DigestJoin: Exploiting Fast Random Reads for Flash-based Joins

Yu Li, Sai Tung On, Jianliang Xu, Byron Choi, Haibo Hu

Department of Computer Science, Hong Kong Baptist University
Kowloon Tong, Hong Kong SAR, China
{yli, ston, xujl, bchoi, haibo}@comp.hkbu.edu.hk

***Abstract*—Flash disks have been an emerging secondary storage media. In particular, there have been portable devices, multimedia players and laptop computers that are configured with no magnetic disks but flash disks. It is envisioned that some RDBMSs will operate on flash disks in the near future. However, the I/O characteristics of flash disks are different from those of magnetic disks. Thus, in this paper, we study the core of query processing in RDBMSs — join processing — on flash disks. Specifically, we propose a new join method, called DigestJoin, to exploit fast random reads of flash disks. DigestJoin consists of two phases: (1) projecting the join attributes followed by a join on the projected attributes; and (2) fetching the full tuples that satisfy the join to produce the final join results. While the problem of tuple/page fetching with minimum I/O cost (in the second phase) is intractable, we propose three heuristic fetching strategies. We have implemented DigestJoin on a real flash disk for performance evaluation. Experiments on TPC-H datasets show that DigestJoin clearly outperforms the traditional sort-merge join under various system configurations.**

I. Introduction

Flash disks have been widely used in portable devices such as PDAs, smartphones and multimedia players, as well as in some laptop and desktop computers in the form of Solid State Drive (SSD). When compared to their magnetic counterpart, flash disks offer comparable storage space, better data access performance, better shock resistance, lower power consumption, lighter weight, smaller dimension and better noise resistance. Given such a wide range of advantages, flash disks have been a competitive candidate for the next-generation mass storage media.

Adopting flash disks as a secondary storage media, recent research (*e.g.*, [11], [16], [17], [22]) has investigated the possibilities for developing flash-based RDBMSs. It has been remarked that flash disks have unique I/O characteristics (see Table I for a comparison of I/O operation overhead between magnetic disks and flash disks). For instance, flash-based storage does not involve any mechanical components and hence there is a negligible seek time and rotational delay in reading or writing a page on a flash disk. While flash disks still have some overhead on each I/O operation, which is caused by the encapsulated logic for such purposes as wear leveling and internal caching [4], [6], such overhead can be more than 20 times smaller than its mechanical counterpart in magnetic disks. Recall that query processing algorithms on magnetic disks often spend an effort to avoid random I/O operations but exploit sequential I/O operations whenever possible. The I/O characteristics of flash disks indicate that such an effort is no longer crucial in flash-based RDBMSs.

TABLE I
Magnetic Disk's Seeking vs. SSD's Overhead [12]

Storage	Seeking/Overhead
hard disk†	8.33 ms
flash SSD‡	0.2 ms (read)
	0.4 ms (write)

† Seagate Barracuda 7200.10 ST3250310AS, average latency for seek and rotational delay
‡ Samsung MCAQE32G8APP-0XA drive with K9WAG08U1A 16 Gbits SLC NAND chips

The state-of-the-art RDBMSs are mainly designed to operate on magnetic disks and therefore assume the I/O characteristics of magnetic disks. Among the operators of SQL (supported by all RDBMSs), joins are particularly I/O intensive and computationally expensive. Therefore, joins have been extensively studied in the literature [15]. In this paper, we focus on joins in the absence of indexes, *i.e.*, non-index joins. There have been a few existing non-index join methods, *e.g.*, nested-loop, sort-merge and hash joins. An objective is to reduce the I/O opertions in performing the join.

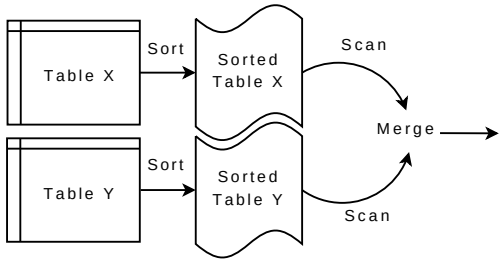

Fig. 1. Sort-Merge Join Algorithm

Consider the nested-loop join algorithm. Assume that the tables, that participate the join, do not fit into main memory. The nested loops encapsulate table scans which minimize the seek operations in performing the join. Similarly, consider the sort-merge join algorithm. It sorts the tables (using an external sort algorithm if necessary) by the join attributes. This facilitates table scans on the sorted tables in the merge phase of the algorithm, as illustrated in Fig. 1. This shows that in the absence of indexes, join algorithms have been minimizing the number of seek operations. Nevertheless, as discussed earlier,

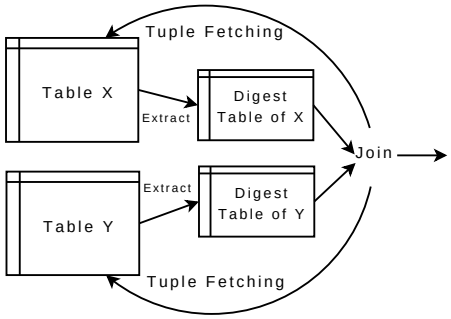

Fig. 2. Overview of DigestJoin

in the context of flash-based RDBMSs, minimizing the number of seek operations does not offer much advantage.

Regarding flash disks, what is desirable is to study how their unique I/O characteristics can be utilized to enhance the performance of joins. In this paper, we propose *DigestJoin*, a join algorithm that exploits fast random reads, among others, of flash disks. *DigestJoin* consists of two phases. In the first phase, *DigestJoin* projects the tuple id (*tid*)[1] and only the attributes that are relevant to the join operator from the tables that participate the join. The projected tables are called the *"digest" tables*. The main intuition here is that flash disks are often installed inside mobile devices with limited main memory. It is often beneficial to reduce the I/O operations required through a scan on the tables to obtain digest tables. A traditional join algorithm is then applied on the smaller digest tables to generate the *digest join results*. The digest join results are just pairs of *tid*s together with the join attributes, thereby minimizing the size of intermediate join results. It is worth noting that the digest join results are similar to join indexes [21], [20], [13] except that they are computed on-the-fly in the first phase of *DigestJoin* because we do not assume the presence of indexes on the join attributes.

The I/O cost of the join operator saved by joining the (smaller) digest tables is paid in the form of a table scan in the first phase *and* random reads in the second phase of *DigestJoin*. In the second phase, based on the digest join results, the algorithm loads the full tuples, that satisfy the join, from the original tables to produce the final join results. This has been known to be the classical *page fetching problem* in index joins. However, random reads are no longer an issue in flash disks. The only concern here is to minimize the amount of I/Os in fetching the full tuples as specified in the digest join results. Consider a digest join result, which contains the join attributes and the *tid*s of the two tuples that satisfy the join. Based on *tid*s, we can locate the full tuples from the original tables to construct the final join result. Whenever a tuple is requested, the entire page containing the tuple is fetched. Ideally, each page should be fetched at most once during the whole process of final result construction. However, this is difficult to achieve in practice (if not impossible) due to memory constraints. As the digest join results are not clustered with respect to the page address, a page may be fetched multiple times during the whole process. This is particularly true for mobile devices, where flash disks are popular. Thus, we propose a few heuristics to schedule page fetching to minimize the number of page accesses in the second phase of *DigestJoin*.

[1]Throughout this paper, we assume that the tuple id is implemented as a page id plus a slot number.

Our main contributions can be summarized as follows:

- We propose a two-phase join algorithm, called *DigestJoin*, that exploits fast random reads of flash disks. The first phase of *DigestJoin* is generic and can be integrated with any traditional non-index join algorithm.
- We propose three heuristic solutions for the page fetching problem arising from the second phase of *DigestJoin*. In particular, we consider the memory constraint in the page fetching problem.
- We evaluate *DigestJoin* with sort-merge join in an experimental RDBMS implemented on a real flash disk. By conducting experiments with TPC-H benchmark datasets, we show that *DigestJoin* outperforms the traditional sort-merge join under various system configurations.

The rest of this paper is organized as follows. In Section II, we give an overview of the *DigestJoin* method and identify its page fetching issue. Section III discusses the page fetching problem and proposes three fetching strategies. Section IV presents the results of performance evaluation. In Section V, we review the related work on flash-based data management and join processing. Finally, this paper is concluded in Section VI.

II. Overview of DigestJoin

In this section, we give an overview of *DigestJoin*. Given the I/O characteristics of flash disks, we minimize the number of page read and write operations required by a join and use sequential write operations whenever possible. As shown in Fig. 2, our proposed *DigestJoin* method is divided into two phases: *digest table joining* and *page fetching*.

Digest Table Joining. Consider two tables $X = \{Attr_{x_1}, Attr_{x_2}, \ldots, Attr_{x_m}\}$ and $Y = \{Attr_{y_1}, Attr_{y_2}, \ldots, Attr_{y_n}\}$, and use tid_x and tid_y to denote *tuple id*s of X and Y, respectively. For simplicity, we discuss $X \bowtie_{Attr_{x_1}=Attr_{y_1}} Y$. In this phase, we first compute the *digest tables* — the projected tables that contain only the join attributes and the tuple ids. In our example, the digest tables are $X' = \{Attr_{x_1}, tid_x\}$ and $Y' = \{Attr_{y_1}, tid_y\}$. Such a projection will obviously reduce much I/O cost in performing the actual join. Then, we apply a traditional join algorithm (*e.g.*, nested-loop, sort-merge, or hash join) to the digest tables to generate the *digest join results*, in the form of $\{Attr_{x_1}, tid_x, tid_y\}$. The digest join results may be written to the flash disk sequentially if it is larger than the memory size.

However, the digest join results $\{Attr_{x_1}, tid_x, tid_y\}$ only tell us which tuples satisfy the join. To output the full join results, we have to fetch the corresponding tuples from the

original tables according to tids. Fetching tuples from the original tables is usually performed at the granularity of pages in RDBMS. Thus, we have the second phase — page fetching. Although the page fetching will pay some cost of random reads, we hope that the cost is only a small part of the I/O saving gained in performing digest table joining.

Page Fetching. A careful fetching schedule of tuples is critical to the page fetching problem. To illustrate that, we give a simple example. Suppose that we have the following digest join results: $(x_1, tid_{x_1}, tid_{y_1})$, $(x_2, tid_{x_2}, tid_{y_2})$, $(x_3, tid_{x_3}, tid_{y_3})$ and $(x_4, tid_{x_4}, tid_{y_4})$, and that tuples tid_{x_1} and tid_{x_3} are stored on page A, tuples tid_{x_2} and tid_{x_4} are on page B, tuples tid_{y_1} and tid_{y_3} are on page C, tuples tid_{y_2} and tid_{y_4} are on page D. If we have sufficient memory space, we may fetch all of these four pages, namely, A, B, C and D, and keep them in the memory throughout the process of constructing the final join results. However, in practice, memory space is limited and, hence, we need to carefully schedule the page fetching to minimize the page read cost. Suppose that the memory space can hold two pages only. If we construct the final results in the order of x_1, x_2, x_3 and x_4, we need to fetch pages A and C for x_1, B and D for x_2, then A and C again for x_3, and finally B and D again for x_4. In this case, each page is fetched twice. Alternatively, we may swap the order of x_2 and x_3 in the schedule and then each page would be fetched only once. Therefore, as can be seen, the page fetching schedule has a great impact on the I/O cost. We will propose three strategies for page fetching in the next section.

Before we proceed to the next section, we present an example to exemplify how much *DigestJoin* can improve over a traditional join algorithm. Consider the following join on two TPC-H tables, where `CUSTOMER` and `ORDERS` are joined through the key `C_CUSTKEY`:

```
SELECT *
FROM CUSTOMER, ORDERS
WHERE CUSTOMER.C_CUSTKEY = ORDERS.C_CUSTKEY;
```

According to the schemas of TPC-H, the tuple size of a digest table is approximately 6% of that of the `CUSTOMER` table, and 9% of that of the `ORDERS` table. Assume that there are $10,000$ and $5,000$ pages for `CUSTOMER` and `ORDERS`, respectively. Consider the traditional sort-merge algorithm. Given a memory size of 20 pages, we need 4 and 3 passes to sort these two tables, respectively, and the total sorting cost is $4\times2\times10,000+3\times2\times5,000 = 110,000$ I/Os. The next merge procedure will alternatively scan both of the sorted tables, leading to a total join cost of $110,000 + 10,000 + 5,000 = 125,000$ I/Os. Now if we employ *DigestJoin*, the digest tables extracted from `CUSTOMER` and `ORDERS` are of size 600 and 450 pages, respectively. Thus, the cost for building these two digest tables is $10,000 + 5,000 + 600 + 450 = 16,050$ I/Os. Then, when applying the sort-merge join to the digest tables, both tables can be sorted in 3 passes, with a join cost of $3 \times 2 \times 600 + 3 \times 2 \times 450 + 600 + 450 = 7,350$ I/Os. Hence, the total cost incurred in the first phase will be $16,050 + 7,350 = 23,400$ I/Os. Thus, as long as the second page fetching phase requires fewer than $101,600$ I/Os, *DigestJoin* would outperform the traditional sort-merge join algorithm.

Discussions. The proposed *DigestJoin* method is appealing to flash-based storage media in two aspects. On one hand, it reduces the size of intermediate join results, thereby saving the I/O operations and particularly the temporary writes to flash disks during the join. This also implicitly leads to fewer erase operations on flash disks. On the other hand, *DigestJoin* is possible due to fast random reads of flash disks. In the second phase, the tuples involved in the join results are likely to be scattered over a flash disk. Thus, page fetching will incur quite a number of random read operations, which are extremely costly for magnetic disk-based storage media. For magnetic disks, the overhead of such random read operations may even exceed the I/O cost saved in the first phase. Flash-based storage media is particularly benefited from joining the digest tables, as random reads on flash disks are efficient.

III. Page Fetching Strategies for DigestJoin

An efficient page fetching strategy is important for *DigestJoin* as it minimizes the number of page accesses when fetching the full tuples from the original tables to produce the final join results. In this section, we first formulate the page fetching problem as a graph problem and discuss its hardness. Then, we propose three heuristic-based strategies to address this problem.

A. The Page Fetching Problem

In *DigestJoin*, the outputs of the first phase are the digest join results. Based on that, in the second phase, disk pages containing the full tuples (or partial tuples containing required attributes in the projection list of the input query) are fetched to generate the final join results. The *page fetching* problem is to schedule a page fetching sequence to read all tuples in the join results with minimum number of page accesses.

To analyze the page fetching problem, we formulate it as a graph problem. A join graph is used to represent the relationship between the disk pages specified by the join results. The join graph is defined as an undirected bipartite graph $G = (V_1 \cup V_2, E)$, where V_1 and V_2 denote the set of pages from the two original tables, respectively, and $E \subseteq V_1 \times V_2$ denotes the set of page-pairs specified by the join results. Specifically, for each edge $(v_a, v_b) \in E$, there exists a tuple on page v_a which joins with a tuple on page v_b. The join graph can be used to dynamically represent the remaining pages to be fetched and joined. An edge (v_a, v_b) is removed from E if pages v_a and v_b have been fetched into the main memory and the corresponding tuples on them have been joined. A vertex v is removed from G once its out-degree becomes zero.

An example of a join graph is shown in Fig. 3, where vertices 1 and 2 represent the pages from one table, while vertices a, b and c represent the pages from the other table. An edge $(1, a)$ means there is/are tuple(s) on page 1 that can be joined with tuple(s) on page a. Similarly, edges $(1, b)$, $(1, c)$,

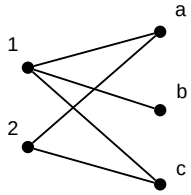

Fig. 3. Example of Join Graph

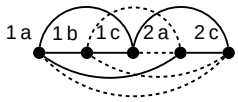

Fig. 4. Transformed Graph

$(2, a)$ and $(2, c)$ represent the "join relationship" between other pairs of pages.

A page fetching sequence is equivalent to a sequence for removing all edges of the join graph. As mentioned previously, an edge is removed if and only if the corresponding pages are fetched into the memory (and the final results are constructed). Therefore, an optimal page fetching sequence is a sequence for removing all edges in the join graph with minimum number of page accesses. If there was sufficient memory space to hold all related pages, obviously each page would be fetched once. Thus, any page fetching sequence would be optimal. However, when the memory space is limited, page swaps are needed and the problem becomes intractable.

The hardness of this problem can be illustrated as follows. We first derive another graph from the join graph. For a join graph $G = (V_1 \cup V_2, E)$, we construct a corresponding weighted complete graph $G' = (V', E')$ by the following method: for each edge $(v_a, v_b) \in E$, we create a new vertex v' in G'. That is, each edge in the join graph is transformed to a vertex in the new graph. Therefore, v' in the new graph represents fetching the pages v_a and v_b into the memory to construct the final join results. We assign a weight of each edge as the page swapping cost between two join operations implied by the vertices of the edge. For example, Fig. 4 shows the transformed graph of the join graph shown in Fig. 3. In the graph, we label each vertex by its corresponding edges in the join graph, *e.g.*, vertex $1a$ corresponds to edge $(1, a)$ in Fig. 3. Regarding the weights of the edges, we assume that the main memory can hold two pages only for analysis purposes. The weight of edge $(1a, 1b)$ (solid line) is 1 as one new page has to be fetched into the memory for performing the second join of pages 1 and b; the weight of edge $(1b, 2a)$ (dashed line) is 2 as two new pages have to be fetched for joining of pages 2 and a. With such a formulation, a page fetching sequence is equivalent to a tour of all vertices on the graph. The number of pages fetched is the sum of the weights of all edges in the tour. Thus, the page fetching problem is to find a tour in the graph with minimum cost. Then, we can reuse a result from Merrett et al. [14]. They proved that finding a Hamiltonian path from our transformed graph is *NP-complete* (Proposition 1 in [14]). We remark that Merrett et al. [14] studied a slightly different problem — determining whether there exists a solution with $n-1$ page swaps for the page fetching problem is *NP-complete* (assuming that there are totally n pages in the join graph).

When the memory space can hold more than two pages, the problem appears to be even more complex, as the weight of each edge will change dynamically. Furthermore, to the best of our knowledge, there has not been any practical approximation algorithm to address the page fetching problem [3]. In the rest of this section, we propose three heuristic strategies for the page fetching problem.

B. Naive Fetching Strategy

One intuition to tackle the page fetching problem is to fetch pages *online*, *i.e.*, as the digest join results are produced. The first strategy — called *naive fetching strategy* — is to fetch the pages of the tuples as soon as they are produced in the digest join phase. Furthermore, recall that seek operations do not incur much overhead on a flash disk. Hence, in the digest join phase, it is no longer necessary to assign as many input buffer pages as possible, *e.g.*, in the merge phase of sort-merge join and the probe phase of hash join. Most of the available memory space can be used to cache disk pages.

More specifically, we present the details of this strategy in Algorithm 1. To illustrate the intuition, we assume sort-merge join is used in the first phase of *DigestJoin*. That is, in Algorithm 1, we assume that the digest tables have been sorted and it is handling the merge phase of sort-merge join. Initially, we assign pages p_1 and p_2 (in main memory) for merging the two digest tables. The remaining memory is assigned to form a page cache. Next, a traditional cache replacement policy, such as LRU, is applied to the management of the page cache. The while loop in the algorithm constructs the final results as the digest join results are produced.

```
Input  : Sorted digest table dt_1, sorted digest table dt_2
Output: Join results

Allocate buffer p_1 for dt_1, p_2 for dt_2
Allocate rest buffers as PageCache with LRU policy
while joining dt_1 and dt_2 with p_1 and p_2 do
    if there is a join result (join_attribute, tid_1, tid_2) then
        page_1 = the page id containing tid_1
        page_2 = the page id containing tid_2
        if page_1 in PageCache then
            Extract tuple with tid_1 to t_1
        else
            Load page_1 into PageCache, extract tuple with
            tid_1 to t_1
        if page_2 in PageCache then
            Extract tuple with tid_2 to t_2
        else
            Load page_2 into PageCache, extract tuple with
            tid_2 to t_2
        Join t_1 and t_2 and then output
```

Algorithm 1: Naive Page Fetching

C. Page-based Fetching Strategy

Although the page fetching problem is *NP-hard*, there have been some special cases where the problem can be solved efficiently, *e.g.*, the case with sufficient memory space to hold all pages of the join results. Another case is that the digest join results are clustered with respect to the page address and the full tuples of the join results are clustered with respect to the join attributes. In this case, directly applying the naive fetching strategy would result in an optimal page fetching

sequence which ensures each page is fetched at most once. This motivates us to propose the *page-based fetching strategy*.

Specifically, we build two kinds of temporary tables to assist page fetching. The first one is called *fetching instruction table*, which archives digest join results. After this table is filled with all digest join results, we sort its digest join results based on their page addresses. Thus, fetching tuples based on such sorted digest results avoids duplicated page fetching requests. However, tuples fetched according to their page addresses are generally not clustered on the join attributes. Hence, we have another temporary table called *join candidate table* to store the tuples fetched according to the fetching instruction table. Sort-merge join or hash join algorithm can then be applied on this table for producing the final join results.

To illustrate the page-based fetching strategy, let us recall the example on digest join results used in Section II. In practice, we may encode the tids in the form of "*page_id:slot_number*" when we generate the digest tables. Here, the results are $(x_1$, A:1, C:1), $(x_2$, B:1, D:1), $(x_3$, A:2, C:2) and $(x_4$, B:2, D:2). Two fetching instruction tables are then created: ft_1 = {A:1, B:1, A:2, B:2},ft_2 = {C:1, D:1, C:2, D:2}. We sort them according to *page_id* and obtain ft_1 = {A:1, A:2, B:1, B:2}, ft_2 = {C:1, C:2, D:1, D:2}. The tuples now can be fetched efficiently. We then obtain two join candidates tables — $jct_1 = \{x_1, x_2, x_3, x_4\}$, $jct_2 = \{y_1, y_2, y_3, y_4\}$. By joining them again, we obtain the final join results.

The page-based strategy gives an efficient page fetching sequence with the cost of introducing some extra I/O cost for maintaining two temporary tables and another extra final join. We summarize the above discussions in Algorithm 2.

Input : Sorted digest table dt_1, sorted digest table dt_2
Output: Join results

while *joining* dt_1 *and* dt_2 **do**
  if *there is a join result* $(join\_attribute, tid_1, tid_2)$ **then**
    Output tid_1, tid_2 to fetching instruction tables ft_1 and ft_2
Sort ft_1 and ft_2 based on *page id*
while *scanning* ft_1 *and* ft_2 **do**
  Fetch clustered tuples according to tids and output the tuples to join candidate tables jct_1 and jct_2
Perform sort-merge join or hash join on jct_1 and jct_2

Algorithm 2: Page-based Fetching

D. Graph-based Fetching Strategy

Instead of archiving the digest join results in temporary tables, an alternative method is to archive it with the join graph. If the memory can hold all digest join results in the form of a join graph, we may find good heuristics to travel all edges to do the page fetching and joining near optimally. In the literature, two heuristics [3], [18] are in that direction, focusing on page fetching for index-based joins. The basic idea is to select a subgraph of a vertex which contains all its adjacent edges and requires the fewest non-resident pages to be fetched. Here, a non-resident page is a page that is not currently cached in the main memory. In other words, we check all subgraphs in the current join graph: 1) to identify whether each subgraph contains all edges of any vertex in the subgraph; and 2) to count how many vertices (*i.e.*, pages) are not in the main memory. Then, the subgraph that satisfies the first condition and has the smallest value of the second quantity is selected. As iterating all subgraphs of the join graph makes the computational cost prohibitively high, an approximation method is to select the vertex with the fewest non-resident neighbors, together with all its neighbors in the join graph, called *segment* [3].

For example, let us reconsider the join graph in Fig. 3. The candidate *segments* are $\{1, a, b, c\}$, $\{2, a, c\}$, $\{a, 1, 2\}$, $\{b, 1\}$ and $\{c, 1, 2\}$. If there has not been any page fetched into the main memory yet, we should select $\{b, 1\}$. This is because it has the fewest non-resident neighbors and implies the least page accesses. Let us assume that some pages have already been fetched into the main memory, say c. Then, we may select $\{2, a, c\}$ or $\{b, 1\}$. The reason is that both pages 2 and b have only one non-resident neighbor, *i.e.*, pages a and 1, respectively.

However, there are some challenges in applying this heuristic to our scenario. First, we observe that the devices installed with flash disks often have a limited memory space. In our research, we study joins on large tables that are larger than the size of available memory. Therefore, the memory space is not likely to hold the entire join graph for the analysis mentioned earlier. Furthermore, in order to avoid producing duplicate join results, we have to attach tids to the edges in the join graph in implementation. This is a rather subtle point. Consider the scenario where only *page id*s are attached to the join graph. Suppose we have an edge in the join graph indicating a join between page a of table X and page b of table Y. When the memory space runs out, we conduct the join on pages a and b to reclaim space by removing the corresponding vertices and edges from the join graph. Next time, when another digest join result requiring page a to join with page b comes, duplicate join results will be generated. As there is no information on what results have already been produced, attaching tids to the edges can address this issue. However, it makes the size of the join graph grows faster and aggravates memory shortage.

Second, pre-allocating sufficient memory for caching pages for the join graph is not feasible in general. The heuristic we discussed above provides a rough bound on the maximum required size of the page cache, *i.e.*, $min\{sizeof(X), sizeof(Y)\}+1$, when joining tables X and Y. However, if we could reserve so many pages for the page cache, we can simply load one table into the memory and use the remaining memory space to scan the other table to perform the join. Therefore, we need a careful implementation to effectively hold both the join graph and the page cache in a limited memory space.

In this subsection, we propose a *graph-based fetching strategy* which aims at achieving acceptable performance even when the memory available is highly limited. Fig. 5 gives an overview of the memory management for the graph-based

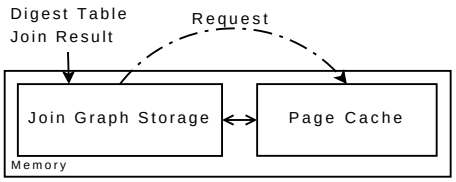

Fig. 5. Memory Management of Graph-based Fetching Strategy

fetching strategy. We divide the available memory space into two parts: one for storing the join graph (*i.e.,* join graph storage) and the other for caching fetched pages (*i.e.,* page cache). We dynamically manage the memory space for the join graph storage and the page cache. When a digest join result comes, if there is space, we directly add it into the join graph. Otherwise, we try to adjust the space for the join graph based on its *required storage size* (RSS) and *required cache size* (RCS). The RSS of a join graph is equal to the number of pages that are required to hold this join graph. We organize the join graph in an adjacency list, and the RSS is larger than the size of the adjacency list, as extra space is needed to store the tids for the edges. The RCS of a join graph is the minimum cache size for fetching and joining any segment of this join graph. Initially, the join graph only takes up one page and the page cache spans the rest of the memory. During the join process, we dynamically adjust the sizes of the join graph and the page cache. Specifically, when we cannot insert a join result from the digest table into the join graph, we perform the following:

- If $RCS <$ the current page cache size, we enlarge the memory space for join graph storage (correspondingly, shrink the space of the page cache by removing some cached pages) to insert that result.
- Otherwise, we try to select a segment of the join graph to load full tuples and join them using the page cache. After that, the size of the join group is reduced. We then insert the pending join result into the join graph, and check RSS to see whether we need to enlarge the space for the page cache (correspondingly, shrink the memory space of the join graph).

We summarize the graph-based fetching strategy in Algorithm 3. And we discuss the main functions used as follows:

- *RequiredStoreSize()*: This function computes the storage requirement of the current join graph. Note that the requirement is measured in pages, since only page-oriented memory management is often supported in RDBMSs.
- *SelectSegment()*: This function selects a segment from the join graph. We follow the same approximation in [3], which has been discussed in the first paragraph of this subsection. Since there may exist edges having both vertices (*i.e.,* pages) in the page cache, we also select those edges in order to join as many pages as possible.
- *FetchAndJoin()*: This function accepts a segment of the join graph, fetches the corresponding pages from the original tables, and joins the tuples indicated by tids. When the page cache is fully occupied, we have to swap some pages out. In particular, we have to decide the order to fetch pages in the selected segment and select the victim pages in the page cache to be swapped out. In our implementation, we select to load the page in the segment with the largest resident degree, which is defined to be the number of its in-cache neighbors in the join graph; we select the page in the cache with the smallest non-resident degree as the victim, where the non-resident degree of a page is defined as the number of its not-in-cache neighbors in the join graph.
- *RequiredCacheSize()*: Following the upper-bound analysis in [3], we assign the value of RCS as the maximum vertex degree d_{max} in the current join graph plus one. However, as we tend to select more edges than the approximation in practice, we adjust the value of RCS with a constant k. Thus, we have the value of RCS to be $k \times (d_{max} + 1)$. The value for k can be decided via experiments.

Input : Sorted digest table dt_1, sorted digest table dt_2
Output: Join results

Allocate page p_1 for dt_1, p_2 for dt_2
Allocate rest pages for *PageCache* and *JoinGraph*, and set *JoinGraph* initial size to be one page
while *joining* dt_1 *and* dt_2 *with* p_1 *and* p_2 **do**
  if *there is a join result* $(join\_attribute, tid_1, tid_2)$ **then**
    Try join tuples tid_1 & tid_2 only with *PageCache*
    if *join succeeds* **then**
      continue to next digest join result
    Try add $(join\_attribute, tid_1, tid_2)$ to *JoinGraph*
    if *fail* **then**
      /* insufficient space for JoinGraph */
      rcs = **RequiredCacheSize**(*JoinGraph*)
      if $rcs <$ *current cache size* **then**
        Increase *JoinGraph* size by one page
        Decrease *PageCache* size by one page
        Add $(join\_attribute, tid_1, tid_2)$ to *JoinGraph*
      else
        seg = **SelectSegment**(*JoinGraph*)
        FetchAndJoin(seg)
        Update in-cache flags of *JoinGraph*
        Remove seg from *JoinGraph*
        Add $(join\_attribute, tid_1, tid_2)$ to *JoinGraph*
        rss = **RequiredStoreSize**(*JoinGraph*)
        if $rss >$ *current JoinGraph size* **then**
          Decrease *JoinGraph* size by one page
          Increase *PageCache* size by one page

Algorithm 3: Graph-based Fetching

IV. Performance Evaluation

In this section, we present the performance evaluation results of our proposed *DigestJoin* method. In what follows, we first describe the experiment setup including the algorithm implementation, test dataset and system settings. Then, we compare *DigestJoin* with the traditional sort-merge join algorithm and evaluate different page fetching strategies under various system configurations.

A. Experiment Setup

We implemented *DigestJoin* with all three page fetching strategies in an experimental database system built on top of 16GB Mtron MSD-SATA3025 SSD. The experimental database system is designed to enable an easy evaluation on the performance of different join algorithms. It is composed of three components: *raw storage manager*, *page-oriented buffer manager* and *query executor*. The raw storage manager maintains a bunch of pre-allocated continuous space on the SSD and provides a page-oriented read/write interface. The other two components rely on this interface to perform I/O access to the SSD. The page-oriented buffer manager maintains a fixed number of memory pages, each of which has the same size as the ones in the raw storage manager. Any query processing in the query executor should interact with the memory pages in the buffer manager. The query executor provides facilities to implement and execute specific join algorithms. The system catalog maintains some statistics information, such as system parameters (*e.g.*, page size, memory size, etc.), table summaries (*e.g.*, # tuples, # pages, etc.), as well as join statistics (*e.g.*, join selectivity).

Tables are organized on page-oriented space of the SSD. The data tuples are stored on the pages following a row-based storage scheme. Since we study non-index joins, we do not build any index for each table. To save space, we store data tuples in a variable-size format. Finally, the data tuples are imported into the storage in a random order. This is to simulate a general-case join where the join attribute values could be in any arbitrary order in the original tables.

The test dataset is taken from the TPC-H benchmark. In particular, we use one `CUSTOMER` table of TPC-H and perform a natural self-join through the key `C_CUSTKEY`. The selectivity is thus 100%. In our implementation, we install a filtering function before we are about to get a join result. The filtering function will flip a coin to decide whether to drop the result or not. By controlling the flipping probability, we can simulate different degrees of selectivity. In particular, we can also control the selectivity on a page basis to simulate skewed join distributions.

We use the sort-merge join algorithm as a representative to evaluate the *DigestJoin* method, while expecting that similar performance results can be observed for nested-loop join. We have not compared the hash join since its performance depends on the quality of the hash function and more on the distribution of the join values. Specifically, the algorithms under evaluation are: traditional sort-merge join (Basic), *DigestJoin* with naive page fetching strategy (Digest(Naive)), *DigestJoin* with page-based fetching strategy (Digest(Page)), and *DigestJoin* with graph-based fetching strategy (Digest(Graph)).

We run all the experiments on a desktop PC equipped with a Core 2 Quad Q6600 CPU and 4GB main memory. The system parameters used in the evaluation are listed in Table 3. The results of various join algorithms are compared with the I/O cost measured in seconds, where the I/O cost is the sum of time used in all kinds of I/O operations including sequential reads/writes and random reads/writes.

TABLE II
PARAMETER SETTINGS

Parameter	Setting
Page size	4 KB
Ratio of tuple size to digest entry size (p)	10
Average tuples per page (t)	25
Table size	32 MB (*i.e.*, 8,192 pages)
Join selectivity	0.005∼0.5 (0.05 by default)
Memory size	256∼8,192 pages (512 by default)

B. Parameter k of Graph-based Page Fetching Strategy

In the first set of experiments, we determine the proper configuration of the parameter k for the graph-based page fetching strategy. Recall that k is a parameter to determine the required cache size of a join graph. Fig. 6 shows the I/O results when k is varied from 0.5 to 4. As can be seen, the value k should not be set smaller than 1. When it is set at 0.5, the memory space for caching fetched pages is too small, which leads to frequent page swaps. On the other hand, when the parameter k is increased from 1 to 4, the I/O cost is almost the same. The indicates that for the page cache, it is enough to reserve as many pages as the size of the segment we may select. Reserving more memory only could restrict the join graph not to grow large enough and thus slightly degrade the performance of page fetching. Therefore, we set k at 1 for the graph-based page fetching strategy in the rest evaluation of the *DigestJoin* method.

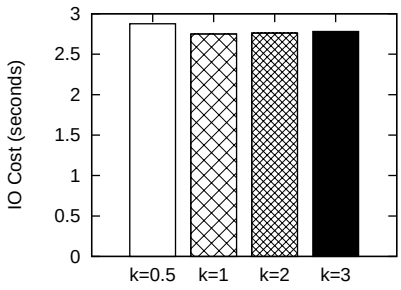

Fig. 6. Impact of Parameter k for Graph-based Fetching Strategy (Selectivity = 0.05, Memory Size = 512 Pages, Uniform Join Distribution)

C. Impact of Join Selectivity

We now compare different join methods and page fetching strategies, and investigate how the join selectivity would affect their performances. Fig. 7 and Fig. 8 show the comparison results with join selectivity varying from 0 to 0.1 and from 0.1 to 0.5, respectively. The results do not include the cost to output the final join results, since usually they are not output to the secondary storage and this cost is the same for all join algorithms. As such, the result of Basic remains the same over different selectivity settings. It is used as the baseline in the performance analysis.

As can be seen from Fig. 7, Digest(Naive) outperforms Basic only when the selectivity is very low (*i.e.*, < 0.08). When the selectivity is higher than 0.08, as the naive page fetching strategy results in too many duplicate page accesses, it becomes worse than Basic. Similarly, Digest(Page) has the best performance among all join algorithms under a low selectivity, but its performance degrades when the selectivity increases. When the selectivity is higher than 0.3 (see Fig. 8), it performs worse than Basic. Overall, Digest(Graph) offers the best performance and outperforms Basic by 15%-64% for all selectivity setting. As predicted, because of the increased page fetching cost, its performance improvement becomes smaller when the selectivity increases.

D. Effect of Graph-based Fetching Strategy

Fig. 9 shows the effect of our graph-based fetching strategy by comparing it with the original heuristic in [3] (for short, Original). Specifically, we implemented Original by allowing it to use as much memory as possible to store the join graph. When the join graph is larger than the memory available, we process some edges by following the heuristic. Since most of memory has already been occupied by the join graph, we have to load the pages on demand. The original heuristic is not expected to be efficient and we can see that, in Fig. 9, after the selectivity is higher than 0.2, it is not only worse than Digest(Graph), but also even worse than the traditional sort-merge join algorithm (Basic). On the other hand, when the selectivity is small (< 0.2), Original is as efficient as Digest(Graph). This is because a low selectivity makes the join graph smaller enough to be held in memory, and since Original and Digest(Graph) follow nearly the same heuristic, their performances are similar.

E. Impact of Page Size

Fig. 10 and Fig. 11 show the performance comparison with page size varying from 4 KB to 32 KB. By default, the page size is 4 KB, which is the smallest page size supported by our SSD; and the memory space available for joins is 2 MB (512 pages). When varying the page size, we maintain the same amount of memory (*i.e.*, 2 MB). Hence, there are 256/128/64 memory pages that could be used when the page size is 8/16/32 KB, respectively. From Fig. 10 and Fig. 11, the page size has a different impact on the join algorithms. *Basic* and *Digest(Page)* become better when the page size increases. The reason is that a big page size implies fewer pages of tables, and the external sort can be done in fewer runs for Basic and Digest(Page). Digest(Graph) also benefits from a big page size. This is because fewer pages of tables result in a smaller join graph, and thus we can make the in-memory join graph more informative, which leads to better page fetching schedules. On the other hand, Digest(Naive) performs worse when the page size increases, and it is even worse than Basic when the page size is bigger than 8 KB with a low selectivity (see Fig. 10). This is because no matter how few the pages are, the number of fetching requests remains the same under a fixed selectivity. As such, when the cost of fetching a page become higher due to a bigger page size, Digest(Naive) becomes worse.

F. Impact of Memory Size

Fig. 12 shows the performance comparison with memory size ranging from 256 to 8192 pages (*i.e.*, 3.1% to 100% of the table size). As expected, the I/O cost decreases for all algorithms when more memory space is available. When the memory space is large enough (*i.e.*, 100%) to hold the entire join table, Basic has the best performance. However, such situation rarely occurs in practice, especially when considering database applications on flash-based mobile devices, which often have very limited memory space.

It can be also observed that when the memory size increases from 3.1% to 100% of the table size, the performance gap between Digest(Page) and Digest(Graph) becomes larger (from 6% to 13%). This is because a large memory space is more helpful for the page-based strategy to join the candidate join tables, thereby achieving a more efficient memory usage.

G. Impact of Join Result Distribution

In the previous experiments, all algorithms are evaluated under the setting where join results are uniformly distributed over the disk pages. In this subsection, we evaluate their performances when the join results follow non-uniform distributions. Specifically, the join results are skewed over the disk pages based on a Zipf distribution. The Zipf distribution is controlled by a skewness parameter of θ. When θ is 0, the distribution is uniform. The larger is the value θ, the more skewed is the distribution. We apply the Zipf distribution to one table only, which emulates the case that one table is joined with a part of another table, *e.g.*, some day's orders are joined with the whole customer table.

Fig. 13 shows the results under the Zipf distribution. Due to the skewed distribution of join results, the selectivity cannot be very high. We set it to be 0.1. We make two observations from Fig. 13. First, when θ becomes larger, *DigestJoin* has a better performance improvement over Basic. Furthermore, Digest(Naive) outperforms Basic in most cases. This is because fewer pages are to be fetched when join results have high skewed distributions. Second, from Fig. 13, Digest(Graph) saves more I/O cost than Digest(Page) when θ is larger. This is because in that case the join graph is generally small and hence an effective page fetching schedule is more likely to be achieved.

H. Impact of Flash I/O Characteristics

In this subsection, we investigate the impact of read/write speeds on the performances of the join algorithms. We use a scaling factor λ to simulate the relative speed of read/write operations. Specifically, we set $Read = \overline{Read}/\lambda$ and $Write = \overline{Write} * \lambda$, where $\overline{Read}$ and $\overline{Write}$ are the read/write speeds of the Mtron SSD. With λ set larger than 1, the read becomes even faster than the write. With λ set smaller than 1, the read might be slower than the write. In the previous performance evaluation, besides the I/O time, we also recorded the number

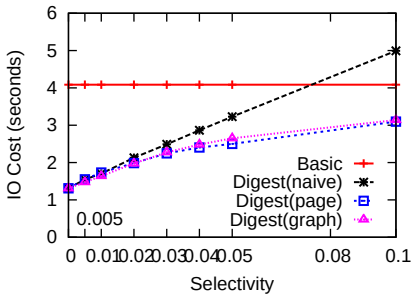

Fig. 7. DigestJoin vs. Traditional Sort-Merge Join under Low Selectivities (Memory Size = 512)

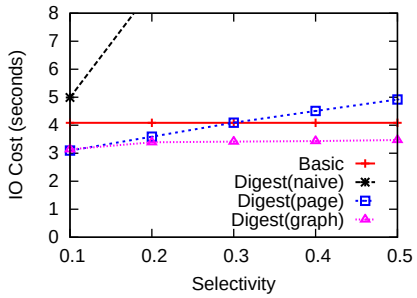

Fig. 8. DigestJoin vs. Traditional Sort-Merge Join under High Selectivities (Memory Size=512)

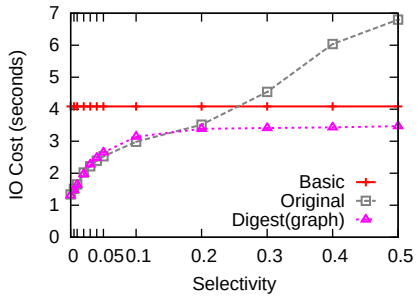

Fig. 9. DigestJoin vs. Original Implementation (Memory Size=512)

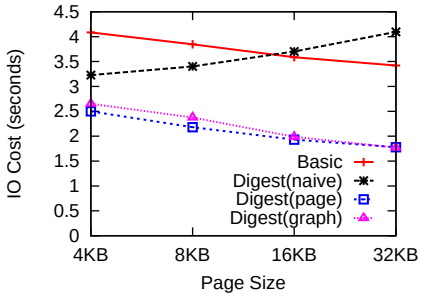

Fig. 10. DigestJoin vs. Sort-Merge Join under Different Page Sizes (Selectivity=0.05, Memory Size=2M)

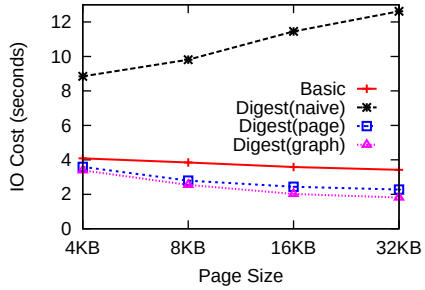

Fig. 11. DigestJoin vs. Sort-Merge Join under Different Page Sizes (Selectivity=0.2, Memory Size=2M)

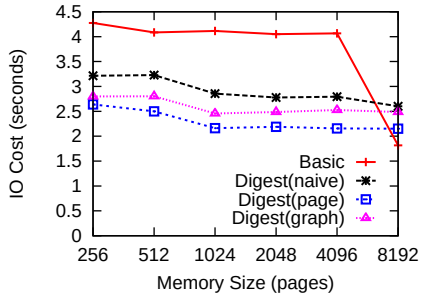

Fig. 12. DigestJoin vs. Traditional Sort-Merge Join under Various Memory Sizes (Selectivity=0.05)

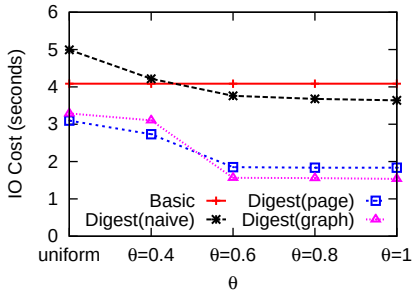

Fig. 13. DigestJoin vs. Sort-Merge Join under Zipf Distribution (Selectivity=0.1, Memory Size=512)

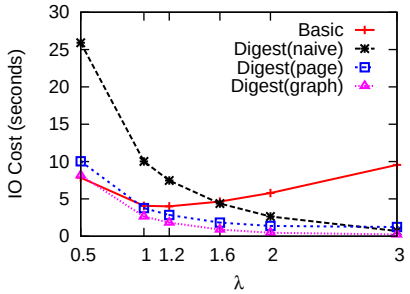

Fig. 14. DigestJoin vs. Sort-Merge Join under Flash I/O Characteristics (Selectivity=0.05, Memory Size=512)

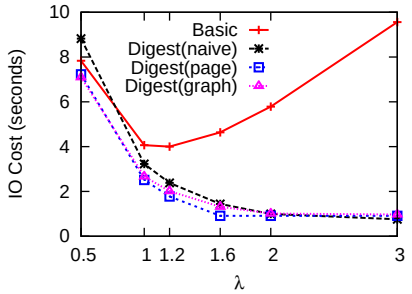

Fig. 15. DigestJoin vs. Sort-Merge Join under Flash I/O Characteristics (Selectivity=0.2, Memory Size=512)

of random reads, random writes, sequential reads and sequential writes. Based on such data, we plot the estimated I/O cost under different settings of λ in Fig. 14 and Fig. 15, where the selectivity is set at 0.05 and 0.2, respectively.

We observe that when the relative read/write speed becomes larger (*i.e.*, when λ becomes larger), *DigestJoin* demonstrates a better improvement over Basic. The reason is that with *DigestJoin*, we reduce the table size to sort and thus save a lot of write operations. When the write cost tends to dominate the overall performance, the advantage of *DigestJoin* becomes more obvious. Another interesting point is that, with a faster read operation (*e.g.*, λ = 3), the performances of different page fetching strategies converge. The reason is that, as page fetching involves read operations only, its cost becomes negligible when the read speed is extremely fast.

V. Related Work

Relational database management on flash-based storage media has attracted increasing research attention in recent years. Early work focused on how to assemble flash chips to simulate traditional hard disks [9], [5], [10] and how to extend the lifetime of flash disks [4], [6], [10]. Based on these research efforts, recent work have exploited the characteristics of flash disks to enhance the performance of RDBMSs. In view of the asymmetric read/write speed and the erase-before-write limitation, Wu et al. [22] proposed a log-based indexing scheme for flash memory. Observing that the log-based indexing scheme is not suitable for read-intensive workload on some flash devices, Nath and Kansal [17] developed an adaptive indexing method that adapts to the workload and the underlying storage device. Lee and

Moon [11] presented a novel storage design called in-page logging (IPL) for RDBMSs. Lee et al. [12] investigated how the performance of standard RDBMS algorithms are affected when the conventional magnetic hard disks are replaced by flash disks. Shah et al. [19] presented a fast scanning and joining method by adapting the PAX storage model [1] to flash disks, which appears the most related work to our study. The main difference, however, is that our work utilizes fast random reads to optimize traditional join algorithms, while PAX presented in [1] is an alternative scheme for storing relations on flash disks.

Join has been one of the important query operators in RDBMSs. Extensive research efforts have been spent on the optimization of join processing. Mihra and Eich [15] surveyed a number of join algorithms and their implementations. In this paper, we focus on exploring the possibility of further improving non-index join algorithms by utilizing fast random reads of flash disks. In particular, our proposed algorithm is inspired by join indexes and page fetching. The idea of join indexes [21], [20], [13] is to precompute join results and record pairs of tuple ids that satisfy the join to speed up future join requests. We find that this idea is useful even when we generate the join indexes on demand for joins on flash disks, *i.e.*, the first phase of *DigestJoin*. The problem of determining an optimal page fetching schedule in the second phase of *DigestJoin* is actually found in index join algorithms. Specifically, index join algorithms first compose a list of tuple pairs that participate in the join by using indexes, and then tuples themselves have to be fetched to construct the final results [2], [7]. Merrett et al. [14] proved the decision problem of optimally scheduling the page fetching to be *NP-complete*. A number of heuristics have been developed, *e.g.*, [8], [18] and [3]. We adapt the ideas behind these heuristics to design page-based and graph-based strategies for page fetching by taking into consideration the memory space constraint.

VI. Conclusion

In this paper, we have proposed a new join method called *DigestJoin* by exploiting fast random reads of flash disks. *DigestJoin* is a generic join method as its implementation could invoke any traditional join algorithm. We have identified a critical issue called page fetching for *DigestJoin*. Three heuristic-based strategies, namely, naive, page-based and graph-based fetching, have been proposed. We have also implemented *DigestJoin* with these page fetching strategies in an experimental database system on top of a real flash disk. The evaluation results show that *DigestJoin* generally improves the traditional sort-merge join algorithm under various system parameter settings. In particular, *DigestJoin* achieves more performance improvement for the scenarios where the read speed is much faster than the write speed. Among those page fetching strategies, the graph-based strategy offers the best overall performance, whereas the page-based strategy wins under a low selectivity and uniform join distribution.

Regarding future work, we plan to extend this study along a few directions. Firstly, we may extend our experiments to analyze both CPU and IO costs of *DigestJoin*. Secondly, to explore the possibilities of "backward compatibility", we may study the performance of *DigestJoin* on magnetic disks. Finally, we may also optimize the current implementation of our prototype used in the experiments. These extensions may make *DigestJoin* more appealing when compared to traditional join algorithms.

Acknowledgment

The authors are grateful to Prof. Qiong Luo for her constructive comments on an early version of this paper. This work was supported by the Research Grants Council of Hong Kong (Grants HKBU210808 and HKBU211307) and Natural Science Foundation of China (Grant No. 60833005).

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB '01*, pages 169–180, 2001.

[2] M. W. Blasgen and K. Eswaran. Storage and access in relational databases. *IBM System Journal*, 16(4):363–377, 1977.

[3] C. Y. Chan and B. C. Ooi. Efficient scheduling of page access in index-based join processing. *IEEE Trans. on Knowl. and Data Eng.*, 9(6):1005–1011, 1997.

[4] L. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC'07*, pages 1126–1130, 2007.

[5] L. Chang, T. Kuo, and S. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *Trans. on Embedded Computing Sys.*, 3(4):837–863, 2004.

[6] Y. Chang, J. Hsieh, and T. Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC'07*, pages 212–217, 2007.

[7] J. M. Cheng, D. J. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, and Y. Wang. An efficient hybrid join algorithm: A db2 prototype. In *ICDE*, pages 171–180, 1991.

[8] F. Fotouhi and S. Pramanik. Optimal secondary storage access sequence for performing relational join. *IEEE Trans. on Knowl. and Data Eng.*, 1(3):318–328, 1989.

[9] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[10] H. Kim and S. Lee. A new flash memory management for flash storage system. In *COMPSAC'99*, page 284, 1999.

[11] S. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD '07*, pages 55–66, 2007.

[12] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD*, pages 1075–1086, 2008.

[13] Z. Li and K. A. Ross. Fast joins using join indices. *The VLDB Journal*, 8:1–24, 1999.

[14] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *VLDB'81*, pages 488–498, 1981.

[15] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[16] D. Myers. On the use of nand flash memory in hgh-performance relational databases. Master's thesis, MIT CSAIL, Cambridge, MA, December 2007.

[17] S. Nath and A. Kansal. Flashdb: Dynamic self-tuning database for nand flash. Technical Report MSR-TR-2006-168, Microsoft Research, 2006.

[18] E. R. Omiecinski. Heuristics for join processing using nonclustered indexes. *IEEE Trans. Softw. Eng.*, 15(1):18–25, 1989.

[19] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *DaMoN08*, pages 17–24, 2008.

[20] P. Valduriez. Optimization of complex database queries using join indices. *Database Engineering*, 9(16):10–16, Dec. 1986.

[21] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[22] C. Wu, T. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.