YourSQL: A High-Performance Database System Leveraging In-Storage Computing

Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang,
Sangyeun Cho, Daniel DG Lee, Jaeheon Jeong
Memory Business, Samsung Electronics Co.

ABSTRACT

This paper presents *YourSQL*, a database system that accelerates data-intensive queries with the help of additional in-storage computing capabilities. YourSQL realizes very early filtering of data by offloading data scanning of a query to user-programmable solid-state drives. We implement our system on a recent branch of MariaDB (a variant of MySQL). In order to quantify the performance gains of YourSQL, we evaluate SQL queries with varying complexities. Our result shows that YourSQL reduces the execution time of the whole TPC-H queries by 3.6×, compared to a vanilla system. Moreover, the average speed-up of the five TPC-H queries with the largest performance gains reaches over 15×. Thanks to this significant reduction of execution time, we observe sizable energy savings. Our study demonstrates that the YourSQL approach, combining the power of early filtering with end-to-end datapath optimization, can accelerate large-scale analytic queries with lower energy consumption.

1. INTRODUCTION

Delivering end results quickly for data-intensive queries is key to successful data warehousing, business intelligence, and analytics applications. An intuitive (and efficient) way to speed them up is to reduce the volume of data being transferred across storage network to a host system. This can be achieved by either filtering out extraneous data or transferring intermediate and/or final computation results [18].

The former approach is called *early filtering*, a typical example of data-intensive, non-complex data processing. Due to its simplicity and effectiveness, it has seen many forms of implementations [3, 12, 19, 22]. However, existing implementations do not fully utilize modern solid-state storage device technology and are limited in cost-effectiveness and performance. First, the software filters [3] presuppose filter metadata like index, but the overheads of metadata management can be prohibitively costly. Thus, high-end filter servers [19] or FPGAs [12, 22] are employed between disk enclosures and database servers. However, they are not a perfect solution, either. Leaving the expense aside, the amount of data transferred from storage devices remains unchanged because the data must be transferred in any case to filter servers or FPGAs before being filtered.

Contrary to the prior work, we argue that early filtering may well take place at the earliest point possible—within a storage device. Besides a fundamental computer science principle—when operating on large datasets, do not move data from disk unless absolutely necessary—, modern solid-state drives (SSDs) are not a dumb storage any longer [7, 9, 11, 13, 16, 17, 20]. Therefore, we have explored a novel database system architecture where SSDs offer compute capabilities to realize faster query responses. Some prior work aims to quantify the benefits of in-storage query processing. Do et al. [11] built a "smart SSD" prototype, where SSDs are in charge of the whole query processing. Even though this work lays the basis for in-storage query processing, there remain large areas for further research due to its limitations. First, it focuses on proving the concept of SSD-based query processing but pays little attention to realizing a realistic database system architecture. Not only would join queries be unsupported, but internal representation and layout of data must be converted to achieve reasonable speed-up. Moreover, the hardware targeted by this work (i.e., SATA/SAS SSDs) is outdated and the corresponding results may not hold for future systems. Indeed, its performance advantages mainly result from the higher internal bandwidth inside an SSD compared to the external SSD bandwidth limited by a typical host interface (like SATA/SAS), but the assumption of such a large bandwidth gap (i.e., a gap of 2.8× or larger) does not hold for a state-of-the-art SSD. We feel a strong need for a practical system design that would work on a state-of-the-art SSD and be able to accelerate complex queries as well.

This paper presents *YourSQL*, a database system architecture that leverages *in-storage computing* (ISC) for query offloading to SSDs. We built a prototype of YourSQL that supports ISC-enabled early filtering. Our prototype works on a commodity NVMe SSD that offers user-programmability and provides built-in support for advanced hardware IPs (e.g., pattern matcher) as well [13]. Not only would extraneous data reduction take place within SSDs, but YourSQL would request relevant pages only. Moreover, it leverages the hardware pattern matcher in the target SSD, which makes our system sustain high, on-the-fly table scan throughput. Our findings and contributions are summarized as follows:

- We seamlessly integrated query offloading to SSDs into MySQL. No prior work reports a full port of a ma-

Proceedings of the VLDB Endowment, Vol. 9, No. 12

jor DB engine. We modified its query planner to include new algorithm that automatically identifies certain query operations to offload to the SSD. Portions of its storage engine are rewritten so that those operations are offloaded to the SSD at run time.

- Contrary to the prior work, YourSQL runs all 22 TPC-H queries. With YourSQL, the average speed-up for the five TPC-H queries with the largest performance gains is over 15×. Also, the execution time of the whole TPC-H queries is reduced by 3.6× compared to "vanilla" system. With the significant reduction of the execution time, it achieves substantially lower energy consumption as well (e.g., about 24× less energy consumption in the case of TPC-H Query 2).

- We present detailed state-of-the-art techniques for accelerating large-scale data processing involving pattern matching (e.g., grep, html/xml parsing, table scan) with the help of SSD.

In the remainder of this paper, we introduce the motivations behind YourSQL's early filtering approach and its overall architecture in Section 2. The implementation details of YourSQL are described in Section 3. Section 4 provides the methodologies of our evaluation and the results. We discuss related works in Section 5 and conclude in Section 6.

2. YOURSQL

2.1 Motivational Example: Early Filtering

The early filtering approach is usually considered one of the most effective ways to achieve the acceleration of data-intensive queries. The level of potential improvement varies depending on *selectivity*, which represents a fraction of relevant rows from a row set (e.g., a table, a view or the result of a join) in a given data set. The value ranges from zero to one, and zero is the highest selectivity value that means no rows will be selected from a row set.

One may think that higher selectivity guarantees higher speed-up with early filtering. However, it is not always the case because the selectivity, which is calculated at row-level, is not always quantitatively correlated with the amount of page I/O that can be reduced by early filtering. Their correlation, in deed, may vary depending on the distribution of rows in pages. Therefore, we define our own metric called *filtering ratio*. This is, in essence, a page-level selectivity, which represents a fraction of relevant pages from a page set. The value of filtering ratio ranges from zero to one likewise, and zero is the highest value. Filtering ratio is highly dependent upon the existence of filter predicates, which in general appear as base relation predicates of the form '`column operator constant`' (e.g., `p_size = 15` and `p_type LIKE '%BRASS'` in `Query 1`).

Based on filter predicates, an early filter determines whether each page from a page set is needed to answer a given query, and passes only relevant pages to subsequent steps. As an example of early filtering, consider a single table query `Query 1` and a join query `Query 2` on TPC-H dataset [1]. To estimate the filtering ratio and the resulting I/O reduction, we ran the example queries with or without *Index Condition Pushdown* (ICP) [3], and counted the number of read requests in both cases. ICP is an early filtering feature of MySQL that can be enabled when the filter columns of a given query are preindexed. In order to enable ICP for the queries in this example, we first preprocessed the dataset by creating a multi-column index on `p_size` and `p_type`.

[1]We used a TPC-H dataset at a scale factor of 100.

Query 1: A simple selection query.

```
SELECT p_partkey, p_mfgr FROM part
WHERE p_size = 15 AND p_type LIKE '%BRASS';
```

Query 2: TPC-H Q2.

```
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr,
       s_address, s_phone, s_comment
FROM part, supplier, partsupp, nation, region
WHERE p_partkey = ps_partkey
      AND s_suppkey = ps_suppkey
      AND p_size = 15 AND p_type LIKE '%BRASS'
      AND s_nationkey = n_nationkey
      AND n_regionkey = r_regionkey
      AND r_name = 'EUROPE'
      AND ps_supplycost = (SELECT MIN(ps_supplycost)
          FROM partsupp, supplier, nation, region
          WHERE p_partkey = ps_partkey
            AND s_suppkey = ps_suppkey
            AND s_nationkey = n_nationkey
            AND n_regionkey = r_regionkey
            AND r_name = 'EUROPE')
ORDER BY s_acctbal DESC, n_name, s_name, p_partkey LIMIT 100;
```

`Query 1` is a simplified version of `Query 2` and refers to the `part` table only. Its filter predicates are very limiting with the high filtering ratio of 0.067, that is, only 6.7% of pages in this table are needed to answer this query [2]. Compared to a single table query, a join query can potentially benefit more from the early filtering approach. For the effective early filtering of a join query, its join order should be optimized first, that is, the early filtering target should be placed first in the join order. With the (early) filtered rows placed first in the join order, intermediate row sets to be processed could significantly be reduced at the earliest stage of join.

Table 1 shows how the query plan and the resulting read amounts change in the presence of ICP for `Query 2`. We show top query plans only since subquery plans are identical in both cases. In this table, `Access method` and `Key` represent the access method to each join table (e.g., full scan, index scan) and the key (index) column that MySQL employed, respectively. In the `Access method` column, `all` denotes a table scan, and `ref` and `eq_ref` [1] denote joins by indexed columns. `pk` and `fk` in the `Key` column denote primary key and foreign key, respectively.

As seen from Table 1, MySQL utilizes an index nested-loop join algorithm [4]. Such join is effective if the outer table is small and the inner tables are preindexed and large [6]. This is why MySQL puts a table with the smallest number of rows first in the join order. `region` is such a table and thus it is accessed first in the top of Table 1. However, if ICP is enabled, it accesses a table that can be filtered (i.e., a table whose filter columns are preindexed) first, as shown in the bottom of Table 1. Even though the filtering ratio of the early filtering target (i.e., the `part` table) is same in both queries, I/O reduction from the optimized join order and irrelevant data elimination is dramatic. The total number of read requests is reduced by a factor of 24.4. With

[2]In this case, each read request from MySQL with ICP is a single page random read, and thus we can consider the number of read requests as the number of page reads.

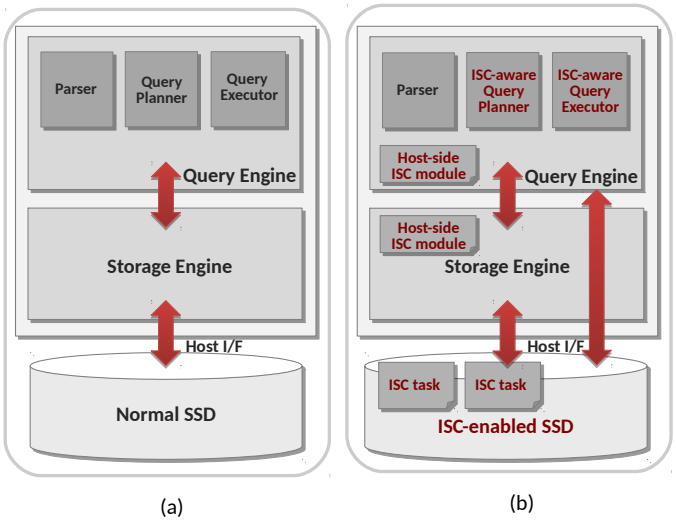

Figure 1: Two database system architectures. (a) Traditional system. (b) YourSQL.

this significant I/O reduction, `Query 2` expects a far more speed-up than `Query 1`.

Table 1: Query plans of MySQL for Query 2.

Join order	Table	Access method	Key	# of read requests
		MySQL without ICP		
1	region	all	null	16
2	nation	ref	nation_fk	13
3	supplier	ref	supplier_fk	36,867
4	partsupp	ref	partsupp_fk	2,842,639
5	part	eq_ref	pk	651,525
Total				3,531,060
		MySQL with ICP		
1	part	ref	p_size & p_type	245
2	partsupp	ref	pk	98,520
3	supplier	eq_ref	pk	45,679
4	nation	eq_ref	pk	5
5	region	all	null	4
Total				144,453

2.2 Our Approach

In the previous section, we illustrated a typical example of early filtering in the case of data-intensive, non-complex query operations. We believe that ISC perfectly fits not only early filtering but also other data-intensive, non-complex query operations such as aggregation and projection. First, it allows a database system to reduce the volume of data traffic within SSDs. Given that the cost of data-intensive operations is largely driven by the number of I/O requests, reducing I/O requests at the earliest stage of query processing could bring noticeable performance improvement. Second, the latest SSDs have sufficient compute capability to process non-complex query operations. Motivated by the unique capabilities of ISC for data-intensive, non-complex query operations, we design an ISC-enabled database system architecture, named as YourSQL. In this architecture, queries would be processed by a host system and SSDs in a distributed manner. One of the YourSQL's key design considerations is to keep the architectural base of the traditional database system intact for the ease of implementation as well as for the compatibility with existing systems.

Fig. 1 highlights the differences between the traditional system and YourSQL. The former is comprised of a database system and normal SSDs. The database system includes a query engine and a storage engine. Given a query, the query engine sets a query plan and executes it, and the storage engine handles I/O requests from the query engine. While YourSQL in Fig. 1(b) generally follows the basic architecture of the traditional system, it is further equipped with ISC-enabled SSDs and the entire software stack is aware of them via ISC modules. YourSQL architecture builds upon a simple yet practical and flexible system design where any database system could accelerate queries by offloading any chosen query operations. The following items are the remaining key design considerations in YourSQL.

ISC tasks. User-programmable SSDs employed in YourSQL allow complex query operations to be offloaded in the form of *ISC task* [13]. While there is no practical limit in the extent of query operations, query offloading should take the aggregate internal bandwidth as well as the compute capability of target SSD into account (relative to the compute capability of host). Therefore, operations to offload should be carefully extracted from the original database system and defined as ISC tasks. In the case of early filtering in YourSQL, operations that are relevant for data relevancy inspection, which are common in data-intensive queries, are offloaded to the ISC-enabled SSDs as ISC tasks.

Interfaces between a host and ISC tasks. For distributed processing, an interface between a host database system and ISC tasks must be well defined so that the host can invoke ISC tasks and retrieve results from them. Such interface should also tolerate a relatively large bandwidth since the size of results generated by ISC tasks could relatively be large. In the case of early filtering in YourSQL, an interface is defined between the storage engine and the ISC filters, through which enabling the filter tasks and retrieving filtering results are done iteratively.

Query planner optimized for ISC. The query plan may considerably differ between the traditional and ISC-enabled database systems. Thus, the existing query planner may be revised to include new algorithms optimized for ISC-enabled database systems. In the case of early filtering in YourSQL, the query planner in MySQL is modified to consider alternative query plans where the early filtering target is accessed first in order. As it will be shown in the following sections, this approach is highly efficient in reducing I/O requests.

Reorganized datapath for ISC. Different from the traditional system, YourSQL does not require the entire data to be transferred to a host to be processed. In the case of early filtering in YourSQL, SSD itself can determine relevant pages and deliver their list to the host. With such list at hand, the host system can boost I/O operations by prefetching relevant pages in bulk. Reorganizing datapath is thus necessary, which is done by revising the existing read logic for YourSQL.

3. DESIGN AND IMPLEMENTATION

We build a prototype of YourSQL that supports ISC-enabled early filtering. Fig. 2 depicts how early filtering in YourSQL is done with the ISC-enabled SSD. Given a query, the query engine (1) parses it and (2) chooses a candidate for the early

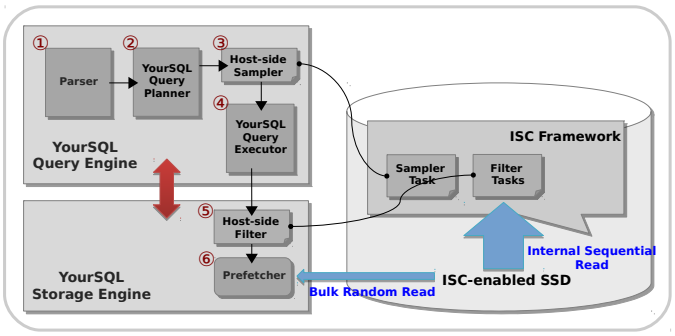

Figure 2: Early filtering of YourSQL.

filtering target. Then, (3) it invokes the ISC sampler to estimate I/O reduction that would result from early filtering for the candidate table. If early filtering is expected to be beneficial, (4) it decides the candidate as the early filtering target and sets the query plan that would trigger early filtering. Instead of table scan, (5) the storage engine invokes the ISC filters so as to perform early filtering for the target table. As the results from the ISC filters become available (e.g., list of relevant pages), (6) the host performs ISC-optimized read (e.g., bulk prefetch) based on the retrieved results.

3.1 Basic Design

3.1.1 Target Table Selection and Join Order

Join order optimization in the early filtering approach is a daunting task because it is notoriously difficult to choose a right target table to filter and a right join order that guarantee overall performance improvement. MySQL's approach is rather simple. If there are no secondary indexes, ICP will not be triggered. That is, no index, no early filtering. This severely restricts the applicability of early filtering in MySQL. Considering analytic dataset, database administrators would not create indexes on non-key attributes, unless absolutely necessary, since it is not realistic to manage secondary indexes at runtime. In the presence of indexes, ICP will be triggered. However, MySQL's overly simplified target selection criteria, which simply choose a table with preindexed filter columns, do not always guarantee I/O reduction since the amount of I/O reduction depends on filtering ratio. In the worst case scenario, queries may experience severe performance drop with such an overly simplified approach.

In contrast, YourSQL neither considers indexes as the target selection criteria nor utilizes them for early filtering. YourSQL instead relies on the filtering ratio of a given table, and considers a table whose estimated filtering ratio is sufficiently high as the early filtering target. Any prerequisite is not necessary and thus the applicability is highly enhanced. Moreover, target selection based on filtering ratio gives more accurate assessment of potential performance gain.

The naive way to calculate the filtering ratio of a given table is counting the number of qualifying pages while scanning it, which must incur huge overhead. In order to choose a table that is expected to have the highest filtering ratio with reasonable overhead, YourSQL introduces two metrics, *limiting score* and estimated filtering ratio. The former is calculated by simple heuristics and therefore incurs negligible overhead. However, it is not quantitatively correlated with filtering ratio. Therefore, estimated filtering ratio is

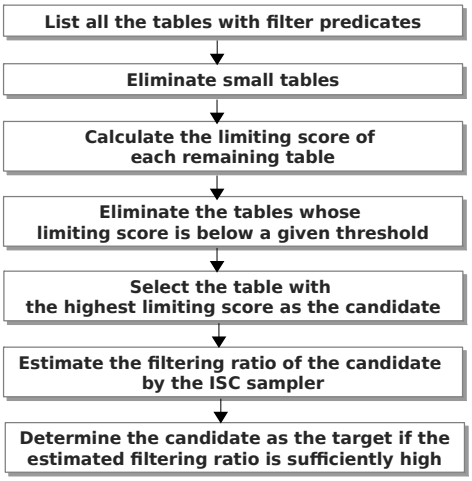

Figure 3: Selection of the early filtering target table.

introduced for further assessment. This estimated filtering ratio, which will be explained in Section 3.2.1 in detail, is quantitatively correlated with filtering ratio, and thus the accuracy of assessment of potential performance gain becomes even more elevated.

If we briefly describe YourSQL's target selection for a given query, it chooses a candidate for the early filtering target by limiting score, and then decides whether to choose the candidate for the target by estimated filtering ratio.

The limiting score of a table represents how restrictive its filter predicates are. Thus, a table with high limiting score can be expected to be one with high filtering ratio. To calculate the limiting score of a given table, YourSQL takes account of the number of filter predicates, types of operations in the filter predicates, and the number of rows in the table. For a small table, a zero score is assigned [3], because it is reasonable to disregard such table as the early filtering target. For a non-small table, YourSQL assigns a score for each filter predicate, which is summed to give the limiting score of the table. A filter predicate gets a higher score as its type of operation is more restrictive. For example, a filter predicate to test equality gets a higher score than a filter predicate to check for value in a range.

Fig. 3 describes how YourSQL chooses the early filtering target of a given query in detail. We explain this figure using `Query 2`. First, it lists the tables with filter predicates. Regarding this query, the `region` table has a single predicate, `r_name = 'EUROPE'`, and the `part` table has two predicates, `p_size = 15` and `p_type LIKE '%BRASS'`. Thus, these two tables are listed. Second, it eliminates small tables from the list by assigning a zero score. In this case, the `region` table has five rows only, and gets zero. After eliminating small tables, it calculates the limiting score of each remaining table in the list. Considering `Query 2`, the `part` table is the only remaining in the list and its limiting score is the sum of scores for two filter predicates. If the limiting score of a table is below the threshold, it eliminates the table from the list. Then, it chooses the table with the highest limiting score as the candidate, and estimates the filtering ratio of

[3]A given table is considered small if the entire table could be read in a single unit of YourSQL's read operation. We will explain YourSQL's read unit in Section 3.1.3.

the candidate by the ISC sampler. If the estimated ratio is higher than the threshold, it determines the candidate as the early filtering target. In the example query, the `part` table is finally chosen as the target.

Once the early filtering target is set, YourSQL places it first in the join order so that intermediate row sets would be narrowed as early as possible. For the remaining join order, it follows MySQL's decision. Considering the example above, YourSQL finally determines a join order that accesses the `part` table first as shown in Table 2. Comparing query plans of MySQL with ICP and YourSQL for `Query 2`, both use the same join algorithm and set the same join order. The essential difference between them is the access method to the first table, i.e., the early filtering target. MySQL performs early filtering by secondary indexes on filter columns, and thus accesses the `part` table through the multi-column index. In contrast, YourSQL performs early filtering with the ISC filters, which scan the early filtering target.

Table 2: YourSQL's query plan for Query 2.

Join order	Table	Access method	Key
1	part	all	null
2	partsupp	ref	pk
3	supplier	eq_ref	pk
4	nation	eq_ref	pk
5	region	all	null

3.1.2 Filtering Condition Pushdown

Filtering condition pushdown (FCP) is an optimization for the case where YourSQL retrieves rows from a table using filter predicates. Upon request from the storage engine, the ISC filters evaluate the pushed conditions against the target table and store match hints. Here, match hints are a byte array whose element is set to one if the corresponding page satisfies filtering conditions. Once starting, these filters iterate a process of early filtering and accumulating match hints till reaching the last page of the target table. The accumulated match hints are pulled by the storage engine and used to avoid irrelevant page access for a given query.

As shown in Fig. 4, two ISC filters are involved in FCP. The hardware filter leverages the hardware pattern matcher equipped with the target SSD. This specialized hardware provides powerful filtering capability that allows a significant reduction of data transfer across the storage network. Nonetheless, hardware-level inspection of data relevancy is limited in that hardware-filtered data can still contain false positives depending on the filtering conditions. Therefore, we introduce another software-based filtering layer in FCP. Depending on the filtering conditions, the match hints by the hardware filter may be in turn redirected to the software filter, so that the relevancy of data first examined by the hardware filter can be further inspected. The software filter will be explained in Section 3.2.2 in detail.

The hardware pattern matcher in the target SSD takes at most three 16-byte binary keys and performs byte-granular matching. Therefore, YourSQL first transforms filter predicates into binary patterns by considering their types of columns and operators, and feeds those patterns to the hardware filter. In the case of `=` or `LIKE` operator, constant is transformed into internal binary representation. Considering `Query 2`, MySQL represents varchar (or variable character) and integer as ascii and four bytes starting with 0x8, respectively. Thus, `p_type LIKE '%BRASS'` and `p_size = 15` are converted into two binary keys, `'42 52 41 53 53'` and `'80 0 0 0F'`. Also, in the case of range operators such as $\leq$ and $\geq$, YourSQL converts each constant into its internal representation and then extracts common sequence if possible. Considering `Query 3`, MySQL converts date into three byte integer that starts with 0x8. Thus, `l_shipdate` $\geq$ `'1995-09-01'` and `l_shipdate` $<$`'1995-09-01' + INTERVAL '1' MONTH` are first converted into `'8F 97 21'` and `'8F 97 41'`, respectively. Then, the common two byte sequence, `'8F 97'`, is extracted.

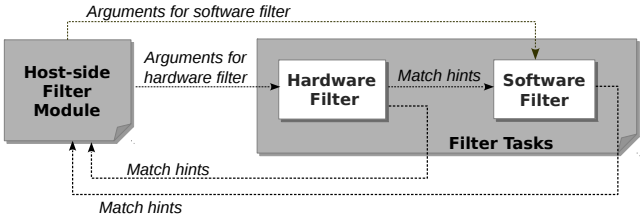

Figure 4: ISC filters.

Query 3: TPC-H Q14.

```
SELECT 100.00 * SUM(CASE WHEN p_type LIKE 'PROMO%'
  THEN l_extendedprice * (1−l_discount) ELSE 0 END) /
  SUM(l_extendedprice * (1−l_discount)) AS promo_revenue
FROM lineitem, part
WHERE l_partkey = p_partkey
  AND l_shipdate >= date '1995−09−01'
  AND l_shipdate < date '1995−09−01' + INTERVAL '1' MONTH;
```

3.1.3 Table Access based on Match Hints

YourSQL performs query processing in a distributed manner. As shown in Fig. 5, an early filterable query would be processed by a series of three tasks: early filtering, match page reads, and row processing. ISC-enabled SSD takes charge of the first task, and the host does the remaining two. These tasks run concurrently in the ISC-enabled SSD and the host. By this concurrent processing (in Fig. 5(b)) rather than sequential processing (in Fig. 5(a)), YourSQL efficiently hides early filtering overhead.

Given a query, YourSQL first requests early filtering for the chosen target to the ISC filters. Once starting, they continue to the last page of the target table without interruption. In doing so, they accumulate match hints in a well-defined iteration unit. An iteration involves reading several hundreds or thousands of consecutive pages and producing the corresponding match hints. This early filtering unit, the number of pages processed by a single iteration, must be set before YourSQL starts. The value is added as `bulk_match_size` to MySQL's configuration file (i.e., `my.cnf`) and fully configurable in YourSQL.

On an as-needed basis, YourSQL's storage engine pulls match hints from the filters, and these hints are used for ISC-optimized page reads of the storage engine. Thus, irrelevant pages determined by the ISC filters would not be transferred to the host. When the storage engine receives a request for the next row from the query engine, it first checks the current page has remaining rows. If so, it finds the next row and passes it to the query engine. Otherwise, it has to read a new page and return the first row in the newly read page.

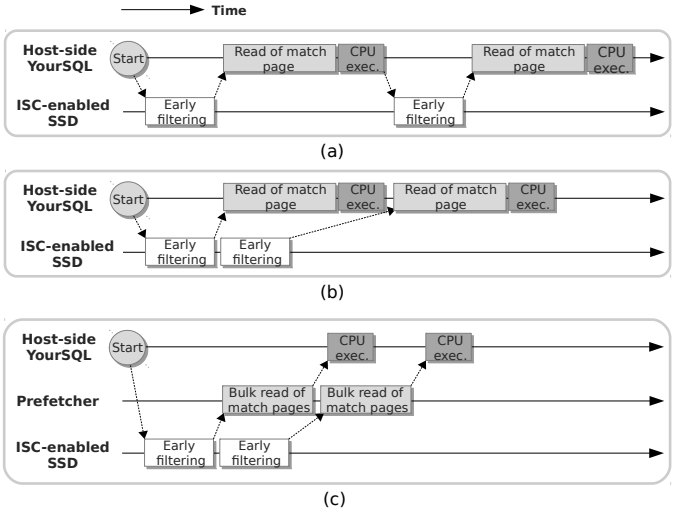

Figure 5: Execution timing diagram in YourSQL.

If the table to read is the early filtering target, it does not issue a read request for the next page. Rather, it checks if the match hint corresponding to the next page is set to one. As long as the match hint is set to zero, it proceeds to check the next match hints. If it finally meets a page whose match hint is set to one, it fetches such page with normal host read and then passes a new row to the query engine. If it has already checked all the match hints for the current unit, it pulls match hints for the next unit from the filters, and then continues to read based on match hints.

3.2 Optimization

3.2.1 Sampling-driven FCP

As explained in Section 3.1.1, our limiting score is not quantitatively correlated with filtering ratio. Thus, YourSQL requires a further check to see if early filtering for a candidate table would really be beneficial in terms of execution time for a given query. A dedicated ISC task called *sampler* is used in YourSQL to provide a quantitative estimation of filtering ratio. This sampler is the same as the hardware filter functionality-wise. It is the range of scan that is different. The former scans a small portion of the table, as the name suggests, while the latter does the whole table. Filtering ratio is estimated by the filtering results from the sampler, i.e., by dividing the number of match pages by the number of scanned pages. If the estimated ratio is higher than the threshold, FCP is enabled. As the precision of sampling is substantially influenced by data distribution and sampling size [4], we left several tuning parameters in the sampler such as sampling size and sampling area.

The key challenge for our sampling-based estimation is to determine an adequate threshold to enable FCP. This threshold depends on storage performance as well as early filtering target in a query. More precisely, relative speed between sequential and random read operations matters since the early filtering approach is likely to rely more on random read operations. Our analysis suggests queries can be categorized into four according to the types of early filtering target. These four are a single table query, a join query with no subquery, a query with a derived table, and a query with a single or multiple subqueries. The last two categories differ in that a derived table is always in the FROM clause. The early filtering targets of the first two categories come from the FROM clause, and those of the remaining categories come from a derived table and a subquery, respectively.

For queries in the first category, the early filtering approach can outperform full scan as long as the time to read relevant pages with random I/O operation is less than the time to read the whole table with sequential I/O operation. The asynchronous random read operation in our target system is faster than sequential read operation as far as the amount of random reads falls below 30% of the whole table size. Thus, we set the threshold for this category at 0.3. For queries in the third category, we simply set the threshold at 1.0. A derived table functions as a reference table from which rows are retrieved. Therefore, reducing the size of the reference table via early filtering is expected to accelerate queries in this category.

Contrary to the prior two, the thresholds for queries in the second and the last categories cannot be predefined. Rather, they should be calculated on the fly. However, various factors come into play such as the size of each join table, the filtering ratio of the early filtering candidate, the size and number of matching row combinations passed to the next inner join loop, and MySQL's join buffer size [4]. Currently, we empirically determined these thresholds to be 0.7 based on our target system performance and the analysis of TPC-H queries in these categories. To calculate proper thresholds for queries in the second and the last categories, we should build a refined cost model, which remains as future work.

We note that there are two exceptions for setting thresholds. The first one is that if the smallest table is chosen as the candidate in the case of the second category queries, we set the threshold at 0.3 again. In this case, query plans of MySQL and YourSQL differ in the access method to the first table only. The second one is that we set the threshold at 0 for the last category queries if subqueries can be materialized [2] (i.e., no early filtering). This is because a query execution with materialization is fast enough and thus potential improvement by early filtering is marginal.

3.2.2 Software Filtering

As explained in Section 3.1.2, hardware-filtered data may still contain false positives depending on filtering conditions. This is due to the limitation of our pattern matcher that performs byte-granular matching. For instance, `'8F 97'` in Section 3.1.2 would match sequences from `'8F 97 00'` through `'8F 97 FF'`, and thus all pages with date between `'1995-08-01'` and `'1995-12-31'` would be considered match.

For a further inspection of the hardware-filtered data, we introduce another software-based filtering layer in early filtering. The software filter is a simple ISC task comprised of several primitive arithmetic operations. Depending on the filtering conditions, the output of the hardware filter may be redirected to the software filter as an input as shown in Fig. 4. Code 1 shows the pseudo code of the software filter. Considering the case exemplified in Section 3.1.2, the hardware filter sets match hints of pages with the pattern `'8F 97'` to one, and passes such hints to the software filter. Given a page whose match hint is set to one, the

[4]TPC-H dataset is uniformly distributed and we found a sample fraction of 0.05 is already sufficient to give a precise (in the few % range) estimation of filtering ratio.

Code 1: Pseudo code of the software filter.

```
1  //IN: Match_Hint
2  //ARG: SW_KEY, OP_TYPE, BULK_MATCH_SIZE
3  //OUT: Match_Hint
4
5  DO Get Match_Hint from the hardware filter
6  FOR i = 0 TO i < BULK_MATCH_SIZE DO {
7    IF Match_Hint[i] != 0 THEN {
8      Do Match_Hint[i] to 0
9      FOR j = 0 TO j < # of rows in Page i DO {
10       FOR each filter column in row j DO {
11         IF 'the column value OP_TYPE SW_KEY' is true THEN
12           DO Set Match_Hint[i] to 1
13           DO Break
14         END IF
15       }
16       IF Match_Hint[i] == 1 THEN
17         DO Break
18       END IF
19 }}}
20
21 DO Put Match_Hint to output
```

software filter checks if `l_shipdate` value of each row is really between `'8F 97 21'` (i.e., `'1995-09-01'`) and `'8F 97 41'` (i.e., `'1995-09-01' + INTERVAL '1' MONTH`). When it meets one, it stops further inspection for this page and the corresponding match hint would remain unchanged. Otherwise, the final match hint would be reset to zero.

While the software filter is capable of inspecting data relevancy to an arbitrary level, running it is costly because of an overhead involved in frequent memory access in the target SSD. Depending on the complexity of its code, the resulting overhead could be sizeable enough to degrade the overall performance. Balancing between the filter accuracy and overhead, we limit software filtering to a certain level to ensure an improved overall performance of YourSQL. In Section 4, we will discuss how much improvement is indeed achieved by introducing software filtering into early filtering.

3.2.3 Highly Accurate Bulk Prefetch

As explained in Section 3.1.3, every iteration of the ISC filters generates match hints whose size can reach up to `bulk_match_size`. Pages whose match hint is set to one would be a superset of pages that are necessary to answer a given query. Prefetching them must be highly beneficial, and thus YourSQL introduces two prefetch techniques.

The first technique is *bulk read* of match pages. Compared to a single page read, bulk read is expected to achieve significantly better performance by the increased queue depth. Therefore, YourSQL issues a bulk read request for match pages that belong to the same early filtering unit. The second technique is *aggressive prefetch*. The ISC filters scan the whole table while the storage engine reads match pages in bulk. Thanks to the relatively high internal bandwidth (see Section 4.1.2), the processing time for the ISC filters is shorter than the time for bulk read of match pages. Thus, we can get match hints for the subsequent early filtering units during the bulk read of pages in the current unit. This allows us to issue multiple prefetch requests based on match hints for each subsequent unit. For this, we create a dedicated thread called *prefetcher*. Fig. 5(c) depicts the query execution pipeline including the prefetcher. The prefetcher iterates a process of pulling match hints and issuing asynchronous read requests for match pages in bulk. This allows YourSQL to hide time for CPU jobs of query processing and thus improve the overall query performance. Linux asynchronous I/O functions such as `io_prep_pread()`, `io_submit()`, and `io_getevents()` are used for prefetch and '`innodb_flush_method=O_DIRECT`' option is added to `my.cnf`.

The prefetcher communicates with the storage engine via a Single-Producer/Single-Consumer (SPSC) queue. To avoid lock contention between them, this queue is implemented as a simple integer array. When a query starts, the prefetcher allocates the memory space for the prefetched pages. The total size comes to page_size × bulk_match_size × #queue_entry. Here, page_size and #queue_entry denote the page size of database instance (e.g., innodb_page_size in MySQL) and the number of the buffer entries for the SPSC queue, respectively. Every time the prefetcher pulls match hints, match pages corresponding to them are read in bulk and enqueued in the SPSC queue. Then, the storage engine dequeues the prefetched pages from the queue on an as-needed basis. At least two entries are required for the SPSC queue: One for the storage engine and the other for the prefetcher.

We note `bulk_match_size` is a key performance parameter. By this, overlapping execution of three processing elements (i.e., the host-side YourSQL, the prefetcher, and the ISC filters in Fig. 5(c)) is exploited in different levels. In the respective iterations, time for early filtering and time for CPU jobs of the host-side YourSQL are directly proportional to this parameter since larger `bulk_match_size` means more pages to filter and more rows processed by host CPU and vice versa. On the other hand, time for reading match pages in bulk is not. Up to a certain `bulk_match_size`, the time will remain unchanged because SSD can process the concurrent requests in parallel. Thus, different proportionality behaviors should be taken into account when this parameter is tuned. The bottom line is it should be tuned in a way that it maximizes the parallelism so that the overall execution time is minimized.

4. EVALUATION

This section evaluates the effectiveness of YourSQL in data-intensive workloads. We use queries of varying complexities, both synthetic and TPC-H genuine, with TPC-H dataset. The evaluations encompass not only the results from the standard benchmark but also the results obtained with different settings of system memory size as well as different optimization techniques. Also evaluated are the energy consumption of YourSQL and the effectiveness of early filtering over a column store.

Our extensive evaluations reveal that YourSQL becomes increasingly effective when a significant reduction of data transfer can be made at the earliest point possible—within a storage device. Overall, YourSQL can outperform a conventional database system up to 167× with significantly less energy consumption accordingly. Out of 22 TPC-H queries, five queries show sizable speed-ups and the overall processing time of the whole TPC-H queries is significantly reduced achieving a speed-up of 3.6×.

4.1 Experimental Setup

4.1.1 System Setup

We conducted experiments on a system comprised of a Dell PowerEdge R720 server and Samsung PM1725, a latest enterprise class NVMe SSD as shown in Fig. 6. The server is equipped with two Intel Xeon(R) CPU E5-2640 @2.50GHz

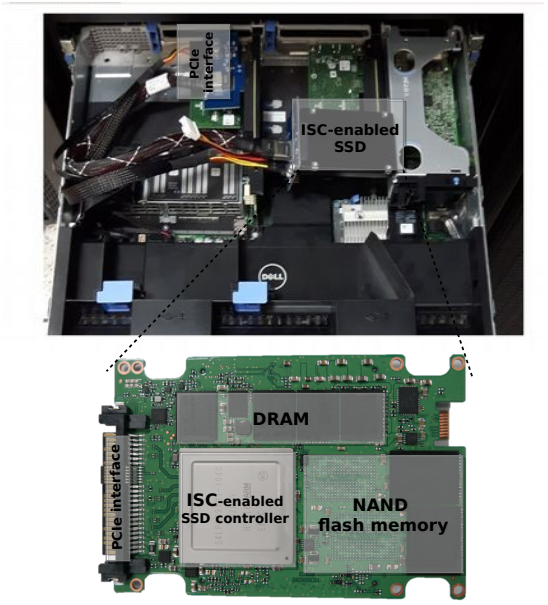

Figure 6: Dell R720 server with PM1725 SSD.

and 64 GB DRAM, and runs 64-bit Ubuntu 15.04. The ISC-enabled SSD is connected to the server via PCIe Gen3 ×4. Details of the SSD are listed in Table 3. It has multiple dedicated ARM cores, among which two ARM Cortex R7 cores are used as compute resources for ISC.

In order to measure the overall power consumption, we also connect the server through an external power measurement device, Power Manager B-200U [5]. It is capable of measuring power up to 2,400 Watts with an accuracy of 97% or higher. Power consumption of the whole server system is collected as a separate data sheet.

Table 3: PM1725 specification.

Host interface	PCIe Gen3 ×4
Device density	1 TB
SSD architecture	Multiple channels/ways/cores
Storage medium	Multi-bit NAND flash
Compute resources for ISC	Two ARM Cortex R7 cores @750MHz with L1 cache, no cache coherence
Hardware IP	Pattern matcher per channel that takes up to three 16 byte-long keywords

4.1.2 Basic Performance of the Target SSD

As seen in [13], the host read bandwidth of the target SSD reaches a plateau at around 3 GiB/s, limited by the host interface. By contrast, the internal read bandwidth surpasses such limitation, fully exploiting the superfluous internal bandwidth. With the pattern matcher enabled, the internal read bandwidth is lowered due to the overhead arising from controlling hardware IP, yet it still outperforms the host read by a few hundred MiB/s at a request size of 256 KiB or larger. It is important to point out that the observed bandwidth demand of MySQL fell much short of the maximum bandwidth limit of our target SSD. On the other hand, the bandwidth limit (internal bandwidth in this case) can easily be exhausted in the optimized scheme of YourSQL.

4.1.3 Baseline System and Workload

As a baseline system, we chose 5.5.42 version of MariaDB, an enhanced fork of MySQL. We tightly integrated the baseline system with the aforementioned ISC framework in the target SSD, *Biscuit* [13]. It is the first reported product-strength ISC implementation, and allows user tasks to run inside the high-performance NVMe SSD. We made non-negligible code changes to MariaDB to integrate its query planner and default storage engine, XtraDB, with Biscuit.

We evaluate the baseline system and YourSQL by queries with varying complexities. We use synthetic queries to evaluate them with the least query complexity, but the main workload is TPC-H. A scale factor of 100 was chosen. Once loaded into the database, the dataset becomes nearly 160 GiB in total. A default page size of 16 KiB for the baseline system is tuned to 4 KiB for YourSQL. While making the page size smaller degrades the performance of the baseline system, an improvement is expected in the case of YourSQL as it improves filtering accuracy. The available memory size for the system is adjusted to 10% of the total dataset size, out of which 75% is allocated for the database buffer.

Our prototype is an ISC-enabled variant of MariaDB, and currently works on page-aligned, uncompressed records. Thus, some may concern that the results of this paper may not be true in other database systems with unaligned data or sophisticated compression scheme [10]. However, YourSQL does not come with such limitations. Not only because our hardware filter is capable of matching patterns that span pages, but because compression does not affect its functionality. As long as patterns are passed from the host as arguments, it matches them. YourSQL can access the internal data structures and functions of the database system, and thus can pass the compressed patterns to the hardware filter.

4.2 Evaluation Results

4.2.1 Selection Query

For the initial evaluation of YourSQL performance, we use a rather simple, synthetic query, `Query 1`. Its filter predicates are so restrictive that the hardware filter can filter out all the unnecessary pages. Thus, no further inspection is needed with the software filter. As shown in Table 4, the execution time is reduced by a factor of seven, i.e., 7× speed-up with YourSQL. The 95% confidence interval of the measurement is less than 2% in both cases.

Table 4: Execution time for `Query 1`.

	MySQL	YourSQL
Execution time [sec]	15.9	2.2

This significant speed-up seen in YourSQL is attributed to the fact that only a small fraction of data needed to be transferred to the host. The filtering ratio of `Query 1` is 0.067, i.e., only 6.7% of the entire pages in the `part` table are needed for answering this query. On the contrary, MySQL requires the entire table to be transferred. The execution time of MySQL, therefore, is mainly driven by the I/O time the host takes to scan the entire table.

Actually, the execution time of YourSQL for a single table query is determined by the slower among early filtering by the ISC filters and bulk read by the host. These two run in a

pipelined manner as illustrated in Fig. 5(c). For a highly selective query like `Query 1`, the processing time of YourSQL is therefore mainly driven by early filtering. YourSQL's early filtering takes a full advantage of high bandwidth internal read, which reaches the maximum possible bandwidth easily with multiple processing of concurrent requests at a large request size. Even though we do not explicitly show in this paper, we also carried out measurements for two filtering queries from a recent related work [22]. While the reported speed-up stays below 3× in [22], about 10× speed-up is achieved with YourSQL thanks to the in-storage filtering and the read logic optimization.

4.2.2 Join Query

While the advantage of YourSQL for highly selective single table queries is distinctively shown from the previous section, it is yet to be shown that YourSQL is effective for those queries that access multiple tables, i.e., join queries. To evaluate its potential gains for join queries, we use `TPC-H Q2` (see `Query 2`). As shown in Table 5, the execution time is reduced by a factor of 44, a drastic speed-up even compared to the speed-up shown in Section 4.2.1. The 95% confidence interval is less than 10% of the reported value in both cases.

Table 5: Execution time for TPC-H Q2.

	MySQL	YourSQL
Execution time [sec]	1,104	25

This dramatic performance gain is attributed to the fact that the early filtering effect is magnified in the case of join queries. By placing the early filtering target first in the join order (e.g., the `part` table in this query), our system reduces the number of rows that must be read and processed at the initial stage of query processing. As a result, the number of I/O operations is significantly reduced, as well illustrated in the motivational example in Section 2.1 (c.f., Table 1). Early filtering utilizing high internal bandwidth, which is pipelined with the boosted host read, allows YourSQL to fully benefit from such a dramatic I/O reduction. In Section 4.2.4, we will provide a quantitative measure of the reduction of data as a consequence of the reduced I/O operations.

It is noted, however, this improved performance by FCP is not always guaranteed for join queries. Compared to a single table query, more various factors exemplified in Section 3.2.1 affect the number of I/O operations for a join query. Therefore, the decision of enabling FCP for a join query is far more complex than that for a single table query. As it will be shown in Section 4.2.4, only eight TPC-H queries are decided to enable FCP by YourSQL.

4.2.3 Available Memory Size

As the buffer size serves as a key performance factor in a database system, we studied its impact in query performance. We measured the corresponding speed-up after changing the system memory size from the default of 16 GiB to 8 GiB (0.5×), 32 GiB (2×), and 64 GiB (4×). We fixed the fraction of memory allocated for the database buffer at 75% of the system memory size. We use `TPC-H Q2`, which is more complicated than `Query 1`, and Fig. 7 gives the result.

In the case of a memory size of 8 GiB, the resulting speed-up is even enhanced up to 64×. As the memory size increases above the default, the resulting speed-up becomes lowered

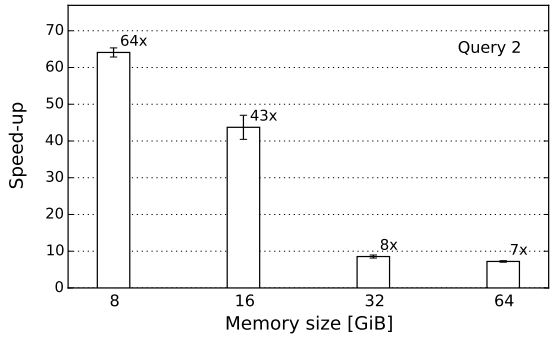

Figure 7: Speed-up with different memory size.

and reaches 7× speed-up when 64 GiB of memory is available for the system. This tendency implies that MySQL benefits more from a larger buffer than YourSQL, which is as expected since a larger buffer makes relative cost of read I/O decreased and therefore the impact of its reduction becomes smaller. On the other hand, when the memory usage becomes tighter, the relative cost of read I/O is increased and the impact of its reduction becomes more prominent as shown in the enhanced speed-up at 8 GiB.

4.2.4 TPC-H Results

For each of 22 TPC-H queries, we measured its execution time and ratio of data transfer reduction. The latter is a ratio of the amount of data transferred across storage network during a query with MySQL to that with YourSQL. This ratio, α_{saving}, illustrates quantitatively how much data is saved from being transferred as a result of FCP in YourSQL.

Fig. 8 shows the results, sorted by speed-up. Out of 22 queries, eight queries (Q14, Q2, Q10, Q8, Q9, Q17, Q12, and Q5) are FCP-enabled. The rest queries are not attempted to be leveraged with FCP for one of the following reasons. Q18 has no filter predicates, and thus is FCP-disabled. For the case of Q1, Q7, Q11, and Q21, the query engine does not enable FCP since the target table is small or their filter predicates are not restrictive (e.g., a single character comparison). Some are FCP-disabled because of the limitation of the pattern matcher in our target SSD (e.g., it cannot handle the "NOT LIKE" operator). The remaining queries are FCP-disabled because their estimated filtering ratio is lower than the threshold. With FCP disabled, relative performance of the rest 14 queries is simply unity.

The FCP-enabled eight queries show meaningful speed-up over 1.1×. Q14 exhibits the highest speed-up of 167×. As a mean of averaging speed-ups from the heterogeneous queries (i.e., different number and order of table accesses, different nominal execution time, and etc), geometric means of the top five accelerated queries as well as all queries are presented next to Q9 and Q22, respectively. As shown in the bottom panel of Fig. 8, the average speed-ups reach 15× and 1.9×, respectively for the two cases. The ratio of reduced data transfer is shown in the top panel of Fig. 8, which highly correlates with the resulting speed-up as expected. Despite the mean speed-up of over 15× from the top five queries, it is still debatable whether YourSQL accelerates the overall TPC-H performance. To this end, we also measured the overall execution time. Running all 22 queries takes almost

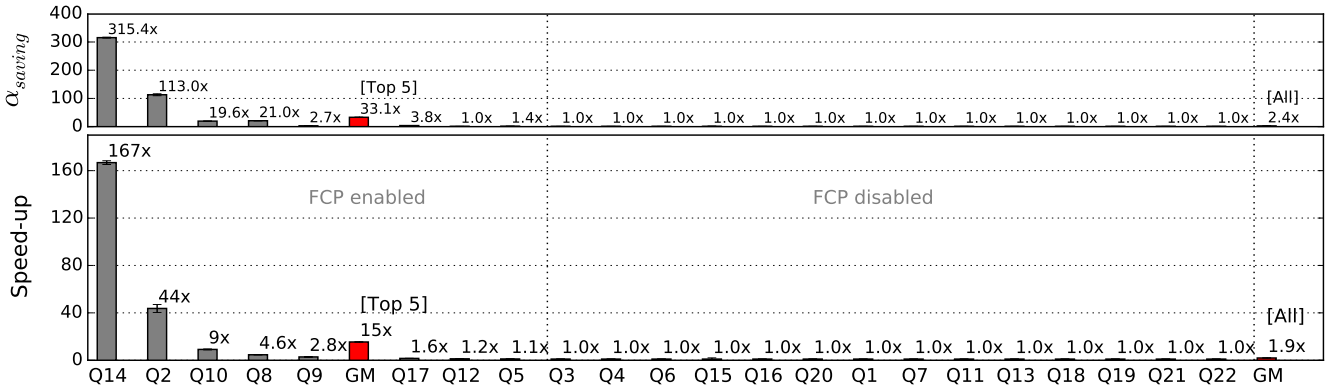

Figure 8: TPC-H results.

two days with MySQL, which is reduced to 13 hours with YourSQL (i.e., by a factor of 3.6). It is interesting to point out while only eight queries are accelerated with YourSQL applied, still 3.6× reduction of the overall execution time is achieved. We find such acceleration is indeed possible because the top five queries, which are highly data-intensive, take more than 70% of the total query execution.

4.2.5 Optimizations

In order to gauge the effect of optimization techniques presented in Section 3.2, we repeatedly ran the top five queries with the largest performance gains under different optimization schemes. Starting from the system with the hardware filter only, which we call Opt-P, we applied different optimization techniques, namely, software filtering and prefetch (HABP), one-by-one incrementally. We call the latter two as Opt-PS and Opt-PSH, respectively. Table 6 lists the optimization schemes with the respective configurations.

Table 6: Different levels of optimization.

Scheme	Configuration
Opt-P	Hardware filter
Opt-PS	Hardware filter + Software filter
Opt-PSH	Hardware filter + Software filter + HABP

Fig. 9 plots the speed-ups of YourSQL with different optimization schemes. In general, more optimizations yield higher speed-up, as one would naively expect since each optimization scheme is orthogonal to one another. The degree of improvement somewhat varies depending on the query. For example, Q14 sees over 3× improvement from Opt-P to Opt-PSH. Meanwhile, it is 1.3× improvement in the case of Q2. Overall, software filtering accelerates the top five queries up to 1.21×, and HABP does up to 3×. The biggest improvement seen in Opt-PSH implies that the host-side read operation was the limiting factor in accelerating the overall performance. Therefore, the biggest gain was made when the host-side read operation was optimized with HABP.

For comparison, we also include the result of ICP-enabled MySQL, which is shown next to the Opt-PSH result in Fig. 9. In order to enable ICP, we preprocessed the target table to create required secondary indexes before running each query. We note MySQL (with ICP) and YourSQL now follow the exactly same join order. The difference is the way

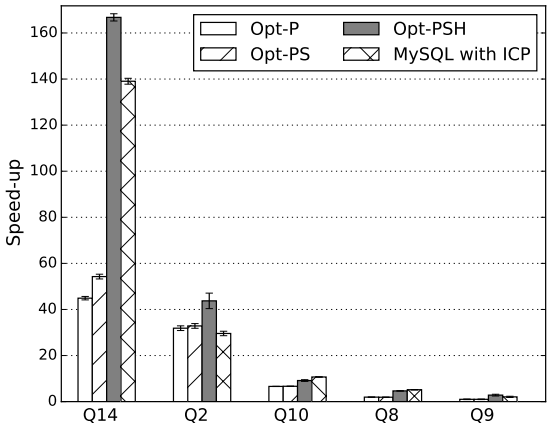

Figure 9: Speed-ups of the top five accelerated queries with different optimization schemes.

each accesses the first table. We see that the ICP-enabled MySQL outperforms the fully-optimized YourSQL in some cases (Q10 and Q8). This is because while index-based page lookup of MySQL is exactly accurate, ISC-filtered data may still contain false positives depending on filtering conditions. However, given that MySQL with ICP required preprocessing of the tables, which was not included in the measurement of execution time, the largest speed-up in each query is still achieved in YourSQL.

4.2.6 Power Consumption

We also evaluated YourSQL in the energy consumption aspect. The results from the previous sections highly indicate that YourSQL would be energy efficient due to the significant reduction of processing time. In order to check energy consumption in both systems explicitly, we measured the energy consumption during the processing of `TPC-H Q2`.

Fig. 10 shows the power consumption in the time interval of one second. As expected from the speed-up seen in `TPC-H Q2`, the duration of power consumption is markedly shorter in the case of YourSQL. It is interesting to point out the power consumption persists even after the query ends. We identify such residual power consumption is originated from the post query processing such as synchronizing the buffer

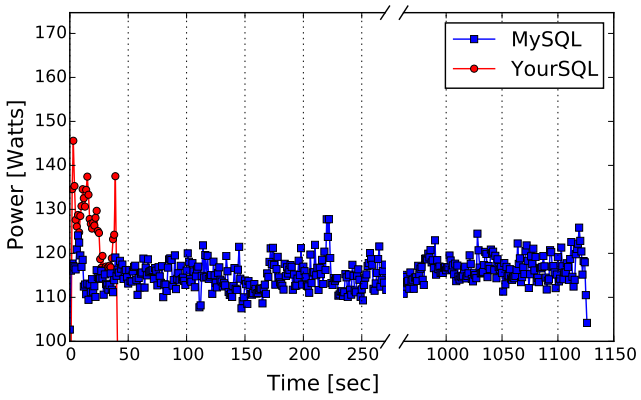

Figure 10: System power consumption during the execution of TPC-H Q2.

cache. YourSQL consumes more power in a given interval since it more fully utilizes the target SSD in terms of exploiting the SSD's bandwidth capability. Table 7 integrates the power consumption over time in Fig. 10 into the overall energy consumption in kJ. This table shows that YourSQL consumes nearly 24× less energy compared to MySQL.

Table 7: Overall energy consumption.

	MySQL	YourSQL
Total energy (kJ)	131.0	5.3

4.2.7 *Queries on Column Store*

The prior sections have shown the effectiveness of YourSQL in a row-oriented database system. To see if the early filtering approach in YourSQL is also effective for a column-oriented database system, i.e., a column store, we designed a micro-benchmark on a column store, and ran it with `Query 1`. In order to create column stores, we partitioned each raw table file of TPC-H dataset into segments. Each segment that serializes all of the values of a single column is stored along with row identification numbers (row IDs) as a separate file with 4K alignment.

We evaluated the potential gains with and without columnar compression. The compression is done by storing repeated column values only once along with the matched row IDs. Table 8 shows the relative file size and the filtering ratio of each segment before and after the compression. With the compression, the file sizes are reduced by 48% and 64%, respectively for the `p_size` and `p_type` segments. It is interesting to see that the filtering ratios in both files are unity, but reduced significantly due to data locality when compressed (c.f., 2% and 20%). It is already evident that early filtering must be ineffective in the raw column store from the lowest possible filtering ratio (i.e., 1).

Table 8: Relative file size and filtering ratio.

	`p_size`		`p_type`	
	Raw	Compressed	Raw	Compressed
Relative file size	1.00	0.52	1.00	0.36
Filtering ratio	0.96	0.02	0.99	0.20

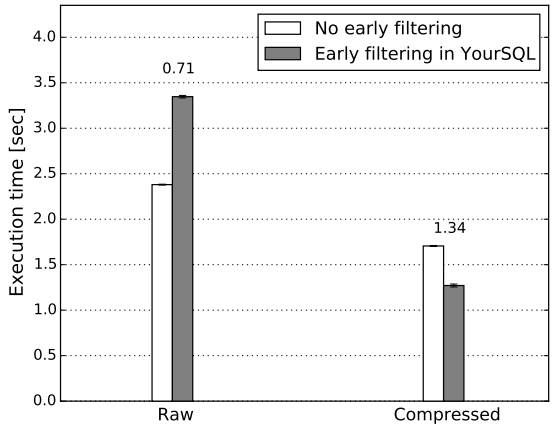

Figure 11: Execution time for the column stores.

We ran the micro-benchmark on the raw column store with and without early filtering. Without early filtering, we fully scanned `p_size` and `p_type` segments in bulk of 64 pages, and combined columns from both segments using row IDs. Based on the row IDs, we then located `p_mfgr` values from the corresponding segment. In the case of early filtering, we followed the same process depicted in Fig. 5(c). We applied the hardware filter to `p_size` and `p_type` segments each, and read match pages in bulk. Locating `p_mfgr` values is done identically in both cases. We repeated the micro-benchmark on the compressed column store. We note the full scan was no longer necessary in the first case (i.e., no early filtering) since it stopped scanning when the columnar value meets the filtering condition (i.e., `p_size` = 15).

Fig. 11 shows the resulting execution time. The value above the histogram represents the achieved speed-up. As expected, the early filtering approach for the raw column store results in a considerable performance drop, i.e., a factor of 0.71×. We note, however, that our sampling heuristic, if applied, would have advised to run without early filtering because of the low filtering ratio. For the compressed column store, the early filtering approach accelerates the query execution resulting in the speed-up of 1.34. Two important observations can be made based on these results. First, the early filtering approach can hardly benefit a column store without compression (or with low data locality) due to low filtering ratio. Second, data structure (i.e., row vs column and/or raw vs compressed) is an important factor to determine the effectiveness of early filtering and its extent.

5. RELATED WORK

Reflecting the trend toward big data, interests in *Near-data processing* (NDP) [7, 8, 9, 15, 16, 17, 18, 20] have soared again recently, as can be evidenced in the widely popular Hadoop framework [21]. Big data analytics is data-centric in nature (rather than compute-centric), and hence, minimizing unnecessary I/O is one of the key challenges. High-end appliances such as Oracle Exadata [19] and IBM Netezza [12] employ intelligent storage servers between disk enclosures and database servers. The key enabler in these storage servers is to identify irrelevant data blocks for a given query and avoid their transfer. In particular, Netezza minimizes data traffic with the help of commodity FPGAs. Similarly, a

research system called Ibex [22] employs an FPGA between the database and an SSD, and offloads filter and aggregation queries to the FPGA. In these cases, however, the amount of data transferred out of the storage devices is not reduced, because data must be sent to filter servers or FPGAs first for relevancy check. In contrast, YourSQL eliminates unnecessary data traffic at the earliest point possible—from within a storage device.

As SSDs become commonplace and their capabilities grow, researchers from academia and industry alike started to pay attention to SSD based NDP. Different from database vendors, they focus on programmable data processing within an SSD [11, 14]. Such an approach is favorable for two reasons: Additional (aggregate) compute capability in SSDs and higher internal bandwidth inside an SSD compared to the external SSD bandwidth limited by a typical host interface (like SATA and SAS). Building on this observation, Do et al. [11] wrote a database system prototype on an SSD, where the modified SSD firmware performs user queries. Kang et al. [14] extended the Hadoop framework to turn SSDs into distributed data nodes. Compared to these earlier studies, this work not only targets a latest high-performance NVMe SSD, but describes a much more fully integrated database system design capable of executing complex queries.

6. CONCLUSIONS

This paper described YourSQL, a database system built on top of a state-of-the-art SSD architecture. We pursued achieving cost-effectiveness as well as high performance of data analytics by realizing an end-to-end datapath optimization that spans both host system and SSD. To the best of our knowledge, it is the first product-strength database system building on the concept of ISC on commodity NVMe SSDs. Moreover, our prototype system proves that full table scans could be accelerated with the help of additional, near-data compute resources in the SSDs.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments, which helped improve the quality of this paper. Many researchers and individuals have contributed at various stages to this work, including: Boncheol Gu, Moonsang Kwon, Man-Keun Seo, Woojin Choi, and Chanik Park.

8. REFERENCES

[1] Explain output format. `http://dev.mysql.com/doc/refman/5.7/en/explain-output.html`.

[2] Faster subqueries with materialization. `http://guilhembichot.blogspot.kr/2012/04/faster-subqueries-with-materialization.html`.

[3] Index condition pushdown optimization. `http://dev.mysql.com/doc/refman/5.7/en/index-condition-pushdown-optimization.html`.

[4] Nested-loop join algorithms. `http://dev.mysql.com/doc/refman/5.7/en/nested-loop-joins.html`.

[5] Power manager. `http://www.powermanager.co.kr`.

[6] Understanding nested loop joins. `https://technet.microsoft.com/en-us/library/ms191318(v=sql.105).aspx`.

[7] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, and C. Park. Intelligent SSD: a turbo for big data mining. In *CIKM*, pages 1573–1576, 2013.

[8] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *Micro*, 34:36–42, 2014.

[9] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: A case for intelligent ssds. In *ICS*, pages 91–102, 2013.

[10] H. A. Desphande. Efficient compression techniques for an in memory database system. *IJIRCCE*, 3(9):8975–8983, 2015.

[11] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *SIGMOD*, pages 1221–1230, 2013.

[12] P. Francisco. Ibm puredata system for analytics architecture. *IBM Redbooks*, pages 1–16, 2014.

[13] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for near-data processing of big data workloads. In *ISCA*, pages 153–165, 2016.

[14] Y. Kang, E. L. Miller, and C. Park. Enabling cost-effective data processing with smart ssd. In *MSST*, pages 1–12, 2013.

[15] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, 1998.

[16] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee. Fast, energy efficient scan inside flash memory. In *ADMS*, pages 36–43, 2011.

[17] Y.-S. Lee, L. C. Quero, Y. Lee, J.-S. Kim, and S. Maeng. Accelerating external sorting via on-the-fly data merge in active SSDs. In *HotStorage*, pages 14–14, 2014.

[18] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB*, pages 62–73, 1998.

[19] M. Subramaniam. A technical overview of the oracle exadata database machine and exadata storage server. *An Oracle White Paper*, pages 1–43, 2013.

[20] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *FAST*, pages 119–132, 2013.

[21] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[22] L. Woods, Z. Istvn, and G. Alonso. Ibex-an intelligent storage engine with support for advanced sql offloading. *the VLDB Endowment*, 7(11):963–974, 2014.