchmark breakdown in the top time consuming operators with MonetDB [38].

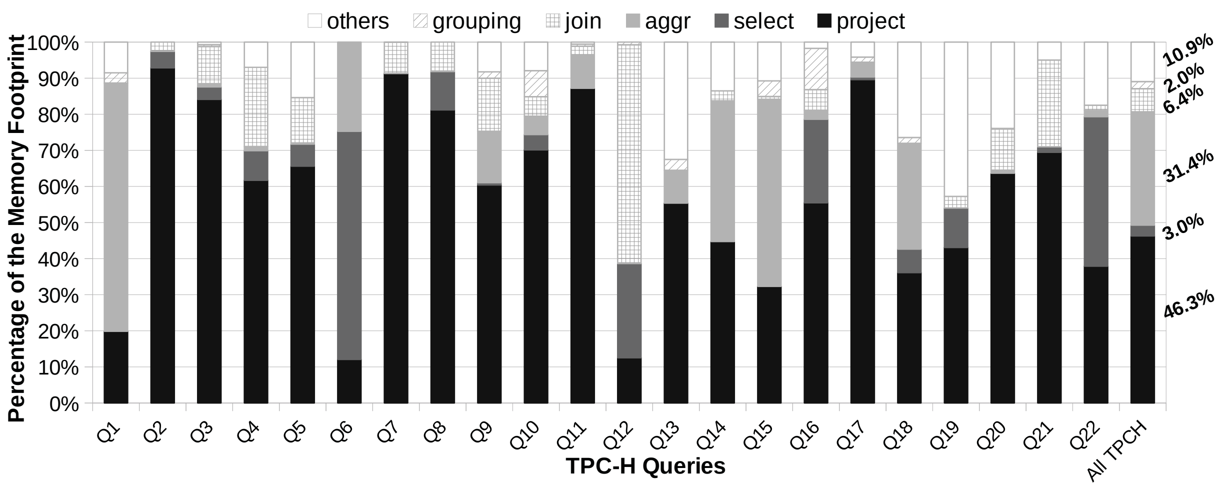


Figure 5.2: The 100 GB TPC-H benchmark breakdown in the top memory consuming operators with MonetDB [38].

Figure 5.1 and 5.2 presents the query execution breakdown plotting the most time and memory, respectively, consuming operators: projection, selection, join, aggregation, grouping, and the remnant ones grouped into the category "others". The last bar summarizes the entire benchmark ("All TPCH"). From those results, we set as the relevant group of operators the projection, selection, join, and aggregation, as they represent almost 90% of the 100 GB TPC-H benchmark for execution time and memory usage.

5.2 WORKLOAD'S DATA DISTRIBUTION

Our goal in the design of the data distributions is to evaluate the impact of different memory accesses. We study this impact in two cases: 1) The case when the input datasets fit into the cache hierarchy; 2) When they do not. In theory, the first case is the best one for the x86 processing because the operators can take advantage of the caching mechanism for data reuse. We assume three particular queries: 1) The TPC-H Query 01 is a low-cardinality group query without joins (fitting inside the cache memory). Most of its execution time is spent projecting columns and computing the aggregation; 2) The TPC-H Query 03 is a high-cardinality group query with joins (does not fit inside the cache memory). Most of its execution time is spent filtering and projecting columns. We run the query operators varying the size of the input columns to fit in the L1, L2, LLC caches, and in DRAM with at least 1 GB; 3) In the following third query, we evaluated the aggregation operation with the Zipf distribution in the caches (L1, L2, and LLC) and the DRAM of 1 GB, for convention we call this query as ZIPF Query.

```
ZIPF Query:  SELECT sum(col_zipf) FROM table GROUP BY col_zipf
```

The Zipf distribution presents a bias based on the frequency of the values, which we use to simulate random memory access to the groups in the hash table of the aggregation operator. As a result, some groups are more accessed than others generating data reuse in the memory caches.

Table 5.1 summarizes the workload and the datasets used in our study. The workload is composed of the three queries (TPC-H Q03, TPC-H Q01, and ZIPF Query) split into the operators: selection, projection, join, sort, and aggregation. The internal cells in the table represent the size of the dataset applied. For example, the first cell of the selection operator for the TPC-H Query 03 contains the value *|L1-64KB|*, which means that all columns and intermediate data structures fit into the cache L1 (64-KBytes), the other cells have the same meaning. We evaluate all the operators using the TPC-H Query 03. With the other two queries, we further analyze the aggregation operator.

Table 5.1: Workload and Dataset Summary.

OPERATORS	**WORKLOAD**		
	TPC-H Q03	**TPC-H Q01**	**ZIPF Query**
Selection	|L1-64KB|	-	-
	|L2-256KB|		
	|LLC-8MB|		
	|DRAM-1GB|		
Projection	|L1-64KB|	-	-
	|L2-256KB|		
	|LLC-8MB|		
	|DRAM-1GB|		
Join	|L1-64KB|	-	-
	|L2-256KB|		
	|LLC-8MB|		
	|DRAM-1GB|		
	|DRAM-2GB|		
	|DRAM-4GB|		
Sort	|L1-64KB|	-	-
	|L2-256KB|		
	|LLC-8MB|		
	|DRAM-1GB|		
	|DRAM-2GB|		
	|DRAM-4GB|		
Aggregation	Original Size TPC-H 1GB	Original Size TPC-H 1GB	|L1-64KB|
			|L2-256KB|
			|LLC-8MB|
			|DRAM-1GB|

5.3 SINUCA: A VALIDATED MICROARCHITECTURE SIMULATOR

We implemented the PIM architecture on top of the SiNUCA (available at [122]) cycle-accurate simulator [123]. Notice that current PIM hardware do not yet implement all the extensions depicted in Figure 4.3. Therefore, we rely on architectural simulators to implement the required hardware extensions for our study, which is the standard approach adopted by processor industries and hardware research [124]. Using SiNUCA, it is possible to execute the database operators in the simulated environment obtaining performance results for x86 and PIM executions. SiNUCA was validated against two real machines [123] that implements a realistic out-of-order processor, advanced multi-banked, and non-blocking caches together with the PIM hardware. Furthermore, SiNUCA was adopted by studies that extend PIM hardware in computer architecture [110, 100], and database [23, 25] contexts.

The baseline architecture was inspired by the Intel Sandy-Bridge microarchitecture that we extended with the AVX-512 instruction set capabilities referred to as AVX512-64B. Although this microarchitecture does not represent the state-of-the-art, the memory bottleneck is still an unsolved problem for newer architectures that rely on the computing-centric design yet. In all the cases, the traditional main memory is a high-bandwidth 3D-stacked memory [18, 19]. Moreover, by adding 3D-stacked memory to this architecture, we are virtually providing up to 32 channels to the x86 processor, which is more than 5× higher than the 2019's Cascade-Lake processor will offer. Table 5.2 presents the parameters of the target architectures with the same setup used by related work [99, 110]. The PIM architecture has 32 vaults with 8 DRAM banks per vault, and the total memory capacity is 8 GB. Also, this architecture has 36 SIMD registers of 256 bytes that operate with operands from 4 to 256 bytes.

Table 5.2: Parameters of the target architectures taken into account to design the experiments [110].

OoO Execution Cores 16 cores @ 2.0 GHz, 32 nm; 6-wide issue; 16 B fetch; Buffers: 18-entry fetch, 28-entry decode; 168-entry ROB; MOB entries: 64-read, 36-write; 1-load, 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle); 1 branch per fetch; Branch pred.: Two-level GAs. 4,096 entry BTB;
L1 Data + Inst. Cache 32 KB, 8-way, 2-cycle; Stride prefetch; 64 B line; MSHR size: 10-request, 10-write, 10-eviction; LRU policy;
L2 Cache Private 256 KB, 8-way, 4-cycle; Stream prefetch; 64 B line; MSHR size: 20-request, 20-write, 10-eviction; LRU policy;
L3 Cache Shared 40 MB (16-banks), 2.5 MB per bank; LRU policy; 16-way, 6-cycle; 64 B line; Bi-directional ring; Inclusive; MOESI protocol; MSHR size: 64-request, 64-write, 64-eviction;
PIM device 32 vaults, 8 DRAM banks/vault; DRAM@166 MHz; 8 GB total size; 256 B Row buffer; Closed-page policy; 8 B burst width at 2:1 core-to-bus freq. ratio; 4-links@8 GHz; DRAM: CAS, RP, RCD, RAS, CWD cycles@166 MHz (9-9-9-24-7);
SIMD units Unified func. units (integer + floating-point) @1 GHz; Latency (cpu-cycles): 2-alu, 6-mul. and 40-div. int. units; Latency (cpu-cycles): 10-alu, 10-mul. and 40-div. fp. units; Op. sizes (bytes): 4, 8, 16, 32, 64, 128, 256; Register bank: 36x 256 B (Originally 16x 8192 B in HIVE proposal);

5.3.1 Validating the Simulation Stability of the Selection Operator

Now, we discuss the simulation stability when the dataset enlarges gradually. Our goal is to show that the simulation of the operators still steady in the face of changes in the size of the dataset. For simulation stability, we mean that the results of the simulations have a proportional changing according to the size of the input dataset:

Definition 1. Simulation stability between consecutive dataset.

Let us consider two consecutive datasets with the number of records varying in one:

R_n: the simulation result for dataset D_n with n records.

R_{n+1}: the simulation result for dataset D_{n+1} with $n+1$ records, i.e.:

$R_{n+1} = R_n + P + \varepsilon$, such that P is a proportional changing of the simulation result that depends on the operator, and ε is the inherent error of simulations[1].

Based on Definition 1 and as a proof of concept, we empirically evaluated the stability of the selection operator. We vary, one by one, the size of the input dataset to be processed: the number of records in the dataset range from 1 to around 509,000 (until the size of the LLC), which fits into the cache hierarchy. For every dataset, we record the number of cycles and energy consumption, and then we accumulate the aggregated average for both metrics so far.

Figure 5.3 presents the graphics with the aggregated average for both metrics. We cut off and zoom up the inflection points of the aggregated curves. After these points, around 400 and 1,200 records, the number of cycles and energy consumption spend by *AVX512-64B* become greater than the *PIM-256B*, this result continues *ad aeternum* due to the limit property of the functions AVX() and PIM(), i.e.:

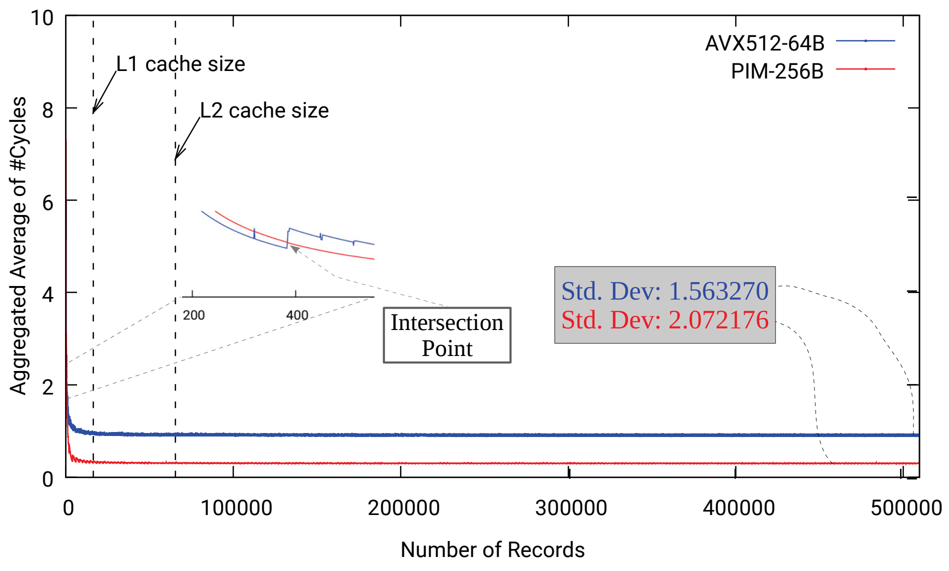
$$\lim_{n\to\infty} AVX(n) > \lim_{n\to\infty} PIM(n) \;\mid\; AVX(n),\, PIM(n) \;>\; 0 \tag{5.1}$$

Therefore, Figure 5.3 shows that the behavior of the operator stays constant after a certain input size. This means that the number of cycles or energy spent per record was kept constant whenever the data did not fit into the LLC (i.e., the average time spent per entry stays constant no matter if we were using 1 GB or 100 GB of data). The stability allows extending the trend of both metrics to the size limit of the DRAM. Thus, we can justify, quantitatively and qualitatively, the use of the SiNUCA simulator to proceed with our study.

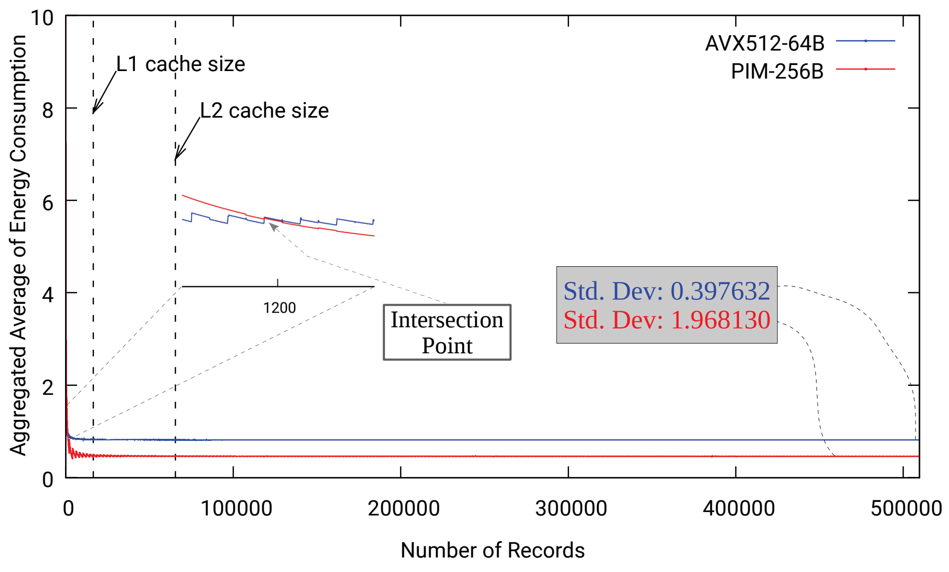
5.4 SUMMARY

We detailed the experiment design used in this study. Firstly, we identified the most relevant group of database operators on a modern DBMS. Note that the TPC-H workload, by on average, spends around 90% of the execution time and memory footprint to process that group of operators. Therefore, they are a good fit for PIM. Also, we present the set of queries used

[1]SiNUCA achieves an average performance error of less than 9% [123].



(a) Aggregated average of the number of cycles.



(b) Aggregated average of the energy consumption.

Figure 5.3: Selection Stability on SiNUCA.

as our workload split into the operators and the dataset applied in each one. Our evaluation metrics consist of execution time and energy consumption. Finally, we present the details of the SiNUCA cycle-accurate simulator and the architecture parameters modeled in the simulation, and empirical evaluation of the simulation stability to corroborate the adoption of SiNUCA. The next chapter details the SIMD instructions used to implement the database operators.

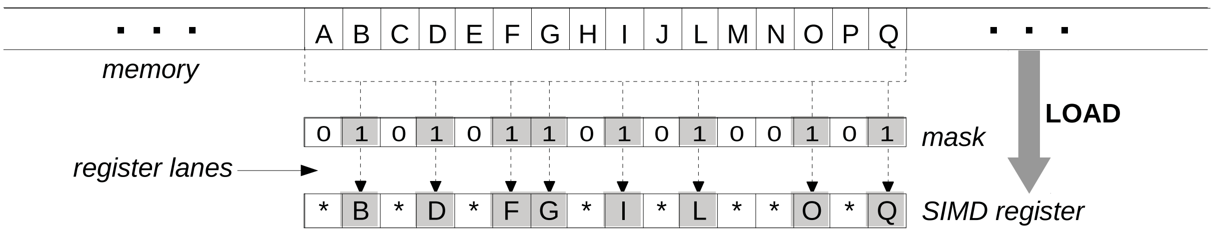
6 IMPLEMENTATION DETAILS OF THE SIMD QUERY OPERATORS

In this chapter, we describe the implementation details of the query operators with SIMD. We also describe the relevant SIMD vectorization features applied in the operators.

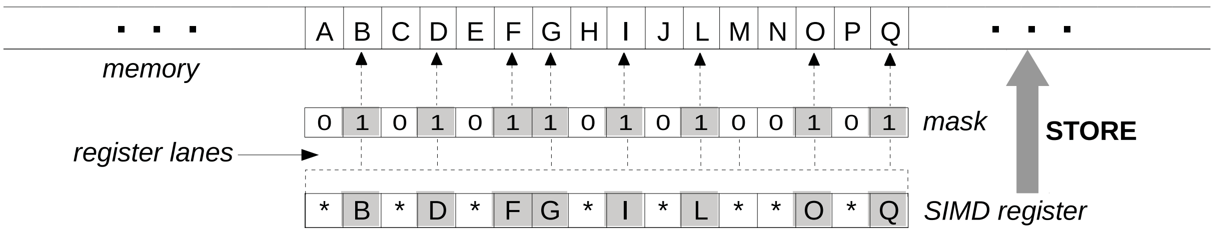
In a nutshell, the implementations of the database operators require selective *load* and *store* SIMD memory instructions (Section 6.1). However, each operator demands a different strategy to better use SIMD instructions. The hash join and aggregation require the *gather* and *scatter* SIMD memory instructions to load and store multiple entries of hash tables (Section 6.2). Finally, the sorting operation and sort-merge join require the *min/max* and *shuffle* SIMD instructions (Section 6.3). Section 6.4 recaps this chapter.

6.1 SIMD SELECTION AND PROJECTION OPERATORS

Selection. The selection operator filters data and generates a bitmap with bits set to 1 for qualified data. We discuss our two SIMD selection implementations with an example of a chain of selections. Figure 6.1(a) depicts the **selective load** SIMD instruction using a bitmap as a bitmask to filter the next selection column from a contiguous memory location. In a chain of selections, the output bitmap of an operator is the input to the next one. Another common implementation of this operator generates a selection vector as output. The output is a SIMD register with the values arranged according to the input selection vector (e.g., the index register in Figure 6.2(a)). In a chain of selections, the **gather** instruction reads the next column from a non-contiguous memory location based on the selection vector from the previous selection.



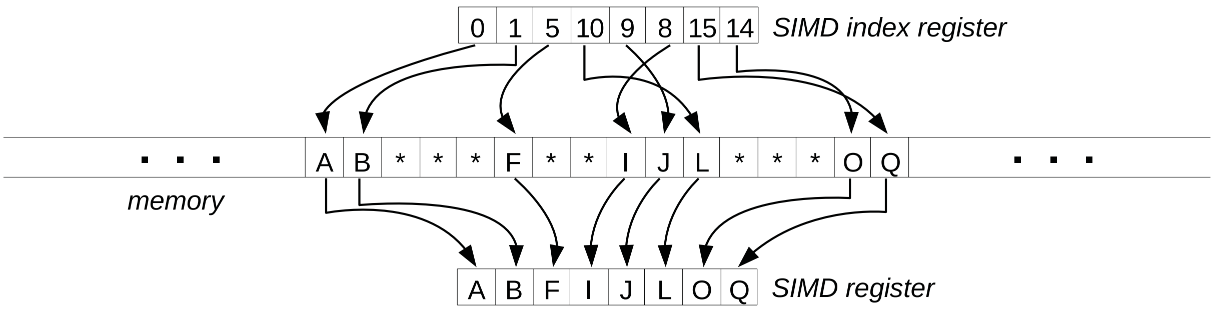
(a) Selective Load (Selection).



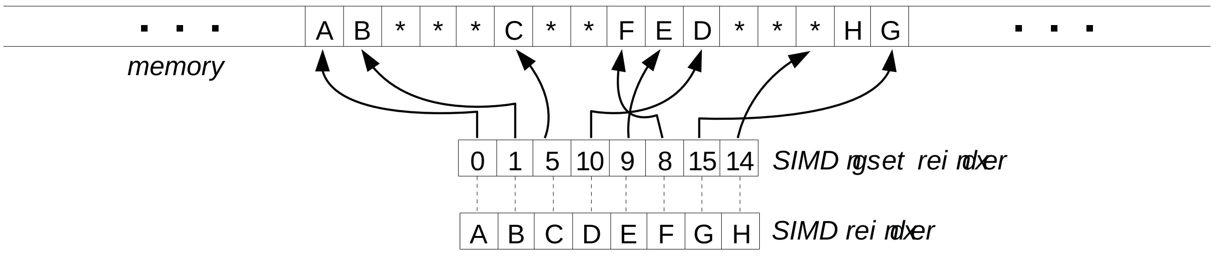
(b) Selective Store (Projection).

Figure 6.1: SIMD memory instructions based on bitmask.

Projection. We present two implementations of the projection operator: 1) Projection with an index register for high selectivity queries, like the selection vector of MonetDB [38], and 2) Projection without an index register for low selectivity queries reducing the memory footprint. Our first implementation uses the *selective store* to project the target column without an index register. Figure 6.1(b) shows an example of this execution. The projection writes data from a subset of register lanes to a contiguous memory location. In the example, the output bitmap generated by the selection is the input of the projection, where the bits set to '1' indicate the values to project. In our second implementation using an index register as input, the projection uses the **scatter** memory instruction to write the non-contiguous values of the target column (see Figure 6.2(b)).



(a) *Gather* Instruction.



(b) *Scatter* Instruction.

Figure 6.2: SIMD memory instructions based on index register.

6.2 SIMD HASH JOIN AND AGGREGATION

Hash Table and Hash Join. Our implementation of the hash join is based on a vectorized SIMD-friendly linear building and probing algorithm [121]. We use the **gather** and **scatter** memory instructions[1] to implement hash tables for the join and aggregation operations. The **gather** instruction loads multiple entries of the hash table (non-contiguous memory locations). The **scatter** is the symmetric instruction that writes data to multiple memory locations based on an index register. For the x86 implementation, those instructions iterate (loop iteration) over the index register, identify the register lanes pointing to the same cache line, then read/write one or two cache lines per iteration until there are no more indexes to process. For the PIM

[1]The opcodes for those instructions are **vpgatherdd** and **vpscatterdd** in the Intel AVX-512 ISA, respectively.

implementation, the instructions iterate over an index register, group the register lanes pointing to the same DRAM banks, and generate up to 32 load/store instructions of 256-bytes per iteration until there are no more indexes to process.

Aggregation. The aggregation operator updates the aggregated values into the hash table using **gather** and **scatter** memory instructions. It **gathers** multiple entries from a hash table and applies the conflict-free [125] to update the aggregation values. Then it **scatters** them back to the hash table.

6.3 SIMD SORT-MERGE ALGORITHM

Sorting. Now, we discuss the implementation of the Sort-Merge algorithm that outperforms other sorting algorithms when exploiting the SIMD instructions [126]. The implementation of the Sort-Merge algorithm in SIMD is more intricate than the previous query operators. Both sort and merge phases of the algorithm rely on SIMD **min/max** and **shuffle**[2] instructions available on current SIMD processors [127].

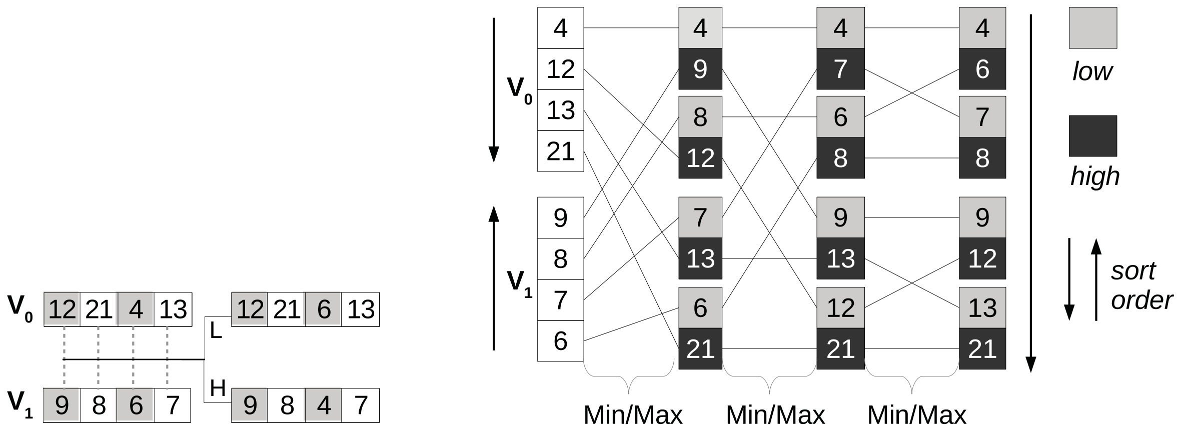
The **min/max** instructions process two SIMD registers V_0 and V_1 of length k (where k is the number of register lanes). These instructions compare the corresponding lanes of the registers and emit as output a new SIMD register that contains the lowest/highest values between V_0 and V_1, respectively. Figure 6.3(a) exemplifies those instructions that receive as input the SIMD registers V_0={12,21,4,13} and V_1={9,8,6,7}. The **min** instruction emits as output the SIMD register L={9,8,4,7} with the lowest values of each lane (dashed gray lines) between V_0 and V_1, and the **max** instruction emits as output the SIMD register H={12,21,6,13} with the highest values.

The **bitonic merge** is a networked merge algorithm that compares every element of two SIMD registers. The execution requires one register sorted in ascending order and the other in descending order. Figure 6.3(b) shows a **bitonic merge** network with two 4-wide SIMD registers (k=4). The network has log_2^k levels applied in parallel. Therefore, the execution of the whole sort-merge requires $2\ log_2^{2k}$ **min/max** and $1+2\ log_2^{2k}$ **shuffle** instructions.

Our SIMD sort instruction consists of two operations: the in-register SIMD Sort and in-block-register Merge. The former sorts a SIMD register with k lanes using an odd-even sorting network. First, it sorts the register lanes by applying successive *min/max* instructions. Then, each lane is sorted (shown as the gray and white lanes in Figure 6.4(a)). Finally, it applies a series of **shuffle** instructions to transpose the k sorted lanes (vertical order) to form k sorted registers (horizontal order).

Figure 6.4(b) brings a general overview of our in-register SIMD Sort with eight registers. The process has two steps: 1) It compares all registers to distinguish the overall lowest and highest values of each lane, resulting in two registers (V_0 and V_7), which requires $k\ log_2^k$ **min/max** instructions; 2) The next step compares the remaining registers (i.e., $k-2$ registers) as a full

[2]The Intel AVX512 opcodes: **vpmaxsd** (SIMD max), **vpminsd** (SIMD min), and **vpshufd** (SIMD shuffle).



(a) SIMD Min/Max instructions. (b) Bitonic Merge.

Figure 6.3: *Min/Max* and *bitonic merge* examples.

binary tree data structure, where the result of each level is the lowest and highest registers. Then the number of registers to compare is reduced by a factor of 2 at every level until remains two registers to process at the last one. As a result, the number of instructions is $\sum_{i=1}^{2^{(log_2^k-1)}-1} 2i$, where i is the number of levels. This process can be generalized and extended to an arbitrary number of k registers since k is a multiple of two. The general formula to calculate the total number of **min/max** instructions to perform our in-register SIMD Sort is:

$$k\ log_2^k + \sum_{i=1}^{2^{(log_2^k-1)}-1} 2i \quad | \quad i \in \mathbb{Z} \tag{6.1}$$

On the other hand, the related work [128] requires:

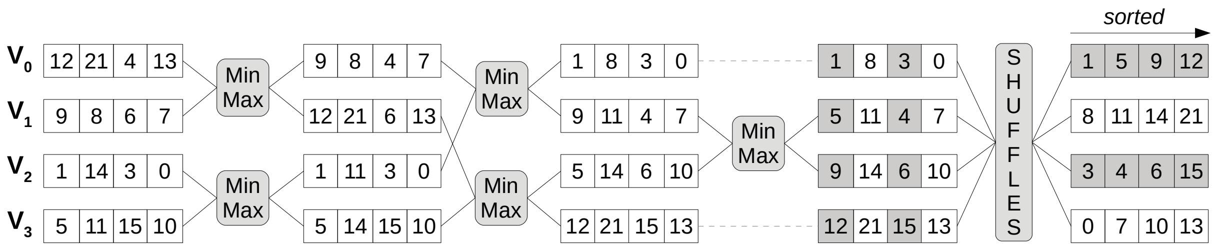
$$2(k-1+(k(log_2^k)(log_2^k-1))/4) \tag{6.2}$$

Table 6.1 presents the number of **min/max** instructions for both approaches on the target architectures. Notice that our SIMD sort algorithm requires less **min/max** instructions on both.

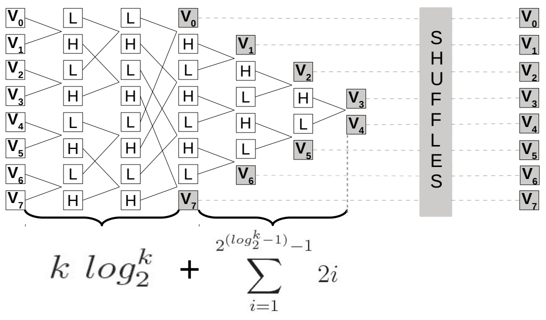
Table 6.1: Number of *min/max* instructions of the in-register SIMD sort computed by Equations 1 and 2.

Architecture	Register Length	Number of min/max inst.	
		In-register SIMD Sort	Related Work [128]
AVX512-64B	16 lanes of 4B	120	126
PIM-256B	64 lanes of 4B	880	1918

After the in-register SIMD Sort, our in-block-register Merge combines sorted registers to produce an overall sorted block of k registers. In Figure 6.4(c), the resulting sorted registers (four registers, $k=4$) of Figure 6.4(a) are the input to the in-block-register Merge. The execution uses an odd-even network to shuffle the registers and applies the *bitonic merge* to compare two individual sorted ones, producing, after all, a sorted block of k registers, i.e., $V_0 \leq V_1 \leq .. \leq V_{n-1}$

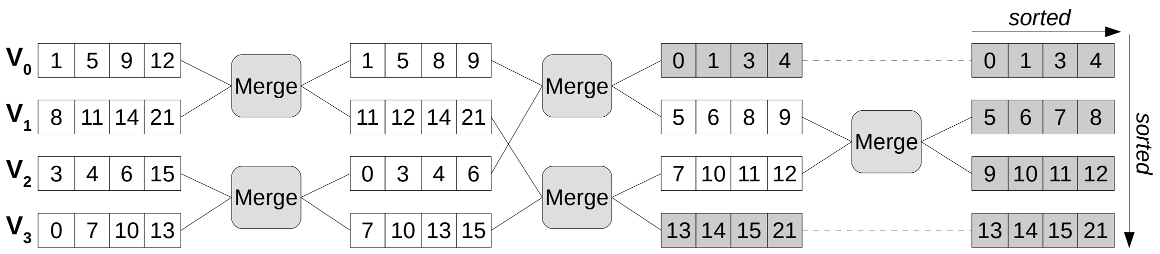


(a) In-register SIMD Sort example.



$$k \; log_2^k \; + \sum_{i=1}^{2^{(log_2^k - 1)} - 1} 2i$$

(b) The number of instructions of the in-register SIMD Sort.



(c) In-block-register Merge for individual sorted registers.

Figure 6.4: SIMD Sort example and number of instructions, and the in-block-register Merge to combine sorted registers.

$\leq V_n$. The execution of the merge phase uses the multiway merge from related work [128, 126] to boost parallelism.

Sort-Merge Join. Now we are ready to discuss the implementation of the Sort-Merge Join algorithm with the SIMD Sort-Merge Operation. The data structure of the join column is of the form "key and object-id" in all of our join implementations. In the particular case of the Sort-Merge-Join, we sort the column by the key using our SIMD sort algorithm, with the addition of one SIMD permutation instruction after the comparison operation to reflect the sort in the object-id, as implemented by related work [129]. With the two relations sorted, we apply the multiway merge to conclude the operation [130, 126].

6.4 SUMMARY

As HIVE (presented in Chapter 4.3) is our baseline PIM architecture that provides support for vectorized in-memory instructions (i.e., SIMD instructions), we have to adapt the database operators to exploit the on-chip processing capabilities. Therefore, we presented the main SIMD instructions applied in the implementation of the operators. The selection and projection require selective load and store SIMD instructions to get and save filtered values from/to the memory. The hash join and aggregation operators generate random memory requests to the hash table that demands special gather and scatter SIMD instructions. The sort-merge join algorithm to better explore the SIMD support requires the instructions: max, min, and shuffle. We also delivered a new sort-merge algorithm that requires less $2\times$ PIM instructions than the state-of-the-art. The next chapter presents results and our analysis of the SIMD operators in PIM and AVX512.

7 RESULTS & ANALYSIS

We present the results of the SIMD query operators in PIM and AVX512. Our goal is to understand the trade-offs between these two highly parallel architectures considering the effect of data movement around memory. We evaluated response time and energy consumption metrics, but we normalized these metrics to resume the data sets in one graphic for each operator. The execution environment is the same described in Chapter 5. We implemented the operators in C++ language and recorded their memory access pattern as input to the Assembly-like memory traces of the simulator. Our query execution design assumes the column-wise storage. Initially, we assume the materialization query execution model (e.g., MonetDB, VoltDB, and Hyrise), but we also discuss the impact of PIM on the pipelined (e.g., PostgreSQL, Hyper, DB2, Oracle) and vectorized (e.g., Vectorwise, Peloton, SQLServer, DB2 BLU) models.

We organized the results according to the taxonomy depicted in Figure 7.1, in which the query operators are grouped based on the memory access pattern, either coalescing or random memory access, and the data reuse, i.e., either high or low/moderate data reuse.

We remember that the footprint of the operators varies according to the workload's data distribution presented in Chapter 5, i.e., among 64 KB (|**L1-64KB**|) up to 4 GB (|**DRAM-4GB**|). Those footprints mean that the input dataset and intermediate data structures fit into the corresponding memory level: L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB up to DRAM-4GB.

The rest of this chapter is organized as follows: We begin, in Section 7.1, with specific related work that classified query operators from their memory access pattern. Section 7.2 shows the results and our analysis of operators with coalescing memory access and low data reuse: the selection and projection operators. In Section 7.3, we discuss the results of operators with coalescing memory access and high data reuse, i.e., the nested loop join algorithm. We evaluate, in Section 7.4, the hash join algorithm and the aggregations that have random memory access with high data reuse. Section 7.5 brings the sort-merge join algorithm that performs random memory access with a moderate data reuse. Those analysis rely on the taxonomy of operators using the materialized execution model. In addition, we perform a comparison between the pipelined and vectorized query execution models in Section 7.6. Section 7.7 presents the effect of selectivity on the pipelined query execution model. We summarize the results in Section 7.8.

7.1 RELATED WORK

Manegold S. et al. [131] performed a pioneer study to analyze the behavior of database operators on memory caches. They designed a cost model based on the settings of the memory hierarchy that estimates the data access pattern of several database algorithms. That study boosted the design of new cache-conscious database algorithms such as radix-decluster projections [132]

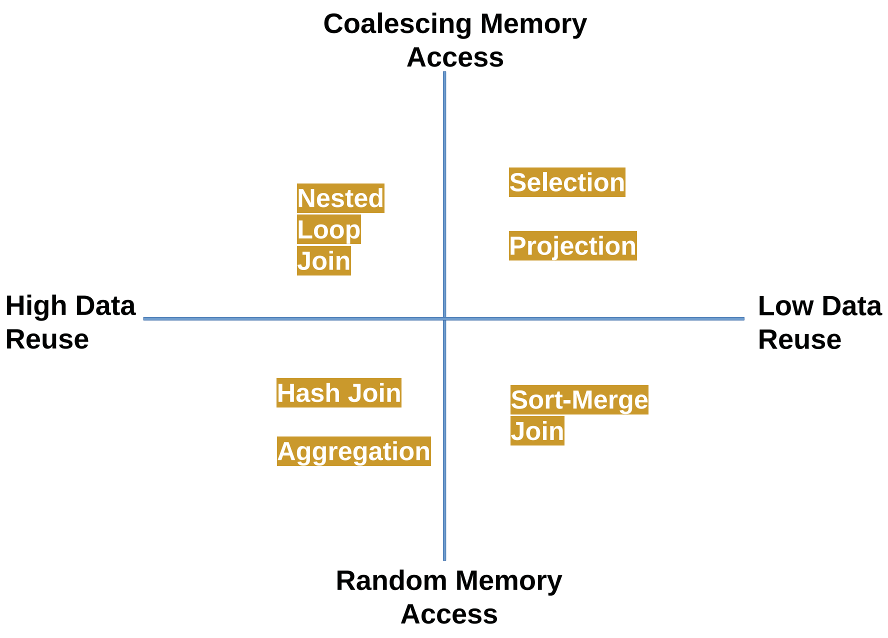


Figure 7.1: Taxonomy of query operators.

and radix-join [103]. Zeuch et al. [133] studied the behavior of selections on modern CPUs, considering some aspects such as parallel execution (levels of parallelism), branch prediction, and cache misses.

Müller S. and Plattner H. [134] performed a deep study about the cost of aggregations on OLAP and OLTP workloads. The main factors involved are data distribution, the size of the input dataset, grouping and sorting attributes, aggregation functions, hash structures, and their implementations. Narayanan R. et al. [135] classified the benchmarks: SPEC INT, SPEC FP, MediaBench, TPC-H, and some data mining applications, according to basic characteristics of memory access, ALU operations, CPI, branch prediction and others.

In the context of flash memory, the uFLIP [136] was designed to categorize I/O operation patterns on distinguished SDD devices, which considers different I/O parameters such as IOSIze, IOShift, TargetSize, parallel access, and others.

Those previous studies analyzed patterns of database operations on modern CPUs, the memory cache hierarchy, and flash disks. They analyzed some characteristics of those hardware. In contrast, our analysis aims to investigate the behavior of database operators on new PIM devices against CPU processing. In our experiments, we studied the data access parallelism of PIM devices, the on-chip processing capabilities, and its energy efficiency. In the CPU side, we analyzed data reuse on the caches and the modern SIMD capabilities provided, i.e., AVX512. We also focus on data movement issues.

7.2 COALESCING MEMORY ACCESS AND LOW DATA REUSE

We classified the selection and projection operators in the same taxonomy of Figure 7.1. Both operators have a coalescing memory access pattern and no data reuse with a streaming data behavior, i.e., after accessing and processing a data portion at the first time, the operators do not access such data portion anymore until the end of the execution.

7.2.1 Selection Operator

Now, we report the results of the selection operator when exploiting the data access parallelism of the PIM-SIMD units. The selection operator applies the predicates of the TPC-H Query 03. We adjusted the size of the columns in our memory traces to fit data into the caches and the DRAM.

In the experiments, we observe that the selection with PIM outperforms the AVX512 execution with at least 4 active vaults. It reaches the best execution when all the 32 vaults are activated in parallel (see Figure 7.2). Therefore, regardless of the size of datasets, the selection operator processes at least $3\times$ faster with PIM than AVX512. Also, with more on-chip processing, PIM uses around 45% less energy than AVX512. This high reduction in energy consumption varies little with a different number of vaults or the size of the datasets (see Figure 7.3).

Such good performance of the selection operator on PIM are due to four folds: 1) It has a sequential memory access pattern that enables the potential of data access parallelism within the PIM device; 2) The SIMD-PIM support allows to execute 64 ($256B/4B$) SIMD instructions per vault with a total of 32 vaults it is possible to execute 2048 arithmetic, logical, or bitwise instructions over 4-byte integers; 3) The operator streams the data that disable the benefits from the CPU caches, i.e., the CPU does not reuse data while processing selections and thus the memory accesses always pay the latency of DRAM; 4) The data movement problem degrades performance because data move until the CPU for processing, which demands a high energy consumption reduced by 50% in PIM.

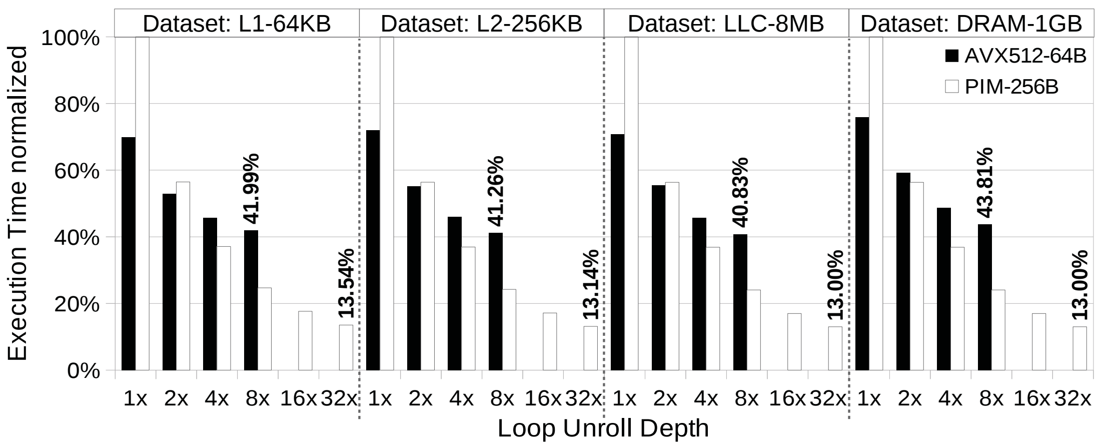


Figure 7.2: Selection: Normalized execution time to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB). The x-axis varies the levels of parallel processing: up to $8\times$ for the AXV512 and up to $32\times$ for PIM.

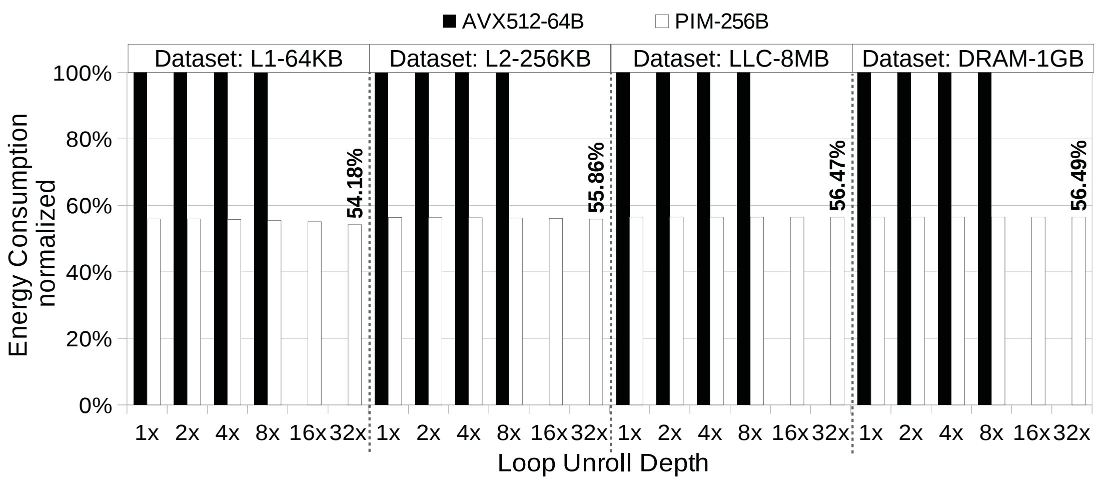


Figure 7.3: Selection: Normalized energy consumption to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

7.2.2 Projection Operator

Now, we discuss the results of the projection operator. As discussed in Section 5.1, the projection operator is responsible for the materialization of intermediate data moving large amounts of data around the memory hierarchy. We observe the same results in Figure 7.4. The execution time of the projection on datasets of the same size as LLC-8MB or less is 7× on average faster in PIM than AVX512. For datasets that do not fit in the caches, e.g., the DRAM-1GB dataset, the execution time is one order (10×) of magnitude faster with PIM. Also, in all datasets, PIM reduces energy consumption in 3x compared to the AVX512. For instance, in the dataset DRAM-1GB, the execution of the AVX512 unrolled 8× spent $1,913$ Joules of energy (see Figure 7.5). On the other hand, the processing in PIM with all vaults, i.e., PIM-256B 32×, generated $0,645$ Joule of energy (an energy reduction of 3x compared to AVX512).

We conclude that pushing the selection and projection operators to PIM has a significant advantage over the x86 processor. Both operators exploit the data access parallelism provided by the 3D-stacked memories. They also can perform 32 parallel SIMD instructions inside the memory device that overcomes the processing power of the x86 due to the latency to move data around the caches until the CPU.

7.3 COALESCING MEMORY ACCESS AND HIGH DATA REUSE

This section presents query operators with coalescing memory access and high data reuse. In this category, we analyzed the Nested Loop Join that is one of the most aplied algorithm for the join operator. The analysis and results on this algorithm shall extend to many other applications that have to read and process datasets into nested loops.

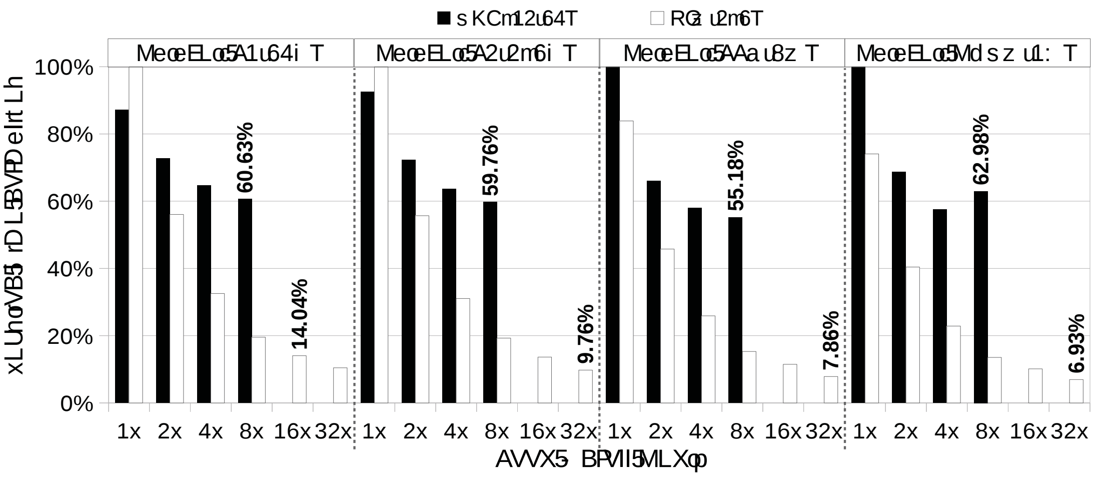


Figure 7.4: Projection: Normalized execution time to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

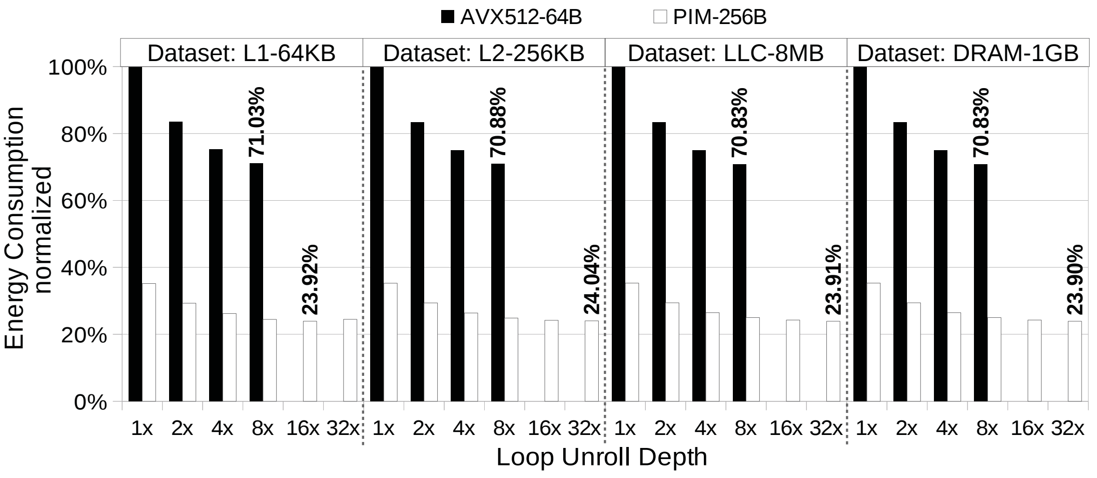


Figure 7.5: Projection: Normalized energy consumption to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

7.3.1 Nested Loop Join

The Nested Loop Join (NLJ) algorithm traverses the join columns with two loops: the outer and the inner. In our implementations, the latter is unrolled up to 32× for PIM and 8× for the AVX512 execution. The goal is to exploit the highest levels of parallel processing and memory access to the devices. Figure 7.6 shows the results for datasets smaller or equal to the L2 cache (L2-256KB). The AVX512 execution unrolled 4× performs better than the PIM execution. The AVX512-style processing re-accesses data in caches for every inner loop iteration, while the inner column fits into the caches resulting in high data reuse (except by the first interaction that causes compulsory memory misses). In contrast, the PIM execution causes compulsory **load** and **store** for every inner loop iteration, because it must access the memory banks at all times.

The PIM execution becomes appealing for datasets bigger than the L2 cache (e.g., LLC-8MB), which inhibit data reuse. The best AVX512 execution, i.e., AVX512 unrolled 8×, spends

3.367 milliseconds to process the LLC-8MB dataset, whereas the PIM unrolled 32× requires 2.428 milliseconds, which represents a reduction of 30% of the execution time. Moreover, the PIM processing saves around 50% of energy consumption in both datasets.

In practice, DBMSs choose the NLJ algorithm only to process small datasets, and for this reason, we suppressed the results for datasets bigger than the LLC cache. Moreover, we analyze the NLJ because it resembles the data access pattern of matrix multiplication that encompasses other applications, such as linear transformation, image processing, and machine learning algorithms. Our analysis on the NJL adds useful insights to that range of applications.

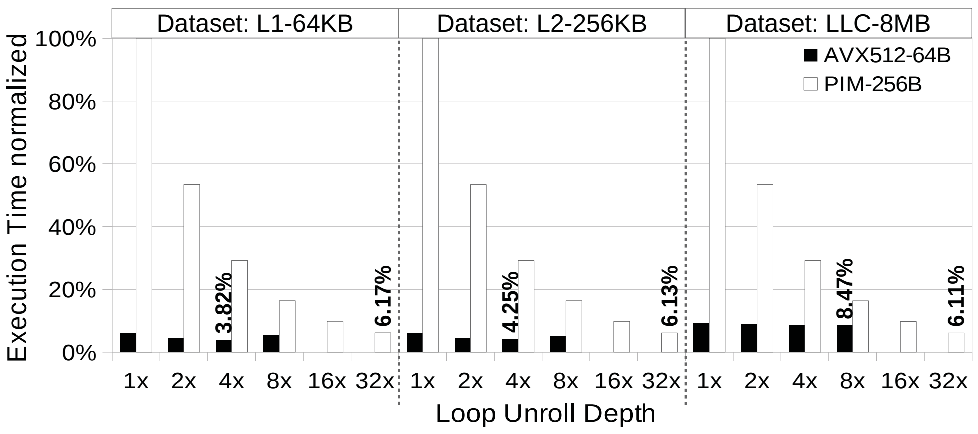


Figure 7.6: NLJ: Normalized execution time to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, and LLC-8MB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

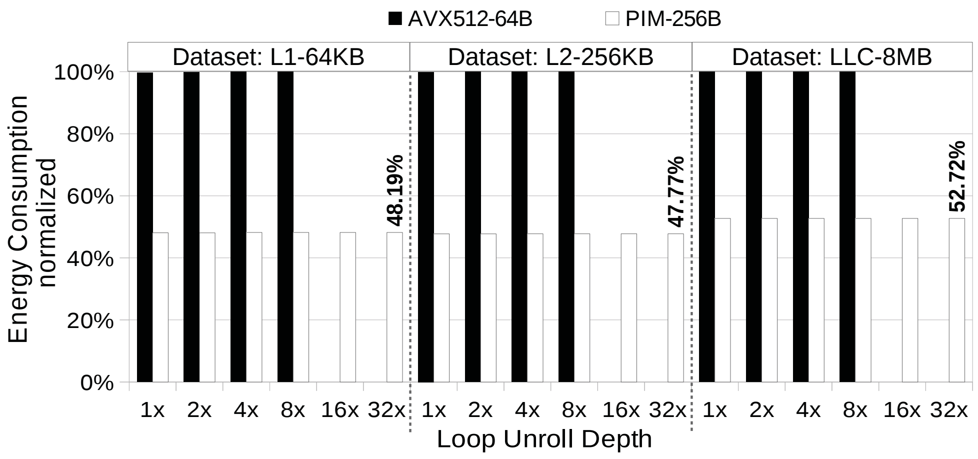


Figure 7.7: NLJ: Normalized energy consumption to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, and LLC-8MB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

7.4 RANDOM MEMORY ACCESS AND HIGH DATA REUSE

In this section, we identified the operators with random memory access and high data reuse, which is the case of the hash join and aggregations. The random memory access pattern of these operators rely on the accesses to hash tables, and the high data reuse depends on how much they re-access the hash table entries.

7.4.1 Hash Join

The hash join algorithm consists of the build and probe phases. These phases have two different memory accesses pattern: sequential memory access to read the join columns and random memory access to access the hash table entries. The build phase generates the hash table from the smallest relation. For instance, the TPC-H Query 03 generates two hash tables for two join operations in the query plan. In the 1GB TPC-H, the hash table on “*c_custkey*” has 30,142 entries with a 173-KB memory footprint. The hash table based on “*o_orderkey*” has 147,126 entries with a 287-KB memory footprint. The probe phase searches the biggest relation to add the join values to the hash table.

Figure 7.8 presents the normalized execution time for the hash join. For all dataset sizes, the AVX512 execution is better than PIM. Two main effects impact the PIM execution: 1) Random access is sparse most of the execution, which means that only one register lane will be useful during PIM **load** operations. 2) Random access shall reuse some cache lines inside the x86 processor, although the reuse ratio may vary depending on the workload and cache size.

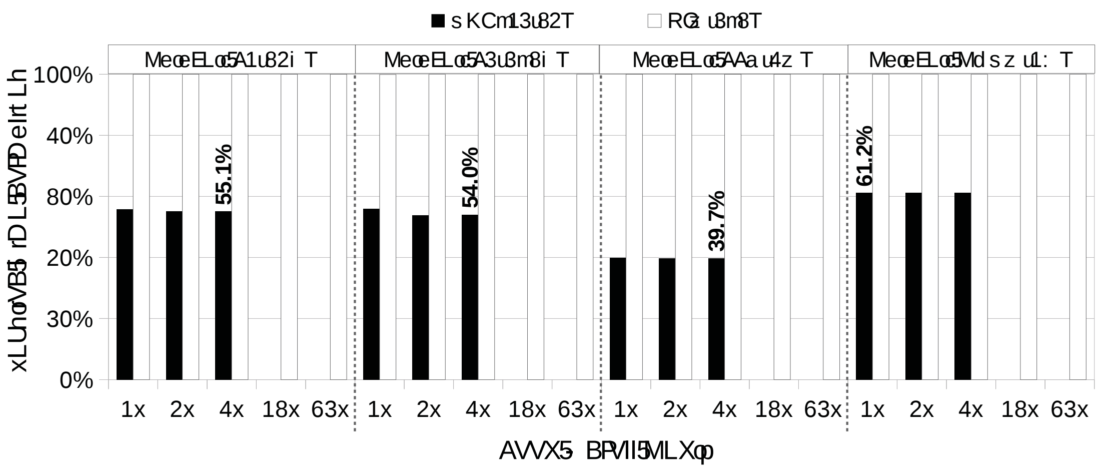


Figure 7.8: Hash Join: Normalized energy consumption to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

7.4.2 Aggregation Operator

The aggregation operator is based on a hash table to hold the aggregation values. It has two memory access patterns: 1) Data streaming while accessing the group columns to compute the hash addresses and the aggregation columns to accumulate the new values; 2) Random memory access while looking up the hash table. In this experiment, the limited number of PIM-SIMD registers restricts the data access parallelism to 16× to build the aggregation and groups.

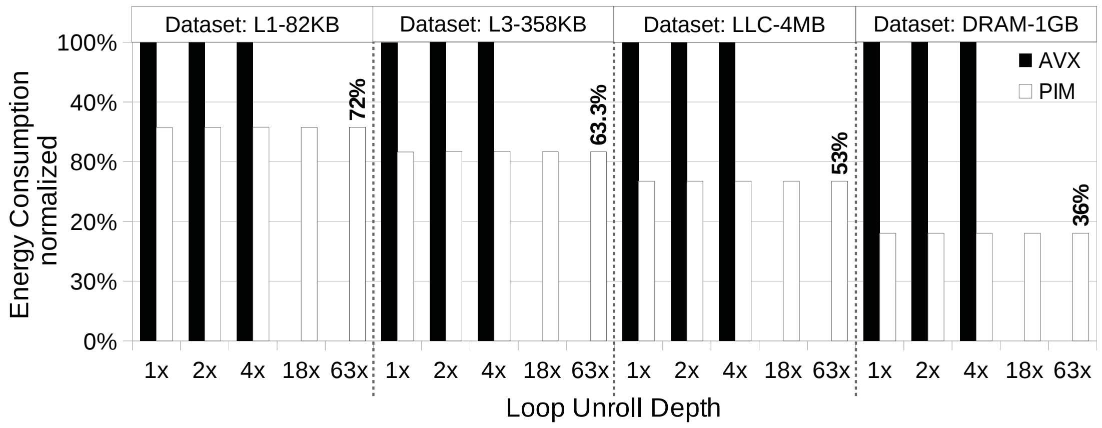


Figure 7.9: Hash Join: Normalized energy consumption to the worst execution on the target architectures according to the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB). The x-axis varies the levels of parallel processing: up to 8× for the AXV512 and up to 32× for PIM.

7.4.2.1 *TPC-H Query 01*

The aggregation operator in the TPC-H Query 01 has two columns for grouping, and eight aggregation functions based on five columns from the **Lineitem** table. With a small number of groups, i.e., hash table entries, the hash table also has a small memory footprint that fits into the L1 cache. Although the operator streams the five columns to compute the aggregation functions, Figure 7.10 shows that the memory access to the hash table dictates the performance regardless of the degree of parallelism used by the PIM device. With an unroll depth of 2×, the **gather** instruction of the AVX512 accesses two cache lines that are sufficient to load the entire hash table to SIMD registers outperforming PIM with all unroll depth versions.

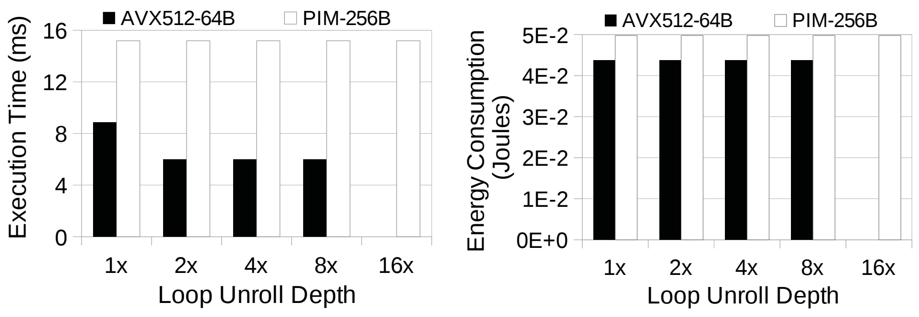


Figure 7.10: Aggregation: the execution time of the TPC-H Query 01, varying the levels of parallel processing.

7.4.2.2 *TPC-H Query 03*

The aggregation operator in the TPC-H Query 03 has three columns of grouping and just one aggregation function based on two columns from the **Lineitem** table. The number of hash table entries is a few hundred, which still fit in the L2 cache. This fact leads the PIM to scale according to the degree of parallelism. The difference in performance decreases between PIM and AVX512 compared to the results of Query 01. However, the execution of the aggregation remains better in AVX512 (see Figure 7.11).

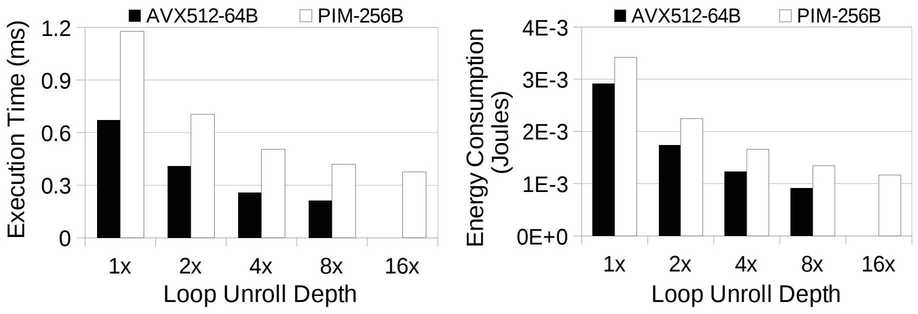


Figure 7.11: Aggregation: the execution time of the TPC-H Query 03, varying the levels of parallel processing.

7.4.2.3 *Zipf Distribution*

In the previous experiments with TPC-H queries 01 and 03, the hash table fit into the L1 and L2 caches. Now, we investigate the aggregation operator with the Zipf workload varying the size of the dataset using bigger sizes than the cache memories. The Zipf distribution was also used by the related work [137] to evaluate the aggregation operator.

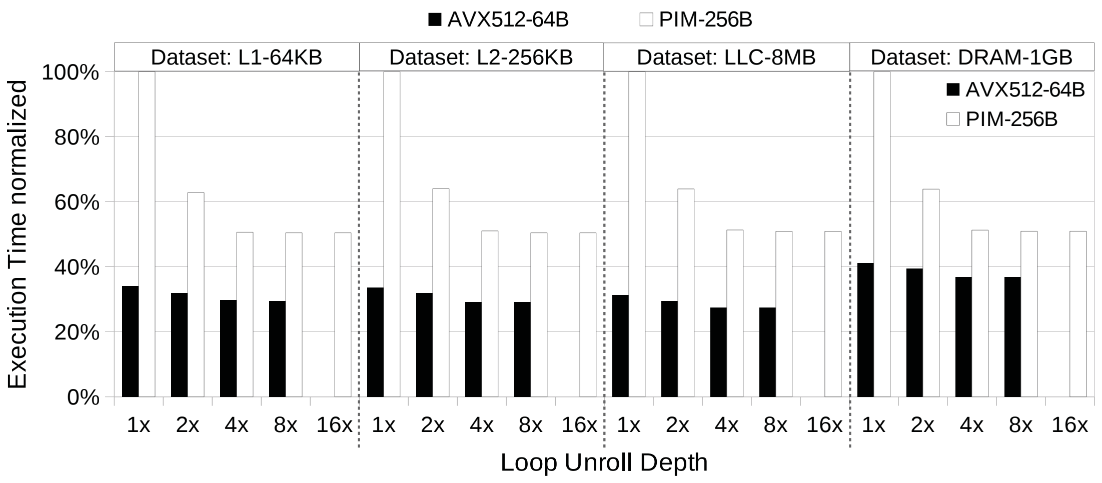
Figure 7.12 shows that the execution time using AVX512 is still better than PIM and that the difference in energy consumption is quite marginal. The AVX512 performance results come from the high reuse of the hash table, especially for small hash tables that fit into the caches. The random access to the hash table restricts the data access parallelism of the PIM device, incurring in the same effects observed in the hash join (see Section 7.4.1), i.e., low usage of SIMD register lanes and x86 cache memory reuse.

7.4.3 Discussion

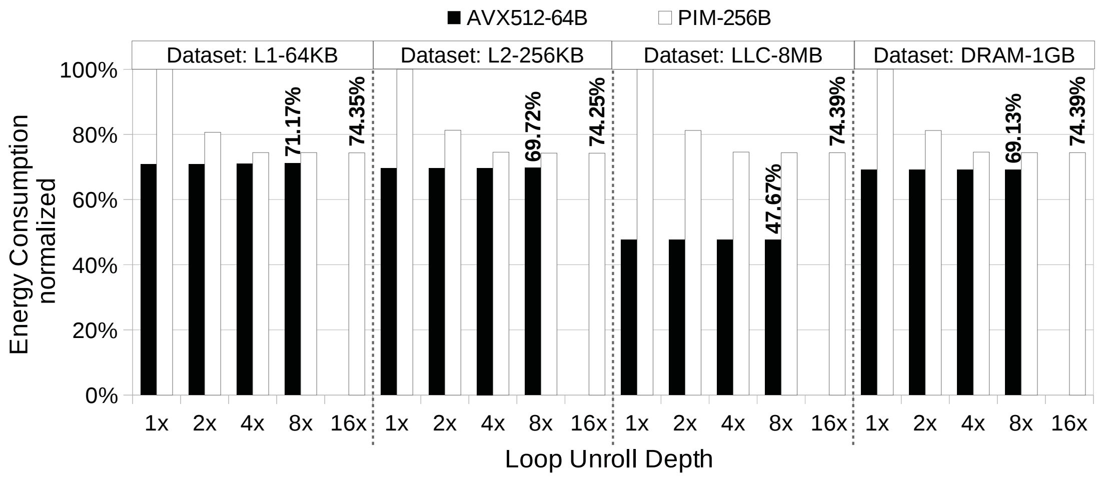
All in all, the hash join algorithm and the aggregation operator are susceptibles to the random memory access to the hash table and to the high reuse of hash entries. Figure 7.13 illustrates the problem of random access pattern inside a 3D-staked memory, it depicts serialization accesses to the vault V_0. We observed, at least, three random requests to memory banks within V_0 that cause memory access serialization.

The results of the hash join reveal the random access pattern problem. For example in Figure 7.8, the PIM unrolled $1\times$ and $32\times$ have the same performance, which means that regardless of the levels of parallelism used, the memory access serialization dictates the performance. However, the PIM execution reduces energy consumption in all datasets. Figure 7.9 shows that the energy savings by PIM increases as the hash table becomes bigger. The reasons behind those results are because AVX512 with bigger datasets generates more data movement than PIM to access the hash table entries from the main memory.

On the other hand, the hash table access pattern dictates the performance of the aggregation operator regardless of the performance metric. The random access shows low data reuse as at most 32 memory addresses from the 64 possible addresses in the SIMD lanes can be accessed at once. In this case, hashing will require two loads to compute the hash keys if the unroll depth



(a) Aggregation with Zipf distribution: execution time normalized to PIM-256B 1x.



(b) Aggregation with Zipf distribution: energy consumption normalized to PIM-256B 1x.

Figure 7.12: Normalized execution time and energy consumption of the aggregation operator with the Zipf distribution. These performance metrics were normalized to the worst execution (i.e., PIM-256B 1×), varying the size of datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, and DRAM-1GB) and levels of parallel processing.

is set to 32×. As a remark, we did not consider, in this study, aggregations without grouping, i.e., no hash table, because it is a corner case in analytic workloads that we keep open for future work.

7.5 RANDOM MEMORY ACCESS AND HIGH DATA REUSE

In this section, we discuss the results of operators with random memory access and a *moderate* data reuse, which is the case of the Sort-Merge Join algorithm.

7.5.1 Sort-Merge Join

The execution of the Sort-Merge Join presents two different memory access patterns from its two phases. The first phase generates random memory access when sorting the join columns, while the second phase generates sequential memory access when merging the sorted

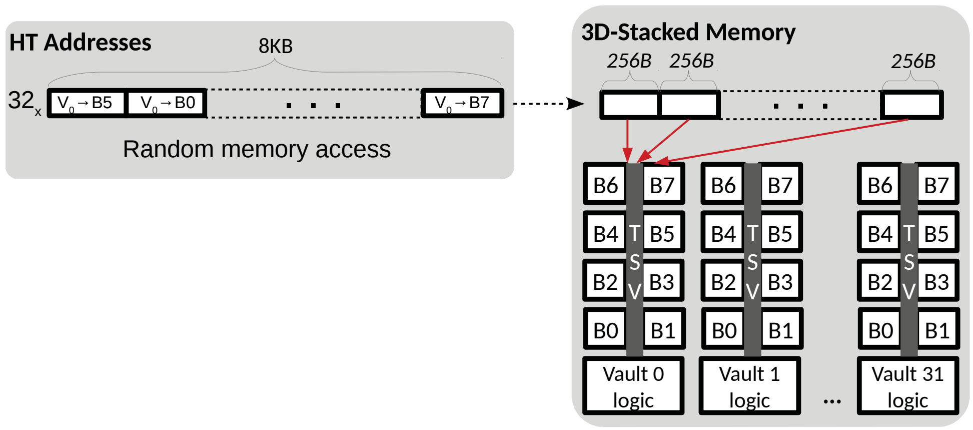


Figure 7.13: Random memory access serialization to the vault V_0 within the PIM device, where $V_0 \rightarrow B_x$ is the access to the memory bank B_x of the vault V_0.

columns. Figure 7.14 presents the execution results using as much parallelism as possible. We use an unroll depth of $8\times$ in both PIM and AVX512, because the SIMD sort-merge algorithm reserves SIMD registers to hold intermediate values, such as lowest/highest values from **min/max** instructions and others from the **shuffle** instructions.

The execution of the AVX512 performs better while the datasets fit into the caches due to faster data access, as observed in both metrics: time and energy. The execution time remains smaller in the AVX512 execution. However, the energy consumption is higher on datasets bigger than the LLC due to the data movement. In those cases, the PIM uses around 40% less energy than AVX512, see Figure 7.15.

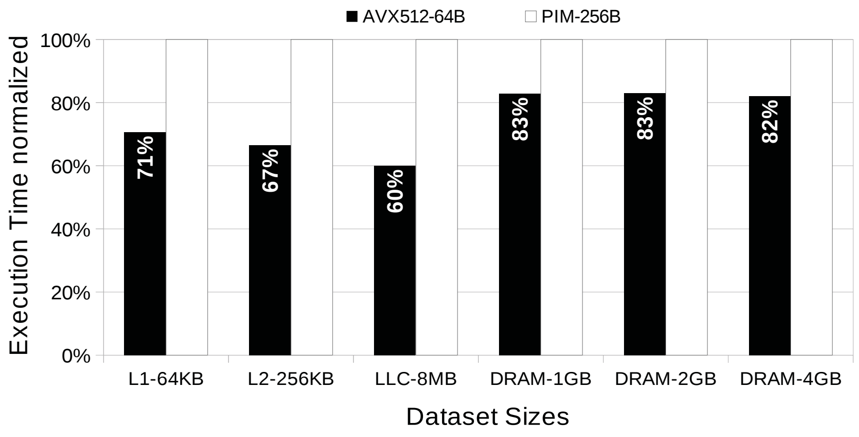


Figure 7.14: Sort-Merge Join: Normalized excecution time on the target architectures with loop unroll depth of 8x, varying the size of the datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, DRAM-1GB, DRAM-2GB, and DRAM-4GB).

7.5.2 Discussion

In brief, the AVX512 overcomes PIM in terms of the execution time in the hash and sort-merge join. The PIM execution saves more energy avoiding off-chip data movement. Another significant join algorithm is the radix-join [103], which could be evaluated to reduce the energy

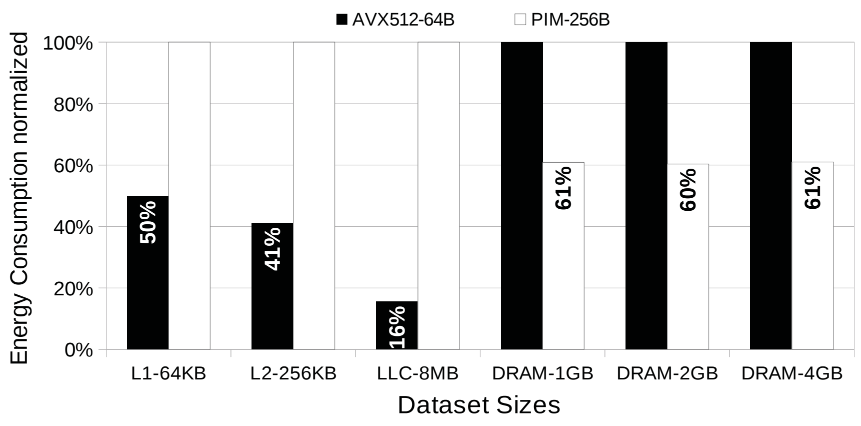


Figure 7.15: Sort-Merge Join: Normalized energy consumption on the target architectures with loop unroll depth of 8x, varying the size of the datasets (i.e., L1-64KB, L2-256KB, LLC-8MB, DRAM-1GB, DRAM-2GB, and DRAM-4GB).

consumption of the AVX512. Roughly, the radix-join has two distinct data access patterns: a random pattern while building radix-clusters for both join relations and a sequential one to probe the clusters with a nested-loop [103]. The random access pattern is also present in the hash join experiments, where our recommendation is to use PIM for energy saving. The sequential memory access is the same pattern evaluated in the NLJ experiments, in which PIM saves around 50% of the energy consumption, even in a dataset fitting in the L1 cache. The main reason for energy waste is the off-chip data movement. In our evaluation, such a factor shall not reduce with radix-join because its memory access patterns are already present in the experiments of the other algorithms. However, radix-join is a compelling case for future work.

All in all, we conclude that the performance of the join operator is very susceptible to the cache settings, the dataset size, and the target performance metric. The AVX512 execution benefits from the caching mechanism when the join columns fit into the caches or during random memory accesses, which enables data reuse inside the caches.

7.6 PIPELINED VS. VECTORIZED QUERY EXECUTION

In this section, we compare the pipelined and vectorized query execution models. We implemented the selection vector and bitmap data structures to support the execution of both models. In the pipelined execution, the selection operator uses those data structures to hold intermediate results in SIMD registers as long as possible, avoiding data re-access. These results are used by the next operators to filter columns along the CPU pipeline. In the vectorized execution, the selection operates on vectors of 1024 elements[1] and stores the intermediate data structures into the memory to be loaded by the next operator in the query plan. Those **store/load** instructions are the main factor that differs between the implementation and performance of these query execution models.

[1]The same quantity defined by related word [8].

We analyze the selection operator that is followed by the build phase of a hash join. We noticed an opportunity to fuse these two operators in the TPC-H Query 03 query plan, the selection filter “c_mktsegment = ’BUILDING’”, and the build of the hash table on *c_custkey* because there is no pipeline breaker [49] between them. Therefore, SIMD registers hold an intermediate selection vector that is used to filter the *c_custkey* column (**gather** instruction). Keeping the selection vector in SIMD registers precludes the exploitation of the maximum 32× data access parallelism of PIM. In our implementation, 16 SIMD registers hold the selection column (*c_mktsegment*), while the selection vector uses the remaining registers.

Figure 7.16 presents the execution time and energy consumption of the pipelined and vectorized execution. Results show that the pipelined execution performs better than the vectorized in both architectures due to the additional **store/load** instructions on the selection vector. PIM reduces the execution time of the pipelined system when the 32 vaults are activated. In the AVX512 hardware, we observed almost 50% of energy saving due to the high selectivity of the selection vector that filters around 80% of the join column, and also the random access pattern to build the hash table.

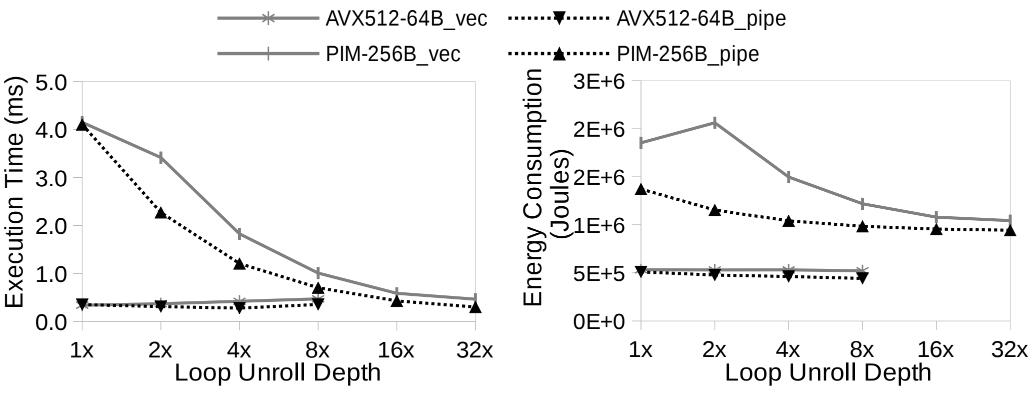


Figure 7.16: Pipelined vs Vectorized execution on TPC-H Query 03 with a selection operator followed by building. The gray lines correspond to the vectorized execution model and black lines to the pipelined one.

Now, we analyze the selection operator followed by aggregation in the query plan of the TPC-H Query 01. The selection predicate filters a small subset (around 1.5%) of the **Lineitem** table, and 98.5% remains to aggregate. The selection operator outputs a bitmap of bytes instead of a selection vector due to the low selectivity of the selection predicate. In this query plan, the pipelined execution with a bitmap as an intermediate structure achieves the maximum degree of parallelism of PIM overcoming the AVX512 processing. The selection operator reads data from all vaults to apply the selection predicate. This strategy compensates for the random memory access of the hash table. In the vectorized execution, SIMD registers hold the bitmap to build the grouping and aggregation columns using the selective load instruction. The aggregation operator applies the conflict-free updates technique [125] to mitigate the concurrence to the hash table. Figure 7.17 shows a marginal improvement to run a selection followed by aggregation on PIM. The vectorized and pipelined executions are worth when at least 4 or 16 vaults active, respectively.

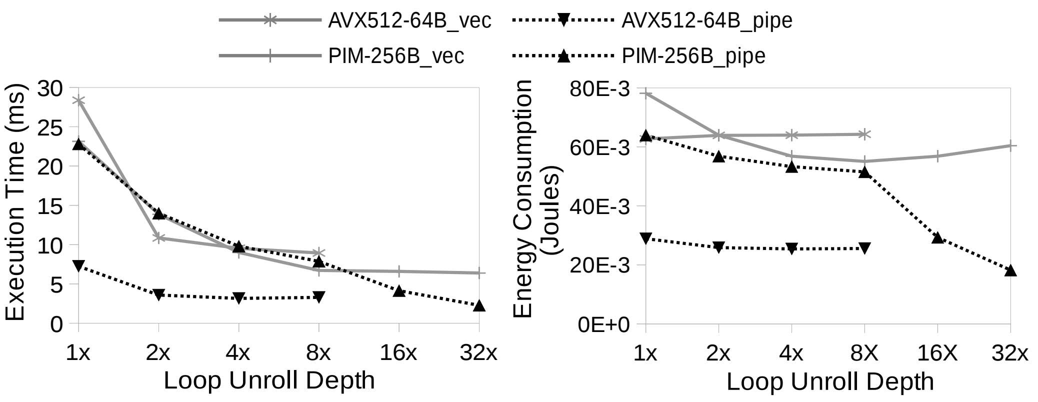


Figure 7.17: Pipelined vs Vectorized execution on TPC-H Query 01 with a selection operator followed by aggregation. The gray lines correspond to the vectorized execution model and black lines to the pipelined one.

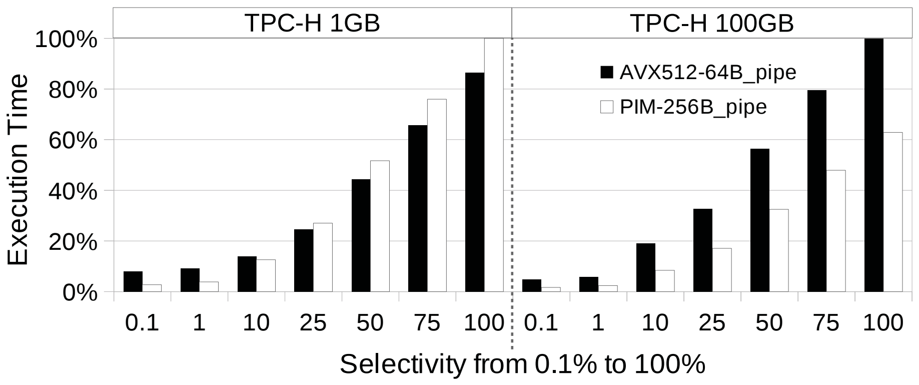
We conclude that random memory patterns hamper the data access parallelism of PIM in both execution systems. This shows opportunities to re-design hash-based algorithms for PIM hardware.

7.7 THE EFFECT OF SELECTIVITY

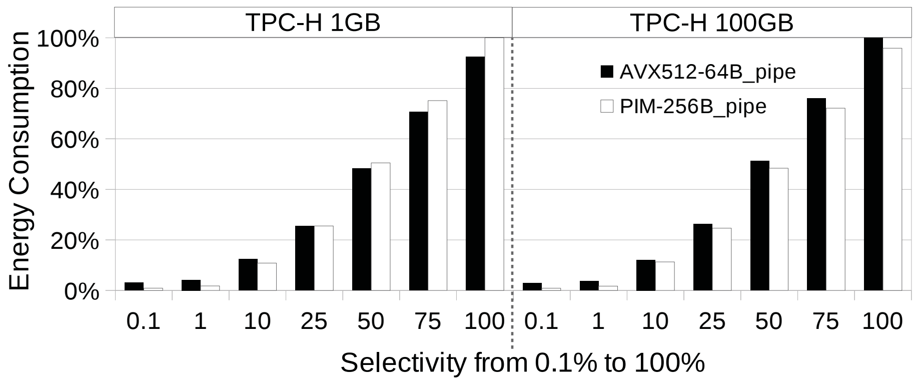
For a more holistic macro-benchmark examination, we evaluate the effect of the selectivity of the TPC-H Query 03 in the pipelined query system (the best performance of AVX512, as observed in Section 7.6). We randomly ranged the selectivity of the *c_mktsegment* between 0.1% and 100%. Varying the selectivity on the pipelined system implies to change the size of the selection vector and the projectivity on column *c_custkey*, and also the cardinality of the join, i.e., the number of entries in the hash table.

Figure 7.18 shows our findings. For selectivities between 0.1% and 10% on small datasets (e.g., TPC-H 1GB), PIM reaches a better performance in both metrics compared to the AVX512 because the selectivity reduces the hash table size alleviating the memory access serialization. For selectivities greater or equal to 25%, the AVX512 outperforms PIM due to two main reasons: 1) The hash table has more entries that imply a higher join cardinality and more memory access serialization; 2) The dataset fits into the caches leading to data reuse. However, for big datasets, e.g., TPC-H 100GB, PIM is faster than the AVX512 regardless of the selectivity because the input columns, the selection vector, and the hash table do not fit into the caches at the same time. Especially in the TPC-H 100GB, PIM execution time ranges from 1.6x to 3x faster than the AVX512 when varying the selectivity from 100% to 0.1%, respectively. Likewise, PIM uses less energy from 5% to 70% compared to AVX512.

Although the selectivity affects query processing, the size of the dataset and intermediate data structures, and the cache settings are the main factors to decide for PIM in the pipelined query execution model.



(a) Varying Selectivity: normalized execution time.



(b) Varying Selectivity: normalized energy consumption.

Figure 7.18: Normalized execution time and energy consumption of the pipelined system according to the worst execution on datasets of 1GB and 100GB, varying the selectivity from 0.1% to 100%.

7.8 SUMMARY

We provided a substantial analysis of the database operators on PIM and AVX512. The results clearly showed that the selection and projection operators take advantage of the PIM capabilities that avoid some pitfalls of the data movement problem.

At the same time, the join algorithms are rather intricate. In the NLJ results, the AVX512 processing demonstrates a better performance while the data fit into the L1 or L2 cache, thanks to the data reuse. However, the performance degrades when the inner table/column is bigger than the L2 cache. In this case, the PIM execution outperforms AVX512 in terms of execution time and energy consumption.

The hash join presents a better execution time for the AVX512 processing due to its random memory access pattern that constrains the data access parallelism of PIM devices. On the other hand, it spends more energy to transfer data along with the memory caches to the CPU. The sort-merge join reached better execution time on the AVX512 processing. In terms of energy consumption, it favors the PIM whether the data do not fit into the caches otherwise AVX512 processing.

Our conclusion about the aggregation operator is that the AVX512 processing still performs better than PIM. The main reason is due to the high data reuse of hash table entries, which leverage the caching mechanism of modern CPUs, i.e., high reuse of HT entries means that most data access is on the latency of caches, which is much lesser than the DRAM latency.

Besides, we provided a comparison of the pipelined and vectorized query execution models with intricate results. Therefore, we further analyzed the impact of selectivity using the pipelined query model on the TPC-H 1GB and 100GB. The results show a clear advantage of PIM on huge datasets, e.g., TPC-H 100GB. Those results open new challenges for a new concept that we define as a hybrid PIM-x86 SIMD query execution system, presented in the next chapter.

8 HYBRID PIM-X86 SIMD QUERY EXECUTION SYSTEM

In this chapter, we discuss the potential of a Hybrid PIM-x86 query execution system. We assume the materialization system again and present our discussion with the execution of a macro-benchmark of TPC-H Query 03. We begin with related work of query scheduling on emerging hardware. Then we introduce our approach for a hybrid query scheduler between PIM and x86, considering the two performance metrics: execution time and energy consumption, Section 8.2. We conclude with challenges and opportunities for the co-design of Database-PIM in Section 8.3.

8.1 RELATED WORK & DISCUSSION

We have classified the related work on query scheduler into two categories, and we discuss the main pros and cons compared with our solution, as follows:

Scheduling On Emerging Hardwares. Current intra-query scheduling focused on co-processing between GPU and CPU to improve execution time based on runtime learning model [138] and operator cost model [139]. The authors of [140, 141] also tested a similar hybrid co-processing in the Intel Xeon Phi co-processor.

Kernel Scheduling on PIM-Assisted GPU. Related work in GPU architectures proposed scheduling techniques with PIM devices installed as GPU main memory. GPU applications are split into independent GPU-kernels and interleave the processing of each kernel between the GPU cores and the PIM device [4, 142, 143]. Although GPUs are devices with high parallel processing degree, data still need to be transferred around the memory hierarchy before moving to the GPU-PIM device.

Discussion. The main difference regarding those approaches to ours is that we focus on in-memory processing with data transfer only when needed. The hybrid scheduling between CPU and GPU tackles compute-intensive applications and neglects the potential of PIM to run data-intensive ones. Even a hybrid co-processing with GPU and PIM needs to move data among the memory hierarchy until the CPU and also generates extra data transfer from the CPU to the shared memory within the graphic card (GPU).

8.2 HYBRID PIM-X86 QUERY SCHEDULER

The results and our analysis in Chapter 7 reveal a surely demand on how to interleave PIM and x86 style-like processing in today's DBMSs. Also, Figure 8.2 shows the execution breakdown in the DRAM-1 GB dataset with the best execution of each operator in the target architectures. For instance, we choose the hash join that showed the best performance among the join algorithms, and it is the most applied join algorithm in today's DBMSs. Processing the

PIM-specific query plan improves the execution time by 12.5% and spends 66% less energy than the AVX512-specific query plan. This result matches the energy efficiency featured by commercial PIM architectures.

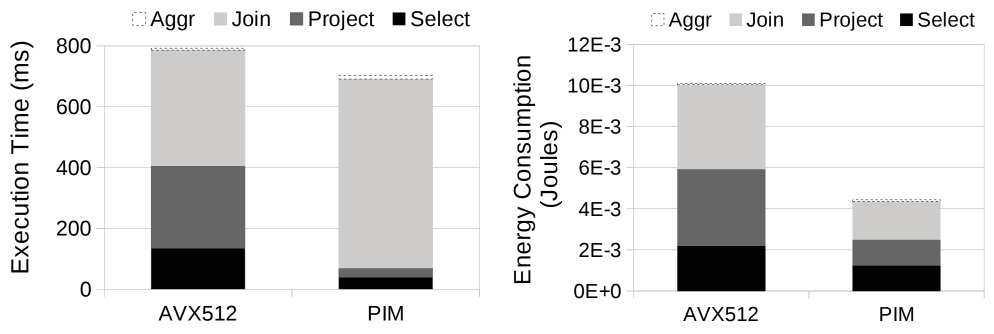


Figure 8.1: TPC-H Query 03 execution breakdown in both target architectures.

In Table 8.1, we correlate the results presented in Chapter 7 with their best processing architectures according to the dataset size and performance metric. We take into account those results to implement the heuristics that coordinates the execution of operators between PIM and x86.

Table 8.1: Database Operators Summary.

Operator		Dataset **Fit in cache?**	Performance **Metrics**	Processing **Architectures**
Selection		no/yes	time/energy	**PIM**
Projection		no/yes	time/energy	**PIM**
Join	Nested Loop	L1/L2	time	**AVX512**
		LLC	time	**PIM**
		yes	energy	**PIM**
	Hash Join	no/yes	time	**AVX512**
		no/yes	energy	**PIM**
	Sort Merge	no/yes	time	**AVX512**
		yes	energy	**AVX512**
		no	energy	**PIM**
Aggregation		no/yes	time/energy	**AVX512**

Figure 8.2 presents the macro-benchmark results of hybrid coordination. The hybrid query plan reduces the execution time by 35% and 45% compared to both hardware-specific PIM and AVX512 plan execution, respectively. For energy consumption, the hybrid query plan consumes less than half of the AVX512 energy but presents a marginal result compared to the PIM plan. All in all, the hybrid query scheduler presents promising results to foster new developments of many-core DBMSs.

Regarding energy results, we observe several trade-offs whenever moving computation from the x86 to the PIM logic: 1) We expect that the functional units (ALU's) and the number of data accesses will consume the equivalent amount of energy. During data streaming, both x86 and PIM execution shall process an equal quantity of computing operations, spending the

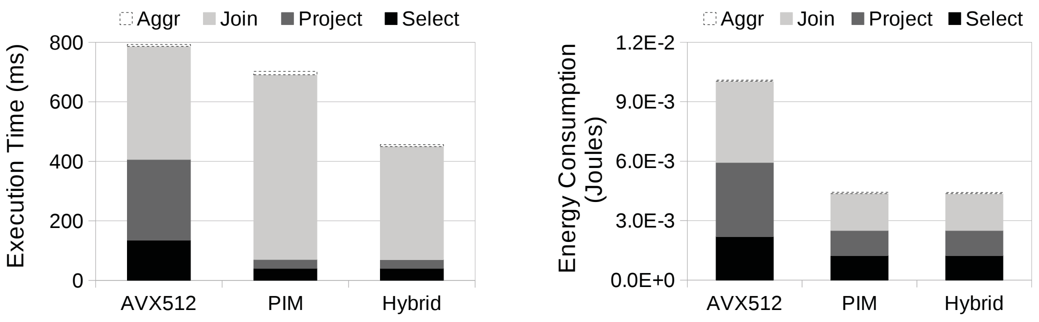


Figure 8.2: Execution breakdown when applying our findings to process a hybrid query plan for Query 3 in the target architectures.

same amount of energy per operation no matter the hardware; 2) We can significantly save energy reducing off-chip data transfers, as they consume 62.7%, on average, of the total system energy budget [1]; 3) We also reduce the energy consumption of the cache subsystem with less data being stored and evicted from the cache memories. The energy consumption of the cache subsystem accounts for 25% to 50% of the full processor energy consumption [144]; 4) We expect that energy consumption increases to send the instructions inside the memory. On-chip processing requires extra hardware to handle the instructions and the messaging of their status to the CPU at the end of each operation. However, the payload of instructions and messages to the CPU is much smaller than a cache line, resulting in a positive trade-off in terms of energy.

8.3 CHALLENGES & OPPORTUNITIES

Hybrid query execution and optimization require a holistic view of a query optimizer to exploit heterogeneous co-processing. Next, we identify a list of challenges for the co-design of Database-PIM.

Simultaneous Co-processing: In heterogeneous processing environments, the optimizer needs to identify opportunities of co-processing to avoid idle devices or inefficient power-consumption (e.g., the power wall problem [117]). Operator-pipelining is also an important technique to be further investigated. Besides, the simultaneous CPU and PIM processing may add hard-to-predict concurrency into the main memory.

Query Plan Optimization: The search space to optimize query plans is already a problem for traditional query optimizer. The addition of hybrid query plans increases the complexity to generate efficient candidate query plans. A PIM-aware query optimizer needs to take into account hardware-specific features to choose the appropriate running device, such as limited processing power and reduced data movement for PIM, and fast processing with caching mechanisms for superscalar CPUs.

Transactions: Intrinsically arithmetic and logical PIM update instructions are atomic [18]. This opens research opportunities for near-data transactions and Hybrid Transactional/Analytical Processing (HTAP). The current PIM ISA supports compare-and-swap instruction to evaluate values. Therefore, PIM update instructions can be synchronized in-memory without wasting cache-check time or extra memory bandwidth. The opportunity is to reduce the overhead of locking and latching, which correspond to 30% of the instructions in OLTP [145].

DBMS Adoption: We envision that Database Management Systems (DBMSs) should invoke PIM instructions at the operator code base, similarly to the SSE and AVX approaches. We also consider code optimization to provide intrinsic functions for PIM ISA.

9 CONCLUSION & FUTURE WORK

In this thesis, we investigated the design and implementation of query execution in modern Processing-In-Memory hardware. The new 3D-stacked memories are promising PIM hardware to tackle the data movement problem. They have been designing to improve memory bandwidth with multiple parallel data access, besides the integration of DRAM dies and logical units through the TSV bus. These emerging memory hardware opens new challenges and opportunities for data-centric systems that require moving large amounts of data around memory, such as DBMSs for OLAP workloads. Therefore, we aim to mitigate the data movement problem on analytic query execution due to the the side effects of the legacy von Neumann computing-centric model.

To the best of our knowledge, a little is known about the potential of PIM for OLAP. We conclude that the choice of the processing hardware paradigm (either PIM-style or x86-style based on the von Neumann model) for query execution depends on the query operators (i.e., the taxonomy of operators: memory access pattern and data reuse), the target performance metric, cache settings, and the size of datasets.

This thesis provides a thorough study of the distinguishable query execution models with PIM support: materialized, vectorized, and pipelined. One of the most efficient modern PIM architecture is HIVE that supports on-chip processing with SIMD registers of 256-bytes wide. For a fair comparison, we evaluated the execution of database operators on HIVE against the widest SIMD architecture of the current x86 processor, i.e., the AVX512. Our goal is to understand how DBMSs can benefit from PIM to alleviate the data movement problem. Thus we gauged the execution time and energy consumption of the most memory demanding query operators.

Materialized Query Execution Model

The materialized query execution model processes just one query operator per call in a column-at-a-time fashion. In such a system, we identified that the selection and projection query operators exploit the maximum of data access parallelism provided by PIM architectures. Hence, they are also able to run in parallel 32 SIMD instructions of 256-bytes, which has a significant advantage over the x86 processor that also must move data around the caches. In Chapter 5, we empirically demonstrated that these operators corresponding to around 70% of the execution time and 50% of energy spent by the 100GB TPC-H benchmark. These operators are around one order of magnitude faster on PIM, with an energy reduction of about 50%, this implies a significant and promising improvement with PIM on DBMSs.

The join operator has different results, because the performance depends on the memory access pattern and the size of the dataset. The Nested Loop Join is susceptible to the cache settings. Therefore, in such an algorithm, the x86 processing has a better execution time, while the input datasets fit into the L1 or L2 caches (due to the data reuse). The PIM execution is faster for datasets equal to or greater than the L3 cache. In all cases, PIM saves around 50% of energy regardless of the size of the dataset.

One valuable contribution of our experimental study appears when analyzing the hash join algorithm. We uncovered the effects of low usage of SIMD register lanes and also the data reuse that appears in the x86 processing. These characteristics inhibit data access parallelism and processing capabilities of PIM, degrading the performance of applications that generates massive random memory access during the execution. For example, the PIM hardware is between 40% to 60% slower for processing the hash join. At the same time, PIM saves energy as bigger is the size of the dataset, reducing by almost 65% the energy spent to process datasets greater than the LLC. Another contribution of this thesis is our SIMD sorting algorithm that requires fewer SIMD instructions than the state-of-the-art in both PIM and AVX512 architectures. We observed that our algorithm presented the best results when executed by AVX512.

We noticed that aggregations on PIM are very intricate, because the hash table access pattern dictates the performance. The aggregation queries that we evaluated have high reuse of hash table entries, which leverage the caching mechanism on x86 processing. As a result, using distinct hash table distributions (e.g., TPC-H Query 01 and 03, and Zipf distribution), the PIM hardware could not improve the execution time and the energy consumption.

Vectorized and Pipelined Query Execution Models

The vectorized query execution model allows compilers to generate efficient loop-pipelined code and avoids the high interpretation overhead caused by the Volcano-style model. Although this system has a better usage of the caching mechanism, it has to materialize intermediate data at a vector granularity. On the other hand, the pipelined query execution model enables *pipeline data*, which keeps data into CPU registers as long as possible.

The experiments showed a slightly better performance of the pipelined system over the vectorized one on both architectures. Varying the selectivity and the size of datasets on the pipelined system, the AVX512 has a better performance with high selectivities on small datasets. However, on big datasets (e.g., TPC-H 100GB), the PIM execution on the pipelined system reduces the execution time from 1.6× to 3× and uses less energy from 5% to 70% compared to the AVX512. These results are due to the cache subsystem settings because the dataset and intermediate data do not fit into the caches at the same time. This is clear evidence of the data movement problem that PIM can mitigate.

Hybrid PIM-X86 SIMD Query Execution System

We have identified a sure demand for building a hybrid query execution system. Our first approach to that system is a hybrid query scheduler that receives a query plan and coordinates the execution of that plan between x86 and PIM. This coordination is based on heuristics from the experiments in Chapter 7 with promising preliminary results. The hybrid query execution reduced the execution time between 35% and 45% compared to hardware-specific query plans and saved 2$\times$ more energy than the AVX512-specific query plan.

We conclude that DBMSs require a clever co-design of Database-PIM to reduce the effects of data movement on analytic query execution. Our findings allow advancing further on that co-design and implementation of SIMD query operators on PIM and x86-style processing. Therefore, the hybrid PIM-X86 scheduler is one of our contributions to that direction.

9.1 FUTURE WORK

Although the hybrid query scheduler achieves promising performance results, at the scheduler level (physical query level) there are not thorough information about the interaction of operators in the query plan. The query optimizer is responsible for choosing the best-estimated execution plan with a holistic view on how query operators iterate and share data among the pipeline of operators, which is crucial for building an optimized hybrid query plan.

Our hybrid query scheduler allows the DBMS to interleave the execution of query operators between x86 and PIM. Our findings in this thesis open new research opportunities for the co-design of Database-PIM. Considering such a many-core environment, one important aspect is to identify *misscheduled* operators, i.e., operator instances running at an incorrect processing hardware that leads to performance loss. Therefore, we envision a dynamic scheduler that monitors the execution of the hybrid query plan, then case the performance of a monitored operator has an unpredictable loss, the scheduler can take a runtime schedule decision or can maintain a learning base with all major information for accurate demand schedule decisions.

We realized an opportunity to design a small cache inside the PIM device. The main goal is to reuse data within the memory device and to provide suitable support for the hash and sort-merge join algorithms, and also aggregations. Such an in-memory cache shall boost PIM against the cases that the x86 processing had a better performance due to the data reuse.

9.2 PUBLISHED PAPERS

The list with published papers during this thesis are present bellow:

1. [1] **Tiago R. Kepe**, Eduardo C. de Almeida, Marco A. Z. Alves. **Database Processing-in-Memory: An Experimental Study**. Proceedings of Very Large Data Bases, PVLDB, 13(3): 334-347, 2019.

2. [2] **Tiago R. Kepe**, Eduardo C. de Almeida, Marco A. Z. Alves, Jorge A. Meira. **Database Processing-in-Memory: A Vision**. Database and Expert Systems Applications, DEXA, 2019. Linz, Austria.

3. **Tiago R. Kepe**, Eduardo C. de Almeida, Marco A. Z. Alves. **Processamento de Banco de Dados em Memória**. Brazilian Symposium on Databases, SBBD, 2019. Fortaleza, Ceará, Brazil.

4. **Tiago Rodrigo Kepe**. **Dynamic Database Operator Scheduling for Processing-in-Memory**. PhD Workshop, PhD@VLDB, 2018. Rio de Janeiro, Brazil.

5. Diego G. Tome, **Tiago R. Kepe**, Marco A. Z. Alves, Eduardo C. de Almeida. **Near-Data Filters: Taking Another Brick from the Memory Wall**. International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018. Rio de Janeiro, Brazil.

6. Aline S. Cordeiro, **Tiago R. Kepe**, Diego Tome, Marco Alves, and Eduardo Almeida. **Intrinsics-HMC: An Automatic Trace Generator for Simulations of Processing-In-Memory Instructions.** WSCAD, 2017.

[1]Represents the one of the most important publications to consolidate our experiments and results.
[2]Represents the one of the most important publications to validate our design for the co-design of Database-PIM.

REFERENCES

[1] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 316–331, Williamsburg, VA, USA, March 2018.

[2] Onur Mutlu. Tutorial on memory systems and memory-centric computing systems: Challenges and opportunities. Tutorial@SAMOS, 2019.

[3] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Sixth Edition*. McGraw-Hill Book Company, 2011.

[4] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB*, 12(5):544–556, 2019.

[5] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231–246, 2000.

[6] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In *23rd International Conference on Data Engineering*, ICDE, pages 466–475, Istanbul, Turkey, April 2007.

[7] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. Monetdb/x100 - A DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Conference on Innovative Data Systems Research*, CIDR, pages 225–237, Asilomar, CA, USA, January 2005.

[9] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[10] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions. *CoRR*, abs/1802.00320, 2018.

[11] David J. DeWitt and Paula B. Hawthorn. A performance evaluation of data base machine architectures (invited paper). In *7th International Conference on Very Large Data Bases*, VLDB, pages 199–214, Cannes, France, September 1981.

[12] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3):42–52, 1998.

[13] Anurag Acharya, Mustafa Uysal, and Joel H. Saltz. Active disks: Programming model, algorithms and evaluation. ASPLOS, 1998.

[14] David A. Patterson, Thomas E. Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos E. Kozyrakis, Randi Thomas, and Katherine A. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[15] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In *The 11th USENIX conference on File and Storage Technologies*, FAST, pages 119–132, San Jose, CA, USA, February 2013.

[16] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *International Conference on Management of Data*, SIGMOD, pages 1221–1230, New York, NY, USA, June 2013.

[17] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active disk meets flash: A case for intelligent ssds. In *International Conference on Supercomputing*, ICS, pages 91–102, Eugene, OR, USA, June 2013.

[18] Jeff Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88, 2012.

[19] Joonyoung Kim and Younsu Kim. HBM: memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium*, HCS, pages 1–24, Cupertino, CA, USA, August 2014.

[20] Anastassia Ailamaki, David J. DeWitt, and Mark D. et al. Hill. Dbmss on a modern processor: Where does time go? In *Conference on Very Large Data Bases*, VLDB, pages 266–277, Edinburgh, Scotland, UK, 1999.

[21] TPC. Tpc-h is a decision support benchmark. `http://www.tpc.org/tpch/`, 2019. Accessed in 25/09/2019.

[22] B.V. MonetDB. Monetdb: The column-store pioneer. `https://www.monetdb.org/`, 2019. Accessed in 17/09/2019.

[23] Diego G. Tome, Tiago Rodrigo Kepe, Marco Antonio Zanata Alves, and Eduardo Cunha de Almeida. Near-data filters: Taking another brick from the memory wall. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, ADMS@VLDB, pages 42–50, Rio de Janeiro, Brazil, August 2018.

[24] Tiago Rodrigo Kepe, Marco Antonio Zanata Alves, and Eduardo Cunha de Almeida. Database processing-in-memory: An experimental study. *PVLDB*, 13(3):334–347, 2019.

[25] Tiago Rodrigo Kepe. Dynamic database operator scheduling for processing-in-memory. In *The VLDB 2018 PhD Workshop co-located with the 44th International Conference on Very Large Databases*, Rio de Janeiro, Brasil, April 2018.

[26] Tiago Rodrigo Kepe, Marco Antonio Zanata Alves, and Eduardo Cunha de Almeida. Processamento de banco de dados em memória. In *Brazilian Symposium on Databases*, SBBD, 2019.

[27] Tiago Rodrigo Kepe, Eduardo C. Almeida, Marco A. Z. Alves, and Jorge Augusto Meira. Database processing-in-memory: A vision. In *Database and Expert Systems Applications - 30th International Conference*, DEXA, pages 418–428, August 2019.

[28] Elmasri and Navathe. *Fundamentals of Database Systems*. Pearson, 2007.

[29] Hugh E. Williams and David Lane. *Web Database Applications with PHP & MySQL*. O'Reilly & Associates, 2002.

[30] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6), June 1970.

[31] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[32] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *International Conference on Management of Data*, SIGMOD, pages 967–980, Vancouver, BC, Canada, June 2008.

[33] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 2 edition, 2000.

[34] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *International Conference on Management of Data, SIGMOD*, pages 268–279, Austin, Texas, May 1985.

[35] Daniel J. Abadi. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008.

[36] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *International Conference on Management of Data*, SIGMOD, pages 102–111, Atlantic City, NJ, USA, May 1990.

[37] Simone Dominico, Eduardo Cunha de Almeida, Jorge Augusto Meira, and Marco Antonio Zanata Alves. An elastic multi-core allocation mechanism for database systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 473–484, 2018.

[38] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[39] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2019.

[40] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *31st International Conference on Very Large Data Bases*, VLDB, pages 553–564, Trondheim, Norway, August 2005.

[41] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[42] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999.

[43] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *21st Symposium on Principles of Database Systems*, pages 109–120, Madison, Wisconsin, USA, 2002.

[44] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[45] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A vectorized analytical DBMS. In *28th International Conference on Data Engineering*, pages 1349–1350, Washington, DC, USA (Arlington, Virginia), April 2012.

[46] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[47] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL server column store indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1177–1184, 2011.

[48] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[49] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[50] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO, pages 75–88, San Jose, CA, USA, March 2004.

[51] Waqar Hasan and Rajeev Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 36–47, 1994.

[52] Luc Bouganim, Olga Kapitskaia, and Patrick Valduriez. Dynamic memory allocation for large query execution. *Networking and Information Systems*, 1(6):629–652, 1998.

[53] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[54] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *International Conference on Management of Data*, SIGMOD, pages 311–326, San Francisco, CA, USA, 2016.

[55] PostgreSQL Org. 3rd october 2019: Postgresql 12 released! `https://www.postgresql.org/about/news/1976/`, 2019. Accessed in 05/10/2019.

[56] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems - Embedded Hardware Design*, 67:28–41, 2019.

[57] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture*, ISCA, pages 24–33, 2009.

[58] Qiaosha Zou, Matthew Poremba, Rui He, Wei Yang, Junfeng Zhao, and Yuan Xie. Heterogeneous architecture design with emerging 3d and non-volatile memory technologies. In *Asia and South Pacific Design Automation Conference*, ASP-DAC, pages 785–790, 2015.

[59] Jim Gray and Prashant J. Shenoy. Rules of thumb in data engineering. In *International Conference on Data Engineering*, pages 3–10, 2000.

[60] Erik Riedel. *Active Disks: Remote Execution for Network-attached Storage*. PhD thesis, Pittsburgh, PA, USA, 1999. AAI9964611.

[61] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[62] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *24rd International Conference on Very Large Data Base*, VLDB, pages 62–73, New York City, New York, USA, August 1998.

[63] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, June 2001.

[64] Gokhan Memik, Alok Choudhary, and Mahmut T. Kandemir. Design and evaluation of smart disk architecture for dss commercial workloads. In *International Conference on Parallel Processing*, ICPP, 2000.

[65] Steve C. Chiu, Wei-keng Liao, and Alok N. Choudhary. Design and evaluation of distributed smart disk architecture for i/o-intensive workloads. In *International Conference on Computational Science*, ICCS, pages 230–241, 2003.

[66] Steve C. Chiu, Wei-keng Liao, and Alok N. Choudhary. Distributed smart disks for i/o-intensive workloads on switched interconnects. *Future Generation Comp. Syst.*, 22(5):643–656, 2006.

[67] Juan Piernas and Jarek Nieplocha. Efficient management of complex striped files in active storage. In *International European Conference on Parallel and Distributed Computing*, Euro-Par, pages 676–685, 2008.

[68] Juan Piernas Cánovas and Jarek Nieplocha. Implementation and evaluation of active storage in modern parallel file systems. *Parallel Computing*, 36(1):26–47, 2010.

[69] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *CM/IEEE Conference on High Performance Networking and Computing*, SC, page 28, 2007.

[70] Vassilis Stoumpos and Alex Delis. Grace-based joins on active storage devices. *Distributed and Parallel Databases*, 20(3):199–224, 2006.

[71] Seung Woo Son, Guangyu Chen, Mahmut T. Kandemir, and Feihui Li. Energy savings through embedded processing on disk system. In *Conference on Asia South Pacific Design Automation*, ASP-DAC, pages 128–133, 2006.

[72] David D. Chambliss, Prashant Pandey, Tarun Thakur, Aki Fleshler, Thomas Clark, James A. Ruddy, Kevin D. Gougherty, Matt Kalos, Lyle Merithew, John G. Thompson, and Harry M. Yudenfriend. An architecture for storage-hosted application extensions. *IBM Journal of Research and Development*, 52(4-5):427–438, 2008.

[73] Masaru Kitsuregawa, Kazuo Goda, and Takashi Hoshino. Storage fusion. In *International Conference on Ubiquitous Information Management and Communication*, ICUIMC, pages 270–277, 2008.

[74] J. Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. `http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt`, 2006.

[75] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. The necessary death of the block device interface. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2013.

[76] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, ADMS@VLDB, pages 36–43, Seattle, WA, USA, September 2011.

[77] Philippe Bonnet and Luc Bouganim. Flash device support for database management. In *CIDR 2011,Conference on Innovative Data Systems Research*, pages 1–8, Asilomar, CA, USA, January 2011.

[78] Philippe Bonnet, Luc Bouganim, Ioannis Koltsidas, and Stratis Viglas. System co-design and data management for flash devices. *PVLDB*, 4(12):1504–1505, 2011.

[79] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 59–72, New York, NY, USA, June 2009.

[80] Goetz Graefe, Stavros Harizopoulos, Harumi A. Kuno, Mehul A. Shah, Dimitris Tsirogiannis, and Janet L. Wiener. Designing database operators for flash-enabled memory hierarchies. *IEEE Data Eng. Bull.*, 33(4):21–27, 2010.

[81] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *International Conference on Very Large Data Bases, VLDB*, pages 169–180, Roma, Italy, September 2001.

[82] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. Active flash: Out-of-core data analytics on flash storage. In *IEEE Symposium on Mass Storage Systems and Technologies*, MSST, pages 1–12, 2012.

[83] Kwanghyun Park, Yang-Suk Kee, Jignesh M. Patel, Jaeyoung Do, Chanik Park, and David J. DeWitt. Query processing on smart ssds. *IEEE Data Eng. Bull.*, 37(2):19–26, 2014.

[84] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. Intelligent SSD: a turbo for big data mining. In *ACM International Conference on Information and Knowledge Management*, CIKM, pages 1573–1576, San Francisco, CA, USA, October 2013.

[85] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 7(11):963–974, 2014.

[86] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. SSD in-storage computing for list intersection. In *12th International Workshop on Data Management on New Hardware*, DaMON, pages 4:1–4:7, San Francisco, CA, USA, June 2016.

[87] Zhenhong Liu, Amin Farmahini-Farahani, and Nuwan Jayasena. Opportunities for processing near non-volatile memory in heterogeneous memory systems. In *Workshop on Hardware/Software Techniques for Minimizing Data Movement*, Min-Move@ASPLOS, 2018.

[88] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.

[89] ASIC2: Architectures Systems Intelligent Computing Integrated Circuits. Memristive memory processing unit (mmpu). `https://asic2.group/research/logic-with-memristors/`, 2019. Accessed in 14/10/2019.

[90] J. Van Olmen, A. Mercha, G. Katti, et al. 3D stacked IC demonstration using a through silicon via first approach. In *Int. Electron Devices Meeting*, 2008.

[91] Amin Farmahini Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. DRAMA: an architecture for accelerated processing near memory. *Computer Architecture Letters*, pages 26–29, 2015.

[92] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Larry T. Pileggi, and Franz Franchetti. Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware. In *IEEE High Performance Extreme Computing Conference*, HPEC, pages 1–6, Waltham, MA, USA, September 2013.

[93] Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry T. Pileggi, and Franz Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *IEEE International 3D Systems Integration Conference*, 3DIC, pages 1–7, San Francisco, CA, USA, October 2013.

[94] Erik Vermij, Christoph Hagleitner, Leandro Fiorin, Rik Jongerius, Jan van Lunteren, and Koen Bertels. An architecture for near-data processing systems. In *ACM International Conference on Computing Frontiers*, CF, pages 357–360, Como, Italy, May 2016.

[95] Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels. Sorting big data on heterogeneous near-data processing systems. In *ACM International Conference on Computing Frontiers*, CF, pages 349–354, Siena, Italy, May 2017.

[96] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Annual International Symposium on Computer Architecture*, ISCA, pages 336–348, Portland, OR, USA, June 2015.

[97] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 190–200, Monterey, CA, USA, March 2014.

[98] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Annual International Symposium on Computer Architecture*, ISCA, pages 105–117, Portland, OR, USA, June 2015.

[99] Marco Antonio Zanata Alves, Matthias Diener, Paulo C. Santos, and Luigi Carro. Large vector extensions inside the HMC. In *DATE*, 2016.

[100] Paulo C. Santos, Geraldo F. Oliveira, Diego G. Tome, Marco Antonio Zanata Alves, Eduardo Cunha de Almeida, and Luigi Carro. Operand size reconfiguration for big data

processing in memory. In *Design, Automation & Test in Europe Conference & Exhibition*, DATE, pages 710–715, Lausanne, Switzerland, March 2017.

[101] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *International Workshop on Data Management on New Hardware*, DaMoN, 2015.

[102] Nooshin Mirzadeh, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. Sort vs. hash join revisited for near-memory execution. In *5th Workshop on Architectures and Systems for Big Data*, ASBD@ISCA, Portland, Oregon, USA, June 2015.

[103] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[104] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 2012.

[105] Yusuf Onur Koçberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T. Lim, and Parthasarathy Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[106] Seth H. Pugsley, Jeffrey Jestes, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro*, 34(4):44–52, 2014.

[107] Jason Lowe-Power, Mark D. Hill, and David A. Wood. When to use 3d die-stacked memory for bandwidth-constrained big data workloads. *CoRR*, abs/1608.07485, 2016.

[108] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios N. Pnevmatikatos. The mondrian data engine. In *44th Annual International Symposium on Computer Architecture*, ISCA, pages 639–651, Toronto, ON, Canada, June 2017.

[109] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios N. Pnevmatikatos. Algorithm/architecture co-design for near-memory processing. *Operating Systems Review*, 52(1):109–122, 2018.

[110] Diego G. Tome, Paulo C. Santos, Luigi Carro, Eduardo Cunha de Almeida, and Marco Antonio Zanata Alves. HIPE: HMC instruction predication extension applied on database processing. In *Design, Automation & Test in Europe Conference & Exhibition*, DATE, pages 261–264, Dresden, Germany, March 2018.

[111] R. C. Minnick, R. A. Short, J. Goldberg, H. S. Stone, and M. W. Green. Cellular arrays for logic and storage. Technical report, Apr 1966.

[112] W. H. Kautz. Cellular logic-in-memory arrays. *IEEE Transaction on Computers*, 18(8):719–727, 1969.

[113] Harold S. Stone. A logic-in-memory computer. *IEEE Trans. Computers*, 19(1):73–78, 1970.

[114] David A. Patterson, Krste Asanovic, Aaron B. Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Christoforos E. Kozyrakis, David R. Martin, Stylianos Perissakis, Randi Thomas, Noah Treuhaft, and Katherine A. Yelick. Intelligent RAM (IRAM): the industrial setting, applications and architectures. In *International Conference on Computer Design: VLSI in Computers & Processors, ICCD '97*, pages 2–7, Austin, Texas, USA, October 1997.

[115] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[116] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

[117] Liang Wang and Kevin Skadron. Implications of the power wall: Dim cores and reconfigurable logic. In *IEEE Micro*, volume 33, pages 40–48, 2013.

[118] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The beckman report on database research. *Commun. ACM*, 59(2):92–99, 2016.

[119] AMD. Radeon™ pro wx 8200 graphics. `https://www.amd.com/en/products/professional-graphics/radeon-pro-wx-8200`, 2019. Accessed in 23/09/2019.

[120] PowerColor. Powercolor red dragon rx vega 56 8gb hbm2 - radeon rx vega. `https://www.powercolor.com/product?id=1521537060#spe`, 2019. Accessed in 23/09/2019.

[121] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In *International Conference on Management of Data*, SIGMOD, pages 1493–1508, Melbourne, Victoria, Australia, May 2015.

[122] Marco Antonio Zanata Alves. Sinuca. `https://bitbucket.org/mazalves/sinuca/src`, 2019. Accessed in 17/09/2019.

[123] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. Sinuca: A validated micro-architecture simulator. In *17th IEEE International Conference on High Performance Computing and Communications*, HPCC, pages 605–610, New York, USA, August 2015.

[124] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[125] Tim Gubner and Peter A. Boncz. Exploring query compilation strategies for jit, vectorization and SIMD. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, ADMS@VLDB, pages 9–17, Munich, Germany, September 2017.

[126] Hiroshi Inoue and Kenjiro Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, 2015.

[127] Intel.Corporation. Intel Intrinsics Guide, 2019. Accessed in 17/09/2019.

[128] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[129] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *16th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 189–198, Brasov, Romania, September 2007.

[130] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[131] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *International Conference on Very Large Data Bases*, VLDB, pages 191–202, Hong Kong, China, August 2002.

[132] Stefan Manegold, Peter A. Boncz, and Niels Nes. Cache-conscious radix-decluster projections. In *Thirtieth International Conference on Very Large Data Bases*, VLDB, pages 684–695, Toronto, Canada, August 2004.

[133] Steffen Zeuch and Johann-Christoph Freytag. Selection on modern cpus. In *VLDB Workshop on In-Memory Data Mangement and Analytics*, IMDM@VLDB, pages 5:1–5:8, Kohala Coast, HI, USA, August 2015.

[134] Stephan Müller and Hasso Plattner. An in-depth analysis of data aggregation cost factors in a columnar in-memory database. In *International Workshop on Data Warehousing and OLAP*, DOLAP, pages 65–72, Maui, HI, USA, November 2012.

[135] Berkin Özisikyilmaz, Ramanathan Narayanan, Joseph Zambreno, Gokhan Memik, and Alok N. Choudhary. An architectural characterization study of data mining and bioinformatics workloads. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 61–70, San Jose, California, USA, October 2006.

[136] Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. uflip: Understanding flash IO patterns. In *Biennial Conference on Innovative Data Systems Research*, CIDR, Asilomar, CA, USA, January 2009.

[137] Orestis Polychroniou and Kenneth A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *International Workshop on Data Management on New Hardware*, DaMoN, page 6, New York, NY, USA, June 2013.

[138] Sebastian Breß, Siba Mohammad, and Eike Schallehn. Self-tuning distribution of db-operations on hybrid CPU/GPU platforms. In *GI-Workshop "Grundlagen von Datenbanken"*, pages 89–94, Lübbenau, Germany, May 2012.

[139] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. Heterogeneity-aware operator placement in column-store DBMS. volume 14, pages 211–221, 2014.

[140] Xuntao Cheng, Bingsheng He, Mian Lu, and Chiew Tong Lau. Many-core needs fine-grained scheduling: {A} case study of query processing on intel xeon phi processors. *J. Parallel Distrib. Comput.*, 120:395–404, 2018.

[141] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. Efficient query processing on many-core architectures: A case study with intel xeon phi processor. In *International Conference on Management of Data*, SIGMOD, pages 2081–2084, San Francisco, CA, USA, June 2016.

[142] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *International Conference on Parallel Architectures and Compilation*, PACT, pages 31–44, Haifa, Israel, September 2016.

[143] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. Transparent offloading and

mapping (TOM): enabling programmer-transparent near-data processing in GPU systems. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture*, ISCA, Seoul, South Korea, June 2016.

[144] Marco Antonio Zanata Alves. *Increasing energy efficiency of processor caches via line usage predictors*. PhD thesis, Federal University of Rio Grande do Sul, Brazil, 2014.

[145] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *International Conference on Management of Data*, SIGMOD, Vancouver, BC, Canada, June 2008.