DigestJoin: Expediting Joins on Solid-State Drives*

Shen Gao, Yu Li, Jianliang Xu, Byron Choi, Haibo Hu
Depart of Computer Science
Hong Kong Baptist University
Kowloon Tong, Hong Kong SAR, China
{sgao,yli, xujl, bchoi, haibo}@comp.hkbu.edu.hk

Abstract. This demonstration presents a recently proposed join algorithm called *DigestJoin*. Optimized for solid-state drives (SSDs), *DigestJoin* aims at reducing intermediate join results and hence expensive write operations while exploiting fast random reads. The demonstration system consists of an implementation of *DigestJoin* in the open-source PostgreSQL database management system on an Intel SSD. In the demonstration, we will showcase the performance benefits of *DigestJoin* in comparison to a traditional join algorithm and highlight the workloads in which *DigestJoin* is particularly favorable.

1 Introduction

Solid-State Drives (SSDs) have recently been a competitive alternative secondary storage for database applications, thanks to their superiority such as low access latency, low power consumption, and excellent shock resistance [1]. However, compared to magnetic disks, SSDs possess a number of distinct I/O characteristics, which affect database applications, among others. First, SSDs do not involve any mechanical components so that there is a negligible *seek time* in reading pages. A random read is almost as fast as a sequential read on SSDs. Second, SSDs have an *erase-before-write* constraint: a page has to be erased before it can be overwritten. Although this can be addressed by the *out-place* update strategy, new issues such as wear leveling and garbage collection arise, rendering a write slower than a read on SSDs. Third, with the short I/O latency (*e.g.*, the random read of an SSD is 150X faster than that of a magnetic disk [1]), I/O cost may no longer dominate CPU computation cost in evaluating a query on an SSD-based database system. These distinct I/O characteristics make the state-of-the-art join algorithms, which assume I/O characteristics of magnetic disks, suboptimal when implemented on SSDs.

In this demonstration, we present *DigestJoin* — a recently proposed algorithm that optimizes join performance for SSDs by reducing intermediate join results and exploiting fast random reads [2]. *DigestJoin* consists of two phases: *digest-table joining* and *page fetching* (see Figure 1). In the first phase of digest-table joining, *DigestJoin* projects the

* This work was partially supported by the Research Grants Council of Hong Kong (Grants HKBU210808 and HKBU211307) and Natural Science Foundation of China (Grant No. 60833005).

tuple id (*tid*) as well as the attribute(s) that participate in the join. The projected tables are called the *digest tables*. A traditional join algorithm is then applied on the digest tables to generate the *digest join results*. The digest join results are simply pairs of *tid*s together with the join attribute(s), thereby minimizing the size of intermediate join results. In the second phase of page fetching, based on the digest join results, *DigestJoin* loads the full tuples that satisfy the join from the original tables to produce the final join results. Whenever a tuple is fetched from disk, the entire page containing the tuple is fetched. Ideally, each page should be fetched at most once during the process of final-result construction. However, this is difficult to achieve in practice due to memory constraints. As the digest join results are not clustered with respect to page address, a page may be fetched multiple times during the construction process. Thus, a page fetching strategy is needed to minimize the number of page accesses. In the following sections, we provide more details of each of the two phases in *DigestJoin* with an example.

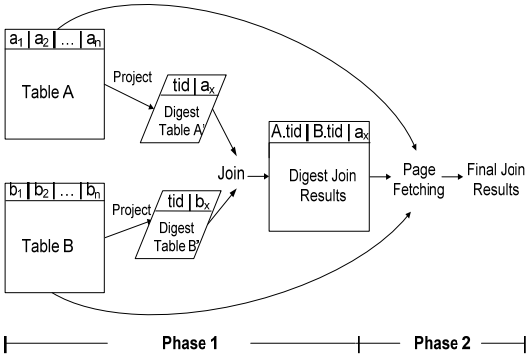


Figure 1: Overview of *DigestJoin*

Figure 2: Connecting SSD to motherboard

2 Digest-Table Joining

Consider two tables $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$. Denote the tuple ids of these two tables by *A.tid* and *B.tid*, respectively. In the first step, we scan tables *A* and *B* and compute the digest tables that contain only the join attributes and the tuple ids. For example, given a simple join $A \bowtie_{A.ax=B.bx} B$, the digest tables will be $A' = \{A.tid, A.a_x\}$ and $B' = \{B.tid, B.b_x\}$. After that, we apply a traditional join algorithm (e.g., nested-loop, hash join, or sort-merge) to the digest tables to generate the digest join results, *e.g.*, in the form of $\{A.tid, B.tid, a_x\}$ for the above example. As the digest tables are often much smaller than the original tables, the I/O of the join, especially the write operations on SSDs, would be greatly reduced.

3 Page Fetching

The digest join results consist of only the *tid*s of the tuples that satisfy the join. To produce the final join results, we fetch the tuples from the original tables according to *tid*s. The fetching is performed at page-level granularity. This has been known to be the classical *page fetching problem* in index-based joins. However, as random reads are no longer an issue on SSDs, we can simply minimize the *amount of I/O*s in fetching the full tuples. On the other hand, due to the short I/O latency of SSDs, the CPU computation cost of page fetching should also be taken into consideration.

As an illustration, one straightforward solution is to fetch the pages of the tuples as soon as they are generated in the digest-table joining phase, and cache them in a buffer for future use. Since random reads are fast on SSDs, we assign as few input buffers as possible in the digest-table joining phase in order to maximize the number of buffers for page caching. For example, suppose that sort-merge is used to join the digest tables. Page fetching is incorporated in the merge phase of the digest-table join (*i.e.*, after the digest tables have been sorted). We assign only two input buffers to merge the two sorted digest tables. The remaining buffer space forms a page cache. A cache replacement policy, LRU, is used for the management of the page cache. This page fetching strategy maximizes the amount of cached pages. Meanwhile, it does not incur much CPU computation cost. More advanced page fetching strategies for SSDs have also been proposed; interested readers may refer to [2] for details.

4 Demonstration Description

We have implemented *DigestJoin* in PostgreSQL 8.3.6 [3], an open-source database management system. We store the TPC-H tables on an Intel 80GB X25-M SSD, which are connected to the motherboard via a SATA II connection (see Figure 2). In this demonstration, we will showcase the performance benefits of *DigestJoin* in comparison to a traditional sort-merge join (*TraditionJoin*) algorithm and highlight the workloads in which *DigestJoin* is particularly favorable.

Figure 3(a) gives a screenshot of the GTK+ interface of our demonstration system. After launching the system, the user can input a join query in standard SQL form. Below is an example:

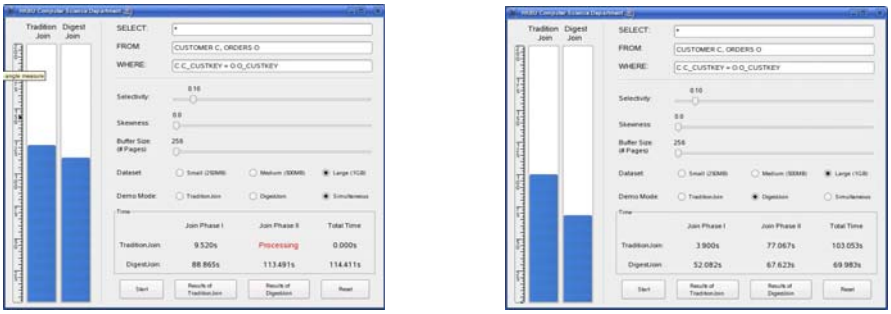
```
SELECT *
FROM CUSTOMER C, ORDERS O
WHERE C.C_CUSTKEY = O.O_CUSTKEY
```

Next, the user can set a number of parameters for execution:

- **Join result selectivity**: This adds to the WHERE clause of the user query an additional filtering function that selects part of the original join results. The selectivity can be ranged from 0.01 to 1.0.

- **Skewness of join results**: When used with the join selectivity, this parameter controls the page distribution of selected join results. When it is set at 0, the join results are evenly distributed; when it is set at 1, the selected join results are highly clustered on a few hot pages.
- **Buffer size**: This sets the size of the buffer (in terms of the number of 8KB pages) used for the join algorithm.
- **Dataset size**: Each dataset under testing has three sizes for selection: small (250MB), medium (500MB), and large (1GB).

The user can also choose an execution mode. In *simultaneous* mode (Figure 3(a)), to contrast the performance difference, *DigestJoin* and *TraditionJoin* will be executed side-by-side, and their elapsed times will be visualized in the performance bars (see the left part of Figure 3(a)) in real time. For *TranditionJoin*, the whole join process is divided into three stages: scanning and sorting the outer table (Phase I), scanning and sorting of the inner table (Phase II), and merging join results (Total Time). For *DigestJoin*, the whole process is divided into generating digest join results (Phase I), page fetching (Phase II), and generating final results (Total Time). During the execution, the "Processing" status will be displayed in the corresponding stage. After completing all stages, the total elapsed time will be reported. In an alternative *separate* mode (Figure 3(b)), the two join algorithms will be executed one after the other in order to eliminate their possible performance interference of simultaneous execution. Finally, the user can check the join results by clicking the Results buttons. An online demo video for *DigestJoin* is available at http://www.comp.hkbu.edu.hk/~db/demo.



(a) Simultaneous mode (b) Separate mode

Figure 3: User interface of the demonstration system

References

1. S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim: A Case for Flash Memory SSD in Enterprise Database Applications. Proceedings of SIGMOD, pp. 1075–1086 (2008)
2. Y. Li, S. T. On, J. Xu, B. Choi, and H. Hu: DigestJoin: Exploiting Fast Random Reads for Flash-based Joins. Proceedings of the 10th International Conference on Mobile Data Management (MDM '09), pp. 152-161 (2009)
3. PostgreSQL: http://www.postgresql.org/