
DATABASE ANALYTICS: A RECONFIGURABLE-COMPUTING APPROACH

THIS ARTICLE PRESENTS A HIGHLY PIPELINED, HIGH-THROUGHPUT QUERY-PROCESSING ENGINE ON A FIELD-PROGRAMMABLE GATE ARRAY TO OFFLOAD AND ACCELERATE EXPENSIVE DATABASE-ANALYTICS QUERIES. THE SOLUTION PRESENTED HERE PROVIDES A MECHANISM FOR A DATABASE MANAGEMENT SYSTEM TO SEAMLESSLY HARNESS THE FPGA ACCELERATOR WITHOUT REQUIRING ANY CHANGES IN THE APPLICATION OR THE EXISTING DATA LAYOUT. THE SYSTEM ACHIEVES AN ORDER-OF-MAGNITUDE SPEEDUP ON VARIOUS REAL-LIFE QUERIES.

••••• Real-time analytics—performing analytics directly on transactional data—has seen widespread adoption in the business world in recent years. Whether to set airline ticket prices to beat competition or to stock store shelves just in time, rapid analysis of transactional data has become a business necessity. Snapshot warehousing, where a snapshot of data is taken from an online transaction processing (OLTP) system for decision-support analysis, is no longer sufficient for making executive decisions. Widespread use of mobile devices and social-media networking has changed the marketplace dynamics, requiring businesses to respond to market changes in real time. For example, just the appropriate coupon for a coffeehouse must be sent to a consumer’s smartphone within seconds of the customer swiping his or her credit card at a nearby store.

Injecting expensive analytics queries involving operations such as sort and join

into OLTP environments leads to sharing of system resources such as the CPU and I/O between transactional and analytical workloads. Transactional workloads relate directly to revenue generation and have strict service-level agreements (SLAs); analytical workloads must run against the same data without impacting these SLAs.

Parallelization techniques commonly applied to speed up query processing include multicore architectures;¹ single-instruction, multiple-data (SIMD) operations;² and accelerators such as GPUs³ and field-programmable gate arrays (FPGAs).⁴ Efficient use of multicore demands careful data partitioning, and exploiting SIMD operations and GPU requires extra preprocessing on database tables.⁵ Prior work using FPGAs involve synthesizing each query into the FPGA, making them not very viable. Commercial data-warehousing appliances—such as Netezza (www.netezza.com), Exadata

Bharat Sukhwani
Hong Min
Mathew Thoennes
Parijat Dube
Bernard Brezzo
Sameh Asaad
Donna Eng Dillenberger
IBM T.J. Watson Research
Center

Employee ID	First name	Last name	Department	Year of joining
1201	Robert	Rose	Engineering	2001
1202	Jack	Smith	Sales	2003
1203	Anna	Morris	HR	2001

Select:	Emp_Id, Dept	} Projection
From:	Employee	
Where:	Joining_year < 2002	} Predicate
Order by:	Emp_Id	} Sort

Figure 1. An example “employee” table and a Structured Query Language (SQL) query. SQL queries use various clauses to represent different data-retrieval operations. This query retrieves the employee ID and department of all employees who joined the company before 2002.

(www.oracle.com/us/products/database/exadata), and Greenplum (<http://gopivotal.com/products/pivotal-greenplum-database>)—focus solely on analytics workloads and operate on more uniformly formatted data compared to OLTP workloads.

To address these challenges, we propose a hardware acceleration approach to offload and accelerate the most CPU-intensive operations in analytics queries on an FPGA. Our system performs analytics on OLTP data within an OLTP environment. Unlike most existing approaches, the FPGA in our system operates on a database management system’s in-memory data, which is the most up-to-date copy of the data, for real-time analytics alongside OLTP without any preprocessing or partitioning of data.

Relational databases and query processing

In relational database management systems (DBMSs), data is organized in the form of records in a table, where each record (row) contains one or more fields (columns). Tables are stored on the disk as a collection of pages, each containing multiple rows.⁶ In most enterprise DBMSs, rows are compressed for storage

and I/O savings, and are decompressed only while the DBMS evaluates the queries. A designated memory space, called a buffer cache or buffer pool, is used for caching the data pages, and the I/O operations between the buffer pool and the disk are managed transparently. Data residing in the buffer pool is the most up-to-date *hot* copy of the data.

Structured Query Language (SQL) is the de facto standard for schema definition, data manipulation, and querying of data from relational DBMSs. SQL queries use various clauses to represent different data-retrieval operations (see Figure 1). The “where” clause represents predicates used to qualify a subset of table records via logical inequality or equality comparisons, or to test set containment for a field in the record. The select clause refers to projection and indicates the record fields to be extracted. Finally, the order by clause specifies the fields for sorting the results. Other common query operations include table joins, which match records with common column values from two or more tables, and group-by aggregations, which involve clustering the output (for example, sum or average) according to certain fields.

DBMSs commonly employ two approaches of traversing a table: indexing and table scanning. Indexing is efficient for locating one or a few records, as is the case for OLTP queries. Scanning involves sifting through the entire table and is commonly used in analytics, where a large number of records typically match the search criteria. It is often expensive to scan large tables on general-purpose CPUs. A mechanism to accelerate the analytics queries involving table scan is thus highly desirable.

Database analytics accelerator on FPGAs

Our FPGA-accelerated database analytics system consists of an off-the-shelf host server with a PCI Express (PCIe)-attached FPGA accelerator card, and it runs a commercial DBMS. Our accelerator is connected directly to the processor’s main memory instead of being in the I/O path, and it processes the latest in-memory data in the DBMS’s buffer pool. To offload a query, the DBMS issues a command to the FPGA via the control software (see Figure 2). The command contains

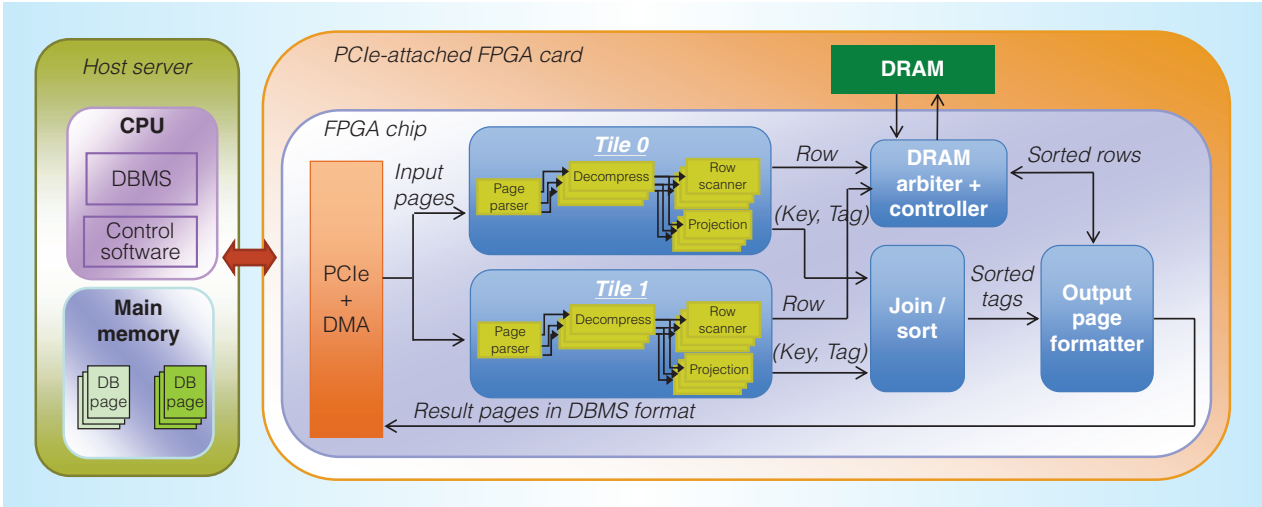


Figure 2. Field-programmable gate array (FPGA)-accelerated database system. Database pages are copied from the host memory into the FPGA over PCIe. Rows from these pages stream through a series of pipelined query operators, and qualifying rows are packed back in database-formatted pages and copied back to the host memory.

the query specification and pointers to the data that is to be processed. The FPGA pulls standard database pages from the main memory, parses the pages to extract, and processes the rows and writes the qualifying rows back to the main memory in database-formatted pages. All the data transfers are managed by the accelerator without any DBMS involvement. Standard techniques such as block DMA operations, job queuing, and double buffering have been implemented to achieve high accelerator utilization and peak PCIe bandwidth. Also, the query-processing engine is designed to match the sustained bus bandwidth and allow trade-off of query processing throughput for complexity to handle complex queries.

Below, we present an overview of the query acceleration pipeline on FPGA, followed by the different query operations implemented on the FPGA.

Query acceleration pipeline on FPGAs

Query processing in our FPGA accelerator proceeds in a streaming fashion; rows stream through a series of pipelined execution units, each implementing one query operation (see Figure 2). Operations such as join and sort are not purely streaming, and they require the rows to be temporarily held in the accelerator. Because the FPGA block

RAM (BRAM) is limited, only the columns necessary for these operations are held in BRAM, whereas the full records are stored in on-card DRAM. Because predicate evaluation and projection are performed before join and sort, disqualified rows and unwanted columns are eliminated, significantly reducing the amount of data to be stored. The filtered rows that pass the query operations are read from the DRAM and streamed back to the host in uncompressed page format for direct consumption by the application.

Because of architectural limitations, current FPGA devices operate at an order of magnitude slower clock than high-end CPUs. To bridge this gap and achieve further performance improvement, FPGA designs rely heavily on exploiting a high degree of parallelism. To that end, our design is architected to exploit parallelism at various levels. Our FPGA design obtains page-level parallelism by maintaining two or more concurrent page streams (tiles). Within each tile, the FPGA achieves row-level parallelism by concurrently evaluating multiple records from a page. Finally, our design obtains finest-grained parallelism by concurrently evaluating multiple predicates against different columns within the row.

Depending on the target workload (set of queries), our design can be scaled to achieve

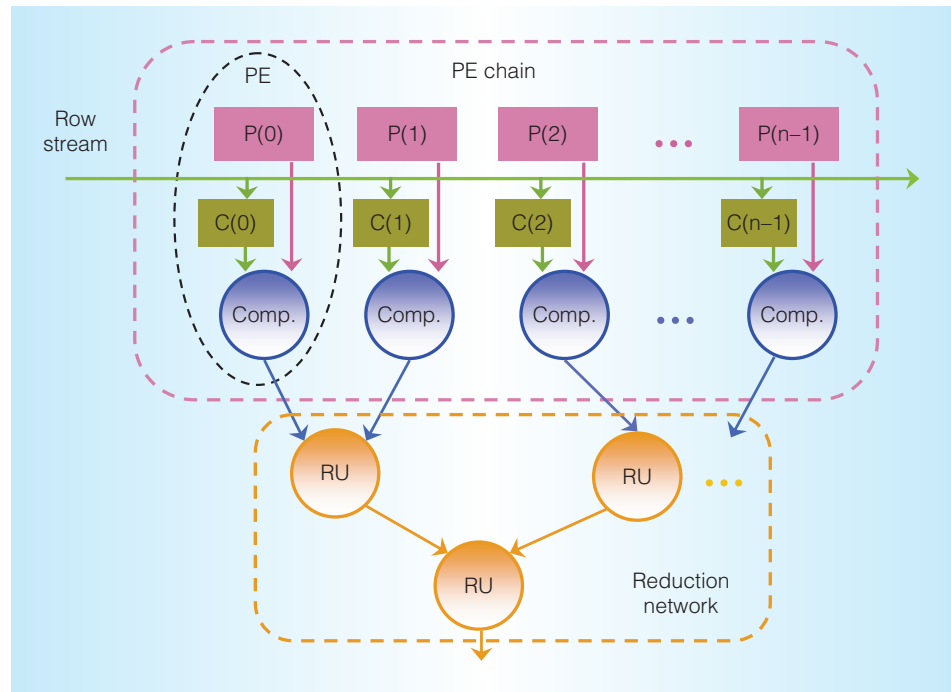


Figure 3. Row scanner for predicate evaluation. PEs operate concurrently and independently of one another.

maximum performance, given the available FPGA resources. In other words, the number of parallel page streams and parallel row-processing engines can be traded against the type and complexity of the query operations. The best-matching configuration can be selected from a library of prebuilt configurations. Loading a new FPGA configuration typically takes a few hundred milliseconds, which is insignificant compared to the runtime of typical analytics queries.

Row decompression

Offloading decompression enables the FPGA to directly consume compressed database pages and accelerate the per-row decompression task while also providing higher “effective” PCIe data-transfer bandwidth. Bandwidth advantage is especially important because the overall accelerator performance is often limited by the bus bandwidth.

Our target DBMS uses a dictionary-based modified Liv-Zempel compression algorithm, in which a compressed row consists of a series of symbols. Decompression is inherently serial, involving symbol expansion

using dictionary lookups, where the next lookup depends on the current one. Moreover, being a variable-byte operation with random lookups, decompression consumes significant CPU cycles and often suffers from huge cache misses.

On the FPGA, an optimized decompression datapath results in a highly efficient decompression operation. For deterministic, single-cycle lookups, the decompression dictionary is preloaded into FPGA Block RAMs. The dictionary must be reloaded only if the database table changes. For improved pipeline efficiency, a symbol prefetch logic is implemented; compared to the nonprefetch FPGA design, prefetching helps reduce the decompression time by more than 50 percent in some cases.⁷ The decompression engine has a very small footprint, allowing the instantiation of multiple engines on the FPGA for row-level parallelism.

Predicate evaluation

Predicate evaluation selects the rows of interest from a table by applying filtering criteria (predicates) on one or more columns

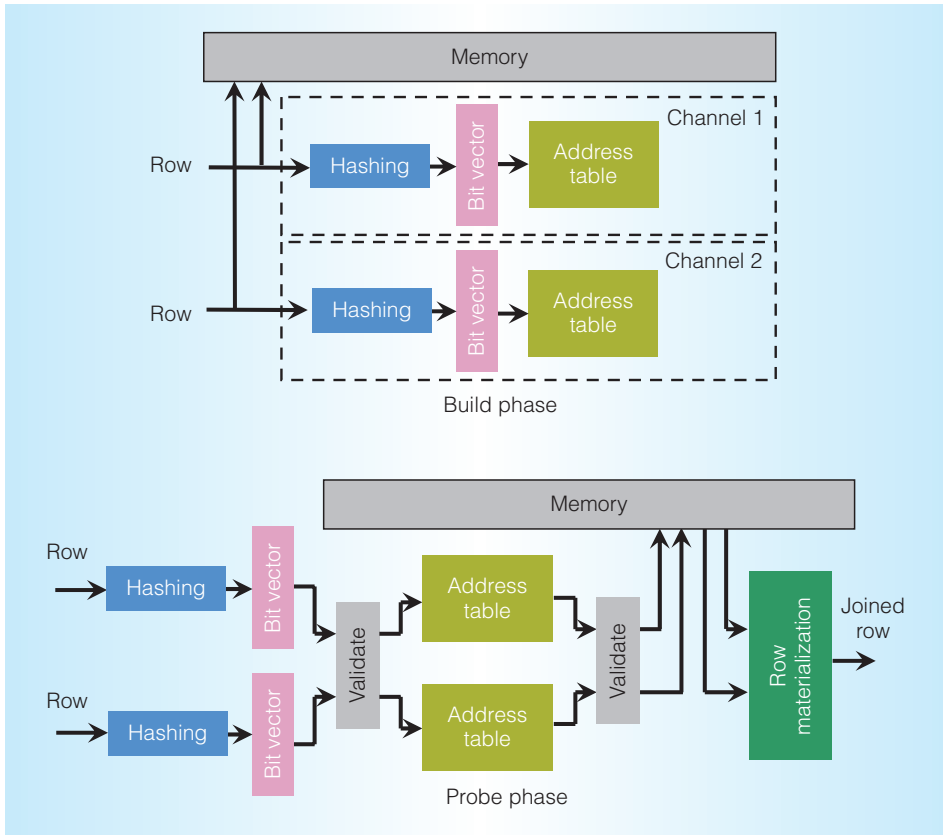


Figure 4. Build and probe phases of hash-join on FPGA. Bit vectors are used to quickly eliminate most of the nonmatching rows, whereas the address tables are used for eliminating the false positives. Only the true positive matches result in off-chip memory access.

of the rows. From a CPU consumption perspective, predicate evaluation against each row becomes an intensive operation as the number of predicates increases, especially for large analytics queries, which often involve filtering millions to billions of rows.

In our FPGA design, predicates are evaluated by a chain of predicate evaluation units (PEs) inside the row scanner (see Figure 3). Each PE evaluates a single predicate by comparing a stored predicate value, $P(i)$, supplied by the query against up to a 64-bit-long column of the streaming database row. PEs operate concurrently and independently of one another. A configurable, binary reduction network (RN) combines the individual outputs of the PEs to implement complex filter criteria according to the query, generating a 1-bit qualify signal. The number of PEs inside a row scanner and the size of the

reduction network are configurable at synthesis time, to allow for area-versus-complexity trade-offs. Software can further customize a given FPGA configuration by loading the PE operation, the predicate value, and arbitrary reduction patterns on a per-query basis.

Table joins

The join operation aims to match records from one table against the other, using the join column. Because the naive nested-loop implementation has quadratic computational complexity, most DBMSs employ sort-merge-join or hash-join for efficient computation. Hash-join often performs better because of its linear algorithmic complexity, though hashing introduces false positives that must be resolved. Besides, hash-join on CPUs suffers from high cache misses due to

random memory accesses based on the hash value.

We implement a hash-based join in our FPGA, which performs hashing, hash conflict resolution, and row materialization (creation of the joined rows) at the PCIe streaming rate.⁸ Two or more join operations can be performed concurrently using parallel join channels.

Figure 4 shows the two phases of FPGA hash-join. During the build phase, one of the join tables (usually the smaller one) is streamed through, and the join columns are hashed to populate a bit vector. The full rows are stored in off-chip DRAM, whereas the join columns and the row addresses are stored in the address table in the FPGA BRAM; rows hashing to the same position are chained in the address table.

The second table is streamed during the probe phase; rows are hashed for probing the bit-vector, for quick elimination of non-matching rows. The FPGA subsequently removes the false positives by comparing against the values in the address table. Only true matches issue reads from off-chip memory and are materialized. This multilevel filtering significantly reduces off-chip accesses and is critical to handling streaming rows without stalls.

Column projection

Queries often request only a few columns from each record for reporting or analytics purposes. Including projection in the accelerator's pipeline thus provides significant bandwidth, processing, and storage savings by removing unwanted columns in early stages. Moreover, projection is required to extract the columns for sort and join operations and format the output for consumption by the DBMS.

Projection in our FPGA is performed concurrently with predicate evaluation and the projected row is carried forward if it qualifies. The FPGA design maintains row-level parallelism by instantiating multiple copies of the projection logic, one for each row scanner.

The DBMS specifies the projection columns by loading their lengths and positions in the row into the FPGA BRAM during the query load phase. We store this information

in the order of those columns in the record, thereby requiring only one comparison at a time. This allows for very efficient implementation of the projection logic, independent of the number of columns to be projected. As the rows stream over the projection unit, the required columns are captured while the rest are discarded.

Database tables often contain columns whose length and position vary from row to row (for example, a string). Projecting such columns requires computing this information for each row from the row metadata. This can significantly affect the performance. Our FPGA projects variable-length columns at the full streaming rate by employing a two-phase hybrid streaming scheme. First, the row is staged in a buffer, and the length and position for all the variable columns to be projected are computed. We refer to this as resolving the variable columns into fixed columns. Next, the row is streamed through, skipping-over the metadata, as it is no longer required; variable columns are then projected, much like the fixed-length columns. The savings from skipping the metadata more than compensates for the extra cycles spent during the resolution step.

Database sort

Database sort keys are created by extracting one or more fields from a much larger row. These keys are thus generated once every few cycles, obviating the need for the fastest sort implementation. Instead, the main requirements for sort in databases are support for long sort keys (tens of bytes or longer), handling large payloads (rows) associated with each key, and generating large sorted batches (millions of records) in multiphased sort. These requirements direct toward a highly hardware-efficient algorithm, ideally having the hardware resource requirements independent of the number of records to be sorted. Furthermore, database applications prefer streaming sort for large-query processing.

For these reasons, we find the tournament tree sort algorithm most suitable.⁹ Among different hardware sorters, tournament tree sort requires the least number of comparators and can generate large sorted batches. A tree with N leaf nodes guarantees a minimum batch of size N ; however, for almost-sorted

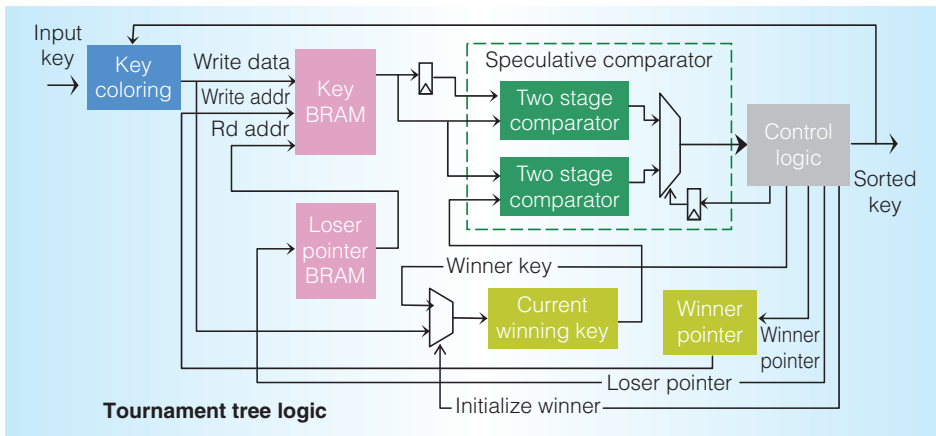


Figure 5. Tournament tree sort implementation on an FPGA. The tree is implemented using block RAMs and only two comparators. Key coloring is used to avoid the need to flush the tree between consecutive sorted runs.

data, which happens frequently in databases, much larger batches can be generated.

To the best of our knowledge, this is the first hardware implementation of the tournament tree algorithm. Our FPGA design implements two independent tournament tree sorters, each with 16 thousand nodes, followed by a merger, thereby generating minimum batches of 32 thousand records.

For each tree, the leaf nodes containing the sort keys are stored in the key BRAM, whereas the corresponding row is stored in off-chip DRAM. Row pointers are stored with the keys and are used to fetch the rows as sorted keys get emitted. The nonleaf nodes contain the pointers to the losing keys (leaf nodes) at different tree levels and are held in the loser pointer BRAM (see Figure 5).

Each new key is entered into the leaf node of the last sorted key, and moves this new key up the tree by comparing it with the parent nodes, updating the loser pointers and the overall winning key as needed. The tree is traversed using simple modulo arithmetic to compute the parent key's address. The entire tree is serviced using a single comparator. To meet timing, the large comparator must be implemented as a multistage pipeline. For pipelining of dependent comparisons, we implement a speculative comparator using two independent two-stage comparators. The overall sort design uses just five two-stage comparators—two per tree, plus one for the

merger—while sustaining full PCIe bandwidth for most row sizes. We can achieve higher throughputs by further splitting the trees at the expense of more comparators.

A tournament tree can continue to sort beyond the minimum batch size, provided that the incoming keys don't violate the order of the already-sorted keys. Unless handled in a special way, such violating keys require flushing the keys in the tree and starting a new batch from an empty tree, incurring extra tree setup and teardown costs and significantly reducing the sorting throughput. We address this by coloring the keys as they enter the tree. In other words, we attach a prefix to each key, thereby implicitly binning the keys on the fly into different sorted batches. A differently colored key always loses against any key of the current color. Coloring thus allows the violating keys to participate in sorting without corrupting the current sorted batch, thereby eliminating the need to drain the tree between batches.

Acceleration enablement of DBMS

Leveraging the FPGA-accelerated query processing requires two modifications to a standard DBMS: restructuring of the DBMS for seamless plug-in of the accelerator into the DBMS' query operation flow, and transforming the different query operations to map to the hardware-accelerated functions.

For efficient accelerator utilization and high query-processing throughput, the interactions between the DBMS and the I/O-attached accelerator must happen at the block level as opposed to the traditional one-row-at-a-time flow. For this purpose, we restructured the DBMS to introduce block-level data operations within the DBMS query flow. The restructured DBMS communicates with the FPGA through a series of control blocks.⁷ A long-running query is divided into multiple jobs. For each job, the DBMS obtains a list of buffer pool pages to be processed and locks them in the host memory; the list of page pointers is sent to the FPGA. Note that locking can affect the OLTP queries, and issuing multiple lightweight jobs is critical to maintaining a fine locking granularity. Multiple outstanding jobs are issued and queued to the accelerator, which processes them sequentially. The FPGA returns the results in the format expected by the DBMS processing engine for further downstream processing; hence, no additional data copying or formatting is required.

For mapping the query operations onto the FPGA, the query is transformed into a query control block (QCB), a data structure that contains information about the record structure, the query predicates, and other query operations. A QCB can be interpreted by the FPGA to tailor the application logic to a specific query. Based on the target query, the DBMS must transform a query into QCB specifications. As SQL provides rich syntax to express data relations, simply parsing the SQL query is not sufficient to provide all the required information in the QCB.

Existing SQL parsing and transformation routines in the DBMS provide some of the intermediate expressions required for generating the query specifications for the FPGA. We devised techniques to efficiently parse these internal expressions to extract the relevant information for creating the QCB. Moreover, because certain queries might not benefit from FPGA offload, the query transformation function also decides if a query should be directed to the FPGA.

Performance evaluation

Our prototype is built on a commercial DBMS running on a 3.8-GHz multicore

superscalar system with a PCIe-attached FPGA card with an Altera Stratix 5SGXA7 and 8 Gbytes of DDR3 at 1,333 megabits per second (Mbps). The FPGA design runs at 200 MHz. Our experimental workload is derived from real customer tables, and we evaluated four types of queries:

1. predicate evaluation only;
2. decompression and predicate evaluation;
3. decompression, predicate evaluation, projection, and sort; and
4. decompression, predicate evaluation, and join.

These queries resemble the TPC-H Q1 and TPC-DS template-3 queries.

Figure 6 shows the CPU savings from offloading analytics queries with different row qualification ratios. Higher offload means more CPU resources being freed up for OLTP. As shown, the savings are higher when a smaller fraction of rows qualify, gradually decreasing with increasing qualification ratio. This is due to the CPU requirements for post-processing the qualified rows—for example, moving data to the application buffer. Overall, CPU savings are higher on type 2 queries compared to type 1 queries; the offload for type 4 queries would be even higher due to a larger fraction of the query task being offloaded. In other words, larger CPU savings can be achieved on such queries, even with a large number of rows qualifying. Type 3 queries, however, could require a merge step on the CPU, depending on the sort batch size generated by the FPGA, potentially resulting in a slight reduction in the overall savings. This depends on the particular workload's characteristics.

From the perspective of performance improvement on the offloaded queries, Table 1 compares the row processing throughput in the FPGA against the baseline unmodified DBMS running on a single core of our system. Overall, the FPGA achieves speedups in the range of 7× to 14× for most queries, except for type 1 queries. Type 1 queries operate on uncompressed data, and the performance is limited by the data transfer bandwidth over the PCIe bus. This is evident in going from type 1 to type 2 queries: throughput in the FPGA increases significantly,

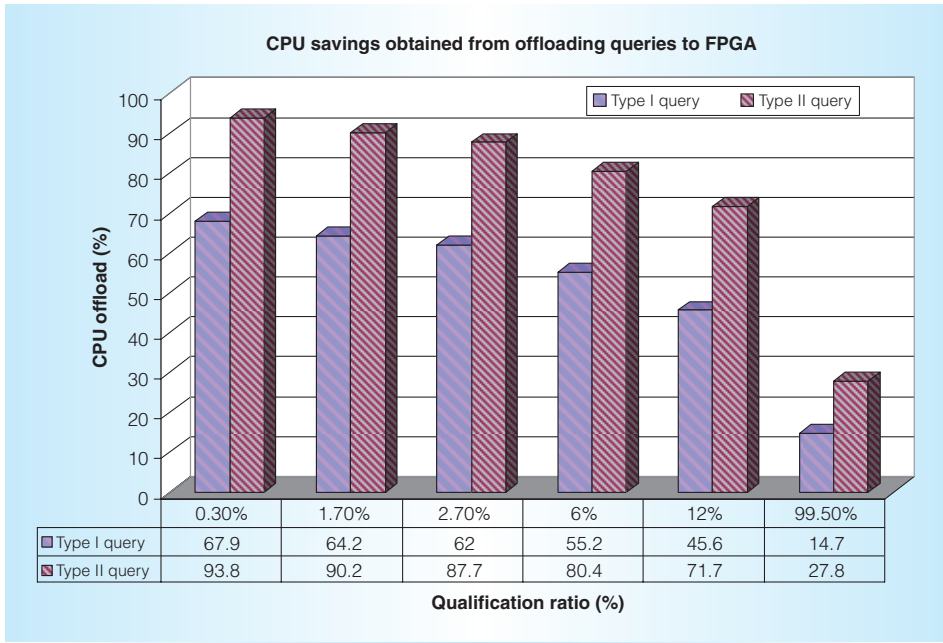


Figure 6. CPU savings in OLTP system from offloading type 1 and type 2 queries to the FPGA accelerator. Higher query qualification ratios reduce the overall CPU savings due to the CPU requirements for post-processing of qualified data.

Table 1. Row processing throughput and FPGA speedup.

Query type*	Row length (bytes)	Compression factor	Throughput (million rows per second)		FPGA speedup
			Baseline	FPGA	
1	170	1×	12.57	14.60	1.1×
1	235	1×	8.16	10.15	1.2×
2	170	5×	3.57	38.20	10.7×
2	235	2×	3.16	21.20	6.7×
3	80	2×	2.48	28.00	11.3×
3	170	2×	2.05	28.00	13.6×
3	235	2×	1.62	21.00	12.9×
3	420	2×	1.30	19.00	14.6×
4	170	5×	1.60	18.00	11.2×

* Query type 1: predicate evaluation only; query type 2: decompression and predicate evaluation; query type 3: decompression, predicate evaluation, projection, and sort; query type 4: decompression, predicate evaluation, and join.

owing to the increase in effective bandwidth, whereas the CPU throughput falls drastically because of the cost of decompressing each row. On the FPGA, better compression results in higher speedups, and very high

compression ratios cause the performance bottleneck to shift from PCIe to the FPGA's query-processing capabilities.

For more complex type 3 and type 4 queries, the throughput on the baseline CPU

further decreases, whereas the FPGA throughput is relatively stable. This is due to the pipelined execution of these operations in the FPGA. The main factor that affects the FPGA throughput is the row size; the smaller the row, the higher the throughput, because more rows can be transferred over a given PCIe bandwidth. For type 3 queries, however, up to 180-byte rows, the performance is limited by the sort tree, and doesn't improve for smaller row sizes. In terms of execution complexity, "join" is the most expensive operation, in both the CPU and the FPGA. For similar-sized rows, the throughput for the type 4 query is about half that of other queries.

Offloading analytics queries to an FPGA-based accelerator provides a viable, attractive solution for running analytics queries against transactional data, for real-time market responsiveness. This approach provides significant improvement in response time of offloaded analytics queries while saving CPU resources for mission-critical OLTP workloads. Our solution can be easily scaled across multiple accelerator nodes, either for a single query or across different queries, without any explicit data partitioning.

MICRO

References

1. J. Krueger, et al., "Fast Updates on Read Optimized Databases Using Multicore CPUs," *Proc. VLDB Endowment*, vol. 5, no. 1, 2012, pp. 61-72.
2. R. Johnson, et al., "Row-Wise Parallel Predicate Evaluation," *Proc. VLDB Endowment*, vol. 1, no. 1, 2008, pp. 622-634.
3. N. Satish, et al., "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD 10)*, 2010, pp. 351-362.
4. R. Mueller, J. Teubner, and G. Alonso, "Glacier: A Query-to-Hardware Compiler," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD 10)*, 2010, pp. 1159-1162.
5. T. Horikawa, "An Unexpected Scalability Bottleneck in a DBMS: A Hidden Pitfall in Implementing Mutual Exclusion," *Proc. Parallel and Distributed Computing and Systems (PDCS 2011)*, 2011, doi:10.2316/P.2011.757-036.
6. R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed., McGraw-Hill, 2002.
7. B. Sukhwani, et al., "Database Analytics Acceleration Using FPGAs," *Proc. 21st Int'l Conf. Parallel Architectures and Compilation Techniques*, ACM, 2012, pp. 411-420.
8. R. Halstead et al., "Accelerating Join Operation for Relational Databases with FPGAs," *Proc. IEEE Symp. FCCM*, IEEE CS, 2013, pp. 17-20.
9. D.E. Knuth, *The Art of Computer Programming, Vol. 3—Sorting and Searching*, Addison-Wesley, 1973.

Bharat Sukhwani is a research staff member at the IBM T.J. Watson Research Center. His research interests include computer architecture; parallel computing; application-specific, high-performance computing systems; and computing using reconfigurable and graphics processors. Sukhwani has a PhD in electrical engineering from Boston University. He is a member of IEEE.

Hong Min is a senior technical staff member at the IBM T.J. Watson Research Center. Her research interests include database systems and acceleration technologies for data processing. Min has a PhD in mechanical engineering from Drexel University.

Mathew Thoennes is a senior engineer at the IBM T.J. Watson Research Center. His research interests include software and hardware for Series z systems. Thoennes has an MS in computer science from the University of Massachusetts. He's a member of the ACM and a senior member of IEEE.

Parijat Dube is a senior engineer at the IBM T.J. Watson Research Center. His research interests include performance modeling, analysis, and optimization of systems. Dube has a PhD in computer science from INRIA.

Bernard Brezzo is a senior engineer at the IBM T.J. Watson Research Center. His


research interests include high-performance computing, FPGA emulation, and Synapse. Brezzo has a Diploma in electronics engineering from Conservatoire National des Arts et Metiers, Paris.

Sameh Asaad is a research staff member at the IBM T.J. Watson Research Center, where he manages the Digital Design Group in the Communication and Computation Subsystems Department. His research interests include domain-optimized architectures and systems, reconfigurable computing, and FPGA-based application acceleration. Asaad has a PhD in electrical engineering from Vanderbilt University.

Donna Eng Dillenger leads IBM's global research on enterprise systems at the

IBM T.J. Watson Research Center. She's also IBM's Chief Technology Officer of IT Optimization, an IBM Distinguished Engineer and Master Inventor, and an adjunct professor at Columbia University. Her research interests include hybrid architectures, Cloud, and IT optimization. Dillenger has an MS in computer science from Columbia University.

Direct questions and comments about this article to Bharat Sukhwani, IBM Research, 1101 Kitchawan Road, Yorktown Heights, NY 10598; bharats@us.ibm.com.

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.



Call for Articles

IEEE Pervasive Computing seeks accessible, useful papers on the latest peer-reviewed developments in pervasive, mobile, and ubiquitous computing. Topics include hardware technology, software infrastructure, real-world sensing and interaction, human-computer interaction, and systems considerations, including deployment, scalability, security, and privacy.

Author guidelines:
www.computer.org/mc/pervasive/author.htm

Further details:
pervasive@computer.org
www.computer.org/pervasive

 **IEEE pervasive COMPUTING**
MOBILE AND UBIQUITOUS SYSTEMS