Column Storage for FPGA-accelerated Data Analytics

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 74

Systems Group, Department of Computer Science, ETH Zurich

Column Storage for FPGA-accelerated Data Analytics

by
David Sidler

Supervised by
Louis Woods
Prof. Gustavo Alonso

October 2012–April 2013

inf | Informatik
Computer Science

Abstract

Data appliances became very popular in recent years. To decrease the network traffic between processing nodes and the disk, some modern systems use smart storage engines to off-load filtering query operators, such as selection or projection. This reduces the amount of data that has to be transferred from the storage engine to the processing nodes.
Ibex is an intelligent storage engine that is able to off-load query operators to a FPGA. This work presents a column storage implementation on a FPGA which replaces the existing row storage in Ibex. Memory on a FPGA is limited therefore it is only possible to load small data chunks from the disk (e.g. 4 KB). Since we are reading multiple columns which are stored at different locations on the disk, we have to deal with random access. Using a SSD and native command queuing (NCQ) it is possible to achieve high throughputs despite reading small data blocks. NCQ returns the data out of order, therefore a complex buffering system was implemented to reorder the data on the FPGA. We present an economical hardware design which is able to adapt to the number of columns processed, thereby optimizing hardware utilization and improving performance. Additionally we implemented two simple statistical operators which are applied to column-shaped data passing the FPGA. Due to the parallelism in FPGAs no additional latency is added to the data processing.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Related Work . . . 2
  - 1.3.1 Query Processing Techniques for Solid State Drives . . . 2
  - 1.3.2 Database Analytics Acceleration using FPGAs . . . 3
- 1.4 Thesis outline . . . 3

2 Fundamentals **5**
- 2.1 FPGA . . . 5
  - 2.1.1 FPGA Internals . . . 5
  - 2.1.2 FPGA Programming . . . 7
- 2.2 Groundhog . . . 7
- 2.3 SIRC . . . 7
- 2.4 Ibex . . . 8
- 2.5 SSD . . . 9
  - 2.5.1 NCQ . . . 10

3 Column Storage on the FPGA **11**
- 3.1 Data layout on disk . . . 11
- 3.2 Implementation on the FPGA . . . 12
  - 3.2.1 Buffering System . . . 12
- 3.3 Architecture Overview . . . 14
  - 3.3.1 Challenges on the Hardware . . . 17
- 3.4 Loader module . . . 18
- 3.5 Reordering structure . . . 18
- 3.6 Software . . . 20
  - 3.6.1 Table Manager . . . 20

4 Data Analytics on the FPGA **23**
- 4.1 Implementation on the FPGA . . . 23
  - 4.1.1 Multiplication pipelining . . . 25
  - 4.1.2 Software . . . 25
- 4.2 Formulas . . . 25

5 Evaluation **27**
- 5.1 Experimental Setup . . . 27
- 5.2 SSD performance . . . 27
- 5.3 Queue Depth . . . 28
- 5.4 Evaluation Column Storage . . . 30
  - 5.4.1 Buffer size and Double buffering . . . 30
  - 5.4.2 Buffer number . . . 31
  - 5.4.3 Full Table Scan . . . 32
  - 5.4.4 Projection . . . 33

5.4.5 Resource consumption 34
5.5 Evaluation Statistics module 35
5.5.1 Performance 35
5.5.2 Resource consumption 35

6 Conclusion **37**
6.1 Thesis contribution 37
6.2 Future work 37
6.2.1 Scalability 37
6.2.2 Compression 38
6.2.3 Data Analytics 38

Acronyms **39**

1 Introduction

Since the early 2000s Data warehouse appliance have become more and more popular. IBM started this trend with the introduction of Netezza [2], other vendors followed and nowadays multiple data appliance systems are on the market. Because the complete system is built by the same vendor, the software is heavily tailored to the hardware. This improves performance, scalability and allows more effective tuning of the system to certain workloads. Data appliances became popular thanks to their lower setup and maintenance costs, additionally they simplified and speed up business analytics.
Usually the data appliance takes care of the data layout and creates indexes and makes other optimizations when loading the data. On Netezza indexes or any tuning is not even required, to accelerate query processing it makes use of a hybrid system composed of multi-core CPUs and FPGAs.
The Systems Group at ETH Zurich is developing its own data appliance, called Swissbox [4]. One component of Swissbox is Ibex [7] an FPGA-accelerated intelligent storage engine which is part of a hybrid FPGA/CPU database system. In this work we implement a column storage that replaces the existing row storage [8] in Ibex. Column storages are widely used for several advantages, such as early projection, simpler compression, late materialization, block iteration. We will mainly leverage the fact that by reading only the necessary columns, the projection is pushed down to the data, thereby the effective transfer rate can be higher than in a row storage which has to read all attributes.
A new trend for Data warehouse appliances are *in-database* analytics which allow to run analytics and data mining on the data without extracting it from the system. Obviously reducing I/O by bringing the operation closer to the data leads to a significant performance gain. In the second part of this work we take a small step into this direction by implementing two simple statistical operators on the FPGA. Thanks to the parallelism on the FPGA no additional latency is added when analyzing the by-passing data.

1.1 Motivation

Like most common database systems the performance of Ibex is I/O bound. One way to improve disk I/O is the introduction of a column storage. In a column storage only the necessary columns are fetched from the disk in contrast to the row storage where every time the complete row is read. Only fetching the necessary columns means the projection is pushed down to the data. Therefore less data is transferred to the FPGA and later to the host system, the effective transfer rate is higher and the load on the FPGA and CPU is reduced.
Ibex is generally row based and can apply selection and group-by on materialized rows, we still want to make use of this existing functionality. So it is crucial that we materialize the rows in the FPGA early on. On the other hand there are operations like compression or statistical evaluation that are easier applied to column-shaped data. Inserting such operators before the materialization is

possible. In fact the Statistics module is placed in way that it can operate on the column-wise data stream.
Because the amount of memory on the FPGA is very limited we are not able to read big chunks from each column and then materialize them into batch of rows. Instead only small chunks (e.g. 4 KB) can be loaded, stored and processed. Loading small chunks from the disk reduces the throughput. To be competitive versus the row storage which is able to read huge chunks of data sequentially, we will make use of native command queuing (NCQ) combined with a sophisticated buffering system on the FPGA. To illustrate the performance challenge, sequential read with a block size of 4 MB is up to 270 MB/s, random 4 KB reads achieves only about 35 MB/s when using NCQ it rises up to 200 MB/s.

1.2 Problem Statement

This thesis has two main goals. First, the development of a prototype implementation of a FPGA-accelerated column storage which is part of Ibex, a hybrid FPGA/CPU database system. Second, hardware-based evaluation of statistical functions on the column-shaped data.
The main objective of the columns storage is to push the projection in Ibex down to the data, thereby reducing the amount of data that is read from the disk and has to be processed. The major challenge was to develop a complex buffering system that hides the latency of the disk and allows retrieval of multiple columns simultaneously. When implementing such a complex system one has to take the FPGA architecture into account otherwise the Design Tools will not be able to map the implementation into hardware. Further, the buffering system was made adaptable such that high throughput can be achieved independent of the number of columns projected. A trade-off had to be found between adaptability and complexity. For optimal integration with Ibex the columns have to be materialized early into compact rows which are fed into other Ibex modules.
The main goal for the hardware-based statistical operators is to add as little overhead as possible to the existing data processing. This was a two step process. Finding operators that can be implemented on a FPGA and fulfill this property. And exploiting the parallelism of the FPGA when implementing these operators such that no additional latency occurs when evaluating the data passing through the FPGA.

1.3 Related Work

1.3.1 Query Processing Techniques for Solid State Drives

Tsirogiannis, Harizopoulos et al. [10] have examined how to optimize query processing for SSDs. Generally database systems are heavily optimized for sequential access and can not benefit from the much faster random access in SSDs. The database pages of their system use a PAX-layout. To reduce the amount

of data read to answer a query, they developed two operators which take advantage of the PAX-layout by reading only the minipages that are necessary. Specifically they implemented *FlashScan*, a scan operator, and *FlashJoin*, a join operator.
FlashScan reduces the amount of data to read by using two methods. First it only reads the minipages of the attributes that are projected or have to be evaluated. Second it first reads the selection attributes and evaluates them proactively, only if the predicate is fulfilled the other attributes and their corresponding minipages are read as well. This technique increases performance especially for low projectivity or selectivity.
FlashJoin executes the join in two passes. In the first pass it only reads the join attributes and creates a temporary join-index. Other projected attributes are fetched in another pass as late as possible. With this technique I/0 and memory usage can be decreased.
Using random access is only beneficial when it is possible to reduce the amount of data that is read, such that it compensates for the lower speed.

1.3.2 Database Analytics Acceleration using FPGAs

Sukhwani et al. [9] also implemented an FPGA-accelerated database system. The main difference to Ibex is that the FPGA acts like a real co-processor and is connected ove PCIe to the host. It reads and writes the data from/to the main memory of the host system. While Ibex is placed in the data path and is used whenever possible, this system has to decide depending on the query size or complexity if it is worth to off-load query processing.
Additionally to predicate evaluation their system is also able to decompress data. This is very beneficial for two reasons. First it increases the effective transfer rate from the host to the FPGA and second the decompression does not compete with the query processing on the CPU.

1.4 Thesis outline

The thesis is structured in the following way:

1. Section 2 introduces some fundamental components which built the basis for our work.

2. Section 3 explains extensively the implementation of the column storage on the FPGA and in software.

3. Section 4 introduces statistical operators that are suitable for FPGAs and shows how two of them are implemented on actual hardware.

4. Section 5 evalutes the performance and resource consumption of the Column storage module and the Statistics module. The column storage performance is evaluated against the row storage and the performance impact of the Statistics module is determined.

5. Section 6 concludes our work and highlights our contributions. It also discusses possible future works on the topic.

2 Fundamentals

In this section we are introducing some fundamental parts which built the basis for this work.

2.1 FPGA

Field-programmable gate array (FPGA) is an integrated circuit that can be reprogrammed by the costumer. Originally FPGAs were used as *glue logic* between off-the-shelf integrated circuits. Over time their speed and capabilities increased and more and more functionality was implemented using FPGAs. Nowadays they are complete System on a Chip (SoC) and are used in many different applications. FPGAs are usually clocked much lower than CPUs (e.g. 100-500 Mhz) which makes them very energy efficient.

2.1.1 FPGA Internals

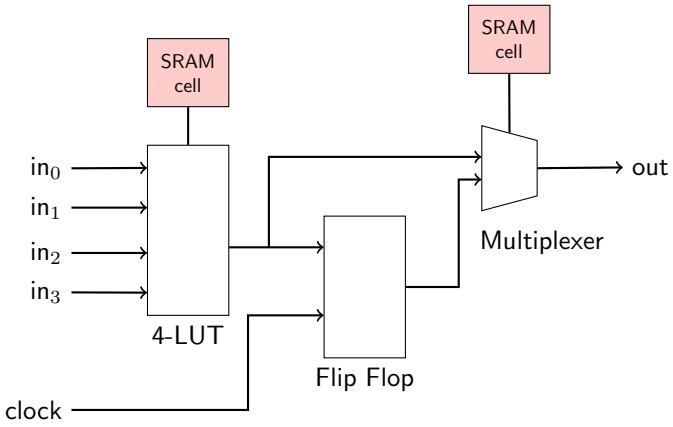

Figure 1: Elementary logic unit

Logic A FPGA mainly consists of an array of Configurable Logic Blocks (CLBs), a grid of routing channels and some I/O blocks along the boundary of the chip. In contrast to specialized hardware where actual hardware gates (e.g. AND, OR, XOR) are wired together, on a FPGA these gates have to be simulated by programmable CLBs. A CLB contains multiple *elementary logic units*, one such unit can be seen in Figure 1. It consists of a 4-input LUT, a D-type flip-flop and a multiplexer. A single *elementary logic unit* is essentially a 4-input/1-output function that is reprogrammable via the SRAM cell of the LUT. When the FPGA is programmed a truth table is stored into the SRAM

cell. During runtime the 4 binary input values define the look up value in the truth table which equals to the output of the function. The flip-flop stores the output value of the function and a multiplexer which is also programmed via a SRAM cell connects the flip-flop value with the output value. To support more complex functions multiple elementary units can be chained together.

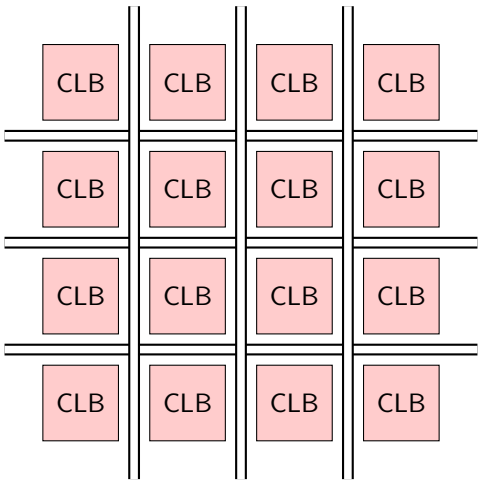

Figure 2: Routing on the FPGA

Routing Chaining neighboring logic units together to build a more complex circuit is done over some direct wires. But to build complete systems on a FPGA, it must be possible to connect components that are distributed over the whole chip area. Therefore FPGAs have a more flexible communication mechanism which allows to connect CLBs. To simplify routing, CLBs are arranged in a two-dimensional array. As can be seen in Figure 2, a grid of bundled wires is located in-between the CLBs, these wires are called *Interconnect*. The intersection points of the Interconnects can be regprogrammed, such that two CLBs can be connected over a few intersection hops.
Clock signals and other special signals (e.g. reset) are routed over a dedicated routing network covering the complete CLB array.

Additional components Vendors fit their FPGAs with dedicated components to increase the functionality. Most FPGAs are equipped with block RAM (BRAM). The BRAMs are placed in-between CLBs and allow to store a few kilobytes of data which would be very resource-intensive when done with LUTs. The Virtex 5 model, used in this work, has 148 BRAMs and each one can hold

36 KB of data.
To compute complex mathematical operations faster, dedicated Digital signal processing (DSP) slices are available. In our work DSP slices are used to multiply integer values.

2.1.2 FPGA Programming

FPGAs are usually programmed in a hardware description language, like Verilog or VHDL. FPGA Design Tools first synthesize the code into hardware components, then these components are placed and routed according to the targeted FPGA. At the end a bitstream file is generated which can be loaded into the FPGA to program it.
FPGAs can be used for Reconfigurable computing. The idea is that during runtime the FPGA is reprogrammed and thereby can adapt to different workloads. To make reprogramming reasonably fast the different bitstream files have to be generated before hand and are just loaded into the FPGA during runtime.
FPGA development consumes much more time than software development which is probably the biggest drawback.

2.2 Groundhog

Groundhog [11] is an open-source SATA *host bus adapter* (HBA) for FPGAs. It allows direct disk access from the FPGA over SATA-I or SATA-II. Additionally to the basic read and write commands it also supports the use of Native command queuing (NCQ) which is explained in Section 2.5.

2.3 SIRC

The simple interface for reconfigurable computing (SIRC) [6] is used to communicate over Ethernet between the FPGA and the host PC. SIRC is open source and provides a communication and synchronization API. The communication is based on a master/slave model, where the PC acts as the master and the FPGA as the slave. This means the FPGA is in an idle state until it gets a signal from the PC to start execution. The FPGA executes while the PC is waiting for the execution to end. When the execution on the FPGA is done, it sends a signal to the PC and goes back into the idle state. The PC is now again in control.
The SIRC is based on four main components, the control unit, just explained, parameter registers, an input and output buffer, as illustrated in Figure 3. As the name implies execution parameters are sent over the parameter registers while data is sent and received over the two buffers. Unfortunately the master/slave design means that only one of the parties is in execution mode at any time and is able to access these buffers. Without concurrent access from the CPU and the FPGA to the two data buffers the data to transfer has either to be limited to the buffer size or must be transferred over multiple control switches

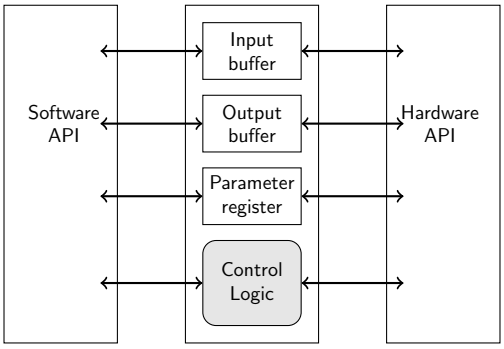

Figure 3: Layout of the SIRC module

between the two parties. This limits the practicability of our implementation considerably, future version of SIRC should be able to stream the data back to the CPU and overcome this limitation.
On the software side the SIRC interface is written in C/C++.

2.4 Ibex

Ibex [7] is an intelligent FPGA-accelerated storage engine. It is developed as a pluggable storage engine for MySQL [1]. In contrast to other hybrid systems the FPGA is not just a co-processor on the side, but is actually inserted into the data path between the disk and the CPU, as illustrated in Figure 4. The goal is do to as many SQL operations on the FPGA as possible and thereby decreasing the data that has to be transferred from the FPGA to the host and also reducing the load on the host CPU. Currently Ibex can push *selection*, *aggregation*, *projection* and *group-by* down to the FPGA. More complex operations are still executed on the host.
Since MySQL is row based and fetches the tuple from the storage engines with

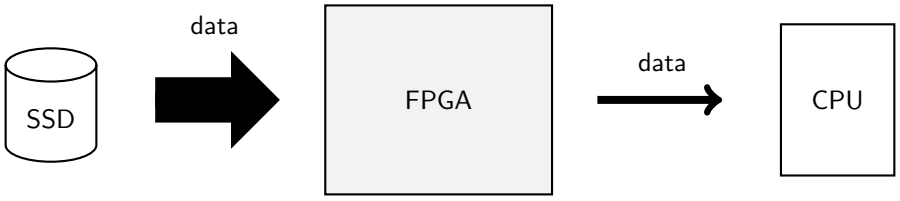

Figure 4: FPGA in data path

volcano-style function calls, ibex is also row based and stores the data row-wise. Ibex uses Groundhog to connected to the SATA disk and SIRC to communicate

to the host.

2.5 SSD

SSDs have diminished the gap between random and sequential (disk) access. Due to increased capacity and decreasing costs over recent years they became an alternative to HDDs. Because their disk latency is by a magnitude lower compared to HDDs, their usage is especially interesting for database systems which are generally limited by I/O.

Unfortunately manufacturers release only little information about the internal

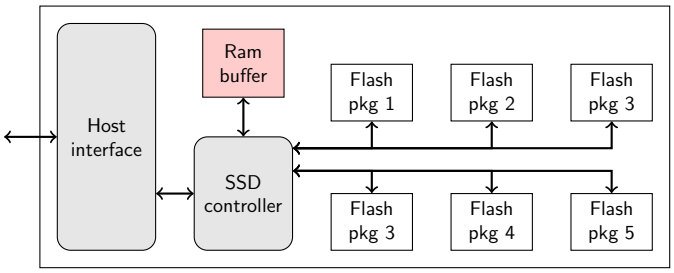

Figure 5: Internal architecture of an SSD

architecture of their SSDs. For that reason they are to some degree like a black box that provides its services through the standardized SATA interface. Feng et al. [5] have undiscovered some general properties of the internal SSD architecture. Figure 5 shows roughly the internal architecture of an SSD. It contains a host controller to communicate over a bus interface. An SSD controller that manages the NAND chips, multiple flash packages and some models contain a RAM buffer to cache data. The flash packages are connected over multiple (2-10) shared channels to the SSD controller.

It is observable that the architecture of SSDs is highly parallel, this is a necessity to achieve high throughputs, since a single flash package has a much smaller speed than the complete SSD. In our implementation we take advantage of the internal parallelism by using NCQ to read the data.

In SSDs parallelism arises on different levels:

- All channels connecting the SSD controller to the flash packages can be operated simultaneously and independently of each other.
- Because channels are shared and connected to multiple flash packages, bus utilization can be increased by interleaving data from different flash packages.
- Parallelism can also occur inside a flash package. A flash package usually contains multiple flash dies. These dies can execute commands independently of each other.

When dealing with LBAs the SSD acts again like a black box. In contrast to HDDs where LBAs map directly to the cylinder-head-sector scheme which guarantees that neighboring LBAs are also physically close, on SSDs LBAs are dynamically distributed and the physical location may change over time. Through the LBA the true physical location of the data is hidden, the mapping from the LBA to the pysical location is done by the SSD controller. Therefore it is not safe to make any assumptions about the physical layout of the data depending on the LBA. Most common SSDs use one of two main techniques to map an LBA to a physical location. For a given value D the *LBA-based mapping* writes the data to the flash domain (LBA mod D), in *Write-order-based mapping* the i_{th} write is assigned to the domain (i mod D). Both mapping policies make sure that data which is written at the same time or sequentially, is distributed over multiple flash packages. When the data is read in the same way as it was written, the SSD can read from multiple flash packages in parallel and achieve a high throughput.

2.5.1 NCQ

NCQ was introduced with SATA-II [3]. It allows to send up to 32 commands to the SATA device. Every command is identified by a 5 bit tag. The controller on the SATA device can execute the commands in parallel and also optimzie the execution order. The data might return out of order but will be identified by the tag. By using NCQ in our implementation, we take advantage of the internal parallelism of the SSD and can achieve a massive performance gain over standard commands.
In general a higher usage of the SSD leads to a higher overall throughput, however Feng et al. [5] showed that especially interleaving writes and reads can have a negative effect on performance. In Data Warehouse workloads with batch writes and read-only queries in-between, this should not be a serious concern.

3 Column Storage on the FPGA

This section discusses how our column storage is implemented on the hardware and in software.

3.1 Data layout on disk

Since most SSDs use 4 KB pages internally, we also use 4 KB pages as the smallest granularity on the FPGA and the software side.

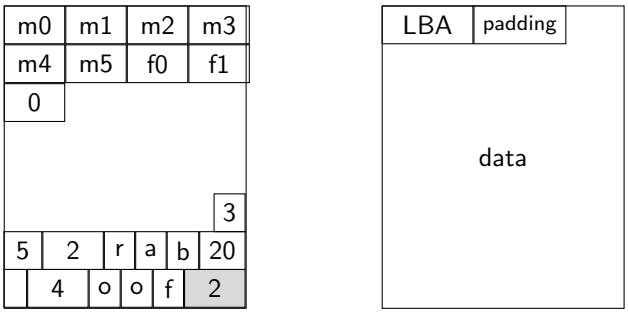

Figure 6: Left: An example meta data page. Right: The first page of a data block.

In a column storage it is necessary to keep track of every single column, this meta data is stored in the first page on the disk, all raw data is then stored sequentially starting from LBA 1. The meta data for all tables using this storage engine and their mapping to the columns is stored on the first page as well.
To simplify the management of the meta data, the table manager[3.6.1] was implemented as part of the MySQL storage engine. The table manager is also responsible for allocating new data blocks on the disk. One data block consists of multiple 4 KB pages and one column is stored across multiple data blocks.

Meta data The meta data contains all information about the tables and their columns, it is stored in the first page on the disk at LBA 0. The table manager manages the meta data of the Ibex storage engine. Because MySQL internally identifies tables by their database name and table name, we adopted the same concept. In the last 32 bit of the first page is the number stored which states how many tables are managed by this engine. From there the table names are stored in backwards order. Every table name is followed by 16 bit for the number of columns and 32 bit for the number of rows that are stored in this table. Figure 6 shows a meta data page which contains the information of two tables called *foo* and *bar*.
Additionally, we have to store three 64 bit values for each column: the LBA where the column begins, the LBA of the current page where new data can

be inserted and the byte-offset on this page to get the actual byte position for the insert. This data is stored in sequential order from the top of the page at LBA 0. Since we only use a few tables which have at maximum 8 columns, we can safely assume that for our experimental setup the two meta data parts will never grow so big that they might overlap somewhere in the middle of the 4 KB page. To support an arbitrary number of tables and columns it has to be detected when the meta data page becomes full and then the meta data needs to be distributed over multiple pages.
To reuse the disk blocks when a table is deleted the LBAs of freed data blocks are stored after the column meta data at the top of the page. The table manager will reuse these LBAs when a new block on the disk is requested. After the freed LBAs a zero value is stored to indicate the end of the meta data.
To guarantee persistence of the meta data, the storage engine can trigger a flush to the disk, this happens every time data is inserted or deleted. On startup the system reads the meta data and can restore the properties of all the tables.
Figure 6 shows an example meta data page, containing two tables named *foo* and *bar*, the former has four columns and 20 rows, the later two columns 35 rows. The meta data of these six columns is stored at the top of the page indicated by m0...m5. After the column meta data, the LBAs of two freed data blocks are stored. The end of the meta data is indicated by a 64 bit zero value.

Data The columns are stored across data blocks consisting of multiple pages. With increasing data block size the amount of disk space that is allocated but unused increases, actually the last data block is on average half empty. Using smaller data blocks would decrease the amount of unused storage but leads to more jumps between data blocks which might have an influence on the performance. In the evaluation, Section 5.4, we will see that there is no measurable impact on the performance when varying the data block size.
The first 48 bit of each data block are used for the LBA of the next data block, in case there is no next block, the LBA equals the LBA of the current block. After these 48 bit for the LBA another 16 bit padding is inserted to optimize the alignment of the data which simplifies reading it out on the FPGA. The exact number of 4 KB pages that are part of a block is fixed a priori in the software and on the FPGA. An example of the first data page in a data block is shown in Figure 6.

3.2 Implementation on the FPGA

The hardware implementation builds the main part of our work and is extensively explained in this section.

3.2.1 Buffering System

As explained in Section 1.1 we want to hide the access latency of the SSD by using NCQ. The disadvantage of using NCQ is that the data might be returned

out of order, to bring it back into order a buffering system on the FPGA is necessary. Since we do not want to reprogram the FPGA between queries or different workloads, we had to fix the number of buffers we want to use before hand. As a consequence the implementation is not able to process more columns simultaneously than specified. We opted for a trade-off between performance and design complexity which has a direct impact on hardware resource consumption. For that reason we implemented a design that supports up to 8 columns simultaneously. Although we strived to make the source code as generic and scalable as possible, we introduced two restrictions to simplify the code base and the hardware design.

- The number of buffers and NCQ-tags needs to be a power of two. This restrictions extremely simplifies the scalability of our design. To double the capacity of the design, in most cases the register can just be extended by a single bit. Supporting other numbers would require much more checks and complexer generics to scale properly.

- During a query every column is assigned to the same number of buffers. Although with this restriction some buffers might be unused, it simplifies the hardware part that has to assign NCQ-tags to columns because the amount of possible assignment configurations is heavily reduced and contains no complex special cases.

Double buffering is used to buffer the data fetched from the disk. With double buffering it is possible to hide the disk latency, because it is possible to fetch the next chunk of data while the current one is still processed. Our default implementation uses 2×4 KB double buffers matching the 4 KB page size on the SSD. Eight double buffers are required to process up to eight columns.
Our initial approach did bind every column to one of the eight double buffers. While this is the simplest approach, it bears one main disadvantage, it would only use as many buffers simultaneously as columns are processed. If the query only contains a single column seven buffers would stay unused. With this approach neither the sparse hardware resources are used in an efficient way nor are we achieve acceptable performance.
So instead of binding a column to a specific buffer, we are binding a NCQ-tag to a specific buffer. This means that we will only use 8 out of 32 possible NCQ-tags, but experiments from Feng et al. [5] and our own evaluation in Section 5.4 have shown that already 8 NCQ-tags lead to a significant performance boost. This approach makes it possible to assign a column at runtime to one or multiple NCQ-tags. If there is more than one NCQ-tag assigned to a column, the data read from the disk is distributed in round robin among the associated buffers, this is necessary to restore the original order later on.
Table 1 shows how the NCQ-tags are assigned to a specific column depending on the number of columns that are processed simultaneously.
An example of how the data chunks are distributed to the buffers can be seen in Figure 7. In this example a query with three columns is executed, this means every column is assigned to two buffers. In the figure Column 0 uses Buffer 0

	# columns processed								
column	1	2	3	4	5	6	7	8	
0	0 - 7	0, 2, 4, 6	0, 4	0, 4	0	0	0	0	NCQ-tags
1	-	1, 3, 5, 7	1, 5	1, 5	1	1	1	1	
2	-	-	2, 6	2, 6	2	2	2	2	
3	-	-	-	3, 7	3	3	3	3	
4	-	-	-	-	4	4	4	4	
5	-	-	-	-	-	5	5	5	
6	-	-	-	-	-	-	6	6	
7	-	-	-	-	-	-	-	7	

Table 1: NCQ-tag assignment depending on number of columns processed.

and Buffer 4, Column 1 Buffer 1 and Buffer 5 and Column 2 Buffer 2 and Buffer 6. Buffer 3 and Buffer 7 are unused. As can be seen the first 4 KB chunk goes to the first buffer and the second chunk to the second buffer. In round robin the third chunk goes again to the first buffer and the forth chunk to the second buffer.
In our implementation at least $\frac{N}{2} + 1$ buffers are used. To put it differently in the worst case $\frac{N}{2} - 1$ buffers are unused. For the example table with 8 columns the worst case, as shown in Table 1, is the utilization of only 5 buffers which fulfills exactly our formula.

3.3 Architecture Overview

The hardware module for the column storage was integrated into the existing Ibex design and uses the Host bus adapter (HBA) module to connect to the SSD and the SIRC module to communicate to the CPU over Ethernet. In this section we will examine the Column Storage module which replaces the existing row storage. The Column Storage module itself consists of multiple modules. Figure 8 shows the modules, which are part of the Column Storage module and also shows the data flow through the module. We briefly describe the meaning of every module and will examine in detail the Loader module and the Reordering structure due to their importance and complexity.
Since data is read in 16 bits per cycle from the SATA port and delivered with that throughput by the HBA, the Column Storage module also processes 16 bits per cycle until it materializes the columns into actual rows.

HBA As stated in Section 2.2 the HBA is provided by Groundhog. The Column Storage module only uses it to read data with NCQ. Direct memory access (DMA) reading and writing to the disk is already provided by the Ibex module.

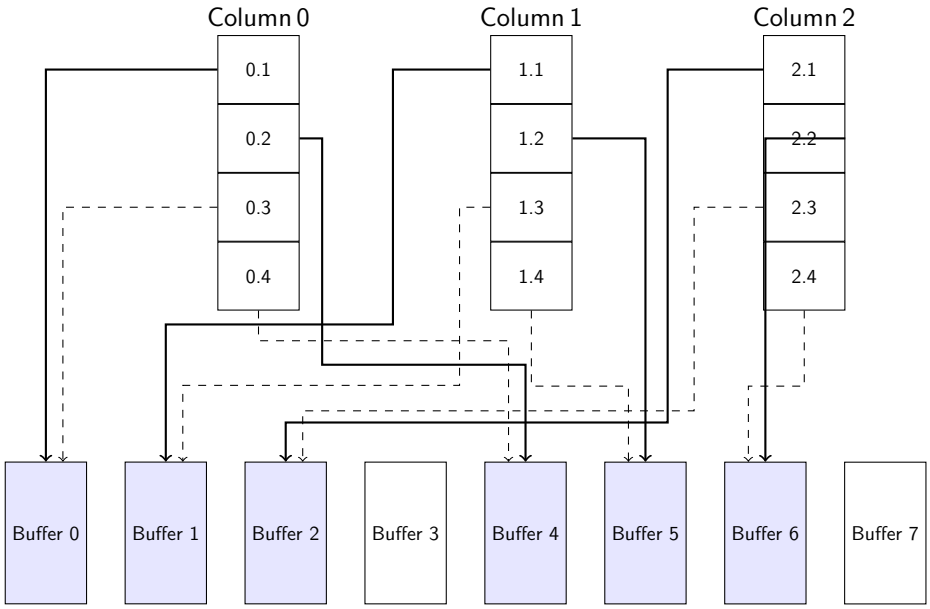

Figure 7: The solid line shows the chunk first going to the assigned buffer and the dashed line the next chunk going into that buffer

Column Storage module The Column Storage module is the module that contains all logic to read the data stored as columns on the disk, materialize it into compact rows and then forward these rows to the next operator in Ibex. As input it takes the LBAs of the requested columns and also a value that encodes the data-type of each column. In the current version the data-type information is not used since all experiments are done with unsigned 32 bit integers. When using multiple data-types, this information would be necessary to materialize the rows correctly.

Loader The Loader module is initialized with the start LBA of each column. The Loader iterates through the 8 NCQ-tags and checks if the corresponding double buffer is half empty which implies it has enough space for another chunk of data. If this is the case the Loader sends an new NCQ request to the HBA module.
To receive data the Loader also has to check at every clock cycle if for any of the pending requests new data is available from the HBA. Together with the data the HBA also sends the NCQ-tag, so that it is possible to identify the command that requested this chunk of data. The Loader forwards the arriving data and the corresponding NCQ-tag to the buffer-pipeline. At every buffer in the pipeline it is checked if the NCQ-tag matches the tag of the buffer if this is the case the data is loaded into the buffer otherwise it is forwarded in the

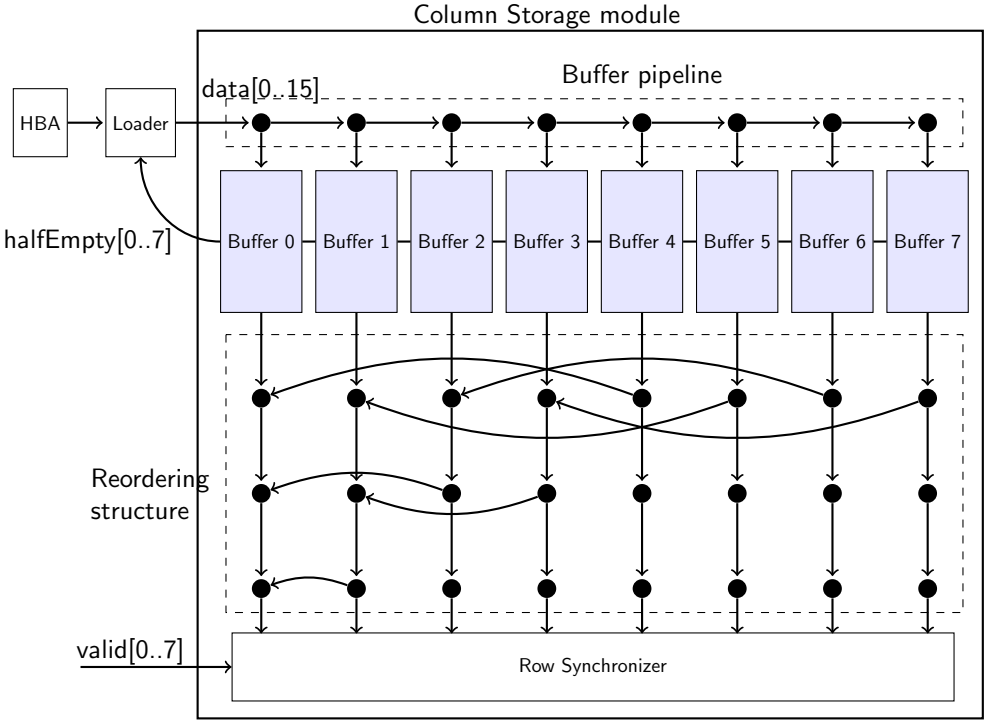

Figure 8: Design overview and data flow through the Column Storage module

pipeline. The Loader is explained in more detail in section 3.4.

Buffers The buffers are double buffers consisting of two single 4 KB FIFO-buffers, double buffering is used to hide the high latency of the disk. When the first buffer is empty the *halfEmpty* signal is set to TRUE. The Loader is observing this signal and reacts by requesting a new chunk of data for the corresponding buffer. Since data is always loaded in 4 KB blocks no further checks on the state of the buffer are necessary and the data can directly be pushed into the buffer when it becomes available.

Reordering structure The Reordering structure is the fan out that combines the output of all buffers and brings the data chunks back into order. It actually consists of multiple modules: *Merger*, *Intersection* and *Delayer*. To minimize the wiring and still being able to combine the data from all buffers into a single stream a tree like design is used. At each level two data paths are interconnected, there are four interconnections in the top level and only one in the bottom level. The ordered data is then feed into the Row Synchronizer. The Reordering structure is explained in more detail in section 3.5.

Row Synchronizer The Row Synchronizer combines the data streams of all columns and produces a compact row. As an input it receives a validity signal with a bit for each column which indicates if the corresponding column is part of the current query. For every valid column the Row Synchronizer sends a request and waits for the data. If it receives the requested data, it holds a compact row on its data inputs and can then forward all attributes at once. The following modules have the guarantee that the data on all parallel data paths belong to the same row. If an attribute of a row is longer than the 16 bit-width of the data path, the row is span over multiple clock cycles. The Row Synchronizer is the last part in our data flow and because it requests the data indirectly from the buffers, it also is the driving part for the dequeuing.

3.3.1 Challenges on the Hardware

The first design approach did not use a pipeline from the Loader to the buffers instead the data was distributed through a fan-out to all buffers which connected the Loader directly to every single buffer. With increasing number of buffers, this design did not scale since the distance between the buffers and the Loader got so long that the timing constraints on the FPGA were no longer met. To avoid the timing issues the current buffer pipeline was introduced, it only connects the first buffer directly to the Loader, all other buffers are connected indirectly. This chain of buffers simplifies the placing on the chip and eliminates the previous timing issues.
Similar issues arose with the modules that dequeue the data from the buffers. At first there were multiple consumer modules which were directly connect to multiple buffers. Again with increasing number of buffers the distance between the consumer modules and the buffers became to long and proper placement was no longer possible. The Reordering structure avoids these timing issues by delaying the data over several registers and thereby elongating the data path and enabling a longer distance between the buffers. This design eliminated the timing issues.
Further timing issues occurred inside the Loader module. The module contains a lot of logic and also needs to store a lot of data in registers. Most of the space is required to store the LBAs with a size of 48 bit each. Because we need to store the LBA of the current page and also the one for the next data block, we actually have to store two LBAs for every buffer. With increasing number of buffers it became impossible for the hardware synthesizer to place the required registers close enough to each other. Therefore the registers that held the LBAs had to be replaced by two BRAMs one for the current LBAs and one for the LBAs of the next blocks.
Despite these timing optimizations the current implementation does not synthesize with 16 double buffers. Either the synthesizer is not able to connect the massive amount of wires and registers or there are not enough hardware resources available. Even for smaller buffers number routing is an issue when deploying the design and takes a lot of time (approx. 10-15min.).

3.4 Loader module

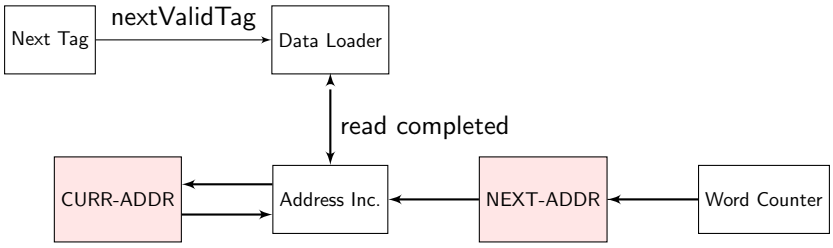

Figure 9: Submodules of the Loader module

The Loader is the most complex component and as can be seen in Figure 9, it actually consists of multiple subcomponents. The software provides the Loader at initialization always with 8 addresses, one for each buffer. If there are multiple buffers for a single column then the corresponding addresses are offset by 8 sectors each. For example with two columns and 8 available buffers, the first column is assigned to buffers 0,2,4,6 and the data loader sends simultaneously a NCQ request for all four buffers. The LBAs of these four requests differ exactly by the value of 8 sectors which means they are offset by 4 KB.

The Word Counter is necessary to determine when a data block starts or ends. When a new data block starts the first 48 bit, containing the next block LBA, have to be read out and are stored to the *NEXT-ADDR* BRAM. The column number is used as the memory address to access the LBAs in the BRAMs. The Addr Inc. module observes which LBAs were successfully requested from the HBA and increments them in the *CURR-ADDR* BRAM. If the current data block is completely read from the disk, Addr Inc. loads the address of the next data block from the *NEXT-ADDR* BRAM into the *CURR-ADDR* BRAM, instead of incrementing the current LBA further.
One output of the *CURR-ADDR* BRAM is directly connected to the HBA module and holds the LBA for the next read command. The address corresponding to this output is set by the data loader.
In order that the data loader knows which NCQ-tags can be reused, the Next Tag module checks all 8 tags for validity and reports the next valid tag to the data loader.

3.5 Reordering structure

To dequeue the data from all these buffers and bring them back into order the data paths were arranged into an upside down tree, see Figure 10. At each level two not yet connected data paths have an interconnection. This structure minimizes the amount of interconnections and also decentralizes the reordering logic which is mainly inside the Mergers. These two conditions make

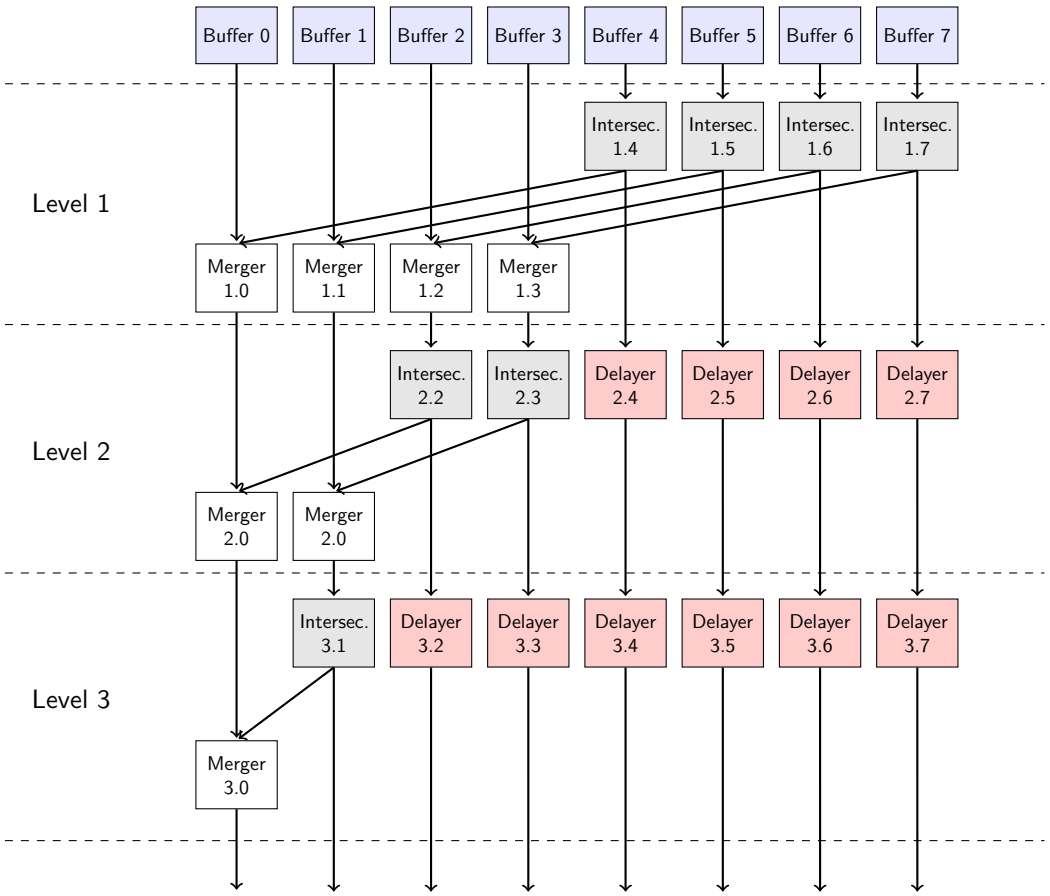

Figure 10: Design of the Reordering structure

sure that the synthesizer can successfully place and map the implementation to the hardware.
The structure brings the data back into order when multiple buffers are used for a single column. In the base case of 5-8 columns where one buffer per column is used, none of these crossing data paths are used and the data flows straight from the buffer through the three levels to the Row Synchronizer. For 3-4 columns the crossing paths on the first level are used, for 2 columns the ones on the first and second level and for a single column all crossing paths are used and all the data will eventually end up in the first data path.
When the data has passed trough all three levels it arrives fully ordered at the Row Synchronizer.

Merger The Merger is the essential node in this hierarchy which makes sure that the data comes back into order. Depending on the query that is processed the Merger knows if it has to read data from both of its sources or only from its primary source. In case it needs to read the data from both sources it will read 4 KB of the first source and then switches to the second source where it also reads 4 KB and then it switches back to the primary source and continues in round robin.
Thanks to the way the data was distributed and loaded into the buffers, it is possible to enable merging on certain levels depending on the amount of columns processed and the data will be brought back into order.

Delayer The Delayer has no actual functional logic, it just stores the data for one clock cycle, thereby the data path to the BRAMs is prolonged. The longer these data paths are the more apart the BRAMs can be placed, this allows the synthesizer to find a mapping on the FPGA despite requiring so many BRAMs at once.

Intersection The Intersection requests the data from its single source and forwards it to the next level. It does not know to which of the two successor modules the data is transferred. Apart from having two successors the Intersection module is equal to the Delayer module.

3.6 Software

The Ibex MySQL storage engine was adapted so that it pushes the SQL projection down on to the FPGA where the compact rows are materialized as early as possible. The software writes the LBAs of all projected columns in the input buffer of the FPGA other necessary parameters are written into the shared 32 bit parameter registers. The Column Storage module then executes the projection. Because the SIRC interface only supports data communication over the input and output buffer which are limited to a size of 16 KB, only a part of the result is in fact transmitted back to the software. If the SIRC interface would support data streaming from the FPGA to the CPU, the complete result could be transfered back to the software. It is possible that in this case the data transfer over Ethernet might become the bottleneck instead of the disk I/O.
Since MySQL is row based and expects rows which contain all fields, the missing fields are inserted and padded with NULL values before the full-length row is returned to MySQL.

3.6.1 Table Manager

The table manager stores the LBA of the first page of each column, the current page LBA and the byte position in the current page where new entries can be appended. When a new block of data is requested the table manager returns the next free LBA. Data blocks are of a predefined size. The table manager also handles drops of tables and stores the data block LBAs that are now freed in a

free list. As explained before all this meta information is stored at LBA 0 on the disk and is read on startup.
The data block size has a lower limit, because we need to be able to distribute a single block among 8 buffers, therefore the block size has to be at least 8 times the buffer size which is 4 KB. In practice there might occur some race conditions between the buffers, to avoid them the block size needs to be 16 times the buffer size. This means the lower bound on the data block size is 64 KB.

4 Data Analytics on the FPGA

Based on the column storage introduced in Section 3, we want to make use of the column-wise data streams and compute statistics on the data passing through the FPGA. If this is done before the columns are materialized into compact rows the parsing of the data is reduced to parsing of the data-type.
The objective is to compute statistical values at the line rate. Therefore we have to use the internal BRAMs to store the data, accessing the DRAM is not an option since it would add too much latency. Assuming large data sets, we focus on statistical values which require not much memory and can be easily computed on a data stream.
For this reason we decided to start with some straightforward computable statistical values, namely the *mean*, *variance*, *skewness* and *kurtosis*. These four values are mathematically related and it is possible to compute all of them from a few summations over the data. The general formula computes the mean value in a first pass and then passes again through the data to compute the final value which leads to two passes through the whole data set. By rearranging the formula as in Section 4.2, it is possible to compute the necessary summations in one pass over the data set.
After all the data has passed, the computed summations are sent to the CPU where a few additional multiplications and divisions are executed to compute the final values. Depending on the data set size one has to make sure that the summation registers on the FPGA will not overflow.
The goal of our implementation is to compute these statistical values with minimal overhead during each query. Since the whole data is streamed through the FPGA anyway, there should be no additional processing time or latency to compute all the necessary summations (aggregations). A slight overhead will occur due to transmitting the values to the CPU and computing the final multiplications and divisions in software.

4.1 Implementation on the FPGA

We successfully simulated the computation of all the four statistical values, for our implementation on the hardware we decided to only implement the mean and the variance. Two main obstacles occurred when moving the code from the simulation to the hardware. Timing issues because of bad placement and integer multiplication which requires the use of dedicated DSP slices.
Initially the idea was to place the Statistics module directly into the data path right before the materialization in the Row Synchronizer occurs. But this led to timing issues, caused by the very large registers used for the summation and the distance to the two DSP slices. To solve the timing issues the Statistics module was placed outside of the data flow path and the data signal is branched of and delayed over some registers to increase the data path between the Statistics module and the other modules. Figure 11 shows the final placement of the Statistics module, it is also connected to two DSP slices to compute the multiplication. The data path between the Statistics module and the DSP slices is

also pipelined over a few registers to avoid any timing issues.

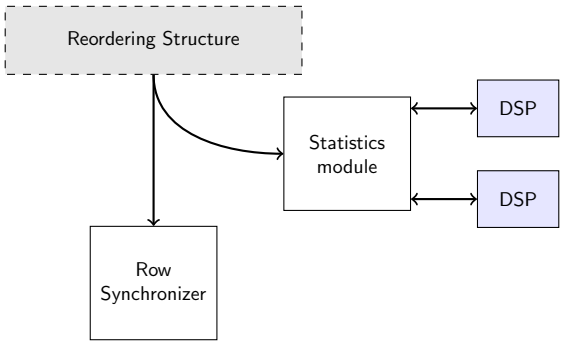

Figure 11: Placement of the Statistics module

A single DSP slice can only multiply at most a 25 bit and a 18 bit number. To multiply two 32 bit unsigned integers, we have to pipeline the multiplication over two DSP slices.

The Statistics module has no influence on the data flow and just observes the

$$A[31:0] = A_H[15:0] \;\&\; A_L[15:0]$$
$$B[31:0] = B_H[15:0] \;\&\; B_L[15:0]$$
$$A \times B = A_L \times B_L + A_H \times B_L + A_L \times B_H + A_H \times B_H$$

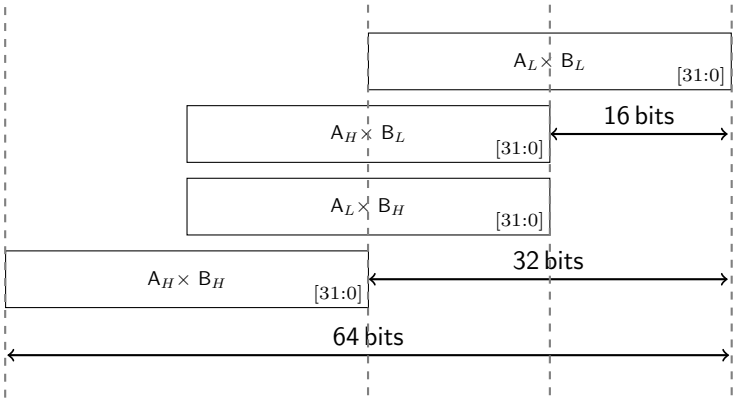

Figure 12: Pipelined 32×32 multiplication using 16×16-DSP slices.

bypassing data, therefore it does not add an additional delay to the current column processing on the FPGA. The data flow is still controlled by the Row

Synchronizer and limited by disk I/O.

4.1.1 Multiplication pipelining

As shown in Figure 12 a 32 bit × 32 bit multiplication can be separated into four 16 bit × 16 bit multiplications. Two 32 bit numbers A and B are split into their upper and lower 16 bit parts and then the four multiplications $A_L \times B_L$, $A_H \times B_L$, $A_L \times B_H$ and $A_H \times B_H$ are computed. Since we compute the square of a number, A and B are equal and therefore $A_H \times B_L$ is equal to $A_L \times B_H$. The partial results have to be shifted accordingly and summed up into a 64 bit register. The module receives only 16 bit of data per clock cycle and stores the first 16 bit of the integer value until the second part arrives.
For the summation two different registers are used. The value of $A_L \times B_L$ is added to first register in one clock cycle and the value of $A_H \times B_H$ is added to the same register in the other clock cycle. The value of $A_L \times B_H$ value is added to the second register when both 16 bit parts are available. This means the 32×32 multiplication is pipelined over two clock cycles. The two summations registers are aggregated before they are sent to the CPU.

4.1.2 Software

On the FPGA we can use registers of an arbitrary size, due to the high values that are possible we used up to 80 bits for the aggregated sum. On the software side we only have support for up to 64 bit unsigned integers. Therefore it was necessary to implement our own division for larger integer values.
The final *mean* and *variance* values are not feed into MySQL, instead they are printed out to the console. A future version might tap into MySQL's own AVG() and VARIANCE() functions, such that theses values can be retrieved over the MySQL interface.

4.2 Formulas

We list the formulas and their rearrangement which are used to compute the four statistical values in one data pass.

Mean The mean is very simple to compute, two summations are required one of the values and one for the amount of rows. Since the number of rows is already stored in the storage engine and the division will be executed in software as well, it is not necessary to compute this value on the FPGA.

$$\mu = \frac{1}{N} \sum_i x_i$$

Variance The variance is usually computed in two passes, one pass for the mean value and a second pass for the final value. We rearranged the formula such that we get two summations which can be computed simultaneously.

$$\sigma^2 = \frac{1}{(N-1)} \sum_i (x_i - \mu)^2 = \frac{\sum_i x_i^2}{N} - \left(\frac{\sum_i x_i}{N} \right)^2$$

Skewness To compute the skewness usually multiple passes through the data set are necessary. Rearrangement of the formula lead to a few summations which can be computed in one pass.

$$\frac{N}{(N-1)(N-2)} \cdot \frac{\sum_i (x_i - \mu)^3}{\sigma^3}$$

where

$$\sum_i (x_i - \mu)^3 = \sum_i x_i^3 - 3\mu \sum_i x_i^2 + 3\mu^2 \sum_i x_i - \mu^3 N$$

Kurtosis The method applied to the skewness is also valid for the kurtosis.

$$\frac{N(N+1)}{(N-1)(N-2)(N-3)} \cdot \left(\frac{\sum_i (x_i - \mu)^4}{\sigma^4} \right) - 3 \cdot \frac{(N-1)^2}{(N-2)(N-3)}$$

where

$$\sum_i (x_i - \mu)^4 = \sum_i x_i^4 - 4\mu \sum_i x_i^3 + 6\mu^2 \sum_i x_i^2 - 4\mu^3 \sum_i x_i + \mu^4 N$$

5 Evaluation

In this section we evaluate the Column storage module and the Statistics module. As a reference point the raw performance of the SSDs is measured as well.

5.1 Experimental Setup

All experiments are done on the same desktop PC running an Intel Core i5-750 with 4 cores clocked at 2.67 Ghz and 8 MB of L3 cache. The machine is equipped with 8 GB of main memory. We use MySQL 5.25 running on Windows 7 (64bit). The FPGA (Virtex5, XC5VLX110T) is directly connected over SATA-II to the SSD. Two SSDs manufactured by OCZ are used.
An OCZ Vertex2(OCZSSD2-2VTXE60G) which has a capacity of 60GB and an OCZ Vertex4(VTX4-25SAT3-256G) with a capacity of, additional specifications can be seen in Table 2. The FPGA was connected to the host PC over gigabit Ethernet. The Ibex and HBA module are running at 150 Mhz matching the frequency of the SATA-controller chip. This leads to the maximum SATA-throughput of 300 MB/s.
If not stated otherwise the data read in the experiments has the size of 1 GB

	Vertex 2	Vertex 4
Capacity	60 GB	256 GB
Flash Memory	MLC	MLC
Controller	SandForce 1222	Indilinx Everest 2

Table 2: Manufacturer specifications of the two SSDs

and only contained unsigned 32 bit integer fields. The tables were written in one batch.
The query was always a simple projection of the form:

```
SELECT FIELD_1 , . . . , FIELD_N
FROM TEST_TABLE
```

Because the current version of SIRC does allow to stream the data over Ethernet, the data is not transmitted back to the CPU instead it is dropped after the compressed row is generated on the FPGA.
In the plots the different SSDs are annotated with the abbreviations V2 for the Vertex 2 and V4 for the Vertex 4.

5.2 SSD performance

To get an idea of the raw performance of the SSDs used for the experiments, sequential and random performance is evaluated.

Figure 13 shows the sequential read performance depending on the read block size. For comparison the random access performance using NCQ with a

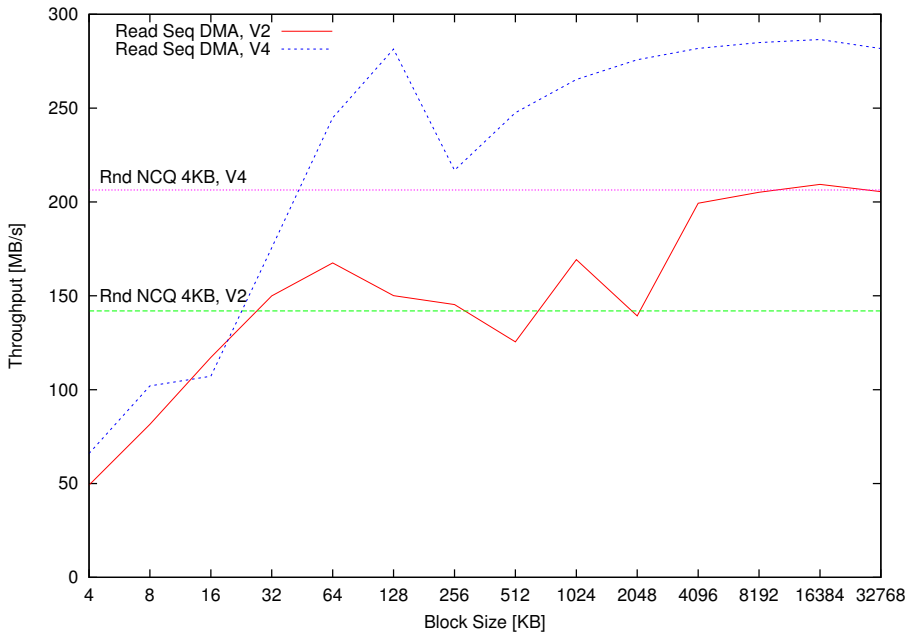

Figure 13: Performance of SSDs

constant block size of 4 KB and a queue depth of 32 is also plotted.
The performance of both SSDs does not increase steadily with increasing block size. The graph shows that in general very small read blocks have a negative effect on read performance while after some threshold, around 64 KB, the performance gain by increasing the block size starts to stagnate. Using NCQ it is possible to achieve high throughputs with random access and small block sizes. The base case for our experiments is the row storage engine in Ibex. It reads the data sequentially in 4 MB blocks, therefore it should be able to achieve throughputs up to 200 MB/s or 270 MB/s. The column storage implemented in this work reads randomly 4 KB data blocks which equals to the page size of most current SSDs. Therefore the random access NCQ performance with 4 KB blocks is the maximum performance our implementation can theoretically achieve.

5.3 Queue Depth

The impact of the queue depth on the performance is evaluated by performing random 4 KB reads. The results are shown in Figure 14. On the older Vertex 2 the performance increases linearly to around a queue depth of 8 and then reaches the maximal throughput of 140 MB/s. Whereas the newer Vertex 4 shows a linear throughput increase up to a queue depth of 32. The scale up in this experiment mainly depends on how the flash packages in the SSD are connected.

As explained in Section 2.5 different models have a different amount of actual hardware channels. Interpreting the graph it appears that the newer model must have more hardware channels, so that it can indeed scale to the maximal queue depth and reach a higher maximal throughput. Since the Vertex 2 already hits its peak performance with 8 NCQ-tags, there is no gain in using more than 8 NCQ-tags. The main reason to only use 8 NCQ-tags in our implementation was to keep the complexity of our design within bound and thereby limiting the occupied hardware resources to a reasonable amount. We also assumed that the performance gain would decrease after some threshold as it is the case for the Vertex 2, but naturally newer SSDs are performing better and this assumption does not hold.
There is a trade-off between performance and hardware consumption but if the hardware resources are available it should be possible to scale our design up to 16 or even 32 buffers and take advantage of the higher throughput.

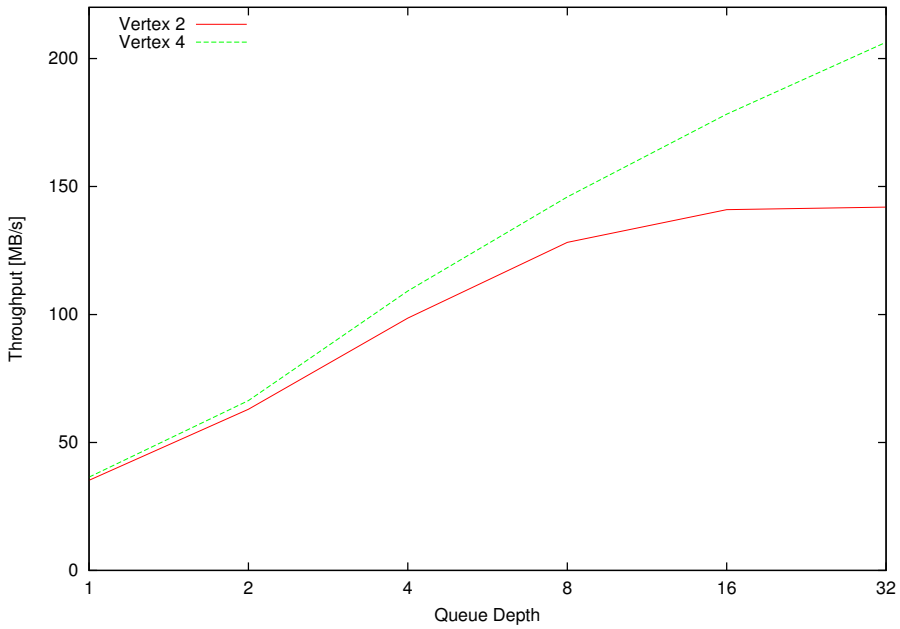

Figure 14: Random access performance depending on queue depth

5.4 Evaluation Column Storage

5.4.1 Buffer size and Double buffering

In this section we want to determine the influence of double buffering and the buffer size. The experiment is only run with the Vertex 2. Two different tables are used, one with a single column and another with 8 columns.

Figure 15 shows the difference between double and single buffers while varying

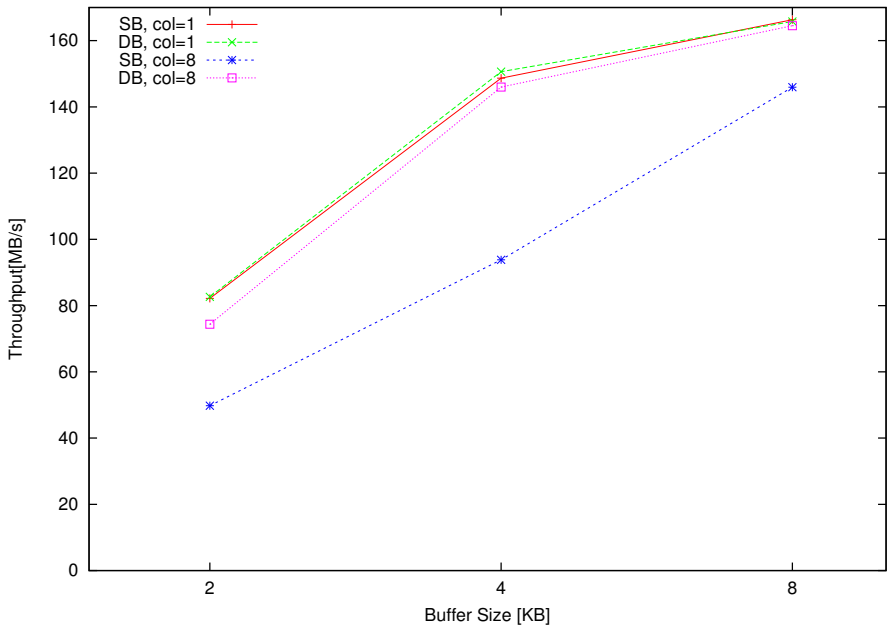

Figure 15: Performance depending on buffer size

the buffer size. Both queries make use of all 8 buffers.

When executing the query with 8 columns the data from all 8 buffers have to be synchronized in the Row Synchronizer before they columns can be materialized into a row. This synchronization blocks the data flow until the corresponding data is available on all 8 data inputs of the Row Synchronizer. With single buffering this synchronization has a negative impact on the performance, since all buffers have to be filled with data before any of them is dequeued. For single buffering the performance increases linearly until the maximal SSD throughput of 140 MB/s is reached. Double buffering can hide the blocking which occurs in the Row Synchronizer and increases performance to 140 MB/s for 4 KB blocks. When running the single column query no synchronization among columns is necessary in the Row Synchronizer, this means the data which already arrives in order at the Row Synchronizer can immediately be forwarded. Therefore the single column query does not experience a negative performance effect with single buffering. For that reason the single column query is a special case in

this experiment.
The experiment shows that with double buffering we are able to hide the data flow blocking in the Row Synchronizer, the throughput of about 140 MB/s matches the maximal random NCQ read performance of the SSD.
It is also visible that doubling the buffer size from 4 to 8 KB is not worth the hardware consumption since the performance increases only slightly.

5.4.2 Buffer number

We have seen that double buffering is able to hide the blocking that is necessary to synchronize the columns. The next experiment evaluates how the number of used buffers affects the performance. A full table scan on a single column table is executed while the number of available buffers is varied. The experiment is run in our default configuration with 4 KB double buffers and on the Vertex 2.
The outcome of the experiment is shown in Figure 16. The throughput almost

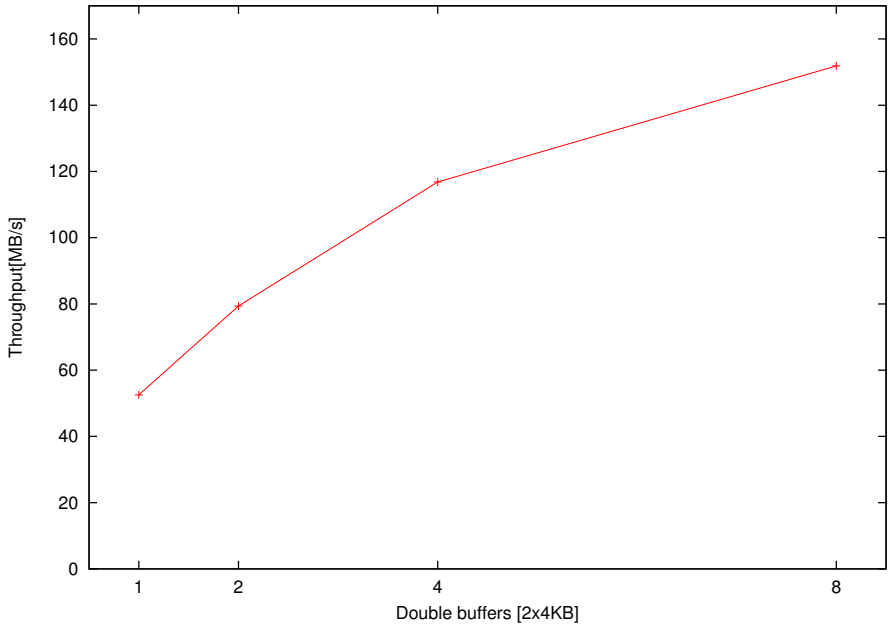

Figure 16: Performance depending on buffer number

scales linearly until it reaches the maximal throughput of 140 MB/s.
The graph also indicates that without the ability to adapt and use all 8 buffers for a single column, the throughput would only be around 50 MB/s. Thanks to our flexible design we can use all 8 buffers and almost triple the throughput.

5.4.3 Full Table Scan

This experiment runs full table scans on tables with a different number of total columns, varying from 1-8. It shows us how tables with a suboptimal number of columns (e.g. not a power of two) affect the throughput.

Figure 17 shows the result for the Vertex 2 and Vertex 4. The result for the

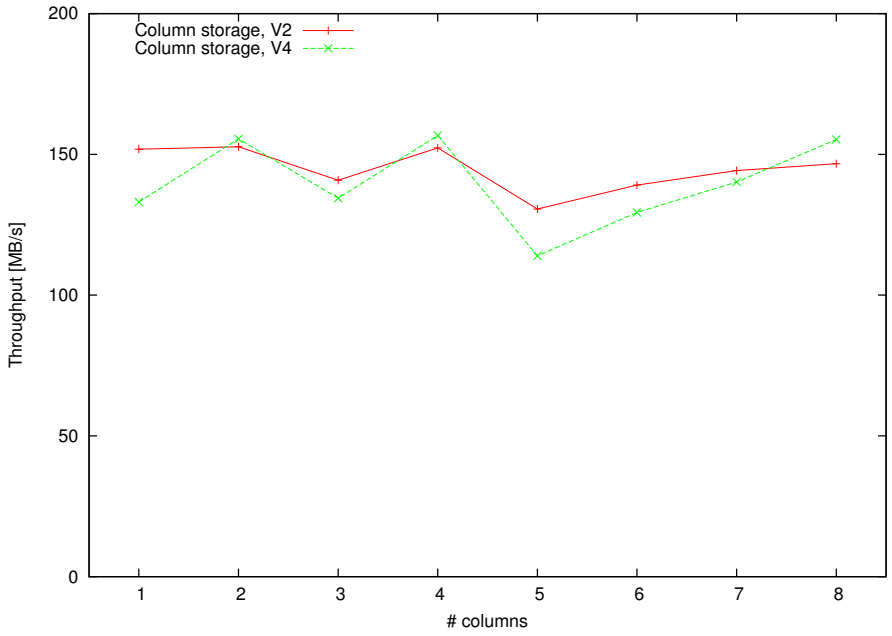

Figure 17: Full table scan, varying number of columns

Vertex 2 fits the properties of our implementation very well. It can be seen that for a single and two columns the performance is almost equal. It decreases for three columns because in this case only 6 out of the 8 buffers are actually used. When querying 4 columns all 8 buffers are used and the throughput is back up on the initial level. The throughput also drops in the case of five, six and seven columns where also not all 8 buffers are in use but it steadily increases from 5 to 8 columns as the number of buffers in use increases as well.

The Vertex 4 shows a similar pattern although the performance penalty when using less than 8 buffers is slightly higher. Oddly the Vertex 4 performs not well when the table only contains a single column despite the fact that all 8 buffers are in use. Furthermore the data lies sequentially on the disk and was also written in that way, this actually should benefit the performance since the data is probably distributed over multiple flash packages.

This experiment demonstrates that our implementation performs quite well and thanks to the adaptability we can retain the performance for varying number of columns. In fact the throughput of around 145 MB/s matches the random

access speed for 4 KB blocks which we measured in Section 5.2.

5.4.4 Projection

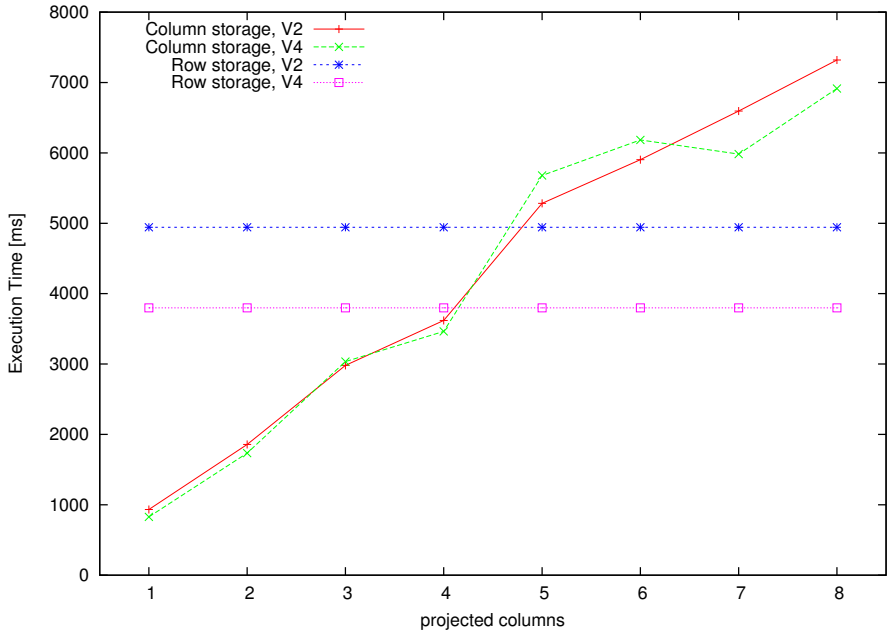

Figure 18: Execution time for Column and Row storage, varying the number of projected columns

In this section we want to compare the performance of our column storage to the existing row storage, the base case for our experiments. Particularly we want to find out how the column storage can benefit from pushing the projection down. We run the projection on a table with 8 columns and vary the number of projected columns. The complete table holds 1 GB of data or 128 MB per column. Instead of the throughput we plotted the execution time as can be seen in Figure18.
Because the row storage can not push the projection down, it has to read the whole table in all cases which leads to a constant execution time. The column storage on the other hand only reads the projected columns and therefore its execution time depends linearly on the number of projected columns. Both storage types have a constant throughput from the disk to the FPGA, the row storage around 207 MB/s for the Vertex 2 and 270 MB/s for Vertex 4, the column storage, as shown in the previous experiment, around 140 MB/s for both SSDs. If more than 4 or 5 columns are projected the row storage with its higher throughput is clearly faster than the column storage. But when less than 4 columns are projected, the column storage has to read less than half of the data

which means it is able to execute the query faster than the row storage.
This experiment shows that if the row storage has to drop more than half of the fields, it is outperformed by the column storage.

5.4.5 Resource consumption

Table 3 shows the resources required by the Column Storage module. When doubling the number of buffers the amount of required resources is less than double, this shows that our design scales well. It is also visible that this module consumes 10-15% of the available resources which is a big expense in comparison to the row storage. On our dated FPGA model a lot of BRAMs are occupied, but current models are easily equipped with over thousands of BRAMs such that the consumption of our module would be minor.

	#buffers	Slices		LUTs		BRAMs	
available	-	17,280	100%	69,120	100%	148	100%
occupied	8	2,803	16%	5,914	9%	23	16%
	4	1,551	9%	3,178	5%	16	11%
	2	1,002	6%	1,893	3%	12	8%

Table 3: Resource consumption of the Column Storage module depending on the number of buffers

5.5 Evaluation Statistics module

5.5.1 Performance

To find out if the Statistics module has a negative impact on performance we run full table scans on our column storage. Once without the Statistics module in place and once with the module in place. The experiment was only run with the Vertex 2.
The measurements are shown in Figure 19. As expected the statistical operators add no or an insignificant overhead and have no measurable impact on the throughput. Thanks to the parallelism of the FPGA the statistical values can be computed for "free". The only expense are the required hardware resources.

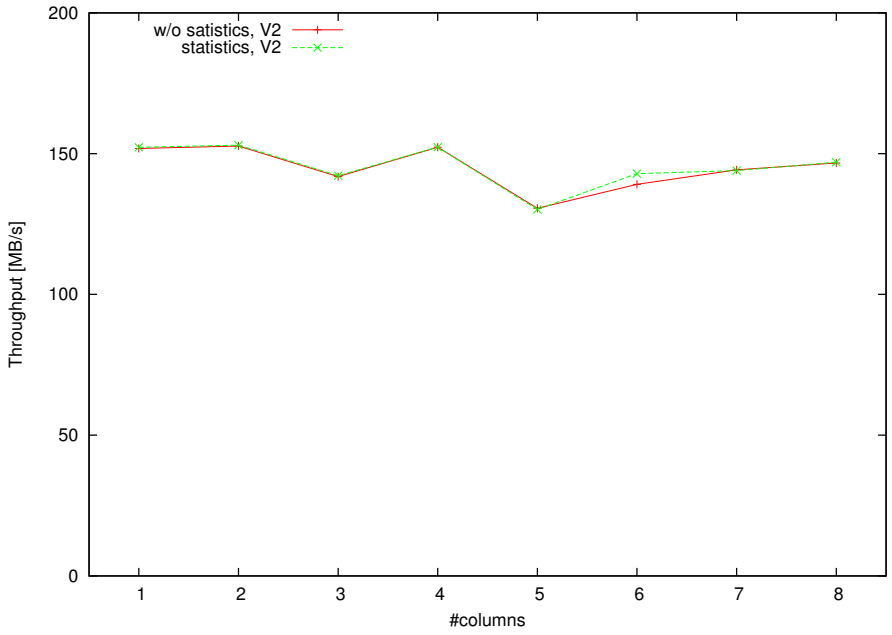

Figure 19: Performance of the Statistical module

5.5.2 Resource consumption

The resource consumption of the Statistics module is listed in Table 4. Although it is a much smaller and less complex module than the column storage it also consumes around 10% of Slices and LUTs. Responsible for the high hardware consumption are the large registers which are necessary to store the summation values. Additionally this module requires two DSP slices per column which are required to compute the multiplication of two 32 bit integers.

	#columns	Slices		LUTs		DSP48	
available	-	17,280	100%	69,120	100%	64	100%
occupied	8	2,262	13%	5,784	8%	16	25%
	1	282	1.6%	724	1%	2	3%

Table 4: Resource consumption of the Statistics module

6 Conclusion

6.1 Thesis contribution

In this thesis we have shown how a column storage can be implemented on a FPGA. Evaluation of the performance has shown that the column storage is a viable option to the row storage. Its key feature is its ability to push down the projection to the data which reduces the amount of data that it has to read from the disk. When less than half of the complete row is projected, the column storage can compensate its lower random performance and becomes faster than the row storage. Due to its adaptable design it can achieve a high and steady throughput independent of the number of projected columns.
The column storage presented is integrated into Ibex which is now able to push the projection down to the data. By reading less data from the disk the load on the FPGA and the host are reduced.
We expect that further advancements in SSDs will favor our approach even more, especially if our design is scaled up to support 16 or 32 NCQ-tags.

The second part of our work considered simple statistical operators on the FPGA. We have shown that thanks to the parallelism on FPGAs, it is possible to compute statistical data on the passing data stream without additional latency. In practice the computational complexity is limited by the FPGA, For example multiplication can be done with some additional effort while division on the FPGA is not possible. Nevertheless it should be possible to implement most statistical operators which use a limited amount of memory and can be applied to a data stream.

6.2 Future work

6.2.1 Scalability

Theoretically it should be possible to scale our design up to 16 or 32 double buffers and therefore taking advantage of the full NCQ queue depth. On our FPGA model this is a huge expense in terms of hardware resources but modern FPGAs have much more resources available and such that the hardware consumption becomes irrelevant.
Another way to scale the design and decrease the I/O bottleneck would be the use of a second or multiple disks. Theoretically it would be possible to achieve a 16 bit throughput per device and therefore scale linearly with the number of connected SATA-devices. An implementation that is able to handle multiple disks will increase the complexity of the design significantly. Since the FPGA can clearly process more data than 16 bit per clock cycle, this would be an interesting approach.

6.2.2 Compression

Compression is a common technique which is heavily used in column stores to reduce the I/O bottleneck even further. There is a trade-off between CPU time used for compression/decompression and query processing. To benefit from compression while keeping the necessary computation to a minimum, most systems use lightweight compression.
Since Ibex is a hybrid FPGA/CPU system, the compression would be executed on the FPGA and will not affect query processing on the CPU in any way. Therefore even heavy compression schemes might be an option. Due to the parallelism on the FPGA it should be possible to hide the additional latency and to decompress the data at line rate. Others [9] have shown how FPGAs can be used for compression.

6.2.3 Data Analytics

We only have shown the implementation of two simple statistical operators. The next step would be to explore more complex operators and how they can be implemented in a way that they can process the data at line rate. Especially data stream operators with a low memory usage are a good fit for FPGAs.
In-database analytics is a trend that recently emerged and because there is a huge potential benefit when these operators can be placed closer to the data. We certainly can expect more research in this area.

Acronyms

BRAM block RAM. 6

CLB Configurable Logic Block. 5

DMA Direct memory access. 14

DSP Digital signal processing. 7, 23

FPGA Field-programmable gate array. 5

HBA Host bus adapter. 14

NCQ Native command queuing. 7, 9, 10, 14, 27

SIRC simple interface for reconfigurable computing. 7, 8, 14

SoC System on a Chip. 5

References

[1] Mysql. `http://www.mysql.com`.

[2] Netezza. `http://www.netezza.com`.

[3] Serial ATA revision 2.6. `http://www.sata-io.org`, 2004. [Online].

[4] Gustavo Alonso, Donald Kossmann, and Timothy Roscoe. Swissbox: An architecture for data processing appliances. In *CIDR*, pages 32–37. www.cidrdb.org, 2011.

[5] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277, 2011.

[6] Ken Eguro. Sirc: An extensible reconfigurable computing communication api. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 135–138, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Woods Louis. Ibex - an intelligent storage engine to improve the energy efficiency and performance of databases. 2013.

[8] Chongling Nie. An FPGA-based Smart Database Storage Engine. Master's thesis, ETH Zurich, 2012.

[9] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 411–420, New York, NY, USA, 2012. ACM.

[10] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 59–72, New York, NY, USA, 2009. ACM.

[11] Louis Woods and Ken Eguro. Groundhog - a serial ata host bus adapter (hba) for fpgas. In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '12, pages 220–223, Washington, DC, USA, 2012. IEEE Computer Society.