Fast Scans and Joins using Flash Drives

Mehul A. Shah   Stavros Harizopoulos   Janet L. Wiener   Goetz Graefe

HP Labs
{firstname.lastname}@hp.com

ABSTRACT

As access times to main memory and disks continue to diverge, faster non-volatile storage technologies become more attractive for speeding up data analysis applications. NAND flash is one such promising substitute for disks. Flash offers faster random reads than disk, consumes less power than disk, and is cheaper than DRAM. In this paper, we investigate alternative data layouts and join algorithms suited for systems that use flash drives as the non-volatile store.

All of our techniques take advantage of the fast random reads of flash. We convert traditional sequential I/O algorithms to ones that use a mixture of sequential and random I/O to process less data in less time. Our measurements on commodity flash drives show that a column-major layout of data pages is faster than a traditional row-based layout for simple scans. We present a new join algorithm, *RARE-join*, designed for a column-based page layout on flash and compare it to a traditional hash join algorithm. Our analysis shows that RARE-join is superior in many practical cases: when join selectivities are small and only a few columns are projected in the join result.

1. INTRODUCTION

With the ever increasing disparity between main memory and disk access times, enterprise applications are hungering for a faster non-volatile store. In this paper, we explore how to leverage one such promising technology, flash drives, for data analysis applications.

Driven by the consumer electronics industry, flash is becoming a practical non-volatile storage technology. Flash drives are ubiquitous in cameras, cell-phones, and PDAs. Major PC vendors are shipping laptops with flash drives. Moreover, flash is starting to make its way into the enterprise market. For example, vendors such as SimpleTech, Mtron, and FusionIO are selling flash-based solid-state drives aimed at replacing SCSI drives and entire disk arrays. But, are flash drives an effective replacement for traditional disks?

Flash drives have several traits that make them attrac-

tive for read-mostly enterprise applications such as web-page serving and search. Table 1 compares flash drives to disks. Flash drives offer more random read I/Os per second (1500 to 100,000 IO/s), offer comparable sequential bandwidth (20-80 MB/s), and use a tenth of the power (0.5 W). Flash is cheaper than DRAM ($\sim$\$18/GB) and is non-volatile. Moreover, flash continues to get faster, cheaper, and denser at a rapid pace. In particular, NAND flash density has doubled every year since 1999 [13].

Unfortunately, flash offers little or no benefit when used as a simple drop-in replacement for disk for data analysis workloads in databases. Traditional query processing algorithms for data analysis are tuned for disks; they stress sequential I/O and avoid random I/O whenever possible. Thus, they fail to take advantage of the fast random reads of flash drives.

In this paper, we investigate query processing methods that are better suited for the characteristics of flash drives. In particular, we focus on speeding up scan (projection) and join operations over tables stored on flash. Our algorithms use a mixture of random reads and sequential I/O. When only a fraction of the input (rows and columns) are needed, these algorithms leverage the fast random reads of flash to retrieve and process less data and thereby improve performance.

To make scans and projections faster, we examine a PAX-based page layout [2], which arranges rows within a page in column-major order. When only a few columns are projected, this layout avoids transferring most of the data while incurring the cost of "random" I/Os to seek between different columns. We explore the tradeoff between row-based and PAX-based layouts on flash experimentally. Our results show that a PAX-based layout is as good or better even at a relatively small page size of 64KB, a size that works well with traditional buffer management.

We then present a new join algorithm, called RARE-join (RAndom Read Efficient Join), that leverages the PAX-based layout. RARE-join first constructs a join index and then retrieves only the pages and columns needed for computing the join result. We show both analytically and using times from our scan experiments that this join outperforms traditional hash-based joins in many practical cases: when join selectivities are small and only a few columns are projected in the join result [6]. Although the specific methods we leverage in our algorithms have been previously studied, motivating their use and applying them in the context of flash storage is our main contribution.

In the next section, we give an overview of flash tech-

	NATA Disk	USB Flash	IDE Flash	FC Flash
GB	500	4	32	146
$/GB	$0.20	$5.00	$15.62	
Watts (W)	13	0.5	0.5	8.4
seq. read (MB/s)	60	26	28	92
seq. write (MB/s)	55	20	24	108
ran. read (IO/s)	120	1,500	2,500	54,000
ran. write (IO/s)	120	40	20	15,000
IO/s/$	1.2	75	5	
IO/s/W	9.2	3,000	5,000	6,430

Table 1: Disk and Flash characteristics from manufacturer specs or as measured where possible. Prices from online retailers as of May 16, 2008. SATA-disk: Seagate Barracuda; USB-Flash: Buffalo; IDE-Flash: Samsung 2.5" IDE; FC-Flash: STech's ZeusIOps 3.5" FibreChannel.

Figure 1: Row and PAX page layout

nology. Section 3 describes our experiments with scans and projections. Section 4 describes our join algorithm and compares it to traditional join methods. In section 5, we present the related work and then we conclude in Section 6.

2. FLASH CHARACTERISTICS

There are two types of flash available: NAND and NOR. NAND flash is typically used for data storage, and NOR is typically used in embedded devices as a substitute for programmable ROM. Since current solid state drives are typically composed of NAND flash, we focus on NAND.

Table 1 summarizes the relevant characteristics of current flash drives compared to disks.[1] Along with the conventional metrics, the table also lists the random I/O rate per dollar (IO/s/$), which measures the drive's price-performance, and the random I/O rate per Watt consumed (IO/s/W), which measures the drive's energy-efficiency. Although flash drives are more costly per gigabyte, they well outperform disk drives on metrics such as IO/s, IO/s/$ and IO/s/Watt.

NAND flash is typically organized into blocks of 128KB, which are the minimum erase units, and these blocks are subdivided into pages of 2KB. Once erased, all bits in the block are set to "1". Subsequently, selected bits can be programmed or set to "0" at a finer granularity. There is no generic rewrite option. Thus, unoptimized random writes are slow because they typically involve a read, erase (which is slow), and program.

Currently, most NAND flash is limited to about 100,000 erase-write cycles per block. To avoid premature failure, most flash drives include wear leveling logic that remaps writes to evenly update all blocks. With wear leveling, writing continuously to a 32GB drive at 40 MB/s would cause the drive to wear-out after 2.5 years. Since most drives are not fully utilized, this typically implies a lifespan of 5-10 years, which is acceptable.

3. SCANS AND PROJECTIONS

Relational scans and projections return some of the columns for some of the rows in a table. Since "seeks" are relatively

---

[1]Although we do not quote a price for the ZeusIOps drive, enterprise flash drives like this one are significantly more expensive in terms of $/GB.
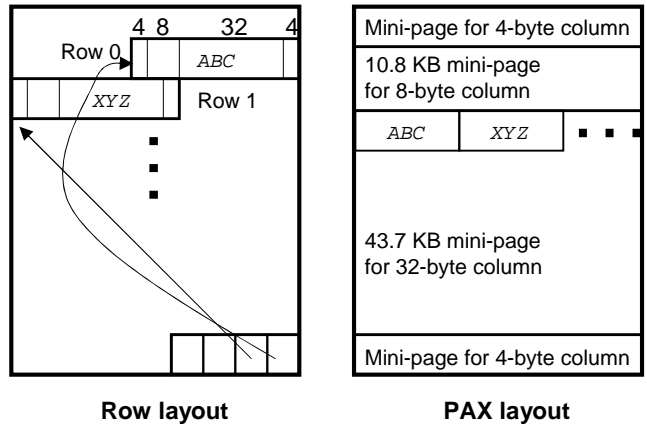
cheap on flash, it can be cost-effective to introduce additional seeks instead of reading data not needed in the query. In this section, we consider the PAX page layout, which is efficient for reading one column at a time but requires a seek to skip over columns.

In Section 3.1, we describe the PAX page layout, discuss the drawbacks for using PAX to reduce disk I/O, and show why it is suitable for flash. In Section 3.2 we present our implementation and experimental results that verify the benefits of our approach.

3.1 PAX on Flash

Most commercial relational DBMS use a row-based page format where entire rows are stored contiguously, as shown on the left side of Figure 1. A slot array of 2-byte slots at the end of the page contains pointers to the start of each row. In this example, the table has four columns of sizes 4, 8, 32, and 4 bytes; the row size is 48 bytes. The page size is 64 KB. A full page contains 64 KB / (48+2) bytes = 1310 rows. We ignore page headers here for simplicity.

In contrast, the PAX (partition attributes across) layout [2] creates *mini-pages* within each page. The rows of the page are vertically partitioned among the mini-pages. Each mini-page stores data for a single column. Each mini-page for a fixed-length column is an array of column values with an entry for every row on that page; the ith entry on each mini-page is the column value for row i. Mini-pages for variable-length columns use slot arrays. The right side of Figure 1 shows this layout for the same example table. The 4-byte columns get 64 KB * (4/48) = 5460 byte mini-pages; the 32-byte column's mini-page is 43,680 bytes. Since no slot arrays are needed for these fixed length columns, the page with PAX layout holds data for 1365 rows.

In the original PAX proposal, Ailamaki et al. argued for a PAX layout to improve CPU cache utilization when scanning a subset of the columns [2]. They did not, however, consider a change to disk I/O access patterns. Here, we consider how a PAX layout can be used with flash to reduce total data transferred and thereby improve performance for scans.

With the row-based layout, a scan query that needs only a subset of the table columns must retrieve all of the columns of the table. With the PAX layout, however, a scan query can read only the required columns by "seeking" to the next column's mini-page (when the columns are not adjacent).

When the time spent seeking from one mini-page to the next is less than the time to read the unneeded mini-pages between them, performing the random read (seek) is better.

Enterprise disk drives can read sequential pages at around 100 MB/s and a short seek takes about 3-4 ms. Therefore, a seek to skip mini-pages on disk must skip at least 300-400 KB (100 MB/s × 3-4 ms) to be worthwhile. If mini-pages are 300 KB, then full pages must be multiple MB. However, the "right" page size in a relational DBMS reflects many other factors, such as buffer pool size, buffer-cache hit ratios, update frequency, and per-page algorithmic overheads. Unfortunately, these and other economic considerations [4] lead most commercial RDBMSs to use much smaller page sizes, typically between 8 KB and 64 KB.

Although the PAX layout does not improve read performance for disks, it is worthwhile on flash. The seek overhead on flash is much smaller. For the IDE flash drive in Table 1 with 28 MB/s sequential read bandwidth and 0.25 ms seek time, it makes sense to skip mini-pages of only 7 KB (28 MB/s × 0.25 ms). Full pages can therefore be only 32-128 KB. With a 32 KB (or larger) page, a scan query that projects less than three-quarters of a table will complete faster using a PAX layout. Since mini-pages are not always aligned at page-size boundaries of flash pages; some extra data will be read when bringing a single column into memory. In the next section, we verify these numbers experimentally.

Column stores also partition the table data by columns to allow fast access to a subset of the columns [14]. Our query processing methods apply to such column layouts as well as to PAX layouts on flash. However, a column layout has two important limitations when compared to PAX layout. First, many parts of a traditional database engine, such as the storage layer, I/O subsystem, buffer pool, recovery system, indexes, some operators, and so on, expect and operate on fixed size pages. Thus, a column layout requires touching all these components and effectively redesigning the engine. A PAX layout, on the other hand, only requires reimplementing the storage layer methods that retrieve data from a page, since only the page organization has changed from a traditional row layout, not the page contents. Second, column stores may require multiple I/Os to update multiple columns of a single row. With a PAX layout, only one I/O is needed, since all columns are stored on the same page. We therefore investigate performance with PAX layouts since they involve a less disruptive change.

3.2 Experiments

In our implementation, we use tightly packed mini-pages. We read data in multiples of 4 KB, even if the needed mini-page is less than 4KB. For our experiments, we modified the code released in [6], which is a bare-bones high-performance storage manager. This code implements relational scanners on a single-threaded C++ code base, using Linux's Asynchronous I/O capabilities to issue multiple outstanding I/O requests and overlap computation with disk transfer time.

We evaluate a single scan query of the form "select O1, O2, .. from ORDERS where predicate(O1)" in which the predicate yielded 10% selectivity. The ORDERS table is loosely modeled after TPC-H data. We simplify the schema: the table contains eight 4-byte integers for a row length of 32 bytes; we create 60 million rows, for a total table size of about 2 GB. We ran the experiments on the Samsung
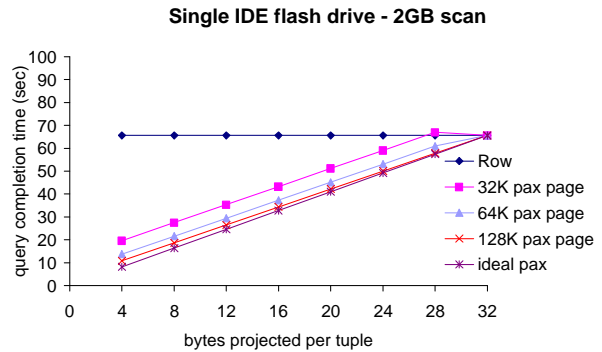

Single IDE flash drive - 2GB scan

Figure 2: Comparing row vs. PAX layouts for scans and projections.

32 GB IDE flash drive (from Table 1) formatted with the Linux ext3 file-system. The baseline case is a scan of the entire table, which corresponds to the performance of a row store.

Figure 2 compares the performance of scans as we vary the number of columns projected for three different page sizes: 32 KB, 64 KB, and 128 KB. For all page sizes, the fewer columns the query projects, the better it performs. The "ideal" curve assumes the true read bandwidth of the IDE drive and no "seek" delays. As we increase the page size (and thus each mini-page read is larger), the overhead of seeking is amortized. PAX pages of 32 KB are as good as the baseline (row layout) case no matter how many columns are read, and better when up to 88% of the data are read. PAX pages of 64 KB always outperform the baseline case and come close to ideal performance.

4. JOINS

In this section, we show how to leverage the PAX layout to compute joins. We analytically compare a traditional Grace-hash join to a new join algorithm, RARE-join, that incurs additional random I/Os to save total data accessed.

4.1 RARE-join

Our new join algorithm, called the RARE-join (RAndom Read Efficient join), has two main conceptual steps.

- It computes a join index by accessing only the join columns of the input tables.

- It adapts Li and Ross' jive-join [10] for PAX layouts on flash. Jive-join uses a join index to compute the join result in a single read pass through the input.

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 savings comes at the cost of increased "seeks" and computing the join index, which we show can be worthwhile when using flash.

To describe RARE-join and illustrate its benefits, we compare it with a well-known hash-based join: Grace-hash [8]. Traditional Grace-hash operates over a row layout. We also include a simple variant, Grace-PAX, which operates over a PAX layout; the underlying scans access only the mini-pages for columns projected in the result. We analyze two basic modes for these joins: when there is enough memory

to compute the join in one pass through the input, and when more passes are needed. We also analyze an important, in-between mode for RARE-join: when it makes one pass over the input but must materialize the join index.

For each algorithm, we assume that I/O cost dominates runtime and give its costs in terms of the number of I/Os required. We assume that the costs of a sequential read, a sequential write, and a random read are the same. To correct for this simplifying assumption, we adjust the I/O costs in our examples using measurements from Section 3 where possible and appropriate. Table 2 shows the notation that we use for the pseudocode and cost equations.

Symbol	Meaning
T_1	Table 1
R	Join result
J_1	Join column of T_1
V_1	Remaining columns of T_1 projected in result
id_2	Row-id of join result from Table 2
I_2	Temp file filled with id_2
JI	Temp file holding join index
M	Memory available for join
h	Hash-table overhead
σ_{p1}	Fraction of T_1 pages needed for computing the join

Table 2: Notation used for cost equations. T_2 is always the smaller table and its symbols are analogous to those for T_1. The $|X|$ notation specifies the number of page I/Os for X.

4.2 One-pass joins

Grace hash can compute the result in one simple pass over the input if a hash-table on T_2, the smaller table, can fit in memory, i.e. $h|T_2| < |M|$, It first reads and builds a hash-table on T_2. Then it reads T_1, probes the hash-table, and spools the results to R. All accesses are sequential and the total I/O cost is simply:

$$|T_1| + |T_2| + |R| \qquad (1)$$

Grace-PAX is better for two reasons. First, it needs less memory to operate in one pass because only the join and projection columns of T_2 must fit in memory, $h(|J_2|+|V_2|) < |M|$. Second, since it skips the unneeded columns, the total I/O cost is less:

$$|J_1 + V_1| + |J_2 + V_2| + |R| \qquad (2)$$

In roughly the same memory as a 1-pass Grace-PAX, a 1-pass RARE-join, shown in Figure 3, reduces the I/O cost further. RARE-join reads and builds a hash-table on the join column, J_2, and row-id, id_2. Then, it probes the hash-table with J_1. Unlike Grace, it fetches only those mini-pages necessary to produce the join result. For all matches, RARE-join fetches and pins the mini-pages containing row id_2 from V_2 and fetches the mini-pages containing row id_1 from V_1. Since it scans in T_1 order, the new V_1 mini-pages can immediately replace old ones while the V_2 pages are buffered. RARE-join spools the result to R. More precisely, the memory requirement is: $h(|J_2| + |id_2|) + \sigma_{p2}|V_2| < |M|$, and the total I/O cost is:

```
1. Read J₂ and build hash-table
2. Read J₁ and probe hash-table
foreach join result <id₁, id₂> do
   Read projected values of row id₁ from V₁
   Read projected values of row id₂ from pinned V₂
   mini-pages else from flash
   Write result into R
```

Figure 3: 1-pass RARE-join: when the hash-table on J_2 and needed mini-pages of V_2 fit in memory.

```
1. Read J₂ and build hash-table
2. Read J₁ and probe hash-table
foreach join result <id₁, id₂> do
   Read projected values of row id₁ from V₁
   /* R and I₂ are both partitioned by id₂   */
   Write projected values into partition of R
   Write id₂ into partition of I₂
3. Read I₂ and process it.
foreach partition of I₂ do
   foreach id₂ in partition do
      Read projected values of row id₂ from V₂
      Write values into partition of R
```

Figure 4: $(1+\epsilon)$ pass RARE-join: when the hash-table on J_2 and output buffers fit in memory.

$$|J_1| + \sigma_{p1}|V_1| + |J_2| + \sigma_{p2}|V_2| + |R| \qquad (3)$$

Thus, given sufficient memory for the 1-pass case, RARE-join outperforms Grace-PAX which outperforms Grace in our cost model. In reality, however, the advantages depend upon the overheads for each I/O of mini-pages and the "page" selectivity. Depending on these parameters, we can adapt RARE-join to make it behave more like Grace-PAX: fetch V_2 with J_2 or V_1 with J_1 or both.

4.3 More than 1 pass

If there is not enough memory to hold the hash-table on T_2 for Grace or on J_2 and V_2 for Grace-PAX, both degrade into a two pass algorithm. The first pass partitions both tables on the join column such that the runs of the smaller table fit into memory. This pass involves a read and write. The second pass reads each partition into memory and computes the join. Thus, the total I/O cost for Grace is:

$$3(|T_1| + |T_2|) + |R| \qquad (4)$$

and likewise for Grace-PAX:

$$3(|J_1 + V_1| + |J_2 + T_2|) + |R| \qquad (5)$$

Most joins will need at most two passes with flash, since the outgoing buffer size can be small, e.g., 64 KB. With 2 GB of main memory, there is room to create 32,000 partitions. Therefore, a two-pass Grace join suffices for T_2 up to 65TB, which is much larger than the size of current flash drives.

4.3.1 $(1+\epsilon)$ pass RARE-join

RARE-join has more flexibility than Grace and thereby provides improved performance. If J_2 fits in memory, but

Name	Address	Age	Team
Ben	18 Main St	7	Orange
Julie	21 Iris Ln	8	Red
Sam	110 Hays Dr	7	Green
Sarah	2 Main St	7	Blue
Alex	90 Primrose	8	Red
Lena	44 Madison	7	Orange

Figure 5: Player Table.

Team	Field	Time	Row Id
Red	Terman	1	1
Orange	Ohlone	9	2
Orange	Carmelo	3	3
Blue	Briones	2	3

Figure 6: Game Table.

V_2 does not, RARE-join can still compute the result with one pass through the input, but must materialize the join index. This $(1 + \epsilon)$ pass RARE-join is shown in Figure 4. To illustrate the algorithm, we step through it for the example join query: "select name, team, time from player, game where player.team = game.team;" using the Player and Game tables shown in Figures 5 and 6.

As in the 1-pass case, RARE-join first builds a hash-table on J_2 and probes it with J_1. Figure 7 shows the hash-table for our example.

The result of probing the table in step 2 is the join index, which has one entry for each row in the join result R. Since the probes occur in J_1 order, the necessary V_1 mini-pages can be read sequentially and written directly to the result R. Since we use a PAX layout, we write only the V_1 columns to R and leave the portion of each page for V_2 columns "blank" until step 3. Note, this write pattern for R is efficient on flash since we pay the cost of the erase in this phase and "program" the V_2 values in step 3.

Unlike in the 1-pass case, we cannot fit V_2 in memory. A simple approach is to read the needed V_2 mini-pages on demand. This approach, however, might retrieve the same mini-pages more than once since we generate results in J_1 order. Instead, like jive-join, we partition the join index into runs in step 2, so that the V_2 mini-pages referenced in each run can fit in memory. Actually, jive join [10] creates sorted runs of the join index, which it then merges so that it can later fetch the rows of T_2 sequentially. We borrow from this idea, but observe that, with flash, we need not access V_2 in sequential order. We only need to ensure that all values needed from a single page are obtained with one page read.

Therefore, in step 2, we simply partition the join index by the T_2 page number (encoded in id_2), so all row ids for the same T_2 pages go in the same partition. We need not materialize the id_1 column of the join index since V_1 is streamlined to the result in step 2. Thus, the I_2 partitions only contain id_2 values and are implicitly in T_1 order.

For step 3, the entire set of V_2 pages in a partition must fit in memory at once. The number of partitions needed is therefore $|V_2|/|M|$ and the partitioning function can be either a hash or range partitioning scheme. We partition R the same way so that in step 3, we can fill in the blank parts of R with corresponding V_2 values, one partition at a time. After step 3, we combine the pages from all partitions of R

Blue, 3
Red, 1
Orange, 2 → Orange, 3

Figure 7: $(1 + \epsilon)$ RARE: hash-table on Game.Team

Figure 8: $(1 + \epsilon)$ pass RARE, step 2: I_2 partitions.

into a single file simply by linking them together. Figures 8 and 9 show the contents of the partitions in our example and Figure 10 shows the final result table.

Since we need one buffer page for each partition of R and I_2 in step 2 and there are at most $|V_2|/|M|$ partitions, the memory requirement is $h(|J_2| + |id_2|) + 2(|V_2|/|M|) < |M|$. The total I/O cost of all three steps is:

$$|J_1| + \sigma_{p1}|V_1| + |J_2| + \sigma_{p2}|V_2| + |R| + 2|I_2| \qquad (6)$$

Combining the previous two equations, RARE outperforms Grace-PAX when:

$$2|J_1| + (3 - \sigma_{p1})|V_1| + 2|J_2| + (3 - \sigma_{p2})|V_2| > 2|I_2| \qquad (7)$$

The left hand side is savings from reading the join columns and only the needed mini-pages of V_1 and V_2 once instead of three times. This savings must outweigh the additional cost of materializing and reading the row-ids id_2 from the join index.

We illustrate the potential benefits of RARE-join with the following example. Suppose T_1 and T_2 each have 8 columns of 4 bytes and the join result contains only 3 columns from each, i.e. 5 total with the common join column. Let T_1 and T_2 contain 256 million rows (8 GB) apiece. Further suppose half the rows in T_1 each match one row in T_2 and the page selectivities are 1. Also, assume a system with 2 GB of memory. We can then estimate the savings using the performance numbers from Section 3 as follows.

Both V_1 and V_2, which hold two columns, are 2 GB, and J_1 and J_2 are 1 GB each. This setup puts Grace-PAX in the two-pass mode and RARE-join in the $(1+\epsilon)$ pass mode. Assuming row-ids are 4 bytes, R and I_2 each will have 128M rows and be 2.6 GB and 512 MB, respectively. Assuming we use 64 KB pages, reading J_1 (one column) takes 55.1 s, reading V_1 (two columns) takes 86.5 s, and reading $J_1 + V_1$ (three columns) takes 117.8 sec; the transfer times are the same for J_2 and V_2. Note, these account for the mini-page "seek" overheads as measured in Section 3. We estimate that writing R takes 91.4 s and one pass through I_2 takes 18.3 s. Therefore, Grace-PAX will take $3(117.8 \times 2) + 91.4 = 798.2$ s while RARE-join will take $2(55.1 + 86.5) + 91.4 + 2 * 18.3 = 411.2$ s, a savings of 387 s or speedup of 1.94x. The savings from making only a single pass through the input is 423 s, and the penalty for reading and writing the join index is only 36.6 s.

4.3.2 Two-pass RARE-join

Figure 11 shows the pseudocode for RARE-join when J_2 and the outgoing buffers do not fit in memory. In this case,

Julie	Red
Sarah	Blue
Alex	Red

Ben	Orange
Ben	Orange
Lena	Orange
Lena	Orange

Figure 9: $(1+\epsilon)$ **pass RARE, step 2: partitioned** R

Julie	Red	1
Sarah	Blue	2
Alex	Red	1
Ben	Orange	9
Ben	Orange	3
Lena	Orange	9
Lena	Orange	3

Figure 10: $(1+\epsilon)$ **pass RARE, end: Result,** R

steps 1 and 2 are similar to those in Grace-hash join. RARE-join hash partitions the join column of both tables so that each J_2 partition can fit in memory. In step 3, it computes and materializes the join index $< id_1, id_2 >$ for each partition. Note that within each partition, the join index is ordered by id_1.

In step 4, RARE-join merges the partitions of JI into T_1 order and fetches the needed projection columns V_1. It spools the result and id_2 values to partitions of R and I_2, as in the $(1+\epsilon)$ pass algorithm. Then, step 5 is the same as step 3 of the $(1+\epsilon)$ RARE algorithm. Note, the join index is exactly twice the size of I_2, since it contains id_1 and id_2. Again, RARE-join fetches only the needed mini-pages of V_1 and V_2, but makes multiple passes over the join columns. The total cost is therefore

$$3|J_1| + \sigma_{p1}|V_1| + 3|J_2| + \sigma_{p2}|V_2| + |R| + 6|I_2|. \quad (8)$$

Two-pass RARE-join therefore beats two-pass Grace-PAX when

$$(3 - \sigma_{p1})|V_1| + (3 - \sigma_{p2})|V_2| > 6|I_2| \quad (9)$$

The left hand side is the savings from only accessing the needed pages of V_1 and V_2 once instead of thrice. The right hand side is the penalty for materializing and passing over the join index multiple times.

We again illustrate the savings from RARE-join with an example. Suppose T_1 and T_2 have the same schema as in the previous example but are four times larger, 32 GB each. Also, consider the same join query as above with the same row and page selectivities. In this case, J_1 and J_2 are 4 GB each, and V_1 and V_2 are 8 GB each. These input sizes place both RARE-join and Grace-PAX in the two-pass mode. The result has 512M rows, with 5 attributes of 4 bytes each. Thus, R is 10.2 GB and I_2 is 2 GB. Assuming the same performance as above, reading J_1 (one column) takes 220 s, reading V_1 (two columns) takes 346 s, and reading $J_1 + V_1$ (three columns) takes 471.2 s. We estimate writing R takes 366 s, and one pass through I_2 takes 73.1 s. Therefore, Grace-PAX will take $3(471.2 \times 2) + 366 = 3193$ s while RARE-join will take $3(220 \times 2) + (346 \times 2) + 366 + (6 \times 73.1) = 2817$ s, a savings of 376 s or speedup of 1.12x. The penalty for I/O on the join index was 439 s, but the savings from making only one pass through the projected columns was 815 s. A more selective query would only improve the RARE-

1. Read J_2 and partition it (hash on join value)
2. Read J_1 and partition it (same hash function)
3. Compute JI
foreach *partition of J_2* **do**
    Read J_2 and build hash-table
    Read partition of J_1 and probe hash-table
    foreach *row in join result* **do**
        Write id_1, id_2 in JI partition
4. Merge partitions of JI on id_1
foreach *join result* $<id_1, id_2>$ **do**
    Read projected values of row id_1 from V_1
    /* R and I_2 are partitioned by id_2 */
    Write projected values into partition of R
    Write id_2 into partition of I_2
5. Read I_2 and process it.
foreach *partition of I_2* **do**
    foreach *id_2 in partition* **do**
        Read projected values of row id_2 from V_2
        Write values into partition of R

Figure 11: Two-pass RARE-join: when the hash-table on J_2 and output buffers do not fit in memory.

join performance relative to Grace-PAX.

Extending RARE-join for more passes is analogous to extending Grace-hash, so we omit the description here.

4.4 Discussion

Although we believe our cost model is sufficient to highlight the potential benefits of RARE-join, we still need to implement and measure its benefits. We need to measure its true performance and map out the tradeoffs in comparison to Grace, Grace-PAX, and more sophisticated variants of hash-based joins. There are a number of complicating factors that might affect performance. For example, our analysis ignores CPU costs, underestimates I/O overheads, and ignores the fact that sequential writes on flash are slower than sequential reads.

A disadvantage of RARE-join, similar to jive-join, is that the join results must be materialized. For some data analysis functions, such as computing materialized views, this is not an issue. However, when used in a pipelined query plan, the above comparison is unfair. In that case, we need to penalize RARE-join with the cost for reading and writing R. Even so, RARE-join can be more efficient if the join result size or selectivity is sufficiently small.

Nonetheless, there are still opportunities to improve RARE-join. The hash-table on the join column could use compression for duplicate values. We could modify the algorithm to pipeline results better at the cost of additional I/Os. As Hybrid hash join does, we could potentially use available memory more effectively on the first pass through the data. We would also like to consider adapting other join algorithms, such as index-nested loops join and sort-merge join, for PAX layouts on flash.

5. RELATED WORK

We briefly review recent work on using flash in databases. Graefe [4] revisits the five-minute rule in the context of flash and suggests that flash serve as the middle level of a 3-level

memory hierarchy. Given current technology, this analysis shows that 32KB is too small for pages stored on a SATA disk and fairly large for pages stored on NAND flash. He lists several potential uses for flash, some which treat flash as an extension of memory and others that treat it as a faster disk. Graefe [5] also considers sorting over flash, although that paper is primarily concerned with improving memory utilization and robustness rather than with improving sort performance. In contrast, we focus solely on query processing over flash. For the future, we should consider adapting our methods to a 3-level hierarchy.

Lee and Moon [9] also present new variants of standard database algorithms that are adapted for the characteristics of flash. They consider techniques for updating rows in pages on flash. To avoid random writes, their approach logs updates to database pages in a clean "log" section at the end of each flash erase block rather than applying the updates in place. Once the log section is exhausted, they relocate the entire erase block and apply the updates. This approach amortizes the cost of the erase over multiple updates.

Next, we outline previous ideas that we adapted for query processing on flash drives. Ailamaki et al. [2] proposed the PAX database page design to improve the cache performance of TPC-H queries rather than to save on disk I/O. Reading only the relevant columns for each query is the central theme of column-oriented DBMSs such as C-Store and MonetDB [3, 14]. These systems reportedly perform well on certain types of queries [1, 6], using traditional disk drives. For example, Harizopoulos et al. [6] show that a carefully designed column store can out-perform a row-store for read-mostly workloads. Further, Abadi et al. [1] look at join processing over column layouts. As mentioned earlier, we can easily apply our algorithms to column stores on flash and provide similar benefits as with PAX. We, however, focus on a PAX layout since it imposes less disruptive changes to traditional database architectures. Moving to a flash storage and using the PAX page layout blurs the line between column-stores and row-stores. Like us, Zhou and Ross [15] use a scheme similar to PAX, called MBSM, that co-locates column values in blocks within larger "super-blocks" to reduce I/O. They optimize their methods, however, for traditional disks rather than flash.

Li and Ross [10] present efficient join algorithms, jive-join and slam-join, that leverage a join index and stores the results in a column-oriented format. We modify the jive-join by streamlining it with join index creation and by avoiding the unnecessary steps used to optimize disk accesses. To make disk I/Os sequential, jive-join sorts the join index before fetching the matching pages from the inner table and re-orders the returned tuples to match the order of the outer. Although this difference does not affect total data transferred, it introduces additional CPU overheads which can be important.

Some have also explored the energy-efficiency benefits of flash. Rivoire et al. [11, 12] show that using flash can improve the energy-efficiency of database operations like sort. Kgil and Mudge [7] employ flash for a buffer cache for web-servers to reduce their energy use.

6. CONCLUSION

In this paper, we present techniques for making core query processing operations, i.e. scans and joins, faster when using flash. Our techniques rely on using a PAX-based page lay-out, which allows scans to avoid reading columns not needed for the query. A PAX layout works well for flash drives since they offer much shorter seek times than traditional disks. We then present a join algorithm, RARE-join, that leverages the PAX structure to read only the columns needed to compute the join result. Roughly speaking, RARE-join first computes a join index by retrieving the join-columns and then fetches the remaining columns for the result. We show that RARE-join using a PAX layout beats traditional hash-based joins when few columns are returned and the selectivity is low.

Several directions suggest themselves for future work. Obviously, additional measurements on new hardware is an ongoing task. We also plan on studying scan and join performance on flash for generalized vertical partitioning, e.g., storing first name and last name together rather than separately. In addition, we plan on investigating merits and issues of RARE-join in complex query execution plans, e.g., pipelining and scheduling in bushy plans, memory management, and materialization of intermediate results. Finally, in addition to strict performance metrics, we plan on reviewing the new techniques with respect to energy efficiency as well as robustness of performance under adverse run-time conditions, e.g., errors in cardinality estimation, distribution and duplicate skew, and memory contention. We expect that steps in these directions will speed the eventual adoption of flash in enterprise systems.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? *SIGMOD*, 2008.

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

[3] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. *CIDR*, 2005.

[4] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. *DaMoN*, 2007.

[5] G. Graefe. Sorting with flash memory. *Unpublished manuscript.*, 2008.

[6] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. *VLDB*, pages 487–498, 2006.

[7] T. Kgil and T. N. Mudge. Flashcache: a nand flash memory file cache for low power web servers. *CASES*, pages 103–112, 2006.

[8] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

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

[10] Z. Li and K. A. Ross. Fast joins using join indices. *VLDB J.*, pages 1–24, 1999.

[11] S. Rivoire, M. A. Shah, P. Ranganathan, and

C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. *SIGMOD*, pages 365–376, 2007.

[12] S. Rivoire, M. A. Shah, P. Ranganathan, C. Kozyrakis, and J. Meza. Models and metrics to enable energy-efficiency optimizations. *IEEE Computer*, 40:39–48, Dec. 2007.

[13] Samsung. Samsung Semiconductor Products. Online. http://www.samsung.com/global/business/ semiconductor/products/flash/Products_NANDFlash.html.

[14] M. Stonebraker et al. C-store: A column-oriented dbms. *VLDB*, pages 553–564, 2005.

[15] J. Zhou and K. A. Ross. A multi-resolution block storage model for database design. *IDEAS*, July 2003.