FPGA based hardware implementation and parallel processing of database operations on streaming projections in C-Store (a column oriented database)

Sanjay Kulhari
University of California, Riverside

Abstract. Due to disk bandwidth limitation, CPU performance is limited. Applications that require reading and processing lots of data in memory do not show a significant improvement even on faster machines. Overcoming disk bandwidth limitation requires efficient caching strategies, algorithms minimizing data transfer to the CPU thus making efficient use of the bandwidth. Another way to overcome bandwidth limitation is to process streaming data on hardware for which FPGAs are well suited. In this report we will look at the query execution in C-Store (a column oriented database) that makes efficient use of disk bandwidth by sending only the desired data to CPU. The operations and data source that are part of query execution are discussed. Parallelism in the query execution is identified and the FPGA based hardware implementation of operations on streaming data is proposed.

Keywords: FPGA, C-Store, sort order, projection, materialization.

1. Introduction

There has been an increased need for efficient querying of vast amount of data to build decision support systems and knowledge based applications. Organizations need analytical processing of the vast amount of data related to the events/transactions/observations of the past. Analytical processing of data is required in many fields such as stock exchanges, medical and security services. Some of the queries that are run for stock exchange require many hours to complete. An effort is required to identify ways to efficiently execute complex queries on large amount of data. Recently, there has been a lot of emphasis on read optimized databases to support such applications. In general, read optimized databases store tables as columns or a set of columns in different data files and since the analysis is generally not based on all the columns, the CPU is fed with the data from desired columns and thus the bandwidth is properly utilized and processing speed is increased.

There are a number of papers published that show the benefits of storing the relational tables as columns instead of rows. Because of the increased speed of the CPU compared to the memory bandwidth, CPU has to wait for the data so to make full use of the processing power of CPU it would be appropriate to send only the desired data and make the full use of the bandwidth and to reduce the I/O demand. In a normal row-store database, a complete record has to be read even if a condition is checked on just one of the column, this causes unnecessary data to be read and sent to CPU.

The query engine architecture for column oriented database and row oriented database is shown in Figure 1. In case of column scan, columns are operated independently of other columns and the intersection of the resultant positions is performed to identify the final positions that satisfy the combination of conditions on all columns.

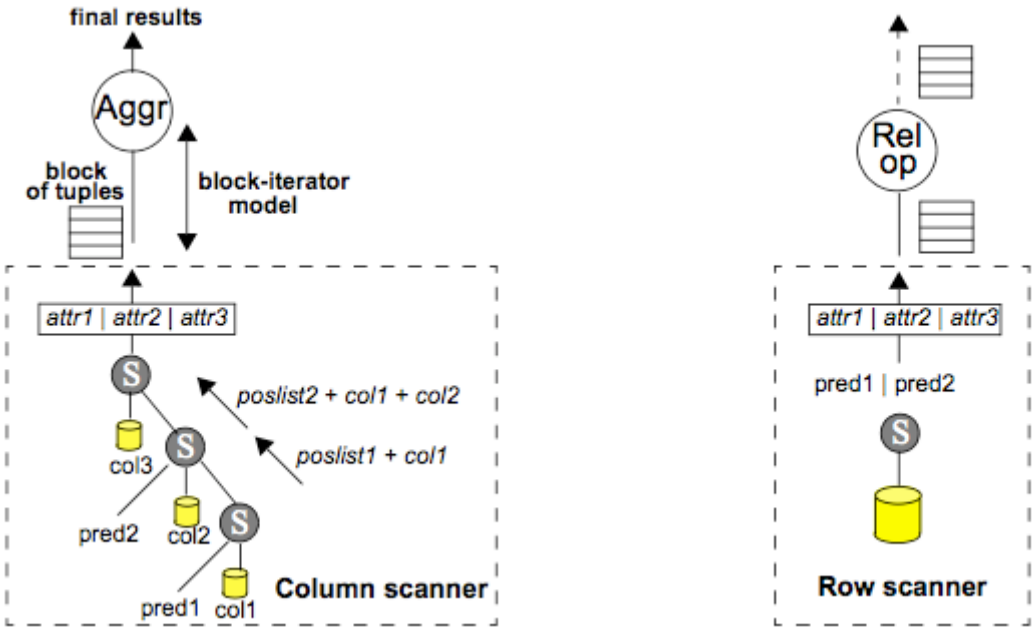

Figure 1: Query Engine Architecture [6]

Since we are interested in column oriented database C-Store, an example of working of a query (Q1) in column scanner based architecture is shown in Figure 2 assuming Col1 and Col2 has same sort order.

Query (Q1): Select col1, col2 from table where col1%2==0 && col2%3==0

One thing can be noted that although by storing the table as columns, disk bandwidth is properly utilized and CPU can work efficiently on data but because of the cardinality of the table and the number of columns involved in the query, the processing can still take a lot of time. In this report, query execution in C-Store (a column oriented database) is explained and different operations have been identified that can work in parallel and thus can give a better throughput. An FPGA based hardware implementation of the operations is proposed that work on streaming data from the projections.

Section 2 talks about the factors that give the motivation for parallel processing and hardware implementation. Also the factors that one needs to be aware of to get the system with the capabilities of efficient query execution on large databases are also presented in this section. In Section 3, query execution in C-Store is explained. Section 4 identifies the parallelism available during query execution and section 5 proposes an FPGA based implementation of the

operations. Section 6 talks about the expected performance improvement because of hardware implementation while section 7 discuss the conclusion.

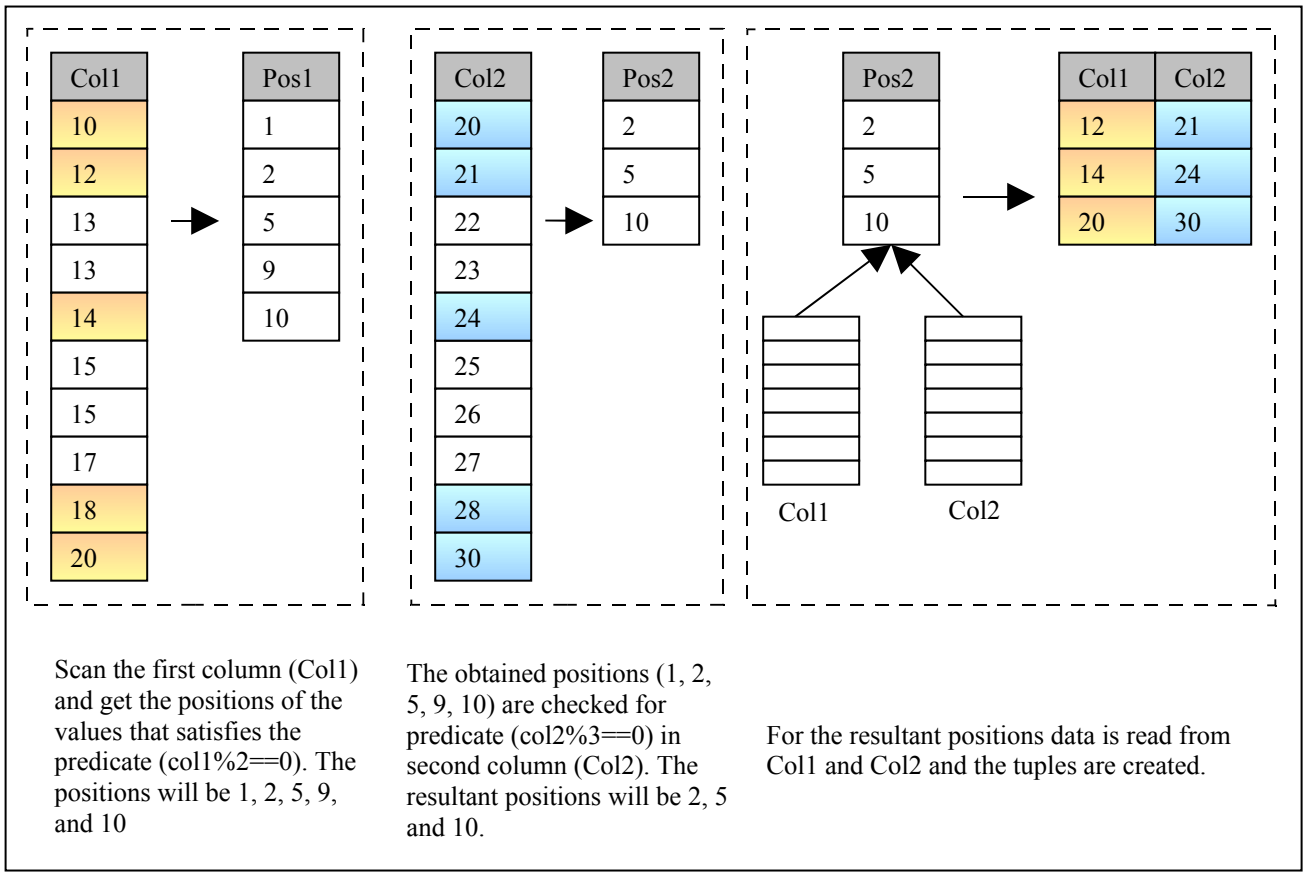

Figure 2: Query Execution in column scanner architecture
Query: Select col1, col2 from table where col1%2==0 && col2%3==0

2. Motivation and Challenges

As some of the research has shown that the column store databases have benefits over row store in read queries and also on queries involving aggregation [1]. But there are number of factors that can affect the performance of read optimized database [6], [2] such as number of predicates and number of returned columns. Time spent in query execution consists of time spent in decompression and time spent in joining columns based on join indexes. Joining of columns is a costly operation, so if there are many columns involved in the query the performance will degrade.

Although column stores provides efficient ways of compressing data but since the performance depends on amount of processing required decompressing the column values. In C-Store, decompression is postponed till the end but when the final tuples are created, data is decompressed at the matched positions and is returned. Factors affecting the query execution plan are the selectivity factor, number of predicates, columns involved in predicates and number of columns to be returned.

Decompression can prove to be costly when there are large numbers of columns to be joined at the end. Although C-store avoids the decompression till the very end of the query execution, but this is query dependent. There can be some queries where there is no other option than decompressing the column values and work with them and the joins over other columns can prove to be quite costly. Inserts and Deletes are also the concerns for column oriented databases. C-Store provides a unique way of read and write store that helps in minimizing the performance impact in updates. The decompression can be implemented in hardware and can operate on streaming column values.

There can be a large number of predicates involved in the query, which basically means we are looking at a large number of projections for positions that satisfies the predicates. Also there can be a large number of columns that are returned by the query that basically means we are taking the positions and scanning the projections for the desired values. So there is a scope of parallelization for the process of getting the positions based on predicates and also for the process of getting the real values (may be after decompression) based on positions.

The results obtained after process of getting the actual values (may be after decompression) based on positions have to be sent for the CPU processing to merge the results. When there is a sort operation in the query and after the FPGAs are configured to look for a particular position in the streaming data, the result has to be properly combined in the required order. Later we will see what operations can be implemented on FPGA and what factors we need to be aware of for correct computation.

3. Query execution in C-Store

In this section we are going to look at how queries are executed in C-Store. In the next section we will use our understanding of the execution process to identify the operations that requires lot of computation and that can operate on data in parallel and whose results can be combined together.

Projections and Join Index: C-Store stores data in projections, which are basically a column or a set of columns as independent files. Each projection can have its own sort order. Join index keeps a relation between the projections by storing an array of integers that indicates where the pointed column is. Join indexes are needed for projections with different sort orders. Projections are result of design decision and are created keeping in mind the type of queries that can be executed.

StudentID	FirstName	LastName	Year	GPA
860897362	Saurabh	Mehta	2005	3.9
860894635	Abhishek	Jain	2006	3.2
860354233	Srikanth	Alaparthy	2007	3.8
860364923	Gaurav	Chaudhary	2004	3.3
860354939	Sanjay	Kulhari	2008	4.0

Figure 3: Table: Student

Let us understand the execution of the following query on table *Student* shown in Figure 3.

Query: Select FirstName from Student where GPA > 3.5

Since projections identified during the design phase, suppose for the *Student* table projections are stored in the file system as shown below in Figure 4. Projections *P1*, *P2*, *P3* and *P4* contain single columns and are sorted by *StudentID* while projection P5 is sorted on GPA.

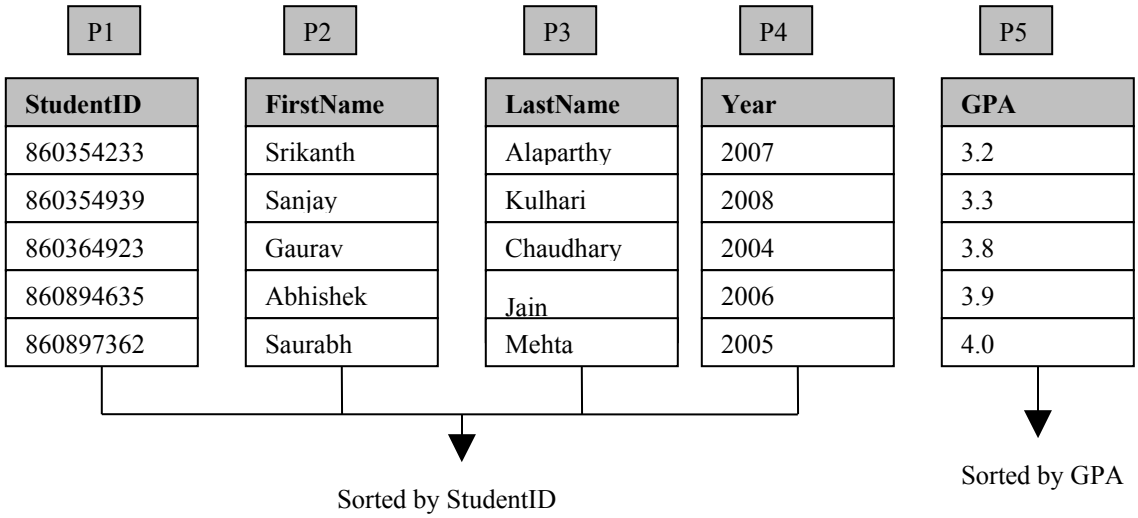

Figure 4: Projections for Student table

From the projection P5 on GPA, we can apply the predicate on GPA and obtained the results by tracking the B-tree of GPA, but for the final result of the query we need the FirstName.

Since projection P5 and P2 has different sort orders, there is a need of join index (JIX) between these two projections. The Middle column in Figure 5 shows the join index for these projections.

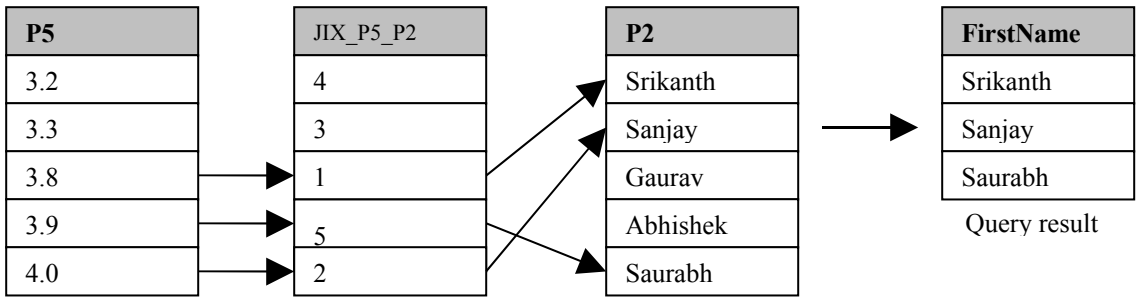

Figure 5: Getting values using join index (JIX_P5_P2)

The way join index is used to identify the corresponding values in another projection is also shown in Figure 5.

4. Parallel processing

In this section we are first going to look at two materialization techniques that decide when to stitch the column values. We are also going to look at different operations that are performed during the query execution and we will understand the phases during which these operations are performed also the data they operate on.

Understanding the working of operations and the data they work on will help us identify the parallelism in the query execution.

Consider the following query.

Query: Select Col1, Col2, Col3 from table where Col1%2==0 and Col2%3 == 0 and Col3%5 ==0

Early materialization: In case of early materialization the columns are stitched together at the very beginning as shown in Figure 6. Memory will contain the data from all the three columns and the predicate is applied row by row to each column's values. This requires keeping lots of data in the memory and processing lot of data.

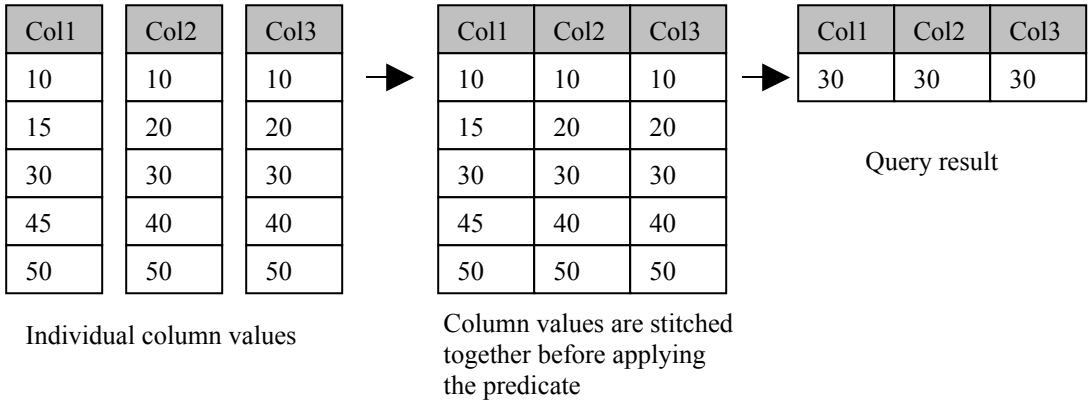

Figure 6: Early materialization

Late materialization: In case of late materialization, predicate is applied separately on the columns, the matching positions are identified and operation (AND) is performed on the positions to get the position that satisfy all the three constraints together. The column data is then re-accessed at those positions to get the actual values. These positions can be ranges, lists or bitmaps. An operation to decompress the actual values at these positions is required. Late materialization is shown in Figure 7. In this example the three columns can be scanned in parallel for 3rd position thus providing an option for parallel processing.

Late materialization requires just the final set of values from all columns to be stitched together which is less compared to early materialization and also not memory intensive. Columns can be stored in compressed form instead of actual values the operators have to work on the compressed columns and just give the positions so that at the end they can be stitched together.

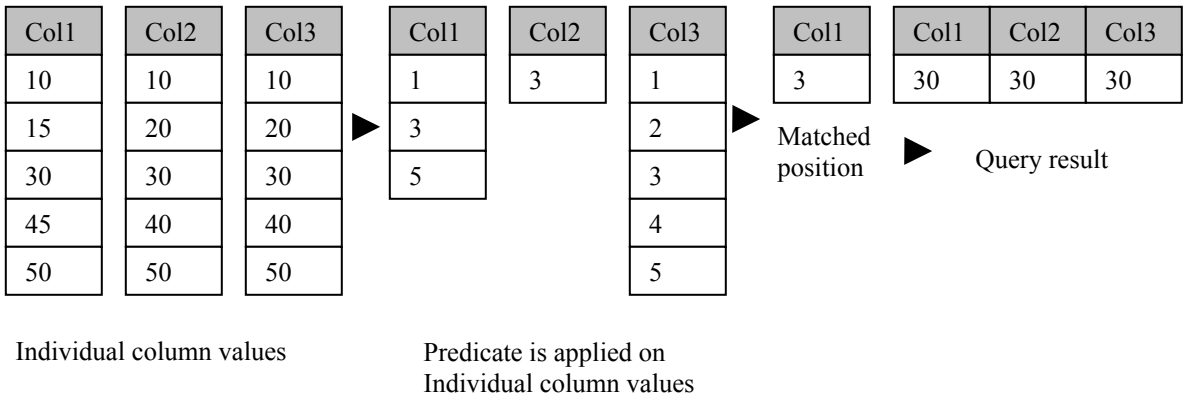

Figure 7: Late materialization

The following operations are part of query execution in late materialization technique.

1. ***DS1***: DS1 (Data Scan 1) is the operation involved in late materialization. It returns a list of positions that satisfies the constraint in a projection. DS1 operator can be applied in parallel to different projections to get different lists of matching positions.
2. ***AND***: AND operator takes the intersection of the positions obtained after applying DS1 operator on the projections.
3. ***DS3***: DS3 (Data scan 3) operator gets the final list of position that satisfies all the constraints and get the actual data present at those positions from the projections. If the projection had stored the data in compressed format, this operator also performs the decompression on the data.
4. ***MERGE***: MERGE operator gets the column values from DS3 and then merges them to get the logical tuple.

The following example helps to understand the operations that are involved in decompression, identifying positions and fetching the values at the positions. Consider the following query on *Student* table, but with projection on Year with different sort order.

Query: Select studentID, firstname, lastname from student where year > 2005 and GPA > 3.5

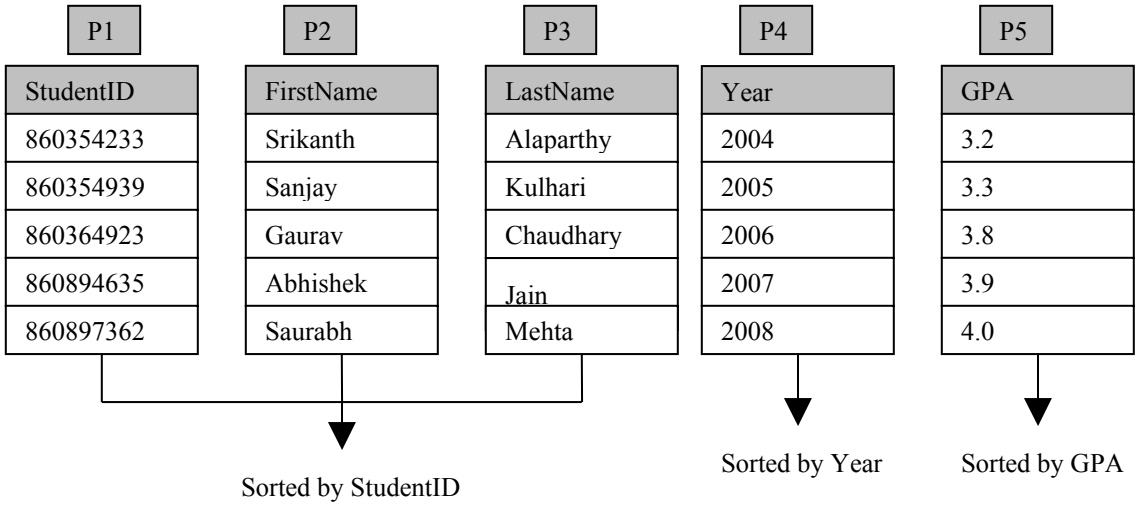

Figure 8: Projections for Student table

Since Year and GPA has different sort order, AND between the positions will be the positions that satisfy both the criteria. It will be done using the join index between P4 and P5. The resultant positions can be either from Year or GPA, which needs to be joined with projections on StudentID, FirstName and LastName.

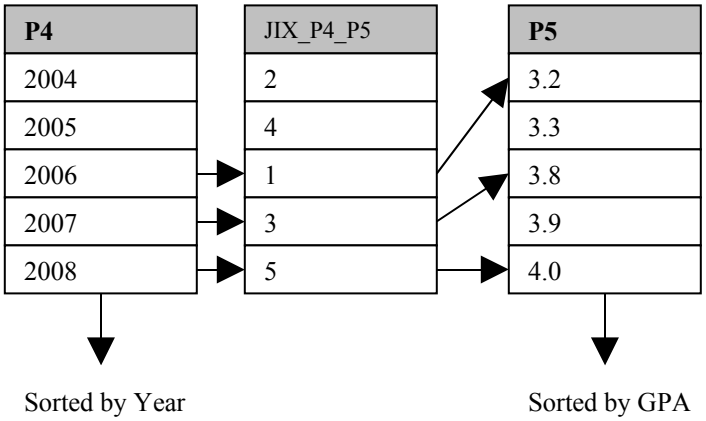

Figure 9: Getting P5 positions using join index (JIX_P4_P5)

The positions returned from Year are 3, 4 and5. Since they have different sort order we cannot simply apply predicate on 3, 4, 5 positions on GPA. These map to positions 1, 3, and 5 on GPA using Join Index. But the values at position 3 and 5 only satisfy GPA > 3.5 therefore AND operation when performed on positions coming from GPA and Year, positions 3, 5 are returned corresponding to projection P5 (GPA).

These positions are then used by DS3 to scan the projections on StudentID, FirstName, and LastName to return the values. Operation of DS3 and the result is shown in Figure 10.

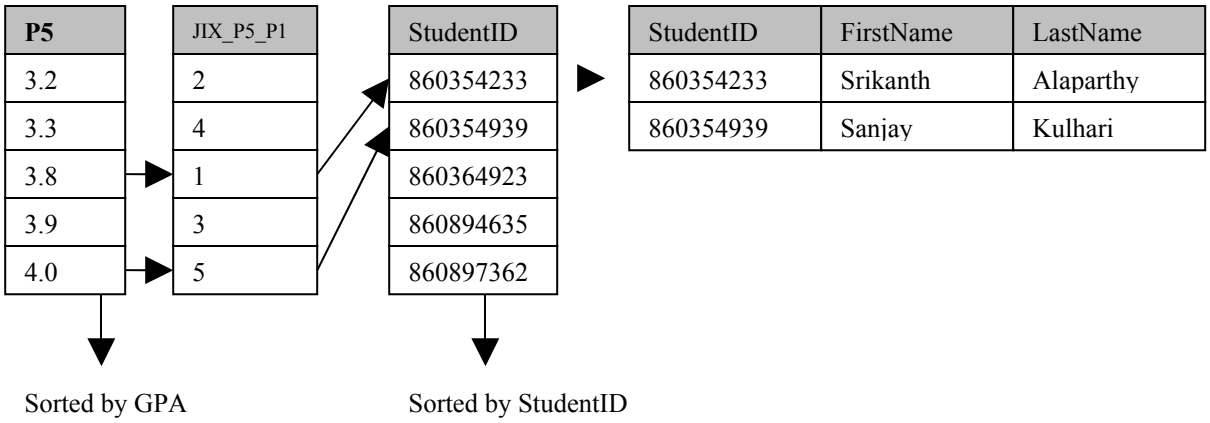

Figure 10: Getting values using join index (JIX_P5_P1)

The operations involved in getting the query results for the previous query are shown in Figure 11. Projections that are worked on by those operators is also shown. We can see that operator DS1 can be applied in parallel on projection P4 (Year) and projection P5 (GPA). The AND operator takes the positions from the output of DS1 and gets the positions that satisfies both the constraints.

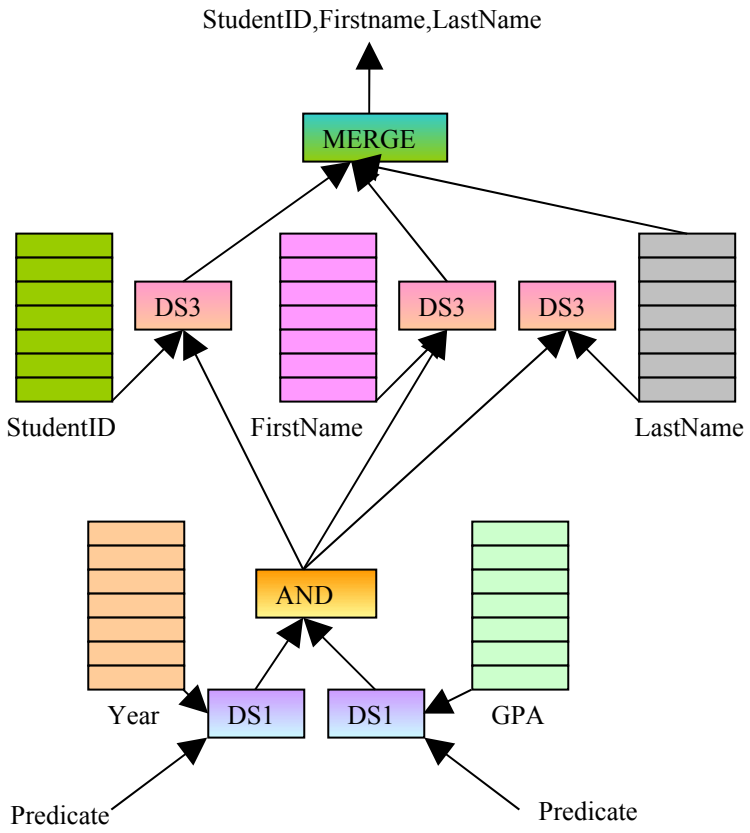

Figure 11: Query execution in late materialization

We see that operator DS3 can also be applied in parallel to three projections that have the columns that are part of query output. DS3 operator may also need to decompress the data and hardware based implementation on FPGAs can be used for that. The data from the projections can be streamed from the data block and can be uncompressed when DS3 finds the data at the required position. After DS3 operation is applied in parallel to different data sources the results has to be combined to get the record.

5. Hardware implementation

After understanding the working of operators in query execution and from Figure 11 we see that DS3 operator simply takes the position values for a particular projection that are going to be the part of the query output. We can also see that the projection data can be streamed to these operators. FPGAs are well suited to implement operations to be performed on streaming data.

FPGA streaming architectures are generally organized as systolic structures in which neighbors communicate directly through dedicated FIFOs. As a result, internal communication bandwidth can be very high while minimizing contention between elements [3]. Thus FPGA implementation seems to be suitable for these operations.

Representation of DS3 operation and streaming data is shown in Figure 12.

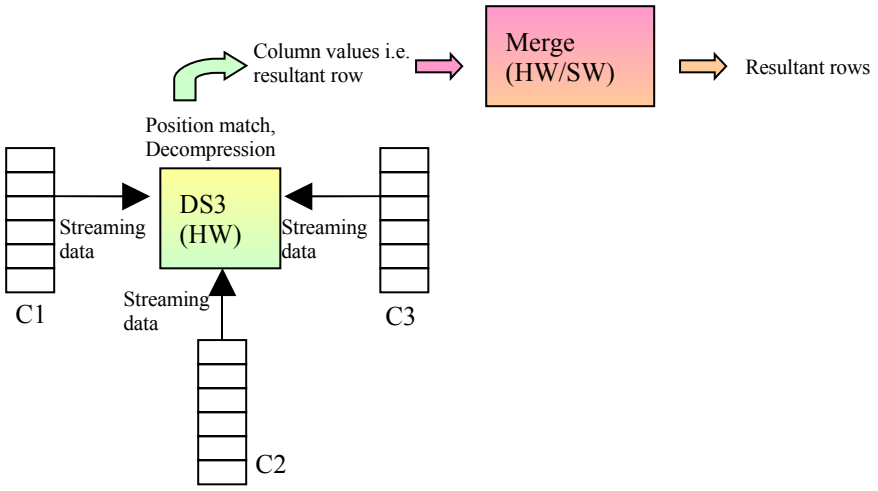

Figure 12: Operations to be performed on FPGA

DS1 and AND operation are required to check the conditions but DS3 just identifies the data based on position and decompress if required and MERGE operator takes a resultant rows from DS3 operators and stitch them together to get the final set of resultant roes. DS3 can take streaming position values and streaming column data, decompress the column data if required. MERGE can take streaming data consisting of the column values coming out of DS3 and merge them in the same order to get the output columns.

The schematic of FPGA board is shown in Figure 13. It shows five Data-Scan 3 (DS3) operators implemented on hardware and each configured for a position to scan for. These DS3 operators operate on streaming data and get the values at the position these are configured with. Since decompression can also be required in some cases, this logic unit can be programmed for decompression. The decompression technique can be different for different projections and thus logic unit should be dynamically reconfigured to decompress the data encoded with different compression techniques.

CPU processing is required to identify the projections that need to be streamed, so the CPU will simply access the required projection and the data will be stream without any extra processing. When different DS3 operators have got the values at the specified positions, they need to be stitched to form tuples. This can be either done in hardware or software. There can be some queries that expect the output in some sort order so the results from DS3 operators have to be stitched in order. The values of the positions that DS3 logic unit has to configured with can be stored on FPGA memory, so that when the task is done for the position, logic unit can be reconfigured and it starts looking for values at next position.

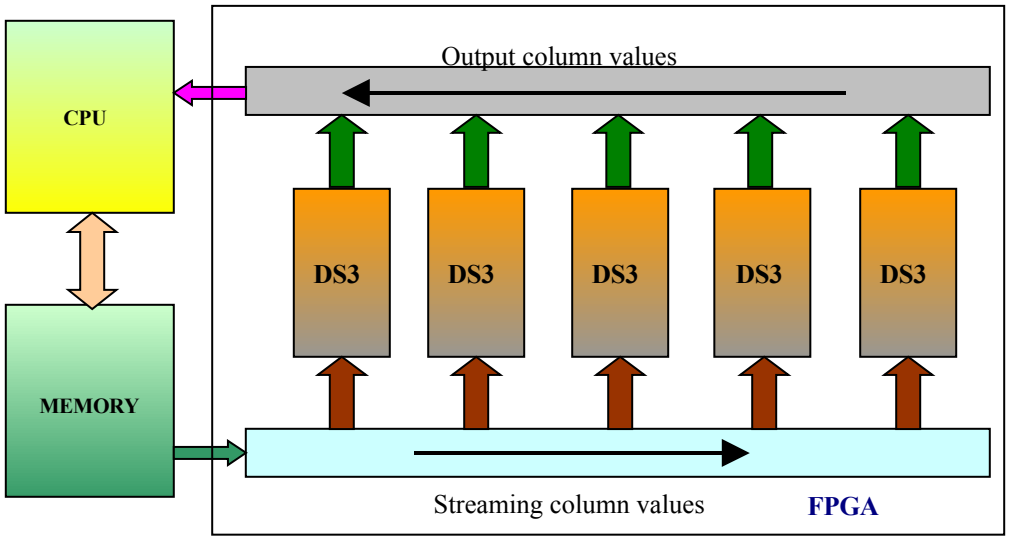

Figure 13: FPGA board (DS3 on HW)

This way the logic units on FPGA can be configured as Data-Scan 3 operator that takes a list of positions and read the entire column for the values at those positions. If the value is in a compressed format, it can be decompressed by the DS3 operator. So there can be *n*-number of DS1 units working in parallel for different positions, later they can be dynamically reconfigured to scan for another set of positions. Logic unit can be reconfigured to work for other position by sending the bitstreams or the list of positions can be stored locally to the FPGA.

6. Expected performance improvement

Performance improvement will be obtained because the operator DS3 can work on the streaming data. In traditional approach a block from the memory is read and the positions are matched. Since there can be millions of records, the benefit of sending single column values to the CPU may not prove to be sufficient. Suppose for a column with million values, the positions matched were present in last few blocks that were read, the memory bandwidth was used to send irrelevant data. In case of streaming data, this limitation is not there and the logic unit is programmed to look for some k^{th} position.

Since the execution time of query depends upon various factors – number of predicates, type of predicates, output columns and time spent on decompressing and joining data, [2].
Consider the following scenario - If there are 10 columns in the predicates, 10 columns as output of the query and the data is to be extracted at 100 positions from these 10 columns. The performance will be enhanced by at least a factor of 10 if 10 DS3 logic units are working in parallel and that too on streaming data.
Let the number of columns that have a predicate be *p,* cardinality of the table be *n,* number of DS3 logic on FPGA be *l,* number of columns to be returned are *r,* and percentage of decompression involved be *d.* Performance improvement will be proportional to all these factors. That is, more the amount of work more will be the gain. If the percentage of decompression (*p*) is quite high, the decompression performed on hardware and that too on the

streaming data will be very efficient. Also if the number of columns to be returned is quite high, the hardware implementation can have multiple logic units to look for positions in the column data at the same time.

If DS1 and AND operators are also implemented on hardware, performance improvement will be very significant. Parallel DS1 operations will be associated with column scans in parallel for the columns that have conditions on them, which returns the positions for those projects that meet the select criteria. If the number of columns that have predicate is quite high the gain will be significant.

7. Conclusion

The queries shown in the report had very few predicates and a few columns to be returned. In real analytical applications, the queries can have large number of predicates, many column to be returned and high cardinality of the table. In such cases parallel processing of columns on hardware will give a good performance enhancement. The operations such as DS1, AND, DS3, MERGE and decompression can be implemented on FPGAs. The example was based on late materialization where the DS3 operator gets the final values after scanning the columns for particular positions. This gives a good parallelism in the system. In case of early materialization both position and values are obtained at the same time. In such a case lot of data has to be kept in the memory for processing, but not required for parallel implementation on streaming data. Early materialization in column store can also get benefited from the parallel processing on hardware. The performance improvement is expected to be huge when operating on very complex queries with compressed column data on a high cardinality table.

References

[1] Stonebraker, M., et al. "C-Store: A Column-Oriented DBMS." In *VLDB*, 2005.
[2] Holloway, A.L., DeWitt, D.J. "Read-Optimized Databases, In Depth" *In VLDB, 2008.*
[3] Neuendorffer, S., Vissers, K. Xilinx Research Labs, "Streaming Systems in FPGAs", *In Springer-Verlag 2008.*
[4] Koch, D., Beckhoff, C., Teich, J., Bitstream Decompression for High Speed FPGA Configuration from Slow Memories, *In IEEE 2007*
[5] Abadi, D. J., Myers, D.S., DeWitt, D.J., Madden, S.R. "Materialization Strategies in a Column-Oriented DBMS." In *ICDE*, 2007.
[6] Harizopoulos, S., Liang, V., Abadi, D., and Madden, S. "Performance Tradeoffs in Read-Optimized Databases." In *VLDB*, 2006.
[7] Abadi, D. J., Madden, S. R., Ferreira, M. C. "Integrating Compression and Execution in Column-Oriented Database Systems." In *SIGMOD*, 2006.
[8] Abadi, D.J., Madeen, S. R., Hachem, N. "Column-Stores vs.Row-Stores: How Different Are They Really?" In *SIGMOD*,2008.
[9] Halverson, A., Beckmann,J.L., Naughton, J.F., DeWitt, D.J. "A Comparison of C-Store and Row-Store in a Common Framework" *In VLDB, 2006.*