DaC-Join: Dividing the Problem of Joining Tables for Conquering an Efficient and SSD-aware Join Operator

Namom Alencar¹, José Maria Monteiro¹ (Orientador), Ângelo Brayner¹ (Coorientador)

¹Mestrado e Doutorado em Ciência da Computação - Universidade Federal do Ceará (UFC)

{namomalencar@gmail.com, monteiro@dc.ufc.br and brayner@dc.ufc.br}

Mês e Ano de ingresso no Mestrado: Janeiro de 2016
Mês e Ano Previsto de Conclusão do Mestrado: Janeiro de 2017
Defesa da Proposta: Agosto de 2016
Etapas Concluídas: Levantamento Bibliográfico, Apresentação da proposta, Desenvolvimento da Solução, Testes
Etapas Futuras: Validação, Publicação

Abstract. Solid state drives (SSDs) have emerged as an attractive alternative for storing large databases. A read operation on SSDs is faster than a write operation. However, database management systems (DBMSs) have been designed assuming read and write operations would be executed in the same amount of time (characteristic of hard drives - HDDs). Thus, to fully exploit benefits provided by SSDs, components of DBMSs should be aware of read/write asymmetry. So, in this paper, we present a new join algorithm, denoted DaC-Join whose key goal is to reduce the amount of write operations. DaC-Join can reduce up to 97% the amount of write operations can be up to 81% faster than FlashJoin, a well-known join operator proposed to be deployed in SSDs.

Keywords: SSD, join operator, parallel execution, database, concurrency

Resumo. As memórias de estado sólido (SSDs) tornaram-se uma alternativa muito atraente para armazenar grandes bases de dados. Neste tipo de memória, uma operação de leitura é mais rápida do que uma operação de escrita. Entretanto, os Sistemas de Gerenciamento de Bancos de Dados (SGBDs) foram criados assumindo que uma operação de leitura e escrita possuem o mesmo custo de execução (característica dos discos rígidos). Assim, para explorar plenamente os benefícios proporcionados pelos SSDs, os componentes do SGBDs precisam ser reescritos levando em consideração a assimetria entre as operações de leitura e escrita. Neste artigo, apresentamos um novo algoritmo de junção, chamado DaC-Join cujo objetivo principal é reduzir a quantidade de operações de escrita. O DaC-Join pode reduzir em até 97% a quantidade de operações de escrita e pode ser até 81% mais rápido do que FlashJoin, o principal algoritmo de junção concebido especificamente para memórias SSDs.

Keywords: SSD, join operator, parallel execution, database, concurrency
1. Introduction

Nowadays, it is well known that solid state drives (SSDs) have become an attractive alternative for storing large databases. The most evident characteristic of SSDs is the absence of mechanical parts in their assembly. Thus, SSDs have different characteristics and capabilities than hard drives (HDDs). First, SSDs provide random access time up to $10^2$ times faster than HDDs. Second, IOPS (Input/Output Operations Per Second) rates presented by SSDs may be over 10 times greater than 15K RPM HDDs [Boboila and Desnoyers 2011]. Third, write operations are much more expensive w.r.t. execution time and energy consumption than read operations on SSDs, phenomenon called read/write asymmetry. Regarding energy consumption, a write operation may consume up to 8 times more energy than a read operation [Park et al. 2011]. Fourth, the number of physical operations is far larger than the logical operations in SSDs. This is because SSDs internally run two processes to minimize the impact of read/write asymmetry, denoted wear leveling and garbage collection [Chen et al. 2009]. Fifth, SSDs have their lifetime determined by the number of write operations. Finally, SSDs present low rates of energy consumption.

Database systems (DBSs) were designed presupposing the usage of hard disks (HDD) for storing databases. Consequently, over the years several DBS components (e.g., query engine and buffer manager) have been optimized based on performance characteristics of HDDs. 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, from a databases technology perspective, simply replacing HDDs by faster SSDs may not fully exploit the capabilities of SSDs, although it can produce performance improvements.

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, the join operation is the one which requires the highest amount of accesses (read/write operations) to the (non-volatile) secondary memory. In order to illustrate that assertion, consider that $P_R$ and $P_S$ represent the size of tables $R$ and $S$ (in pages). To compute $R \bowtie S$, the grace hash join operator, for example, requires $2(P_R + P_S)$ disk accesses to build the partitions of $R$ and $S$ to be used during the probe phase. From the $2(P_R + P_S)$ disk accesses, the number of write operations to store the partitions on disk is $P_R + P_S$. Thereafter, $(P_R + P_S)$ disk accesses are required to process the hash-join algorithm’s probe phase.

From the aforementioned discussion, one can infer that an efficient hash join algorithm for SSDs should avoid write operations. We have identified that some well-known join algorithms present such a property. For example, the symmetric hash join [Willschut and Apers 1993] does not execute any write operation on secondary memory, since it creates hash partitions on main memory. The hybrid hash join operator [Dewitt et al. 1984] reduce the number of write operations on secondary memory by creating hash partitions on main memory whenever possible. Although interesting, the aforementioned join operators fall short in having their efficiency dependent on the amount of available main memory. In case that there is not enough main memory space to allocate any hash partition, hybrid hash join degrades to grace hash join [Garcia-Molina et al. 1999] and symmetric hash join does not work at all.

Recently, several SSD-aware join algorithms have been proposed in order to
take better advantage of IOPS rates provided by SSDs [Chen et al. 2011, Fan et al. 2014, Li et al. 2009, Shah et al. 2008, Tsirogiannis et al. 2009]. Most of them implements a late materialization strategy in order to inject into the intra-operator data flow (join partial results) the triple \(<\text{JoinAttribute}, \text{RID}_r, \text{RID}_s>\). By doing this, their join partial results are smaller, reducing this way the number of write operations. Because the partial results are incomplete join results, additional random read operations are necessary to fetch attribute values to correctly produce final join results. Although being SSD-aware, those algorithms suffer from critical drawbacks. Some of them have been designed to run on column-oriented databases, which is the case of the operators proposed in [Shah et al. 2008], [Tsirgiannis et al. 2009] and [Graefe and Harizopoulos 2010]. Others have been proposed to run on devices with specific physical characteristics presented by some SSD devices. For instance, ParaHashJoin runs on SDD device implementing RAID-0 data storage [Fan et al. 2014].

To properly take advantage of high random IOPS rates delivered by SSDs, this paper introduces a novel join operator, referred to as DaC-Join, which stands for Divide and Conquer Join. DaC-Join is an n-way physical join operator, implemented as a pipeline of \(n - 1\) binary joins.

For empirically validating the proposed join operator, simulations have been conducted on TPC-H benchmark (scale factor 10) database. Besides, the execution of flash join [Tsirgiannis et al. 2009] and hybrid hash [Dewitt et al. 1984] join algorithms have been simulated as well and their results compared with those achieved by DaC-Join. Overall, the results achieved point to significant gains in reducing the number of write operations, evidencing the suitability of the proposed approach.

2. Related Work

Tsirgiannis et al. present in [Graefe and Harizopoulos 2010, Tsirgiannis et al. 2009] two SSD-aware query operators: FlashScan and FlashJoin. FlashScan is a scan operator whose main functionality is to efficiently read from the SSD required attributes to process a given query. FlashJoin in turn is composed of two operators, join kernel and flash kernel. Join kernel is responsible to compute the join operation. In fact, the authors do not propose a novel join operator, since they state that the join kernel can be implemented by using any existing join algorithm (e.g., hybrid hash join). After join kernel has computed the join operation, it outputs a join index, which is forwarded to the fetch kernel. Actually, fetch kernel implements the FlashScan operator. Thus, fetch kernel is responsible to read only the required attributes for processing the next join operation (or the final result). This feature is called late materialization and has the functionality of reducing the amount of main memory used by FlashJoin. As RARE join, FlashJoin have also been designed for column-oriented databases.

Julio et al. proposes in [Tavares et al. 2012] an algorithm called SMC Join, which takes advantage of the SSD characteristics. Although, it has not been implemented, it brings some important steps that an SSD-aware join operator could follow such as: tuple compression, a new over flow approach and exploit the high speed random access on the SSD device.
3. DaC-Join: An SSD-aware Join Operator

The join operator from the relational algebra is by definition a binary operator. Let \( R \) and \( S \) be the two tables to be joined (see Figure 1a). DaC-Join is an n-way physical join operator, implemented as a pipeline of \( n - 1 \) binary (2-way) joins (\( n \geq 2 \)). Thus, DaC-Join is able to yield its output tuples as early as possible. DaC-Join is composed of two phases, multi-scan and join (depicted in Figure 1a). By doing this, we make DaC-Join able to run on several threads or processors in parallel, increasing this way intra-operator parallelism.

3.1. Multi-Scan Phase

The multi-scan phase is responsible for consuming data from two different data sources by running several scan operations in parallel, which in turn inject read data into several pipes. The number of scan operations and pipes in an n-way join is determined by the number of available processors or threads. To explain the behaviour of the multi-scan phase, consider the join operation \( R \bowtie S \) depicted in Figure 1a. 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 minimizes the probability of occurring memory overflow, reducing this way the number of write operations on secondary memory device.

3.2. Join Phase

The join phase implements a novel mechanism for computing a join operation. The main goals of the proposed mechanism is to reduce the number of write operations on the secondary storage media and 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 1a) and each super-hash-table is composed of several sub-hash-tables (the second level of the hash-table hierarchy).

3.3. Inside the DaC-Join Engine

In order to illustrate the manner DaC-Join works, let us consider the scenario illustrated in Figure 1b, where DaC-Join should compute a 3-way join, i.e., \( R \bowtie S \bowtie T \), which should be decomposed by the optimizer into two 2-way joins (the join ordering problem [Garcia-Molina et al. 1999]). Let us assume that the optimizer has defined the following join execution order \( (R \bowtie S) \bowtie T \). Thus, the multi-scan phase on tables \( R \) and \( S \) begins to scan both tables. Recall that multi-scan phase divides the scan action into several scan operations, each of which responsible for reading a portion of the operand, based on the following round-robin rule: pages \( p, p + k, p + 2k, \ldots \) from \( R \) (or \( S \)) are read by a scan operator running on a processor/thread \( p \) 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 a processor/thread 2 and are injected into pipe 2.

The hash-table building step and the multi-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 \( hR_{ti} \), where \( hR_{ti} \subseteq HT_R \), \( t \) should be inserted. After 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 \( hR_{ti} \) are exclusively compared with tuples belonging to its counterpart \( hS_{si} \).

4. Results

We have employed the TPC-H Benchmark database, with scale factor of 10, to carry out the experiments. The database size is 13GB, for which the lineitem table has been populated with approximately \( 6 \times 10^7 \) tuples.

For a fair comparison, we have implemented DaC-Join, FlashJoin and Hybrid Hash Join in Java using a non-transactional row storage engine. That 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 algorithms corresponds to a direct access on SSD. Moreover, in all experiments, the same amount of main memory has been allocated for each analyzed join operator, i.e., DaC-Join, FlashJoin and Hybrid Hash Join.

Next, we present and discuss the results of experiments with a n-way join query called Q3n. It important to emphasize that Q3n is a 3-way join query.

<table>
<thead>
<tr>
<th>Query</th>
<th>Sql Statements</th>
<th>Rows returned</th>
<th>Join Selectivity(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3n</td>
<td>select * from customer, orders, lineitem where c.mktsegment = 'building' and c.custkey = o.custkey and l.orderkey = o.orderkey and o.orderdate &lt; 1995-03-15 and l.shipdate &gt; 1995-03-15</td>
<td>302,114</td>
<td>0.000203008</td>
</tr>
</tbody>
</table>

Tabla 1. Query Q3n

4.1. Write Operations

Analyzing the results presented in Figures 2a, one can observe that for all scenarios DaC-Join (with 2, 4 and 8 threads) presents better results than hybrid hash join and FlashJoin w.r.t. the number of write operations during the execution. Thus, for processing Q3n with 50MB, FlashJoin needs to execute \( 2.92 \times 10^9 \) write operations, while DaC-Join executes \( 1.54 \times 10^9 \) write operations, a reduction of \( 1.38 \times 10^9 \) write operations.
4.2. Response Time

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

5. Conclusion

Due to the phenomenon, denoted read/write asymmetry, write operations on SSDs are more time and energy consuming than read operations. Existing database systems have been designed to run on HDDs, for which read and write operations consumes the same amount of time. In this paper, we have shown that to fully exploit benefits provided by SSDs, components of DBSs should be aware of read/write asymmetry in SSDs. It is well known that join operation is the query operator, which requires the highest amount of
read/write operations to be computed. Therefore, in this paper, we presented a novel approach for computing join operations. The proposed join algorithm, called DaC-Join, can run on multiple threads or processors in parallel and implements two dynamic hash functions, defined on the number of threads/processors. For that reason, DaC-Join minimizes the number of write operations on SSDs and maximizes intra-operator parallelism.

Referências


