Database Analytics Acceleration using FPGAs

Bharat Sukhwani[§], Hong Min[§], Mathew Thoennes[§], Parijat Dube[§],
Balakrishna Iyer[†], Bernard Brezzo[§], Donna Dillenberger[§], Sameh Asaad[§]

[§]IBM T. J. Watson Research Center
Yorktown Heights, NY, 10598

[†]IBM Santa Teresa Lab
555 Bailey Ave, San Jose, CA 95141

{bharats, hongmin, tardis, pdube, balaiyer, brezzo, engd, asaad}@us.ibm.com

ABSTRACT

Business growth and technology advancements have resulted in growing amounts of enterprise data. To gain valuable business insight and competitive advantage, businesses demand the capability of performing real-time analytics on such data. This, however, involves expensive query operations that are very time consuming on traditional CPUs. Additionally, in traditional database management systems (DBMS), the CPU resources are dedicated to mission-critical transactional workloads. Offloading expensive analytics query operations to a co-processor can allow efficient execution of analytics workloads in parallel with transactional workloads.

In this paper, we present a Field Programmable Gate Array (FPGA) based acceleration engine for database operations in analytics queries. The proposed solution provides a mechanism for a DBMS to seamlessly harness the FPGA compute power without requiring any changes in the application or the existing data layout. Using a software-programmed query control block, the accelerator can be tailored to execute different queries without reconfiguration. Our prototype is implemented in a PCIe-attached FPGA system and is integrated into a commercial DBMS platform. The results demonstrate up to 94% CPU savings on real customer data compared to the baseline software cost with up to an order of magnitude speedup in the offloaded computations and up to 6.2x improvement in end-to-end performance.

Categories and Subject Descriptors

C.3 [Special-purpose and application-based systems];
H.2.4 [Database Management]: Systems

Keywords

Relational database, analytics, FPGA, acceleration

1. INTRODUCTION

We find ourselves, today, in the midst of the widespread adoption of analytics in the business world. Whether it is to stock store shelves just-in-time, set airline ticket prices to maximize yield, or purchase hedge contracts on fuel prices to make a transportation company's forthcoming quarter's budget more predictable, rapid analysis of transactional data in corporate database management systems has become a business necessity. Snapshot warehousing, such as in [1], is the existing practice where a snapshot of data is taken from an online transaction processing (OLTP) system to a warehouse system for decision support analysis.

Snapshot warehousing has been acceptable in the past when updates to the database were infrequent. Performing analysis on a month old, week old or day old snapshot of business data was good enough for making executive decisions pertaining to business. Interpersonal networking, the empowering of individuals by personal communication devices and the wide use of social media networking has changed the dynamics of the marketplace. Increasingly, businesses are finding themselves required to respond to changes in market conditions in real time. For example, an airline may have the luxury of no more than a few minutes to respond to fare modifications by its competitor. Real time analytics is the capability to process analytical queries and other operations directly on transactional data.

Injecting expensive analytics query operations into the data operating environment also poses challenges. One challenge is that system resources such as CPU and I/O have to be shared between transactional and analytical workloads. Normally, transactional workloads have stringent Service Level Agreements (SLAs); further, because these are directly tied to revenue generation, they are often the primary focus of the business. Analytical workloads need to run against the same data without impacting the SLA of the transaction workloads. Meeting this challenge requires consideration of both CPU and I/O resource issues.

In this paper we focus on the CPU issue. We propose a hardware acceleration approach that offloads and accelerates some of the most CPU-intensive query operations on an FPGA. Our architecture enables the FPGA to retrieve DBMS in-memory data, which is mostly up-to-date, perform expensive query operations, and send the results back to the database engine. The accelerator has been made customizable by architecting a query control block, and making the FPGA logic behave as an interpreter of the query control block. In this sense, one principle of object oriented programming, late binding, has been

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

implemented on the FPGA, making it versatile for DBMS queries. The presented work includes the following key contributions:

- an architecture that integrates an FPGA into a host system and processes data extracted from the memory of a DBMS,
- the design and prototype of FPGA functions including data decompression and predicate evaluation, both of which consume considerable CPU cycles on conventional processor architectures,
- a software integration approach that enables FPGA-based acceleration for an existing DBMS,
- experimental and performance observations from our integrated prototype.

The rest of the paper is organized as follows. Section 2 outlines the previous work. Section 3 introduces related background information on database systems. In section 4, we present the overview of our proposed architecture and system integration. Section 5 describes the FPGA design of the data decompression and predicate evaluation functions. Section 6 describes software integration of the FPGA-based accelerator into an existing DBMS system. Section 7 presents our experiments and results. Section 8 concludes the paper with discussions and future work.

2. PREVIOUS WORK

Over the years, there have been many efforts to speed up query processing in database management systems. As the increase in the speed of processors diminishes each year, further focus has been placed on parallelization techniques.

One approach is to increase the number of processors [2] and threads [3] exploited by the DBMS. The problem here lies in how to partition the data, metadata, updates and data deltas to reduce memory conflicts and contention between cores [4]. Moreover, suboptimal utilization of the cores in large multi-core systems results in power inefficiencies [5].

Single Instruction Multiple Data (SIMD) operation provides data level parallelism by processing multiple data entities in a single instruction [6][7]. Effective SIMD exploitation, however, requires the DBMS data to be laid out in specific formats and packed tightly into 64 or 128 bit registers. This requires extra processing or compilation for each database table [6]. Further work has been done to exploit Graphics Processing Units (GPUs) to aid database sort operations [8]. GPUs offer a relatively fixed architecture, however, leading to limitations similar to those of SIMD architectures.

Prior work [9][10][11] use FPGAs for parallel query processing. One of the differentiators of our work is the enablement of FPGAs to execute different queries without reconfiguration. This is different than the compilation-based approach [12] which first compiles queries into a hardware description and then synthesizes into FPGA circuit.

Data warehouse appliances such as the Netezza Performance Server [13] or the IBM DB2 Analytics Accelerator (IDAA) [14] also use FPGAs to accelerate query operations. In the former, the FPGA is in the I/O path, hence requiring a write-through I/O protocol which would increase the IOPS to unacceptable levels in an OLTP system. In IDAA, data from an OLTP system is periodically copied, which is appropriate for applications without stringent real time analytics needs.

3. DBMS DATA PAGE PROCESSING

3.1 Relational Database Tables and Pages

In relational DBMS, records are stored in objects called tables. Records are often referred to as rows, and record attributes are called columns (or fields). Table 3-1 is a simplified illustration of a three-row table with six attribute columns (PhoneNumber, FirstName, LastName, Age, State, SalesTotal($)) per row.

Table 3-1 Example "Customers" table

Phone Number	First Name	Last Name	Age	State	SalesTotal($)
212-111-1111	Ann	Smith	25	NY	250.54
212-111-0000	Steve	Jones	31	NY	500.00
203-222-2222	Emily	Brown	29	CT	900.01

Typically, the physical unit of storage and I/O processing of a non in-memory database table is a page. All pages in a table are the same size such as 4KB, 8KB, 16KB, 32KB etc. A database has designated memory space, called buffer cache or buffer pool (BP), for temporarily storing the data pages. All relational data operations get the data pages from the BP and the I/O operations between the BP and the disk are managed transparently. When a page is updated, including insertions and deletions, its BP image is committed first, before eventually being written to the disk. Enabling processing of the latest data in the BP is the main reason that our accelerator is connected to the memory instead in the I/O path, as is done in [13].

3.2 Page Format

In transactional database systems, data is typically stored in a row-based layout where all the columns of a row are stored in contiguous storage. A page is a collection of slots that each contains a row [15]. At the end of a page, there is an array whose entries contain the offsets of the rows within the same page. The pair <pageID, slot number> is often referred to as record ID (RID), which uniquely identifies a row within a table. Figure 3-1 illustrates such page format.

When processing a row in a table, the corresponding page is read from the buffer pool (if necessary, disk I/O is involved) and the row offset is used to extract the row from the page. If a row is deleted, its corresponding slot number holds an invalid value.

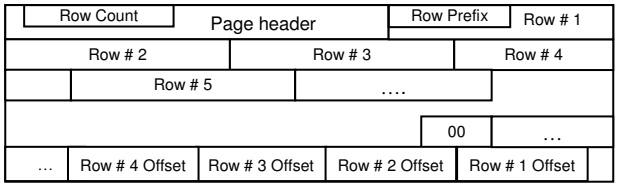

Figure 3-1 Example page format

3.3 SQL Predicate Evaluation

Structured Query Language (SQL) has become the de facto standard language for schema definition, data manipulation and data query for relational DBMS. Large high throughput enterprise class OLTP applications based on SQL power a huge number of commercial applications today.

SQL predicate evaluation refers to the process of retrieving those DBMS table rows that qualify under some criteria. The query typically may require logical inequality or equality comparisons of fields from records against constants, or test set containment for a field in a record. For example, the SQL statement *"SELECT salesTotal FROM Customer WHERE state ='NY' AND age <30"* asks for sales dollar amount from all customers in state 'NY' that are younger than 30 years old from the example Table 3-1.

Efficient access to one or few records is usually achieved in DBMS by means of an I/O efficient B+ tree index data structure based on the data retrieval key. B+ tree indexes have also been explored to retrieve large number of records.

In a DBMS that supports OLTP and analytics simultaneously, the indexes that support OLTP serve the purpose to speed up OLTP. Any indexes that are intended merely to speed up analytics, however, have a detrimental impact on OLTP because insert, update, delete operations on records could entail index updates, resultant CPU and I/O consumption and hence impact on OLTP throughput. A mechanism is needed to accelerate the evaluation of SQL predicates on relational data without building B+ tree indexes specifically to accelerate their execution.

3.4 Data Compression and Decompression

Since the early 90's, data compression has been embedded into DBMS [16]. Since OLTP applications typically only access a single or a small number of related rows, good OLTP systems select the database row as the unit of compression. DBMS data structures allow database logic to find the row; the DBMS merely decompressed the row before processing it. In the absence of indexes, the DBMS has to scan the table, decompress each row, and then apply SQL predicates against the decompressed row. The decompression technique built into the DBMS we worked with is typical of common DBMS in that decompression proceeds by taking some part of the input string and matching it against strings in a dictionary and retrieving its decompressed representation. Concatenating various decompressed fragments reproduces the decompressed row.

From a processor CPU consumption perspective, predicate evaluation against each row becomes an intensive operation as the number of predicates increase, especially against a very large number of rows. Furthermore, decompression, being a per-byte operation, significantly increases required CPU cycles. Reducing the cost of both predicate evaluation and decompression on the processor executing OLTP is critical to maintain OLTP performance.

4. DATABASE ACCELERATOR ON FPGA - SYSTEM OVERVIEW

Figure 4-1 shows the high-level system architecture for the FPGA-accelerated DBMS. The prototyped system is implemented on a PCIe-attached FPGA card and is integrated with a commercial database system. The FPGA operates on DBMS' in-memory data and writes the results back into the host CPU's main memory. Data is transferred between the FPGA and the host using direct memory access (DMA) operations. Once the DBMS sends a job request to the FPGA, all the DMA operations are initiated by the FPGA without any intervention from the host CPU.

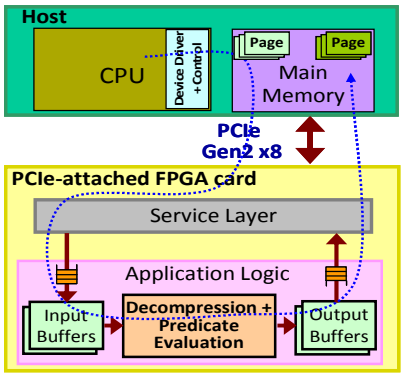

Figure 4-1 System overview of the FPGA accelerator

The FPGA solution has been structured in a modular fashion. There are two distinct pieces of logic that implement the required functionality. The service layer provides all of the logic that interfaces to the PCIe, DMA engines and job management logic. The application logic implements just the functions required to process database queries on the FPGA. A set of well defined interfaces exists between the two that include data buses for input and output data, queues for DMA requests, and control signals.

On the host CPU, a job queue is maintained and the device driver and service layer logic cooperate to dispatch the jobs to the FPGA. Once a job has been dispatched, the service layer passes the job structures to the application logic and signals it to begin. From then on, the service layer only processes the DMA requests and updates the status of the jobs to the host. This structure allows the application logic to be developed independent of the service layer.

To maximize the overall performance, the service layer must perform data transfers at full PCIe bandwidth and the application layer must process the data at the rate sustained

by the service layer. For our current bus interface, the peak bandwidth is 3.2 GB/sec (16B at 200 MHz); maintaining such rate is a challenge and must be addressed at all levels.

The first challenge is to maximize the available bus bandwidth; this is addressed in a number of ways. First, a set of job structures, which are passed to the application logic, are created for each job; these specify the DMA addresses of all the data for the job. Second, a set of queues between the service layer and the application logic is implemented that allow the application logic to queue up DMA requests, thereby maximizing the use of the DMA engines. Finally, multiple input and output buffers enable double buffering, thereby completely hiding the host-FPGA transfer latencies. With these techniques, we achieve a data transfer rate of up to 2.7 GB/sec over the PCIe bus.

Next, the FPGA must consume the 16B of data delivered every cycle. The predicate evaluation/decompression engine is designed to consume 1B every clock cycle. Replicating this engine sixteen times will enable the FPGA to consume all data being delivered. For uncompressed data, this results in a balance system. For compressed data, however, creating a balanced system becomes more difficult. For data compressed down to 40% of its original size, for example, the 16 bytes transferred on the PCIe per cycle expands to 40 bytes, consequently requiring a 2.5-fold increase in the number of engines to sustain the uncompressed bandwidth. Simply increasing the number of parallel engines may require more logic resources than the capacity of a given FPGA. Our FPGA design is thus architected such that the number of engines can be traded against the complexity of the query handled by each engine. Thus for very high levels of compression, high throughput can still be maintained, albeit for relatively simpler queries.

5. QUERY PROCESSING ON FPGA

The query processing engine on the FPGA is designed with two goals in mind: to support the most common cases in the target database system and to achieve maximum performance from the available hardware resources. Consequently, the size of the largest predicate supported, the database page buffer size and the decompression dictionary buffer size were chosen based on real-life customer workloads. While these sizes are fixed in the current design, supporting other sizes is trivial.

Figure 5-1 shows the overall architecture for database row processing on FPGA. The design is architected to exploit parallelism at various levels. The core computational unit is a predicate evaluation unit (PE), which evaluates a single predicate by comparing two up to 64-bit long quantities: a stored predicate value supplied by the query and a row field streamed in from main memory. All the predicates within a row are evaluated concurrently by the multiple instances of PEs inside a row scanner (Figure 5-4). The number of PEs inside a row scanner is configurable at synthesis time.

Multiple database rows are processed concurrently using parallel instances of row decompression and predicate evaluation logic within a *scan tile* (see Figure 5-1). Obtaining a balanced system with multiple parallel execution units requires careful rate matching and data staging. A scan tile forms a balanced unit for scanning the rows. It encapsulates the entire design flow for scanning database rows on the FPGA; the design can be scaled simply by replicating the tiles.

A scan tile contains 8 row scanners, each preceded by a decompressor, plus buffers to store the input and output database pages, logic to extract the rows from the input pages and logic to write the qualified rows in the output buffers in a database-specific page format. Each input and output page buffer is 4KB in size, to match the target database page size. Each instance of the expansion dictionary is 32 KB and is shared by two decompression units; both have concurrent access to the dictionary using two independent read ports.

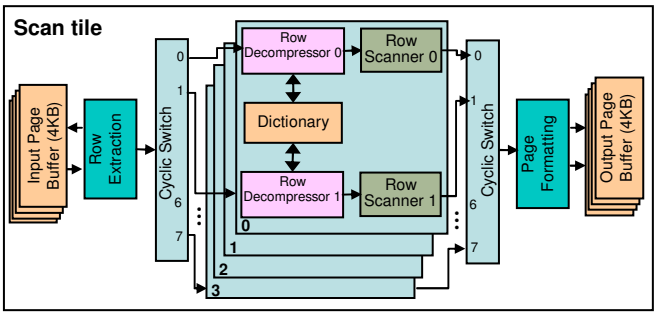

Figure 5-1 Architecture for row processing on FPGA

A scan tile scans one database page at a time. More than one page can be scanned in parallel by having multiple independent scan tiles on the FPGA. Processing at full PCIe bus bandwidth requires a minimum of two tiles, provided the pages are uncompressed or have a small compression ratio. Utilizing full bus bandwidth for highly compressed pages would require larger number of parallel scan tiles. For a given FPGA area, the number of tiles can be traded against the number of PEs within each row scanner. Depending on the query complexity and the compression ratio, different hardware configurations can be used. In one configuration, we instantiate 2 tiles for a total of 16 row scanners, each with 64 PEs, thus decompressing 16 rows concurrently and evaluating 1024 predicates in parallel. A query with fewer predicates, on the other hand, can allow more tiles and thus higher page-level parallelism. This flexibility to exploit parallelism at different granularities is an essential part of the design as it allows different database queries to obtain the highest performance from the available FPGA resources. Since generating an FPGA image is a time-consuming process, we pre-generate FPGA bit files for a variety of different hardware configurations and load the one that is best-matched to the given workload, where a workload is a collection of different queries.

5.1 Row Decompression on FPGA

In our design, the compressed rows are decompressed on-the-fly on the FPGA before being fed to the predicate evaluation logic. Performing decompression on the FPGA brings numerous benefits. First, database pages can be sent directly from the host to the FPGA without the need to pre-filter and decompress on the host. Secondly, offloading decompression to the FPGA increases the amount of computation per datum transferred to the FPGA. Thirdly, efficient parallel hardware implementation of the decompression algorithm results in improved performance. Finally, transferring compressed rows increases the "effective" PCIe transfer bandwidth. This is especially important since the overall accelerator performance is often limited by the available host-to-FPGA data transfer bandwidth. Depending on the compression ratio, transferring compressed rows increases the effective bandwidth by a factor of 2 to 5.

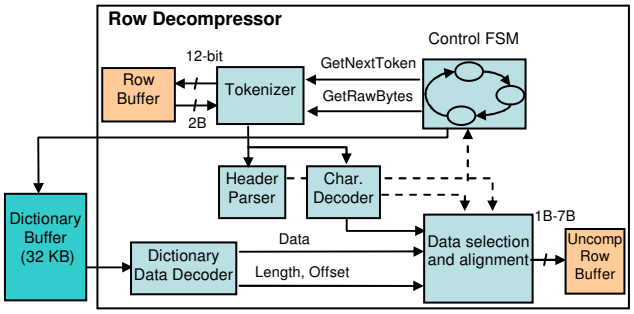

Figure 5-2 Row decompression logic on FPGA

As mentioned in section 3, our target row decompression algorithm performs dictionary-based expansion. A compressed row consists of one or more 12-bit compressed symbols (tokens); a symbol may either represent a character of the uncompressed row or a pointer to a dictionary entry, which in turn may contain up to 7 characters of the uncompressed row. The decompression operation essentially involves decoding all the compressed symbols in a row and building the uncompressed row by *stitching* together character data from each of them.

Row decompression logic on FPGA is shown in Figure 5-2. During an initial setup phase, the expansion dictionary is downloaded from the host into the dictionary buffers on the FPGA. This is a one-time process; it need not be repeated for each job, unless the table and hence the dictionary changes. During the scan phase, as the database pages are streamed to the FPGA, rows are extracted and stored in the row buffer. This extra buffering is required for rate matching between the decompressor and the row extraction logic and to provide each decompressor instance dedicated access to its respective rows in parallel to the rest. At the output of the decompression logic is an uncompressed row buffer which stores the decompressed rows. It provides a variable-bytes write interface, from 1 to 7 bytes, to support variable output rates from the decompressor.

A given database page may contain compressed rows mixed with rows in raw form; the decompression logic thus works in two modes – decompression mode and pass-through mode. As a new row is fetched from the row buffer, the header parser determines if the row is compressed or raw. If raw, the row is simply passed along to the uncompressed row buffer, two bytes per cycle. Even though a raw row can be copied over faster, only two bytes are transferred per cycle to keep a common read interface from the row buffer for both raw data as well as compressed tokens. This reduces extra multiplexing at the output of the row buffer. Note that reading just 2 bytes a cycle does not cause any starvation of the downstream predicate evaluation logic since it consumes data at a rate of 1 byte per cycle.

For compressed rows, the tokenizer module fetches the compressed token from the row buffer, one token at a time, as requested by the controller FSM. For a character-type token, the 8-bit character data is decoded from the token and written into the uncompressed row buffer. For a dictionary token, the controller reads the 8-byte entry from the dictionary.

A dictionary entry can be *unpreceded* or *preceded* and is decoded appropriately by the dictionary data decoder. An *unpreceded* entry contains up to 7 bytes of data and a length field. It is a terminal entry, indicating that the current token has been fully decompressed. A *preceded* entry contains up to 5 bytes of data and its length. Additionally, it contains a pointer to the next *chaining* dictionary entry that must be decoded to continue decompressing the current token, and an offset that indicates the relative position of the data bytes from the current entry within the complete uncompressed data for the token. Decompression of a compressed token is continued until an unpreceded entry is found, at which point the next token is fetched from the row buffer.

Optimal implementation of the decompressor design on the FPGA requires the operations described above to be staged in pipelined fashion. This, however, presents a potential problem – since the algorithm is not purely feed-forward, a new token cannot be fetched until the previous one is completely decompressed. Similarly, a new dictionary entry cannot be read until the current one has been read and decoded. These lead to empty cycles or *bubbles* between consecutive token reads and dictionary reads. This is not a problem as long as the average output rate of the decompressor is higher than 1 byte per cycle, which is the rate at which the predicate evaluation logic consumes data. For data with a very high compression ratio, this rate is easily achieved. For data with a lower compression ratio, however, where each compressed token expands to a few bytes, the average rate may drop below one byte per cycle, leading to stalling of the predicate evaluation pipeline.

We address this issue by adding token prefetch logic in the tokenizer, wherein the 8 next tokens are prefetched and stored in a FIFO. With this approach, the next token is

ready for processing as soon as the current one is finished. When the entire row is fully decompressed, any outstanding tokens in the FIFO are flushed and a new set of tokens is prefetched from the next compressed row. Figure 5-3 shows the effect of adding the prefetch logic on the overall pipeline utilization, for a specific real-life workload. Adding the prefetcher results in a more than 50% reduction in row decompression time; for the 165 byte long rows, the average row decompression time is reduced from 190 cycles to 80 cycles, bringing the average decompressor output rate well above the required 1 byte/cycle. Actual savings and the average output rate, of course, are dependent on the compression ratio and the number of tokens to be analyzed to decompress a row of certain length. The worst case of compression arises where a token does not access the dictionary and simply generates a single byte. With prefetching, such tokens can be processed in a single cycle, thus maintaining a throughput of 1 byte/cycle.

Figure 5-3 Effect of prefetching on pipeline utilization

Note that similar optimization is not possible on the dictionary read side since the reads from the dictionary are not from sequential locations and depend on the value of the pointer. This, however, is a concern only for the dictionary tokens that generate three or fewer bytes of data. Such tokens would appear in data with a very low compression ratio and can potentially lead to sub-optimal utilization of the predicate evaluation pipelines.

5.2 Predicate Evaluation on FPGA

Once the rows are decompressed, they are sent to downstream predicate evaluation logic for filtering based on the query predicates. A row scanner is used to evaluate the database rows against the query (Figure 5-4). It consists of a chain of independent predicate evaluation units and a reduction network (RN). Each PE evaluates a single predicate on a particular column of the database row. The reduction network then reduces the outputs of the PEs, as per the query, to a 1-bit qualify signal.

To evaluate a query on the FPGA, each PE must perform a specific operation on specific bytes of the database row, and the reduction network must combine the outputs of the PEs in a certain manner. The operation to perform and the fields to evaluate are specific to a given query and change with the query. Generating a new FPGA hardware image for each query and reprogramming the FPGA is not viable since synthesis and place-and-route typically take hours to finish. Reprogramming the FPGA with a pre-generated FPGA image is plausible, though for queries that operate on relatively small amounts of data, the reconfiguration time may dominate the overall processing time. Moreover, even a slight change in the query would require regeneration of the FPGA image and reconfiguration of the FPGA.

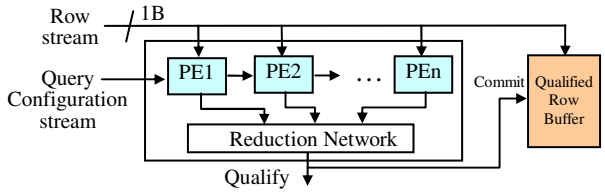

Figure 5-4 Row scanner for evaluating database rows

To address this issue, we designed the row scanner such that a given hardware image can be tailored to a variety of different queries. To that end, each PE is designed to perform 1 of 6 inequality operations; the actual operation to be performed is chosen during the query load time. Moreover, some of the PEs can also be disabled and excluded from participating in the query. Similarly, the reduction network is constructed as a binary tree of reduction units. Each reduction unit is a 2 to 1 reducer capable of performing one of 6 1-bit operations between the two inputs: AND, OR, NOT a, NOT b, PASS a, or PASS b.

During the query load phase, the configuration options are propagated down the PE chain and the reduction tree. PEs are configured using 5 options: (i) enable, to indicate whether the current PE is being used, (ii) predicate value, the constant against which the row field is compared, (iii) the inequality operation to be performed, (iv) offset of the first byte of the desired field within the row and (v) length of the field. A reduction unit has only one configuration option: the operation to perform between two predicates.

During the scan phase, database rows are streamed over the PEs, one byte per cycle. Streaming at a granularity of 1 byte is essential since fields in a row can be of varying length and may start at any byte position within the row. To reduce the latency of evaluating a row, the streaming bytes are broadcast to all the PEs as opposed to being rippled down the chain. Further, they are also *speculatively* written into the qualified row buffer; this write is later committed or invalidated based on whether the row qualifies. Since broadcasting requires large fan out, we use a register tree to feed all the PEs concurrently while keeping the fan out low.

As the row is streamed in, each PE captures the required bytes (using the start offset and length fields), evaluates its respective predicate and outputs a 1-bit match/mismatch signal. These funnel down the reduction network to generate the row qualify signal. For PEs disabled during the query load phase, the corresponding reduction unit either disables itself (if both the PEs feeding into it are disabled) or replaces the output from the disabled unit with the default pertaining to the function programmed into the reduction unit (e.g. 1 for an AND operation).

For an N-byte long row, it takes N + log2(n_{PE}) cycles to qualify the row, where n_{PE} is the number of PEs in the row

scanner. At that point, the row, if qualified, is committed into the qualified row buffer. The qualified rows from different row scanners are copied over to the output page buffer by the page formatting logic (see Figure 5-1), at a rate of 16 bytes per cycle. Formatted pages are sent to the host for further processing. Note that the FPGA returns the rows in uncompressed form, for direct consumption by the DBMS. This does not affect the processing rate on the FPGA since the filtering of the database rows typically results in lower bandwidth requirements on the return path.

6. SOFTWARE ENABLEMENT FOR FPGA ACCELRATION

Enabling FPGA acceleration for a DBMS is an end-to-end effort. One aspect of this effort is to restructure the data flow. Secondly, the capability of transforming a DBMS query into a format that the FPGA accelerator can interpret for dynamic self-re-customization is also critical for accelerating various workloads without the need for reconfiguring the FPGA.

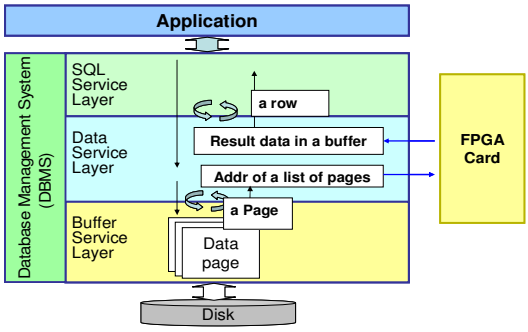

Figure 6-1 Block operation to enable FPGA acceleration

6.1 Block Level Data Operation

An important design aspect for integration performance is to reduce the *chattiness* during the interactions between the host and the accelerator. For this purpose, we introduce a block level data operation within the DBMS query processing engine. More specifically, a long running predicate evaluation query is divided into multiple jobs for an FPGA to process sequentially. Each job consists of a number of data pages as input for the FPGA to read, and an output buffer into which the FPGA writes the results. Both data transferring actions are initiated by the FPGA.

Figure 6-1 illustrates the changes made for enabling the block level data processing, as opposed to a traditional one page or one row at a time processing flow. For FPGA acceleration, a list of BP pages (addresses) is obtained by the Data Service Layer in DBMS and read by the FPGA. For output data from the FPGA, the DBMS pre-allocates a large enough buffer that is filled by the FPGA engine with its results. The data format in this output buffer conforms to the structure that is understood by DBMS processing engine for further downstream processing so additional data copy and formatting software can be avoided.

6.2 DBMS-FPGA Communication Protocol

Communication between the DBMS and the FPGA is achieved through a series of control blocks that are passed from the host to the FPGA. These carry the necessary information for describing the operations and data transfers.

6.2.1 DMA Addressing

Since the FPGA does not have direct addressability to host memory, all in-memory data blocks and control information blocks need to be transferred to the FPGA via DMA over PCIe. When constructing the communication protocol between software and the FPGA, the DMA addresses for the memory are used instead of the host addresses, as shown throughout this section.

6.2.2 Host Control Block

A query may be broken up into multiple jobs. A job is submitted to the FPGA via a host control block (HCB), which encapsulates the job information but is independent of the application logic. The HCB is interpreted by the service layer in the FPGA; it carries information such as whether the current HCB is the last job in the queue, the DMA address of the query control block (see 6.2.3), as well as updatable fields indicating an active job's status. A queue of HCBs is maintained which allows more jobs to be queued while a job is active on the FPGA. FPGA will continue to the next job in the queue, if one is available, when the current one is completed.

6.2.3 Query Control Block

A query control block (QCB) is a data structure that is used to tailor the FPGA application logic to a specific query. To address different invocation scenarios, we devise two formats of the QCBs. Both formats contain an address list of the input data pages to be processed and the address of the output buffer. The first format, shown as QCB1 in Figure 6-2, is used when the FPGA needs to interpret new query instructions and customize its internal logic. It thus also contains predicate function information and an address list of the 8 4K dictionary pages to be loaded. When the second format, shown as QCB2 in Figure 6-2, is used, the FPGA performs the same task as in the previous job on a different set of data. This saves the time needed to re-customize the FPGA logic and reload the dictionary.

To process more data pages per job than the limit imposed by a single QCB, a query control data continuation block (QCDCB), shown in Figure 6-2, enables more input pages to be chained to a job. The last entry in QCB's data list points to the next QCDCB when this is needed.

6.2.4 Query Mapping to FPGA Engines

To express predicate instruction to the FPGA, QCB1 contains a list of predicate function control blocks and a list of reduction control blocks. A predicate function control block, shown in Figure 6-3, is a 16 byte data structure that has information on the column, the comparison operator

and the constant to compare against the column value that personalizes each PE.

Reduction control blocks, logically structured as a tree for the RN, contain the Boolean operations information among the simple predicates in a complex predicate expression.

Figure 6-4 shows an example mapping of a predicate expression to the PEs and the reduction network.

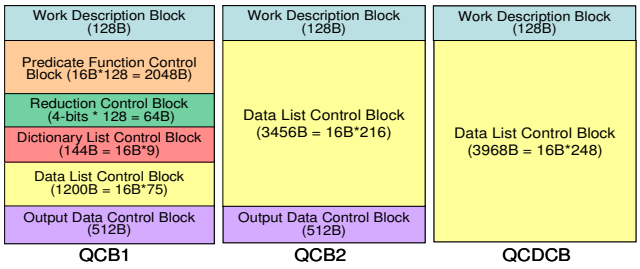

Figure 6-2 Different formats of the query control block

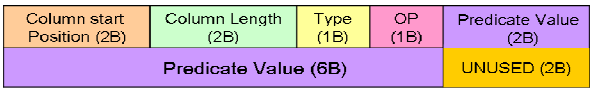

Figure 6-3 16B predicate function control block

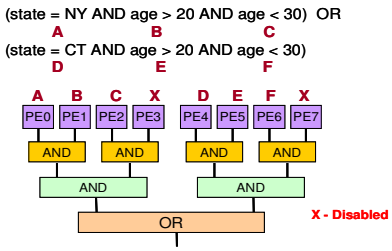

Figure 6-4 Mapping predicate expression to PE and RN

6.3 Transforming Query to QCBs

SQL provides rich syntaxes to express data relations. Predicate expression are not limited to just comparisons and Boolean operations. For example, expression *"WHERE state IN ('NY', 'CT') AND age between 20 AND 30"* tests set containment for field "state" and it can not be directly mapped to PEs and RUs as is. However, existing SQL parsing and transformation capabilities in the DBMS are capable of converting the predicate into the comparison and Boolean only expressions shown in Figure 6-4 and representing it in an internal format [17]. A realistic approach for query transformation integration is to inject the FPGA QCB build function into the query processing flow such that it can take advantage of the transformed expression for its input. It is then straightforward for the QCB build function to map the transformed expression to predicate function control blocks and correct topology in the reduction network.

7. EXPERIMENTS AND RESULTS

Our prototype is built upon a commercial DBMS running on a 3.8GHz multi-core super scalar system. Our target FPGA system is a PLDA XpressGX4LP card [18] with an Altera Stratix IV GX530 FPGA. The card communicates with the host system over a PCIe gen2 x8 interface.

Table 7-1 FPGA resource requirements

Tiles	PEs	Logic	ALUTs	Registers	Memory Bits
1	16	9%	24637 (6%)	27338 (6%)	2179840 (10%)
	32	16%	39893 (9%)	48949(12%)	
	64	30%	70,240 (17%)	91114 (21%)	
2	16	19%	47,877 (11%)	54,318 (13%)	4359680 (21%)
	32	33%	76,681 (18%)	97,441 (23%)	
	64	59%	136149 (32%)	181750 (43%)	
4	16	38%	95,795 (23%)	108269 (25%)	8,719,360 (41%)
	32	66%	152,687(36%)	194416 (46%)	

Table 7-1 lists the FPGA resource requirements for the different configurations of the database acceleration engine. This does not include the service layer, which occupies a fixed area of around 25% of the target FPGA. For a given number of tiles, the logic requirements increase proportionately with the number of PEs within a row scanner, while the memory requirements remain unchanged. Increasing the number of tiles, on the other hand, increases the number of row scanners, leading to higher logic as well as memory requirements. While the former is due to the increased number of PEs, the latter results from having more dictionary and data staging buffers. With 64 PEs per row scanner, the current FPGA can support up to 2 tiles; 4 tiles can be instantiated with fewer PEs per scanner.

We now present the results of evaluating our FPGA-based accelerator for database systems, using three metrics: CPU savings, speedup on offloaded computations, and overall end-to-end speedup. Additionally, we discuss the cost and relative merit of invoking the FPGA kernel. In our experimental workload, the data is derived from real customer database tables, and the query operations include decompression and predicate evaluation, which are two of the most CPU intensive functions. Both the baseline and hardware-accelerated versions of the code are single-threaded and run on a single processor core. All the measurements were taken with the data in buffer pools and do not include disk I/O time. Measurements were taken with both uncompressed and compressed rows; the uncompressed row lengths for the two workloads evaluated (customer 1 and customer 2 in Figure 7-2) are 165 bytes and 229 bytes, with the average compression space savings of 80% and 50% respectively.

CPU savings represent the CPU runtime reduction obtained by offloading the computations from the CPU to the FPGA. This is an important metric since it effectively represents freeing-up of the CPU resources for online transaction processing in real-time analytics systems. Figure 7-1 shows the savings obtained while processing 468K database pages of uncompressed data and 101K pages of compressed data from customer 1 workload, for a range of data qualification

ratios. Note that the reduced CPU time shown for the FPGA-assisted database system does not include the time spent on the FPGA to perform the actual computations.

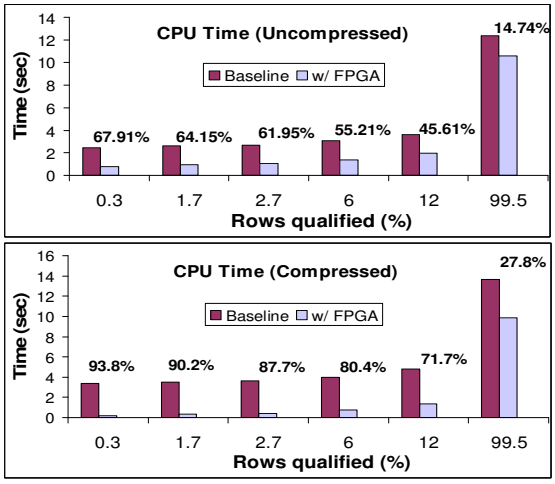

Figure 7-1 CPU savings obtained from FPGA offload

As shown, for compressed data, an 27.8% to 93.8% reduction in CPU time is obtained, while the reduction is between 14.7% and 67.9% for uncompressed data. This difference is expected; for uncompressed data, no decompression operation is involved, so the offload, and hence the savings is smaller. Moreover, for both compressed and uncompressed data, the savings are higher when a smaller fraction of rows qualify. This is due to the post-processing of the qualified data by the CPU, such as moving data to application buffer. When a larger fraction of rows qualify, there is more processing on the CPU and hence lower overall CPU savings. Overall, both compressed and uncompressed cases achieve significant saving of CPU resources, which in turn can be diverted to OLTP work.

Next we compare the raw power of the FPGA to that of the host processor in performing the same work, expressed in Figure 7-2 in terms of 'rows processed per second', for compressed and uncompressed data from two customer workloads. Also shown is the *effective* uncompressed data processing throughput achieved by the FPGA for each case. The FPGA design was instantiated with 4 scan tiles, for a total of 32 row scanners, and runs at 200 MHz. Note that the processing rate shown here measures just the offloaded computations and is thus independent of data characteristics such as qualification ratio. These measurements do, however, include the time for host-FPGA data transfers.

As shown, the FPGA achieves speedups of 10.7x and 6.7x on compressed data, whereas the speedups on uncompressed data are relatively modest. The reason is as follows: for uncompressed data, the overall performance is limited by the data transfer bandwidth; therefore, in this case, the FPGA engines are starved and not fully utilized. With uncompressed transfer bandwidth of 2.6GB/sec, the design achieves pipeline utilization of just 37%. In other words, the row scanners on the FPGA are idle for 63% of the time, waiting for data. Transferring compressed data, on the other hand, results in data expansion on the FPGA, thus leading to higher effective uncompressed bandwidth, thereby overcoming the effects of the PCIe limitation. For customer 2 workload, where the data is compressed down to 50%, the effective bandwidth is 4.8GB/sec and the FPGA engines achieve 75.7% of their peak performance. Here, all the available raw bandwidth has been utilized and the overall performance is still limited by PCIe. For customer 1 data, however, which is compressed down to 20%, the effective bandwidth is 6.8GB/sec with the actual transfer bandwidth of just 1.5GB/sec. In this case, the design achieves 98.5% pipeline utilization and the performance is limited by the available FPGA resources.

As can be seen, compression plays an important role; due to the high compression ratio, even a portion of the available bus bandwidth is sufficient to saturate the predicate evaluation engines on the FPGA. Higher PCIe utilization and further performance improvement can be achieved by instantiating more scan tiles on the FPGA. Data with a smaller compression ratio, alternatively, can utilize higher PCIe bus bandwidth to achieve high data processing rate. Note that the actual and effective bandwidths for uncompressed data are one and the same.

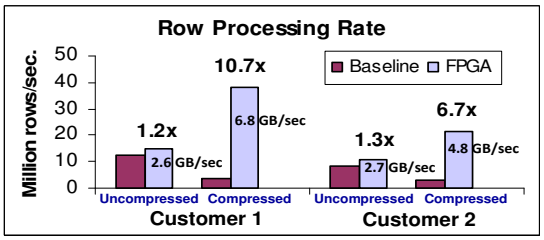

Figure 7-2 FPGA speedup over the baseline software

To evaluate the ultimate benefit of the proposed approach to an end customer, we measure the improvements on the end-to-end query response time. Figure 7-3 plots response times for compressed and uncompressed data. The dataset used is the same as that reported in Figure 7-1. The measured time includes data transfer time, the time for pre and post processing of the data on the host CPU and the time spent on the FPGA accelerator.

Again, the end-to-end response time is dependent on the filtering ratio, due to the post-processing of the qualified data. For compressed data, the graph shows similar trend as that for CPU savings – larger end-to-end speedups for lower qualification ratio, with speedups in the range of 1.2x to 6.2x for various filtering ratios. Uncompressed data, on the other hand, does not show any improvement in response time. This is mainly due to two reasons: first, the amount of processing that is offloaded is not enough to amortize the costs associated with invoking the FPGA routine and second, the computation-only speedup obtained from the FPGA is modest, as discussed earlier, due to the bus

bandwidth limitations. While the latter can be addressed by tuning the current PCIe DMA interface or using a higher bandwidth bus, addressing the former requires offloading more computations to the FPGA. These include some of the post processing operations performed by the CPU. Nonetheless, the fact that there is no regression in response time in the uncompressed case still makes offloading an effective improvement due to CPU savings. More importantly, the compressed data performance is particularly significant in that most enterprise customers do compress their large tables for storage savings.

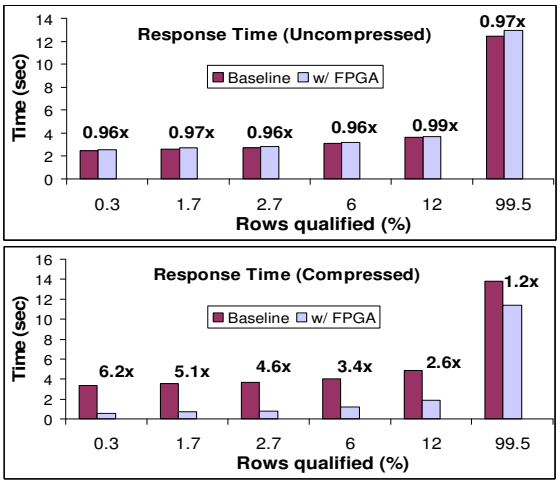

Figure 7-3 Query response time improvement

As is clear from the above discussions, offloading the decompression operation to the FPGA is crucial to obtaining overall performance improvement. Our FPGA design performs streaming, on-the-fly decompression of the compressed rows, thereby offsetting PCIe bus bandwidth limitations without incurring any performance penalties for large, throughput-centric analytics workloads.

Finally, we discuss the overhead of calling the FPGA routine. Since sending a query to the FPGA incurs some start-up cost, we performed experiments to find the minimum query data size needed to offset that cost and obtain an overall performance benefit. In terms of absolute time, the CPU cost for FPGA invocation is around 50 to 100 milliseconds on this processor. Based on our experiments, this cost gets offset for tables with 4000 to 10,000 uncompressed pages or 400 to 1,000 compressed pages. For smaller queries, invoking the FPGA routine does not pay off and the original software path should be used instead. A future work in deploying this acceleration technology to DBMS is to enhance the query optimizer such that it can select the appropriate execution path between software and the acceleration based on database statistics and the optimizer's analysis of the query.

8. CONCLUSIONS AND FUTURE WORK

We have presented an FPGA-accelerated database system for evaluating expensive analytics queries. The results show up to 94% savings of CPU resources and up to an order of magnitude speedup in the offloaded computations on the tested workload. We presented a general approach for an existing DBMS to seamlessly leverage the FPGA accelerator and validated with a prototype that integrates the accelerator into a commercial DBMS and achieves significant end-to-end performance improvements.

As part of our future work, we are investigating offload of other database computations to the FPGA to further improve the overall performance. Moreover, in our current work, we focused on the issue of CPU resources while running analytics queries in parallel with mission-critical transactional work. While CPU is the most critical resource in such environments, in some cases, I/O resources can also affect the overall performance. Addressing this also forms a part of our future work.

9. REFERENCES

[1] http://www.teradata.com/

[2] Krueger, J., et. al. "Fast Updates on Read Optimized Databases Using MultiCore CPUs" Proceedings of the VLDB Endowment, Vol. 5, No. 1, August 2012.

[3] Low, B., Ooi, B., and Wong, C., "Exploration on Scalability of Database Bulk Insertion with Multi-threading" Int. J. New Computer Architectures and Their Applications, 2011.

[4] Horikawa, T., "An Unexpected Scalability Bottleneck in a DBMS: A Hidden Pitfall in Implementing Mutual Exclusion", Parallel and Distributed Computing and Systems, 2011.

[5] Scofield, T. C., et. al., "XtremeData dbX: An FPGA-Based Data Warehouse Appliance" Computing in Science and Engineering, IEEE 2010.

[6] Johnson, R., Raman, V., Sidle, R., Swart, G. "Rowwise Parallel Predicate Evaluation" Proc. Int. Conf on VLDB'08.

[7] Zhou, J. and Ross, K. A., Implementing Database Operations Using SIMD Instructions", ACM SIGMOD, 2002, 145-156.

[8] Satish N., et. al., "Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort". ACM SIGMOD, 2010.

[9] Leung, K., Ercegovac, M., and Muntz, R. "Exploiting Reconfigurable FPGA for Parallel Query Processing in Computation Intensive Data Mining Applications" In UC MICRO Technical Report Feb. 1999.

[10] Mueller, R. and Teubner, J. "FPGA: What's in it for a Database?." In ACM SIGMOD, 2009, 999-1004.

[11] Mueller, R., Teubner, J. and Alonso, G., "Data processing on FPGAs", Proceedings of the VLDB Endowment, v.2 n.1, August 2009.

[12] Mueller, R., Teubner, J. and Alonso, G., "Glacier: a query-to-hardware compiler", In ACM SIGMOD'2010, 1159-1162.

[13] http://www.netezza.com/

[14] http://www-01.ibm.com/software/data/db2/zos/analytics-accelerator

[15] Ramakrishnan, R., and Gehrke, J. Database Management Systems. McGraw-Hill, 3rd edition.

[16] Iyer, B. and Wilhite, D. 1994. Data Compression Support in Databases. In Proceedings of Int. Conf on VLDB.

[17] Lorie, R.A., and Nilsson, J.F.. 1979. "An access specification language for a relational data base system", IBM J. Res. Dev.

[18] http://www.plda.com/prodetail.php?pid=184