Query Processing Techniques for Solid State Drives

Dimitris Tsirogiannis
University of Toronto
Toronto, ON, Canada
dimitris@cs.toronto.edu

Stavros Harizopoulos
HP Labs
Palo Alto, CA, USA
stavros@hp.com

Mehul A. Shah
HP Labs
Palo Alto, CA, USA
mehul.shah@hp.com

Janet L. Wiener
HP Labs
Palo Alto, CA, USA
janet.wiener@hp.com

Goetz Graefe
HP Labs
Palo Alto, CA, USA
goetz.graefe@hp.com

ABSTRACT

Solid state drives perform random reads more than 100x faster than traditional magnetic hard disks, while offering comparable sequential read and write bandwidth. Because of their potential to speed up applications, as well as their reduced power consumption, these new drives are expected to gradually replace hard disks as the primary permanent storage media in large data centers. However, although they may benefit applications that stress random reads immediately, they may not improve database applications, especially those running long data analysis queries. Database query processing engines have been designed around the speed mismatch between random and sequential I/O on hard disks and their algorithms currently emphasize sequential accesses for disk-resident data.

In this paper, we investigate data structures and algorithms that leverage fast random reads to speed up selection, projection, and join operations in relational query processing. We first demonstrate how a column-based layout within each page reduces the amount of data read during selections and projections. We then introduce FlashJoin, a general pipelined join algorithm that minimizes accesses to base and intermediate relational data. FlashJoin's binary join kernel accesses only the join attributes, producing partial results in the form of a join index. Subsequently, its fetch kernel retrieves the attributes for later nodes in the query plan as they are needed. FlashJoin significantly reduces memory and I/O requirements for each join in the query. We implemented these techniques inside Postgres and experimented with an enterprise SSD drive. Our techniques improved query runtimes by up to 6x for queries ranging from simple relational scans and joins to full TPC-H queries.

Categories and Subject Descriptors

H.2.4, H.2.2 [**Database Management**]: Systems – *query processing*, Physical Design – *access methods.*

General Terms

Algorithms, Design, Performance.

Keywords

Flash memory, SSD, columnar storage, join index, late materialization, semi-join reduction.

1. INTRODUCTION

Solid state drives (SSDs) are quickly penetrating the marketplace with promises of improved performance and energy efficiency for both desktop and enterprise applications. SSDs perform random reads more than 100x faster than traditional magnetic hard disks, while offering comparable sequential read and write bandwidth. Although current offerings are more expensive than magnetic hard disk drives (HDDs), a 2007 IDC study [13] showed that SSD prices are dropping much faster than HDD prices and predicted an annual decline of 50% for SSDs (in $/GB) through at least 2012. Due to their improved performance, low power consumption, small footprint, and predictable wearing, we expect SSDs to gradually replace HDDs as the primary permanent storage media in large data centers.

Such a shift in enterprise computing infrastructure has the potential to affect all storage-intensive applications. Database workloads may not benefit immediately from SSDs, however, since several DBMS architectural decisions, including data structures, algorithms and tuning parameters, were made based on the fundamental performance characteristics of HDDs. The common wisdom behind these decisions is to avoid random I/O as much as possible and instead emphasize sequential accesses, which are orders of magnitude faster on HDDs. SSDs, in contrast, perform random reads nearly as fast as sequential reads. The transition from HDDs to SSDs thus forces us to reexamine database design decisions.

1.1 Query Processing Techniques for SSDs

Previous work has mainly focused on quantifying instant benefits from the fast random reads of SSDs [16, 20] and addressing their slow random writes [15, 17, 21, 23]. Our focus instead is on investigating query processing techniques that improve the performance of complex data analysis queries, which are typical in business intelligence (BI) and data warehousing workloads. Towards this goal, we evaluate data structures and algorithms that leverage fast random reads to speed up selection, projection, and join operations. Along

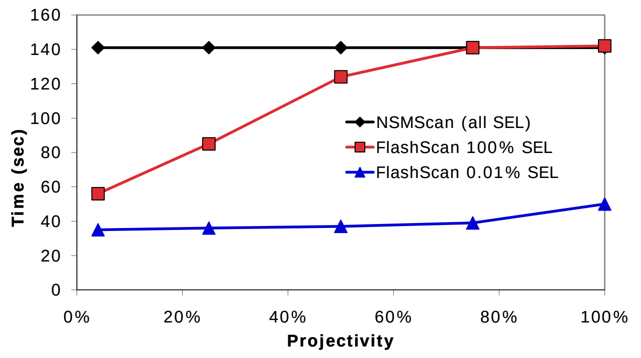

Figure 1: Scans on SSDs: FlashScan is much faster than a traditional scan when either few attributes (small projectivity) or few rows (small selectivity – SEL) are selected. Data analysis queries contain many scans on large relations that project few attributes and select few rows.

with sort (which can benefit from SSDs without algorithmic changes [16]) and aggregation (which typically applies to in-memory data or to the output of scan or join operators and therefore cannot benefit from SSDs), these operations are the most common in complex query plans.

We consider an "SSD-only" DBMS in which all data (tables, metadata, logs, etc.) are stored in SSDs. Our intention is to first evaluate which query processing techniques best exploit the characteristics of SSDs and then build on these new techniques for hybrid SSD/HDD configurations.

We advocate using a column-based layout within each database page, such as PAX [3]. While PAX was originally proposed to improve CPU cache performance, we show it can reduce the amount of data read from SSDs. In Section 3.3, we discuss the similarities between PAX layout on SSDs and column-store layout [4, 27] on both HDDs and SSDs. Using a PAX-based page layout, we implemented *FlashScan*, a scan operator that reads from the SSD only those attributes that participate in a query. FlashScan proactively evaluates predicates before fetching additional attributes from a given row, thus further reducing the amount of data read when few tuples are selected.

Building on FlashScan's ability to efficiently extract needed attributes, we introduce *FlashJoin*, a general pipelined join algorithm that minimizes accesses to relation pages by retrieving only required attributes, as late as possible. FlashJoin consists of a binary join kernel and a separate fetch kernel. Multiway joins are implemented as a series of two-way pipelined joins. The join kernel accesses only the join attributes, producing partial results in the form of a join index for each join node. Our current implementation uses a hash-based join kernel that employs the hybrid-hash join algorithm [25]; any other join algorithm may be used instead. Subsequently, FlashJoin's fetch kernel retrieves the attributes for later nodes in the query plan as they are needed, using different fetching algorithms depending on the join selectivity and available memory. We show that FlashJoin significantly reduces the amount of memory and I/O needed for each join in the query.

1.2 Evaluation inside PostgreSQL

We implemented the proposed techniques inside PostgreSQL.

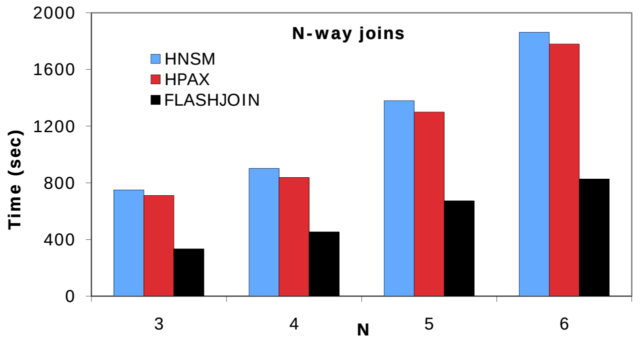

Figure 2: Multi-way joins on SSDs: FlashJoin is at least 2x faster than hybrid hash join over either traditional row-based (HNSM) or PAX layouts.

In addition to incorporating a new scan and join operator, we modified the buffer manager and the bulk loader to support both the original and PAX layouts. We experimented with an enterprise SSD drive using synthetic datasets and TPC-H queries. Figure 1 compares the performance of FlashScan to Postgres' original scan operator (NSMScan) as we vary the percentage of tuple length projected from 4% to 100% (experimentation details are in Section 3). FlashScan (100% selectivity, middle line) is up to 3X faster than the traditional scan for few projected attributes as it exploits fast random accesses to skip non-projected attributes. For low percentages of selectivity (0.01%, bottom line), FlashScan consistently outperforms the traditional scan by 3-4x, as it avoids reading projected attributes that belong to non-qualifying tuples. The original scan reads all pages regardless of selectivity. As we discuss in Section 3.3, column-store systems already enjoy similar performance benefits when scanning large relations on HDDs. Our techniques, however, are easier to integrate in existing row-based database systems than building a column-store from scratch.

Figure 2 shows the performance improvements when running multiway joins in Postgres using our FlashJoin algorithm. FlashJoin uses the drive more efficiently than hybrid-hash join over both traditional row-based (NSM) and column-based (PAX) layouts, by reading the minimum set of attributes needed to compute the join, and then fetching only those attributes that participate in the join result. By accessing only the join attributes needed by each join, FlashJoin also reduces memory requirements, which is beneficial in two ways: it decreases the number of passes needed in multi-pass joins (hence speeding up the join computation), and it frees up memory space to be used by other operators (hence leading in improved overall performance and stability in the system).

1.3 Contributions and Paper Outline

The introduction of a new, fast primary storage technology in enterprise computing warrants a thorough reexamination of database design choices. Towards this goal, we make the following contributions.

- We demonstrate the suitability of a column-based page layout for accelerating database scan projections and selections on SSDs, through a prototype implementation of *FlashScan*, a scan operator that leverages the columnar layout to improve read efficiency, inside PostgreSQL.

- We present *FlashJoin*, a general pipelined join algorithm that minimizes memory requirements and I/Os needed by each join in a query plan. FlashJoin is faster than a variety of existing binary joins, mainly due to its novel combination of three well-known ideas that in the past were only evaluated for hard drives: using a column-based layout when possible, creating a temporary join index, and using late materialization to retrieve the non-join attributes from the fewest possible rows.

- We incorporate the proposed techniques inside PostgreSQL. Using an enterprise SSD, we were able to speed up queries, ranging from simple scans to multiway joins and full TPC-H queries, by up to 6x.

The rest of the paper is organized as follows. SSD characteristics and trends, along with related query processing techniques are presented in Section 2. In Section 3 we describe and experiment with FlashScan and also discuss how column stores are related to our work. Section 4 describes in detail the join kernel and fetch kernel of FlashJoin. We present experiments with multi-way joins and full TPC-H queries in Section 5 and conclude in Section 6.

2. SSD TRENDS AND RELATED WORK

2.1 SSD Characteristics, Costs, and Trends

Most current commercial SSDs use NAND Flash for bulk data storage. NAND flash is a purely electronic, non-volatile store whose performance and price characteristics put it between DRAM and traditional disks. Table 1 summarizes the relevant characteristics of current Flash SSDs compared to traditional magnetic hard disks. All of the SSDs provide orders of magnitude (10-100x) faster random read IO/s than traditional drives and comparable sequential read and write bandwidth. Their random write performance, however, is much worse than read, especially on the consumer-grade drives. Finally, SSDs well outperform traditional drives on the price-performance metric IO/s/$ (random IO rate per dollar) and the energy-efficiency metric IO/s/W (random IO rate per Watt consumed).

The asymmetry between read and write performance is due to the underlying technology. NAND flash is organized into large 128K erase blocks while read and write IOs are to 4K pages. However, a 4K page write succeeds only if the page has been previously erased. Erasing is expensive in terms of both time and power consumed. Thus, a naive random write requires an expensive 128K read, erase, and rewrite operation. Further, most NAND flash is limited to about 100,000 erase-write cycles per block.

Fortunately, SSDs embed logic to hide these details. All SSDs 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, which implies an acceptable lifespan of 5-10 years with average utilization. Moreover, as Table 1 shows, enterprise SSD vendors are improving random write performance by overprovisioning the underlying flash capacity and embedding additional logic in the drive. Enterprise SSDs have so far been much more expensive than consumer SSDs, but these cost differences are shrinking as enterprise SSD volumes increase.

Overall, flash SSDs are seeing an annual $/GB decline of 50% per year [13], which is a faster drop rate than for hard disks. We expect flash SSDs to eventually be competitive with hard disks in terms of $/GB and continue to outperform them by orders of magnitude in read and write IO/s/$ and IO/s/W. Thus, we expect that for many data analysis applications, SSDs will replace hard disks in the future.

	SATA Disk	**SATA Flash**	**FC Flash**	**ioD Flash**
GB	500	32	146	320
$/GB	$0.12	$15.62	$85	$30
Watts (W)	13	2	8.4	6
seq. read (MB/s)	60	80	92	700
seq. write (MB/s)	55	100	108	500
ran. read (IO/s)	120	11,200	54,000	79,000
ran. write (IO/s)	120	9,600	15,000	60,000
IO/s/$	2.0	11.2	4.4	8.3
IO/s/W	9.2	5,600	6,430	13,166

Table 1: Disk and Flash characteristics from manufacturer specs or as measured where possible. Prices from online retailers as of Nov 25, 2008. SATA-disk: Seagate Barracuda; SATA-Flash: Mtron; FC-Flash: STech's ZeusIOps 3.5" FibreChannel; ioD: FusionIO ioDrive.

2.2 Databases and SSDs

Several recent studies have measured the read and write performance of flash SSDs [5, 20, 22]. uFLIP [5] defines a benchmark for measuring sequential and random read and write performance and presents results for 11 different devices. Of particular note, they identify a "startup" phase where random writes may be cheaper on a clean SSD (since no blocks need to be erased) but quickly degrade as the disk fills. Polte et al. perform a similar study using less sophisticated benchmarks, but focus on the behavior of filesystems running on top of SSDs. They show the degraded mode is 3-7X worse than the "startup" phase but still an order of magnitude faster than any current HDD [22].

Graefe [9] reconsiders the trade-off between keeping data in RAM and retrieving it as needed from non-volative storage in the context of a three-level memory hierarchy in which flash memory is positioned between RAM and disk. The cited paper recommends disk pages of 256KB and flash pages of 4KB to maximize B-tree utility per I/O time. Interestingly, these page sizes derive retention times of about 5 minutes, reinforcing the various forms of the "five-minute rule".

Both Myers [20] and Lee et al. [16] measure the performance of unmodified database algorithms when the underlying storage is a flash SSD. Myers considers B-tree search, hash join, index-nested-loops join, and sort-merge join, while Lee et al. focus on using the flash SSD for logging, sorting, and joins. Both conclude that using flash SSDs provides better performance than using hard disks. For example, fast random reads allow a larger fan-in during the merge phase of sorting, thus increasing the maximum size relation that can be sorted in two passes.

Then, there are a set of papers that propose new database algorithms designed especially for flash characteristics: these algorithms generally emphasize random reads and avoid random writes (where traditional algorithms stress sequential reads and writes and try to avoid any random I/O). Lee et al. [15] modify database page layout to make writing and logging more efficient. Ross [23] proposes new algorithms for counting, linked lists, and B-trees that minimize writes. Shah et al. [24] present a new hash-based join algorithm that, in combination with a new page layout, uses random reads to retrieve less data than hybrid hash join. However, their algorithm focuses on binary joins and there is no implementation of the proposed join algorithm. Nath and Gibbons [21] define a new data structure, the B-file, for maintaining large samples of tables dynamically. Their algorithm writes only completely new pages to the flash and they observe that writes of pages to different blocks may be interleaved efficiently on flash SSDs. Li et al. [17] propose a new index structure: it uses a B-tree in memory to absorb writes and then several levels of sorted runs of the data underneath the tree. This structure uses only sequential writes to periodically merge the runs and random reads to traverse the levels during a search.

Finally, Koltsidas and Viglas [14] consider hybrid SSD/HDD configurations of databases systems and design a buffer manager that dynamically decides whether to store each page on a flash SSD or a hard disk, based on its read and write access patterns. However, their approach assumes that all page accesses are random.

2.3 Related Query Processing Techniques

Our aim is to modify traditional query processing techniques to leverage the fast random reads of flash SSDs. Our modifications have three goals: (1) avoid reading unnecessary attributes during scan selections and projections, (2) reduce I/O requirements during join computations by minimizing passes over participating tables, and (3) minimize the I/O needed to fetch attributes for the query result (or any intermediate node in the query plan) by performing the fetching operation as late as possible. We discuss related techniques here and revisit some of those techniques in later sections.

Traditionally, database systems use the N-ary storage model (NSM), a page-based storage layout in which tuples (or rows) are stored contiguously in pages. NSM may waste disk and memory bandwidth if only a small fraction of each row is needed. In contrast, the decomposition storage model (DSM) [6], proposed in the 80s, decomposes relations vertically, allocating one sub-relation per attribute. DSM had its own disadvantages, including storage overhead for storing tuple IDs and expensive tuple reconstruction costs. With changing market needs and more favorable technology trends, newer DSM-like (column-store) commercial products and academic prototypes have recently appeared (such as SybaseIQ, Vertica, C-store [27], and MonetDB/X100 [4]). These systems eliminate storage overhead through virtual IDs and offer fast scans of few attributes [10] at the cost of additional disk seeks to fetch non-contiguous attributes.

PAX [3] (Partition Attribute Across) is a hybrid approach, essentially a DSM-like organization within an NSM page. While the disk access pattern of PAX is indistinguishable from that of NSM, it improves on the memory bandwidth requirements. On SSDs, as we show in the next section, PAX, in combination with the much faster seek time, allows reading only those columns needed by the query, essentially enjoying the read efficiency of DSM while retaining existing NSM functionality.

The ability of column-stores to read only part of a tuple led to an examination of different tuple materialization strategies [2]. A *late* materialization policy is particularly effective in reducing the amount of data passed through the query operators and we discuss its use in FlashJoin in Section 4.3. Late materialization was originally implemented by Semijoin reducers [26] which first computed a semi-join (using an index, if one was available) and then revisited base relations to obtain remaining attributes, without needing to redo predicate evaluation. Late materialization was also the essence behind TID hash joins [19], which compute a join index first and then construct the final join result. Although TID hash joins waste disk bandwidth when reading the input relations and rely on an inefficient fetching strategy for constructing the final result, they were shown to outperform traditional hash joins for highly selective joins on large inputs. A more efficient fetching strategy, given a pre-existing join index, was described for Jive/Slam-joins [18]. However, these algorithms were limited to two-way non-pipelined joins.

Our work builds on these ideas, re-examining fundamental tradeoffs in light of the different performance characteristics of SSDs.

3. SCAN SELECTIONS & PROJECTIONS

In this section, we demonstrate that a PAX-like page organization is a natural choice for database systems over SSDs. We first describe our implementation of *FlashScan*, a scan operator that leverages the PAX layout to improve selections and projections on flash SSDs. We evaluate the implementation of FlashScan inside Postgres in Section 3.2. Then, in Section 3.3 we discuss the similarities and differences between PAX layout and DSM or column layout and describe how FlashScan applies to column-stores.

3.1 FlashScan Operator

Figure 3 shows the page organization of both NSM and PAX for a relation with four attributes. In general, for an n-attribute relation, PAX divides each page into n *minipages*, where each minipage stores the values of a column contiguously. The size of each minipage is computed based on the average sizes of the attribute values. In our implementation, we do not handle overflowing of minipages caused by variable-length attributes; Ailamaki et al. describe how PAX can support variable-length attributes in [3]. The column-based layout of PAX offers a physical separation of values from different columns, within the same page, thus allowing an operator to access only the attributes that are needed in a query. Because regular scanners access a full page from disk, PAX does not have an impact on disk bandwidth. Once a page is brought into main memory though, PAX allows the CPU to access only the minipages needed by the query, thus reducing memory bandwidth requirements [3]. Data transfer units in main memory can be as small as 32 to 128 bytes (the size of a cacheline), whereas a database page is typically 8K to 128K. With SSDs, the minimum transfer unit is 512 to 2K bytes, which allows us to selectively read only parts of a regular page.

FlashScan takes advantage of the small transfer unit of SSDs to read only the minipages of the attributes that it needs. Consider a scan without selection predicates that

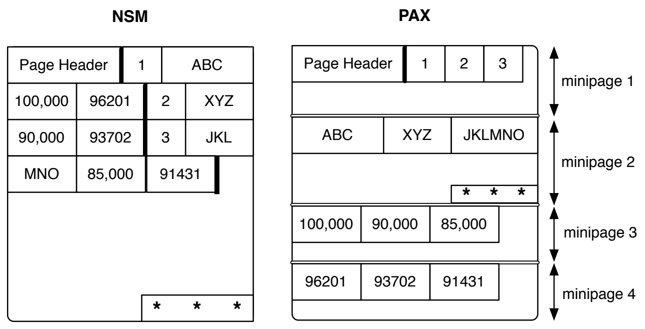

Figure 3: Disk pages in NSM (left) and PAX (right) storage layout of a relation with four attributes.

projects the first and third column of the relation in Figure 3. For each page, FlashScan initially reads the minipage of the first attribute and then "seeks" to the start of the third minipage and reads it. Then it "seeks" again to the first minipage of the next page. This procedure continues over the entire relation, resulting in a random (albeit strided) access pattern.

In general, every "seek" results in a random read. The only exception is when the scan query requests contiguous minipages; in that case, FlashScan coalesces the reads and performs one random access for each set of contiguous minipages. Using random instead of sequential reads is beneficial only if the storage medium supports fast random access and the reduction in the amount of data read compensates for the overhead induced by the random I/O. As we demonstrate in the next section, this is indeed the case for flash SSDs.

Note that we cannot expect the minipages to be exact multiples of the SSD minimum transfer unit. We also did not want to align minipages to OS page boundaries since that approach could result in severe fragmentation and poor space utilization. Instead, in our implementation of PAX and FlashScan, we decouple minipage size from SSD transfer units and compute the size of each minipage solely as a function of the page size and the average sizes of attribute values. Inside Postgres, we implemented PAX by dividing every database page into tightly packed minipages. Therefore, some unneeded data may be transferred when reading a single minipage.

We modified the Postgres buffer manager and bulk loader to work with the new page layout. The buffer manager allocates space and performs replacements at the granularity of a database page. However, a page in the buffer pool may be partially full containing only the minipages transferred by FlashScan. Although this design wastes space in the buffer pool, it simplifies our implementation and ensures that the invocation of the buffer manager does not influence our results in the experimental evaluation. It also allows us to reuse existing Postgres functionality and utilities.

We added FlashScan to Postgres as a new scan operator that produces tuples in row-format, after having read in the buffer pool all minipages containing the projected attributes. One of the features of Postgres that adds a considerable per-tuple overhead, is multi-version concurrency control (MVCC). Currently, our implementation does not support in-place updates (and therefore we have no use of MVCC), as our goal was to evaluate read-only queries. While certain database systems and several data warehousing applications typically operate under alternating periods of read-only queries and bulk-loading data, adding support for updates in PAX, if needed, could be performed without any penalty, as shown in [3]. To ensure a fair comparison with Postgres' default page layout (NSM) that includes MVCC information, we added the same overhead to our PAX pages by reserving an extra minipage inside each page. Additionally, we removed any calls related to MVCC when NSM is used.

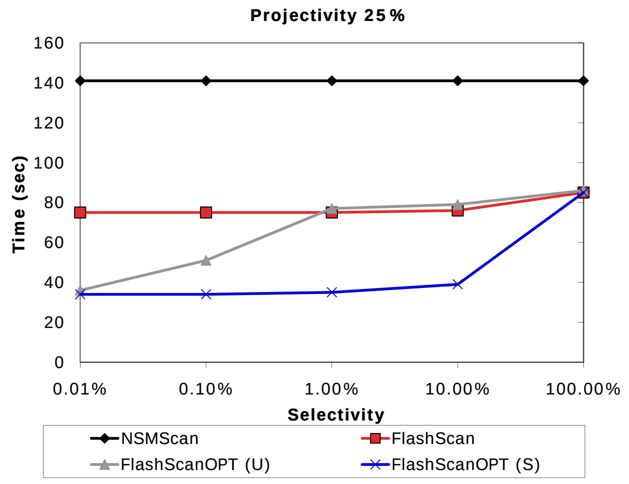

Figure 4: FlashScan avoids reading attributes that are not projected; FlashScanOpt additionally avoids reads of minipages without selected tuples. When run over sorted predicate attributes, FlashScanOpt(S) is able to skip many minipages of projected attributes.

3.1.1 Optimizations for Selection Predicates

For selection predicates, FlashScan can improve performance even further by reading only the minipages that contribute to the final result. Consider a scan query that specifies a set of projected attributes P (in the *select* clause in an SQL query) as well as a set of attributes S that participate in selection conditions (in the *where* clause). Assume for simplicity that S and P are disjoint, hence only the attributes in P are in the final result. FlashScan reads from each page only the minipages that correspond to the attributes in S and evaluates the selection conditions. If, for a given page, there is at least one tuple that passes the selection conditions, then FlashScan reads its minipages for the attributes in P. If none of the tuples of that page satisfy the selection conditions, FlashScan skips to the next page.

As we demonstrate next, this technique is beneficial for highly selective conditions (with a small number of tuples in the result) and for selection conditions that are applied to sorted or partially sorted attributes. In general, the more clustered the attribute values that satisfy the conditions, the bigger the performance improvement.

3.2 Scan Experiments

For the experiments in this section we generated a relation

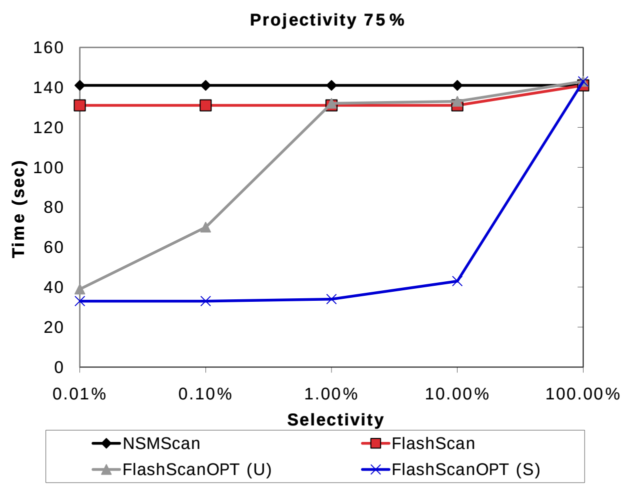

Figure 5: When more attributes are projected, FlashScan must read nearly as much data as NSMScan. The contrast between FlashScan and FlashScanOpt is therefore much greater (see Figure 4) when not all pages contain selected tuples.

with 70 million tuples occupying about 10GB. The relation consists of 11 columns (eight 4-byte and three 32-byte attributes) for a tuple length of 128 bytes. NSM layout in Postgres includes a 23-byte header for every tuple. Hence, in order to ensure that relations stored in NSM and PAX layouts have the same size, we allocated an extra minipage to each PAX page for the tuple headers. The page size for both NSM and PAX was set to 64KB. PAX pages contained 12 minipages and the minimum transfer unit from the SSD was 4KB. All experiments were performed on a system with an Intel Core 2 Duo CPU at 2.33GHz and 4GB of RAM, running Ubuntu 8.04 Linux with kernel 2.6.24-21. We used an MTron 32GB SSD, the performance characteristics of which are presented in Table 1, formatted with the Linux ext2 file-system.

3.2.1 Varying the Number of Projected Attributes

In the first experiment, we compare the performance of FlashScan to Postgres' original scan operator (*NSMScan*) as we vary the percentage of the tuple projected (its *projectivity*) from 4% to 100%. The results are in Figure 1 in Section 1 (top two lines). NSMScan always reads the whole relation, regardless of projectivity, and exhibits constant performance in scan queries. FlashScan is up to 3X faster than NSMScan for low projectivity: it exploits fast random accesses to "seek" efficiently to the required minipages. As projectivity increases, FlashScan reads more data from every page. At the point where it reads the entire tuple (100% projectivity), FlashScan performs the same sequential read of the relation as NSMScan.

3.2.2 Varying Selectivity

In this experiment, we consider a scan query with a single equality predicate and vary its selectivity from 0.01% to 100%. We consider two versions of FlashScan. Plain FlashScan first reads all minipages of the projected attributes, discarding non-qualifying tuples after evaluating the predicate. *FlashScanOpt* implements the optimization described in Section 3.1.1: minipages of projected attributes are read only if there is at least one tuple in the page that satisfies the predicate. We experiment with an equality predicate on both sorted and unsorted attributes. When the attribute is sorted, then all matching tuples are stored contiguously. The results for 25% and 75% projectivities are in Figures 4 and 5 respectively (we use the letters U and S to distinguish between the runs of FlashScanOpt on unsorted and sorted attributes) .

The execution time for NSMScan and plain FlashScan remains flat across different selectivities (the small increase for FlashScan for large percentages of selectivity is due to the higher tuple reconstruction cost of PAX). The optimized version of FlashScan, however, performs significantly better with lower percentages of selectivity. For predicates on unsorted attributes and selectivity below 1%, FlashScanOpt skips entire minipages that do not contain any qualifying tuples, thus outperforming plain FlashScan by up to 3x (for 0.01% selectivity and 75% projectivity). For more than 1% selectivity there is at least one tuple that satisfies the predicate in every page (in our relation there were 400 tuples per page). When applying the predicate on a sorted attribute, however, FlashScanOpt outperforms plain FlashScan for all selectivities below 100%: only a few pages contain the contiguous matching tuples and all other minipages can be skipped.

3.3 FlashScan and Column Stores

We discuss next the relation of FlashScan to column oriented storage on both HDDs and SSDs. For scans with many qualifying tuples, FlashScan's behavior is in many ways similar to a column-store scan on a traditional hard drive. FlashScan needs to "seek" between minipages, and that seek, although very fast, adds a small overhead (but is preferable over a full sequential scan). Column-stores on HDDs read a large portion of a single column at a time (called *chunk* in MonetDB/X100), to amortize the real disk head seek between different columns. For SSDs and HDDs with comparable bandwidths, these two scans would perform similarly. For highly selective scans, however, FlashScan has an advantage: it can skip minipages that do not contain qualifying tuples. A column-store on HDDs can only skip entire chunks which are *two* orders of magnitude larger than flash SSD transfer units. This means that column-store scans on HDDs need to be 100 times more selective than FlashScan to witness similar benefits.

If we were to run a column-store on SSDs, however, we would find similar behavior to FlashScan regardless of selectivity. In fact, by getting rid of the notion of a chunk, and instead rely on the transfer unit of SSDs, both FlashScan and column-stores would exhibit the exact same SSD reading times (since "seeks" on SSDs are constant in time). While this assertion needs experimental validation, if true, it would be a step towards converging row-store and column-store functionality (which, for current HDD systems, has been a subject of much debate [1, 12]). Several other aspects of row- and column-stores would need to be reexamined, however, this is a subject of future (and much promising) research. Compression, for example, might be implemented differently on a PAX-based row-store than in a pure column-store. One important aspect of our work is to demonstrate that the proposed techniques can be relatively easily integrated inside a full-blown relational DBMS, retaining much of the existing, rich functionality that otherwise would have to be re-implemented, if one was to build a column-store from scratch.

Since in the next section we describe a general join method which is based on FlashScan for reading data off disk, existing column-stores on SSDs could potentially adopt the same method, by plugging a column-scanner in the place of PAX-based FlashScan. Since the scope of the paper is limited to traditional, row-based DBMS, we do not revisit column-store applicability.

4. FLASHJOIN

In this section, we present *FlashJoin*, a multi-way join algorithm tailored for solid state drives. We first give an overview of our algorithm in Section 4.1 and then describe its main components in the following sections.

4.1 FlashJoin Overview

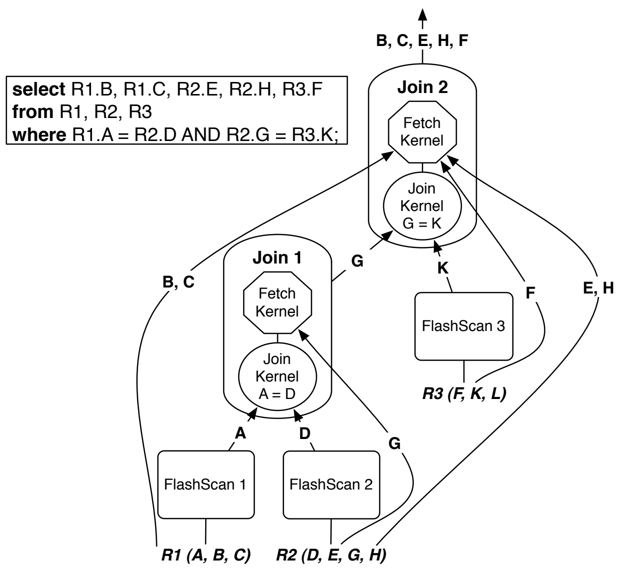

Figure 6: Execution strategy for a three-way join when FlashJoin is employed.

To take advantage of the fast random reads of SSDs, FlashJoin uses the same principal techniques as FlashScan. It avoids reading unneeded attributes and postpones retrieving attributes in the result until absolutely necessary.

FlashJoin is a multi-way equi-join algorithm, implemented as a pipeline of stylized binary joins. Each binary join in the pipeline is broken into two separate pieces, a *join kernel* and a *fetch kernel*, each of which is implemented as a separate operator, as shown in Figure 6. The join kernel computes the join and outputs a join index. Each join index tuple consists of the join attributes as well as the row-ids (RIDs) of the participating rows from base relations. The fetch kernel retrieves the needed attributes using the RIDs specified in the join index. FlashJoin uses a *late materialization* strategy, in which intermediate fetch kernels only retrieve attributes needed to compute the next join and the final fetch kernel retrieves the remaining attributes for the result. This approach offers some important benefits over traditional joins, which use an *early materialization* strategy.

First, FlashJoin reduces the amount of data retrieved from the input relations to compute the join result. It accesses the join attributes and accesses other projected attributes only from rows that participate in the result. Second, since join kernels process join indices instead of all projected attributes from the input relations, the indices on the build input are smaller. Thus, the join kernel is more memory-efficient. Moreover, when multiple passes are needed, the join kernel incurs lower partitioning costs than traditional joins.

These benefits come at the cost of additional random reads for retrieving the join attributes and other projected attributes separately. Moreover, they come at the cost of additional passes over data in the fetch kernel. In the experimental section, we show that this tradeoff is worthwhile when using SSDs.

4.2 Join Kernel

The *join kernel* leverages FlashScan to fetch only the join attributes needed from base relations. The join kernel is implemented as an operator in the iterator model. In general, it can implement any existing join algorithm: block nested loops, index nested loops, sort-merge, hybrid hash, etc.

In this paper, we explore the characteristics of a join kernel that uses the hybrid-hash algorithm [7]. This kernel builds an index (hash table) on the join attribute and RID from the inner relation and probes the index in the order of the outer. Depending on the available memory and the size of the input, hybrid-hash may make multiple passes over the input. Compared to a traditional hash join, this join kernel is more efficient in two important ways because it does not need to manage all of the projected attributes from the input. First, the join kernel needs less memory, thus many practical joins that would otherwise need two passes can complete in one pass. Second, when the join kernel spills to disk, materializing the runs is cheaper. Although we only explore hybrid-hash in this paper, we believe a sort-merge kernel would offer similar results because of the duality of hash and sort [8].

4.3 Materialization Strategy

To implement different materialization strategies, the query plan must also change at a logical level internal to FlashJoin. Typically in a query plan, associated with each node is a description of the tuple the node produces. For example, in Figure 6, a traditional plan would indicate that the first join (*Join 1*) produces tuples with attributes (B, C, E, G, H). By adjusting these descriptions, which indicate to the fetch kernel the attributes to retrieve, we can specify any strategy we like for retrieving projected attributes, as long as it is consistent with the plan.

For any individual binary join, there is a tension between the cost of retrieving projected attributes from base relations immediately after the join and the cost of retrieving the attributes further downstream. The former increases the cost for partitioning in subsequent join kernels, if those joins cannot complete in one pass. The latter forces the final fetch kernel to make additional read and write passes over the output, which can be expensive if the output cardinality is large. Choosing the optimal strategy requires estimating the cost of these options and minimizing the total cost across the whole multi-way join. The number of possibilities is large since at every node in the plan, we can choose to materialize any subset of the remaining needed attributes. To make matters more complicated, the cost for a particular strategy may be significant enough to affect the total cost and thereby affect the optimal join order.

Instead of exploring this optimization space, in this paper, we use a simple heuristic: late materialization. This heuristic postpones retrieving projected attributes as far downstream as possible. Every join produces the minimum set of attributes needed by the next operator and the final join produces the output needed by the remaining plan operators. This heuristic works well as we show in Section 5 because typical query plans reduce the output cardinality at each level of the plan. In that case, an extra pass over the output is usually cheaper than paying the additional materialization cost through multiple levels of the plan. In order to perform the late materialization, the join kernel output carry forward the RIDs for each of the base relations needed downstream.

Figure 6 shows an example of this strategy. Each tuple produced by *FlashScan 1* in contains only attribute A, which is one of the join attributes of *Join 1*, and each tuple produced by *FlashScan 2* contains only attribute D, the second join attribute of *Join 1*. Similarly, the tuples produced by *Join 1* contain only attribute G, which is one of the join attributes of *Join 2*. Finally, *Join 2*, the last join node, produces tuples with all of the attributes needed in the result: B, C, E, H, F. Moreover, if there was a sort operator after *Join 2* that sorted the results based on attribute L (R_3), then each tuple produced by *Join 2* would also contain L.

4.4 Fetch Kernel

The *fetch kernel* uses the join index, produced by the join kernel, to materialize the attributes of the join result from their base relations. For example, in Figure 6, the join kernel of *Join 1* outputs pairs of RIDs (id_1, id_2) from relations R_1 and R_2, respectively. A RID specifies the page and offset within the page for that row. The fetch kernel uses id_2 to locate and retrieve attribute G from relation R_2. Similarly, *Join 2* produces a join index containing (id_1, id_2, id_3) pointing to rows of R_1, R_2, and R_3. The corresponding fetch kernel, uses id_1 to retrieve attributes B, C, id_2 to retrieve attributes E, H and id_3 to retrieve attribute F.

A straightforward strategy for the fetch kernel is to retrieve projected attributes in a tuple-at-a-time, non-blocking fashion. For each join index tuple, the kernel locates the needed mini-pages in the buffer pool or retrieves them from the underlying relation, and composes the result tuple. This approach is reasonable when all of the pages needed to generate the result can fit in memory because random reads are cheap. When available memory is insufficient, however, this approach may result in reading some pages multiple times because the RIDs in the join index are usually unordered. The larger the result, the worse is the overhead for re-reading pages. TID hash joins implemented this approach, and this overhead was their biggest weakness [19].

To mitigate this overhead, we present a fetching strategy, inspired by Jive-Join [18], that reads each containing page from each base relation only once at the cost of additional passes over the join index. Algorithm 1 makes multiple passes over the join index fetching attributes in row order from one relation at a time. Roughly speaking, in each pass, it sorts the join index based on the RIDs of the current relation to be scanned. Then, it retrieves the needed attributes from that relation for each tuple and augments the join index with those attributes. If the join index does not fit in memory, it uses an external merge sort. Sorting ensures that once a mini-page from a relation has been accessed, it will not need to be accessed again, thus placing minimal demands on the buffer pool. Sorting, however, does not ensure sequential access to the underlying relation because the needed pages can be far apart. Hence, this strategy of decoupling and postponing materialization is better suited to SSDs than HDDs.

Algorithm 1 Fetch Kernel to produce the join result with multiple passes.

```
Input: $R_1, \ldots, R_k$: base relations, $id_1, \ldots, id_k$: corresponding RID attributes, $A_1, \ldots, A_k$: $A_i$ is the set of attributes to fetch from relation $R_i$, $|A_1| < \ldots < |A_k|$, $JI$: join index
Output: $JR$: join result
 1: $T$ = SortedStream($JI$, $id_1$)
 2: for $i = 1$ to $k$ do
 3:     $Z = \{\}$
 4:     while $T$ is exhausted do
 5:         for all memory resident tuples $t$ of $T$ do
 6:             Add GeneratePartialResultTuple($t$, $id_i$, $R_i$, $A_i$) to $Z$
 7:         end for
 8:         Produce sorted run of $Z$ on $id_{i+1}$
 9:     end while
10:     $T$ = SortedStream($Z$, $id_{i+1}$)
11: end for
12: $JR = T$
```

We explain some important optimizations in Algorithm 1 above. First, the final merge of the sort and attribute retrieval are pipelined to avoid the unnecessary final write. **SortedStream** sorts the join index but leaves out the last merge step that creates a final sorted run (line 1). As tuples are fetched from T (a sorted stream), it merges the underlying runs on demand. For each tuple, **GeneratePartialResultTuple** retrieves the needed attributes and augments the join index (line 6). We ensure that only enough tuples are read so that the result Z, the new join index with the attributes A_i projected, can fit in memory (line 5). Finally, we sort Z on the RID for the next relation (line 8). The sorted Z runs are then merged into a single sorted stream T (line 10) which is then pipelined with attribute retrieval from the next relation.

A second optimization is that our fetch kernel processes relations in order of the width of the attributes retrieved, from smallest to largest. This order reduces the data spilled to SSD when producing the intermediate runs. Third, if Z and JI fit in memory, we avoid spilling them to disk. Fourth, if a fetch kernel (esp. intermediate ones in the plan) already has its input sorted on the RIDs for one of the relations, it processes that relation first to avoid the sort. The cost for this fetch strategy depends on the cardinality of the result, the width of the attributes, and the number of relations that need to be accessed.

Although we did not implement it, an analogous fetching strategy would be to use hashing instead of sorting, as done in RARE-join [24]. In each pass, we could hash join index tuples into buckets based on the page id in the RID. This would ensure that all tuples that need the same page fall into the same bucket. The buckets could be sized such the number of distinct pages referenced fit in the available memory. We could then process each bucket fetching the the appropriate pages from base relations. This strategy requires less CPU overhead than the sort-based one, but would need more memory than the sort-based one for the same number of partitions.

In summary, if the optimizer estimates that all the pages needed to retrieve projected attributes can fit in memory, we opt for the naive, tuple-at-a-time strategy. Otherwise, we resort to our sort-based relation-at-a-time fetching strategy.

5. EXPERIMENTAL EVALUATION

This section presents our experiments with FlashJoin. Our objective is to demonstrate the effectiveness and efficiency of FlashJoin during the execution of multi-way joins and complex BI queries. In Section 5.1, we describe the algorithms evaluated, the datasets and queries used, and implementation details. The results are presented in the remainder of the section.

5.1 Experimental Setting

FlashJoin was implemented inside PostgreSQL as a new join operator. Significant changes had to be performed in PostgreSQL's *planner* component to support the late materialization strategy. The planner employes a cost-based optimizer to determine the most efficient way to execute a query. The output of the planner is a query execution plan that details which access methods to use and which attributes to access from each base relation. Additionally, it determines for each operator in the query plan the schema (attributes) of output tuples. We altered the planner in order to generate query execution plans which comply to the late materialization strategy. Specifically, we added a recursive function that takes as input the query execution plan and process it in a top-down fashion (recall that a query execution plan is a tree of operators). For every operator (node in the query execution plan) our function alters the schema of output tuples by removing unnecessary attributes. Overall, we added approximately 7K new lines of code in PostgreSQL, half of which was the implementation of FlashJoin and planner component and the rest divided between FlashScan, bulk-loading utilities, the buffer manager and the storage layout.

There are several limitations with respect to our implementation of FlashJoin and the execution of complex queries. Currently, we support query execution plans that contain a pipeline of hash joins followed by other operators, such as sort and aggregation. Furthermore, the optimizer is not aware of the late materialization strategy and the cost of FlashJoin and so it produces the same plan that it would using the hybrid hash operator. In the future, we plan to integrate the cost of FlashJoin into the optimizer.

In all of our experiments, we compare the performance of FlashJoin with the hybrid-hash join algorithm implemented in PostgreSQL. We consider two variations of the hybrid-hash join algorithm: one for each storage layout. We refer to the default implementation over the NSM layout as *HNSM*. We call our version of hybrid-hash over the PAX layout *HPAX*. HPAX and HNSM use the same materialization strategy, but when HPAX is used, the scan nodes use the FlashScan operator to read only the projected attributes of each relation.

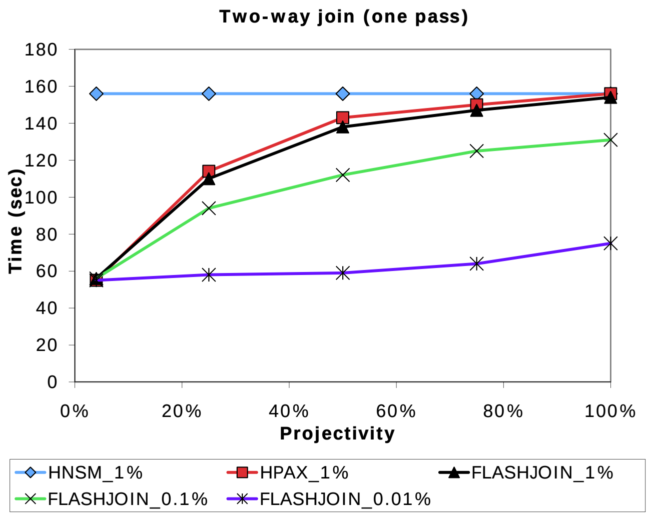

Figure 7: Performance comparison of join algorithms for two-way joins when the join is computed in one pass.

We conducted a series of experiments using synthetically generated datasets and queries. We also evaluated the performance of FlashJoin using full TPC-H queries. All of the experiments were conducted with the same page sizes and relations and on the same system that we described in Section 3.2. The sizes of the relations and the amount of memory allocated are described separately for each experiment. After the execution of each query, PostgreSQL was restarted and all cached pages were flushed from memory using the *drop_caches* utility of the Linux kernel. In all the experiments, the performance metric measured is wall-clock time to produce the full join result.

5.2 Two-way Join Results

In this section, we assess FlashJoin during the execution of two-way joins. We consider the equijoin of two relations R, S on a single join attribute. R contains 70 million tuples with a total size of approximately 10GB and S contains 7 million tuples with a total size of 1GB. We consider queries of the following form: "***SELECT*** $R.a_n, \ldots, R.a_m,\ S.a_n, \ldots, S.a_m$ ***FROM*** R, S ***WHERE*** $R.a_i = S.a_j$". We use *projectivity* to refer to the percentage of the tuple length projected from each join relation. When projectivity is 100%, a join result tuple contains all of the attributes of R and S and has a total tuple length of 256 bytes.

5.2.1 One-pass Joins

In the first experiment, we vary projectivity from 4% to 100%. The amount of memory allocated to the join is 1GB to ensure that all algorithms compute the join in one pass. Figure 7 shows the runtimes of HNSM, HPAX, and FlashJoin with three different result cardinalities. The percentage next to each algorithm's label indicates the cardinality of the join result, expressed as a percentage of the cardinality of the larger join relation. In this experiment, join result cardinality had a negligible effect on the performance of HPAX and HNSM, so we only present one set of execution times

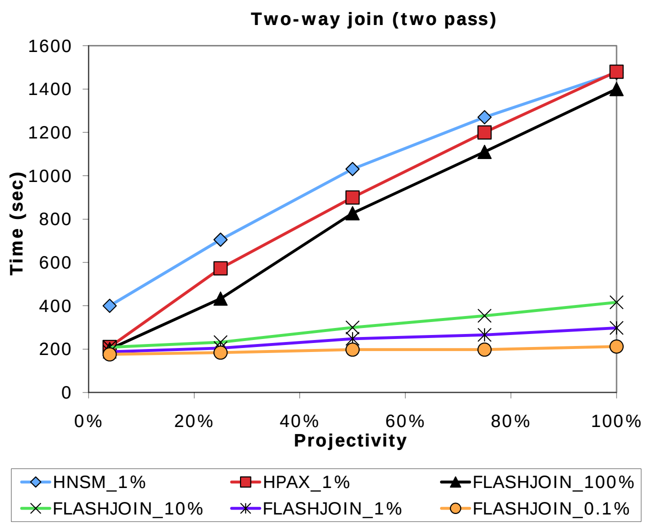

Figure 8: Performance comparison of join algorithms for two-pass two-way joins.

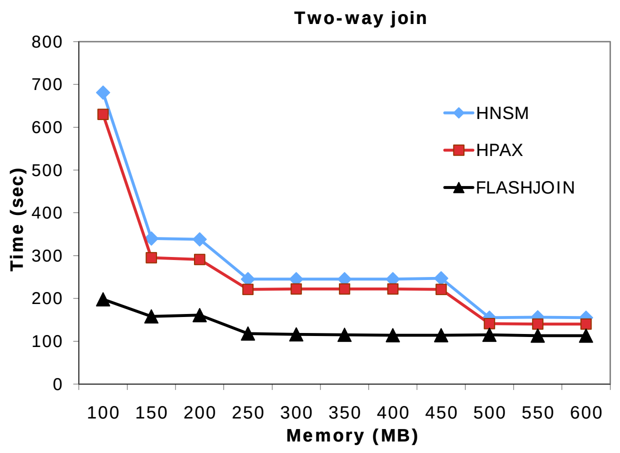

Figure 9: Performance comparison of two-way join algorithms as a function of memory size.

for them.

The performance of HNSM is independent of projectivity. The input scans of HNSM read relations R and S entirely, regardless of how many attributes participate in the join result. HPAX, however, uses the FlashScan operator to read from flash disk only the attributes that are actually needed in a query. For low projectivities (4% in our experiment), HPAX is 3X faster than HNSM. However, as projectivity increases, more attributes are read and the performance difference between HPAX and HNSM diminishes.

The performance of FlashJoin depends on both the projectivity and the join result cardinality. When the cardinality is 1% or greater, FlashJoin and HPAX perform the same since every disk page of R and S contains at least one tuple that belongs to the join result and must be read. For cardinalities less than 1%, FlashJoin reads all of the minipages of the join attributes but only a fraction of the minipages containing projected attributes. Consequently, it reads less data than HPAX. As projectivity increases, the number of minipages that FlashJoin does *not* read increases accordingly and causes a more pronounced performance difference of up to a factor of 3x.

5.2.2 *Two-pass joins*

In the second experiment, we compare the performance of the join algorithms when they require two passes to compute the join result. The amount of memory allocated to the join is 100MB and the same query was used as above. Figure 8 shows results for HNSM and HPAX with a join result cardinality of 1% and for FlashJoin with cardinalities, varying from 0.1% to 100%.

The execution times of HNSM and HPAX increase linearly with projectivity: since projections are performed early in the query execution plan, more data participate in the partitioning phase as projectivity increases. Consequently, the partitioning cost increases. HPAX is faster than HNSM for lower projectivities because it reads only the attributes needed in the query, until at 100% projectivity, both HPAX and HNSM read all attributes and perform the same.

FlashJoin, however, is much faster than HNSM and HPAX, and increasingly so as projectivity increases. FlashJoin accesses only the join attributes during the expensive partitioning phase, when two passes are required for the computation of the join. Consequently, the partitioning cost of FlashJoin does not depend on projectivity.

The performance of FlashJoin does depend on the join result cardinality, however. When the cardinality is low, FlashJoin reads only a few minipages of projected attributes in order to construct the join result tuples and performs up to 7X faster than HPAX and HNSM. As cardinality increases, FlashJoin reads a larger fraction of minipages, thus increasing the join result construction cost. Furthermore, when the join index does not fit in memory, FlashJoin pays the cost of materializing the join index, whose size depends on the result cardinality. When the result cardinality is 100%, FlashJoin must read as many minipages as HPAX and is only marginally faster than HPAX.

5.2.3 *Memory impact on joins*

In the third experiment, we compare the performance of the join algorithms as we vary the amount of memory allocated. We set the join result cardinality at 1% of the cardinality of R (larger relation) and the projectivity at 25%. We vary the amount of memory from 100MB to 600MB.

Figure 9 shows that FlashJoin is faster than both HPAX and HNSM for all memory sizes examined. HPAX and HNSM require at least 500MB to compute the join in one pass. The hash table on the join attribute of the build relation requires 270MB and the projected attributes of the build relation require 230MB. When memory is less than 500MB, HPAX and HNSM compute the join in two passes. Since they both use the hybrid hash join algorithm, they exploit any memory available to avoid writing all of the partitions. When the allocated memory is between 250MB and 500MB, one partition fits in memory and only the second partition gets written.

By reading only the join attributes, FlashJoin has a smaller memory footprint and increases the range of memory sizes at which a two-way join is executed in one pass. FlashJoin requires only 270MB of memory (for the hash table)

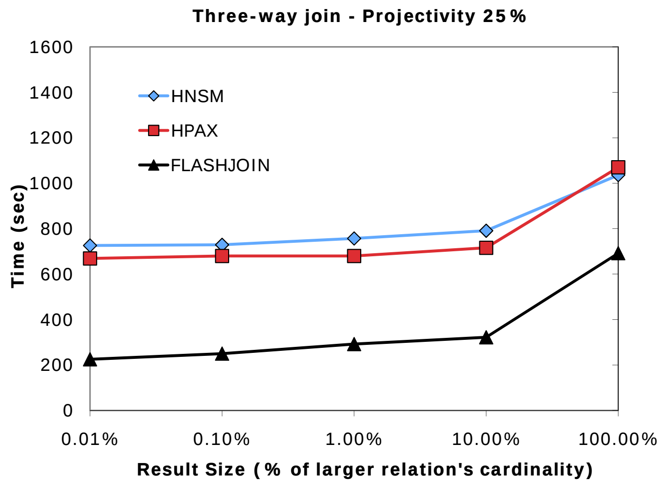

Figure 10: Performance comparison of join algorithms during three-way joins as a function of join result cardinality, when projectivity is 25%.

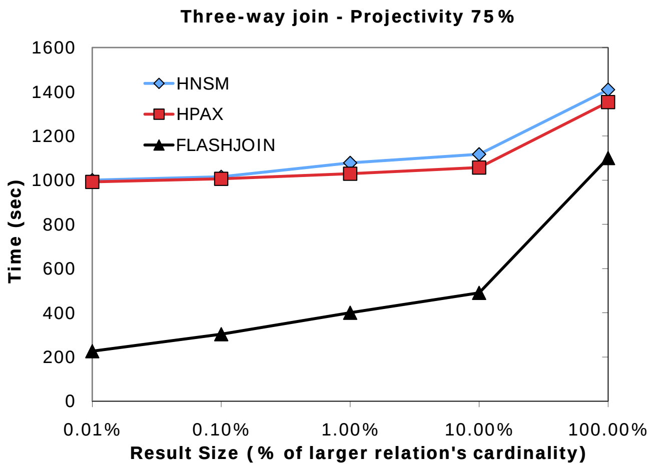

Figure 11: Performance comparison of join algorithms during three-way joins as a function of join result cardinality, when projectivity is 75%.

to compute the join in one pass. Consequently, FlashJoin is 2X faster than the other algorithms when memory size is between 270MB and 500MB. When memory is larger than 500MB, FlashJoin is 1.2X faster. When memory is less than 270MB, FlashJoin computes the join in two passes; it is 3X faster than HPAX and HNSM due to the reduced partitioning cost.

5.3 Multi-way Joins

Next, we assess FlashJoin during the execution of multi-way joins. We consider three-way joins first, then look at star joins of up to six relations. Finally, we examine the impact of changing the join algorithm on more complex queries such as those in TPC-H.

5.3.1 *Three-way Joins*

For this experiment, we executed queries of the form: "***SELECT*** $R.a_n, \ldots, R.a_m,\ S.a_n, \ldots, S.a_m,\ T.a_n, \ldots, T.a_m$ ***FROM*** $R, S,\ T$ ***WHERE*** $R.a_i = S.a_j$ *AND* $R.a_k = T.a_l$". R contains 50 million tuples, S contains 20 million tuples and T contains 7 million tuples; their sizes are 7.5GB, 3GB and 1GB, respectively. The join between R and S is a primary-foreign key join returning 50 million tuples (R contains the foreign-key). We control the cardinality of the join result by varying the selectivity of the join between R and T. The join order is determined by the optimizer and is the same for all queries executed. The amount of memory allocated to each join is 100MB. All of the joins are executed in two passes.

Figures 10 and 11 show the results for two different projectivities: 25% and 75%, respectively. For each projectivity, we vary the cardinality of the join result from 0.01% (5000 rows) to 100% (50 million rows) of the larger relation's cardinality.

FlashJoin is up to 3X faster than HPAX when projectivity is 25% and up to 5X faster than HPAX when projectivity is 75%. For a given projectivity, increasing the cardinality of the join result increases the execution time of all algorithms: higher cardinalities causes more rows in the intermediate results and hence higher partitioning costs. However, FlashJoin suffers the least since it only partitions the join attributes. It benefits more at higher projectivity since there are more attributes that it avoids partitioning.

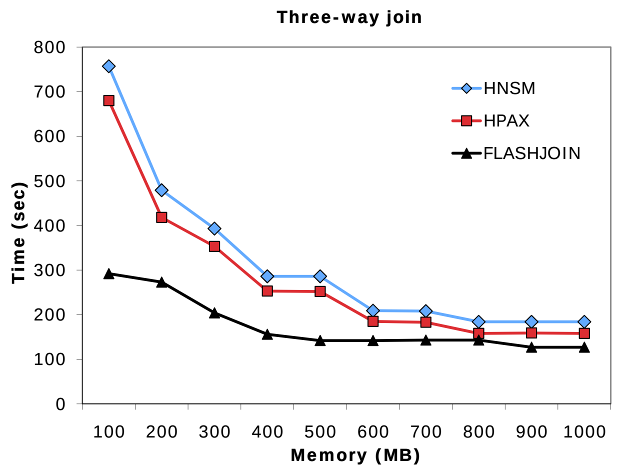

Figure 12: Performance comparison of join algorithms during three-way joins, as a function of memory size allocated per join.

5.3.2 *Memory impact on joins*

In the next experiment, we compare the algorithms' performance for three-way joins as a function of the amount of memory allocated per join. We consider the same three-way join queries as in the previous experiment. We set the join result cardinality at 1% of the cardinality of relation R and the projectivity of each relation at 25%. We vary the amount of memory allocated per join from 100MB to 1GB. Figure 12 presents our results.

Both HPAX and HNSM require 500MB to execute the join between R and T in one pass and 1600MB to execute the join between R and S in one pass. By accessing only the join attributes, FlashJoin only requires 270MB to compute the join between R and T in one pass and 900MB to compute the join between R and S in one pass. FlashJoin is therefore 1.3X to 2.5X faster than HNSM and 1.1X to 2.3X faster than HPAX. We also conducted experiments with larger join result cardinalities and got consistent results.

5.3.3 *Star joins*

Next, we assess FlashJoin during the execution of N-way STAR joins for different values of N, ranging from 3 to 6. The relations and their sizes are presented in Table 2. R_0 is the fact table and $R_1, \ldots, R_5$ are dimension tables. In this experiment, an N-way join involves the first N relations $(R_0, \ldots, R_{N-1})$ of Table 2. The join between R_0 and R_{N-1} returns 5 million tuples (10% of R_0's cardinality). Every other join between R_0 and R_i, where $i = 1, \ldots, N-2$, is a primary-foreign key join. We set the projectivity of each relation at 25%. Hence, for the 6-way join, the size of each result tuple produced is 192 bytes.

Relation	R_0	R_1	R_2	R_3	R_4	R_5
Cardinality (M tuples)	50	20	14	7	3.5	2
Size (GB)	7.5	3	2	1	0.5	0.3

Table 2: Relations participating in N-way STAR joins.

Figure 2 in Section 1 presents our results. FlashJoin outperforms both HPAX and HNSM by at least a factor of 2X for all of the values of N examined.

5.3.4 *TPC-H Queries*

Finally, we evaluate FlashJoin in the context of more complex queries. We chose queries Q3 and Q10 from the TPC-H benchmark. Q3 retrieves unshipped orders. It involves a three-way join among tables *CUSTOMER*, *ORDERS* and *LINEITEM*. There is a *GROUP BY* clause, applied on two attributes of *ORDERS* and one attribute of *LINEITEM*. Q3 also sorts orders by decreasing revenue. Q10 identifies customers having problems with shipped parts. In involves a four-way join among tables *CUSTOMER*, *ORDERS*, *LINEITEM* and *NATION*. A *GROUP BY* clause is applied on six attributes of *CUSTOMER* and one attribute of *NATION*. Customers are sorted in decreasing order of lost revenue. We generated data using a scale factor (SF) of 10. In all tables, we replaced variable length attributes with fixed length attributes. In both queries, we set the memory allocated per operator at 100MB.

Figure 13 presents results for both queries. For Q3, FlashJoin is 1.6X faster than HNSM and 1.4X faster than HNSM. For the amount of memory allocated, all join algorithms require two passes to execute one of the joins in Q3. Hence, FlashJoin performs better than HPAX and HNSM due to its reduced partitioning cost.

For Q10, FlashJoin is 1.8X faster than HNSM and 1.5X faster than HPAX. As in Q3, there is one join in Q10 that is executed in two passes by all join algorithms. However, Q10 exhibits higher projectivity than Q3, producing larger tuples. Consequently, the partitioning cost of HNSM and HPAX is even higher for Q10 and their performance difference with FlashJoin increases.

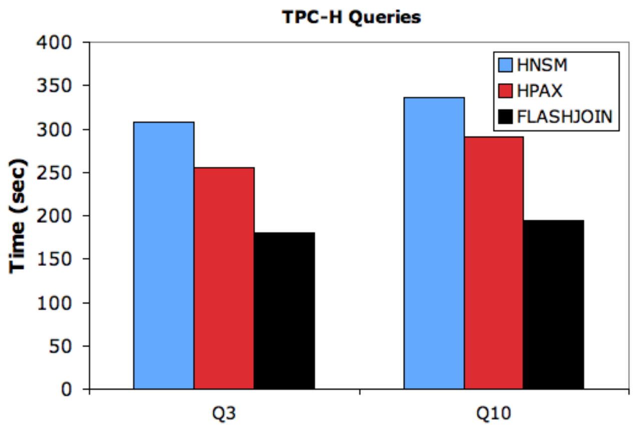

Figure 13: Performance comparison of join algorithms during the execution of TPC-H queries.

6. CONCLUSIONS

SSDs constitute a significant shift in hardware characteristics, comparable to large CPU caches and many-core processors. In database systems, SSDs change not only power efficiency but also query execution efficiency. In this paper we demonstrate that SSDs can improve database performance for business intelligence and other data analysis queries. We first show that a column-based page data layout is a natural choice for speeding up selections and projections on SSDs. By combining a column-based storage layout with temporary join indexes and late materialization, we produce a new pipelined join algorithm that is much more efficient on SSDs than previous algorithms such as hybrid-hash join.

Our results show performance improvements of up to a factor of six for scans, multiway joins, and complex queries from the TPC-H benchmark. These improvements stem from three factors: First, FlashJoin only reads attributes as they are required by each operator, which reduces the amount of data read as compared to hybrid-hash join. Second, FlashJoin has a reduced memory footprint and therefore increases the range of memory sizes for which each join can be computed in one pass; one-pass joins require much less I/O than two-pass joins. Third, it employs a late materialization strategy to access the minimum set of attributes needed at any point in the query execution plan, thus reducing the amount of data accessed from base relations and the partitioning cost when the joins are executed in two passes.

Our work has focused so far on storage formats and query execution algorithms that can take advantage of the performance characteristics of SSDs. One new consideration for query optimization is that many query execution plans that were not appropriate for HDDs become appropriate for SSDs. In addition to the algorithms presented here and in related work, query optimization may need to reconsider when to use index navigation, including index-nested-loops join and other forms of nested iteration. We expect SSDs to expand the cases in which index-based query execution plans are competitive with set-oriented query execution plans using hash join, hash aggregation, and sorting. In addition, the new efficiency trade-offs of late versus early materialization greatly expand the set of interesting join permutations for a given query. We expect that query optimization for databases on SSDs — and databases with both HDDs and SSDs — will be an interesting area of future research.

Although we focus mainly on performance in this paper, an important optimization criteria for many systems is price-performance. To improve this metric, we must optimize not only for purchase price but also for energy-related costs [11]. In doing so, we expect that a hybrid disk, SSD, and DRAM system will be typical. As SSD prices fall, SSDs will claim a larger role in system — trading away disk for better performance and trading away DRAM for lower purchase cost and power dissipation. The best ratios will depend on the workload as well as prevailing technology costs and trends. Determining these ratios with total cost in mind as well as devising methods for seamlessly spanning across all three memory technologies are promising areas of future research.

7. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475, 2007.

[3] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3), 2002.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[5] L. Bouganim, B. Jonsson, and P. Bonnet. uFlip: Understanding flash IO patterns. In *Proc. CIDR*, 2009.

[6] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[9] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *ACM Queue*, pages 1–9, 2007.

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

[11] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy Efficiency: The New Holy Grail of Data Management Systems Research. In *CIDR*, 2009.

[12] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *Proc. VLDB Endow.*, 1(1):502–513, 2008.

[13] J. Janukowicz, D. Reinsel, and J. Rydning. Worldwide solid state drive 2008-2012 forecast and analysis. Technical Report 212736, IDC, June 2008.

[14] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, 2008.

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

[16] 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.

[17] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proc. ICDE*, 2009.

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

[19] R. Marek and E. Rahm. TID hash joins. In *CIKM*, pages 42–49, 1994.

[20] D. Myers. On the use of NAND flash memory in high-performance relational databases. *MIT Msc Thesis*, 2008.

[21] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.*, 1(1):970–983, 2008.

[22] M. Polte and J. Simsa and G. Gibson Enabling enterprise solid state disks performance *Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, 2009.

[23] K. A. Ross. Modeling the performance of algorithms on flash memory devices. In *DaMoN*, pages 11–16, 2008.

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

[25] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[26] K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper. Integrating semi-join-reducers into state of the art query processors. In *ICDE*, pages 575–584, 2001.

[27] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.