Accelerating In-Memory Database Selections Using Latency Masking Hardware Threads

PRERNA BUDHKAR, ILDAR ABSALYAMOV, VASILEIOS ZOIS, SKYLER WINDH, WALID A. NAJJAR, and VASSILIS J. TSOTRAS, University of California, Riverside, USA

Inexpensive DRAMs have created new opportunities for in-memory data analytics. However, the major bottleneck in such systems is high memory access latency. Traditionally, this problem is solved with large cache hierarchies that only benefit *regular* applications. Alternatively, many data-intensive applications exhibit *irregular* behavior. Hardware multithreading can better cope with high latency seen in such applications. This article implements a multithreaded prototype (MTP) on FPGAs for the relational selection operator that exhibits control flow irregularity. On a standard TPC-H query evaluation, MTP achieves a bandwidth utilization of 83%, while the CPU and the GPU implementations achieve 61% and 64%, respectively. Besides being bandwidth efficient, MTP is also 14.2× and 4.2× more power efficient than CPU and GPU, respectively.

CCS Concepts: • **Hardware** → **Hardware accelerators**; • **Information systems** → *Query operators*; • **Computer systems organization** → *Multiple instruction, multiple data*; *Reconfigurable computing*;

Additional Key Words and Phrases: Hardware multithreading, database, selection operator, FPGA accelerator

ACM Reference format:
Prerna Budhkar, Ildar Absalyamov, Vasileios Zois, Skyler Windh, Walid A. Najjar, and Vassilis J. Tsotras. 2019. Accelerating In-Memory Database Selections Using Latency Masking Hardware Threads. *ACM Trans. Archit. Code Optim.* 16, 2, Article 13 (April 2019), 28 pages.
https://doi.org/10.1145/3310229

1 INTRODUCTION

The rapidly decreasing cost of DRAM has enabled in-memory analytics solutions, where fairly large datasets can be stored and processed entirely in memory. Many commercial in-memory databases (Oracle TimesTen [50], SAP HANA [28], MS SQL Hekaton [23], IBM BLU [15], MemSQL [68]) as well as academic research prototypes (HyPeR [45], Peloton [3]) have been recently developed.

While memory capacity continues to increase, the past decade has seen a stagnation of processor clock speeds due to the end of Dennard scaling and power dissipation concerns. This leaves parallelism as the only option to achieve fast processing for the growing amount of memory-resident data. Two approaches have been considered for leveraging parallelism: (1) off-the-shelf multicore

New paper, not an extension of a conference paper. This research was partially supported by NSF grants: IIS-1447826, IIS-1527984 and NSF CCF-1219180.
Authors' addresses: P. Budhkar, I. Absalyamov, V. Zois, S. Windh, W. A. Najjar, and V. J. Tsotras, University of California, Riverside, Department of Computer Science & Engineering, Riverside, CA, 92521, USA; emails: {pbudh001, iabsa001, vzois001, windhs, najjar, tsotras}@cs.ucr.edu.

1544-3566/2019/04-ART13
https://doi.org/10.1145/3310229

architectures (including CPUs and GPUs) [14, 29, 41] or (2) customizable architectures (such as CPUs with FPGAs) [1, 46, 64, 73].

The main factor influencing in-memory processing performance is the growing gap between the memory bandwidth and the speed of the processing unit [17] (the so-called memory wall). This is even more important for multicores, given their higher clock speeds. Multicore CPUs addressed this problem by introducing large *cache hierarchies* that rely on data locality. This solution does not come for free: cache hierarchies can take up to 80% of the chip area. Caching is a major limiting factor on the number of cores that can be accommodated on a single chip as well as a primary contributor to energy consumption through leakage current. GPUs offer a different solution that leverages massive SIMD parallelism and specialized high-bandwidth memory (i.e., GDDR). Similarly modern CPUs introduced SIMD support to enable data level parallelism. However, these architectural solutions still inherently rely on data locality.

At the same time, many data-intensive computational problems are moving away from data locality towards irregular memory access behavior. Such irregularity can be of two types: (1) *Dataflow irregularity* is caused by the indirection in the data access, leading to cache misses. For example, workloads from complex database analytics, hash joins [35], data mining, graph algorithms, bioinformatics, and so on exhibit a very poor degree of spatial and/or temporal localities and do not benefit from large cache hierarchies. (2) *Control flow irregularity* is caused by the dynamic control flow and leads to branch mis-prediction; this contributes to a large fraction of stall time on CPUs [62, 74] or thread divergence on GPUs [70]. An example that exhibits such behavior is the *selection* operator. While the *selection* operator has a seemingly regular execution pattern (i.e., exhibits spatial locality), it often leads to *control flow* irregularity while evaluating the selection predicates (see Section 2.1). Since the selection operator typically appears early on within a query plan—right above the data scan operator—its performance directly affects the total runtime of the whole query [62].

Clearly, cache hierarchies do not provide an effective solution for irregular memory accesses. As a consequence, long latencies are introduced to fetch data from the memory. An alternative approach is to *mask* these long latencies by using hardware *multithreaded* execution [40, 66, 76]. Several multithreading models (*simultaneous, fine-grained, coarse grained*) have been proposed. They can be categorized by how temporally close instructions from different threads may be executed. On general purpose processors, executing multiple threads concurrently requires saving the full context of each thread. This limits the amount of parallelism that can be achieved on these systems.

In a custom architecture (e.g., FPGA) where the datapath is designed for a small number of predefined operations, the required context for each thread is much smaller than in a general-purpose CPU and, hence, more threads can be supported. In this model, parallelism is limited only by the number of active threads (ready, executing, or waiting). Furthermore, when the application consists of a large number of concurrent threads, this model supports the masking of memory latency. We have recently applied this multithreading approach to implement an in-memory hash join algorithm [35] and the group-by aggregation [5]. Those results demonstrated up to 10× higher throughput over the best multi-core software alternatives with comparable memory bandwidth.

In this article, we implement a multithreaded selection engine on custom hardware. While selection is a different operator, there are subtle similarities with the hash-join and group-by aggregation operators. The hash join is implemented in two stages: (1) the *build* phase, which scans the first table (usually the smaller one) and builds a hash table on the join key, and (2) the *probe* phase, during which each tuple (row) in the second table is accessed and a probe to the hash table (for matches) is performed. In the group-by aggregation, a hash table is used to build a list of <*key, count*>. Each tuple of the aggregated relation is accessed and its key is first searched in the hash table; then it either updates a key's count (if the key is found) or inserts a new entry <key, count>

into the hash table. Note that both the aforementioned operators use hardware multithreading to mask the latencies caused by *dataflow* irregularities.

While performing a selection (which typically consists of many predicates), our custom engine issues a separate thread for each tuple, thus generating a large number of independent and concurrent threads. Moreover, it also facilitates fine-grain data access, which means that a particular thread fetches only the first column (attribute) value needed to evaluate a given selection predicate. Based on the result of predicate evaluation, this thread will fetch the next column value from this tuple only if it is needed. To facilitate such multithreaded fine-grained data access, we also need a memory sub-system that can provide near-peak bandwidth for data accesses at this fine granularity.

As a proof-of-concept, we implement a multithreaded prototype of the selection engine (referred hereafter as **MTP**) on the Micron (Convey) HC-2ex FPGA machine. The HC-2ex is a heterogeneous computing platform—a system consisting of an off-the-shelf x86 CPU paired with four FPGAs and a highly parallel memory system.

MTP offers several advantages. Prior FPGA-based solutions [22, 56, 72, 79] stream the whole database through the execution engine. Similarly, row-oriented software selection applications, in the worst case, bring in the entire cache line to access a single column or fetch the column values for the tuples that already were disqualified by the selection predicate in a columnar storage layout. However, MTP reduces the number of memory accesses and improves bandwidth efficiency by accessing *only* the values that are necessary to evaluate a predicate. Furthermore, the *storage independence* nature of selection enables efficient analytics on transactional row-oriented data without replicating or converting it to a columnar representation. Moreover, unlike traditional multithreading architectures [7, 8], MTP requires a very small context per thread (*super-lightweight threads*), which allows us to generate a large number of in-flight requests, further improving latency masking. Finally, previous hardware accelerated proposals [72] required a separate hardwired implementation for each specific query. Our MTP engine is *runtime programmable* for predicates arranged in disjunctive normal form (DNF) and thus avoids prohibitive hardware reconfiguration or synthesis overheads.

We compare MTP's performance to that of an optimized scalar CPU, a vectorized CPU (SIMD), and a GPU implementation in terms of both raw and bandwidth normalized performance. Both CPU and GPU run at much higher clock frequencies (GHz compared to an FPGA's MHz) and have more dedicated hardware. Given the memory-bounded nature of the problem, our focus is on achieving better bandwidth utilization rather than focusing exclusively on the raw performance. Better efficiency leads to better scaling, where performance increases with more resources. Our experimental evaluation shows that in an important subset of the parameter space (often observed in practical workloads), MTP did fairly well when compared with the aforementioned architectures. On a standard TPC-H query evaluation, MTP achieves a bandwidth utilization of **83%** while the CPU and the GPU implementation achieves 61% and 64%, respectively. Besides being bandwidth efficient, MTP is also **14.2**× and **4.2**× more power efficient.

The following summarizes the contributions of this article:

- We present MTP, a novel in-memory selection engine based on a runtime programmable, hardware multithreaded architecture that is independent of physical data storage layouts.
- We thoroughly evaluate the throughput of our MTP selection engine against the state-of-the-art CPU (both scalar and vectorized) and GPU implementations using synthetic and TPC-H benchmark data.

This article is organized as follows: Section 2 summarizes basic selection evaluation algorithms and describes how they are implemented on CPUs and GPUs. Section 3 describes the design of the

MTP selection engine and also discusses different factors that affect MTP throughput, followed by experimental results in Section 4. Section 5 describes the related literature and conclusions appear in Section 6.

2 BACKGROUND

2.1 Selection Evaluation Algorithms

We assume OLAP setting where the select query directly operates on the input array of records and writes qualified tuples into a new output array, *out[]*. This is a common scenario for query plans where selection is pushed all the way down to the data scan operator. Depending upon the selectivity, some platform specific implementations build an index that fetches predicates based on the previous evaluation. However, in this work, we do not assume the availability of any such index. For each tuple t_i, we will evaluate k predicates. There are three commonly used selection evaluation algorithms. Listing 1 presents the most straightforward way of implementing selection. It shows the *branching scan* method for a conjunctive query using '<' comparison. The evaluation continues until one of the attributes is assessed to be *false* or all the attributes of a query have been examined. This technique is often called a "short-circuit evaluation," because the computation of further predicates can be skipped when the first predicate is already evaluated to false. The logical-AND (&&) operator is typically compiled into k conditional branch instructions. Assuming that the predicates have increasing selectivity, this method is optimal in terms of processing cycles. However, it was shown that on CPUs it leads to heavy branch mispredictions, causing considerable performance penalties [19, 62].

```
for (i=0; i < num_tuples; i++)
  if((t_i[0] < v_1) && .. (t_i[k] < v_k)
   { out[j++] = i; }
```

Listing 1. Algorithm - Branching Scan.

```
for (i=0; i < num_tuples; i++)
  if(t_i[0] < v_1 & .. t_i[k] < v_k )
  { out[j++] = i; }
```

Listing 2. Algorithm - Predicate Scan with Bitwise-AND (&).

```
for (i=0; i < num_tuples; i++)
  { out[j] = i;  j += (t_i[0] < v_1 &
       .. & t_i[k] < v_k );}
```

Listing 3. Algorithm - No-Branch.

An alternative implementation, presented in Listing 2, uses bitwise-AND (&) instead of logical-AND (&&). This approach reduces k conditionals to a single branch. Here, all predicates for a given tuple, t_i, are evaluated and then, depending upon the result of the evaluation, a branch is executed. This method reduces branch misprediction penalties at the cost of higher computational work.

Finally, a no-branch implementation [62] is shown in Listing 3. This approach completely eliminates branch mispredictions penalties by increasing computation costs. The techniques presented in Listings 2 and 3 either reduce or completely eliminate the branches. Yet, both approaches process all predicates and miss the opportunity to skip irrelevant evaluation as in the *branching scan*.

2.2 Implementations on Multithreaded Architectures

Multithreaded architectures seamlessly exploit thread-level parallelism and improve the performance of any of the aforementioned algorithms by partitioning the input over multiple threads [51]. Most modern processors support *simultaneous multithreading* (SMT) that allows multiple instructions from different threads to execute in the same cycle. This implies that all threads (contexts) are active at the same time and there is no notion of context switching. Examples include the latest Intel and AMD processors that are 2-way SMT, supporting 2 threads per core, and a recent SPARC64, which is 8-way SMT [54]. There is a vast literature [47, 48, 83, 85] that further enhances SMT with pre-fetching techniques. These techniques mainly include helper threads and cores [58] that prefetch cache blocks [83] to tolerate the memory latency, provide better exception handling [85], and clever instruction fetch [4] to prevent I-cache misses and instruction stream interleaving [48] to attain better core utilization.

However, *temporal multithreading* is another form of multithreading that allows only one thread of instructions to execute in any given pipeline stage. It is further classified into fine-grained and coarse-grained multithreading. In the former case, a thread switches the execution at a fixed time interval (Tera MTA now Cray XMT [38]). In the latter, switching is done during the long memory latency (SUN UltraSparc T5 [29]).

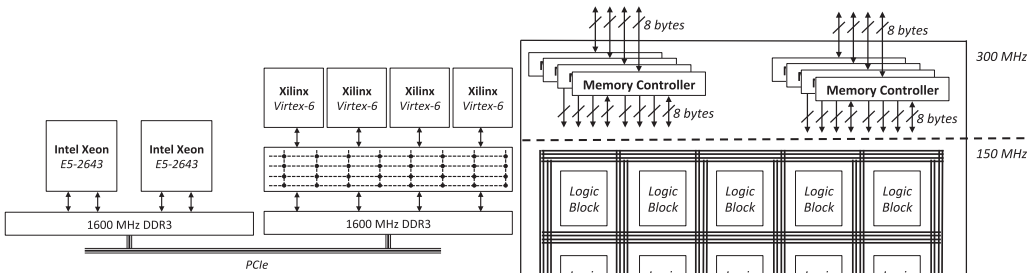
All the above architectures support a relatively *small* and predetermined number of threads (up to 128 on the MTA and up to 8 threads per core on the UltraSparc T5). This is because for each thread a heavy context (i.e., current state) in the form of registers, program counters, stack pointers, and so on is maintained. These resources are either used to save the context of *all* active threads (in SMT) or the waiting threads that switch the execution to the ready thread (in temporal multithreading). As a result, the number of threads that can run in parallel is limited by the total number of physical cores (or hardware contexts for CPUs with hyper-treading).

In the MTP model, the architecture is designed for a narrow range of specific applications and thus requires very small thread context. The execution of a thread is pipelined and, hence, several threads can be *active* in different stages of the pipeline. Unlike in other models, threads in MTP are not persistent: a thread is created when the evaluation of the selection query is started on a new row (tuple) and that thread is terminated when the evaluation concludes and the result is written for that row. The thread execution could lead to the evaluation of every attribute in a tuple or a single attribute. As a thread is stalled during a long latency memory access, other ready threads are executing on the same datapath. The only limit to the number of active threads (ready and waiting) is the total size of the buffers in the datapath and the memory interface.

We prototype MTP on the Convey HC-2ex machine, shown in Figure 1(a). We choose the Convey HC-2ex machine, because it is the only FPGA platform we know of that provides near-peak bandwidth for random memory accesses using Scatter-Gather DIMMS. Instead of a cache line access, these types of DIMMS allows us to gather/scatter values of strided access from multiple memory banks using a single memory request. However, note that this multithreaded paradigm is independent of any platform. As long as the memory system supports non-sequential memory accesses, the multithreaded architecture can efficiently mask memory latency.

2.3 Convey HC-2ex Prototype System

The Convey HC-2ex is a heterogeneous platform that offers a shared global memory space between the CPU and FPGA regions. It uses SG-DIMMs (scatter-gather DIMMs) to be able to access 8-byte values randomly, and at near-full throughput. As shown in Figure 1(a), the memory is divided into regions connected through PCIe with portions closer to the CPU and portions closer to the FPGAs. Each processor (CPU or FPGA) can access data from both regions, but data accesses across PCIe



(a) The Convey HC-2ex software and hardware regions **(b)** Convey HC-2ex FPGA AE Wrapper

Fig. 1. The Convey HC-2ex architecture is divided into software and hardware regions, as shown in (1(a)). Each FPGA has 8 memory controllers that are split into 16 channels for the FPGA's logic cells, as shown in (1(b)).

are significantly longer. The software region has an Intel Xeon E5-2643 processor. The hardware region has 4 Xilinx Virtex6-760 FPGAs, called application engines (AE), connected to the global memory through a full crossbar. The crossbar not only interfaces the AEs to the memory modules but also supports the in-order return of all memory requests. MTP utilizes this re-ordering to serve all the concurrent threads in the order of their arrival.

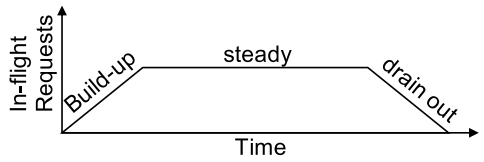
Each AE also has 8 64-bit memory controllers (MCs) running at 300MHz (Figure 1(b)) that support a total of 16 DDR2 *memory channels*. MCs provide a highly parallel connection between the AEs and the coprocessor physical memory [77]. The MCs also translate virtual to physical addresses on behalf of the AEs. The hardware logic on each AE run in a separate 150MHz clock domain to ease timing. On the HC-2ex, all the reads and writes to the memory are done through MCs. Each channel supports independent and concurrent read-write accesses to memory.

2.4 Latency Masking Multithreaded Architecture

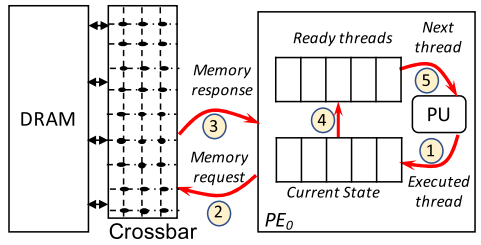
As shown in Figure 2(a), a latency masking multithreaded setup typically has three stages of execution: (1) build-up (2) steady state, and (3) drain-out. Figure 2(b) shows different components of multithreaded implementation. In a *build-up* stage, the *processing unit* (PU) of each *processing engine* (PE) generates multiple requests to the DRAM. Each request corresponds to a unit of work done by a thread. For instance, while executing a select query, each row is a *thread* and evaluating a particular column of a row is a unit of work.

The number of requests that are generated by the *PU* must be commensurate with the number of in-flight requests required to mask memory latency. This number is governed by *Little's Law* [10], which states that *Concurrency* (C) = *Performance* (P) *x Latency* (L). On the Convey HC-2ex, the average memory latency to the DRAM is ~100–200 6ns cycles ($L = 600ns\ to\ 1200ns$). Since *selection* is a memory bound operation, performance is measured by achieving peak memory bandwidth ($P = 76.8GB/s$). Substituting request size (8 bytes) and independent channels (64) in the aforementioned equation ($P * L/(8 * 64)$) yields C = 90 to 178 in-flight requests per memory channel. In our simulations, we verified this number to be 150. Moreover, our internal FIFOs are capable of maintaining up to 512 in-flight requests. Therefore, as long as PU can generate/sustain 150+ memory requests per channel, we can achieve high memory bandwidth utilization.

Using this observation, we conclude that hiding memory latency is a pure function of inflight-requests. In a steady state, a kernel/core can maintain sufficient in-flight requests that can hide the memory latency. However, towards the end of execution, when threads in a pipeline are terminating, the number of in-flight requests start decreasing and can no longer mask the latency. If the



(a) The execution starts with build-up state (new requests are generated), reaches steady state (in-flight requests > latency) and finally drain-out state (in-flight requests < latency)



(b) Components of multithreaded implementation, numbers represent execution steps. Design is scaled by adding as many PEs as possible.

Fig. 2. Multithreaded architecture details.

steady state execution is longer than the other two states, then the build-up and the drain costs are amortized.

3 MTP SELECTION ENGINE

For our MTP implementation, we take advantage of the *early termination* provided by the *branching-scan* algorithm without incurring branch misprediction penalties and independently of the data layout. MTP spawns multiple lightweight threads (one for each tuple) that allows it to do fine-grained data access (fetching only the values needed for the query evaluation on each tuple). This section describes the details of the MTP design and its workflow on FPGA. Later, we discuss several factors affecting the throughput of the MTP custom datapath, hereafter referred to as the *selection engine.*

3.1 Predicate Control Block

The selection engine is capable of processing different selection queries **without** needing to reconfigure the logic on the FPGA. To program the selection engine at runtime for different queries, we arrange our queries in a standard *disjunctive normal form* or DNF, which is a common way of storing queries in data warehouses [42]. We build the DNF by parsing a query and creating a data structure called the Predicate Control Block (PCB).

The PCB size is linear to the number of predicates in a query as it maintains a single record for each predicate. Each record of the PCB stores the comparison operator (e.g., less than, equal, etc.), the constant, and two column offsets (*True or False*) that are needed to direct further evaluation.

Figure 3 shows a sample SQL query and its corresponding PCB. It has three rows that correspond to the three predicates in the query. We parameterize all three parts of our predicate: (1) the column ID, (2) the comparison operator, for which one out of six possibilities is selected (=, <>, <, >, <=, >=), and (3) a constant value. Given a tuple from *OrderDetails*, after evaluating the first

```
SELECT (*) FROM OrderDetails
WHERE (UnitPrice > 5 AND
Quantity > 10)
OR (TotalPrice > 100)
```

(a) Sample SQL query

Column	Opr	Const	Col_True	Col_False
UnitPrice	>	5	Quantity	TotalPrice
Quantity	>	10	TRUE	TotalPrice
TotalPrice	>	100	TRUE	FALSE

(b) Corresponding Predicate Control Block (PCB)

Fig. 3. Figures 3(a) and 3(b) describe the translation of control flow to data flow in the selection operator.

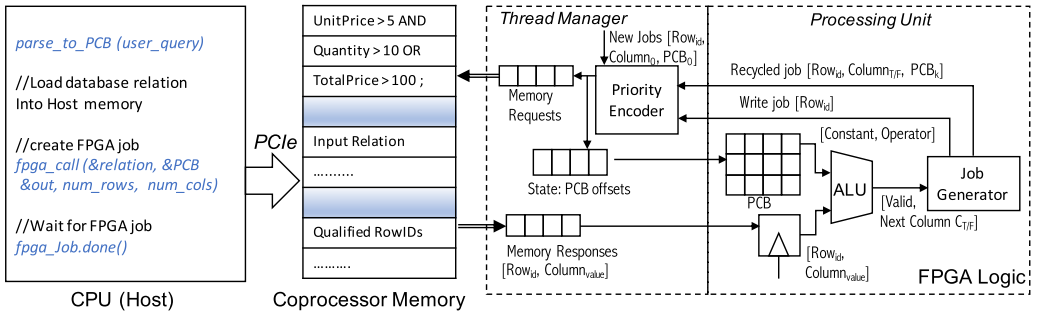


Fig. 4. Selection Accelerator Engine: showing different building blocks and memory channels that read/write from memory. Double-edged arrows indicate memory transaction.

predicate (*UnitPrice* > *5*) in the query, the next column offset requested is *Quantity* if the condition evaluates to true. Otherwise, MTP request *TotalPrice*. Notice that if *UnitPrice* > *5* is false, then we continue immediately with *TotalPrice* instead of finishing the next comparison. This provides *early termination* within clauses as well as queries.

3.2 Engine Workflow

Figure 4 shows the overall execution of the selection engine. The host CPU accepts the selection query, parses it, and builds its corresponding PCB. The database relation and the PCB are loaded in the host memory. Subsequently, using the Convey-provided memory allocators and datamover functions, the query and the relation are copied from the host to the co-processor memory through PCIe. Finally, the execution is off-loaded to the FPGA. During this dispatch process, local FPGA registers are programmed with pointers to the database table, an *out* array holding qualifying RowIDs, the PCB, the number of tuples, the row size (in number of columns), and the starting column offset (column ID in the first predicate).

The MTP implementation consists of two main blocks, namely, the *thread manager* (TM) and the *processing unit* (PU). Each block consists of queues that are used to mask memory latency while performing other tasks. The execution starts by requesting the PCB (*query*) from the DRAM. The PCB is stored locally by the *PU* in BRAM. Concurrently, the *TM* prepares a job for each thread. As mentioned before, a *thread* is a row and evaluating each column of a row is a *job*. The *TM* handles two types of read jobs: *new* and *recycled* jobs as well as the *write* jobs. Every row (thread) then requests a column that corresponds to the first predicate of a query. These requests are mandatory memory requests and are treated as *new* jobs. The *TM* tags each request with a Row_{id} that is used as a unique thread identifier by the system to keep track of all threads. As addresses are issued to the external memory, the *TM* registers the *PCB address* of the next predicate that needs to be evaluated as the *state* information. For new jobs it is PCB_0. Data is returned from memory in the same order it was requested. Data responses (Row_{id}, $Column_{value}$) are received in a FIFO. These responses, along with the thread state PCB_0, enter the pipeline handled by the *PU*. Using the PCB address, query

parameters are read from the PCB BRAM and set the ALU circuit for column evaluation. These parameters include an operator, a constant, and the next column offset for further evaluation. The outcome of an ALU can lead to three possible cases: (1) the current row qualifies a query condition, indicated by the "TRUE" opcode, that triggers the write-back process of the Row_{id}; (2) the current row disqualifies the query condition, indicated by the "FALSE" opcode, leading to thread termination; and (3) *recycled jobs* are generated to further evaluate the next attribute in a query. The recycled jobs $<Row_{id}, Column_{T/F}>$ are again submitted to the *TM* in a FIFO. The recycled addresses are calculated and different column values for the same row are requested from the memory. The state of recycled requests is a PCB address that is different for every row. This state information is maintained in a *state FIFO*.

At the steady-state, a job, new or recycled, enters the pipeline every cycle. This achieves the multithreading: multiple reads are waiting in queues for memory while other reads are processed. Priority is given to the *write* jobs, followed by *recycled* and *new* requests to ensure forward progress. Terminating recycled jobs will make room for new jobs. Additionally, this decision ensures that the design will not deadlock. New jobs will generate more recycled jobs that will eventually fill up the recycled job FIFO. The back-pressure from this FIFO will stall the memory responses as well as prevent the *TM* from issuing more new jobs.

The design uses FIFOs to store requests and responses from the memory system. The HC-2ex provides a fully buffered memory system that allows several in-flight requests per channel. The number of outstanding memory requests in a multithreaded system is a measure of the effective parallelism. A FIFO raises a stall signal whenever it is close to being full. This signal propagates backwards, stalling all the circuits upstream and eventually stopping fetching jobs from the memory. Obviously, a larger queue size would reduce the probability back-pressure being activated. In the current MTP design, the queue size is 512 entries.

3.3 Factors Influencing MTP Throughput

MTP selection engine implements the branching-scan algorithm and therefore exhibits a strong correlation between the measured throughput, *selectivity (S)* of the query and *predicate probability (p)*. We define S as the fraction of records in the input relation that satisfies the given query conditions, p is the probability of an individual predicate to be true, and (k) is the total *number of predicates*.

The evaluation order of predicates also plays a central role in achieving a quick query response time for selection. In this work, predicates are ordered by *increasing* selectivity, as described in References [71, 87]. This ordering scheme inherently assumes non-correlated predicates, simplifying our analysis. There have been other optimization techniques [37, 44] that determine the optimal evaluation order of selection predicates. MTP can take advantage of any ordering scheme that leads to a quick decision on a row.

We also group all the predicates on a single column into one; hence, in the rest of this article, we assume that every predicate is applied on a different column. With these assumptions, we calculate expected number of predicates evaluated for a given value of S and k and describe the variability in throughput based upon these predicate evaluations. We define all the parameters used in this analysis in Table 1.

Query evaluation involves reading column values from the memory and writing back the IDs of the rows that qualify. Therefore, the total work done to process a query can easily be partitioned into the amount of work done to evaluate the query, $\mathbf{W_e}$, and the work done to write back the ID of the qualifying rows, $\mathbf{W_{wr}}$. Note that W_e includes sending a request and receiving a response from the memory, evaluating the value of a column against a constant, and sending further requests for other predicates until a decision on a row can be made.

Table 1. Definition of Query and Input Relation Parameters Used in the Analytical Model

S	Selectivity
p	Predicate probability
k	Number of predicates
N	Total number of Tuples
W_e	Work done to evaluate predicates
W_{wr}	Work done to write back the qualifying rows

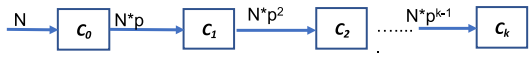


Fig. 5. Number of qualified rows after each predicate for a conjunctive query. N is the total number of rows, p is a predicate probability, $C_0, C_1, C_2, \ldots, C_k$ designate k different predicates.

Clearly, W_e is proportional to the total number of predicates evaluated, and W_{wr} is proportional to the total number of qualifying rows. Hence, the overall throughput of the system can be defined as follows:

$$W_e \propto \text{Total number of evaluated predicates}, \tag{1}$$

$$W_{wr} \propto \text{Total number of qualified rows}, \tag{2}$$

$$\text{Throughput} \propto 1/(W_e + W_{wr}). \tag{3}$$

The rest of this section further elaborates upon W_e and W_{wr} on two types of queries; namely, a query that contains only conjunction of predicates and a query that consists only of disjunction of predicates. Finally, we discuss the case of a mixed query (that contains combination of ANDs and ORs).

Conjunctive Queries. Figure 5 shows the expected number of qualifying rows after each predicate in a conjunctive query.

$$\text{Exp. number of evaluated predicates} = N * \sum_{i=0}^{k-1} p^i, \tag{4}$$

$$\text{From Equation (1)}, W_e \propto N * \sum_{i=0}^{k-1} p^i, \tag{5}$$

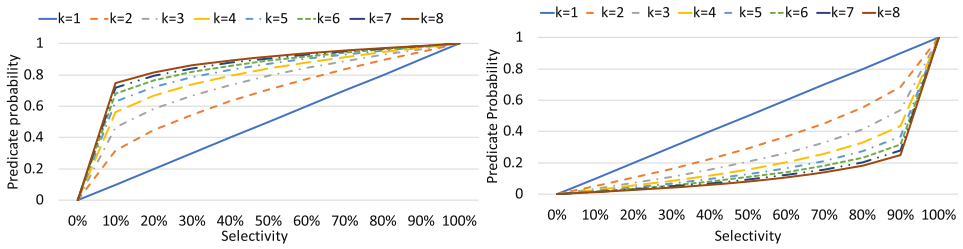
$$\text{Selectivity of a conjunctive query}, S_{and} = p^k, \tag{6}$$

$$\text{Expected number of qualified rows} = N * S_{and}, \tag{7}$$

$$\text{From Equation (2)}, W_{wr} \propto N * S_{and}. \tag{8}$$

Equation (4) calculates the expected number of *evaluated* predicates by summing up the number of rows reaching each predicate. Equation (6) shows that the selectivity, S, is a function of p and k. Equation (7) defines the expected rows satisfying the last predicate condition based upon query selectivity. As a result, we obtain W_e and W_{wr} for conjunctive query in Equations (5) and (8), respectively.

Equation (6) is plotted in Figure 6(a). An important observation is that as S and k increase, p increases, too. Higher values of p imply that more predicates per row are evaluated, i.e., increasing W_e. With increasing value of S, W_{wr} increases, too.



(a) Predicate probability vs selectivity for a conjunctive query

(b) Predicate probability vs selectivity for a disjunctive query

Fig. 6. Variation in predicate probability with respect to selectivity for conjunctive (a) and disjunctive (b) query.

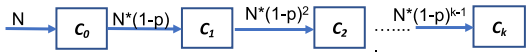


Fig. 7. Number of qualified rows after each predicate for a disjunctive query. N is the total number of rows, p is a predicate probability, $C_0, C_1, C_2, \ldots, C_k$ designate k different predicates.

Table 2. Summary of Relationships Between S, k, W_e, and W_{wr}

	Conjunctive		Disjunctive	
	S Const, $k \uparrow$	k Const, $S \uparrow$	S Const, $k \uparrow$	k Const, $S \uparrow$
W_e	linearly $\uparrow$	$f(p,k) \uparrow$	linearly $\uparrow$	$f(p,k) \downarrow$
W_{wr}	$\propto S$	linearly $\uparrow$	$\propto S$	linearly$\uparrow$

Disjunctive Queries. We perform the same analysis for a disjunctive query, shown in Figure 7, and obtain the following equations:

$$W_e \propto N * \sum_{i=0}^{k-1} (1-p)^i, \tag{9}$$

$$S_{or} = 1 - (1-p)^k, \tag{10}$$

$$W_{wr} \propto N * S_{or}. \tag{11}$$

Once again, we plot p with varying values of S and k in Figure 6(b). Key observations from this graph are: (1) For a given k, p increases with S. Higher probabilities cause disjunctive queries to terminate early; hence, W_e decreases but W_{wr} increases. (2) Keeping S constant, if we increase k, then probability p decreases, increasing W_e. Also, note the opposing nature of probabilities for conjunctive and disjunctive queries in Figure 6. It visually illustrates Equations (4) and (9), showing that for any given selectivity, S, the number of predicates evaluated for both types of queries are complementary to each other. We observe the same trend in our experiments. Table 2 summarizes the different trends in W_e and W_{wr} for both conjunctive and disjunctive queries.

Mixed Queries. We can extend the same analysis to loosely bound the throughput of a query with different combinations of *AND-OR*, arranged in the standard DNF form. The throughput of a mixed query is also governed by Equation (3), i.e., W_e and W_{wr}. Any query with k predicates will at least evaluate 1 predicate and maximum k predicates. For a *pure* conjunctive query, the former case happens at $p = 0$ and the latter at $p = 1$, vice versa for *pure* disjunctive queries. This behavior directly translates into performance achieved by each of these queries. A pure conjunctive query

Table 3. System Configuration of Different Architectures Used for Evaluation

Device	Make & Model	Clock (MHz)	Cores	Memory (GB)	Memory Bandwidth (GB/s)
CPU	Intel Xeon E5-2650V3	3000	10	768	60
GPU	NVIDIA Titan X	1500	3584	12	480
FPGA	Virtex6-760	150	N/A	64	76.8

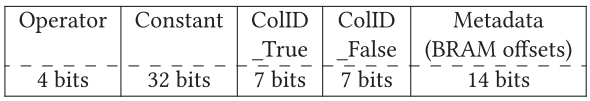


Fig. 8. 64-bit format of Predicate Control Block (PCB).

achieves maximum throughput at $p = 0$ and minimum at $p = 1$ (vice versa for disjunctive queries). Since a mixed query is a disjunction of pure conjunctive queries, W_e for a mixed query is always bounded by Equations (5) and (9). Similarly, W_{wr} for a mixed query is directly affected by its selectivity. Thus, we can infer that the total throughput achieved by a mixed query is bounded by the maximum and minimum performance of *pure* conjunctive and disjunctive queries.

4 EXPERIMENTAL EVALUATION

We proceed with the details of the MTP implementation on the Convey HC-2ex as well as discuss the CPU and GPU implementations. Table 3 summarizes the characteristics of all the platforms used. We then present the experimental validation of different factors affecting the MTP throughput, followed by a thorough comparison of the MTP selection engine with CPU and GPU implementations.

4.1 MTP Implementation

The Convey HC-2ex machine supports 16 memory channels that can concurrently read or write data from the DRAM. With this configuration, the selection engine uses each memory channel: (1) to read PCB (2) to request *new* rows and (3) to request *recycled* rows, and (4) to write back qualifying rowIDs. One engine per channel allows us to place a total of 16 selection engines on a single AE. Each engine holds the same select query and applies it to different sets of rows. With more engines, more rows are evaluated in parallel. We also replicate the selection engine across four AE, thus leveraging inter-engine parallelism.

To execute queries at runtime, queries are parsed into a PCB. To generate a PCB, we wrote a simple C++ program that converts an SQL query to the corresponding PCB, as discussed in Section 3.

Figure 8 shows the bit allocation for a query. We choose a 64-bit wide PCB to match the Convey HC-2ex data bus. A 64-bit PCB enables six basic comparison operators (<=, <, =, ! =, >, >=), 4-byte constants, and 128 column indexes. These values are sufficiently large to test the functionality and performance of our design. The PCB can be made wider if we need to support more operators or larger constants. A wider PCB will require more memory requests to get the complete information about the query and column data.

Table 3 describes our other evaluation platforms. Note that the Convey HC-2ex CPU host does not support AVX2 registers. Therefore, to compare the throughput of MTP and CPU in-memory selection implementation, we use a newer Intel Xeon E5-2650 v3 CPU. In this work, we do not assume the availability of any *a priori* information about the query selectivity. However, to identify the average selectivity of standard workloads, we profiled queries from the TPC-H benchmark

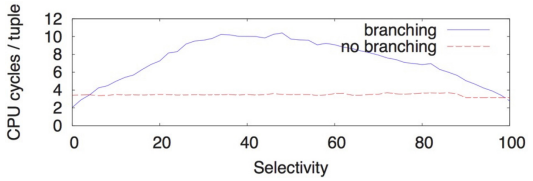


Fig. 9. Branching and No-Branching algorithm cost vs selectivity (taken from Reference [63]).

(more details in Section 4.7). In this profiling, the average query selectivity was observed to be less than 30%. Finally, we assume that the input relation is already transferred to the DRAM closer to the FPGA region. We hold the same assumption for the CPU and the GPU. We do not consider any transfer time for all the devices, which is usually the case for in-memory data analytics [5, 35, 70].

4.2 CPU and GPU Implementations

We implement a *scalar* and an *SIMD vectorized* version of the no-branch algorithm (Listing 3) on the CPU. In related literature [18, 61–63] both branch (Listing 1, 2) and no-branch (Listing 3) variants have been benchmarked with the latter exhibiting on average higher throughput. The former technique is often sensitive to branch mispredictions, only marginally outperforming the no-branch algorithm in extreme cases (i.e., extremely low or high selectivity), as shown in Figure 9 (taken from Reference [63]). This property holds true for both scalar and vectorized implementations [18, 62], guiding our choice to implement only the no-branch variant.

In both the scalar and vectorized implementations, we leverage multithreading by partitioning the input relation across different threads, pinned to physical CPU cores to avoid cache trashing.

For each approach, we use the tuple storage format that allows it to extract the greatest benefits; namely, row-major format for scalar implementation and columnar data layout for the vectorized version. Predicate constants are statically compiled, and predicate loops are manually unrolled to avoid additional cache misses and allow additional compiler optimizations. Code is compiled with GCC version 4.8.5 with the `-O3` optimization flag. To materialize the resultant recordIDs, the permutation table technique presented in Reference [61] is used.

Our GPU implementation relies on the findings of References [36, 70]. We assign each GPU thread to a distinct tuple for processing and evaluate the conditions of a query using a for-loop. Each evaluation is aggregated to a local register using bitwise-AND operands. Also, as suggested in Reference [70], we use optimal GPU plans to mitigate the effects of thread divergence on GPU. The final evaluation results are written back to global memory into a flag vector. To identify the qualifying tuple-ids, we utilize a stream compaction kernel available from NVIDIA's CUB [55] library. This kernel applies the given selection criterion, as indicated by the flag vector, to construct the corresponding output sequence from the selected items in the tuple-id input sequence. Gathering the qualifying tuples using the aforementioned method was also proposed in Reference [36].

4.3 Datasets and Queries

To study in detail how the number of predicates and the total selectivity affects runtime on different platforms, we used synthetically generated data. We experimented with different dataset sizes by varying the number of tuples from 2^{20} to 2^{27}. Each tuple consists of 8 fixed-size 64-bit columns, which is generally used for performance evaluation of in-memory query processing algorithms [13, 16, 17, 79]. However, none of the design choices prevent the use of wider tuples. The tuple width is solely limited by the machine architecture. We have considered both row-major and column-major storage formats. In the former case, the tuple's data is aligned contiguously,

occupying exactly one cache line. In the latter scenario, values of a particular column for all tuples are stored adjacent to each other. Values of different columns are drawn from a uniform distribution and are not correlated between individual tuples. This allows us to use Equation (6) to calculate the total query selectivity from the probability of each predicate.

Query selectivity was varied from 0% to 100% (0%: no row qualifies, 100% all rows qualify) with 10% increments, while the number of predicates in queries was independently changed from 1 to 8. Overall, we experimented with a total of 160 such queries for both conjunctive and disjunctive selection conditions and experimentally validated the complementary performance achieved for both types of queries, as discussed in Section 3.3. Unless and until specified, all experiments are conducted on the dataset size of 2^{27} tuples (~1GB) on different parameters for conjunctive queries.

To the best of our knowledge, there is no separate benchmark concentrating on evaluating the selection operator. Standard analytical database benchmarks (like TPC-H) evaluate complex SQL queries, consisting of multiple operators such as projections, joins, aggregations, and so on. Among the TPC-H queries, only Q_6 involves single table selections; Section 4.7 presents our experiments for this query using data created by the TPC-H benchmark.

We note that our experiments concentrate only on evaluating single-column numeric predicates; however, MTP can be extended to compare string or variable length columns in several ways. To achieve this goal, we can pad their contents with zeros to the nearest 8-byte chunk and treat all chunks as k predicates of a conjunctive query where each predicate performs numeric comparison. This solution still provides the benefit of early termination; however, we also waste some space on paddings. Alternatively, we can store the reference to the actual value of such a column. In this case, accessing the value will require an additional memory access. Handling complex string matching is another use-case with the similar requirements, so we leave this problem for a future work.

4.4 Throughput Validation

In this section, we experimentally verify the effect of selectivity and number of predicates on the MTP throughput, as discussed in Section 3.3. We present results for a conjunctive query with selectivity S varying from 0% to 100% and number of predicates k varying from 1 to 8.

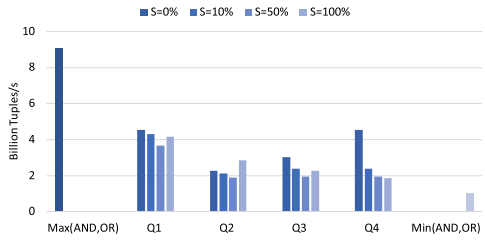
Peak performance is dependent on the total number of concurrent engines and the clock frequency. The Convey HC-2ex machine has 4 FPGAs and each FPGA interfaces to the DRAM via 16 individual duplex memory channels, allowing us to place a total of 64 selection engines. Since the FPGAs run at 150MHz and assuming no stalls, the design could evaluate an aggregate of P = 9.6 billion predicates/s ($150 * 10^6 \frac{cycle}{s} * 64 \frac{predicates}{cycle}$). A row can evaluate anywhere between 1 to k predicates, putting an upper bound of P tuples/s and the lower bound of P/k tuples/s on the overall design. For $k = 8$, the maximum and the minimum theoretical throughput values are 9.6 billion tuples/s and 1.2 billion tuples/s, respectively.

Figure 11(a) presents the achieved throughput, and Figure 11(b) presents the total number of predicates processed for a conjunctive query while varying the selectivity (S) and the number of predicates (k). The trends are in accordance with Equation (4). The observed throughput (Figure 11(a)) decreases with increasing values of S and k in accordance with Equation (3). At $S = 0\%$, only one predicate is evaluated, resulting in a high throughput of 9.06 billion tuples/s. At $S = 100\%$ and $k = 8$, all 8 predicates are evaluated, resulting in a lower throughput of 1.04 billion tuples/s. The prior condition represents the best case and the latter represents the worst case of the predicate evaluation. Hence, the throughput achieved in both cases is close to the upper and the lower bound. The throughput achieved by other combination of S and k lies between these two cases.

The impact of W_{wr} on the throughput can be clearly seen with higher values of S. For instance, even though the *evaluated predicates* remain the same for a query with $k = 1$ (Figure 11(b)), the throughput still drops with increasing value of S.

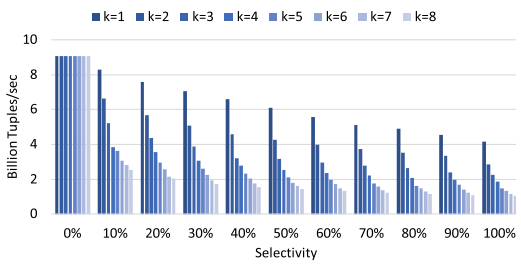
Q1	(C_1) v $(C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 \wedge C_7 \wedge C_8)$
Q2	$(C_1 \wedge C_2)$ v $(C_3 \wedge C_4)$ v $(C_5 \wedge C_6)$ v $(C_7 \wedge C_8)$
Q3	$(C_1 \wedge C_2 \wedge C_3)$ v $(C_4 \wedge C_5)$ v $(C_6 \wedge C_7 \wedge C_8)$
Q4	$(C_1 \wedge C_2 \wedge C_3 \wedge C_4)$ v $(C_5 \wedge C_6 \wedge C_7 \wedge C_8)$

(a)

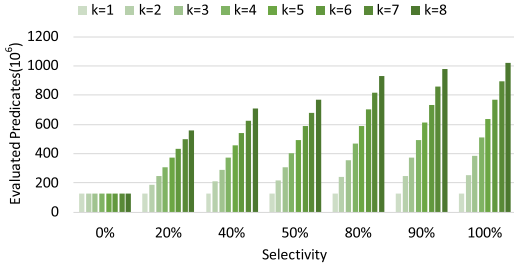


(b)

Fig. 10. (a) The mixed queries used for evaluation; (b) Their performance evaluation with varying selectivity.



(a) Measured throughput for varied selectivities and number of predicates



(b) Total number of predicates evaluated for varied selectivities and number of predicates

Fig. 11. Experimental validation of (a) Equation (4) and (b) Equation (3).

Next, we evaluate the performance of mixed (AND-OR) queries described in Figure 10(a). Notice that the number of predicates ($k = 8$) is the same, but the attributes are grouped differently for each query. As discussed in Section 3.3, the performance achieved by a mixed query will always be bounded by the performance of a pure conjunctive and disjunctive queries. As depicted in Figure 10(b), the performance of Q1, Q2, Q3, and Q4 lies between the **Max(AND, OR)** and **Min(AND, OR)** queries. Max(AND, OR) presents the maximum performance achieved by a pure conjunctive and disjunctive query with $k = 8$ predicates. Similarly, Min(AND, OR) presents the minimum performance achieved by both of these queries. Empirically, we also observed that for any given value of selectivity and number of predicates, the performance of a query is governed by the work done to evaluate the total number of predicates, i.e., W_e. For instance, for $S = 0$, the number of predicates evaluated by queries Q1, Q2, Q3, and Q4 are 268,435,456, 536,870,912, 402,653,184, and 268,435,456, respectively. Q2 processes more data, hence achieves lower throughput.

4.5 Throughput Evaluation

Figure 12 compares the absolute query runtime on the CPU, GPU, and MTP implementations. The GPU delivers the best raw performance across different predicate values and selectivities, followed by MTP that performs better on low selectivity values and smaller values of predicates. The CPU

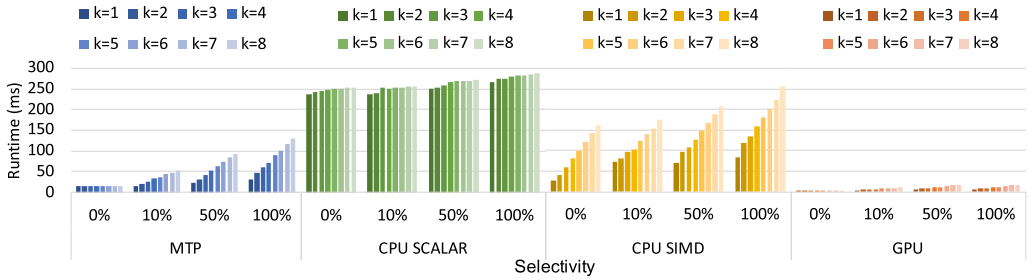


Fig. 12. Runtime (in ms) achieved by MTP, CPU, and GPU implementation.

SIMD implementation comes next, and finally the CPU Scalar implementation achieves the highest runtime among all other architectures.

In our experiments, we define S using p and k assuming that all predicates have the same probability. On the contrary, real-world queries might have a different combination of predicate probabilities for the same total selectivity. If a query optimizer performs a good job of arranging predicates in the order of their likelihood of being false (true) for conjunctive (disjunctive) queries, then MTP will work independently of the number of predicates in a query and is only limited by the number of memory accesses required for query evaluation. This can be seen in Figure 12, where the MTP runtime does not change with the number of predicates for 0% selectivity.

The performance of MTP is sensitive to predicate probability. The predicate probability of a conjunctive query increases with the selectivity (S) as well as with the number of predicates (k), as can be seen in Figure 6(a). As a consequence, the query evaluation can terminate early on the lower value of selectivities but builds up for higher values. Additionally, for high values of S, the writeback work (W_{wr}) increases, too, resulting in a quick drop in throughput.

Unlike the MTP implementation that is based on early termination, all other platforms implement a variant of the *No-Branch* algorithm. The CPU Scalar graphs clearly show that its execution time is independent both from the number of predicates and from the query selectivity. This independence is an expected behavior, because in a row-major storage format, selection is a memory-bounded computation. Each access to a particular tuple will bring from memory the values for *all* its columns, whether they will be evaluated later or not.

However, the runtime of the CPU SIMD implementation grows linearly as we increase the number of predicates in a query from 1 to 8. Again, this behavior is explained by the fact that, in a columnar storage format, we are fetching only the values that will be later used for evaluating the predicate. For the queries with 8 predicates, the runtimes of Scalar and SIMD converge, because they perform the same amount of memory accesses. However, we can also see another trend for the vectorized implementation: its runtime grows as we move from 0% selectivity to 100%. This is explained by the increasing amount of qualifying recordIDs that need to be written to the output buffer. The SIMD implementation is susceptible to growing W_{wr}, because it requires additional permutations to retrieve IDs of the rows that were qualified from an SIMD lane.

We should also note that for the CPU implementations, the runtime is a function of the main memory bandwidth utilization, not the penalty of fetching data into CPU cache. In both experiments, the data access patterns (contiguous load for Scalar or load with constant strides for SIMD) are easily recognized by the CPU prefetcher, which was verified by preliminary experiments where we had disabled the prefetcher.

Similarly, the GPU implementation evaluates all predicates despite their different selectivities, resulting in more evaluation work for the respective query. This translates to a higher number of

Table 4. Number of Reads and Writes for Selection Algorithm Across MTP, CPU, and the GPU

	Read Words	Write Words
MTP	$N * \sum_{i=0}^{k-1} p^i$	$N * S$
CPU Scalar	$N *$ ⌈Row Size/Cache line size⌉	$N * S$
CPU SIMD	$k *$ ⌈N/Cache line size⌉	$N * S$
GPU	$N + S * N * (k-1)$ $(N + N * flag\_vector_{RowIDs})$	$N * flag\_vector_{RowIDs}$ $N * S$

Note that Nvidia Cub library accepts a vector of 1byte flags.

memory fetches that quickly dominate the total execution time, as their cost is several magnitudes higher than that of evaluating the predicate conditions. Therefore, an increase in the number of predicates corresponds to increasing runtime, as indicated by our experimental results.

4.6 Throughput Efficiency

To better capture the memory-bounded nature of the selection and provide a direct comparison between widely different architectures, we measure bandwidth utilization as well as predicates evaluated/s on our evaluation platforms. Better bandwidth utilization directly translates into higher predicate evaluation, which also indicates efficient core utilization. As discussed in Section 4.1, the Convey HC-2ex has 4 FPGAs with cumulative bandwidth of 76.8GB/s, the CPU system has a memory bandwidth of 60GB/s, and the GPU system has a memory bandwidth of 480GB/s.

In this experiment, both read and write operations contribute towards total memory access. Table 4 summarizes these operations on each of the evaluation platforms.

Both CPU and GPU implementations require *gather* operation to materialize the resultant rowIDs. While on CPU, gathering the result from a SIMD register is a pure vector instruction and does not incur additional memory read/write latency; GPU implementation write intermediate query results back to the global memory.

Figure 13 shows the bandwidth utilization across different evaluation platforms. There are two parts of this experiment: (1) minimum amount of work required to achieve maximum bandwidth utilization; (2) absolute peak bandwidth utilization achieved. To evaluate part (1), we varied dataset sizes (N in Table 4) from 2^{20} to 2^{27}. For brevity, we chose to show the results for 2^{20}, 2^{23}, and 2^{27}. The MTP bandwidth utilization is affected by the total number of predicates evaluated (W_e). It can be seen in Figure 13(a) that for $N = 2^{20}$ and $S = 0\%$, MTP evaluates ~1M predicates, achieving the utilization of 69%. On the same dataset size, for $S = 100\%$, a total of ~8M predicates are evaluated, and the bandwidth utilization quickly reaches 85%. MTP bandwidth saturates at ~8M predicate evaluations. Beyond this limit, bandwidth utilization minimally increases or remains constant, irrespective of N. Similarly, CPU-Scalar and CPU-SIMD implementations saturate bandwidth at $N = 2^{23}$ and $N = 2^{26}$, respectively. The GPU implementation achieves its peak at $N = 2^{27}$. This limit was verified by running an additional experiment with $N = 2^{28}$. On CPU and GPU, smaller dataset size does not provide enough work for their cores, limiting the bandwidth utilization.

To address part (2), we compare the bandwidth utilization achieved by each of the platforms on the dataset size of 2^{27} in Figure 13. MTP achieves fairly constant bandwidth utilization across different query parameters (S and k). This behavior is expected, because all memory accesses (reads or writes) on the MTP are interleaved on a single memory channel. More channel activity leads to a higher bandwidth utilization. On average, MTP achieves 88.6% bandwidth utilization across varying values of S and k.

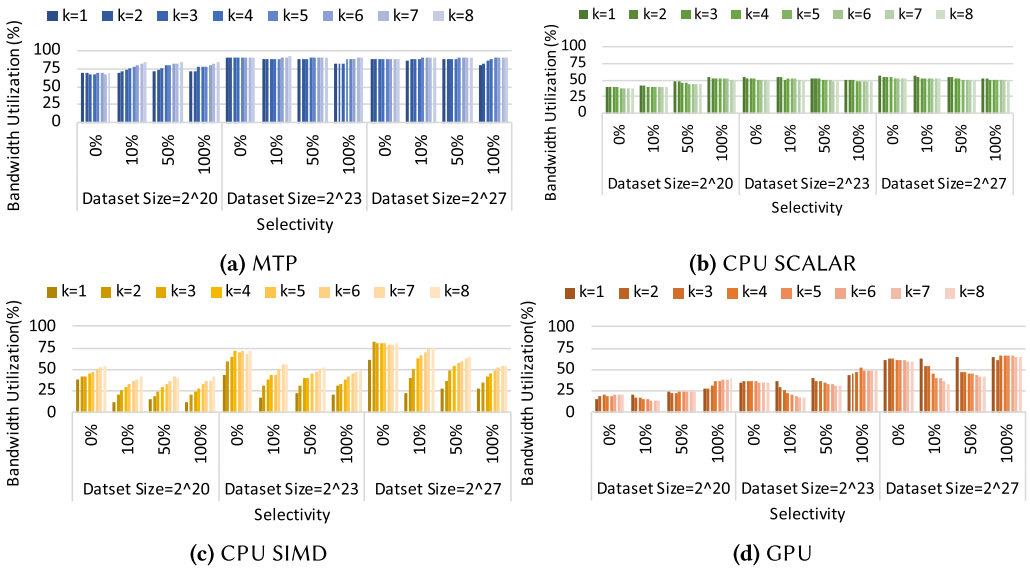


Fig. 13. Bandwidth Utilization of (a) MTP (b) CPU-Scalar (c) CPU SIMD (d) GPU implementations by varying dataset size and the selectivity.

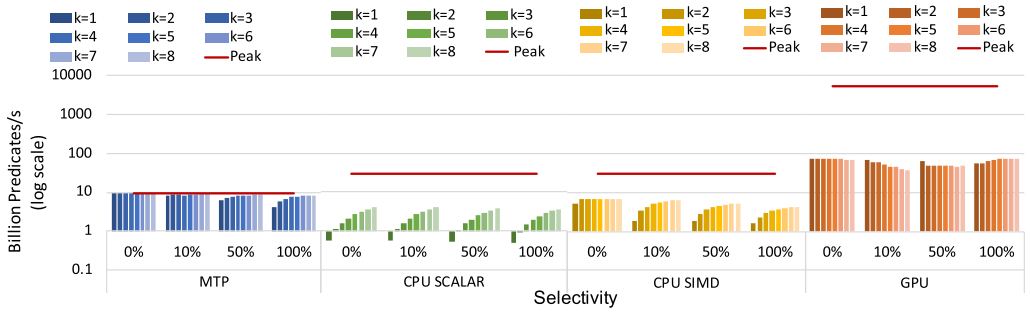
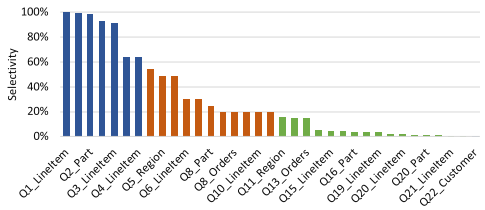


Fig. 14. Predicates/s compared to the respective peak performance.

The CPU SIMD implementation is remarkably efficient for $S = 0\%$. Since only *one* predicate is evaluated, the columnar data layout makes the cache access extremely effective in this case. Moreover, with 0% selectivity there is no result materialization overhead. On $S = 0\%$, CPU SIMD implementation achieves up to 81% of the bandwidth utilization. However, the bandwidth utilization quickly drops to 30%–54% as k and S are increased. Higher value of k and S directly increases W_e and W_{wr}, respectively. However, CPU Scalar implementation achieves on average 52.2% bandwidth utilization. It fairly remains constant across different query parameters.

The bandwidth utilization of the GPU implementation drops almost linearly up to the selectivity of 40%. However, in the selectivity range of 50%–100%, the bandwidth utilization increases again and reaches up to 66%.

Figure 14 compares predicates evaluated/s on each platform to its respective theoretical peak. MTP runs 64 parallel selection engines running at 150MHz, achieving a theoretical peak of 9.6 BPredicates/s. On $S = 0\%$, the observed throughput is 9.03 BPredicates/s. However, for smaller values of k ($k \leq 3$), as the value of S increases, total predicates/s evaluated drop down. For instance,



(a)

```
SELECT count(*)
FROM lineitem
WHERE l_shipdate >= date `1995-01-01'
and l_shipdate < date `1996-01-01'
and l_discount between 0.04 and 0.06
and l_quantity < 24;
```

(b)

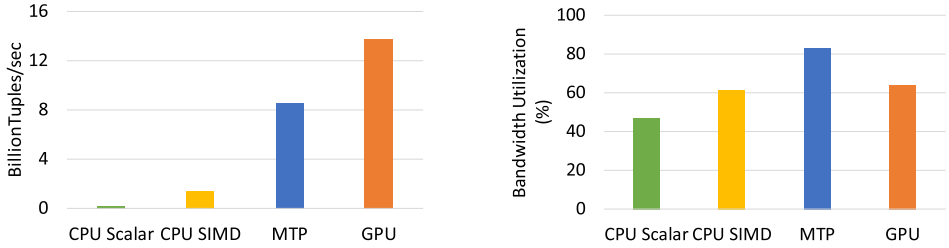
Fig. 15. (a) describes the selectivity of TPC-H queries. Each color marks the range of selectivity: (blue) above 60%, (orange) 50%–20%, (green) below 10%; (b) presents TPC-H Query6.

at $S = 100\%$ only 4.1–6.7 BPredicates/s are achieved for $1 \leq k \leq 3$. This behavior is a consequence of threads terminating early for smaller values of k, unable to completely mask memory latency as well as high selectivity. Similarly, CPU and GPU run 10 cores and 3,584 cores in parallel, running at 3GHz and 1.5GHz, respectively, achieving a peak of 30 BPredicates/s and 5.3 TeraPredicates/s, respectively. As shown in Figure 14, both CPU and GPU evaluate significant lower percentage of predicates/s with respect to their corresponding peaks.

4.7 TPC-H Query Evaluation

To evaluate the performance of our implementations on a standard workload, we considered the well-known TPC-H benchmark [75]. We have profiled all 22 TPC-H queries to understand the various characteristics (selectivity, number of predicates, predicate types) of the selection operator in this benchmark. Most queries in the TPC-H workload involve complex joins and group-by aggregations, so not all predicates in the WHERE clause might be used as filtering conditions in selection operators. Instead, we have considered optimized plans where selections are pushed down and executed before the joins and right after table scan operators. Figure 15(a) presents the selectivity(%) of different TPC-H queries. In this experiment, the average number of predicates in the selection was 2, with an exception of queries Q_6 and Q_{19}, which have 5 and 8/12 (Part/Lineitem tables) predicates, respectively. All of the selections in the benchmark were conjunctions, again excluding Q_{19}, which has a mix of conjunctions and disjunctions.

To capture the real effect of the MTP design in the selection operation, we would like to isolate the effect of all relational operators in the query. This makes Q_6, shown on Figure 15(b), an ideal candidate for our evaluation. We run the query Q_6 on the TPC-H Lineitem table with the scale factor of 10. The measured selectivity of this query is 1.91%. Figure 16(a) presents the raw performance of query Q_6 executed by the various architectures. We observe the same throughput trend as discussed in Section 4.5. Due to the high memory bandwidth, GPU achieves the highest raw performance, followed by the MTP design, the CPU SIMD, and the CPU Scalar implementation. However, when we compare the bandwidth utilization in Figure 16(b), the MTP achieves 83% while CPU Scalar, GPU, and CPU SIMD implementation achieves 47%, 61%, and 64%, respectively.



(a) Raw performance comparison of CPU, MTP and GPU implementations **(b)** Bandwidth utilization of CPU, MTP and GPU implementation

Fig. 16. TPC-H query Q6 performance evaluation.

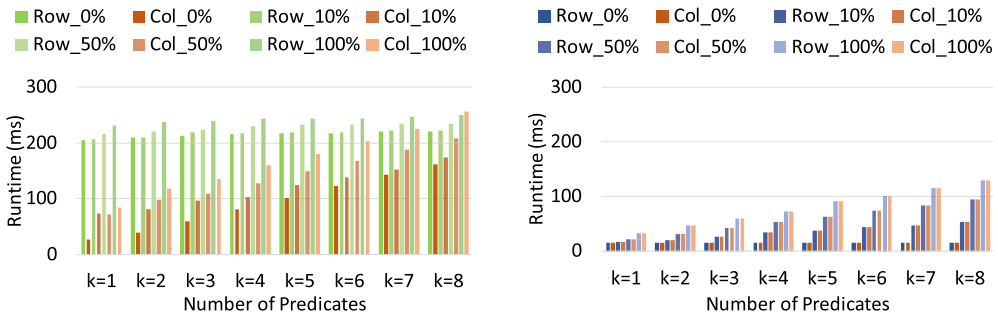
Table 5. Performance Evaluation of TPC-H query Q6 on CPU, GPU, and MTP Implementations

	CPU Scalar	CPU SIMD	GPU	MTP
Memory Fetches (%)	100	18.75	18.75	7.25
Average Evaluations (per Row)	3	3	3	1.27
Effective Bandwidth speedup	0.47	3.44	3.53	13.7

Furthermore, to confirm the advantage of our MTP design, we also measured the total number of predicate evaluations and memory fetches. The CPU and GPU implementations access the common columns ($l_{shipdate}$, $l_{discount}$) only once. However, MTP treats them as two independent attributes and fetches the same column again only if required for further evaluation. The CPU Scalar implementation accesses all 16 columns per row of the table, therefore it is considered as a 100% memory access. The CPU SIMD and the GPU implementations need only 3 out of 16 columns to evaluate the query, contributing to 18.75% of memory accesses. We used counters to keep track of the number of memory fetches and the number of evaluated predicates for the MTP implementation. Memory fetches with MTP amount to 7.25% of total memory accesses. As suggested in Reference [72], we compute the effective bandwidth speed-up by taking the ratio of the total size of the Lineitem relation processed per unit time and the peak bandwidth, summarized in Table 5.

4.8 Data Layout Independence

Both CPUs and GPUs are optimized for cache line access. We believe this is the reason why their performance varies so much for different storage formats (row and column major). Architecturally, GPUs operate similarly to SIMD on CPUs, with the difference being the GPU grouping of execution threads into warps. This grouping is not only relevant to computation but also to global memory accesses. In fact, the related literature has unanimously promoted the use of column major [11, 24, 31, 70] data layouts to maximize the effective memory bandwidth. Leveraging the aforementioned data layout, database researchers focused on defining a set of fairly standard parallel primitives [36] having a direct correlation to relational operators. Therefore, we do not consider GPU for this experiment. MTP design, along with the Convey HC-2ex memory subsystem, achieves performance that is free from utilizing any form of data alignment in memory and only depends upon accessing individual columns of a tuple for query evaluation.



(a) CPU throughput measured for row vs column data layouts

(b) MTP throughput for the row vs column layouts

Fig. 17. Performance comparison of the CPU and the MTP implementations with row-major and columnar data layouts.

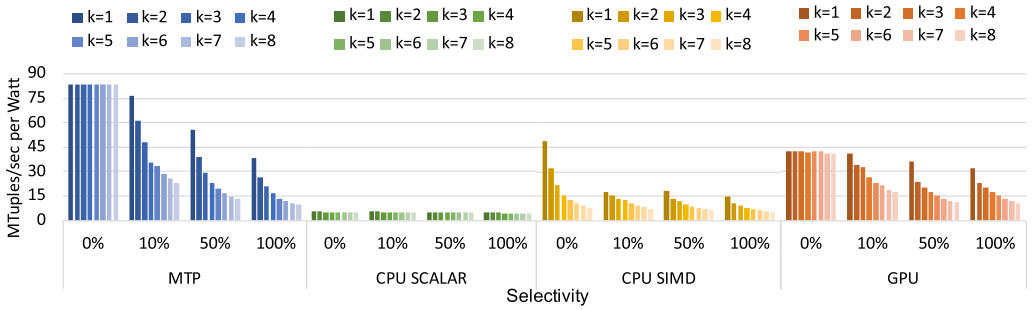


Fig. 18. Comparison of power efficiency on MTP, CPU, and GPU systems.

Figure 17(a) compares the performance achieved by the CPU implementations; namely, Scalar on row and SIMD on column store. The columnar data layout leads to efficient cache access when only few predicates are required for evaluation. This directly translates into high throughput that is 8× over the performance achieved by a row-store data layout. However, as the number of predicates increases, the amount of data accessed by both the row-store and the column-store implementations converge and so does their performance. Figure 17(b) shows the MTP performance on row- and column-store data layouts. For a given number of predicates and selectivity, the number of memory accesses does not depend upon the type of data layout and, therefore, the MTP performance remains unaffected. These observations on the MTP are coherent with the results presented in Reference [67].

4.9 Power Efficiency

To further justify our MTP design, Figure 18 presents the power efficiency results measured in Million Tuples/s per Watt. We compared the power efficiency of CPU, GPU, and MTP implementations.

The measured on-chip power consumption on each FPGA of HC-2ex machine is 21W and total power supply is 25W, which gives us the total FPGA power consumptions as 109W (25 + 21*4). On CPU, we use the manufacturer TDP (thermal design power) rating of 105W, as prescribed in Reference [59]. On GPU, the average power consumption was measured to be 155W for the design with a power supply of 600W [2]. We compare the power efficiency of all the devices in Figure 18.

Table 6. FPGA Area Utilization for Selection Engine

# Engines	Registers	LUTs	BRAMs
1	129,251	78,746	175
	13.6%	16.6%	8.2%
4	149,151	88,801	250
	15.4%	18.76%	11.6%
8	172,171	108,451	295
	18.15%	22.82%	14.2%
16	201,241	120,321	358
	21.2%	25.22%	16.5%

The power efficiency is coherent with our throughput evaluation. Since the MTP throughput drops with the selectivity and number of predicates, it directly affects the power efficiency, too. It is 1.7×–10.6× better than CPU-SIMD on 0% selectivity. However, for $S = 100\%$, the power efficiency drops to 2.5×–1.97×. Similarly, in comparison to GPU, we get 1.9× power savings at $S = 0$ but for $S = 100\%$, the efficiency is on par with GPU.

We use the same power statistics to evaluate the power efficiency of different architectures on query Q_6 of TPC-H benchmark. MTP design is 14× and 4.2× more power efficient than GPU and CPU-SIMD implementations on this query.

4.10 FPGA Area Utilization

Table 6 shows the area utilization. Many resources are shared between the engines as their number increases. For example, one selection engine uses 13.6% of the available registers, whereas 16 engines use only 21.2%. Also note that with increasing number of engines there is very minimal increase in the number of logic resources (LUTs). Overall, the space utilization on the FPGA is low, leaving sufficient space to extend our design with various optimizations or operators (projections, aggregation, join). This also gives us insight into how well the design could scale on another platform with more, or less, memory channels. These results include Convey's memory interface wrapper, which does not occupy a significant portion of the area. We see that even with 16 engines the Virtex-6 still has plenty of room for more engines; only 25.22% of the logic is utilized.

5 RELATED WORK

All modern, medium to high-end CPUs are equipped with SIMD registers that can be used to accelerate many database operations [78, 84], including selection. The data layout (row- or column-oriented) of an in-memory database table also plays a significant role in achieving better response times, as it directly impacts the bandwidth and cache utilization. Despite many efficient algorithms proposed for both row [25, 62] and column [25, 60, 86] storage layouts, columnar format provide better opportunities for data compression [30, 52, 78]. However, all these approaches suffer from the overhead of converting data in row-major format to a columnar layout either in the background or by keeping two separate representations of the same record. Different variants of the Listings described in Section 2.1 are explored in References [44, 52, 61].

Recently, hardware accelerators, such as FPGAs and GPUs, have also been considered a viable alternative to modern CPUs for accelerating common relational operators [36, 57], including selections [32, 72]. FPGAs are either integrated into a datapath between a network and processor to perform database operations [1, 65, 79], viewed as a co-processor or accelerator [20], or

domain-specific processors [21, 53, 57, 69]. Similarly, GPUs have also demonstrated significant speed-up in data processing [12, 24, 70] by proposing kernel scheduling, algorithms, and data structures to overcome architectural challenges like avoiding thread divergence and offering coalesced memory access pattern.

Embedding reconfigurable hardware in storage devices is a growing area of interest [6, 33, 43, 65]. For instance, Ibex [79] is a MySQL accelerator platform where a SATA SSD is coupled with an FPGA. The work in Reference [39] explores offloading part of the computation in database engines directly to the storage with FPGA-based accelerators while targeting row-oriented databases and implements a no-branch selection algorithm. JAFAR [82] is a near-data processing hardware accelerator embedded into DRAM modules that implements the select operator of a modern column-store. Commercial platforms such as Kickfire [46] and Baidu [9] offer FPGA solutions for database management systems. However, because of their proprietary nature, specific implementation details and measurements are difficult to obtain.

Domain-specific processors are proposed in References [1, 27, 34, 57], including Q100 [81] and LINQits [21]. Q100 is a data-flow style processor and uses ASIC functional units to exploit operator pipelines without supporting data management in off-chip storage. Although ASICs can provide higher efficiency, their fabrication process is time-consuming. By contrast, FPGAs allow relatively easy development cycle. Another emerging trend is the use of dark silicon [26, 49]. Widx [49], in particular, is one such on-chip accelerator for hash index lookups in main memory databases.

Hybrid CPU-FPGA cache-based architectures have been proposed to mitigate long memory latencies [21, 69, 80]. For instance, LINQits [21] maps LINQ, a query language, to a set of FPGA-based hardware templates on a heterogeneous SoC (FPGA + ARM). This accelerator focuses on the range-partitioning step of query processing to insure that each partition can fit on an on-chip cache. Similarly, Reference [69] exploited an Intel Xeon + FPGA prototype system to speed up complex pattern-matching SQL constructs supported by MonetDB.

In this work, rather than *mitigate* the memory latency by relying on large cache hierarchies, the MTP architecture *masks* it by switching to the next ready thread upon a memory operation in a cache-less architecture. All the database accelerators discussed in this section assume a continuous stream of data onto the FPGA for processing. With the MTP, non-streamable applications can also achieve higher performance on FPGAs. As a consequence of relying on streaming paradigm, other accelerators also assume a particular data layout, either row [22, 57, 72, 79] or column [20] format for query processing. Instead, the MTP selection engine can handle either format without compromising the performance. In the future, we want to explore the aforementioned near-data processing ideas for upcoming memory technologies without being limited by random access performance. In fact, the distributed storage system presented in Reference [39] is a good alternative option to the Convey HC-2ex SG-DIMMs.

6 CONCLUSIONS

We presented a lightweight hardware multithreaded implementation of the *selection* operation for in-memory relational databases, the MTP, prototyped on an FPGA platform. The MTP design facilitates fine-grained data access, thus is completely oblivious to a particular data layout (row or column) and avoids fetching irrelevant data. Furthermore, MTP does not rely on hardwired filters for the selection. Instead, it can execute new queries at runtime. We thoroughly evaluate it against the best implementations on CPU and GPU. Experimental results show that the MTP throughput varies only with the predicate probability and is bounded by the memory channels. The GPU achieves the best raw performance over the entire parameter space. Due to the memory-bounded nature of the selection operation, we attribute this performance to the high GPU memory bandwidth. However, while comparing the bandwidth utilization across different platforms, we

observe MTP achieves 88% of the bandwidth utilization while GPU implementation peaks at 66%. We also evaluate MTP on the TPC-H query *Q6*. On this benchmark query, MTP achieves 83% of the bandwidth utilization while saving 93% of total evaluations. Additionally, the MTP engine occupies only 25.2% of chip area, leaving plenty of room for future improvements. Furthermore, we are examining how partitioning and thread load balancing can be utilized on the MTP-based implementations to deal with extremely skewed datasets.

REFERENCES

[1] NETEZZA. 2014. http://www.ibm.com/software/data/netezza/.

[2] Nvidia. 2016. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

[3] Peloton. 2016. http://pelotondb.io/.

[4] Tor M. Aamodt, Paul Chow, Per Hammarlund, Hong Wang, and John P. Shen. 2004. Hardware support for prescient instruction prefetch. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE Computer Society, Washington, DC, 84. DOI: https://doi.org/10.1109/HPCA.2004.10028

[5] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J. Halstead, Walid A. Najjar, and Vassilis J. Tsotras. 2016. FPGA-accelerated group-by aggregation using synchronizing caches. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'16)*. ACM, Article 11, 9 pages. DOI: https://doi.org/10.1145/2933349.2933360

[6] Sandeep R. Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. 2017. A many-core architecture for in-memory data processing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. New York, NY, 245–258. DOI: https://doi.org/10.1145/3123939.3123985

[7] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. 1992. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the International Conference on Supercomputing*.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The tera computer system. In *Proceedings of the International Conference on Supercomputing*. 1–6.

[9] Baidu. 2016. https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/.

[10] David H. Bailey. 1997. *Little's Law and High Performance Computing*. Technical Report. In RNR Technical Report.

[11] Tukora Balázs and Pollack Mihály. 2008. High performance computing on graphics processing units. *Pollack Periodica* 3, 2 (2008), 27–34.

[12] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*. ACM, New York, NY, 10. DOI: https://doi.org/10.1145/1735688.1735706

[13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endowment* 7, 1 (2013), 85–96.

[14] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE'13)*. 362–373.

[15] Ronald Barber, Guy Lohman, Vijayshankar Raman, Richard Sidle, Sam Lightstone, and Berni Schiefer. 2015. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE'15)*. IEEE, 1246–1252.

[16] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. ACM, 37–48. DOI: https://doi.org/10.1145/1989323.1989328

[17] Peter A. Boncz, Stefan Manegold, Martin L. Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Databases (VLDB'99)*, Vol. 99. 54–65.

[18] David Broneske, Sebastian Breß, and Gunter Saake. 2015. Database scan variants on modern CPUs: A performance study. In *Memory Data Management and Analysis*, Arun Jagatheesan, Justin Levandoski, Thomas Neumann, and Andrew Pavlo (Eds.). Springer International Publishing, Cham, Switzerland, 97–111.

[19] David Broneske, Andreas Meister, and Gunter Saake. 2017. Hardware-sensitive scan operator variants for compiled selection pipelines. In *Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*. 403–412.

[20] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'14)*. 151–160.

[21] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*. ACM Press. DOI: https://doi.org/10.1145/2485922.2485945

[22] C. Dennl, D. Ziener, and J. Teich. 2013. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'13)*. 25–28. DOI: https://doi.org/10.1109/FCCM.2013.38

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 1243–1254.

[24] Gregory Frederick Diamos, Haicheng Wu, Ashwin Lele, and Jin Wang. 2012. *Efficient Relational Algebra Algorithms and Data Structures for GPU*. Technical Report. Georgia Institute of Technology, Atlanta, GA.

[25] Amr El-Helw, Kenneth A. Ross, Bishwaranjan Bhattacharjee, Christian A. Lang, and George A. Mihaila. 2011. Column-oriented query processing for row stores. In *Proceedings of the International Workshop on Data Warehousing and OLAP (DOLAP'11)*. ACM, New York, NY, 67–74. DOI: https://doi.org/10.1145/2064676.2064689

[26] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 365–376. DOI: https://doi.org/10.1145/2000064.2000108

[27] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 2017. UDP: A programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 55–68. DOI: https://doi.org/10.1145/3123939.3123983

[28] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database—An architecture overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[29] John Feehrer, Sumti Jairath, Paul Loewenstein, Ram Sivaramakrishnan, David Smentek, Sebastian Turullols, and Ali Vahidsafa. 2013. The Oracle SPARC T5 16-core processor scales to eight sockets. *IEEE Micro* 33, 2 (March 2013), 48–57.

[30] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the envelope of main memory data processing with a new storage layout. Retrieved from www.comp.polyu.edu.hk.

[31] Pedram Ghodsnia. 2012. An in-GPU-memory column-oriented database for processing analytical workloads. In *Proceedings of the VLDB PhD Workshop. VLDB Endowment*, Vol. 1.

[32] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, 215–226.

[33] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 153–165. DOI: https://doi.org/10.1145/3007787.3001154

[34] Sebastian Haas et al. 2016. An MPSoC for energy-efficient database query processing. In *Proceedings of the Design Automation Conference (DAC'16)*. ACM, Article 112. DOI: https://doi.org/10.1145/2897937.2897986

[35] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. 2015. FPGA-based multithreading for in-memory hash joins. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'15)*.

[36] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V Sander. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Data. Syst.* 34, 4 (2009), 21.

[37] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD* 22, 2 (June 1993), 267–276. DOI: https://doi.org/10.1145/170036.170078

[38] Cray Inc. 2006. Cray XMT. http://www.craysupercomputers.com/downloads/crayxmt/crayxmt_datasheet.pdf.

[39] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1202–1213. DOI: https://doi.org/10.14778/3137628.3137632

[40] Chris Jesshope, Mike Lankamp, and Li Zhang. 2009. Evaluating CMPs and their memory architecture. In *Proceedings of the International Conference on Architecture of Computing Systems*. Springer, 246–257.

[41] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: A scalable storage manager for the multicore era. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 24–35. DOI: https://doi.org/10.1145/1516360.1516365

[42] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 622–634. DOI: https://doi.org/10.14778/1453856.1453925

[43] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for big data analytics. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 1–13. DOI: https://doi.org/10.1145/2749469.2750412

[44] Fisnik Kastrati and Guido Moerkotte. 2016. Optimization of conjunctive predicates for main memory column stores. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1125–1136. DOI: https://doi.org/10.14778/2994509.2994529

[45] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE'11)*. IEEE, 195–206.

[46] Kickfire. 2014. http://www.teradata.com/.

[47] Dongkeun Kim, Steve Shih-wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John P. Shen. 2004. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=977395.977665.

[48] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 252–263. DOI: https://doi.org/10.14778/2856318.2856321

[49] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 468–479. DOI: https://doi.org/10.1145/2540708.2540748

[50] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13. http://dblp.uni-trier.de/db/journals/debu/debu36.html#LahiriNF13.

[51] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM, 743–754. DOI: https://doi.org/10.1145/2588555.2610507

[52] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 289–300. DOI: https://doi.org/10.1145/2463676.2465322

[53] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. 2018. In-RDBMS hardware acceleration of advanced analytics. *Proc. VLDB Endow.* 11, 11 (July 2018), 1317–1331. DOI: https://doi.org/10.14778/3236187.3236188

[54] T. Maruyama. 2017. SPARC64TM XII: Fujitsu's latest 12 core processor for mission critical servers. In *Proceedings of the IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS'17)*. 1–3. DOI: https://doi.org/10.1109/CoolChips.2017.7946375.

[55] Duane Merrill and NVIDIA-Labs. 2015. CUDA UnBound (CUB) Library. Retrieved from https://nvlabs.github.io/cub/.

[56] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2010. Glacier: A query-to-hardware compiler. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, 1159–1162. DOI: https://doi.org/10.1145/1807167.1807307

[57] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires—A query compiler for FPGAs. *PVLDB* 2, 1 (2009), 229–240. http://www.vldb.org/pvldb/2/vldb09-622.pdf.

[58] K. Papadopoulos, K. Stavrou, and P. Trancoso. 2008. HelperCore lt;inf gt;db lt;/inf gt;: Exploiting multicore technology to improve database performance. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 1–11. DOI: https://doi.org/10.1109/IPDPS.2008.4536288

[59] Meikel Poess and Raghunath Othayoth Nambiar. 2008. Energy cost, the key challenge of today's data centers: A power consumption analysis of TPC-C results. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1229–1240. DOI: https://doi.org/10.14778/1454159.1454162

[60] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, 1493–1508. DOI: https://doi.org/10.1145/2723372.2747645

[61] Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized bloom filters for advanced SIMD processors. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'14)*. ACM Press, New York, New York, 1–6. DOI: https://doi.org/10.1145/2619228.2619234

[62] Kenneth A. Ross. 2002. Conjunctive selection conditions in main memory. In *Proceedings of the Symposium on Principles of Database Systems (PODS'02)*. New York, NY, 109–120. DOI: https://doi.org/10.1145/543613.543628

[63] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro adaptivity in vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 1231–1242. DOI: https://doi.org/10.1145/2463676.2465292

[64] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. 2012. Multi-query stream processing on FPGAs. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE'12)*. IEEE Computer Society, Washington, DC, 1229–1232. DOI: https://doi.org/10.1109/ICDE.2012.39

[65] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sonmez. 2017. AxleDB: A novel programmable query processing platform on FPGA. *Microprocessors and Microsystems* 51 (2017), 142–164. DOI: https://doi.org/10.1016/j.micpro.2017.04.018

[66] Simone Secchi, Antonino Tumeo, and Oreste Villa. 2012. A bandwidth-optimized multi-core architecture for irregular applications. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'12)*. IEEE, 580–587.

[67] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2015. Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 267–280. DOI: https://doi.org/10.1145/2830772.2830820

[68] Nikita Shamgunov. 2014. The MemSQL in-memory database system. In *Proceedings of the International Workshop on In Memory Data Management and Analytics (IMDM@ VLDB)*.

[69] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso. 2017. doppioDB: A hardware accelerated database. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–1. DOI: https://doi.org/10.23919/FPL.2017.8056864

[70] Evangelia A. Sitaridi and Kenneth A. Ross. 2013. Optimizing select conditions on GPUs. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'13)*. ACM Press, New York, New York, 1. DOI: https://doi.org/10.1145/2485278.2485282

[71] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'11)*. ACM, New York, NY, 33–40. DOI: https://doi.org/10.1145/1995441.1995446

[72] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database analytics acceleration using FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 411–420.

[73] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. ACM, 625–636. DOI: https://doi.org/10.1145/1989323.1989389

[74] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in wonderland: Where do cache misses come from in major OLTP components? In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'13)*. ACM, 8.

[75] TPC. 2007. TPC-H Benchmark. Retrieved from http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.

[76] Antonino Tumeo, Simone Secchi, and Oreste Villa. 2012. Designing next-generation massively multithreaded architectures for irregular applications. *Computer* 45, 8 (2012), 53–61.

[77] K. Wadleigh, J. Amelio, K. Collins, and G. Edwards. 2012. Poster: Hybrid breadth first search implementation for hybrid-core computers. In *Super Computing C Companion*. IEEE, 1355–1355. DOI: https://doi.org/10.1109/SC.Companion.2012.185

[78] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 385–394. DOI: https://doi.org/10.14778/1687627.1687671

[79] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. DOI: https://doi.org/10.14778/2732967.2732972

[80] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating big data with high-throughput, energy-efficient data partitioning. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 249–260. DOI: https://doi.org/10.1145/2508148.2485944

[81] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The architecture and design of a database processing unit. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 255–268. DOI: https://doi.org/10.1145/2541940.2541961

[82] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. 2015. Beyond the wall: Near-data processing for databases. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'15)*. ACM, New York, NY, Article 2, 10 pages. DOI: https://doi.org/10.1145/2771937.2771945

[83] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. 2005. Improving database performance on simultaneous multithreading processors. In *Proceedings of the International Conference on Very Large Databases (VLDB'05)*. VLDB Endowment, 49–60. http://dl.acm.org/citation.cfm?id=1083592.1083602.

[84] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. ACM, 145–156. DOI: https://doi.org/10.1145/564691.564709

[85] Craig B. Zilles, Joel S. Emer, and Gurindar S. Sohi. 1999. The use of multithreading for exception handling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'99)*. IEEE Computer Society, Washington, DC, 219–229. http://dl.acm.org/citation.cfm?id=320080.320114.

[86] Marcin Zukowski, Niels Nes, and Peter Boncz. 2008. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'08)*. ACM, New York, NY, 47–54. DOI: https://doi.org/10.1145/1457150.1457160

[87] M. Zukowski, M. van de Wiel, and P. Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE'12)*. 1349–1350. DOI: https://doi.org/10.1109/ICDE.2012.148

Received May 2018; revised January 2019; accepted January 2019