UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

NAMOM ALVES ALENCAR

SCAN AND JOIN OPERATORS FOR ASYMMETRIC MEDIA

FORTALEZA

2019

NAMOM ALVES ALENCAR

SCAN AND JOIN OPERATORS FOR ASYMMETRIC MEDIA

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciências da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. José Maria Monteiro Filho

Coorientador: Prof. Dr. Ângelo Alencar Brayner

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A354s Alencar, Namom Alves.
Scan and Join Operators for Asymmetric Media / Namom Alves Alencar. – 2019.
108 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2019.
Orientação: Prof. Dr. José Maria Monteiro Filho.
Coorientação: Prof. Dr. Ângelo Alencar Brayner.

1. Memória de estado sólido. 2. Processamento de consulta de banco de dados. 3. Operador de junção paralela. 4. Operador de leitura paralela. I. Título.

CDD 005

NAMOM ALVES ALENCAR

SCAN AND JOIN OPERATORS FOR ASYMMETRIC MEDIA

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciências da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. José Maria Monteiro Filho (Orientador)
Universidade Federal do Ceará - UFC

Prof. Dr. Ângelo Alencar Brayner (Coorientador)
Universidade Federal do Ceará - UFC

Prof. Dr. José de Aguiar Moraes Filho
Universidade de Fortaleza - UNIFOR

This work is dedicated especially to my parents, Rivaldo e Nazide, my wife Amanda and my siblings Júnior, Verlaine e Rivaney. You all represent the most amazing feelings experienced in life.

ACKNOWLEDGEMENTS

I want to thank:

My parents, for their unconditional and sincere love demonstrated throughout my life, unquestionably on all days, at all times.

My wife, for everything.

Júnior, my brother, who on numerous occasions has supported me, you are amazing.

My advisors, Prof. Dr.-Ing. Ângelo Brayner, Prof. Dr. José Maria Monteiro Filho and Prof. Dr.-Ing. José de Aguiar Moraes Filho that graced me with interesting and challenging topics. Thank you all for the dedication and patience. You are a great, amazing and inspiring example of professors!

Finally, to all my friends that supported me in this work, let registered my sincere vote of thanks for all the support given.

“All we have to decide is what to do with the time that is given us.”

(J.R.R. Tolkien)

RESUMO

Memorias de estado sólido (Solid State Drive (SSD)), se tornaram uma realidade para armazenamento de grandes bases de dados. SSDs não possuem partes mecânicas em sua composição. Consequentemente, é dotado de características e capacidades diferentes quando comparados com Discos Rígidos (Hard Disk Drive (HDD)). A indústria da computação está melhorando, cada vez mais, o paralelismo interno dos circuitos integrados com a fabricação em larga escala de processadores com centenas e centenas de núcleos. Uma das características mais importantes dos SSDs é que eles possuem diferentes níveis de paralelismo interno para a execução de operações de leitura e escrita. Estão surgindo computadores com SSD que possuem petabytes de capacidade de armazenamento. No entanto, os sistemas de banco de dados foram projetados com base em duas premissas. Primeiro, computadores usam HDDs para armazenar seus bancos de dados. A segunda premissa é que os sistemas de banco de dados distribuídos podem ser dimensionados para mais de uma única instância de um Sistema Gerenciador de Bancos de Dados (SGBD). Entretanto, a última premissa somente considera um pequeno número de núcleos por CPU e um número limitado de instâncias. Assim, para tirar o máximo proveito dos benefícios fornecidos pela paralelização e pelas altas taxas de operações por segundo (IOPS (Input/Output Operations Per Second)) fornecidas por máquinas de muitos núcleos com dispositivos SSDs, os sistemas de banco de dados devem estar preparados para as futuras arquiteturas de processadores e de armazenamento. Baseado nisto, esta pesquisa defende que, para tirar o máximo de proveito das características dos SSDs, componentes do SGBD devem ser cientes da assimetria entre leitura/escrita. A junção é o operador de consulta que requer a maior quantidade de acessos (operações de leitura/escrita) à memória secundária. Esta dissertação apresenta um novo algorítimo de leitura e de junção, chamados respectivamente de DaC Scan e DaC Join. O objetivo principal destes algoritmos é explorar ao máximo o paralelismo interno dos dispositivos SSDs, DaC Join, também, é capaz de reduzir a quantidade de operações de escrita durante sua execução de uma operação de junção entre $R \bowtie S$. Ao realizarmos menos escritas em memória secundária, estendemos a vida útil do dispositivo e utilizamos menos espaço de memória principal. Os experimentos foram realizados em banco de dados com o benchmark TPCH e os operadores propostos foram analisados em duas perspectivas, eficácia e eficiência. Os resultados obtidos mostraram que os algoritmos propostos são bastante eficientes. DaC Join conseguiu reduzir em cerca de 77% o número de operações de escrita w.r.t. quando comparado com os números

apresentados pelo Flash join (TSIROGIANNIS *et al.*, 2009; GRAEFE; HARIZOPOULOS, 2010) e, consequentemente, mostrou-se ser cerca de 61% mais rápido.

Palavras-chave: Memória de estado sólido. Processamento de consulta de banco de dados. Operador de junção paralela. Operador de leitura paralela.

ABSTRACT

Solid State Drive (SSD) has become an attractive alternative for storing large databases. SSDs do not present mechanical parts in their assembly. Consequently, SSD has different characteristics and capabilities than that of Hard Disk Drive (HDD). The computer industry is moving towards the construction in large scale of chips with hundreds of cores in order to increase on-chip parallelism. One of the most important features of SSDs is the fact that they implement different levels of internal parallelism for executing read/write operations. Computers with SSD that provides petabytes of storage area is emerging. Nonetheless, database systems were designed based upon two premises. The first one is the usage of HDD for storing databases. The second premise is that distributed database systems could scale beyond what a single-node Database Management System (DBMS) can support. However, the latter premise only holds for a small number of CPU cores in a node and for a limited number of nodes. Thus, to fully exploit benefits provided by the parallelism and high Input/Output Operations Per Second (IOPS) rates supported by many-core machines with SSDs, database systems should be aware of upcoming CPU architectures and storage technologies. Thus, this research claims that to take full profit from SSD characteristics, DBMS's components should be aware of read/write asymmetry in SSD devices. It is well-known that the join operation is the query operator which requires the highest amount of accesses (read/write operations) to the secondary memory. This dissertation presents new scan algorithm and a new join algorithm, called respectively Divide and Conquer Scan (DaC Scan) and Divide and Conquer Join (DaC Join). The key goal of these algorithms are take advantage of the SSD's internal parallelism devices, DaC Join also reduces the amount of write operations during the execution of any join operation $R \bowtie S$. By making less writes, we intend to extend the lifetime of SSD media by requiring less main memory space. Furthermore, the proposed operators are evaluated by, effectiveness and efficiency, measured experiments on a database with the TPCH benchmark. The achieved results have shown that the proposed algorithms are quite efficient. For instance, DaC Join can reduce up to 77% of the amount of write operations w.r.t. and the number of write operations presented by Flash join (TSIROGIANNIS *et al.*, 2009; GRAEFE; HARIZOPOULOS, 2010), and, consequently, it can be up to 61% faster than Flash join.

Keywords: Solid state memory. Database query processing. Parallel join operator. Parallel scan operator.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

SSD	Solid State Drive
HDD	Hard Disk Drive
DBMS	Database Management System
IOPS	Input/Output Operations Per Second
DaC Scan	Divide and Conquer Scan
DaC Join	Divide and Conquer Join
Bt-Join	B three join
RPM	Rotations per minute
FTL	Flash Translation Layer
LEP	Logical Execution Plan
SQL	Structured Query Language
PEP	Physical Execution Plan
EMS	Early Materialization Strategy
LMS	Late Materialization Strategy
MMS	Mixed Materialization Strategy
JS	Join selectivity factor
RARE join	Random Read Efficient Join
PAX	Partition Attributes across
RID	row-id

SUMMARY

1 INTRODUCTION AND MOTIVATION

1.1 Introduction

From a computer hardware perspective, we are witnessing nowadays the existence of two movements towards, perhaps, a singularity point in computer science. First, the computer industry is moving towards the construction in large scale of chips with hundreds of cores in order to increase on-chip parallelism. In a near future we may have several-chip machines, each of which with hundreds of cores. In parallel to the development of several-core chips, a new type of non-volatile memory is emerging, called solid state memory or solid state drive (SSD). Examples of solid state memories are Flash Memory, Phase Change Memory, Memristors and Non-Volatile RAM (NV-RAM), among others. The most evident characteristic of SSD is the nonexistence of mechanical part.

SSDs present distinct characteristics and capabilities from HDDs. IOPS rates supported by SSDs may be over 10^2 times greater than 15K RPM HDDs. Write operations on SSDs are much more expensive w.r.t. execution time and energy consumption than read operations, a phenomenon called read/write asymmetry. A read operation may be up to 3 times faster and consume up to 8 times less energy than a write operation (PARK *et al.*, 2011). The number of physical write operations on SSD may be far larger than the logical operations, since SSDs internally run two processes (wear leveling and garbage collection), which may induce a rise in the amount of physical write operations (CHEN; ZHANG, 2009). SSD lifetime is determined by the number of write operations on it. Finally, SSDs present low levels of energy consumption.

Nonetheless, database systems were designed presupposing the usage of HDDs for storing data. Consequently, over the years several database systems components (e.g., query engine and buffer manager) have been improved based on HDD's characteristics and performance. For instance, existing database systems have been implemented considering that the cost to execute a read operation is similar to the cost to execute a write operation. Therefore, it is true that only replacing HDDs by faster SSDs may not fully exploit SSDs' capabilities, although it may yield performance improvements. Moreover, read/write asymmetry poses challenges to database technology. Write-intensive components of database systems (e.g., query engine and logging components) may negatively impact SSD's write bandwidth.

Regarding the query engine component, for the most database systems, a join operation requires a huge amount of accesses to secondary memory, i.e, there are several

read/write operations. In order to demonstrate that assertion, let's consider that P_R and P_S represent the size of tables R and S (in pages). To compute $R \bowtie S$, the hash join operator, for instance, requires $2(P_R + P_S)$ accesses, to the secondary memory, to build R and S partitions, which will be used during the probe phase (GARCIA-MOLINA *et al.*, 2008). From the $2(P_R+P_S)$ disk accesses, the number of write operations to store the partitions on disk is $P_R + P_S$.

Accordingly, several SSD-aware join algorithms have been recently proposed to take advantage of SSDs features (CHEN *et al.*, 2011; FAN; MENG., 2014; LI *et al.*, 2009; TSIROGIANNIS *et al.*, 2009; EVANGELISTA *et al.*, 2015). Most of them are based on the notion of late materialization. Thus, they inject into the intra-operator data flow the triple $<JoinAttribute, RID_r, RID_s>$. Thus, they yield smaller join partial results. This way, it is possible to decrease the number of write operations. Nevertheless, since partial join results are incomplete, additional random read operations are necessary to fetch attribute values to correctly produce final join results. Those algorithms suffer from critical drawbacks. Some of them have been designed to run on column-oriented databases (TSIROGIANNIS *et al.*, 2009; GRAEFE; HARIZOPOULOS, 2010). They do not hold the same performance rates, whenever executed on row-oriented database. Others have been proposed to run on devices with specific physical characteristics presented by some SSD devices. For instance, ParaHashJoin runs on SSD device implementing RAID-0 data storage (FAN; MENG., 2014).

Database systems have been designed based upon on the premise that distributed database systems could scale beyond what a single-node DBMS can support. However, that premise only holds for a few CPU cores in a node. Yu et al. present in (YU *et al.*, 2014) evidences that many-core chip machines require a completely redesigned database system, which should be modern-hardware aware. There are some join operators which take profit of running modern-hardware machines (MAKRESHANSKI *et al.*, 2016; BALKESEN *et al.*, 2015). Nonetheless, those join operators have been designed to run on very large portion of main memory available in such machines.

1.2 Problem Definition

It is known that SSDs may provide IOPS rates up to two orders of magnitude greater than the rates delivered by HDDs (PARK *et al.*, 2011). For that reason, SSD technology has emerged as a feasible substitute for traditionalHDD for storing large databases.

An important feature of the SSD device is the internal parallelism for reading and

writing data. Generally, SSD IO parallelism is implemented in different four levels, each of them is specific for an SSD structure: channel-level, package-level, die-level and plane-level. Combining such SSD parallelism levels makes possible to access multiple blocks simultaneously across separate chips, as a unit called clustered block. Therefore, internal parallelism provided by SSD manufacturing parts and assembly opens opportunities to improve data access rates in SSDs.

Accordingly, we define our research problem as to design SSD-aware scan and join algorithms, according to the following guidelines:

1. Novel scan algorithms should take advantage of the SSD's internal parallelism;
2. Novel join algorithms should take into consideration read/write asymmetry of SSD media;
3. Improve the query performance in databases stored on SSD media;

1.3 Hypothesis

Based on information in Sections 1.1 and 1.2, we may state the following assumptions.

Hypothesis 01 The exploitation of internal parallelism on SSD devices may reach better performance for scan operators.

Hypothesis 02 Join algorithms designed to be SSD-aware can reach better performance when they take advantage of the whole environment characteristics.

Throughout this research, we will examine each of the aforementioned hypotheses and pursue scientific evidences that may lead to conclusive facts concerning the assumptions raised.

1.4 Objectives

The general objective of this work is to propose a new scan algorithm and a new join algorithm that take advantage of the peculiar characteristics of the SSD devices. Furthermore, the proposed operators are evaluated by, effectiveness and efficiency, measured on experiments on a database with the TPCH benchmark.

1.5 Contributions

Among the computer science technical contributions, we may highlight the following ones:

- To design and implement new scan and join algorithms regarding SSD characteristics and capable to exploit its internal parallelism.
  - Parallel execution on several cores.
  - Use of dynamic hash functions.
  - Reduction of the amount of write operations on secondary memory.

1.6 Document Structure

This study is structured as follow. Chapter 2 discuss the main technological SSD characteristics and compare it to HDD devices. Chapter 3 describes query execution plans, parallelism on DBMS and materialization strategies. In Chapter 4 non SSD-aware join algorithms are discussed. Chapter 5 related works are studied. Chapter 6 depicts the scan algorithm proposed in this research. In Chapter 7 depicts the join algorithm proposed, then chapter 8 brings an extensive evaluation of the proposed algorithms, regarding distinct workloads and scenarios. Finally, Chapter 9 concludes this work and points out some possible future works.

2 SECONDARY STORAGE MEDIA

2.1 Introduction

This chapter describes the most important features of the Hard Disk Drive (HDD) and Solid State Drive (SSD), highlighting the difference between them. Furthermore, we have a comparative analysis of their main characteristics, which reveals significant research opportunities.

Section 2.2 describes key features presented by HDD, regarding physical media characteristics. Next, section 2.3 describes key features presented by SSD and details related to the read/write operations. Section 2.4 we present a comparative analysis between HDD and SSD. Finally, section 2.5 summarizes this chapter.

2.2 Hard Disk Drive (HDD)

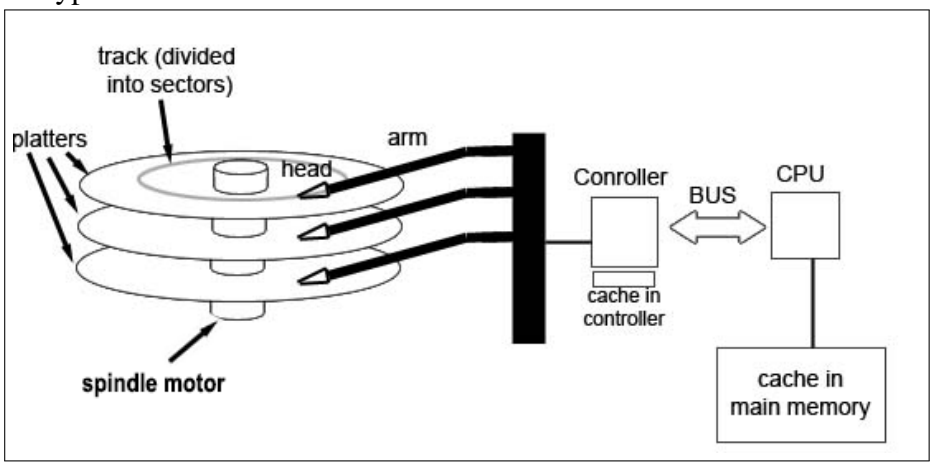
HDD is a non-volatile memory, i.e., data stored in these devices are kept even when they are not energized. These types of devices have been considered the main way for storing persistently mass of data. This is because of its low production cost and the large storage capacity.(GARCIA-MOLINA *et al.*, 2008)

In Figure 1, we may observe the main components of an HDD. An unit of a hard disk is composed by several parts, one of them are the platters which are divided into concentric tracks, data are stored on these tracks. The arms extend to each platter and the tip of an arm is a head which actually reads data. The controller handles interaction between the actual hard disk and the CPU, it is important to mention that there is a cache in the controller. The Spindle motor is responsible for rotating the platters (also called faces or simply disks) at a constant speed, for example, 15,000 Rotations per minute (RPM).

To access data on the surface of the magnetic disks the tips of the arm are used. These arms can move transversely across the tracks. It is important to note that these tips stay very close to the surface, but they never touch it. If it happens the tip will break, consecutively the read and write operation do not work correctly.(GARCIA-MOLINA *et al.*, 2008)

In Figure 2 one can observe the surface anatomy of the magnetic disk. Each surface is covered by a thin layer of magnetic material. Data are stored on this layer as a direction of the magnetic field. Thus, depending on the direction of the magnetic field we have a bit representing

Figure 1 – A typical disk

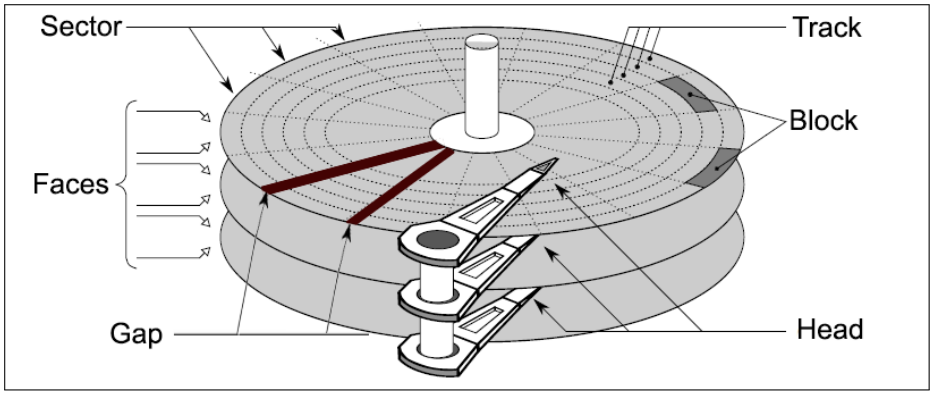


Source: The author

"0" or "1".

As we have said before, each surface is organized in concentric tracks. The tracks are divided into sectors it is in these sectors that the information are stored. It is important to say that each sector has the same storage capacity. The Gaps are the border among sectors of a track. They are small areas non-magnetized, and they represent about 10% of the total size of a track.(GARCIA-MOLINA *et al.*, 2008)

Figure 2 – Disk surface



Source: The author

Access time (AT) is the interval between the time that a request command is sent to a given drive and the moment that the required information returns back to the processor. This time is calculated as follow:

$$AT = ST + LT + TT \tag{2.1}$$

ST (seek time) is the time spent to position the read-write head on the right track. This is the factor that most influence on an HDD access time. There are four variables to calculate

the ST (GARCIA-MOLINA *et al.*, 2008). First, initial acceleration time, this variable represents the time used for the mechanical arm to accelerate until it reaches the maximum fixed speed or until it arrives at the half of the distance that it needs to position itself in the proper HDD track. After that, the arm starts to slow down, once the acceleration can reach about 550 gravitational force equivalent. The second variable is the travel time through the track, the variable represents the time that the mechanical arm takes to travel over the tracks. Usually, this time is less than 1 millisecond. Third, deceleration time is the time considered to stabilize the arm close to the desired track. Finally, the fourth variable, adjustment time which represents the time that the disk controller needs to adjust the read/write head in the correct track.

LT (Latency rotational time) is the average time required for the desired sector pass under the read/write head, set as the half time of one complete rotation. The equation 2.2 shows how to calculate the LT. It is important to note that "r" is the rate of revolutions/second.(GARCIA-MOLINA *et al.*, 2008)

$$LT = (2r)^{-1} \tag{2.2}$$

TT (transfer time) is the time in seconds required to transfer data from a sector defined by Equation 2.3, where n is the size of a sector and rN is the rate of the data transfer, r is the rotation speed and N is the number of the tracks in words.(GARCIA-MOLINA *et al.*, 2008)

$$TT = n(rN)^{-1} \tag{2.3}$$

The performance of an HDD is basically determined by the access time described above. Being composed of mechanical components, there is not so much of what the manufacturers can do to improve considerably the performance in these devices. Thus, due to mechanical limitations of the HDD, the access time tends to be high.

2.3 Solid State Drive (SSD)

SSD has become an attractive alternative for storing large databases. SSDs do not present mechanical parts in their assembly. Consequently, SSDs have different characteristics and capabilities than magnetic disks (HDD).

There are several types of physical media sharing the main features of an SSD, such as: Flash Memory, PCM (Phase Change Memory), Memoristor and NV-RAM (Non-Volatile

RAM), among others. The main distinction among them is in manufacturing features and hardware technology used. For instance, flash memory stores data in floating-gate transistors, denoted cells and PCM stores in a chalcogenide glass.

SSD and NV-RAM are the main solid states devices, which have reached commercialization stage. An NV-RAM device is in fact composed of three different modules: DRAM (Dynamic-RAM), flash memory and an UPS (Uninterrupted Power Supply) (LI V. J. S.; WEIKUAN., 2012). Thus, in NV-RAM device, the DRAM module is used as a cache memory and the flash memory is responsible for data persistence. The UPS component ensures the necessary power to retain data in DRAM module, while those data are flushed from DRAM to flash memory. Therefore, the UPS module behaves as batteries and for that reason it requires frequent monitoring and replacement (NARAYANAN; HODSON, 2012).

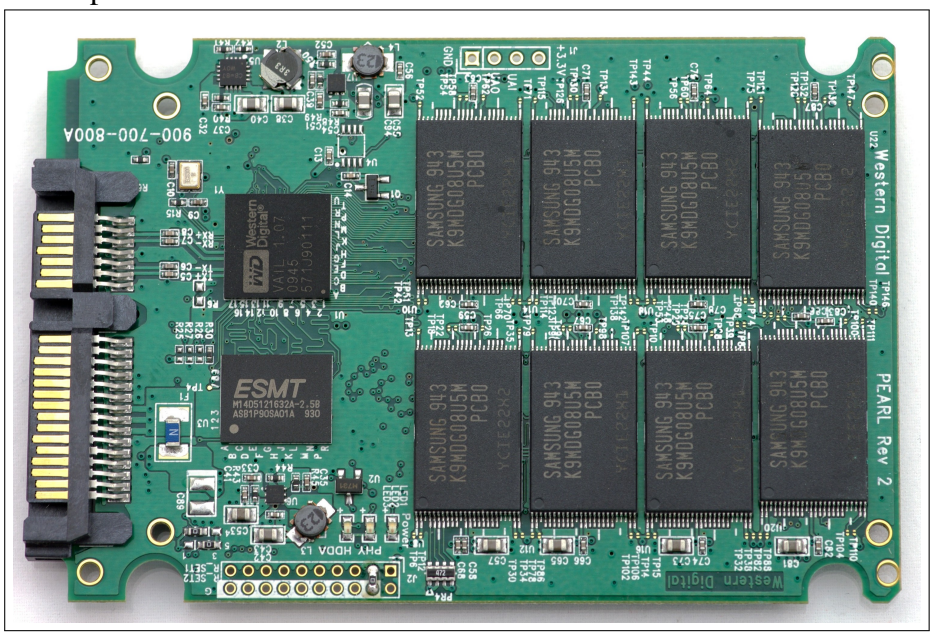
Flash memory is a computer chip which can be electrically reprogrammed and erased. Flash memory stores data in an array of floating-gate transistors, called cells (BRAYNER; NASCIMENTO, 2013). In fact, cells in flash-based SSD "store" two different voltage levels. Typically, a cell with a voltage level higher than 5v means a bit 0. On the other hand, a level less than 5v represents a bit 1, which is the default state for any flash-based SSD.

There are three different types of flash memory: single-level cell (SLC) in which only one bit can be stored in a cell, and multilevel cell (MLC) which, in turn, stores two bits per cell and finally triple-level cell (TLC), whose transistors can store three bits per cell, but at an even higher latency and reduced lifespan expenses.

In Figure 3, we may see an SSD device. Looking carefully one can observe some differences when compared to HDD. First of all, the mechanic parts are absent in SSD devices, and all the information are stored using flash memory chips. It is also possible to observe that, just like HDD, SSD also uses SATA interfaces to flow data with other computing devices. Furthermore, one can note two chips that are separated by the other, these are the cache buffer chip and the Flash Translation Layer (FTL) microchip, responsible for managing the read and write operations on physical media.

Generally, a flash-based SSD integrates three different components, namely: host interface, internal processor, SDRAM buffer, flash controller and several flash memory packages (PARK *et al.*, 2012). Figure 4 brings an abstract model of those components in flash-based device. The host interface connects the flash memory to a computer CPU. SATA, SAS or PCIe are different types of host interfaces. The main functionality of the SDRAM buffer is to hold the

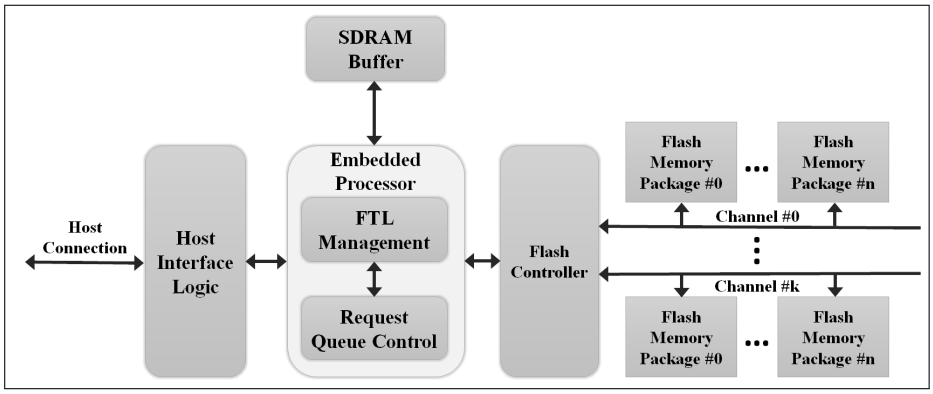
Figure 3 – Example of an SSD Device



Source: Sullivan (2010).

address mapping table, which in turn associates computer main memory addresses to physical addresses in flash memory. I/O requests are sent and received by the flash controller. Moreover, the flash controller connects flash memory packages through a multi-channel bus.

Figure 4 – SSD Components



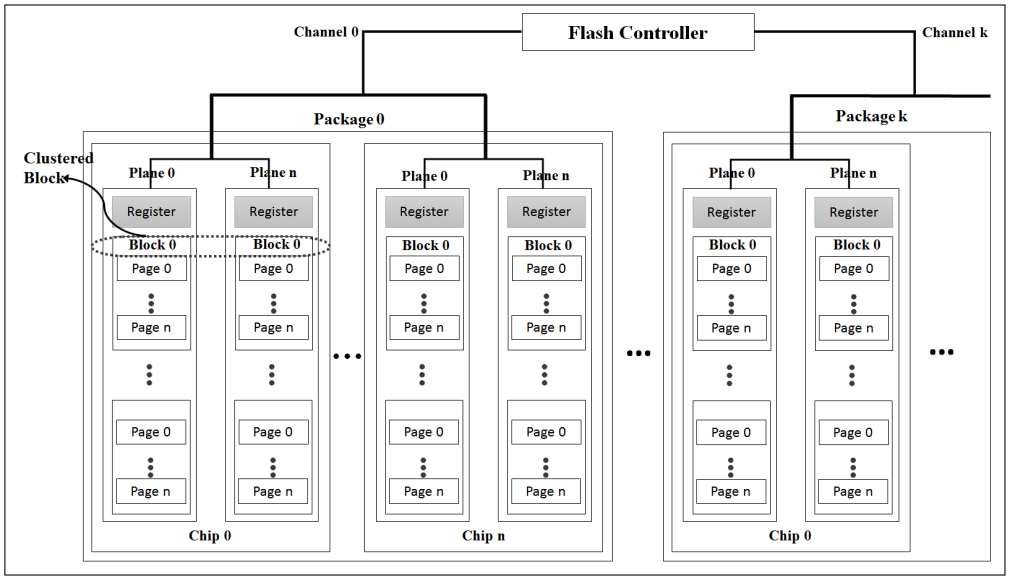
Source: The author.

The main goal of the flash memory internal processor is to manage the execution of the request queue control and of the FTL. FTL plays a key role in flash-based SSDs, since it is responsible for running the following processes: wear leveling, garbage collection and address mapping, which is responsible for managing the address mapping table (stored in SDRAM buffer). Before describing the wear leveling and garbage collections, some physical features of flash-based SSDs should be presented.

A flash memory package is composed of several chips (also called dies). Each chip

(die) presents several planes, each of which contains a set of blocks. In turn, each block is divided into pages. Typically, the page size may vary from 2KB up to 16KB. Most SSDs have blocks of 64, 128 or 256 pages, which means that the size of a block may vary from 128KB to 4MB (DIRIK; JACOB., 2009). Figure 5 depicts an abstract model of a flash device.

Figure 5 – SSD internal architecture



Source: The author.

SSDs drives are designed in a such way that multiple regions can be accessed in a parallel or interleaved way. In order to achieve this access pattern, flash-based SSDs present four different levels of parallelism. The first level, denoted channel-level parallelism, arises from the fact that the FTL communicates with flash packages through multiple channels. These channels can be accessed independently and simultaneously. Each individual channel is shared by multiple packages as shown in Figure 5. The next parallelism level is the package-level parallelism, which assures that packages connected by a given channel can be accessed in parallel.

The third parallelism level stems from the characteristic that a package contains two or more dies. Thus, the dies in a package can be accessed simultaneously. Finally, the last parallelism level is the plane-level. This level assures that the same operation (read, write or erase) can be executed simultaneously on multiple planes inside a given die. Planes also contain registers (small RAM buffers), which are used for plane-level operations (see Figure 5).

In order to exploit the aforementioned parallelism features provided by SSDs, FTL stores data of a given file in a logical unit, called clustered block (KIM *et al.*, 2012). A clustered block contains SSD blocks belonging to different planes (of a chip), but which have the same address. Figure 5 illustrates a clustered block composed of blocks with address 0 in planes 0 and

1 of chip 0. This way, several SSD blocks of different planes can be accessed simultaneously. In most SSDs the size of clustered blocks may be 16 or 32MB.

There are three operations which should be executed on a flash device: read, erase and program (KIM; KOH., 2004). A *read* operation may randomly occur anywhere in a page within a flash device. An *e*rase operation has the functionality of setting to 1 all bits within a block. Erase operations is block addressable, i.e., it can not be executed on a single page. A *program* operation sets a bit to 0. A program operation can only be executed on a ”clean” block, which is a block with all bits set to 1. For that reason, the program operation is page addressable (BRAYNER; NASCIMENTO, 2013).

A flash block has its lifespane determined by the number of write operations. Consequently, the higher the number of write operations on a flash memory device, the shorter its lifespan is. In order to minimize such a restriction, the wear leveling process is employed to evenly spread write operations out across the storage area of the medium.

The garbage collection process is triggered whenever clean blocks are necessary to reduce the amount of erase operations. The combination of these two techniques induce to the rise of a phenomenon denoted write amplification. Due to the write amplification phenomenon, the number of physical write operations is,therefore, far larger than that of logical write operations (those in fact submitted by users or applications).

Table 1 summarizes some physical characteristics of three different SSD devices (Samsung 64GB, IntelX25-M and Samsung 840 EVO) available in the market. The specifications are those provided by manufacturers.

Brand/Model	Samsung SATA-II OVA	Intel X25-M	Samsung 840 EVO mSATA
Memory cell type	MLC	MLC	TLC
Interface	SATA 2.0	SATA 2.0	SATA 3.0
Capacity	64 GB	80 GB	1 TB
Pages per block	128	128	256
Page size	4 KB	4 KB	8 KB
Block size	512 KB	512 KB	2048 KB
Seq. Read (MB/s)	100	254	540
Seq. Write (MB/s)	92	78	520
4K Rand. Read (MB/s)	17	23.6	383
4K Rand. Write (MB/s)	5.5	11.2	352

Table 1 – SSD physical characteristics.

Usually, sequential writes consume less time than random writes do (see Table 1). Nonetheless, random writes may be faster than sequential writes. It happens when the size of the

written buffer is equal to or greater than the size of the clustered block. For example, when the size of the file is smaller than 32MB, a sequential write is faster than a random write. However, if a file is larger than 32MB the internal parallelism makes random parallel writes faster.

It is worthwhile to mention the feature of random read operations on a given file (see Table 1) tends to decrease IOPS rates w.r.t. sequential read in flash-based SSDs. Such a phenomenon occurs even when the file has been written using the notion of clustered block units. (CHEN *et al.*, 2011)

2.4 SSD versus HDD

A comparison between SSD and HDD is necessary, because these devices have been designed to storage data. Three approaches are qualitatively analyzed as follow.

2.4.1 Performance

One of the main reasons that SSD easily surpasses the performance of HDD is the lack of mechanical components. It is also important to note that SSD has its random access time less than 0.1 millisecond, while the most modern hard drives spend about 1 to 12 milliseconds, showing that the SSD might be up to 100 times faster than HDD. Furthermore, we know that the performance of HDD can be greatly hampered by fragmentation. (KIMM; PARK, 2013)

Moreover, HDD may suffer about 50% of reduction in its higher performance when it is full or contains a lot of fragmented data. On the other hand, SSD provides consistent performance regardless of the amount of data that they contain. SSD in the reading speed of a data can reach 540 MB/s, while HDD reading speed does not exceed 140 MB/s. (KIMM; PARK, 2013)

2.4.2 Endurance

The lack of mechanical parts in SSD impacts also its endurance. When a device has mechanical parts, there is an increasing risk of damaging its components. A SSD supports up to 40 times more vibrations than a HDD and is up to 4 times more resistant to falls. (KIMM; PARK, 2013)

2.4.3 Energy Efficiency

When a HDD needs to search any data, it is required to put the arm with the read/write head in the right track. It is also necessary to turn its magnetic disc at a very high speed. Therefore, the HDD consumes a considerable amount of energy. On the other hand, SSD has no mechanical parts and can access data anywhere consuming a very small amount of energy. An SSD spends on average only 0.127 watts per request, while a HDD requires an average about 1.75 watts. However, in the most of time, both SSD and HDD are in standby mode. In standby mode, SSD spends in average about 0.046 watts, while HDD spends about 0.8 watts (KIMM; PARK, 2013).

2.5 Summary

Certainly SSD is unmatched in the performance category, one can note that the SSD device offers a very low random access time and a read/write higher speed compared to HDD. On the endurance category, it is remarkable how an SSD is more durable than a HDD. On the issue of efficiency, we realize that SSD consumes much less power than HDD.

It is well-known that SSD have been enjoying wide acceptance in the market. Currently these devices are already presented as solution for data storage. Therefore, under DBMS perspective, it is necessary to consider SSD physical characteristics and review the algorithms used in DBMS in order to take advantage of such storage media.

3 THEORETICAL BACKGROUND

3.1 Introduction

In this chapter, the most relevant basic concepts that support this research will be described and analyzed.

For a better understanding the chapter is organized in sections: Section 3.2 describes the essential characteristics of a Logical Execution Plan (LEP) and how it works on a DBMS; Section 3.3, brings how some commercial DBMS deal with parallelism in query optimization; Section 3.4 conceptualizes the notions of temporary result and materialization; Section 3.5 concludes this chapter.

3.2 Query Execution Plan

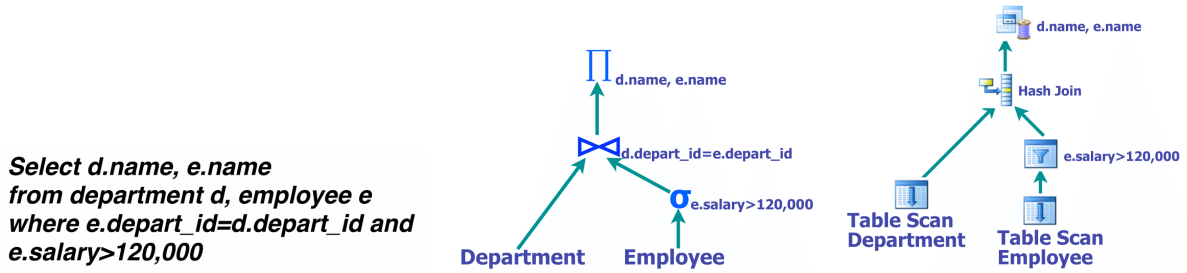
A query execution plan (QEP) describes the steps and the order used to access or modify data in the database. The process starts taking in consideration a query written in a language like Structured Query Language (SQL), which is analyzed and turned into a parsed tree representing the structure of the query in a more useful way. Then, parse tree is transformed into an expression tree of relational algebra (or a similar notation), which we term a LEP. Finally, the LEP plan must be turned into a Physical Execution Plan (PEP), which indicates not only the operations performed, but the order in which they are performed, the algorithm used to perform each step, and the ways in which stored data are obtained and how data are passed from one operation to another (GARCIA-MOLINA *et al.*, 2008).

To process database queries, the query engine should firstly map a given query Q expressed by means of SQL into an internal representation form, denoted LEP. A LEP for a query Q can be defined as a directed graph, whose nodes represent the tables or algebraic operations (of the relational algebra) in Q (GARCIA-MOLINA *et al.*, 2008), and directed edges (arrows) indicate data flow among operators. Thereafter, the query engine maps the generated LEP for Q into a PEP in which the algebraic operators are represented by their physical implementation. For instance, a join operation in a LEP may be represented in a PEP as merge join or hash join operator.

In order to illustrate the idea of mapping a SQL query to a LEP and from LEP to PEP, consider the SQL query Q depicted in Figure 6a. Additionally, let us assume that there are

no indexes defined on database tables Employee and Department. Examining query Q, one can see that there exists three operations from relational algebra, namely projection, selection and join. Regarding join operation, in Figure 6b, we may observe a join between 2 tables, namely Department and Employee. Is is called binary join or 2-way join. If a QEP contains a join among 3 or more tables, this join is represented as a sequence of binary joins and it is called n-way join. For instance, if there would exist a join among 3 tables (e.g. tables Department, Employee, and Dependent), the corresponding QEP would contain 2 joins: one operating on tables Department and Employee, and another one operating on the result of the preceding join and table Dependent. In this case, it would called a 3-way join.

In Figure 6b, a possible logical execution plan for Q is presented in which: *(*i) a selection operation, denoted by σ, with predicate $e.salary > 120.00$ is executed on the table Employee; *(*ii) thereafter there is a join operation, denoted by $\bowtie$, between the result of the selection and table Department, and finally; *(*iii) a projection operation, represented by Π on the join result is processed. In turn, Figure 6c brings a physical execution plan derived from the logical execution plan in Figure 6b. Observe that the initial step for processing Q is to read tables Department and Employee. The operation in a PEP responsible for reading a table is denoted table scan.



(a) SQL query Q. (b) LEP for Q. (c) PEP for Q.

Figure 6 – Query Q and its LEP and PEP.
Source: The author.

3.3 Parallelism on Commercial DBMS

Some commercial database management systems (for example, MS SQL Server (PINTO; HANSON., 2010), (BEN-GAN., 2011)) are able to make use of implicit parallelism in query optimization. MS SQL Server can scan a (non partitioned) table (or index) in parallel by using a consumer-producer strategy.

Depending on query context, the query engine produces plans in which operators, generically called exchange operators, are used in several ways for reading a table in parallel. The parallelism strategies, how a producer consumes a row or page from the database, may be: hash, round robin, broadcast, on demand and by range. On Demand and by range strategies are only used on table physically partitioned. For all strategies available, the only driven factor is the degree of parallelism, i.e., the number of available CPUs in query execution time. The higher number of CPUs/cores, the higher degree of parallelism is and the higher probability of executing a parallel scan on a table. Therefore, the MS SQL Server's parallel scan is not targeted to profit from characteristics of underlying media device.

DB2 can use a parallel table scan method which requires the following preconditions to work properly (CORPORATION, 2002): (i) DB2 UDB (DB2 Universal Database) Symmetric Multiprocessing feature must be installed; (ii) data must be distributed across multiple disk devices, and; (iii); the server on which DB2 is running has multiple processors. Based on that, one can note that DB2 works well in parallel, but it does not work using only one device, once the data must be distributed through multiple devices.

Oracle database may execute a parallel scan when the table involved is enabled to use parallel execution by a simple command query (KYTE; KUHN., 2014). The DBMS executes a parallel scan when at least one of the tables is accessed through a full table scan and the statement must contain a parallel hint specifying the corresponding table, or the corresponding table must have a parallel declaration in its definition. Once Oracle decides to execute a parallel scan, the degree of parallelism is determined by the instances' specification from the definition of all tables and indexes involved in the query. To choose the highest values found for those settings, it checks the statement for a parallel hint, when a hint is found, it overrides the degree of parallelism obtained as a result of the previous step.

Table 2 summarizes the critical features of the scan operators discussed in this section, i.e., the scan operators implemented by MS SQL SERVER, DB2 and Oracle. The first feature, denoted Enabled Controller Interface (ECI), indicates if the scan operator requires interface-specific settings (such as NCQ and AHCI) in order to be executed in parallel. The Hardware-Specific (HS) property specifies the fact that the approach is dependent on some physical characteristic of the SSD device, such as the adoption of RAID-0. In turn, User Intervention (UI) signalizes whether the operator requires or not a direct human action in order to execute the scan operator in parallel. Multiple Processors Availability (MPA) shows that the scan

operator can only run in multiple processors machines. Finally, Manufacture Information (MI) shows that the evaluated scan operator has to know information from the manufacture industry on the SSD device, on which the scan operator will run, such as the number of channels for setting a proper concurrency level and to avoid over-parallelization.

Approach	MS SQL Server	DB2	Oracle
ECI	No	Yes	No
HS	No	No	No
UI	Yes	No	Yes
MPA	No	Yes	No
MI	No	No	No

Table 2 – Table Scan Approaches in Commercial DBMS

3.4 Materialization Strategies on an execution plan

In this section, we will discuss the materialization strategies used in join algorithms operation, their main characteristics with some examples.

The join results may not include the entire tuples of source relations if only some attributes are needed to accomplish the join and some others are present in final projection. The query engine can choose a materialization strategy to reduce the size of intermediate results at the cost of rereads.

There are three types of materialization strategies. First, Early Materialization Strategy (EMS), which projects columns at beginning of query plan. Second, Late Materialization Strategy (LMS), which waits as long as possible for projecting columns and finally, Mixed Materialization Strategy (MMS) which tries to combine the best of the two first strategies.

3.4.1 Early Materialization Strategy

In EMS, all the attributes used on the query plan are added to the intermediate result since from the point which this attribute is read, it does not take into consideration when the attribute will be used in the query plan, whether immediately after or later on, after many operations. In other words, it may keep tuple values as soon as they are read in the query plan. EMS stores attributes which are needed later in the query plan (e.g., next join operations and/or final projection operation) as part of join result as soon as tables which have the attributes used in the plan. Take a simple 2-way join query $\prod(R.b, S.d)((R \bowtie S)_{(\text{R.a=S.c})})$, the attributes "a" of relation R and "c" of relation S are the join attributes and attributes "b" of R and "d" of S are the

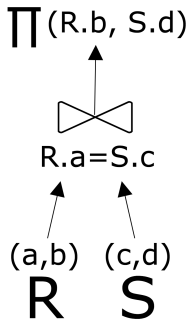
attributes of the query result.

The query in Figure 7a could be processed using any join algorithm. Figure 7b depicts a graphical representation for the query plan. In the arrows (edges) are indicated the attributes which will be used on next operator. It is important to note that the attributes used on the final project have been already read and kept for the first operator.

Figure 7 – SQL Sample and Query Plan for EMS

```
SELECT R.b, S.d
FROM R, S
WHERE R.a=S.c
```

(a) SQL for EMS



(b) EMS Query Plan

Source: The author.

3.4.2 Late Materialization Strategy

For LMS, the tuples are formed after some part of the query plan has been already executed. The attributes can be added to intermediate result only at the point where they are necessary. With our simple join query example in 7a, the join result will be composed only for the attributes R.a and S.c, once they are the attributes needed for join operation. LMS implies the need of another query operator which should reread attributes needed for next plan operations. This reread operator may be implemented as a standalone operator in the query plan or embodied in the next plan operator. It is important to note in Figure 8b that after completion of join operation, in the projection operation, R will be reread to retrieve attribute b and S will be reread to retrieve attribute d. Only the tuples which are part of the join result will be reread.

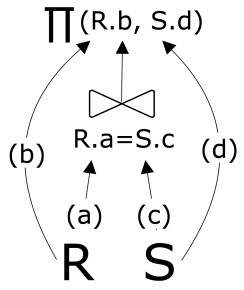
3.4.3 Mixed Materialization Strategy

MMS tries to combine the best of the two strategies, EMS and LMS. The MMS uses EMS for join attributes and LMS only at the end of the query plan for final projection attributes. MMS may represent a gain in the plan execution time of n-way join queries because no

Figure 8 – SQL Sample and Query Plan for LMS

```
SELECT R.b, S.d
FROM R, S
WHERE R.a=S.c
```

(a) SQL for LMS



(b) LMS Query Plan

Source: The author.

rereads are made between join operations and it reduces the size of intermediate results because final attributes are not stored in them. In other words, MMS represents a trade-off between the memory size needed to intermediate results and the processing costs of reread operations. Comparing with LMS, MMS uses more memory to store next join attribute values but do not perform rereads to retrieve these join attribute values. Note that this strategy may represent an advantage when the selectivity increases along the query execution plan because a small number of tuples are needed to be reread for retrieving final projection attributes.

Take a simple 3-way join query $\prod(R.b, S.d, T.f)(((R \bowtie S_{(\text{R.a=S.c})}) \bowtie T)_{(\text{S.e=T.f})})$, the attributes "a" of relation R and "c" of relation S and "f "of relation T are the join attributes, and attributes b of R and d of S and f of T are the attributes of the query result. Figure 9b illustrates a query plan for our query example when MMS is applied. One can note that the difference between LMS and MMS is the fact that there is no reread on intermediate results, i.e., for intermediate results MMS works as EMS and for final projection it works as LMS.

3.5 Summary

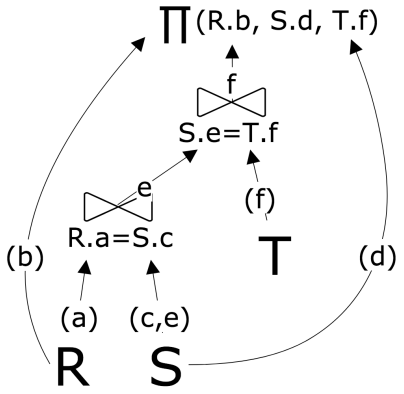
As SSDs may provide IOPS rates up to two orders of magnitude greater than the rates delivered by HDDs (PARK *et al.*, 2011), SSD technology has emerged as a feasible substitute for traditional magnetic disks HDD for storing large databases.

An important feature of the SSD device is the internal parallelism for reading and writing data. Generally, SSD IO parallelism is implemented in different four levels, each of them is specific for an SSD structure: channel-level, package-level, die-level and plane-level. Combining SSD parallelism levels make possible to access multiple blocks simultaneously across separate chips, as a unit called clustered block. Therefore, internal parallelism provided by SSD

Figure 9 – SQL Sample and Query Plan for MMS

```
SELECT R.b, S.d, T.f
FROM R, S, T
WHERE
     R.a=S.c
and S.e=T.f
```

(a) SQL for MMS



(b) MMS Query Plan

Source: The author.

manufacturing parts and assembly opens opportunities to improve data access rates in SSDs.

In classical database query processing, a scan operator is responsible for sequentially reading a database table. Thus, scanning very large tables may negatively impact on query response time. In this sense, several investigations have been carried out in order to implement a parallel scan operator on single table. Some existing database systems, such as MS SQL Server, IBM DB2 and Oracle, already implement strategies to parallelize the execution of a table scan operator. Those strategies aim at increasing data delivery throughput, decreasing, this way, query response time. Nonetheless, they do not take any profit from internal parallelism provided by SSDs.

The choice among EMS, LMS and MMS must consider two important factors: the projectivity and the selectivity of the query, where projectivity represents the number of projected attributes in a given query operation and selectivity represents the number of tuples which satisfy a predicate in a selection or in a join operation. Join selectivity factor (JS) was first defined by (VALDURIEZ, 1987) as follow:

$$JS = \frac{(R \bowtie S)}{(RxS)} \tag{3.1}$$

The value of JS ranges from 0.0 to 1.0. For example, suppose initially there are 600 tuples on table R and 400 tuples on table S, and the query produces 800 tuples. The JS is the result of the operation above:

$$JS = \frac{800}{(600 * 400)} \tag{3.2}$$

$$JS = 0.00334 \tag{3.3}$$

For high projectivities LMS fits well, once it is not necessary to store a high amount of data in main memory. On the other hand, EMS will be penalized by high projectivities because more columns will be stored in the intermediate results demanding as much memory as is needed to execute the operation. Then, when there is no enough memory an overflow may occur. Based on that, additional writes on secondary memory may occurs and as we have already known it is detrimental for a SSD device due the fact that read/write asymmetry and the lifespan dependent on amount of writes on this type of media.

LMS may contribute to reduce the number of writes on SSD, once it will access the media more frequently to reread needed attributes. More, the impact of a high projectivity can be distributed between operands and final projection. It might happen when an attribute is necessary for selection, for joining or for final result, exclusively, i.e., it will be read on demand. On the other hand, if an attribute appears in all operations of the query plan, it will be carried from the beginning to the end as like as in EMS.

(ABADI; DEWITT, 2007) have shown that for column-oriented database architectures, LMS is the better choice when there is no join on the query plan. It is also true that selectivity has a higher influence for both row-oriented and column-oriented databases and for both materialization strategies, but the authors have affirmed that depending on the amount of memory available and the number of projected attributes LMS might be two times slower than EMS.

When a query has low JS, the amount of both input and output tuples will be small. This approach is beneficial for both EMS and LMS. However, LMS may be more efficient than EMS, because, although no rereads are made in EMS, the amount of main memory needed for intermediate result can increase considerably for joins between large relations with high projectivity, and an overflow may occur reducing considerably the query plan performance.

4 NON SSD-AWARE - JOIN ALGORITHMS

4.1 Introduction

In this chapter, we will discuss the join algorithms (also called join operators)[1], showing their main characteristics and their execution costs.

Traditional join algorithms are implemented under three different strategies. The first one is the nested loop strategy, which is used in the nested-loop join and block nested-loop join algorithms. The second one, we have a tuple ordering strategy by join attribute, which is used in the merge-join algorithm. The third one is a hash function-based strategy, which is used in the hash join and hybrid hash join algorithms.

The query processor uses the estimated cost to choose the best execution plan. In this study, the number of pages read and written during the algorithm execution will be considered as estimated cost of the join operator. In traditional join algorithms this estimate does not take into consideration the most peculiar feature of SSD, which is the asymmetry between the read and write operation on SSD devices. We will now describe the traditional join algorithms and show their estimated cost.

Some acronyms used in this chapter have been listed as follows.

- B: Pages on buffer
- buck: Buckets kept in main memory
- DHri: Bucket i from table R kept in secondary memory
- DHsi: Bucket i from table S kept in secondary memory
- max: Maximum number of different results calculated from the hash function
- maxMem: Main memory reserved for execution
- MBuckHr: Buckets in main memory for table R
- MBuckHs: Buckets in main memory for table S
- MHri: Bucket i from table R kept in main memory
- MHsi: Bucket i from table S kept in main memory
- NR: Number of tuples for table R
- NS: Number of tuples for table S
- NumMaxMem: Maximum number of bucket remain in main memory

[1] The terms join algorithm and join operator are used interchangeably throughout the text. The term join operation refers to the algebraic join operation, independently of any algorithm used.

- PE: Number of pages for external table
- PR: Number of pages for table R
- PS: Number of pages for table S
- TR: Tuple of table R
- TS: Tuple of table S
- usedMem: Main memory used for algorithm execution

4.2 Nested Loop-Join

The nested-loop join, which uses the nested loop strategy, is an algorithm that checks if the join condition is satisfactory at each iteration. For a join operation R $\bowtie$ S, the join condition is checked for each tuple contained in the table R against all tuples in table S. Tuples and pages do not receive any special treatment to improve performance, which consequently generates a very large number of inputs and outputs (I/Os) (GARCIA-MOLINA *et al.*, 2008).

```
1    FOR each tuple r in R DO
2        FOR each tuple s in S DO
3            IF r and s join to make a tuple t THEN
4                output t ;
```

Algorithm 1 – Pseudo-code for Nested Loop-Join

Considering the algorithm above, we can assume that the estimated cost of the nested loop join algorithm is PR + (NR * PS)

4.3 Block Nested Loop-Join

The algorithm 2, block nested-loop join, also uses the nested loop strategy. This algorithm is an improved version of 4.2, organizing access to both relations by blocks and using as much main memory as it can to store tuples belonging to the relation R, the relation of the outer loop (R $\bowtie$ S). In other words, block nested loop-join make sure that when it is running through the tuples of S in the inner loop it uses as few disk I/O's as possible to read R, enabling to join each tuple of table R read with as many tuples of S as will fit in memory.(GARCIA-MOLINA *et al.*, 2008)

For each block of table R, table S is read once completely, verifying tuple the tuple

if the join condition is satisfactory. In this way, there is a decrease in the number of times the internal table is read, consequently decreasing the number of I/O's.

```
1  FOR each chunk of M-1 blocks of S DO BEGIN
2   read these blocks into main-memory buffers;
3   organize their tuples into a search structure whose
4    search key is the common attributes of R and S;
5   FOR each block b of R DO BEGIN
6    read b into main memory;
7    FOR each tuple t of b DO BEGIN
8     find the tuples of S in main memory that
9      join with t;
10    output the join of t with each of these tuples;
11   END;
12  END;
13 END;
```

Algorithm 2 – Pseudo-code for Block Nested Loop-Join

With the modifications that the chunk reading brought to the algorithm, we can conclude that the execution cost for block nested-loop join is PR + (PR * PS).

4.4 Merge Join

Merge-join algorithm 3 uses the tuple sort strategy. It is important to note that is unlikely that the query processor will choose the merge-join if the tables are not previously sorted by the join attribute, once they are sorted its execution cost is very low PR + PS (GARCIA-MOLINA *et al.*, 2008).

However, for the sake of clarification we will consider that the tables are not sorted, in this case we might say that the merge-join is an algorithm divided in two phases. The first phase is to rewrite the tables sorting them by the join attribute. In the second phase, the first tuple of table S is read and for each tuple of table R whose respective join attribute is smaller than or equal to the corresponding join attribute of the tuple of table S, it verifies if it satisfies the join condition. If it satisfies, then (TR, TS) are included in the result. If it does not satisfy, the algorithm checks whether there are more tuples in table S, and if it does, it reads the next tuple of table S. This operation is repeated until the entire table S is read.

```
1  /*SORT PHASE*/
2  SORT_TABLE(R)
3  SORT_TABLE(S)
4
5  /*JOIN PHASE*/
6  READ first tuple of S
7   FOR each tuple tr of R DO BEGIN
8    WHILE (tr[atr_join] >= ts[atr_join]) AND (~(end(S))) DO
9      IF match (tr, ts)
10      output the join
11   READ next tuple of S
12   END WHILE;
13  END
```

Algorithm 3 – Pseudo-code for Nested Loop-Join

We have already considered that merge-join is a two-phase algorithm, based on that, we will analyze each phase separated for a better execution cost understanding. In the first phase, merge-join executes a sort algorithm for both, table R and table S, taking into consideration that this algorithm avoids some I/O using main memory into B for better execution. In the second phase, each table will be read once comparing if the join attribute matches.

Based on that, we must say that the execution cost for merge-join is the execution cost for sort Table R:

$$2PR(\log_{B-1}(PR/B)+1) \tag{4.1}$$

Sort Table S:

$$2PS(\log_{B-1}(PS/B)+1) \tag{4.2}$$

Read each table once:

$$(PR+PS) \tag{4.3}$$

Thus, we conclude that the total cost for merge-join algorithm is:

$$2PR(\log_{B-1}(PR/B)+1)+2PS(\log_{B-1}(PS/B)+1)+(PR+PS) \tag{4.4}$$

4.5 Hash Join

Hash join, algorithm 4, is also an algorithm divided into two phases, where the first phase is used to partition the tuples of the tables into buckets, based on the values of a hash function applied on the join attribute. With the tuples separated by buckets, the second phase is used to execute the join and only the tuples which are on the same bucket address will be compared and joined. This comparison will be done using the algorithm index nested loop-join, which in turn is a nested loop-join 4.2 that build a index mapping the join attribute of the inner bucket relation, then seeks on it for the searched value(s) and stops looking further (GARCIA-MOLINA *et al.*, 2008).

```
1  /*BUCKET PHASE*/
2  /*Table R*/
3  FOR each tuple tr of R DO BEGIN
4   i=hashFunction (tr [atr_join)
5  Hri = Hri U {tr}
6
7  /*Table S*/
8  FOR each tuple ts of S DO BEGIN
9   i=hashFunction (ts [atr_join)
10 Hsi = Hsi U {ts}
11
12 /*JOIN PHASE*/
13 From i=0 to max DO BEGIN
14  READ Hsi build hash index
15  FOR each tuple tr of Hri DO BEGIN
16  find(ts, Hsi)
17    IF match (tr, ts)
18      output the join
19  END;
20 END
```

Algorithm 4 – Pseudo-code for hash join

Hash Join is a two-phase algorithm, based on that, we will analyze each phase separated for a better execution cost understanding. In the first phase, the algorithm builds the buckets for both, table R and S, using a hash function that returns a limited number of results,

called as *max* and it represents the maximum number of different results calculated from the hash function. In the second phase buckets will be read only once, comparing if the join attributes match.

Based on that, the execution cost for hash join is the execution cost for build buckets for both tables, R and S:

$$2(PR+PS) \tag{4.5}$$

Join phase:

$$(PR+PS)+2*max \tag{4.6}$$

Thus, we conclude that the total cost for hash join algorithm is:

$$3(PR+PS)+2*max \tag{4.7}$$

4.6 Hybrid Hash Join

In order to save reads and writes in secondary memory, hybrid hash join has been developed based on conventional hash join. The difference between them is in the first phase of the algorithm. When the algorithm is partitioning the table, some buckets will remain in main memory and, because of that, it is not necessary to write them to disk, reducing the number of writes. Consequently, the second phase will read from the second memory only those buckets written in the disk on the first phase, this approach improves the performance of the join and reducing some I/O's (GARCIA-MOLINA *et al.*, 2008).

There are two strategies for implementing hybrid hash join. The first one, limits the number of buckets that will remain in memory. This number is given by the formula:

$$(max/NumMaxMem+max-(maxMem \leq PE)) \tag{4.8}$$

The second strategy is to limit the use of main memory. Thus, the number of buckets is determined by the ratio of maximum memory usage expressed in bytes divided by the page size, also expressed in bytes. Buckets that remain in memory for TR must be the same as those for TS and the buckets that remain on disk for TR must have their correspondents for TS also on disk.

Next we will introduce the hybrid hash join pseudo code algorithm 5, using the second strategy, that is, limiting the main memory usage. It is important to say that in our experiments in chapter 8 we have used this same strategy.

```
1  /**BUCKET PHASE**/
2  /*Table R*/
3  FOR each tuple tr of R DO BEGIN
4   i=hashFunction (tr [atr_join)
5    IF(memUsed < maxMem)
6     MBuckHr = MHri U {tr}
7     memUsed += tr(bytes)
8    else
9     FOR each tuple MHri of MBuckHr DO BEGIN
10     DHri = MHri U {tr}
11     memUsed -= tr(bytes)
12    END
13 END
14
15 /*Table S*/
16 FOR each tuple ts of S DO BEGIN
17  i=hashFunction (ts [atr_join)
18   IF(memUsed < maxMem)
19    MBuckHs = MHsi U {ts}
20    memUsed += ts(bytes)
21   else
22    FOR each tuple MHsi of MBuckHs DO BEGIN
23     DHsi = MHri U {ts}
24     memUsed -= ts(bytes)
25    END
26 END
27
28
29 /**JOIN PHASE**/
30 /*MAIN MEMORY JOIN*/
31 From i=0 to max DO BEGIN
32  FOR each tuple tr of MHri DO BEGIN
33   FOR each tuple tr of MHsi DO BEGIN
34    IF match (tr, ts)
35    output the join
36   END;
37  END
38 END
```

```
39 /*DISK JOIN*/
40 From i=0 to max DO BEGIN
41  READ Hsi build hash index
42  FOR each tuple tr of Hri DO BEGIN
43  find(ts, Hsi)
44    IF match (tr, ts)
45      output the join
46  END;
47 END
```

Algorithm 5 – Pseudo-code for Hybrid hash join

The execution cost for hybrid hash join follows the same line as the conventional hash join, but there are some partitions remain in memory, because of that, we need to take in consideration buck. Thus, we conclude that the total cost for hybrid join algorithm is:

$$(3-2(buck/max))*(pr+ps)+(3-2(buck/max))*max \tag{4.9}$$

4.7 Summary

This chapter presented the classical join algorithms, showing their main characteristics and their execution cost. One can classify them as nested loop algorithms and two-phase algorithms. *Nested Loop-Join* and *Block Nested Loop-Join* are the nested loop join algorithms, these simple join algorithms work even when neither argument fits in main memory. They read as much as they can of the smaller relation into memory, and compares that with the entire other argument; this process is repeated until all smaller relation has had its turn in memory.

Two-phase algorithms can be sub-split in two different approaches. First, *Sort-Based Algorithms* like *Merge Join*, this approach parts their tables into main-memory-sized, sorted sub-lists. The sorted sub-lists are then merged appropriately to produce the desired result. Second, the *Hash-Based Algorithm*, this approach uses a hash function to partition the tuples into buckets, then the join operation is applied to the buckets individually (GARCIA-MOLINA *et al.*, 2008).

It is known that *Hash-Based Algorithms* are often superior to *Sort-Based Algorithms*, since they fit well for the most common daily base use. Sort-based algorithms, on the other hand, work well when at least one of the table is already sorted or there is another reason to keep some data sorted (GARCIA-MOLINA *et al.*, 2008).

5 SSD-AWARE - JOIN ALGORITHMS

5.1 Introduction

In this chapter, we will discuss three join algorithms operation which are adapted for SSDs characteristics. First, Random Read Efficient Join (RARE join) (MEHUL HARIZOPOULOS STAVROS, 2008). Then, Flash join (TSIROGIANNIS *et al.*, 2009) and finally, Bt-Join (EVANGELISTA *et al.*, 2015).

5.2 RARE join

(MEHUL HARIZOPOULOS STAVROS, 2008) have proposed an algorithm that uses mixture of random reads and sequential I/O allowing to speed up projections and join operations on SSDs devices, once this kind of device may have fast seek time. RARE join operator was designed for a type of column-oriented database called Partition Attributes across (PAX) (ABADI DANIEL J., 2008), which has row-oriented pages within a column-store layout. PAX is able to create mini-pages and store data on each column of the table's vertical partition.

RARE join has two main conceptual steps. First, it executes a join index by accessing only the join columns of the input tables. Then, it uses PAX layouts on flash to compute the join result in a single read pass through the input. Based on that, we may say that the main idea is to save I/Os by accessing only the join columns and mini-pages holding the values needed in the result rather than the entire input.

The join algorithm was analyzed into two basic modes. First, when there is enough memory to compute the join in only one pass through the input. Second, when more passes are needed, in other words, when there is not enough memory to hold the hash-table in the main memory.

On the pseudo-code algorithm 6 we may see the 1-pass RARE join, i.e., when the memory is sufficient to execute the join in one pass over source relations, R and S. The hash-table for inner relation (S) fits in memory (line 1). In line 3 the algorithm reads and builds a hash-table on the join column for outer relation (R). In lines 5-8, for all matches, RARE join fetches the result into join-result from source relations R and S.

```
1 Read join attr of S and build hash-table
2
3 Read join attr of R and probe hash-table
4
5 foreach join result <rowId_S, rowId_R> do
6  Read projected values of row rowId_R from R
7  Read projected values of row rowId_S from S
8  Write result into join_result
```

Algorithm 6 – RARE-join: when join index and projected attributes of inner relation fits in memory

(MEHUL HARIZOPOULOS STAVROS, 2008) in algorithm 7 also have shown the (1 + x) pass RARE join, i.e., when the hash table of S fits in memory but does not fit its projected attribute values. On the pseudo-code above one can note that the algorithm can still compute the result with only one pass through the input, but it must materialize the join index.

```
1  Read join attr of S and build hash-table
2
3  Read join attr of R and probe hash-table
4
5  foreach join result <rowId_S, rowId_R> do
6   Read projected values of rowId_R from R
7   Write projected values into partition of join_result
8   Write rowId_S into partition of temporary_file
9
10 Read temporary_file and process it.
11
12 foreach partition of temporary_file do
13  foreach rowId_S into partition do
14   Read projected values of rowId_S from S
15   Write values into partition of join_result
```

Algorithm 7 – RARE-join: when only join index fits in memory

Algorithm 8, RARE join needs more than one pass over source relations, R and S, to accomplish the join. Two first steps read R and S. Then, RARE join compute the partitions

of the join column for both tables by doing that each S partition can fit in memory. In the next step, the algorithm computes and materializes the join index over <rowId-S, rowId-R> for each partition. It is important to note that each partition of the join index is sorted by rowId-S. In line 11, RARE join merges the partitions of temp-JI on rowId-R. It spools the projected values into the join-result and the rowId-S into a partition called temp-file-S. Finally, for all partition from temp-file-S, the algorithm reads the projected values of rowId-S from S and write those values into the join-result.

```
1  Read join attr of S and partition it (hash on join value)
2  Read join attr of R and partition it (same hash function)
3
4  Compute temp_JI holding join index
5  foreach partition of S do
6   Read S and build hash-table
7   Read partition of R and probe hash-table
8   foreach row in join_result do
9    Write rowId_S, rowId_R in temp_JI partition
10
11 Merge partitions of temp_JI on rowId_R
12 foreach join result <rowId_S, rowId_R> do
13  Read projected values of rowId_R from R
14  Write projected values into partition of join_result
15  Write rowId_S into partition of temp_file_S
16
17 Read temp_file_S filled with rowId_S and process it
18 foreach partition of temp_file_S do
19  foreach rowId_S in partition do
20   Read projected values of rowId_S from S
21   Write values into partition of join_result
```

Algorithm 8 – RARE-join: when join index does not fit in memory

In order to analytically evaluate RARE join, the authors have compared it with Grace-hash join and its variant for PAX layout, called Grace-PAX. It is important to note that the empirical evaluations of this work were made through a formula of its own cost model. However, there are no evidence of time consumption, memory usage or CPU cost. Nevertheless, we may assume that RARE join is an adaptive algorithm for SSD devices once it takes advantages of the

SSD characteristics.

5.3 Flash Join

(TSIROGIANNIS *et al.*, 2009) presented two SSD-aware query operators called FlashScan and Flash join. FlashScan is a scan operator whose main functionality is to efficiently read the required attributes to process a given query from the SSD device, in addition, it uses PAX-layout to reduce the amount of data read. Flash join is a multi-way equi-join algorithm, implemented as a pipeline of stylized binary joins. Flash join is composed for two separated pieces, join kernel and fetch kernel. In a few words, we can say that join kernel is responsible to compute the join and outputs a join index tuple composed by the join attributes as well as the row-id (RID) of the participating rows from base relations. Fetch kernel is responsible for re-read the needed attributes using the RID specified in the join index.

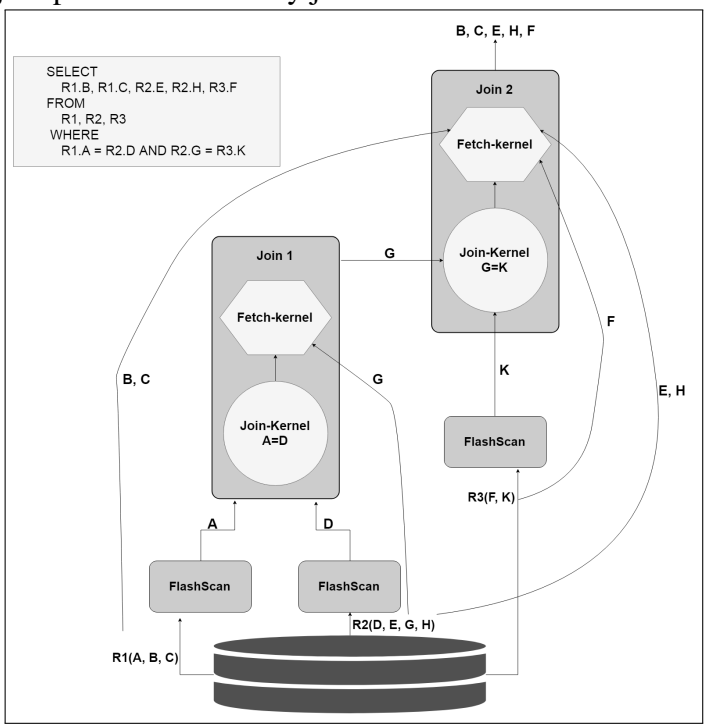
As we have seen before, join kernel is responsible to execute the join algorithm, it leverages FlashScan to fetch only the join attributes needed from base relations. It is important to understand that join kernel is implemented as an operator in the iterator model. In other words, it can implemented by any existing join algorithm: block nested loops, index nested loops, sort-merge, hybrid hash, etc. The authors work uses the hybrid-hash join algorithm (DEWITT R. H. KATZ, 1984).

Fetch kernel implements the FlashScan operator. Thus, fetch kernel is responsible for reading only the required attributes for processing the next join operation (or the final result). This feature is called LMS and has the functionality of reducing the amount of main memory used by Flash join, this approach offers some important benefits over traditional joins, which use an EMS. As RARE join, Flash join has also been designed for PAX-layout, which the authors advocate that is especially suitable for SSD devices due to its small transfer unit and its column-store layout. They also have shown that the better performance intended for scans is achieved just for highly selective conditions (with a small number of tuples in the result) and for selection conditions that are applied to sorted or partially sorted attributes.

Flash join's algorithm was implemented inside the PostgreSql DBMS as a new join operator. Additionally, a version of Hybrid-hash join over PAX layout, called HPAX, was also implemented and compared with Flash join. The Hybrid-hash join originally implemented in PostgreSql DBMS was also compared with Flash join.

Figure 10 represents flash-join's process for a n-way join. The arrows represent

Figure 10 – Flash join process for a n-way join



Source: The autor.

the data flow. In the first join, mini-pages of relation R1 containing the join attribute "A" and mini-pages of relation R2 containing the join attribute "D" is read by FlashScan operator. The join kernel of Flash join performs the join using Hybrid-hash join and fetch kernel re-reads relation R2 to retrieve "G" which is the join attribute of next join. The second join is between the attribute ,"G" , retrieved from table R2 in the first and the attribute "K" from the table R3. Final projected attributes of source relations R1 are "B" and "C", from the table R2 are "E" and "H". Finally, from the source relation R3 it is only the attribute "F".

As we have already known, Flash join uses LMS and this approach may lead additional reads in source relations to retrieve attributes needed for the next join process or for the projection operation. Fetch kernel phase is responsible for make those additional reads, it uses the join index produced on the join kernel step to materialize those attributes. It is important to understand that this approach may result in reading the same page multiple times. It occurs, when there is no memory enough to keep all pages needed, in main memory, to generate the result. It also happens because of the RIDs within join index are not ordered. In order to mitigate this overhead, Flash join has implemented additional passes over the join index.

The algorithm 9 makes multiple passes over the join index fetching attributes in row order from one relation at a time. For each pass, it sorts the join index based on the RIDs of the current relation to be scanned. Then, each needed tuple from that relation is retrieved and added

to the join index. If the join index does not fit in memory, it uses an external merge sort.

```
1  /*
2  Input: R1, . . . , Rk: base relations, id1, . . . , idk: corresponding
3  RID attributes, A1, . . . , Ak: Ai is the set of
4  attributes to fetch from relation Ri, |A1| < . . . < |Ak|,
5  JI: join index
6  Output: JR: join result
7  */
8
9  T = SortedStream(JI, id1)
10  for i = 1 to k do
11   Z = {}
12   while T is exhausted do
13    for all memory resident tuples t of T do
14     Add GeneratePartialResultTuple(t, idi, Ri, Ai) to Z
15    end for
16    Produce sorted run of Z on idi+1
17   end while
18   T = SortedStream(Z, idi+1)
19  end for
20 JR = T
```

Algorithm 9 – Fetch Kernel to produce the join result with multiple passes.

Sorting the join index does not guarantee sequential access to the underlying relation because the needed pages might be far apart from each other. However, this strategy of decoupling and postponing materialization is better suited for SSDs than HDDs devices.

(TSIROGIANNIS *et al.*, 2009) also explain four optimization on the algorithm. First, to avoid unnecessary final write, they pipelined the final merge of the sort and the needed attribute retrieval. *SortedStream* sorts the join index but leaves out the last merge step that creates a final sorted run (line 9). As tuples are fetched from T (a sorted stream), it merges the underlying runs on demand. For each tuple, *GeneratePartialResultTuple* retrieves the needed attributes and augments to the join index (line 14). It is important to understand that only enough tuples are read so that the result Z, the new join index with the attributes Ai projected, can fit in memory (line 13). Finally, Z is sorted on the RID of next relation (line 16). The sorted Z runs are then merged into a single sorted stream T (line 18) which is then pipelined with attribute retrieval

from the next relation.

Second, fetch kernel may process relations in the order of attributes length, from smallest to largest. This approach may reduce the data spilled to SSD when producing the intermediate runs. The third optimization will be realized only if Z and JI fit in main memory, avoiding writes on secondary memory. Fourth, if fetch kernel already has its inputs sorted by RIDs for one of the relations, it processes that relation first to avoid the sort execution.

They also explain another optimization which was not implemented, it would be an analogous fetching strategy would be to use hashing instead of sorting, as done in RARE join. For the Flash join they could, in each pass, hash join index tuples into buckets based on the page ID in the RID. This approach would ensure that all tuples which are in the same page will be keep into the same bucket, avoiding sorting algorithm. According to the authors, this strategy requires less CPU overhead than the sort-based one, but it would need more main memory than the sort-based one for the same number of partitions.

In the experimental evaluation section, the authors noticed that when memory allocated allows the algorithms to compute the join in one pass, Flash join's performance has a direct relation with both projectivity and join result cardinality.

The authors highlighted that, when the cardinality is 1% or greater, Flash join and HPAX perform like the same. On the other hand, for cardinalities which are less than 1%, Flash join reads less data than HPAX.

Experimental evaluation has been made when it is necessary two passes to compute the join. The execution time of Hybrid-hash join and HPAX increases linearly with projectivity, since more data participate in the partitioning phase, for the reason that these two algorithms use EMS in the query execution plan. Hence, Flash join reaches a more pronounced improvement when less memory is allocated to join and join needs two passes to be accomplished, reaching response times 7x faster than Hybrid-hash join and HPAX for a query which projectivity is 100% and result cardinality is less than 1%.

One can note that, when result cardinality reaches 100% of the largest relation's cardinality, Flash join's performance follows the curve through different projectivities of Hybrid-hash and HPAX joins. Based on that, we may say that selectivity has a major effect in Flash join's performance improvement than projectivity, i.e., as the cardinality increases, the performance difference between Flash join, Hybrid-hash and HPAX joins decreases.

5.4 Bt-Join

(EVANGELISTA *et al.*, 2015) proposed a SSD-aware join operator called Bt-Join. This algorithm uses an extended version of the B^+-tree (CORMEN CHARLES E. LEISERSON; STEIN, 1990), called B^{+Bt}-tree, as underlying structure for processing the join algebraic operation. The extension takes place in leaf nodes which hold join attribute values along with the so called *rowID List*. The idea is to use this notably structure in memory for storing and efficiently retrieve data in a block-oriented storage. It is important to understand that on a B^+-tree all data is kept in leaf nodes, so the leaf nodes are linked together as a double-linked list.

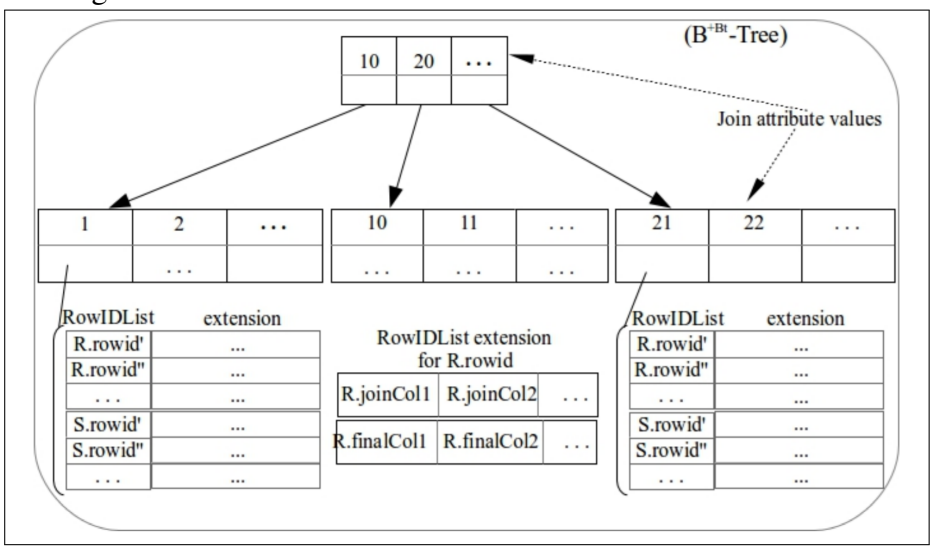
The algorithm is divided in two phases. First, *Tree-build phase*, an B^{+Bt}-tree is built upon the inner (smaller) relation. This relation is read and, for each tuple, the join attribute value is inserted in the B^{+Bt}-tree building a row-id list with the tuple information. The second phase is called *join phase*, where the outer (bigger) relation is read and, for each tuple, the algorithm searches for the join attribute value. Whether the join attribute value is present in the B^{+Bt}-tree, the correspondent row-id list is updated receiving the tuple information of outer relation. On the other hand, if value is not found, the tuple of outer relation is discarded.

(EVANGELISTA *et al.*, 2015) said that the intention behind of using a tree structure instead of a hash for join $R \bowtie S$ is that it may save 2(PR + PS) SSD accesses because it does not need to create partitions. The result of a join operation using Bt-Join algorithm is stored in a B^{+Bt}-tree.

Figure 11 shows the concept of a B^{+Bt}-tree. Each entry in a leaf node is composed of the join attribute values, which can be composed by several attributes, and their correspondent rowID Lists. By storing only, the required join attributes and the correspondent row ID in temporary result, Bt-Join reduces the number of writes on temporary results in secondary memory. Depending on the main memory available, Bt-Join is able to process a join operation in main memory.

Bt-Join explores two types of materialization. First, EMS, which implies to store the attributes needed for the next operations as part of the join result as soon as the relation which has the attributes appears in the query execution plan. This materialization strategy works better when the result of a join operation does not include the whole tuples with all its attributes. Second, MMS works better when the result of a join operation includes the whole tuples with all its attributes. Once, the MMS reread the needed attributes only on the final projection, saving an amount of memory on the intermediates results.

Figure 11 – Using B^{+Bt}-tree in Bt-Join



Source: (EVANGELISTA *et al.*, 2015)

Algorithm 10 shows the core of Bt-Join operator. The first loop (lines 4 – 15) performs the *Tree-build phase* scanning the inner relation and, for each tuple, inserting the tuple's join attribute value in the B^{+Bt}-tree. Lines 7 - 10 will be executed only for EMS and lines 11 - 13 will be executed only for MMS. The second phase (lines 16 – 29) proceeds reading each tuple of outer relation to accomplish the *join phase*. The behaviour is almost identical to the first phase, but now a method called insertB^{+Bt}-tree is executed only if the join attribute value is found in the B^{+Bt}-tree, which means that a match happened (line 27).

```
1  /*Input: outerOperand, innerOperand, matStrategy
2    Output: B+Bt-tree*/
3  begin
4   foreach tuple in innerOperand do
5    joinAtts = tuple.getValues(innerOperand.joinAttribs);
6    rowid = tuple.getRowid();
7    if matStrategy = EMS then
8     nextJoinAtts = tuple.getValues(innerOperand.nextJoinAttribs);
9     finalAtts = tuple.getValues(innerOperand.finalAttribs) ;
10   end if
11   if matStrategy = MMS then
12    nextJoinAtts = tuple.getValues(innerOperand.nextJoinAttribs);
13   end if
14   insertB+Bt-tree(btResult, joinAtts, rowid, nextJoinAtts,finalAtts);
15  end foreach
16  foreach tuple in outerOperand do
17   joinAtts = tuple.getValues(outerOperand.joinAttribs);
```

```
18    rowid = tuple.getRowid();
19    if matStrategy = EMS then
20     nextJoinAtts = tuple.getValues(outerOperand.nextJoinAttribs);
21     finalAtts = tuple.getValues(outerOperand.finalAttribs);
22    end if
23    if matStrategy = MMS then
24     nextJoinAtts = tuple.getValues(outerOperand.nextJoinAttribs);
25    end if
26    if find(btResult, joinAtts) != Null then
27     insertB+Bt-tree(btResult, joinAtts, rowid, nextJoinAtts, finalAtts);
28    end if
29   end foreach
30  end
```

Algorithm 10 – Bt-Join – main

Algorithm 11 presents the procedure insertB^{+Bt}-tree is responsible for maintaining the B^{+Bt}-tree. It also creates a new tree or a new node, and inserts the attribute values on the correct existing node (line 9), and still creates and update the rowID List of the attribute values (line 11).

```
1  /*Input: btResult, joinAtts,
2    rowid, nextJoinAtts, finalAtts*/
3  begin
4   if btResult = null then
5    root = build_new_B+Bt-tree;
6   else
7    leaf = find(btResult, joinAtts);
8    if leaf = null then
9     insert_new_node(btResult, joinAtts, rowid, nextJoinAtts, finalAtts);
10   else
11    update_rowidlist(leaf, rowid, nextJoinAtts, finalAtts);
12   end if
13  end if
14 end
```

Algorithm 11 – insertB^{+Bt}-tree

(EVANGELISTA *et al.*, 2015) experiments have shown that for both, EMS and MMS, they could execute the query plan faster than Flash join algorithm and particularly they have shown that MMS is well suited for SSD devices and n-way queries plan which are composed.

5.5 Summary

This chapter presented some SSD aware join algorithms, showing their main characteristics. First, RARE join algorithm takes advantages of the fast random read/write operations on SSD devices. (MEHUL HARIZOPOULOS STAVROS, 2008) This algorithm uses mixture of random reads and sequential I/O allowing to speed up projections and join operations.

Flash join is an algorithm implemented inside the PostgreSql DBMS which is divided into two operators, FlashScan and Flash join. FlashScan is a scan operator whose main functionality is to efficiently read the required attributes to process a given query from the SSD device, in addition, it uses PAX-layout to reduce the amount of data read. Flash join is a multi-way equi-join algorithm, implemented as a pipeline of stylized binary joins. Flash join is composed for two separated pieces, join kernel and fetch kernel. Join kernel is responsible to execute the join algorithm, it leverages FlashScan to fetch only the join attributes needed from base relations. Therefore, fetch kernel implements the FlashScan operator.

Bt-Join uses an extended version of the B^{+}-tree (CORMEN CHARLES E. LEISERSON; STEIN, 1990), called B^{+Bt}-tree, as underlying structure for processing the join algebraic operation. This structure might reduce the memory space required to accomplish a join operation, allowing less writes in secondary memory, extending the lifetime of the SSD device and improving the join performance. As we have seen in section 2.3 a write operation is more expensive than a read operation for this kind of media.

6 DAC SCAN

6.1 Introduction

This chapter introduces a novel scan operator, denoted DaC Scan, for reading database tables stored in SSDs (ALENCAR *et al.*, 2017). The features of DaC Scan and how it works are minutely described through pseudo-codes which represent DaC Scan implementation and a running example.

The chapter is organized as follows: Section 6.2 gives an introduction of characteristics and behaviour of DaC Scan; in Section 6.3 we minutely describe DaC Scan through its algorithms and follow a running example to better explain the DaC Scan operation; and Section 6.4 summarizes this chapter.

6.2 Overview

In Section 2.3, we have shown that, among the four levels of internal parallelism in SSDs, the lowest parallelism level is the plane-level (see Figure 5). In other words, the plane-level parallelism limits the I/O bandwidth in SSDs. On the other hand, SSD FTL stores data of a given file in clustered blocks. Recall that a clustered block contains SSD blocks belonging to different planes (see Figure 5). This way, several SSD blocks of different planes can be accessed simultaneously. Thus, several blocks of a given file F may be stored in a clustered block. Based on those observations, one may assume that case the clustered blocks of a file be concurrently consumed by different processors or threads, the IOPS rates delivered by an SSD device can be significantly improved.

DaC Scan implements a novel technique for computing scan operations on database tables, which take profit of the clustered blocks characteristic. The key goal of the proposed scan operator is to fully exploit the parallelism provided by SSDs. Thus, DaC Scan provides higher read bandwidth than existing scan operators.

Unfortunately, SSD manufacturers do not inform how many planes there are in an SSD chip (die). Accordingly, the number of physical SSD blocks in a clustered block is unknown as well. Nonetheless, DaC Scan implements the concept of jump function J, which tries to infer the amount of physical blocks in a clustered block regardless the SSD device on which DaC Scan is being executed.

The proposed jump function makes DaC Scan able to decompose a full scan operation into several mini-scans, which may be executed in parallel by different processors or threads. Each mini-scan is responsible for reading a specific portion of a table.

The jump function is defined on four variables: (i) the size in bytes of a physical block (*PBS*); (ii) the number of available processors/threads (NT) for executing the scan operator; (iii) the amount of SSD physical pages required to store the table to be scanned (TS), and; (iv) the size in bytes of a database page (*PS*). Thus, the jump function maps values of the four aforementioned variables to a positive integer.

The jump function J is computed by the following expression:

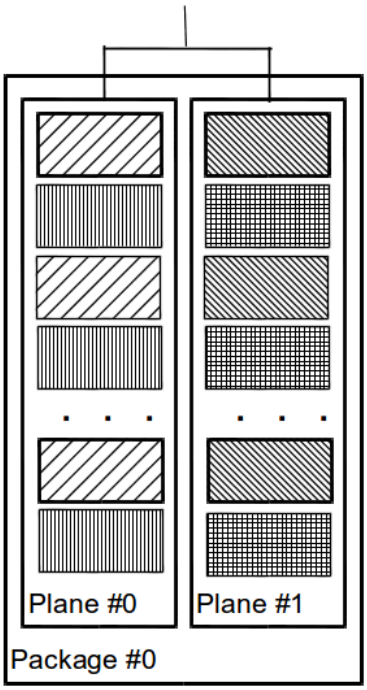
$$J(TS, PS, PBS, NT) = \lfloor \frac{((TS * PS)/PBS)}{NT} \rfloor \tag{6.1}$$

Observe that the jump function J tries to infer the amount of pages of a given table R is stored in an SSD block by means of the term $(TS * PS)/PBS$. By doing this, it is possible to predict the manner pages of R are distributed over SSD blocks for a given combination of TS, PS, PBS and NT. Thus, DaC Scan is able by transitivity to take profit of clustered block abstraction (see Section 2.3) for reading blocks stored in different planes in parallel.

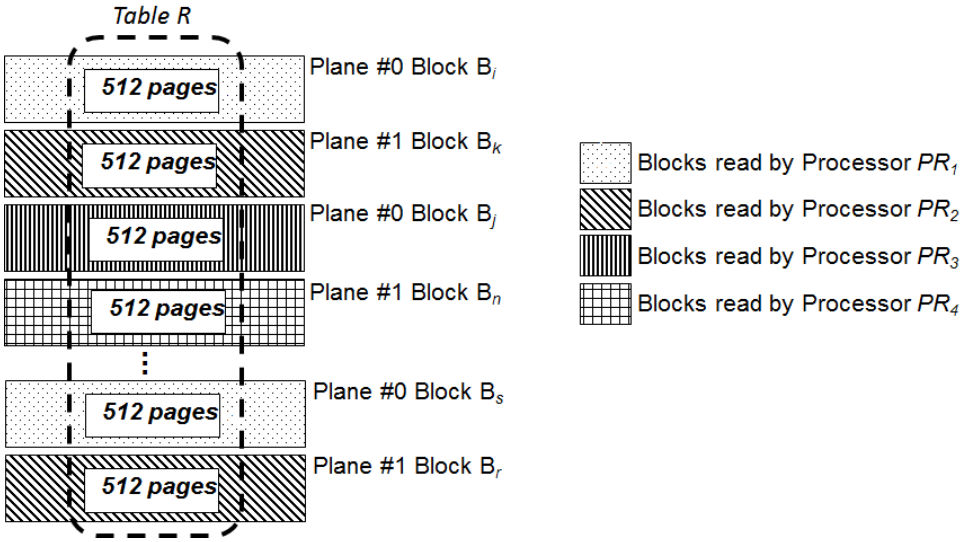
In order to illustrate the way DaC Scan and its jump function work, let us consider the scenarios illustrated in Figures 12a and 12b. Consider that a table R is stored in a flash-based SSD device, which is composed of one chip with two planes (Figure 12a). For the sake of simplicity and without loss of generality, let us assume that a database page (*PS*) and a SSD physical block (*PBS*) have the same size of 8192 bytes. Further, let us consider that the number of the pages necessary to store a table R (TS) and of available processors/threads (NT) are, respectively, 2051 and 4. After applying those values to Equation 6.1, the function J returns 512. Thus, DaC Scan infers that 512 pages of R may be stored per SSD physical block. Recall that the blocks are spread through the existing two planes.

Figure 12b illustrates a logical abstraction of how DaC Scan "views" the distribution of $R's$ pages in several blocks of the two existing planes. Afterwards, DaC Scan starts, in parallel, the execution of 4 mini-scans, each of which is processed by a given processor.

Looking more closely to Figure 12b, one can observe that the first mini-scan PR_1 starts at the block B_i of Plane 0 (block 1) and scans from page 0 to page 511. Mini-scan PR_2, in turn, reads pages 512 to 1024 from block B_k in Plane 1. Mini-scan PR_3 scans pages 1025 to 1537, reading block B_j in Plane 0. Mini-scan PR_4 touches the block B_n in Plane 1 scanning pages



(a) Physical distribution of $R's$ pages.



(b) Logical view of table R.

Figure 12 – DaC Scan: Example of functioning
Source: The author.

1538 to 2050. After mini-scan PR_1 finishes the scan operation on block B_k in Plane 1, it starts to scan another block, for example block B_s in Plan 0, reading pages from 2051 to 2563. The other mini-scans have similar behavior. Therefore, each mini-scan operation consumes different blocks of a given plane and every mini-scan profits from the plane-level parallelism of an SSD.

The computers are usually manufactured with the number of processor based on power of two. Over the last years, the cloud computing paradigm has provided a different view of the number of available processors. This way, one could argue that the jump function would

not work with prime numbers, for example. Nonetheless, in the experiments presented in Section 8.2, simulations have been carried out by using prime numbers for computing the jump function as well.

It is worth to note that DaC Scan does not guarantee that effectively each mini-scan access a different physical plane. However, the results presented in Section 8.2 show that DaC Scan is quite efficient to take adequately profit of the internal parallelism of the SSD devices.

6.3 Algorithm

The algorithm 12 shows the core of DaC Scan. The algorithm can be divided into three parts. First, we initialize variables which will be used during the execution. Second, process initialization and finally, the scan process.

```
1  Input: nThreads, PBS, TS, PS
2  Begin
3  jump = ((TS*PS)/PBS)/nThreads;
4
5  for (int t_id = 1; t_id <= nThreads; t_id++)
6   ProcessScan(nThreads, jump, t_id);
7  End
8
9  Begin.ProcessScan(nThreads, jump, t_id)
10  firstBlock = (t_id * jump) + 1;
11  next = nThreads * jump;
12  block = getBlock(firstBlock);
13  currentBlock = firstBlock;
14  while (block != null) do
15   for (int j = 0; j < jump; j++) do
16    tuple = getNextTuple(block);
17    while (tuple != null) do
18     //do anything
19     tuple = getNextTuple(block);
20    end while
21    currentBlock++;
22    block = getBlock(currentBlock);
23   end
24   currentBlock += next;
25   block = getBlock(currentBlock);
```

```
26  end while
27 End.ProcessScan
```

Algorithm 12 – Pseudo-code for DaC Scan

Line 3 calculates jump function J, then lines (5-7) may start $nThreads$. Each process receives as parameter the number of process $nThreads$, the process number t_id and finally the jump factor $jump$. Lines (10-26) represents the scan process by itself and it is important to note that this process will be executed $nThreads$ times. At line 10, the algorithm calculates the number of the first block which is going to be read by that specific process, the number of the jumper process is calculated at line 11, line 12 reads the block and line 13 sets the current block, finally the loop process to read all tuples designated for that process starts at line 14 and ends at line 26. One can note that at line 18, the algorithm is able to make any kind of process with the tuple that was just read.

6.4 Summary

This chapter presented novel scan operator for storage class memory. In order to fully take profit of the high IOPS rates provided by SSDs, components of database systems should be aware of SSD features. The proposed scan operator, denoted DaC Scan, is able to decompose a scan operation into several mini-scans and to run them in parallel in different processors or threads. This is possible due to the novel concept of jump factor $jump$, whose main goal is to infer the amount of physical blocks in a clustered block.

In Section 3.3, Table 2 has been presented for detailing the main characteristics of existing approaches for implementing parallel scan operators. Now, we present Table 3 in which DaC Scan is introduced.

As already mentioned, in order to read database tables in parallel some scan operators require the use of a controller interface (ECI in Table 3), such as NCQ and AHCI. Nonetheless, such a requirement is not necessary to DaC Scan to run in parallel.

The second feature in Table 3, denoted Hardware-Specific (HS), indicates if the approach depends on some physical characteristic of the SSD device in order to work properly. The third characteristic, indicates whether or not the parallel scan operator requires user intervention (UI) to be turned on, e.g., by means of an SQL expression. DaC Scan does not require any

Approach	MS SQL SERVER	DB2	Oracle	DaC-Scan
ECI	No	Yes	No	**No**
HS	No	No	No	**No**
UI	Yes	No	Yes	**No**
MPA	No	Yes	No	**No**
MI	No	No	No	**No**

Table 3 – DaC Scan vs Other Approaches

human intervention.

In turn, multiple processor availability (MPA) is required by some approaches for implementing parallel scan operation. For DaC Scan, such a feature is not mandatory. Finally, regarding the Manufacture Information (MI) characteristic, there are parallel scan operators, which require knowing in advance manufacture industry information (e.g., the number of available channels in the SSD device) in order to work properly. Nonetheless, DaC Scan is quite efficient even without such information.

To conclude this section, it is important to emphasize that DaC Scan can be used on a partitioned table as well. The experiments presented in 8.2 have been performed considering the worst case, i.e., on a non-partitioned table. With a partitioned table, DaC Scan behavior is the same, calibrated by the number of partitions. In other words, with partitioned tables DaC Scan delivers better I/O bandwidth than with non-partitioned table. Moreover, DaC Scan is not dependent on key-value store mechanism, such as presented in (NGUYEN; MINH, 2015).

7 DAC JOIN

This chapter introduces a novel join operator, denoted DaC Join, the main goal of the proposed operator is to properly take advantage of on-chip parallelism and high random IOPS rates delivered by many-core machines with SSDs as secondary memory (ALENCAR *et al.*, 2019). The features of DaC Join and how it works are minutely described through pseudo-codes which represent DaC Join implementation and a running example.

The chapter is organized as follows: Section 7.2 gives an introduction of characteristics and behaviour of DaC Join; in Section 7.3 we minutely describe DaC Join through its algorithms and follow a running example to better explain the DaC Join operation; and Section 7.4 summarizes this chapter.

7.1 Introduction

DaC Join is a multi-way physical join operator, implemented as a pipeline of $n-1$ binary joins. The main features delivered by DaC Join are:

- Parallel execution on several cores. DaC Join is able to run on multiple cores (or threads) in parallel, increasing this way intra-operator parallelism.
- Use of dynamic hash functions. The proposed operator implements two dynamic hash functions. The functions are defined based on the number of threads/processors for making DaC Join able to scale as the number of cores increases.
- Reduction of the amount of write operations on secondary memory. DaC Join uses as much main memory as possible. However, in case there is not enough main memory for joining very large tables, DaC Join continues its execution using secondary memory. In this case, DaC Join effectively decreases the activity of writing back join temporary results on SSDs.

Additionally, a novel SSD-aware cost model has been defined for estimating execution cost of join operations. The proposed model captures the premise that write operations on SSDs are much more expensive than read operations. In this sense, DaC Join may be executed in main memory (the best case) and, case there is not enough main memory, it is able to use SSDs as secondary memory.

7.2 Overview

The join operator is by definition a binary operator. DaC Join is an n-way physical join operator, implemented as a pipeline of $n-1$ 2-way joins. Thus, DaC Join is able to yield its output tuples as early as possible. DaC Join is composed of two phases, Scan phase and Join phase.

In order to illustrate all DaC Join's phases, consider the join operation $R \bowtie S$, depicted in Figure 13. As we have seen in 6.2, the Scan phase divides the action of reading the join operand R into several scan operations, each of which is responsible for reading a portion of R. Each scan operation runs on a specific core C and injects data in a dedicated pipe for C. Thus, the several scan operations consume data from the join operand – enabling to run on several cores – in parallel. Such a feature makes DaC Scan able to take full advantages of *on-chip parallelism* and *high IOPS rates* delivered by SSDs. In the scenario illustrated in Figure 13, there are k available cores to execute DaC Scan. Thus, the Scan phase can consume data from each join operand (R and S) by means of k scan operations, which inject data into k pipes.

In order to optimize the use of available main memory, DaC Join implements a materialization strategy, called MMS, in which join attributes are materialized as early as possible and final projection attributes are materialized as late as possible (EVANGELISTA *et al.*, 2015).(See section 3.4.3) Thus, whenever the Scan phase reads data from a table R, it injects into the pipes tuples of the type $<RowID_R, JoinAttr_R,>$, where $RowID_R$ and $JoinAttr_R$ denote the tuple identifier and the join attribute from R. Similarly, sub-hash-tables from R built during the Join phase contains a set of $<RowID_R, JoinAttr_R>$ tuples. On the other hand, the output of a DaC Join phase is a set of $<RowID_R, JoinAttr_R, RowID_S, JoinAttr_S>$ tuples. In the case of projection attributes, DaC Join only reads them on the secondary memory at the moment DaC Scan has to yield the final join result.

The Join phase is executed by two steps. The first step builds a two level hash-table hierarchy. The first level of that hierarchy is composed of super-hash-tables. In turn, each super-hash-table contains several sub-hash-tables (see Figure 13). Two different hash functions are employed to build the hash-table hierarchy (h_1 and h_2 in Figure 13). It is important to note that both functions are dynamic hash functions, since they are defined based on the number of cores available to process DaC Join. The functions are detailed in the next section.

The construction of the hash-table hierarchy is performed in parallel with the Scan phase. Therefore, as soon as a tuple t which belongs to table R is read (by the Scan phase), it is

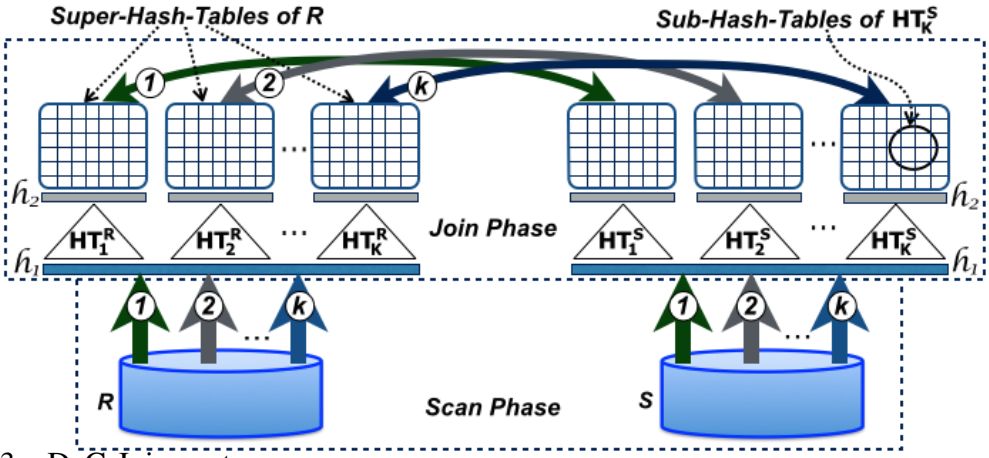


Figure 13 – DaC-Join anatomy.

hashed by the first hash function (h_1 in Figure 13) into a super-hash-table (HT_i^R in Figure 13). Thereafter, t is hashed again into sub-hash-tables $ht_{i_n}^R$ by means of the second hash function (h_2). Note that $ht_{i_n}^R \subseteq HT_i^R$ (Figure 13). Finally, after all tuples from both join operands have been hashed, the second join step, called probe step, is triggered. During the probe step, tuples of R allocated to sub-hash-table $ht_{i_n}^R$ are compared with tuples hashed to its S counterpart $ht_{i_n}^S$.

Next, each phase of the proposed join operator is described in more details.

7.2.1 Scan Phase

As we have seen in 6.2 Scan phase is responsible for consuming data from join operands by running several scan operations in parallel. Each scan operation is responsible to inject read data into a given pipe. The number of scan operations and pipes in an n-way join is determined by the number of available cores. To illustrate the behaviour of the Scan phase, consider the join operation $R \bowtie S$ depicted in Figure 13. Furthermore, let P_R and P_S be the number of SSD pages required to store tables R and S, respectively. DaC Join starts to scan the smallest join operand w.r.t. the number of pages. Let S be the smallest table, i.e., $P_S < P_R$. Thus, DaC Join first reads S. After S is completely read, DaC Join scans R. By doing this, DaC Join tries to minimize the probability of occurring main memory overflow, reducing, this way, the number of write operations on secondary memory.

Due to MMS used in DaC Join (see Section 3.4.3), whenever DaC Scan is reading data from a table R, it injects only the row ID and the join attribute from each tuple (i.e., $<RowID_R, JoinAttr_R,>$) into the pipes. Case the Scan phase is consuming data produced by a previous join operation in an n-way join (see Figure 14), it forwards tuples with the following

attributes: $<RowID_R, JoinAttr_R, RowID_S, JoinAttr_S>$.

7.2.2 Join Phase

As already mentioned, the key goals of the proposed mechanism are: (i) to reduce the number of write operations on the secondary storage media, and; (ii) to increase intra-operator parallelism. In order to achieve such goals, the Join phase initially constructs a hash-table hierarchy for each operand. Thus, to compute $R \bowtie S$, the Join phase builds a hash-table hierarchy for R and another for S. The first level of the hierarchy is composed of several super-hash-tables (see Figure 13). In turn, each super-hash-table is composed of several sub-hash-tables (the second level of the hierarchy). The hash function to construct the super-hash-tables is the following:

$$h_1(JoinAttr) = (JoinAttr \ mod \ k) + 1 \tag{7.1}$$

In Equation 7.1, k represents the number of available cores and $JoinAttr$ represents the join condition attribute.

For building sub-hash-tables, the following hash function is applied on the attribute of the join condition:

$$h_2(JoinAttr) = JoinAttr \ mod \ \left\lfloor \frac{(AvailableMem/PagSize)}{k} \right\rfloor \tag{7.2}$$

The factors $AvailableMem$ and $PagSize$ represents the amount of available main memory to execute DaC Join and the database page size, respectively. It is important to note that both hash functions adapt themselves to the number of available cores, i.e., k, which means they are dynamic hash functions.

The intuition of applying function h_1 is to explore the use of all available cores. For that reason, the amount of super-hash-tables built by h_1 is equal to the number of cores.

In turn, function h_2 is responsible for optimizing the use of available main memory. Beside employing h_2, the Join phase minimizes main memory usage by applying the MMS explained in Section 3.4.3. Accordingly, tuples belonging to the output of the Join phase has the following structure: $<RowID_R, JoinAttr_R, RowID_S, JoinAttr_S>$.

One may claim that the functions do not work well whenever k is prime number. Nonetheless, we have carried out some experiments, in which k is prime number. The results presented in Section 8.3 show that the functions are in fact efficient even in those scenarios. Additionally, it is not quite often to have machines with a prime number of cores or processors.

DaC Join initially builds the hash-table hierarchy in main memory. However, the size of main memory may be often smaller than, actually, the necessary memory area to build the hash-table hierarchy of very large tables. Accordingly, memory overflow may occur during DaC Join execution. To overcome that problem, the Join phase triggers a memory cleaning process, whenever memory overflow events occur. Obviously, the memory cleaning is a write-intensive process and DaC Join should reduce somewhat write activity. The criterion to free main memory space depends on the execution order of the join operand during the Scan phase. Consider that, for computing $R \bowtie S$, DaC Join identifies that $P_R < P_S$ (R is smaller than S). Recall that in this case R is the first operand to be scanned and S the second one (see Section 7.2.1). Thus, there are two different procedures, which may be executed by the memory cleaning process, depending on the moment the memory overflow event happens:

Case 1. The memory overflow occurs during the hash-table building step of R (first operand processed by Scan phase). In this case, the memory cleaning process flushes to SSD the sub-hash-table, in which the Join phase was attempting to allocate a tuple, when the memory overflow occurred. Nonetheless, if the chosen sub-hash-table is empty, the memory cleaning process randomly chooses another sub-hash-table from the same super-hash-table. For instance, suppose that DaC Join tries to allocate an entry into sub-hash-table $ht^R_{i_n}$, where $ht^R_{i_n} \subseteq HT^R_i$ (Figure 13). The memory cleaning process has to flush $ht^R_{i_n}$ to SSD. However, if $ht^R_{i_n}$ is empty, the memory cleaning process picks another non-empty sub-hash-table $ht^R_{i_k} \subseteq HT^R_i$.

Case 2. The memory overflow arises when the hash-table hierarchy for S has been building. The memory cleaning process flushes to SSD any sub-hash-table $ht^R_{i_n} \subseteq HT^R_i$, where HT^R_i is the corresponding super-hash-table of R to the super-hash-table of S which was being manipulated when the memory overflow happened. For instance, suppose that DaC Join tries to allocate an entry into sub-hash-table $ht^S_{i_n}$, where $ht^S_{i_n} \subseteq HT^S_i$ (Figure 13). The memory cleaning process has to flush any $ht^R_{i_j} \subseteq HT^R_i$ to SSD.

The latter criterion (Case 2) reduces write activity on SSD, since it chooses a sub-hash-table for which the building step has already finished. In other words, the chosen sub-hash-table has all possible tuples which can be allocated to it. Thus, that sub-hash-table is flushed to SSD just once. It is also important to mention that a memory overflow event occurs when a given processor/thread i is trying to allocate a tuple into a sub-hash-table of super-hash-table HT^R_i. Thus, only the processor/thread i is responsible for executing the memory

cleaning process, while the others $k-1$ processors/threads continue their job, i.e., building the hash-table hierarchy.

After all tuples from both join operands have been hashed to the hash-table hierarchy, the probe step is initiated. Once more, the k available processors/threads are utilized. Recall that for each join operand the function h_1 hashes tuples to k super-hash-tables (see Equation 7.1). Thus, there is a dedicated processor for probing tuples of each pair of super-hash-tables HT_i^R and HT_i^S, where $1 \leq i \leq k$ and R and S are the join operands. The process of probing super-hash-tables HT_i^R and HT_i^S implements a divide and conquer algorithm as well. It works as follows: tuples belonging to the sub-hash-table $ht_{i_n}^R$ are exclusively compared with tuples belonging to $ht_{i_n}^S$. As early as tuples which satisfy the join condition are identified, the Join phase forwards them to the next query operator or to next Scan phase in the case of computing an n-way join.

During the execution of hash-table hierarchy building step, sub-hash-tables may be flushed to SSD. Thus, to ensure completeness of the join result produced by DaC Join, as soon as all sub-hash-tables have been probed, DaC Join starts to compare sub-hash-tables from one operand on SSD with sub-hash-tables to the other operand on memory and SSD.

7.2.3 DaC-Join Engine Through a Magnifying Glass

In order to investigate the manner DaC Join works, let us consider the scenario illustrated in Figure 14. In that scenario, DaC Join should compute a 3-way join, $R \bowtie S \bowtie T$. The query optimizer is responsible for efficiently decomposing the 3-way join into two 2-way joins. Let us assume that the optimizer has defined the following join execution order $(R \bowtie S) \bowtie T$. Thus, the Scan phase begins to scanning tables R and S. Recall that Scan phase divides the scan action into several scan operations, based on the following rule: pages $p, p+k, p+2k, \ldots$ from R (or S) are read by a scan operator running on a core p and injected into pipe p. For example, assuming $k = 4$, pages 2, 6, 10, 14 and so on belonging to S are read by a scan operator running on core 2 and are injected into pipe 2.

The hash-table building step and the Scan phase are executed in parallel. Thus, as soon as a tuple t comes in to the Join phase, the hash function h_1 is applied to the join attribute and the result represents the address of the super-hash-table t should be allocated. Nonetheless, tuples are physically allocated in sub-hash-tables. Consequently, h_2 is applied to the join attribute in order to define in which sub-hash-table $ht_{i_n}^R$, where $ht_{i_n}^R \subseteq HT_i^R$, t should be inserted. After

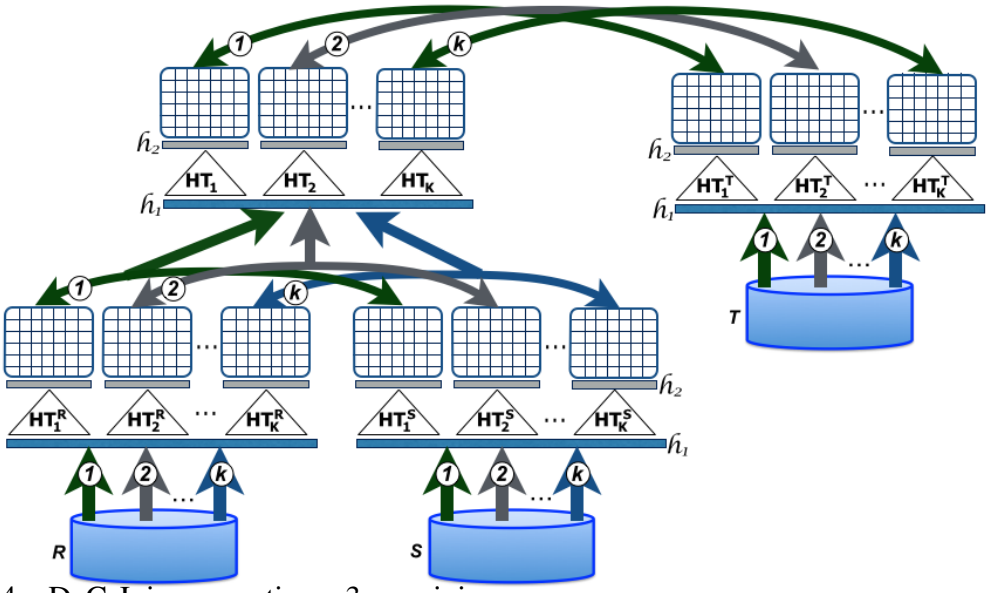


Figure 14 – DaC-Join computing a 3-way join.

all tuples from both join operands have been hashed to the hash-table hierarchy, the probe step is triggered. During the probe step, tuples belonging to the sub-hash-table $ht_{i_n}^R$ are exclusively compared with tuples belonging to its counterpart $ht_{i_n}^S$.

7.2.4 Cost Model

DaC Join has been designed to be able to minimize the number of write operations SSDs. Having this in mind, we defined a cost model for DaC Join, which takes into account the number of read and write operations for a given amount of available main memory for executing SSD. Nevertheless, differently from HDDs, write operations on SSDs are much more expensive than read operations. Thus, it is reasonable to introduce an asymmetry factor for capturing the SSD feature that write operations are much more expensive than read operations. Let RT and WT be the number of read and write operations executed on an SSD per second, respectively. Thus, we define the asymmetry factor as

$$\omega = \lceil RT/WT \rceil \tag{7.3}$$

Whenever a main memory overflow occurs during the build step, the memory cleaning process flushes sub-hash tables to SSD (see Section 7.2.2), i.e., write operations are executed on SSD. In order to find the amount of write operations executed by DaC Join, let P_R and P_S be the number of SSD pages required to store tables R and S, respectively. The cardinality of R and S is represented by $\|R\|$ and $\|S\|$. The amount in pages in main memory available

to perform $R \bowtie S$ is denoted as P_M. Page factor f_M specifies the number of tuples $<RowID_R$, $JoinAttr_R$, $RowID_S$, $JoinAttr_S\>$; that fit in a page in main memory. Thus, the number of write operations on SSD can be estimated by applying the following formula:

$$|P_M - \frac{(\|R\| + \|S\|)}{f_M}| \tag{7.4}$$

Nevertheless, the total estimated cost for executing DaC Join should capture the asymmetry factor defined in Equation 7.3. For that reason, the expression in Equation 7.4 has to be multiplied by ω, which gives the following factor to be considered in total estimated cost:

$$\omega \cdot |(P_M - \frac{(\|R\| + \|S\|)}{f_M})| \tag{7.5}$$

For computing the amount of read operations, one should take into account the number of operations to read the join operands and to read sub-hash-tables flushed to disk due to memory overflow events. Therefore, the number of read operations can be estimated by the following formula:

$$P_R + P_S + |(P_M - \frac{(\|R\| + \|S\|)}{f_M})| \tag{7.6}$$

Therefore, to specify the total estimated cost for DaC Join, one sums the factors expressed in Equations 7.5 and 7.6, which gives:

$$P_R + P_S + (1+\omega) \cdot |(P_M - \frac{(\|R\| + \|S\|)}{f_M})| \tag{7.7}$$

7.3 Algorithm

The algorithm 13 represents the main class of DaC Join. This class is responsible for execute those two phases which are part of DaC Join presented in 7.2. Line 3 calculates jump function J, which will be used on the Scan phase, then lines (5-7) may start $nThreads$. Each process started receives as parameter the number of process $nThreads$, the process number t_id and finally the jump factor $jump$. By this way the algorithm will finish the first phase. The second loop at Lines (9-11) is responsible to start the second phase of the algorithm, called join phase. Join phase as Scan phase will start $nThreads$ and execute the join by itself in a pipeline.

```
1 Input: nThreads, PBS, TS, PS
2 Begin
3 jump = ((TS*PS)/PBS)/nThreads;
4 for (int t_id = 1; t_id <= nThreads; t_id++)
```

```
5  ProcessScan(nThreads, jump, t_id, tableIdT1);
6  ProcessScan(nThreads, jump, t_id, tableIdT2);
7 End
8
9 for (int t_id = 1; t_id <= nThreads; t_id++)
10  ProcessJoin(t_id, tableIdT1, tableIdT2);
11 End
```

Algorithm 13 – Pseudo-code for DaC Join Main Program

Algorithm 14 have already been presented in section 6.3, it details the scan process. The difference between them is on line 12, where it executes a function called, *InsertInto-DaCHashTables*. This function receives three variables as parameters. First, the tuple, second, the id of the process and finally, the table's id where the tuple came from.

```
1  Input: nThreads, jump, t_id
2  Begin
3  Begin.ProcessScan(nThreads, jump, t_id, tableId)
4   firstBlock = (t_id * jump) + 1;
5   next = nThreads * jump;
6   block = getBlock(firstBlock);
7   currentBlock = firstBlock;
8   while (block != null) do
9    for (int j = 0; j < jump; j++) do
10    tuple = getNextTuple(block);
11    while (tuple != null) do
12     InsertIntoDaCHashTables(tuple, t_id, tableId);
13     tuple = getNextTuple(block);
14    end while
15    currentBlock++;
16    block = getBlock(currentBlock);
17   end
18   currentBlock += next;
19   block = getBlock(currentBlock);
20  end while
21 End.ProcessScan
```

Algorithm 14 – Pseudo-code for DaC Join - Scan phase

Important functions for DaC Join are described on algorithm 15. First, the function presented at lines 3-11 is responsible for build the hash tables that will be used in the next phase. Line 4 calculates a hash for the super-hash-tables and line 5 calculates a hash for sub-hash-tables. Next, line 6 check whether is there enough remained available memory to keep the tuple on it's sub-hash-table at line 7. Line 9 will be executed to release memory, by doing this the algorithm is capable to keep a new tuple in memory, i.e., it calls the same function at line 10 with the intention of keep that tuple in a new sub-hash-table.

```
1  Input: tuple, nThreads, pageSize, tableId
2
3  Begin.InsertIntoDaCHashTables(tuple, nThreads, tableId)
4   h1 = CalcH1(JoinAtt(tuple), nThreads);
5   h2 = CalcH2(JoinAtt(tuple), nThreads);
6   if(RemainAvailableMem())
7    HashTableMemList.add(tableId, joinAtt, nextJoinAtts(tuple), rowid(tuple
       ), h1, h2);
8   else
9    ReleaseMemory();
10   InsertIntoDaCHashTables(tuple, nThreads, tableId);
11 End.InsertIntoDaCHashTables
12
13 Begin.CalcH1(joinAtt, nThreads)
14  return (joinAtt % nThreads) + 1;
15 End.CalcH1
16
17 Begin.CalcH2(joinAtt, nThreads)
18  return joinAttr % ((TotalAvailableMem() / pageSize) / nThreads)
19 End.CalcH1
20
21 Begin.ReleaseMemory()
22  WriteOnDisk(BucketMemList.getRandom());
23 End.ReleaseMemory
```

Algorithm 15 – Pseudo-code for DaC Join - Build buckets

The join phase is detailed on algorithm 16. Once the tables were divided in *n* super-hash-tables, each process will join only those sub-hash-tables that belongs to it. Lines 4-5 put into a variable the path of the super-hash-tables that represents their *threadId*, then lines 7-8 get

the map of the super-hash-table that were written on disk and then, lines 10-11 get for each table their own first sub-hash-table.

Next, on lines 13-48, there are four different loops. First, lines 14-21, may join tuples that for both tables were kept in main memory. The second loop, will join tuples that from *table 1* were kept in main memory and from *table 2* were kept in disk. Next loop, third, may join tuples that from *table 2* were kept in main memory and from *table 1* were kept in disk. Finally, fourth loop, will join tuples that for both tables were kept in disk.

```
1  Input: t_id, tableIdT1, tableIdT2, nThreads
2
3  Begin.ProcessJoin(t_id, tableIdT1, tableIdT2)
4   HashTableMemListByProcessT1 = HashTableMemList.getSuperHashTable(t_id);
5   HashTableMemListByProcessT2 = HashTableMemList.getSuperHashTable(t_id);
6
7   SubHashTableT1 = HashTableMemListByProcessT1.getNextSubHashTable();
8   SubHashTableT2 = HashTableMemListByProcessT2.getNextSubHashTable();
9
10  SubHashTableFromDiskT1 = MapHashTableDiskListByProcessT1.
      getNextSubHashTable(t_id);
11  SubHashTableFromDiskT2 = MapHashTableDiskListByProcessT2.
      getNextSubHashTable(t_id);
12
13  /*Memory x Memory*/
14  for each tuple on SubHashTableT1 do
15   for each tuple on SubHashTableT2 do
16    if(Match(AttJoinT1, AttJoinT2))
17     InsertResut(bTupleT1, bTupleT2);
18    SubHashTableT2 = HashTableMemListByProcessT2.getNextSubHashTable();
19   End
20   SubHashTableT1 = HashTableMemListByProcessT1.getNextSubHashTable();
21  End
22
23  /*MemoryT1 x DiskT2*/
24  for each tuple on SubHashTableT1 do
25   for each tuple on SubHashTableFromDiskT2(SubHashTableT1) do
26    if(Match(AttJoinT1, AttJoinT2))
27     InsertResut(bTupleT1, bTupleT2);
28   End
```

```
29    SubHashTableT1 = HashTableMemListByProcessT1.getNextSubHashTable();
30   End
31
32   /*MemoryT2 x DiskT1*/
33   for each tuple on SubHashTableT2 do
34    for each tuple on SubHashTableFromDiskT1(SubHashTableT2) do
35     if(Match(AttJoinT1, AttJoinT2))
36      InsertResut(bTupleT1, bTupleT2);
37    End
38    SubHashTableT2 = HashTableMemListByProcessT2.getNextSubHashTable();
39   End
40
41   /*Disk x Disk*/
42   for each tuple on SubHashTableFromDiskT1 do
43    for each tuple on SubHashTableFromDiskT2(SubHashTableFromDiskT1) do
44     if(Match(AttJoinT1, AttJoinT2))
45      InsertResut(bTupleT1, bTupleT2);
46    End
47    SubHashTableFromDiskT1 = MapHashTableDiskListByProcessT1.
       getNextSubHashTable();
48   End
49  End.ProcessJoin
50
51  Begin.InsertResut(bTupleT1, bTupleT2)
52   if(FinalResult())
53    ReRead(rowidT1, rowidT2)
54   else
55    InsertIntoDaCHashTable(BuildIntermediateTuple(bTupleT1, bTupleT2),
       nThreads, tableId)
56  End.InsertResut
```

Algorithm 16 – Pseudo-code for DaC Join - Join phase

It is important to note, on lines 51-56, a function called *InsertResult*. This function, on line 53, is responsible to re-read those tuples that may be part of the final result and on line 55 it builds an intermediate table which will be used for the next join.

7.4 Summary

This chapter presented novel hash based join operator for storage class memory. DaC Join is divided in two steps. First, scan phase may take profit of the high IOPS rates provided by SSDs, once it runs several scan operations in parallel. Second, join phase also take advantage of the SSD characteristics, it may run several operations in parallel building intermediate table or final results as fast as possible. The materialization strategy used on DaC Join is MMS which as minuted discussed on section 3.4.3. This strategy is well suited for storage class memory, once it may provide less writes and a balanced number of re-reads to build intermediate table in a n-way join.

It is important to emphasize that DaC Join as DaC Scan can also be used on a partitioned table as well. The experiments presented in Section 8.3 have been performed considering the worst case, i.e., on a non-partitioned table. We also provide a cost model on Section 7.2.4, which takes into account the number of read and write operations for a given amount of available main memory.

8 EMPIRICAL EVALUATION

8.1 Introduction

In order to assess the potentials of the proposed scan operator DaC Scan, experiments on the TPC-H database and the main results achieved so far are presented and discussed in this section. In what follows, section 8.2 provide information on how the experiment environment was set up. Thereafter, on Section 8.2 results are quantitatively presented and qualitatively discussed.

The join algorithm, DaC Join, proposed in this research have been empirically evaluated and compared against our implementations of Flash join (TSIROGIANNIS *et al.*, 2009) and Hybrid-hash join algorithms. Flash join has been chosen because it is a well-known join operator proposed to be deployed in SSDs. The main results achieved so far are presented and discussed in Section 8.3. In this sense, we first provide information on how the experimentation environment was set up. Thereafter, as we have done with DaC Scan, empirical results of DaC Join's effectiveness and efficiency are discussed in Sections 8.3.1 and 8.3.2 respectively. Finally, Section 8.4 concludes this chapter.

8.2 DaC Scan

A DaC Scan prototype has been written in Java on a row storage engine. The used storage engine provides methods for reading and writing 8k pages. The buffering functionality has been disabled for not impacting the results. By doing this, each data access (read/write) corresponds to direct access on the SSD device. In other words, there is no possibility to find the required page on the storage-engine's buffer area (located in main memory). The experiments have been conducted on the data of the TPC-H table *lineitem*, with scale factor of 10 and 20, whose main features are presented in Table 4.

Table name	Scale Factor	Rows	Kylobytes
Lineitem	10	59,986,052	8,428,864
Lineitem	20	119,972,104	16,860,664

Table 4 – TPC-H Lineitem table.

All experiments reported here have been performed on a server with an 6-Core Intel Xeon E5 (3.7GHz) with hyper-threading and 32GB 1866MHz of RAM, running OS X Yosemite.

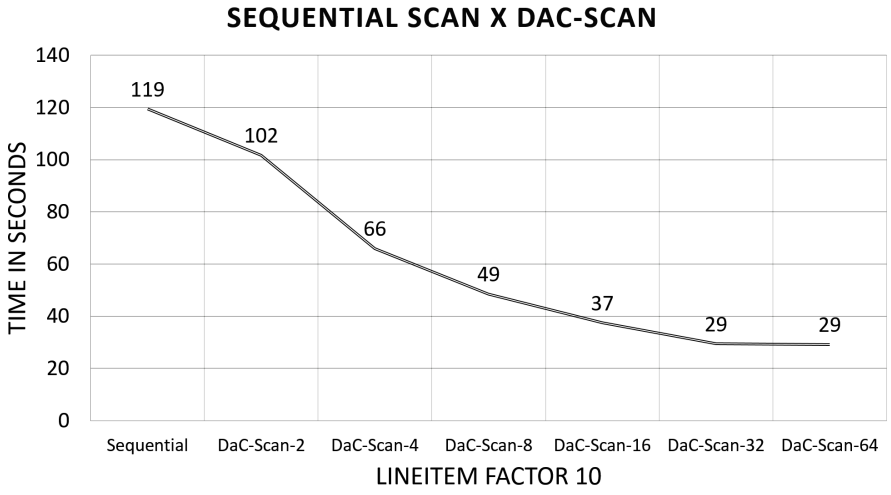
The used SSD device was a 256GB PCIe-based Apple SM0256F. To remove buffer influence in experiment results, a cold start strategy has been used for each experiment. In other words, the file system cache of machine's operating system has been cleaned before running a given experiment. This strategy eliminates any influence of the underlying file system manager and of its interface with the SSD device's FTL. The results presented in this research have been collect as follows. We run each experiment ten times. From the ten obtained values, the maximum and the minimum values have been removed. Thereafter, the average of the remaining values has been computed.

DaC Scan is able to decompose a scan operation into several mini-scans and to run them in parallel in different processors or threads. For that reason, during the experiments six different scenarios have been considered: DaC Scan running on 2 (DaC-Scan-2), 4 (DaC-Scan-4), 8 (DaC-Scan-8), 16 (DaC-Scan-16), 32 (DaC-Scan-32) and 64 (DaC-Scan-64) threads. In this research we analyze the results considering a page size (the *PBS* variable of the jump function) of 8MB.

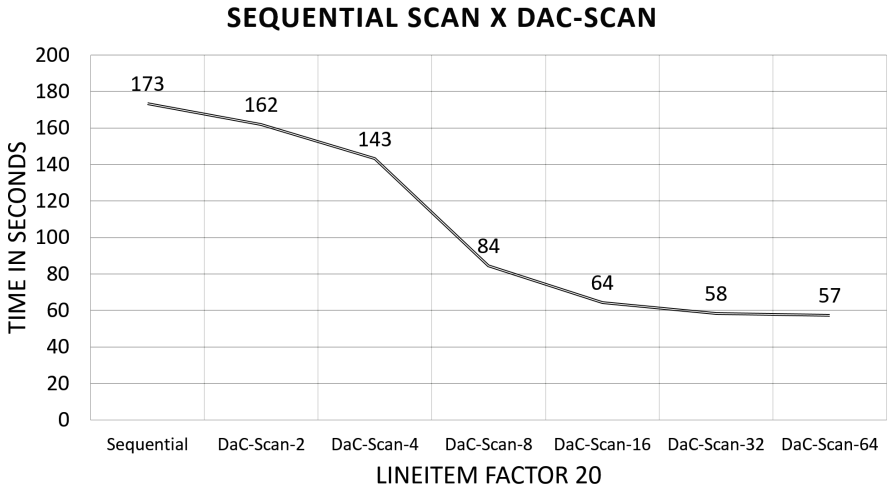
8.2.1 Experimental Results

The results presented in Figures 15a and 15b compares the performance of a sequential scan operation with a scan operation applying DaC Scan without taking profit of using SSD internal parallelism. For that reason, the jump function in both figures is 1. The goal of such experiments is to show that, even without considering SSD internal parallelism, the proposed scan operator delivers better performance than the classical sequential scan operator. Figure 15a depicts the results of running sequential scan and DaC Scan on TPC-H table $lineitem$ with scale factor 10. In turn, Figure 15b illustrates the results of the same experiment on $lineitem$ with scale factor 20.

Figures 15a and 15b show that for all scenarios DaC Scan is faster than a sequential scan. DaC-Scan-2 is 17.4% faster than sequential scan for scale factor 10 and 7.07% faster for scale factor 20. DaC-Scan-4 is 54.2% faster than DaC-Scan-2 for scale factor 10 (Figure 15a) and 13.14% faster for scale factor 20 (Figure 15b). DaC-Scan-8 is 53.8% faster than DaC-Scan-4 for scale factor 10 and 69.44% faster for scale factor 20. DaC-Scan-16 is 29.5% faster than DaC-Scan-8 for scale factor 10 and 31.41% faster for scale factor 20. DaC-Scan-32 is 27.2% faster than DaC-Scan-16 for scale factor 10 and 10.27% faster for scale factor 20. Finally, DaC-Scan-64 is only 0.82% faster than DaC-Scan-32 for scale factor 10 and only 1.5% faster for



(a) Scale Factor 10 - $Jumpfunction = 1$



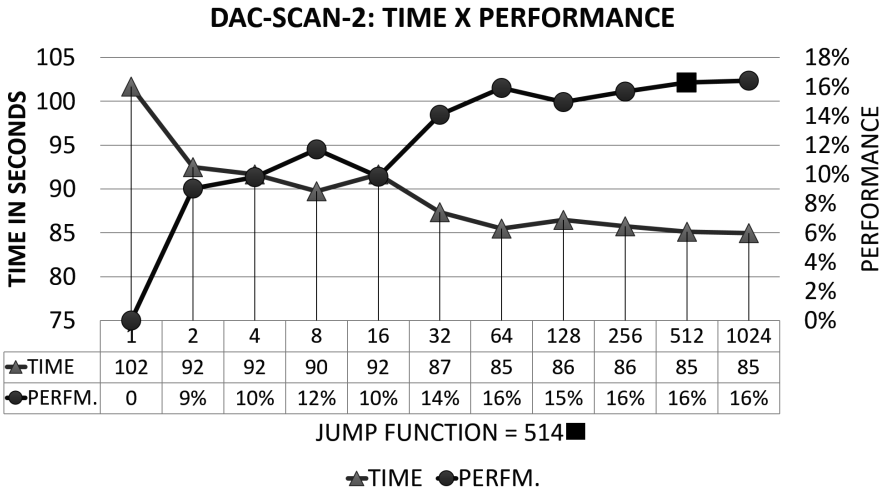
(b) Scale Factor 20 - $Jumpfunction = 1$

Figure 15 – Sequential and DaC Scan

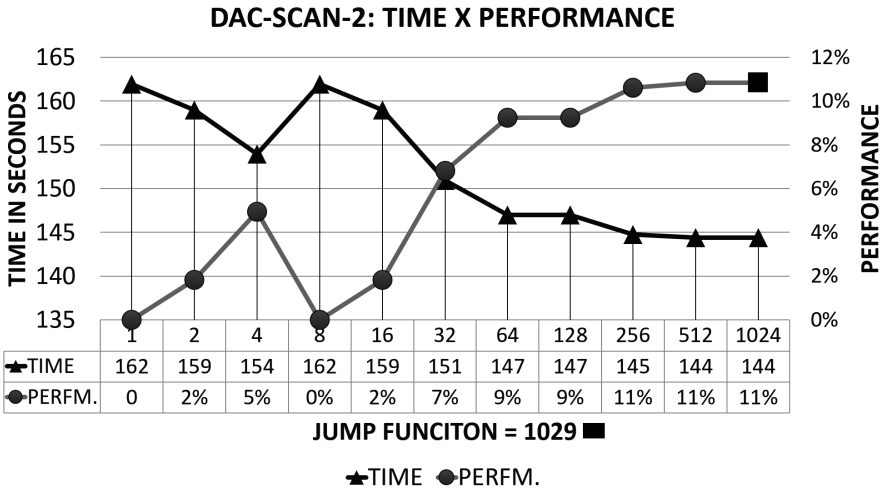
scale factor 20, this small variation between 32 and 64 threads shows that it is highly unlikely to significantly enhance performance with more than 32 threads (or processors).

The next figures (from Figure 16 to Figure 21) bring the results of running DaC Scan with different amounts of threads, namely, 2, 4, 8, 16, 32 and 64. Furthermore, to show the jump function efficiency, we have investigated the behavior of DaC Scan for the computed value of the jump function (see Equation 6.1 in Section 6.2) and other empirical "jump" values. Such values are depicted on x-axis of Figures from 16 to 21. In those figures, value equal to 1 on x-axis means the use of classical round-robin strategy for reading the table in parallel. In this case, thread n reads a block which is neighbour to blocks read by threads $n-1$ and $n+1$. Recall that a jump value infers the number of pages in an SSD block.

Two metrics have been employed for plotting the next figures: *r*esponse time and



(a) Scale Factor 10 - *Jumpfunction* = 514



(b) Scale Factor 20 - *Jumpfunction* = 1029

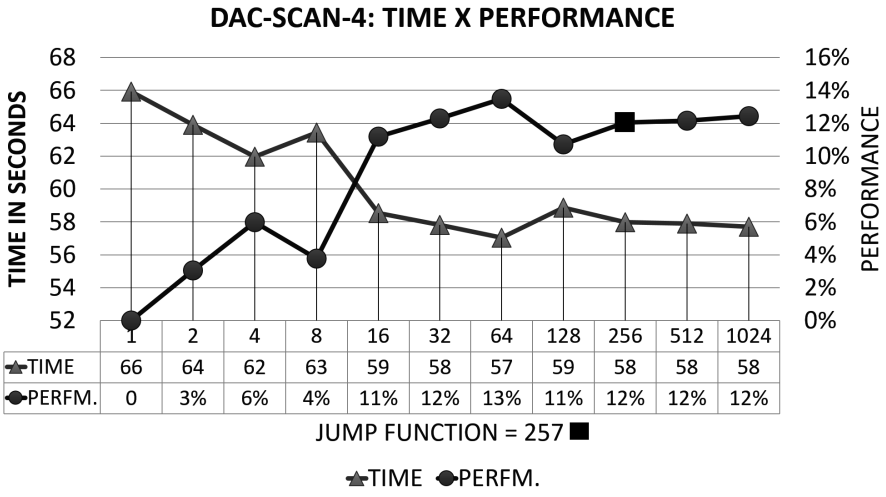
Figure 16 – DaC Scan - 2

*p*erformance gain. The response time metric represents the amount of time consumed DaC Scan to compute a scan operation on table *lineitem*, scale factors 10 and 20 (see Table 4). In turn, the performance gain metric captures time decrease ratio for running DaC Scan operator with different "jump" values (depicted on x-axis of Figure 16 to 21) compared to the time necessary for executing the classical round-robin strategy, i.e., a jump value equal to 1.

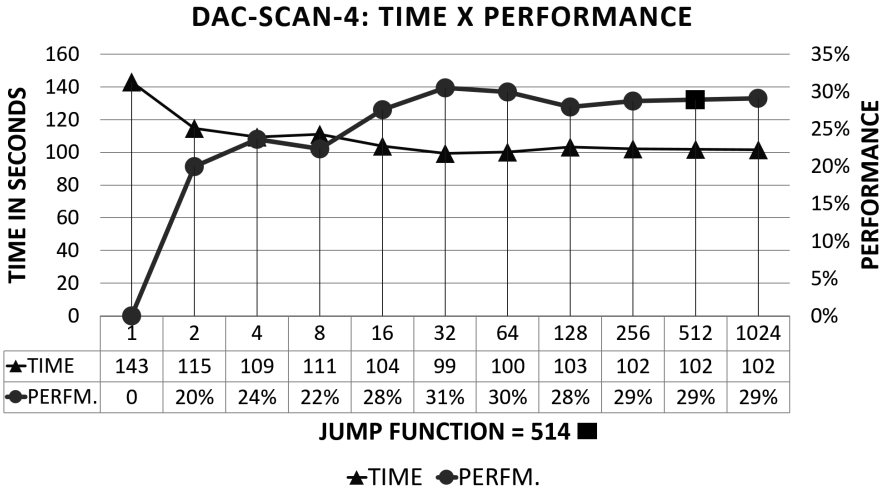
Figure 16a and Figure 16b bring the performance of executing the proposed scan operator with two threads (for short, DaC-Scan-2) for scale factor of 10 and 20, respectively. For two threads, the proposed *J* function returns 514 for *lineitem* with scale factor 10 (Figure 16a) and 1029 for *lineitem* with scale factor 20 (Figure 16b). Thus, one can note that the value computed by the jump function is 16% faster than the value 1 for scale factor of 10 and 11% faster than the value 1 for scale factor of 20. Therefore, in both cases, the use of the jump

function brought the best result for DaC-Scan-2.

Figures 17a and 17b depict the DaC Scan behavior when there are four available threads (DaC-Scan-4). In Figure 17a, the proposed jump function *J* returns 257 (a prime number). In Figure 17b, *J* function delivers the jump value of 514. These Figures bring the performance of DaC-Scan-4 for scale factor of 10 and 20, respectively. Thus, one can note that the value provided by the jump function is 12% faster than the value 1 for scale factor of 10 and 29% faster than the value 1 for scale factor of 20.



(a) Scale Factor 10 - *Jumpfunction* = 257



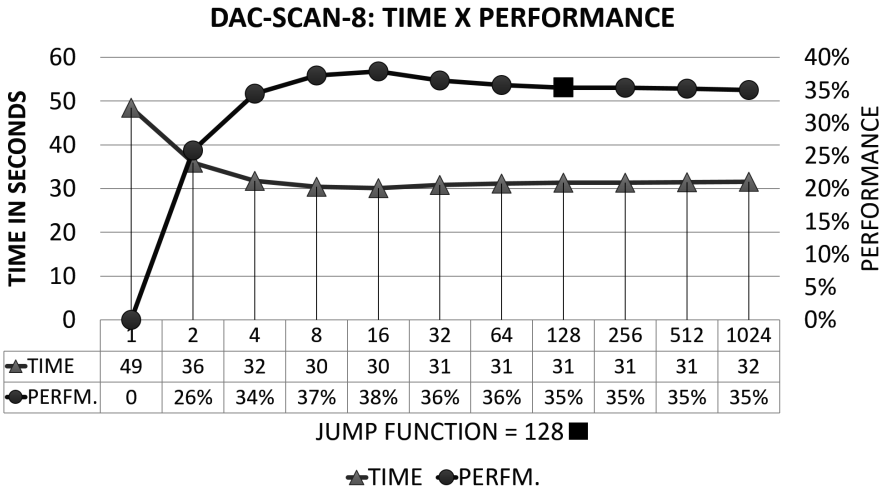
(b) Scale Factor 20 - *Jumpfunction* = 514

Figure 17 – DaC Scan - 4

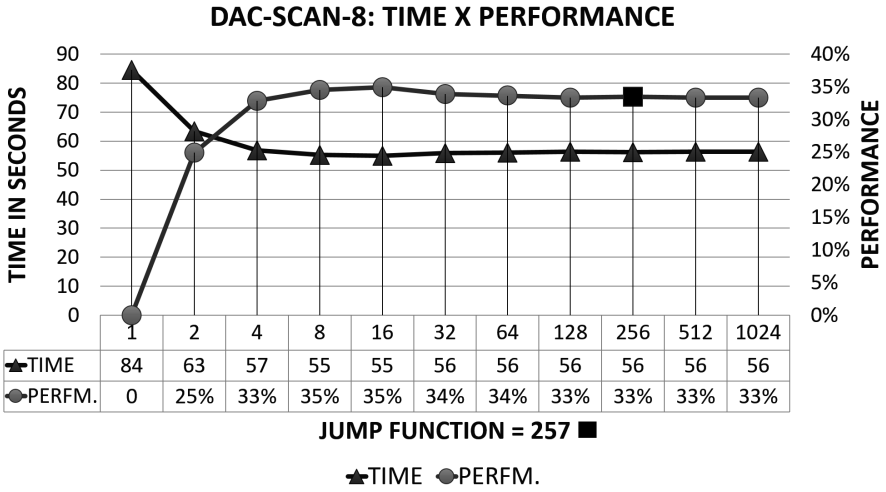
Nevertheless, in both scenarios depicted in Figures 17a and 17b, the values produced by the *J* function do not provide the lowest time for scanning the table *lineitem*. The best performance is achieved by a jump value of 64 (Figure 17a) and by a jump value of 32, when *lineitem* has scale factor 20 (Figure 17b). However, the difference between the best results and

the result obtained by using the jump values computed by J is only of 1% for the first scenario and 2% for the second scenario. This small performance variation leads us to claim that the use of the jump function J yields a quite good approximation to the best value of the jump function.

Figures 18a and 18b bring the results of the experiments with DaC-Scan-8 (eight threads) for scale factor 10 and 20, respectively. The result of function J is 128 and 257 for DaC-Scan-8 Scale factor 10 and 20. Both cases present similar gains in performance being 33% and 35% faster than a jump of 1 and deviating only 2% from the best result.



(a) Scale Factor 10 - $Jumpfunction$ = 128

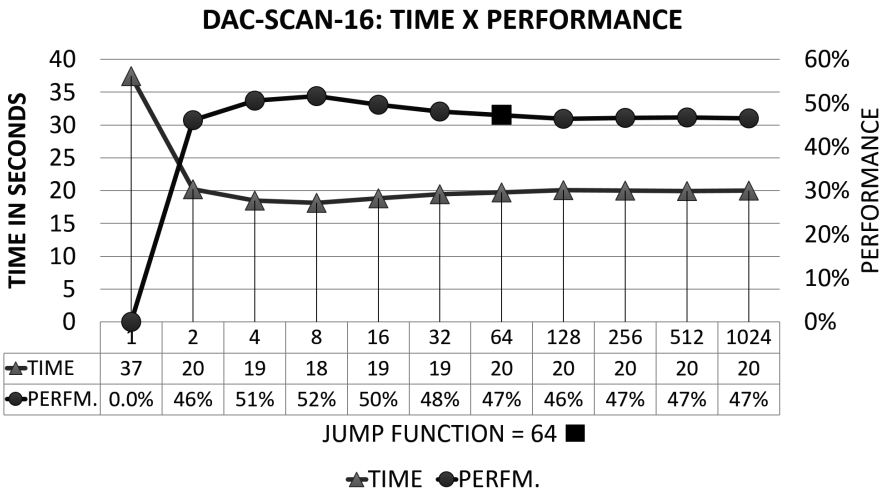


(b) Scale Factor 20 - $Jumpfunction$ = 257

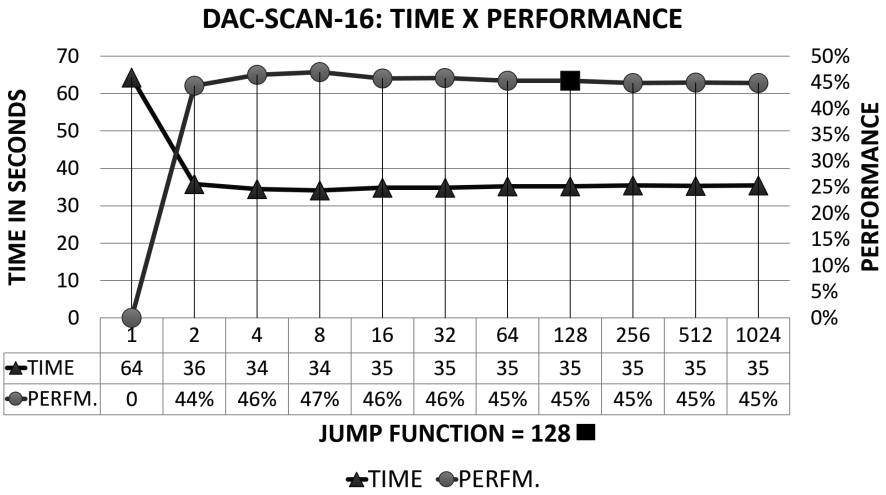
Figure 18 – DaC Scan - 8

Dac-Scan-16 (sixteen threads) for factor 10 and 20 can be seen in Figures 19a and 19b, respectively. We may see that Dac-Scan-16 follow the same trend, where the result of function J is 64 and 128 for DaC-Scan-16 Scale factor 10 and 20, respectively. Note that for scale factor of 10 (Figure 19a) the performance gain is of 47% and for scale factor 20 (Figure

19b) the gain is 45%. For both cases, there is a small variation performance from the best result.



(a) Scale Factor 10 - $Jumpfunction = 64$

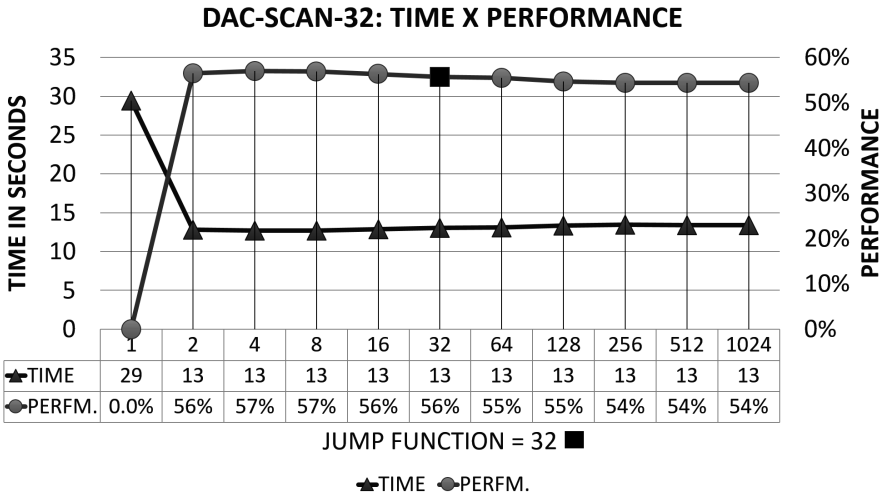


(b) Scale Factor 20 - $Jumpfunction = 128$

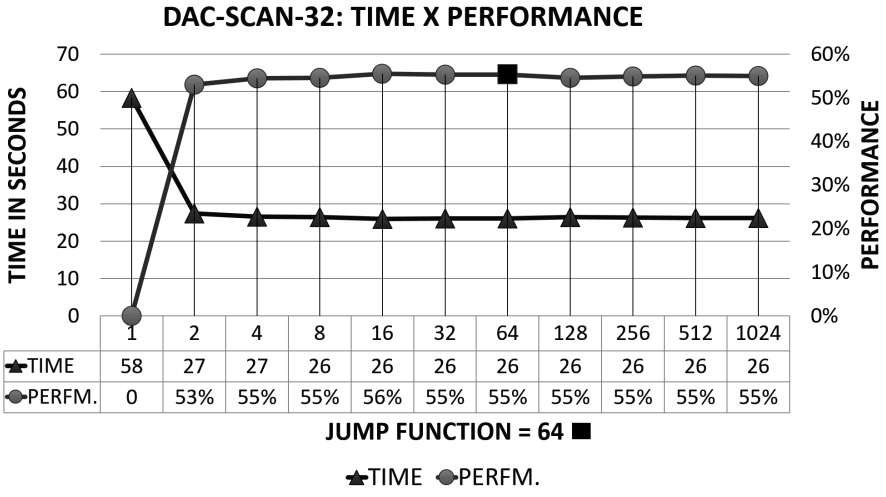
Figure 19 – DaC Scan - 16

Figure 20a and Figure 20b illustrate the performance enhancement of Dac-Scan-32 (thirty-two threads). For this scenario, using the value provided by the function *J*, the performance of DaC Scan is 56% better than of Dac-Scan-32 using a jump of 1 for scale factor 10 (Figure 20a) and 55% for scale factor 20 (Figure 20b) delivering the best result for both scale factors.

Figures 21a and 21b show the performance gains of DaC-Scan-64 (sixty-four threads). In this experiment, the function *J* delivered by DaC Scan is 32 for scale factor 10 and 64 for scale factor 20. One can note that the value delivered by the jump function is 56% faster than a jump of 1. Once more, there is a small performance variation between the best result and the result produced using the value delivered by the jump function *J*. However, such a variation is



(a) Scale Factor 10 - $Jumpfunction = 32$



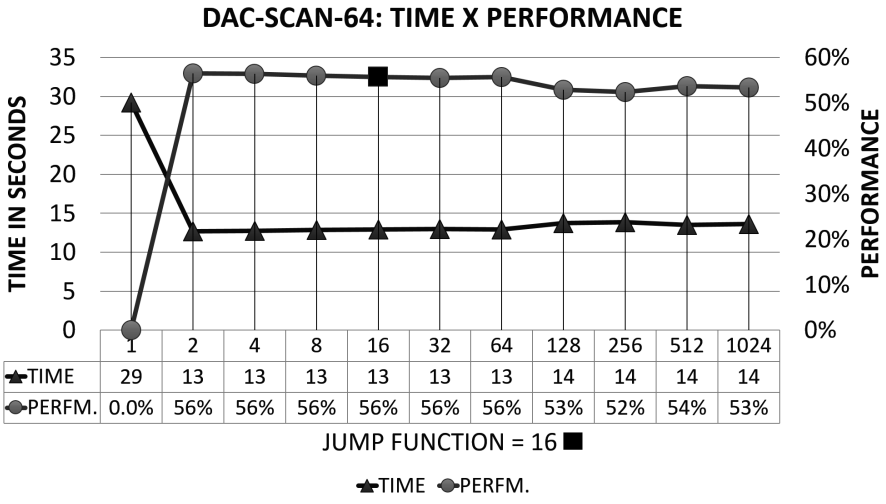
(b) Scale Factor 20 - $Jumpfunction = 64$

Figure 20 – DaC Scan - 32

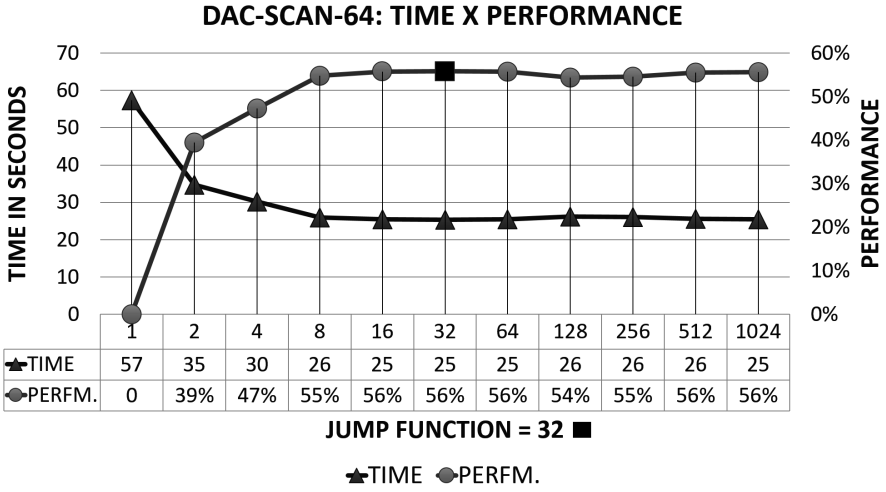
less than 5%.

As already mentioned in Section 6.2, the cloud computing paradigm has introduced a new perspective of the number of available processors, by means of which one may define a virtual machine with is a prime number of processors. Since the number of processors is the denominator for the jump function J, one can argue that the jump function J would not work with prime numbers. In order to show that J works efficiently even for prime numbers of processors or threads, experiments with prime numbers of threads have been conducted. The values employed in the experiments were 3, 7, 17, 31 and 61 (Figures 22 to 27).

The first experiments with prime numbers of threads compare the performance of a sequential scan operator and DaC Scan without taking profit of SSD internal parallelism (Figures 22a and 22b). For that reason, the jump function in both figures is 1. The goal of such



(a) Scale Factor 10 - *Jumpfunction* = 16

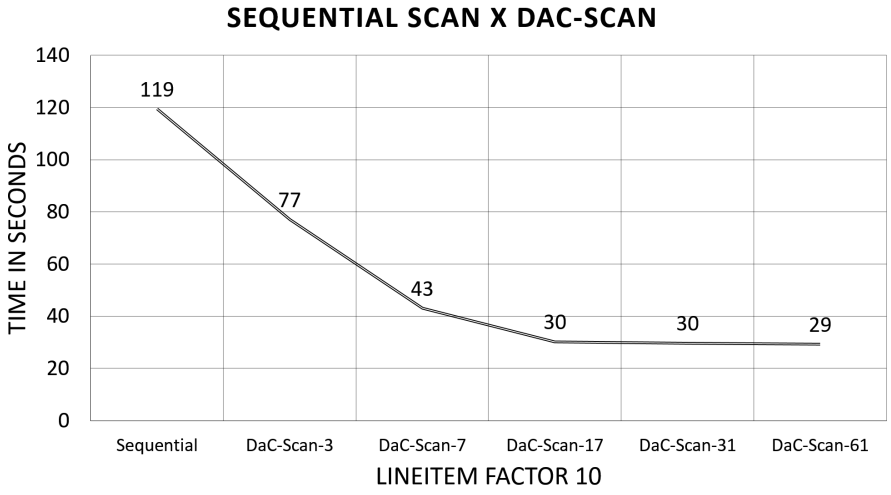


(b) Scale Factor 20 - *Jumpfunction* = 32

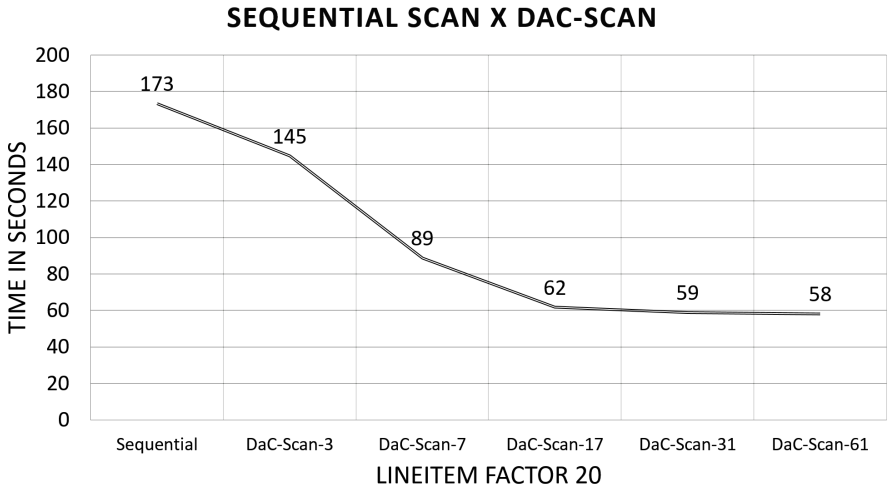
Figure 21 – DaC Scan - 64

experiments is to show that, even without considering SSD internal parallelism, the proposed scan operator delivers better performance than the classical sequential scan operator. Figure 22a depicts the results of running DaC Scan and a sequential scan on TPC-H table *lineitem* with scale factor 10. In turn, Figure 22b illustrates the results of the same experiment on *lineitem* with scale factor 20.

Observing Figures 22a and 22b more closely, one can see that for all scenarios DaC Scan is faster than the sequential scan operator. DaC Scan running on a three threads scenario (DaC-Scan-3) is 55.5% faster than sequential scan for scale factor of 10 and 17.8% faster for scale factor of 20, DaC-Scan-7 is 78.8% faster than DaC-Scan-3 for scale factor of 10 and 63.04% faster for scale factor of 20, DaC-Scan-17 is 42.48% faster than DaC-Scan-7 for scale factor of 10 and 43.8% faster for scale factor of 20. DaC-Scan-31 is 1.8% faster than DaC-Scan-17 for



(a) Scale Factor 10 - Jump function = 1



(b) Scale Factor 20 - Jump function = 1

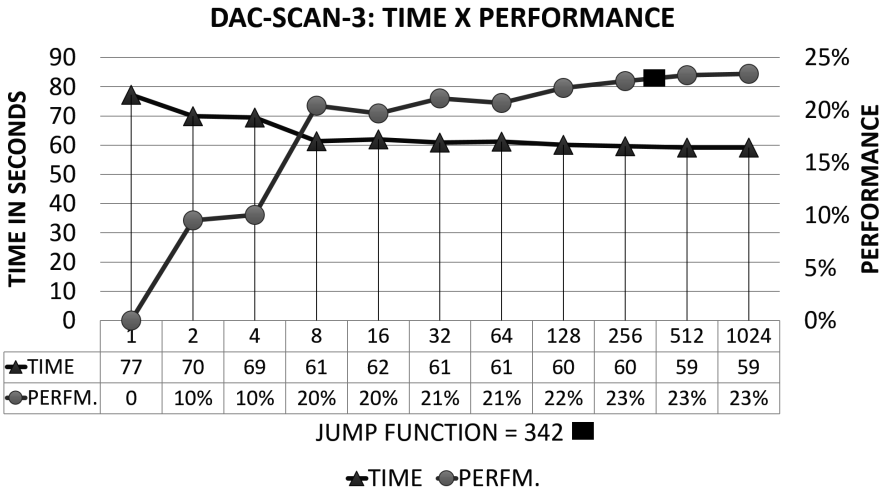
Figure 22 – Sequential and DaC Scan

scale factor of 10 and 4.9% faster for scale factor of 20. Finally, DaC-Scan-61 is only 1.26% faster than DaC-Scan-31 for scale factor of 10 and only 1.26% faster for scale factor of 20. As previously observed, this is because there is a slight performance variation with more than 32 threads (see Figures 15a and 15b).

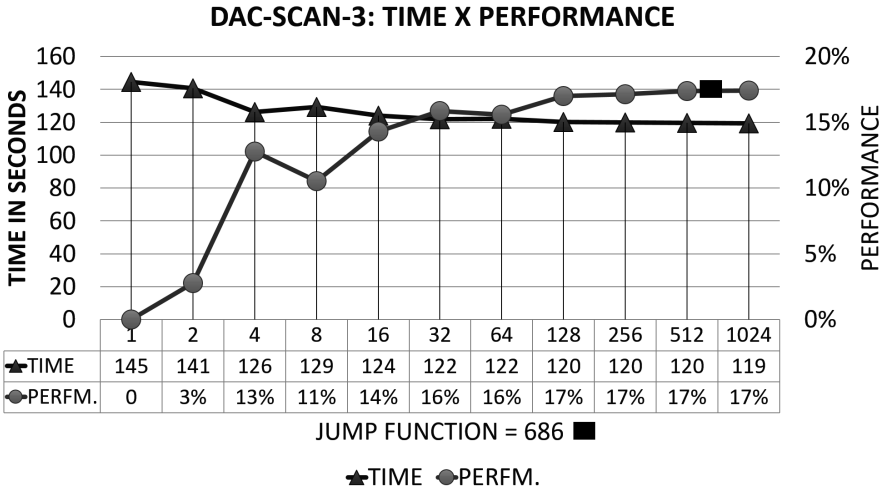
The next figures (from Figure 23 to 27) depict the results of DaC Scan running on different prime numbers of threads. For the sake of clarity, the results of those experiments have been approximated to integer values. As we have done before, the performance gains delivered by using different values for the jump function (i.e., 1, 2, 4, 8, 16, 32, 64, 128, 256. 512 and 1024) are confronted to the performance delivered by the value of function J yielded by DaC Scan (illustrated as a hatched square in the figures). A value 1 for the jump function means the use of all available threads for running classical round-robin strategy for correctly reading

the table. Recall that, in this case, thread n reads a block which is neighbour to blocks read by threads $n-1$ and $n+1$.

Figures 23a and 23b bring the results of running DaC Scan on three threads (DaC-Scan-3) for scanning the table *lineitem* with scale factors 10 and 20, respectively. For DaC-Scan-3, the computed value by J function is 342 for scale factor 10 (see Figure 23a) and 686 scale factor 20 (see Figure 23b). Observe that the value provided by the jump function is 23% faster than the value 1 for scale factor of 10 and 17% faster than the value 1 for scale factor of 20.



(a) Scale Factor 10 - Jump function = 342

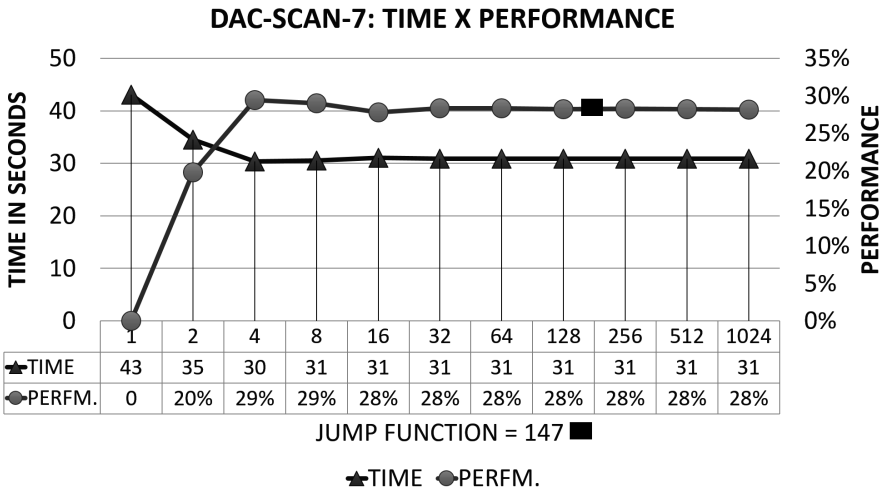


(b) Scale Factor 20 - Jump function = 686

Figure 23 – Running DaC Scan on three threads.

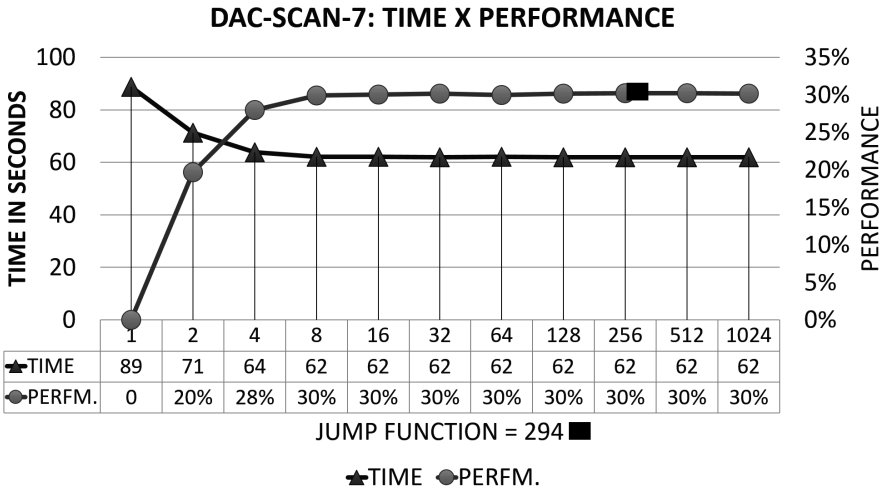
The results of experiments in which DaC Scan has been executed on seven threads are illustrated in Figures 24 and 25. Those results reveal that there is a similar trend to results produced when running DaC-Scan-8 and DaC-Scan-16. As we have seen before, there is a little difference between the best results and the actual results computed by jump functions J. This

slight performance variation leads us, once more, to consider that even when the real size of the clustered block is not known, the use of the proposed jump function *J* yields a quite suited result w.r.t. performance response.



	1	2	4	8	16	32	64	128	256	512	1024
TIME	43	35	30	31	31	31	31	31	31	31	31
PERFM.	0	20%	29%	29%	28%	28%	28%	28%	28%	28%	28%

(a) Scale Factor 10 - Jump function = 147



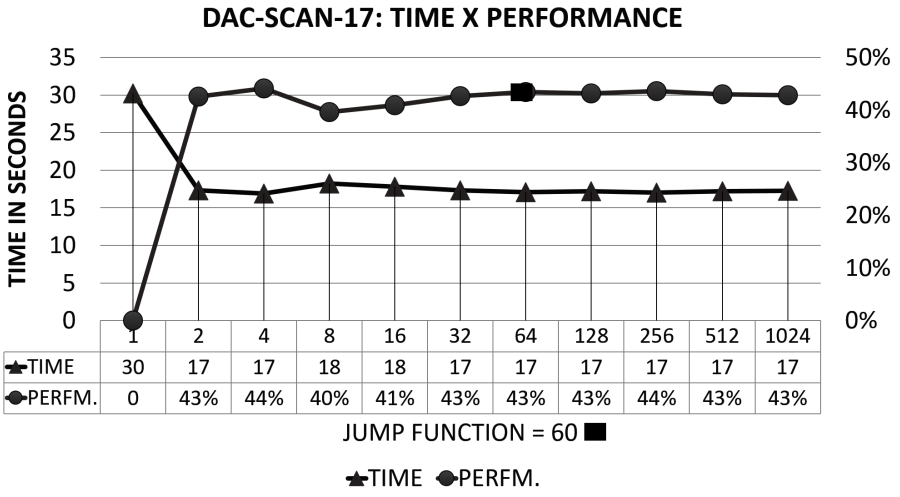
	1	2	4	8	16	32	64	128	256	512	1024
TIME	89	71	64	62	62	62	62	62	62	62	62
PERFM.	0	20%	28%	30%	30%	30%	30%	30%	30%	30%	30%

(b) Scale Factor 20 - Jump function = 294

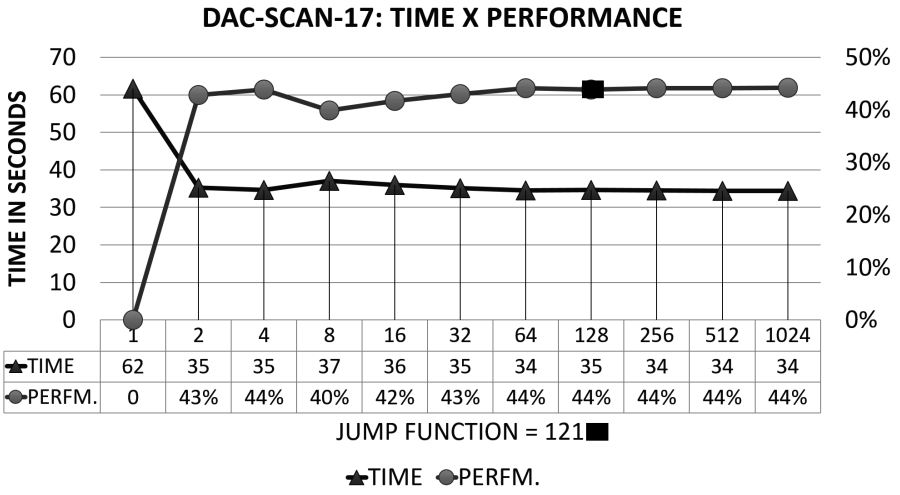
Figure 24 – Running DaC Scan on seven threads.

The experiments of running DaC Scan on 31 threads (DaC-Scan-31) yields the results which is depicted in Figures 26a and 26b. In this scenario, using the value provided by the function *J*, the performance of DaC Scan is 56% better than of DaC Scan using a jump of 1 for scale factor of 10 (Figure 26a) and 55% for scale factor of 20 (Figure 26b) delivering the best result for both.

Finally, Figure 27a and Figure 27b show the performance gains of DaC-Scan-61 (sixty-one threads), the function *J* delivered us 16 for scale factor of 10 and 66 for scale factor of 20, for both cases, the value delivered by the jump function is 56% faster than a jump value



(a) Scale Factor 10 - Jump function = 60

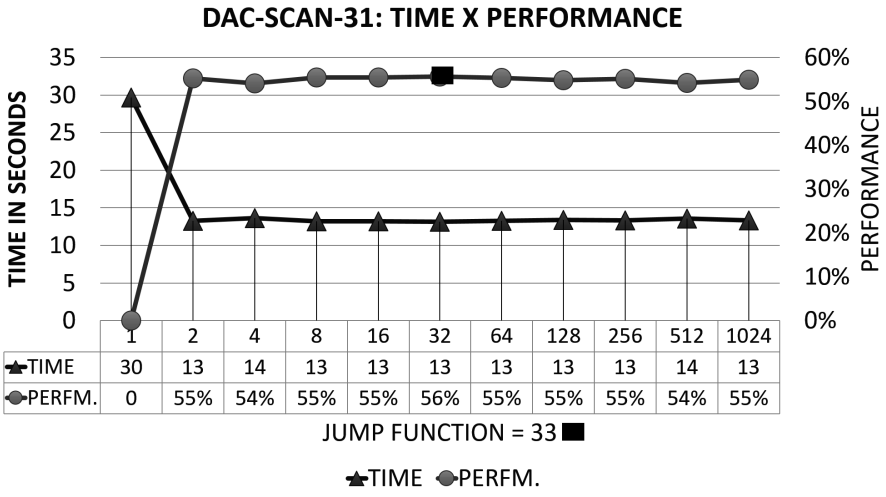


(b) Scale Factor 20 - Jump function = 121

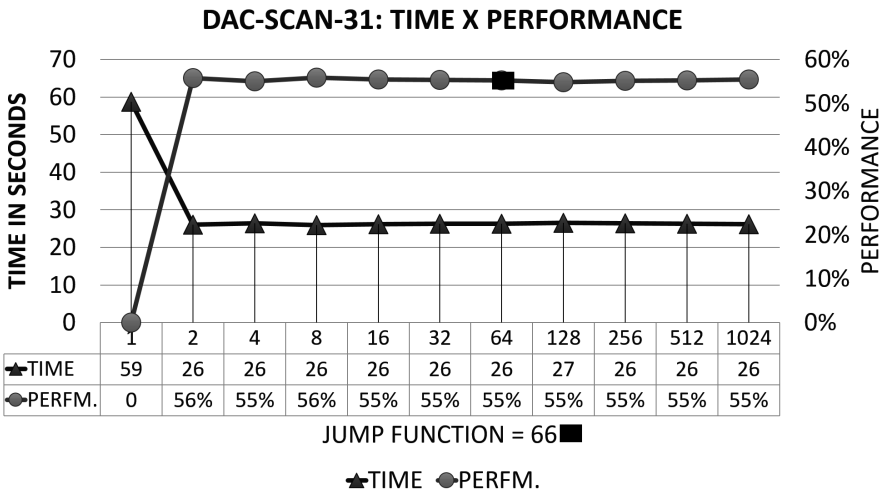
Figure 25 – Running DaC Scan on seventeen threads.

equal to 1.

As already mentioned, to explore internal SSD parallelism, applications should be aware of SSD block size (see Section 2.3). However, each flash-based solid state memory device may have a different physical block size from others SSDs. It is not so trivial to obtain such information from SSD manufacturer. Therefore, to conclude this section, it is important to highlight that the results presented in this section show the efficiency of the proposed jump function *J* to infer the size of physical blocks of a given SSD. Introducing the use of the jump function in the proposed scan operator, make it able to efficiently explore SSD internal parallelism. Furthermore, DaC Scan is able to run on several threads or processors.

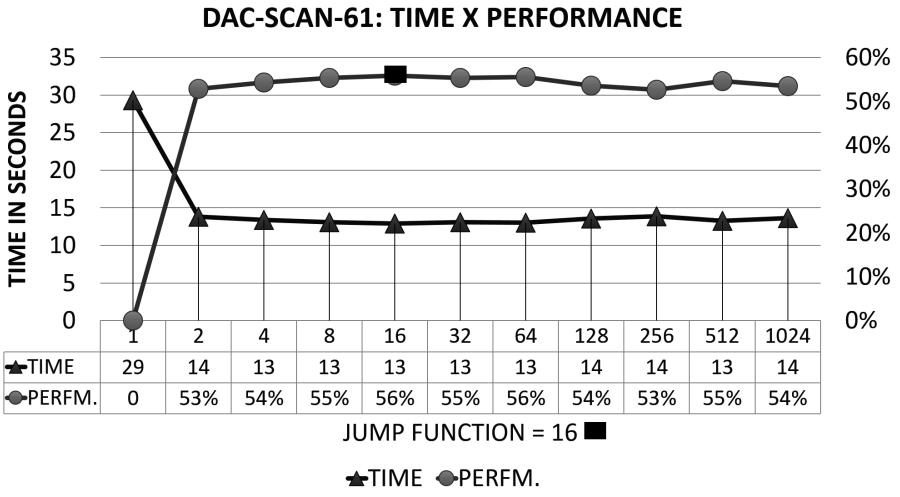


(a) Scale Factor 10 - Jump function = 33

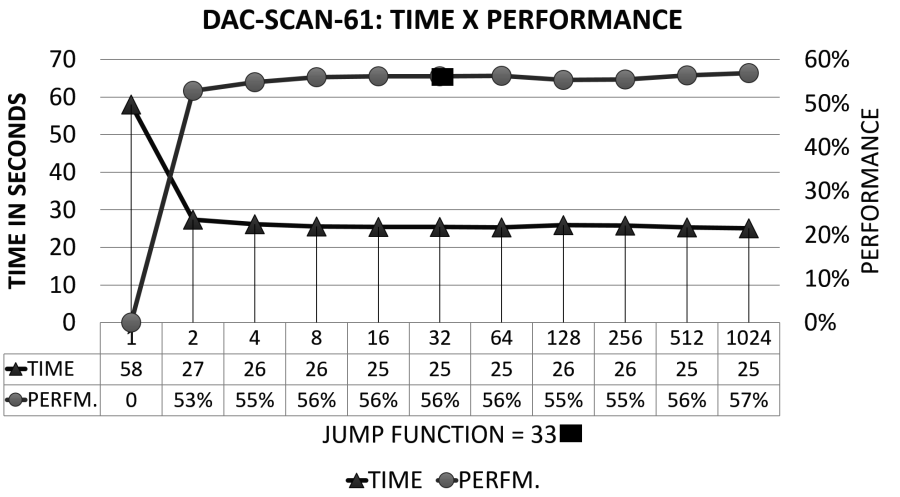


(b) Scale Factor 20 - Jump function = 66

Figure 26 – Running DaC Scan on thirty one threads threads.



(a) Scale Factor 10 - Jump function = 16



(b) Scale Factor 20 - Jump function = 33

Figure 27 – Running DaC Scan on sixty one threads threads.

8.3 DaC Join

All experiments have been performed on a server with an Intel Core CPU i7-3770 at 3.4GHz and 16GB of RAM, running Windows 7 Professional Service Pack 1. The used SSD device was a SATA Samsung SSD 840 SCSI. In spite of using a machine with 16GB of RAM, we limited the memory size used by the join operator. Thus, we have used three different memory sizes: 50MB, 100MB and 200MB. The key idea behind that strategy is to stress the DaC Join execution in environments with severe restrictions of main memory. By doing this, we simulate the worst scenarios for DaC Join.

The experiments have been executed over the TPC-H database, scale factor 10. This database stores approximately 13GB for storing 8 tables, from which the table *l*ineitem is populated with approximately 6×10^7 tuples and has a size of approximately 8GB. Table 5 brings the mains characteristics of the used tables.

Table Name	Rows	Kylobytes
PART	2,000,000	258,960
PARTSUPP	8,000,000	1,211,816
CUSTUMER	1,500,000	253,536
ORDERS	15,000,000	1,825,584
LINEITEM	59,986,052	8,428,864
NATION	25	16
REGION	5	16
SUPPLIER	100,000	14,944

Table 5 – TPC-H Tables.

The used workload is composed by two different classes of queries: 2-way join queries and n-way join queries. The class of 2-way join queries is characterized by presenting two-way join operations between two large TPC-H database tables, namely lineitem and orders. This class is composed of queries QA and QB, depicted in Table 6. The difference between QA and QB is the existence of a selection operation in QA, which should be computed before the join operation due to an optimization heuristic. For that reason, QA returns 1,147,084 tuples, while QB delivers 59,986,052 tuples. The idea of using those queries is to induce a high frequency of memory overflow events.

Besides QA and QB, experiments using query QC (see Table 6) have been performed. QC presents a join between table lineitem (with approximately 6×10^7 tuples stored in approximately 8GB) and region (with only 5 tuples occupying 16kb). The goal of such experiments is to make clear that even for join operations between tiny memory-fitting table and a huge fact table,

DaC Join presents quite efficient results.

In turn, the second class of queries presents three TPC-H-like queries, denoted Q3n, Q5n and Q10n, which have been defined based on the original TPC-H queries Q3, Q5 and Q10 by removing the columns involved in the projection operation, and the operations GROUP BY and ORDER BY (see Table 6). Those queries aim at evaluating DaC Join behavior in the presence of n-way joins.

Query	Sql Statements	Rows returned	% of biggest table (%)
	2-way join queries		
QA	select * from orders, lineitem where o_orderdate >= 1993-10-01 and o_orderdate < 1994-01-01 and l_returnflag = 'R' and l_orderkey = o_orderkey	1.147.084	2%
QB	select * from orders, lineitem where l_orderkey = o_orderkey	59.986.052	100%
QC	select * from region, lineitem where l_orderkey = r_regionkey	14	9.3e-7%
	N-way join queries		
Q3n	select * from customer, orders, lineitem where c_mktsegment = 'building' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < 1995-03-15 and l_shipdate > 1995-03-15	302.114	0.5%
Q5n	select * from customer, orders, lineitem, supplier, nation, region where l_orderkey = o_orderkey and c_custkey = o_custkey and l_suppkey = s_suppkey and c_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'ASIA' and o_orderdate >= 1994-01-01 and o_orderdate < 1995-01-01	1.825.856	3%
Q10n	select * from customer, nation, orders, lineitem where c_custkey = o_custkey and l_orderkey = o_orderkey and c_nationkey = n_nationkey and o_orderdate >= 1993-10-01 and o_orderdate < 1994-01-01 and l_returnflag= 'R'	1.147.084	2%

Table 6 – Queries used in the experiments

For a fair comparison, we have implemented DaC Join, Flash join and Hybrid Hash Join in Java using a non-transactional row storage engine. This engine provides methods for reading and writing data in 8k pages (blocks). The buffering functionality has been removed

from the storage engine. Therefore, each data access (required by the evaluated join algorithm) corresponds to direct access on SSD. Moreover, in all experiments, the same amount of main memory has been allocated for each analyzed join operator.

The experiments have been conducted for measuring effectiveness and efficiency delivered by DaC Join during the execution of 2-way and n-way joins. Regarding effectiveness, two metrics have been employed: number of write operations on SSD and number of additional read operations. The latter metric is necessary due to the fact that DaC Join implements a mixed materialization strategy (see Section 7.2). The metric for verifying efficiency was execution time for computing the join operation. To compute those metrics, each query has been executed 10 times. The highest and the lowest values have been discarded, and the average of the remaining values was computed.

In order to simulate the use of multiple core or processors, we have implemented DaC Join for using multiple java threads. In the experiments, three different environments have been utilized: DaC Join running on 2 (*DaC*-2), 4 (*DaC*-4) and 8 (*DaC*-8) threads. Although it is not quite often to have machines with prime number of cores, we run experiments with prime numbers of threads, more specifically with 3 and 7 threads, as well.

8.3.1 DaC-Join Effectiveness

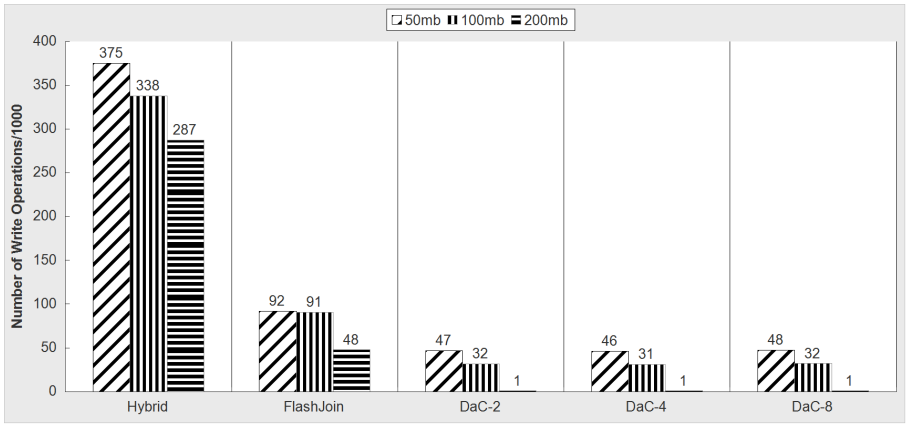
For the sake of clarity, this section is divided according to the analyzed metric. Thus, we firstly analyze the results regarding the metric, which reports the number of write operations. Thereafter, we present and discuss the results w.r.t. the number of additional read operations. In both cases, we initially address the results of experiments with queries belonging to the class of 2-way join queries, namely QA, QB and QC. After that, we evaluate the results of experiments with queries belonging to the class of n-way join queries: Q3n, Q5n and Q10n (see Table 6).

8.3.1.1 Number of Write Operations

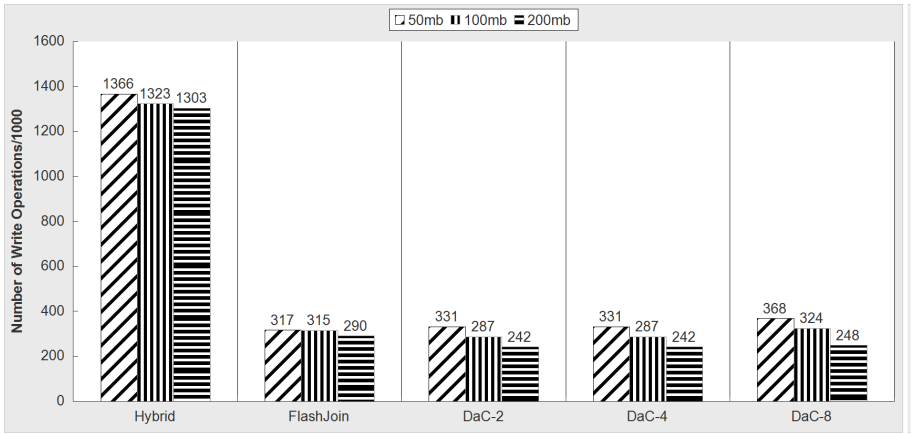
Figures 28a and 28b depict the amount of write operations for processing QA and QB w.r.t. the used join operator (hybrid hash join, Flash join and DaC Join) and the amount of available main memory (50MB, 100MB and 200MB).

Analyzing Figure 28a, one can see that for all scenarios DaC Join (with 2, 4 and 8 threads) outperform hybrid hash join and Flash join w.r.t. the number of write operations. For instance, DaC Join running with 8 threads (*DaC*-8 in Figure 28a) and 50MB of available main

memory, it requires $48x10^3$ write operations to process QA, while Flash join executes $92x10^3$ write operations. In this specific case, DaC Join performs 47.82% less write operations than Flash join. DaC Join running with 8 threads and 100MB of memory, it writes $32x10^3$ times on SSD. With the same 100MB of main memory, Flash join executes $91x10^3$ write operations. Therefore, both operators decrease the number of write operations, in case the amount of main memory is increased. Nonetheless, DaC Join performs 64.83% less write operations than Flash join. However, the best result for DaC Join has been achieved with the experiment involving 8 threads and 200MB of main memory. In this scenario, DaC Join executes 97.91% less write operations than its major competitor, the Flash join operator.



(a) Number of Write Operations for QA.



(b) Number of Write Operations for QB.

Figure 28 – Write Operations for QA and QB

Query QB has been used to examine the behavior of DaC Join for processing time and memory consuming queries. QB joins the two largest tables of TPC-H database (lineitem and orders) and returns $59,986,052$ tuples (see Table 5). Figure 28b shows the number of writes yielded by the evaluated join operators during the execution of QB. DaC Join outperforms once

more hybrid hash join and Flash join. For instance, consider the worst scenario for executing QB, i.e., with 50MB of available memory. In this case, DaC Join executes $368x10^3$ write operations, while Flash join executes $317x10^3$. Thus, Flash join performed 13.85% less write operations than DaC Join. This is because DaC Join triggered the memory cleaning process several times in order to free space in main memory to load part of the hash-table hierarchy. Recall that the memory cleaning process requires write operations on SSD. For the scenario with 200MB of available memory for processing QB, DaC Join with 8 threads writes $248x10^3$ times on SSD, while Flash join writes $290x10^3$ times. In latter case, DaC Join writes 14.48% less times than Flash join.

Figure 29 depicts the amount of write operations for processing QA with prime numbers of threads. The idea of having x-axis of Figure 29 both prime numbers and numbers, which are multiple of 4, is to facilitate a comparative analysis of the DaC Join behavior in both scenarios. Looking more closely to Figure 29, one can observe that the use of a prime number of threads induce an increase in the amount of write operations. Nevertheless, such an increment is sub-linear.

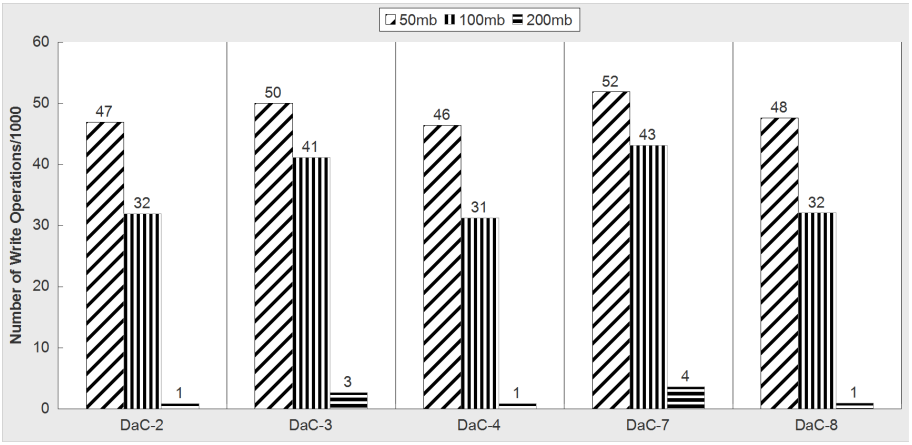
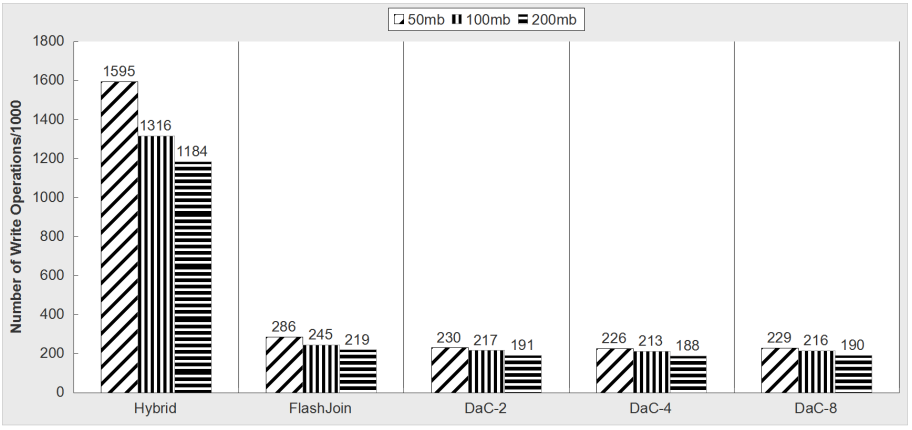


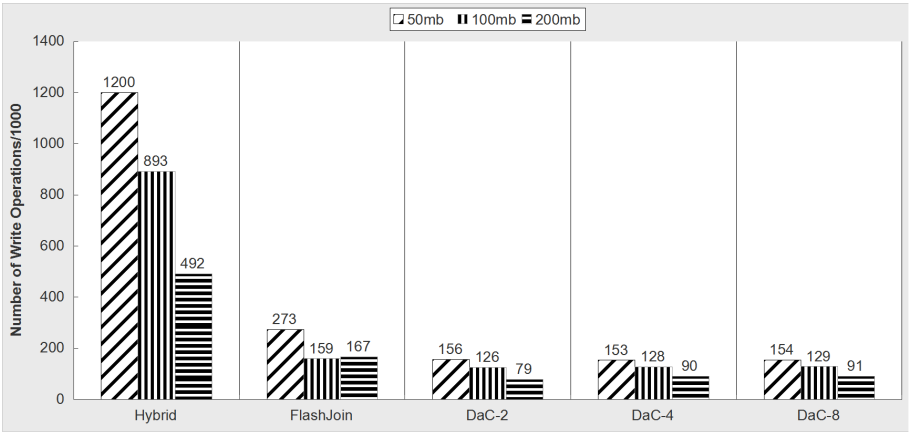
Figure 29 – Number of Write Operations for QA with prime numbers of processors.

As already mentioned, experiments with query QC has the intention of showing the efficiency of DaC Join in computing join operations between memory-fitting table and a huge table. Examining Figure 30a, one may observe that DaC Join preserves its behavior, which is to minimize the number of write operations on secondary memory.

It is important to note that theoretically the amount of writes performed by the DaC Join, for the same amount of memory, should be constant, regardless the number of threads. However, the results presented in this section show that there is a small variation in the number of



(a) Number of Write Operations for QC.



(b) Number of Write Operations for Q3n.

Figure 30 – Write Operations for QC and Q3n.

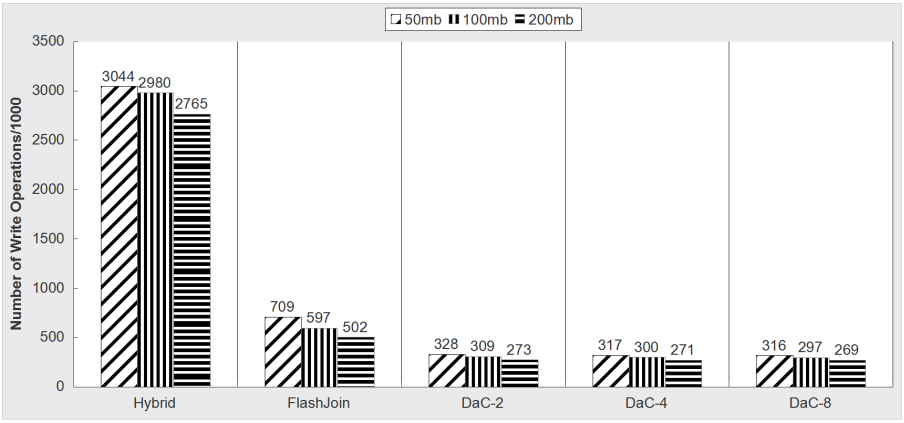
writes observed for DaC Join for the same number of memory. This variation is induced by two factors: *(*i) the utilized hash functions are dynamic, since they are defined based on the number of available threads or processors (see Equations 7.1 and 7.2), and; *(*ii) the threads' concurrency control model implemented by the Java Virtual Machine.

Next, we present and discuss the results of experiments with queries belonging to the class of n-way join queries: Q3n, Q5n and Q10n (see Table 6). It important to emphasize that Q3n is a 3-way join query, Q5n is a 6-way join query and Q10n is a 4-way join query. Figures 30b, 31a and 31b brings the amount of write operations for processing Q3n, Q5n and Q10n w.r.t. the used join operator (hybrid hash join, Flash join and DaC Join) and the amount of available main memory (50MB, 100MB and 200MB) for running the join operator.

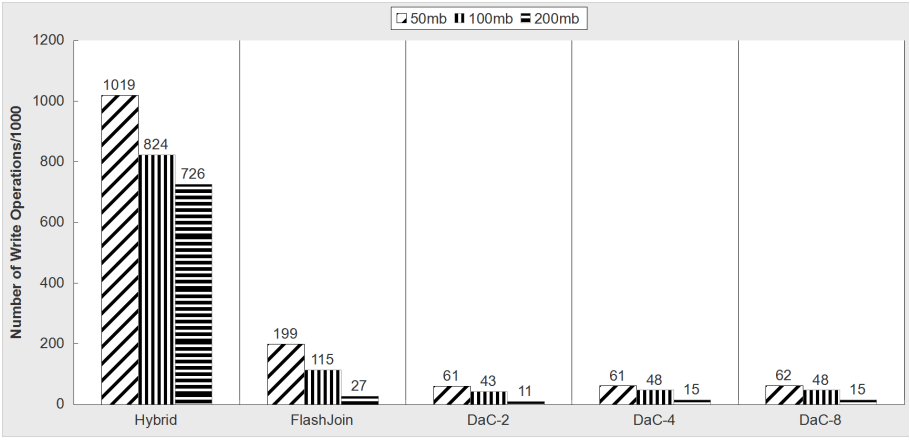
Analyzing the results presented in Figures 30b, 31a and 31b, one can observe that for all scenarios DaC Join (with 2, 4 and 8 threads) presents better results than hybrid hash join and Flash join w.r.t. the number of write operations during the execution of n-way join queries. To illustrate this fact, consider the results in Figure 30b. For processing Q3n with 50MB, Flash join

needs to execute $292x10^3$ write operations, while DaC Join executes $154x10^3$ write operations, a reduction of $138x10^3$ write operations.

The results depicted Figure 31a are more representative for revealing the better performance provided by DaC Join. For computing the Q5n's 6-way join operation (see Table 6) with only 50MB of main memory, DaC Join with 8 threads (*DaC*-8) has executed $316x10^3$ write operations. For the same experiment, Flash join has written $755x10^3$ times on SSD. Therefore, DaC Join has executed 58.14% less write operations than Flash join, which represents $439x10^3$ less write operations.



(a) Number of Write Operations for Q5n.



(b) Number of Write Operations for Q10n.

Figure 31 – Write Operations for Q5n and Q10n.

The results presented in this section show that DaC Join effectively reduces the number of write operations w.r.t. hybrid hash join and Flash join.

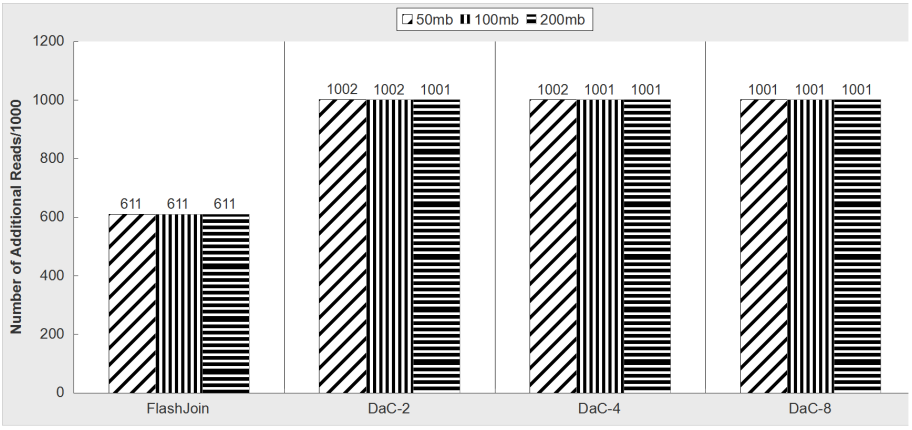
8.3.1.2 Number of Additional Read Operations

First of all, it is important make it clear that hybrid hash join does not perform any additional read, since it uses early materialization. For that reason, hybrid hash join does not appear in the results presented in this section.

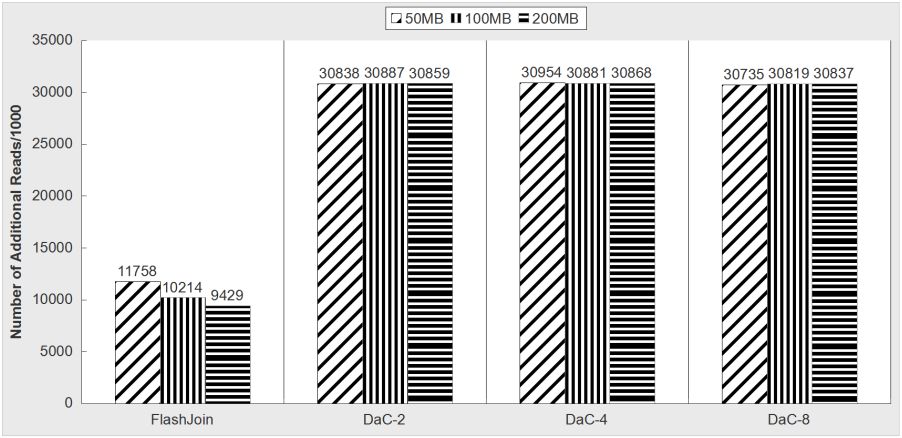
As already mentioned, Flash join implements a late materialization strategy. In that strategy, attributes are materialized (fetched) when they are needed (for computing a join operations or for delivering the final projection operation). On the other hand, DaC Join implements a mixed materialization strategy, in which join attributes are materialized as early as possible and final projection attributes are materialized as late as possible. Both strategies aim at optimizing the use of main memory. Nonetheless, they imply in executing more read operations. Although, a read operation may be up to 10^3 times faster than HDDs, increasing the number of read operations may impact on the overall time to execute the join operator. In this sense, we investigate in this section the number of additional reads required by DaC Join and Flash join. First, we analyze the behavior of both operators processing 2-way join queries and thereafter processing n-way join queries.

Figures 32a and Figure 32b depict the results of the metric, which report the number of additional read operations for queries QA and QB. Examining both figures, one can note that Flash join presents better results than DaC Join. To illustrate this observation, consider the execution of QB in a scenario with 50MB of main memory. One can observe in Figure 32b that Flash join executes $11,758x10^3$ additional reads. For the same amount of memory, DaC Join requires $30,838x10^3$ additional reads, almost three times more additional read operations than Flash join. Nonetheless, this worse performance regarding this metric is compensated by the better performance regarding the metric number of write operations achieved by DaC Join (see Figure 28b). The results depicted in Figure 35b, which are analyzed in the next section, show that our claim is correct.

Flash join outperforms DaC Join w.r.t. the number of additional reads by running a sort operation in order to guarantee sequential reads instead of random reads (GRAEFE; HARIZOPOULOS, 2010). On the other hand, a sort operation implies an increase of computational cost and in the number of write operations (see Figures 28a and 28b). Notwithstanding, Flash join presents better results regarding this metric only for 2-way join queries, as we show next. Furthermore, the results in Section 8.3 indicate that DaC Join presents lower execution time than Flash join. This is because DaC Join implements a more aggressive strategy for reducing the



(a) Number of Additional Reads for QA.



(b) Number of Additional Reads for QB.

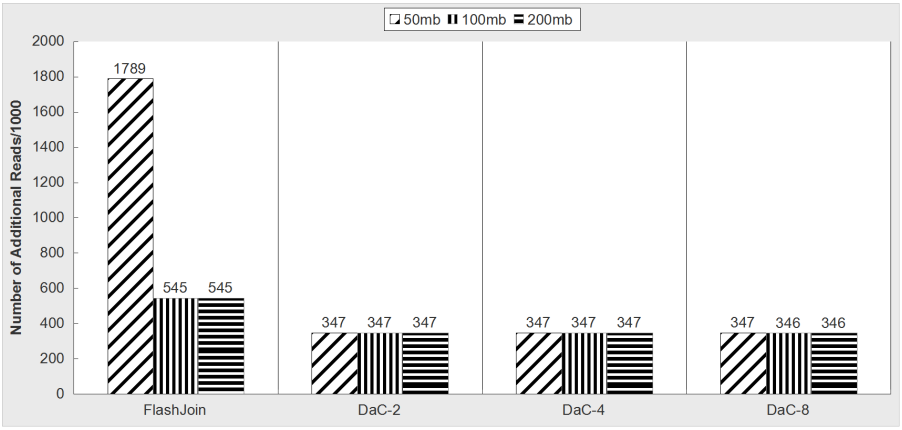
Figure 32 – Additional Reads for QA and QB.

number of write operations (a time-consuming operation in SSDs).

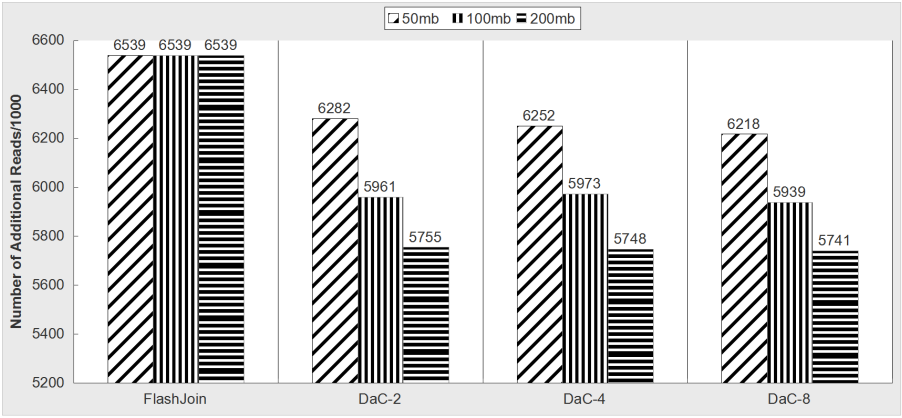
Figures 33a, 33b and 34 show the number of additional reads required for processing queries Q3n, Q5n and Q10n, respectively. Differently from the experiments with 2-way join queries, in all experiments using n-way join queries, DaC Join presents better results for the additional-read-operations metric. This is because DaC Join implements mixed materialization, where all the columns needed to perform all join operations (in an n-way join) are read just once, i.e., at the beginning. In other words, DaC Join does not require any additional read for fetching the join attributes during the execution of an n-way join query.

In Figure 33a, one can see that DaC Join with 8 threads needs $347x10^3$ additional reads for processing Q3n (a 3-way join) with 50MB of main memory. Flash join in turn executes $1,789x10^3$ additional reads, which means that *DaC*-8 performs 80.60% less additional reads than Flash join.

Regarding the execution of Q5n with 50MB of memory (see Figure 33b), DaC Join with 8 threads needs $6,218x10^3$ additional reads to performing Q5n, while Flash join executes



(a) Number of Additional Reads for Q3n.



(b) Number of Additional Reads for Q5n.

Figure 33 – Additional Reads for Q3n and Q5n.

$6,539x10^3$ additional reads. Hence, *DaC*-8 runs 4.9% less additional reads than Flash join.

For processing Q10n with 50MB, DaC Join with 2 threads needs $1,524x10^3$ additional reads, while Flash join runs $2,051x10^3$ additional reads. Thus, *DaC*-2 executes 25.69% less additional reads than Flash join.

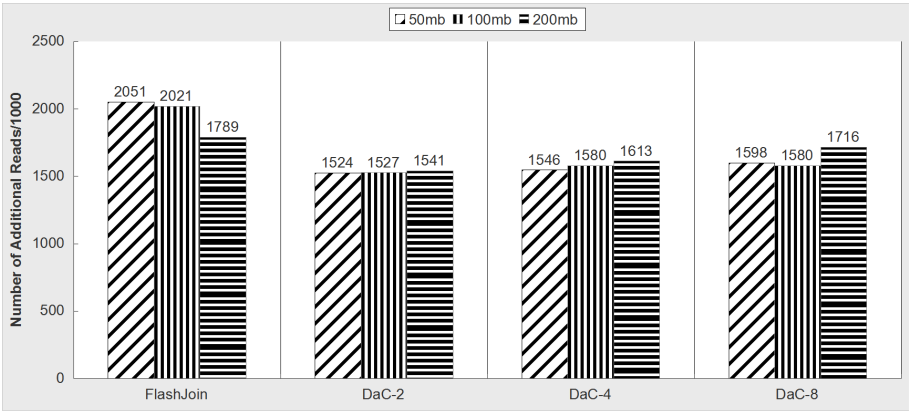


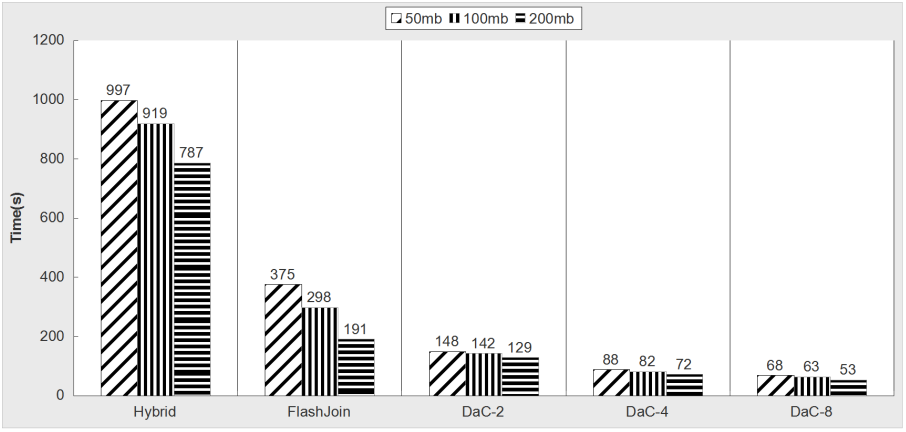
Figure 34 – Number of Additional Reads for Q10n.

8.3.2 DaC-Join Efficiency

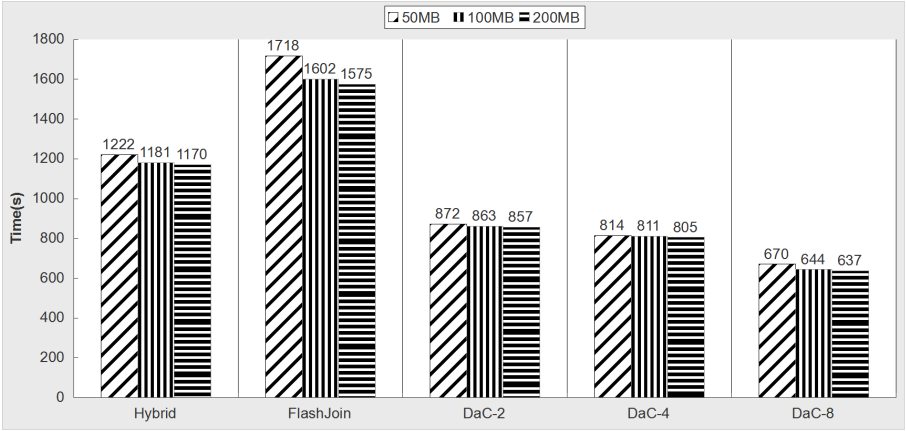
For evaluating DaC Join efficiency, we have utilized the metric execution time. Thus, during the experiments, we have measured the time consumed to execute the queries depicted in Table 6 by using Hybrid Hash Join, Flash join and DaC Join with 2, 4 and 8 threads.

First, we examine the efficiency of the evaluated join operators for executing 2-way join queries, i.e., QA and QB.

Figure 35a reveals that DaC Join achieves better performance than hybrid hash join and Flash join for executing QA. To illustrate this statement, DaC Join with 2 threads (*DaC*-2) in a context of 50MB of available memory processes QA in 148s, while Flash join consumes 375s. Therefore, *DaC*-2 is 60.53% faster than Flash join for processing QA. However, the best case for DaC Join is to run using 8 threads, exploring intra-operator parallelism in a higher degree. In this case, DaC Join requires only 68s to process the join operations in QA. Thus, *DaC*-8 is 81.86% faster than Flash join, i.e., approximately six times faster.



(a) Response Time for QA.



(b) Response Time for QB.

Figure 35 – Response Time for QA and QB.

Regarding the execution of QB, a time-consuming query, which returns 59,986,052 tuples, DaC Join outperforms hybrid hash join and Flash join. Figure 35b indicates that, for a memory size of 200MB, DaC Join with 2 threads consumes 857s to execute QB, while Flash join ran QB in 1575s. Thus, DaC Join is 45.58% faster than Flash join even when it uses only 2 threads, its lowest degree of intra-operator parallelism. For the experiment, in which DaC Join runs with 8 threads, it processes QB in 637s, which makes *DaC*-8 59.55% faster than Flash join. In fact, DaC Join with 8 threads (*DaC*-8) presents the smallest response time for executing QB in all analyzed scenarios (50MB, 100MB and 200MB of memory size). Such a result validates our hypothesis that dividing the problem of join tables in several threads or processors, it may be possible to conquer efficient results.

It is worth to highlight the fact that Hybrid Hash Join outperforms Flash join w.r.t. the experiment with QB in all three analyzed scenarios, although Flash join implements hybrid hash join in its join kernel (see Section 5.3) and has been proposed to be an SSD-aware join operator. This phenomenon rises due to the following factors. First, Flash join uses late materialization strategy, which demands a high number of additional reads (see Figure 32b). Recall that hybrid hash join does not perform any additional read. Second, for processing QB, Flash join needs to execute a high amount of write operations (see Figure 28b), e.g., due to memory overflow events.

Figure 36 depicts the results of running DaC Join with a prime number of processor.

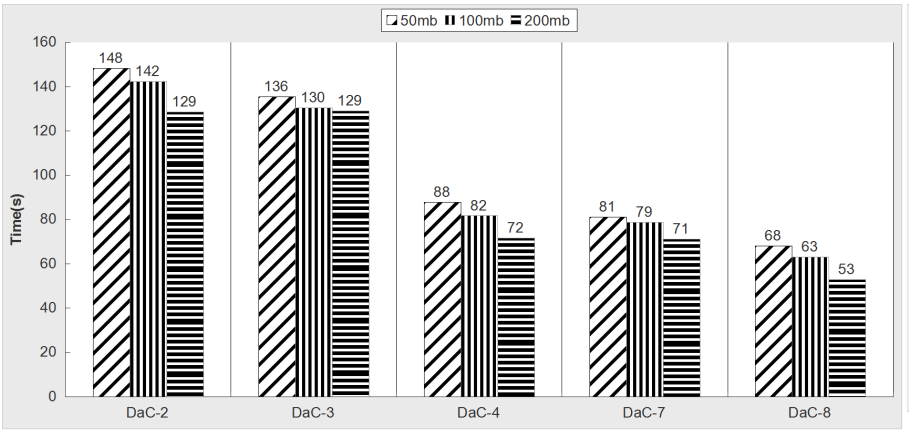


Figure 36 – Response Time for QA with prime numbers of processors.

Figure 37 brings the response time results of DaC Join executing query QC. Those results show that DaC Join preserves the tendency of reducing response time even for queries with skewed features as QC.

Next, we analyze the results of experiments with n-way join queries: Q3n, Q5n and

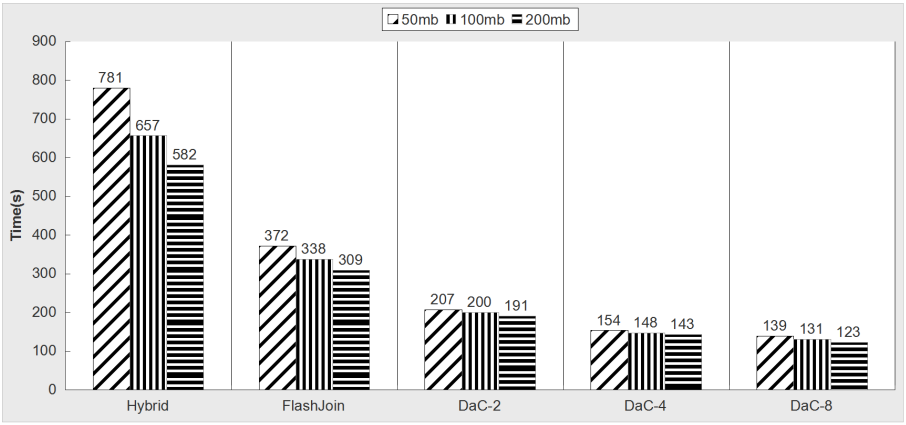


Figure 37 – Response Time for QC.

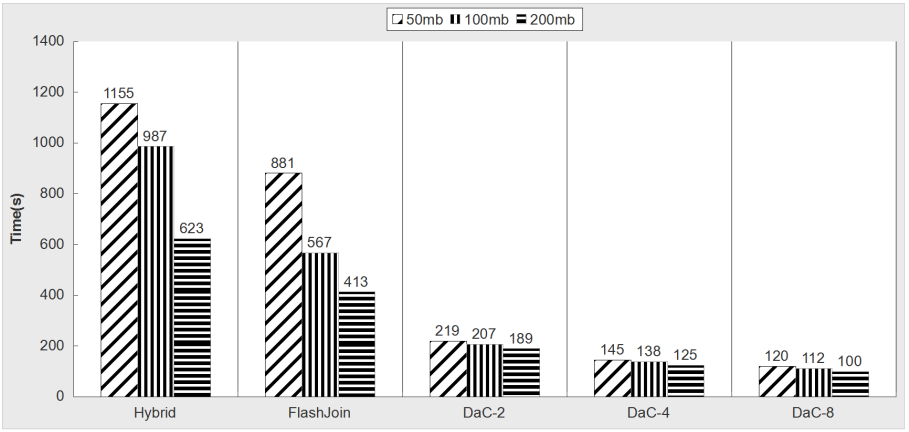
Q10n (see Table 6). The results presented in Figures 38a, 38b and 39 reinforce our hypothesis that DaC Join is quite efficient SSD-aware join operator. For queries Q3n, Q5n and Q10n, DaC Join with 2 threads and 50MB of main memory is 75.14%, 69.07% and 75.03% faster than Flash join, respectively. Running DaC Join with 8 threads to process the same set of queries, it becomes able to be 86.37%, 82.95% and 87.37% faster than Flash join, respectively. Then, we can claim that DaC Join is efficient in the context of complex queries that contain n-way joins.

Figure 38a shows that using a memory size of 200MB DaC Join with 2 threads performed Q3n (n-way join) in 189s, while Flash join ran Q3n in 413s. Then, *DaC*-2 was 54.24% faster than Flash join. Meanwhile, DaC Join with 8 threads performed Q3n in 100s. So, *DaC*-8 was 75.78% faster than Flash join.

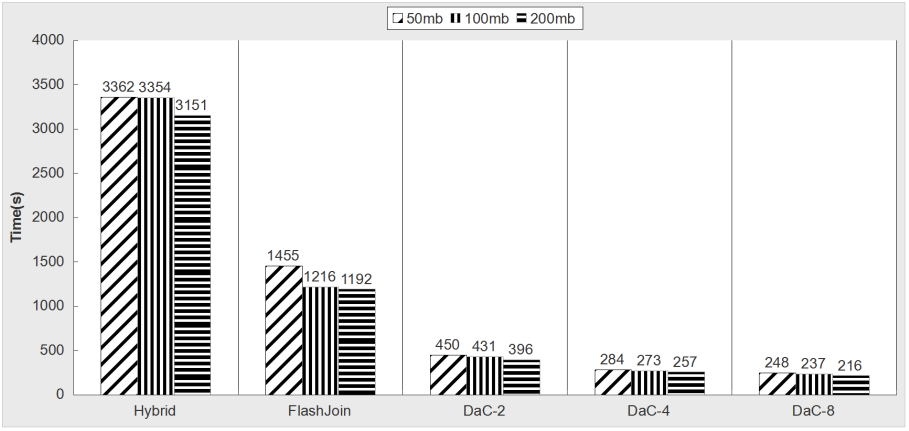
Figure 38b brings the results of experiments running Q5n, a 6-way join query. Looking more closely to that figure, one can verify that, using a memory size of 200MB (the best case for Flash join), DaC Join with 2 threads demands 396s to process Q5n. Flash join, in turn, requires 1,192s to process the same query, which means that *DaC*-2 (the worst case for DaC Join) consumes 66.78% less time than Flash join. In case we analyze the best case for DaC Join, in which it runs in 8 threads, it is able to execute Q5n in 216s, which makes *DaC*-8 81.88% faster than Flash join.

Figure 39 depicts the results of executing Q10n, which contains a join operation among 4 tables. DaC Join outperforms Flash join and hybrid hash join. For instance, using a memory size of 200MB DaC Join with 2 threads, DaC Join performed Q10n in 152s, while Flash join ran Q10n in 205s. In this case, *DaC*-2 was 25.85% faster than Flash join. DaC Join with 8 threads consumes 72s to process Q10n, thus *DaC*-8 is 64.88% faster than Flash join.

It is important to emphasize that the proposed join operator is quite efficient to be



(a) Response time for Q3n.



(b) Response Time for Q5n.

Figure 38 – Response Time for Q5n and Q3n.

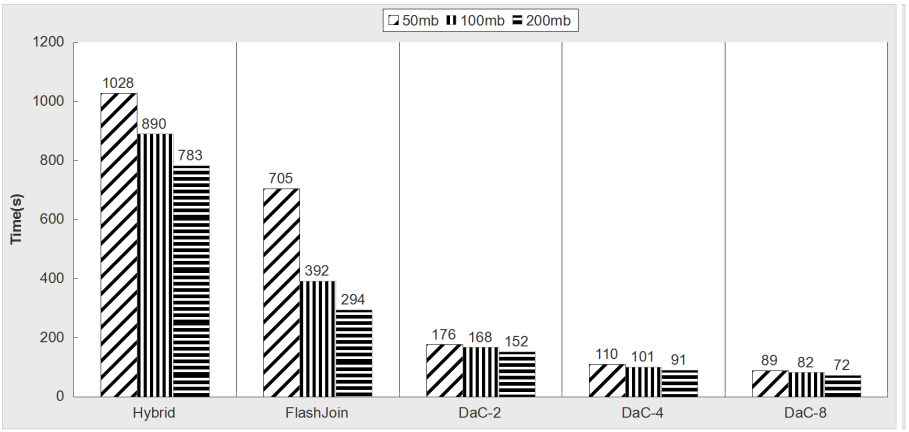


Figure 39 – Response Time for Q10n.

used in asymmetric storage devices. This is because it guarantees small response time. For that reason, client queries can run more quickly, which increases the system throughput.

8.4 Summary

This chapter has conducted a detailed analysis to different queries used to join tables. DaC Scan was compared with a sequential scan. DaC Join was compared with our implementation of Flash join and hybrid hash-join. For both algorithms, DaC Scan and DaC Join, our focus was performance and one could perceive that the characteristics of such kind of secondary memory may cause a substantial difference on the final result.

9 CONCLUSION AND FUTURE WORK

We have started this work by presenting the characteristics of the SSD memories and the hard disks. When compared, one can notice that the SSDs devices offer a very low random access time and a read/write higher speed comparing to HDD. One of the most important characteristics of the SSD devices is the speed asymmetry between reading and writing operations.

A query engine is responsible for modify and access data on a DBMS. This is one of the most important component of the DBMS and it works based in a query execution plan. A query in a high level language is translated into a LEP, which will be used to generate PEP. The query engine must choose the lowest estimated cost execution plan, and then execute the physical operators defined in the chosen PEP.

Performance is very important to a DBMS and there are some strategies that can be used for improve it. First, parallelism, in section 3.3 we have shown that some commercial DBMS are able to make use of implicit parallelism in query optimization and it may deliver a high performance for their product. Second, materialization strategy, EMS, which keep all columns that will be projected at beginning of query plan. Then, LMS, which wait as long as possible for projecting columns and finally, MMS which tries to combine the best of the two first strategies.

Traditional join algorithms are implemented under three different strategies. Loop-based is the simplest one, but also the most expensive. Sort-based works well when at least one of the table is already sorted. Hash-based approach uses a hash function to partition the tuples into buckets, then the join happens on each bucket. RARE join, Flash join and Bt-Join are join operators, which take into consideration the SSD's characteristics. Due this fact, they may perform better with this type of device.

Database systems were designed based upon two premises: The usage of HDDs for storing databases and distributed could scale beyond what a single-node of a DBMS could support. The latter premise only holds for a small number of cores in a node. Thus, database systems should be aware of upcoming CPU architectures and storage technologies in order to take profit on on-chip parallelism and high IOPS rates supported by many-core machines with SSDs.

This research brought a new scan operator called, DaC Scan, it is able to fully exploit the parallelism provided by SSDs. Thus, DaC Scan provides higher read bandwidth

when compared with sequential scan operators. The proposed scan operator decomposes a scan operation into several mini-scans and run them in parallel in different processors or threads. This is possible due to the novel concept of jump function (J), whose main goal is to infer the amount of physical blocks in a clustered block.

It is well-known that the join operation requires the highest amount of accesses (read/write operations) to storage media. In this sense, this research has also introduced a novel join operator, denoted DaC Join (*Divide and Conquer Join*). The main goal of the proposed operator is to properly take advantage of on-chip parallelism and high random IOPS rates delivered by many-core machines with SSDs as secondary memory. In this sense, DaC Join may be executed in main memory (the best case). In case there is not enough main memory, it is able to use SSDs as secondary memory.

As a future work, we could implement a pipeline, whereby two phases, of the DaC Join algorithm, can be started together, as scan phase for producing sub-hash-tables while join phase will consume them.

Increasingly servers use solid state memory. These devices are bringing a significant melting point for the DBMS's performance, and due to their peculiar characteristics, we can improve even more or take into consideration new operators. Based on that, we demonstrate with this work that, as the hardware evolves, we also need to evolve the algorithms, taking advantage of the new features that those hardware may offer to us.

BIBLIOGRAPHY

ABADI, D. S. M. D. J.; DEWITT, D. J. Materialization strategies in a column-oriented dbms. **IEEE 23rd International Conference on Data Engineering**, v. 23, 2007.

ABADI DANIEL J., S. R. M. N. H. Column-stores vs. row-stores: how different are they really? **ACM SIGMOD international conference on Management of data**, v. 1, n. 1, p. 967–980, 2008.

ALENCAR, N.; BRAYNER, A.; FILHO, J. d. A. M.; LOPES, H. Dac scan: a novel scan operator for exploiting ssd internal parallelism. **Concurrency and Computation: Practice and Experience**, CCPE, v. 29, 2017.

ALENCAR, N.; BRAYNER, A.; FILHO, J. d. A. M.; MONTEIRO, J. Dac join: A join operator for improving database performance on modern hardware. **Concurrency and Computation: Practice and Experience**, CCPE, 2019.

BALKESEN, C.; TEUBNER, J.; ALONSO, G.; ÖZSU, M. T. Main-memory hash joins on modern processor architectures. **IEEE Trans. Knowl. Data Eng.**, IEEE, v. 27, n. 7, p. 1754–1766, 2015.

BEN-GAN., I. **Parallelism in SQL Server Query Tuning**. 2011. Disponível em: https://www.itprotoday.com/sql-server/parallelism-sql-server-query-tuning. Acesso em: 17 Mar. 2020.

BRAYNER, A.; NASCIMENTO, M. Solid-state disks: How do they change the dbms game? In: **XXVIII Brazilian Database Symposium**. [*S. l.*]: SBBD, 2013. v. 1.

CHEN, D. a. K. F.; ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. **Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems - SIGMETRICS'09**, ACM, 2009.

CHEN, F.; LEE, R.; ZHANG., X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. **2011 IEEE 17th International Symposium on High Performance Computer Architecture**, IEEE, p. 266–277, 2011.

CORMEN CHARLES E. LEISERSON, R. L. R. T. H.; STEIN, C. **Introduction to Algorithms**. [*S. l.*]: The MIT Press, 1990.

CORPORATION, I. B. M. **DB2 Universal Database for iSeries - Database Performance and Query Optimization**. [*S. l.*]: IMB Corporation, 2002. v. 1.

DEWITT R. H. KATZ, F. O. L. D. S. M. R. S. D. W. D. J. Implementation techniques for main memory database systems. **ACM SIGMOD international conference on Management of data**, v. 14, n. 2, p. 1–8, 1984.

DIRIK, C.; JACOB., B. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. **ISCA '09**, ACM, p. 279–289, 2009.

EVANGELISTA, N. L.; FILHO, J. d. A. M.; BRAYNER, A.; ALENCAR, N. Bt-join: A join operator for asymmetric storage device. **SAC '15: Proceedings of the 30th Annual ACM Symposium on Applied Computing**, ACM Press, p. 988–993, 2015.

FAN, W. L. Y.; MENG., X. Optimizing database operators by exploiting internal parallelism of solid state drives. **Bulletin of the IEEE on Data Engineering**, IEEE, p. 12–18, 2014.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The complete book - Second Edition**. Department of Computer - Science Stanford University: Pearson, 2008. v. 2.

GRAEFE, G.; HARIZOPOULOS, S. Designing Database Operators for Flash-enabled Memory Hierarchies. **IEEE Data Eng. Bull.**, v. 33, n. 4, p. 21–27, 2010.

KIM, J.; SEO, S.; JUNG, D.; KIM, J.-S.; HUH., J. Advances in flash memory ssd technology for enterprise database applications. **IEEE Transactions on Computers**, IEEE, v. 61, p. 636–649, 2012. ISSN 0018-9340.

KIM, K.; KOH., G.-H. Future memory technology including emerging new memories. **Microelectronics 24th International Conference**, IEEE, v. 1, p. 377–384, 2004.

KIMM, J. K. S.; PARK, J. T. Samsung solid-state drive. v. 1, n. 1, 2013. Disponível em: https://images-eu.ssl-images-amazon.com/images/I/C17jMBe-EjS.pdf. Acesso em: 17 Mar. 2020.

KYTE, T.; KUHN., D. **Expert Oracle Database Architecture**. [*S. l.*]: Apress, 2014. v. 3.

LI V. J. S., M. G. M. C. C. C. L. Z. D.; WEIKUAN., Y. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. **Parallel and Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International**, IEEE, p. 945–956, 2012.

LI, Y.; ON, S. T.; HU, H.; XU, J. DigestJoin : Fast Join Method for Flash-based Storage Media. **MDM'09**, p. 152–161, 2009.

MAKRESHANSKI, D.; GIANNIKIS, G.; KOSSMANN, G. A. andDonald. Mqjoin: Efficient shared execution of main-memory joins. **PVLDB**, VLDB, v. 9, n. 6, p. 480–491, 2016.

MEHUL HARIZOPOULOS STAVROS, W. L. J. G. G. S. A. Fast scans and joins using flash drives. **Proceedings of the 4th international workshop on Data management on new hardware**, v. 1, n. 1, p. 17–24, 2008.

NARAYANAN, D.; HODSON, O. Whole-system persistence with non-volatile memories. **ASPLOS'12**, ACM, p. 401–410, 2012.

NGUYEN, T. T.; MINH, H. Zing database: high-performance key-value store for large-scale storage service. **Vietnam Journal of Computer Science**, v. 2, n. 1, p. 13–23, 2015. ISSN 2196-8896.

PARK, S.; KIM, Y.; URGAONKAR, B.; LEE, J.; SEO., E. A comprehensive study of energy efficiency and performance of flash-based {SSD}. **Journal of Systems Architecture: the EUROMICRO Journal**, ACM, v. 57, p. 354–365, 2011.

PARK, S. yeong; SEO, J.-Y. S. E.; MAENG, S.; LEE., J. Exploiting internal parallelism of flash-based ssds. **IEEE COMPUTER ARCHITECTURE LETTERS**, IEEE, p. 9–12, 2012. ISSN 1556-6056.

PINTO, D.; HANSON., E. **Understanding and Controlling Parallel Query Processing in SQL Server**. 2010. Disponível em: http://download.microsoft.com/download/B/E/1/BE1AABB3-6ED8-4C3C-AF91-448AB733B1AF/SQL2008R2_Parallel_QP_Understanding_and_Controlling.docx. Acesso em: 17 Mar. 2020.

SULLIVAN, T. **Diagram of a typical Solid State Drive**. 2010. Disponível em: http://www.storagereview.com/western_digital_siliconedge_blue_ssd_review. Acesso em: 17 Mar. 2020.

TSIROGIANNIS, D.; HARIZOPOULOS, S.; SHAH, M. A.; WIENER, J. L.; GRAEFE, G. Query processing techniques for solid state drives. **SIGMOD '09**, ACM, p. 59–72, 2009.

VALDURIEZ, P. Join indices. **ACM Transactions on Database Systems (TODS)**, v. 12, n. 2, p. 218–246, 1987.

YU, X.; BEZERRA, G.; PAVLO, A.; DEVADAS, S.; STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. **Proc. VLDB Endow.**, VLDB Endowment, v. 8, n. 3, p. 209–220, 2014. ISSN 2150-8097.