Ibex—An Intelligent Storage Engine with Support for Advanced SQL Off-loading

Louis Woods Zsolt István Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Modern data appliances face severe bandwidth bottlenecks when moving vast amounts of data from storage to the query processing nodes. A possible solution to mitigate these bottlenecks is query off-loading to an *intelligent storage engine*, where partial or whole queries are pushed down to the storage engine. In this paper, we present *Ibex*, a prototype of an intelligent storage engine that supports off-loading of complex query operators. Besides increasing performance, *Ibex* also reduces energy consumption, as it uses an FPGA rather than conventional CPUs to implement the off-load engine. *Ibex* is a *hybrid* engine, with dedicated hardware that evaluates SQL expressions at line-rate and a software fallback for tasks that the hardware engine cannot handle. *Ibex* supports `GROUP BY` aggregation, as well as *projection*- and *selection*-based filtering. `GROUP BY` aggregation has a higher impact on performance but is also a more challenging operator to implement on an FPGA.

IBEX是一种混合引擎，其专用硬件以行速率计算SQL表达式，并为硬件引擎无法处理的任务提供软件备份。

1. INTRODUCTION

Data warehousing and a move to the more service-oriented business model of cloud computing have led to the rise of so-called *database appliances*. By combining hardware and software in a single closed box, vendors can carefully tune both aspects of the system for maximum efficiency. In these appliances, specialized hardware components together with large scale parallelism are often used to improve performance. Oracle's Exadata [23] or IBM's Netezza [17] are commercial examples of this trend. These systems use heterogeneous architectures to different degrees, all having the notion of an *intelligent storage engine* in common.

各种各样的

An intelligent storage engine is a specialized component that turns the classical storage engine of a database into an active element capable of processing query operators, allowing the database to off-load partial or entire queries to the storage engine for more efficient processing in terms of both *performance* and *energy consumption*. For instance, J. Do et al. have recently explored this idea [11], by using the processor inside SSD devices for query off-loading.

Executing an SQL query can be a fairly complex process, involving many different components of a DBMS, ranging from query parsing and query plan generation/optimization to actual data retrieval. For many of these tasks, the flexibility of a general-purpose CPU is needed. Nevertheless, *scanning*, *filtering* and *aggregating* large volumes of data at high throughput rates can be more efficiently implemented using dedicated hardware. Thus, a *hybrid* database system, composed of specialized hardware and commodity hardware can provide the best of both worlds.

分析

通用处理器完成此类功能

特定硬件完成此类功能

In this paper, we explore the implementation of an intelligent storage engine using FPGAs and SSDs. Our prototype, *Ibex* [34, 35], supports query off-loading to an FPGA-based accelerator. This first version of *Ibex* has been implemented as a pluggable storage engine for MySQL as a proof of concept, with the goal of creating an open research platform for hardware/software co-design in a database context. Compared with the little public information on commercial systems [17], we provide an exhaustive description of the architecture and design trade-offs behind intelligent storage engines that opens up interesting research directions. Moreover, *Ibex* supports more complex operators (such as multi-predicate `WHERE` clauses and `GROUP BY` aggregation) that are significantly more expensive when performed in software [26], and that have so far not been sufficiently addressed in both commercial [17] and related research systems [6, 7, 28]. In *Ibex*, these operators are implemented in an efficient and generic way.

从句

Contributions. *(i)* We present a first prototype of an *intelligent storage engine* implemented with FPGAs. Both the software and hardware code of *Ibex* will be released as *open source* to facilitate research in this field. *(ii)* We discuss several new techniques that advance the state of the art of SQL query processing in hardware. In previous work, only projection- and selection-based filtering is being pushed to specialized hardware [6, 17, 28]. By contrast, *Ibex* also evaluates complex `WHERE` clause expressions and `GROUP BY` aggregation queries at line-rate. *(iii)* Our experiments show several advantages of *Ibex* compared to well-established MySQL storage engines such as MyISAM and INNODB, both in terms of *performance* and *energy consumption*. *(iv)* Finally, our results prove that hardware accelerators can be integrated into a real DBMS, complex SQL operators can be pushed down and accelerated, and that an intelligent storage engine can improve performance, as well as lower power consumption.

Proceedings of the VLDB Endowment, Vol. 7, No. 11
Copyright 2014 VLDB Endowment 2150-8097/14/07.

2. RELATED WORK

Databases have a long history of exploring 特制的 tailor-made hardware, dating back to the idea of a database machine [8, 9] in the seventies. But in those times the rapid evolution of commodity CPUs rendered such approaches uneconomical. However, in the last ten years, after clock frequency scaling has more or less come to an end [4], hardware awareness has become an increasingly important topic. This has lead to a number of novel approaches that exploit specialized hardware to accelerate data processing, using, for example, GPUs [14,16], network processors [13], or FPGAs [20,21,25].

2.1 Intelligent Storage Engines

In the *big data* era, databases are spending an increasing amount of CPU cycles on scanning vast amounts of data. To produce a query result, often gigabytes of data are moved to the CPU although most of this data is either irrelevant to the final query result or will contribute to it only in the form of aggregates. Thus, *intelligent storage engines* such as in Oracle's Exadata [23] have recently been suggested to support early filtering of data to both increase query performance and reduce energy consumption. J. Do et al. [11] evaluated pushing query processing into *smart SSDs*, concluding that the idea has a lot of potential but that the hardware inside current SSDs is too limited, and that the processor quickly becomes a bottleneck. As another example, with the NDB[1] storage engine for cluster environments, MySQL changed the storage engine interface to allow pushing `WHERE` clause predicates down to the storage layer because moving entire tables between nodes of a cluster is too costly.

2.2 Data Processing with FPGAs

To accelerate data-intensive applications, FPGAs are interesting because they allow building custom hardware at relatively low development cost. Furthermore, FPGAs have the potential to improve performance and at the same time reduce energy consumption. It has already been shown that FPGA-based solutions 擅长 excel at a number of data processing tasks, *e.g.*, XML pattern matching [20], network intrusion detection [37], traffic control information processing [32], or algorithmic trading [24], to name a few. While these solutions demonstrate the potential of FPGAs, they all focus on accelerating only very specific tasks, not on supporting a general-purpose database engine.

2.3 SQL Acceleration with FPGAs

Initial approaches that used FPGAs for database tasks were mostly targeted at stream processing, *e.g.*, *Glacier* [21] is a compiler that translates SQL queries into VHDL code. Similarly, Takenaka et al. [29] developed a synthesis framework that generates *complex event processing* (CEP) engines for FPGAs from an SQL-based language. Stream processing applications typically have long-running, 长时间运行的常设查询 *standing queries*, which justifies invoking a time-consuming synthesis and reconfiguration step of the FPGA for every new query.

In contrast to stream processing, in a data warehouse scenario the query workload is unpredictable and queries are not always long-running. Thus, several minutes (or even hours) of synthesis time for every query is unacceptable. Dennl et al. address this problem in [6], by applying a special

[1]The *Network DataBase* (NDB) storage engine is used to enable the MySQL Cluster distributed database system.

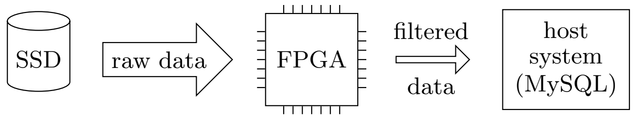

Figure 1: Data path architecture of *Ibex*.

FPGA technique called *partial reconfiguration*, which allows them to build query plans from pre-compiled components at runtime. Partial reconfiguration is useful to 时间复用 time-multiplex 多重电路 multiple circuits that would otherwise not fit on the same FPGA. In our case, however, the entire circuit fits onto one FPGA, and we use *runtime parameterization* instead to load new queries, using techniques similar to the ones proposed in several other systems [17,22,28].

2.4 State of the Art

In IBM's data warehouse appliance Netezza [17], simple selection- and projection-based filtering is pushed to multiple parallel FPGAs. However, more complex operations such as `GROUP BY` aggregation are handled by the CPU. Two different solutions to handle multi-predicate `WHERE` clause expressions in an FPGA have been proposed in [6,28]. However, both approaches have unnecessary limitations, as we show in Section 4.2, where we present our solution. The only attempt to accelerate `GROUP BY` aggregation queries with FPGAs that we know of is [7] but note that the proposed approach only works with pre-sorted tuples. In this paper, we present a novel algorithm for implementing `GROUP BY` aggregation in an FPGA that does not require sorting. Furthermore, we provide a holistic solution for both `GROUP BY` and multi-predicate `WHERE` clauses that also handles the case where the intermediate state of the operator exceeds the capacity of an FPGA.

3. SYSTEM OVERVIEW

To give a high-level overview of the system, we first discuss physical architecture options before we cover challenges of integrating an FPGA with a database, such as interfacing with the DBMS, operator pushdown, accessing data via FPGA, and the implementation of an FPGA driver.

3.1 Architecture Options

FPGAs can be integrated into a system in many ways, *e.g.*, via PCIe [28], Ethernet [17], or even by putting the FPGA in a socket using the frontside bus [5]. The way the FPGA is integrated determines the type of data processing that can later be efficiently performed using that FPGA.

Explicit 显式协处理器 **Co-Processor.** One option is to incorporate the FPGA as a PCIe-attached co-processor, *i.e.*, the same way GPUs are typically integrated. This is done in systems like Microsoft's Cipherbase [1][2] and IBM's database analytics accelerator [28]. The drawback is that data needs to be copied to and from the FPGA *explicitly*. Thus, it only pays off to use the co-processor for workloads that are compute-intensive enough to justify the data transfer. 计算紧密类数据更高效

[2]Note that in Cipherbase [1] data confidentiality and not performance is the goal, as the FPGA is used as a trusted computing base and not as an accelerator.

隐式协处理器

Implicit Co-Processor. Alternatively, the FPGA-based co-processor can be inserted in the *data path* between storage and host CPU, as illustrated in Figure 1. The FPGA circuits can be designed to operate on the data in a stream processing manner such that routing data through the FPGA does not hurt throughput and adds only negligible latency. The advantage of this design is that there are no additional transfer costs since the FPGA only operates on data that is being transmitted to the CPU anyway. In *Ibex*, we are interested in this type of integration and place the FPGA between SSD and MySQL (cf. Figure 1). An interesting alternative would be to insert an FPGA directly into the memory bus between CPU and main memory, applying the same techniques that we present here to main memory databases. We leave this option to future work.

3.2 Integration Challenges

3.2.1 Interfacing with the DBMS

To extend a database with an intelligent storage engine, the parts of the code base that communicate with the disk driver need to be replaced by code that interfaces with the FPGA. MySQL features a *pluggable* 可插拔 storage engine (since version 5.1) that allows multiple storage engines to co-exist in a single database instance, and even combining different storage engines. For example, in an existing database, one could migrate several tables, for which query off-loading makes sense, to *Ibex*, and leave other tables unchanged. While the migrated tables now would benefit from hardware acceleration, higher-level operations like joins across tables associated with different engines would still be possible.

MySQL implements the Volcano iterator model [15]. The query processor communicates with its storage engines at a tuple granularity, *e.g.*, when a table is scanned the query processor repeatedly calls `rnd_next(...)` until the storage engine does not return any more tuples. Inserting into a table follows a similar pattern. Hence, the storage engine has complete freedom as of where to fetch or store tuples, *e.g.*, on a local storage media or—as in the case of *Ibex*—via database-aware hardware implemented on an FPGA.

3.2.2 Operator Pushdown

The main purpose of the storage engine is to translate pages stored on disk into tuples (and vice versa), *i.e.*, query processing typically takes place in the upper layers of a database. However, as mentioned earlier, MySQL introduced a mechanism to push `WHERE` clause conditions to the storage engine. In *Ibex*, we take this approach one step further—we have extended the existing MySQL storage engine interface to also allow pushing projection and `GROUP BY` aggregation down to the storage engine.

To implement SQL operators on an FPGA, an important challenge is to find the right balance between flexibility and performance [24]. To give an example, best performance is achieved if we implement each query on the FPGA as a hard-wired 无法改变的 circuit. However, this approach would allow us to only execute a limited set of known queries. On the other hand, the most flexible solution would be to implement a microprocessor on the FPGA to execute queries, resulting in a slow solution because all of the benefits of dedicated hardware would be lost. *Ibex* is in between these two extremes, where for every component we had to carefully decide how to divide the work between the flexible CPU on the host and

要性能：失去灵活性，仅能实现有限的查询功能

要灵活性：失去专用硬件的优势

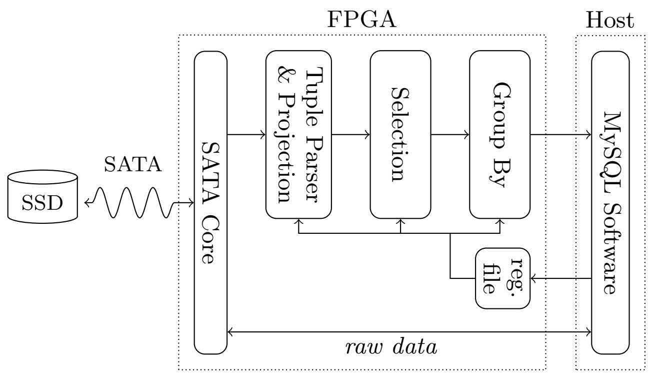

Figure 2: Hardware SQL engine embedded in the data path between an SDD and MySQL, supporting block-level access (raw data) and a tuple-level query pipeline, which is parameterizable via register file.

our FPGA engine to get the best of both worlds. Furthermore, all major components (Parser, Selection, `GROUP BY`) are runtime-parameterizable to the right degree to allow us to execute a wide range of queries without costly reconfiguration of the FPGA, as will be discussed in more detail in Section 4.

3.2.3 Data Access Modes

In *Ibex*, the FPGA has a direct SATA link to an SSD. The hardware engine transfers data to/from the SSD via a SATA IP core instantiated 实例化的 on the FPGA. Thus, the hardware engine has to operate on blocks of raw data. In answering block requests for an upstream system, it is not immediately clear how an accelerator could filter or modify individual tuples within those blocks.

Tuple-Level versus Block-Level Operation. The semantics of any filtering or aggregation task have to be expressed on the tuple level, which is why these tasks are typically handled above the storage engine. To let the hardware engine perform query processing tasks, the *mismatch* problem between block-oriented disk access and the tuple-oriented semantics of these tasks needs to be solved. Since *Ibex* is designed for hybrid query processing, the *Ibex* hardware supports both *block-level* and *tuple-level* access to base tables. Figure 2 illustrates on a high level how we designed the hardware part of *Ibex* internally to support block- *and* tuple-level access modes.

Tuple-Level Access. When sub-queries are off-loaded to the FPGA, the hardware switches to a tuple-based interface, *i.e.*, disk blocks are parsed in hardware and processed by a parameterizable query pipeline, and the *result tuples* are forwarded to the host system as a sequence of tuples, which are directly fed into the query evaluation pipeline of the database. Tuples provide the right abstraction here in a Volcano-style execution engine like the one of MySQL.

Block-Level Access. In all other cases, data is accessed in the conventional block-oriented mode, *i.e.*, using the *raw data* path in Figure 2. This includes not only un-predicated table scans, as in the above example, but also any operation that does not require hardware acceleration can use block-level access just like an off-the-shelf 现成的 system would, *e.g.*, update operations, maintenance tasks (backup/recovery, etc.),

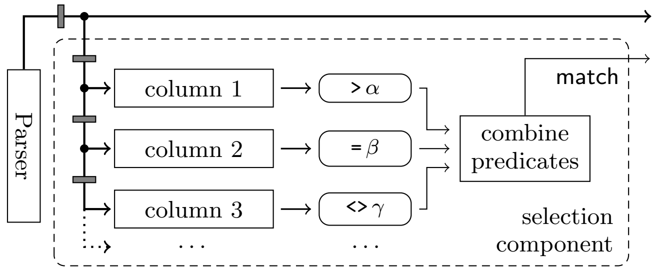

Figure 3: Selection component with three base predicate circuits parameterized for the expression WHERE $(\text{col}_1 > \alpha$ AND $\text{col}_2 = \beta)$ OR col_3 <> γ.
不等于

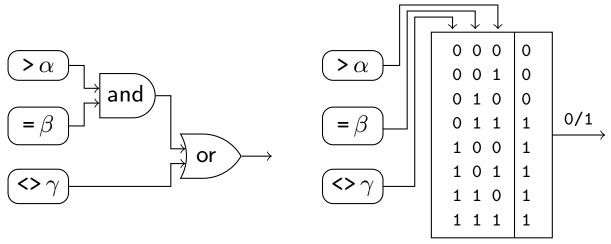

Figure 4: Hard-wired circuit (left) for combined predicate WHERE $(\text{col}_1 > \alpha$ AND $\text{col}_2 = \beta)$ OR col_3 <> γ **versus truth table approach (right).**

or index-based plans. However, note that the focus of this paper is the tuple-level access path, for which we consider solely read-only workloads, at this stage.

3.2.4 FPGA Driver

For communication with the FPGA, we implemented a driver, which is accessible from within the MySQL source code. The communication abstraction is based on three fundamental components that reside on the FPGA: *(a)* an input buffer, *(b)* an output buffer, *(c)* and a bi-directional register file. 双流向的 C++ functions allow writing to the input buffer and reading from the output buffer, as well as reading and writing individual registers of the register file. The incentive is to use the input and output buffers to transfer data between host system and FPGA, while the register file is used to control the hardware accelerators. Under the hood, 在底层 communication to the FPGA is implemented over Gigabit Ethernet. Unfortunately, Gigabit Ethernet provides less bandwidth (125 MB/s) than SATA II (300 MB/s), *i.e.*, with fast SSDs the Ethernet bandwidth could become the bottleneck if the FPGA does not filter out enough data. However, this limitation is due to the FPGA board that we are currently using. In a production system, a higher bandwidth connection is realistic, *e.g.*, PCIe, 10G Ethernet, or InfiniBand.

4. QUERY PIPELINE

This section covers the three main components enabling query processing on the FPGA (cf. Figure 2): *(1)* Parsing & Projection, *(2)* Selection, and *(3)* GROUP BY Aggregation.

4.1 Parsing and Projection

Before fetching a table from disk, the corresponding catalog information is loaded from the host into a small local RAM on the FPGA. For each column, we also store a *projection flag* (that indicates whether the column is part of the projection). We then use standard techniques to parse the raw data stream coming from the SATA link at line-rate and annotate it with catalog information. Furthermore, an additional signal is generated, indicating which parts of the data stream are part of the projection.

4.2 Selection-based Filtering

After the parsing stage, the annotated 经注释的 data stream is loaded into the selection component. The selection component processes this data in a pipelined manner and forwards it to the GROUP BY component. In the selection component an additional signal (match) is generated (cf. Figure 3), which is set to logic high, whenever a tuple matches a given WHERE clause. Subsequent components will use this signal to ignore tuples not satisfying the WHERE clause condition. For queries without a WHERE clause, match will be set to logic high for every tuple.

4.2.1 Base Predicates

The selection component can be configured to support n_p base predicates. We refer to a *base predicate* as simple comparison between a column value and a constant, *e.g.*,

$$\texttt{WHERE}\ \textit{column}\ \theta\ \textit{constant}\ .$$

All three parts of every base predicate are *parameterizable*, *i.e.*: *(1)* the column, which is set by specifying the corresponding *column ID*, *(2)* the comparator, for which one out of six possibilities is selected ($\theta \in \{=, <>, <, >, <=, >=\}$), and *(3)* a constant value. Currently, our selection component only supports predicates on fixed-length columns such as integers. That is, base predicates with string comparisons still need to be handled in software. However, one could implement such functionality on an FPGA in a streaming manner, similar to how Teubner et. al implemented XPath matching [30]. Thus, the higher-level architecture would not change much if we added this functionality.

Data is loaded into each base predicate circuit in a pipelined fashion, *i.e.*, the data stream passes by all base predicate circuits, as illustrated in Figure 3. If the column ID of the annotated stream matches the *column ID parameter* of a particular base predicate circuit, the corresponding part of the data stream will participate in the comparison of that base predicate. This even allows evaluating several base predicates on the same column, *e.g.*, WHERE clause expressions of the following form are also possible:

$$\texttt{WHERE}\ \text{col}_1 = \alpha\ \texttt{OR}\ \text{col}_1 = \beta\ .$$

Hence, we can evaluate n_p arbitrary base predicates in parallel, independent of which columns will participate in the comparisons. Notice that n_p only limits the number of base predicates that can be handled by the FPGA, n_p does *not* in any way limit the number of columns that an *Ibex* table can have, nor does it constrain queries to only consist of n_p base predicates—for more complex WHERE clauses there still exists the option of *partial filtering*, *i.e.*, only the part of the WHERE clause that fits on the FPGA is handled in hardware, and the rest of the expression is evaluated in software.

4.2.2 Combined Predicates

Combined predicates are composed of *base predicates* and connected via Boolean operators, *e.g.*:

$$\texttt{WHERE}\ (\text{col}_1 > \alpha\ \texttt{AND}\ \text{col}_2 = \beta)\ \texttt{OR}\ \text{col}_3 <> \gamma\ .$$

To implement combined predicates in hardware, intuitively we would like to use logic AND and OR gates, as illustrated on the left of Figure 4. However, it would take too long to synthesize the corresponding hard-wired circuit at runtime.

Existing Approaches. In [6] the WHERE clause expression is dynamically mapped to an operator pipeline using partial reconfiguration. Unfortunately, "spare chunks" need to be inserted into every tuple beforehand so that intermediate results can be stored. The number of spare chunks depends on the complexity of the WHERE clause, and it is not clear how these chunks are inserted at runtime. Moreover, inserting spare chunks into a data stream reduces its actual bandwidth. In [28] a hard-wired reduction tree with parameterizable operators (tree nodes) is instantiated. However, this tree introduces additional latency, *i.e.*, for an n-byte long tuple, it takes $n + log_2(\#predicates)$ cycles to qualify a tuple. Furthermore, no details are given of how to map an arbitrary WHERE clause expression to such a tree.

Truth Tables. Fortunately, a much simpler solution exists to support combined predicates, which does not have the shortcomings discussed above. Namely, we can use a method of parameterization that resembles the implementation of the FPGA hardware itself: lookup tables. With n_p base predicates, there can be at most 2^{n_p} different evaluation results for those base predicates. Thus, 2^{n_p} bits of storage are sufficient to materialize the output of the operator tree for every possible input bit vector. Note how this is independent of the complexity of the Boolean expression.

In *Ibex*, we use on-chip *block memory* (BRAM) to store operator trees as explicit *truth tables*. Each base predicate circuit has one output signal that carries the result of the base predicate evaluation. The output signals of all base predicates together compose the lookup address in the truth table, and are directly connected to the respective BRAM block, as depicted on the right of Figure 4.

A single BRAM block can hold up to 36×2^{10} bits of data, which is enough to store truth tables for up to $n_p = 15$ base predicates. To support $n_p > 15$ there are several possibilities. First of all, multiple BRAM blocks can easily be combined to form larger BRAMs, using the Xilinx tool chain. However, BRAM consumption grows exponentially with the number of predicates. Therefore, a better solution would be to split the Boolean expression of the WHERE clause into groups and assign each group a separate smaller truth table. The results from those truth tables could then be combined via lookup in a subsequent, higher-level truth table.

Generating the Truth Table. Before executing a query, the *Ibex* software computes the truth tables values corresponding to the operator tree of the WHERE clause, and then loads them into the corresponding BRAM on the FPGA. Computation of the truth table in software is simple. First, the Boolean expression is converted into *disjunctive normal form* (DNF). 析取范式 Then, starting with a truth table that has all bits set to zero, the corresponding bits are set to one iteratively for every conjunctive clause. Finally, the truth table is transmitted to the FPGA, which can also be done efficiently, *e.g.*, transfer time of 36 Kbit (for $n_p \leq 15$) over Gigabit Ethernet is only $36\mu s$.

4.3 GROUP BY Aggregation

The GROUP BY component handles *grouping*, using a specially designed hardware *hash table*, the implementation of which we are going to discuss separately, in Section 5. For now, we treat the hash table as a black box and focus on the GROUP BY component on a high level. A typical GROUP BY query is illustrated below.

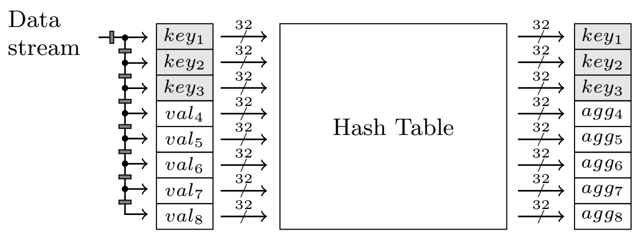

Figure 5: High-level architecture of the GROUP BY component. Data is loaded in a pipelined fashion (pipeline registers → ▯).

```
    SELECT col2, col7, MAX(col1), MIN(col1)
      FROM table                                  (Q1)
  GROUP BY col2, col7;
```

标准
The grouping criteria that defines the individual groups is specified in the GROUP BY clause. This can be a single column or a combination of several columns, as in Q_1. In the following, we refer to this grouping criteria as the *group key*.

In the presence of a GROUP BY clause, only columns that are part of the GROUP BY clause can be projected without an aggregation function, all other columns need to be part of an aggregate since for every group there will be a single result tuple. Notice that the same column can appear in multiple aggregates, *e.g.*, MAX(col$_1$) and MIN(col$_1$) in Q_1.

Group Keys and Aggregates. The high-level design of the GROUP BY component is depicted in Figure 5. After the *selection* stage, relevant data is loaded into a wide input buffer (here, 256 bits wide), which is divided into multiple 32-bit slots. If the match signal from the selection component indicates that a tuple did not satisfy the WHERE clause, the data in the buffer will be invalidated and overwritten by the next tuple.

At runtime, a bit mask determines how many slots belong to the group key and how many slots are used for aggregation. In Figure 5, the first three slots are used for the group key and the remaining five for aggregation. Notice that we do not require all slots to be used. It is perfectly valid to use, say, only the first two slots, one for the group key, and the other for a single aggregate. Our design exhibits flexibility not only to support combined group keys, as in Q_1, but also group keys on columns that are wider than 32 bits. For group keys that are smaller than 32 bits, we simply add a padding of zeros to the 32-bit slot.

Input Buffer Loading. Besides assigning slots for *grouping* and *aggregation*, each slot can be mapped to a column at runtime. Furthermore, also the type of aggregation (COUNT, SUM, MIN, MAX)[3] can be set for every slot. Data is loaded into the slots in a pipelined fashion, using the same technique that we described earlier, in Section 4.2.1, to load data into the base predicate circuits of the selection component. This allows us to compute multiple aggregates on the same column. Moreover, even the order of aggregates can be specified already in hardware, *i.e.*, reordering in software is not

[3]For the average, we compute SUM and COUNT in hardware, and perform the division AVG ≡ SUM/COUNT in software.

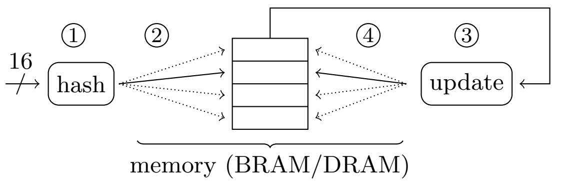

Figure 6: Abstract view of our hash table with the four fundamental operations: ① *hash*, ② *read*, ③ *update*, and ④ *write*.

necessary. Another advantage is the scalability of the technique. Here, we set the buffer width to 256 bits (eight slots), because it matches the DRAM word-width used in the hash table later. However, wider buffers to support more aggregates are also possible since the pipelined loading mechanism easily scales to wider buffers, without causing timing or routing problems on the FPGA.

Hash Collisions and Bypass Tuples. Once the `match` signal from the selection component asserts that the data in the input buffer is valid, all slots are read out from the buffer in parallel. The group key slots are used to probe 探查 the hash table, upon which the hash table returns an entry the size of the input buffer, containing the group key, as well as the current running aggregates. All aggregates are updated in parallel and the entry is written back to the hash table.

As discussed in more detail in Section 5, we only detect but do not resolve hash collisions in hardware. When a colliding tuple is detected we simply *bypass* the hash table and forward it to the host instead. Thus, during query processing *bypassed* tuples of the form $\{key_1, \dots, key_i, val_{i+1}, \dots, val_n\}$ are forwarded to the output buffer. After the entire table has been read, the hash table contents are flushed and aggregated tuples of the form $\{key_1, \dots, key_i, agg_{i+1}, \dots, agg_n\}$ are forwarded to the output buffer.

5. GROUP BY WITH HASHING

At the heart of our `GROUP BY` component is a hardware implementation of a hash table. While hash tables on FPGAs have been studied in prior work [10, 18, 27, 31], the hash table design we present here is specially tailored 特制的 at supporting `GROUP BY` queries at line-rate. The hash table allows us to execute `GROUP BY` aggregation in a single pass, without having to first sort the input data. Figure 6 illustrates how this works: ① First, the group key is hashed. ② The bits of the hash value (or a subset thereof) serve as the memory address, and the corresponding memory word is read from that address. ③ A special flag indicates whether we are processing a particular group for the first time. If this is the case, we perform an *insert*, *i.e.*, we completely overwrite the just read memory word. Otherwise, we simply *update* the running aggregates in the memory word. ④ Finally, the processed memory word is written back to memory.

As mentioned previously, our goal is to achieve line-rate performance for `GROUP BY` aggregation inside the FPGA (*i.e.*, 300 MB/s or 16 bits per clock cycle at 150 MHz). However, to do so we face several challenges. First of all, constant time lookup is only guaranteed if there are no *hash collisions*, *i.e.*, if no two groups map to the same memory address. Instead of *stalling* the input stream to perform collision resolution on the FPGA, we choose to *bypass* colliding tuples to host software, and implement a first-come-first-served policy in our hash table. Getting rid of collision resolution, however, only partially solves the problem since the four steps depicted in Figure 6, each require at least one clock cycle. Performing these four steps in a sequential way would again cause stalling of the input stream. Therefore, we propose a fully-pipelined hash table that hides these latencies.

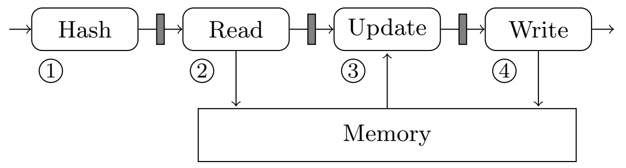

Figure 7: Pipelined hash table hides latencies, allowing concurrent processing of four tuples.

5.1 Fully-pipelined Hash Table

The pipelined version of the hash table is displayed in Figure 7. The four stages *hash*, *read*, *update*, and *write* can now be executed in parallel for different tuples in the pipeline. The number of clock cycles that we are allowed to spend in each stage depends on the size of the tuples in the database table—the larger the tuples the more time we have. The smallest tuple that our database engine supports is 32 bits wide. This means that in the most extreme case we have only two clock cycles in every stage.

The *multiplicative* hash function that we use is itself fully pipelined, and consumes 16 bits per clock cycle matching the input rate. Using BRAM, reading and writing each take one cycle (we will discuss the DRAM case separately). Finally, updating all the aggregates of a tuple is done in parallel and can also be handled in a single cycle. Thus, with BRAM we can guarantee line-rate processing. Nevertheless, care needs to be taken when two tuples of the same group follow closely after each other since this could lead to *data hazards*.

5.2 Avoiding Data Hazards

When two consecutive tuples access the same memory location because they belong to the same group, it is essential that the second tuple sees the modifications of the preceding one, otherwise updates will get lost. The design in Figure 7 cannot guarantee this when tuples are small.

One approach is to detect situations where potential data hazards could occur, and then stall the pipeline for an appropriate amount of time. However, such stalling is only acceptable if it happens infrequently. Unfortunately, tuples of the same group stored close together is not an unlikely scenario. Moreover, pre-sorted tables would exhibit the worst performance, which is counter-intuitive.

To solve this problem, we introduce a caching layer between memory and the pipelined hash table. This layer implements a *write-through cache* that holds the n last writes to memory. As shown in Figure 8, all read requests are logged temporarily in a queue, and the following writes to memory are cached during the *write* stage. When a new read is performed, the address of the key is first checked in the queue of recently accessed memory locations. For this purpose, the queue exposes a CAM-like[4] interface for read-

[4]Content-addressable memory (CAM) is a storage device in which the information is identified by content rather than by an address.

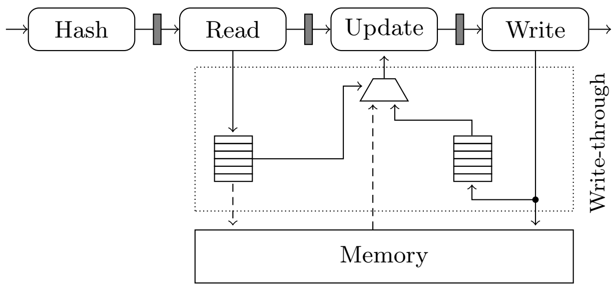

Figure 8: Pipelined hash table with *write-through cache* to avoid data hazards.

ing, which returns the index inside the queue of the most recent access to the memory address in question. Using this index, the write cache, holding the data recently written to memory, can be accessed. Thus, in the *read* stage, actual read requests to memory are only issued for memory addresses that have *not* been accessed in the recent past. The logic in the *read* stage then instructs the subsequent *update* stage to either fetch the next tuple from the write cache, or wait for it to be delivered from memory.

5.3 DRAM-based Hash Table

Using BRAM to store the hash table has the advantage of very low access latencies but BRAM is also a 稀缺的 scarce resource on the FPGA. A hash table that can store a few thousand groups in BRAM takes up a big portion of the available BRAM blocks. Thus, for workloads with many different groups, using on-board DRAM (in the order of hundreds of megabytes) could be a preferable option.

Thanks to the caching layer, discussed above, DRAM can be used instead of BRAM transparently with our hash table pipeline. The only required modification to the hash table logic is the correct sizing of the memory access cache. The size of the cache depends on the actual memory latency, because items in the cache can be evicted only once they have been written to memory for effective protection against the data hazards discussed earlier. For instance, with BRAM a capacity of eight is sufficient, while with the DRAM on our platform the cache needs to hold at least 32 entries.

A side effect of the caching layer for the DRAM-based hash table is that it not only helps avoiding data hazards but also may increase performance. With DRAM it is not always possible to completely hide the memory latency since it is significantly higher than for BRAM but the caching layer at least mitigates the impact of memory read latency.

6. PERFORMANCE EVALUATION

In this section, we compare the performance of MyISAM and INNODB (two common MySQL storage engines) to *Ibex*, running the same queries with each engine, and comparing the execution times reported directly by MySQL. Our main focus is to thoroughly evaluate one of the key contributions of this paper—the `GROUP BY` acceleration component. Furthermore, at the end of this section, we also show experiments that illustrate performance gains of selection-based filtering and how *Ibex* can impact more complex queries such as those in the TPC-H benchmark.

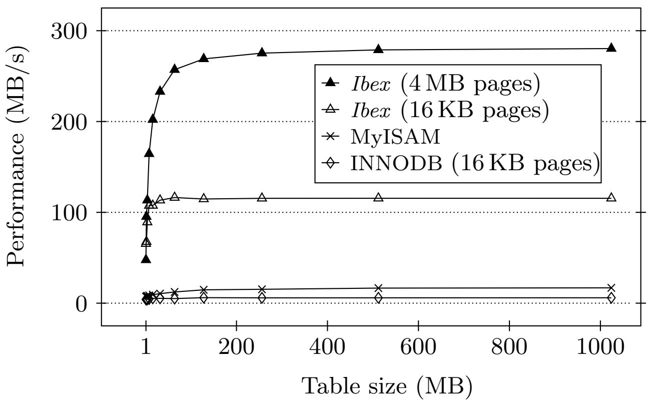

Figure 9: Performance ($^{\text{table size}}/_{\text{execution time}}$) of *Ibex* versus MyISAM and INNODB for table sizes ranging from one to 1024 megabytes.

6.1 Experimental Setup

All experiments were conducted on a Desktop PC featuring a Quad-Core Intel (i7-2700K, 3.50 GHz) processor with 8 GB of main memory. We ran MySQL 5.5.25 on a 64-bit Windows 7 platform, which was installed on a 256 GB OCZ Vertex 4 SATA III 2.5" SSD. An identical SSD was connected directly to the FPGA (Virtex 5, XC5VLX110T)[5].

6.2 GROUP BY Aggregation Queries

In this first experiment, we want to compare how fast MyISAM, INNODB, and *Ibex* can execute `GROUP BY` queries. In particular, we want to show how far MyISAM and INNODB are from SATA II (300 MB/s) line-rate query processing. To this end, we ran the following simple `GROUP BY` query on a synthetic workload:

$$\begin{array}{r}\texttt{SELECT col}_1\texttt{, COUNT (*)}\\ \texttt{FROM table}\\ \texttt{GROUP BY col}_1\texttt{;}\end{array} \qquad (Q_2)$$

We varied the table size between one and 1024 megabytes and every table always consisted of exactly 16 groups, each containing an equal amount of tuples stored in unsorted order. In Figure 9, on the x-axis we plot the table size and on the y-axis performance as $^{\text{tablesize}}/_{\text{executiontime}}$.

MyISAM stores tables in files managed by the operating system, *i.e.*, a notion of database pages does not exist. By contrast, INNODB has all the bells and whistles of a full-fledged storage engine, including a buffer pool to cache database pages, with a default page size of 16 KB[6].

As can be seen in Figure 9, MyISAM performs better than INNODB. In all of our experiments, MyISAM always exhibited better performance than INNODB, which is due to its simplicity compared to INNODB. For example, MyISAM does not support database transactions and therefore

[5]The OCZ Vertex 4 SSD is SATA III compatible but we used it exclusively in SATA II mode since our FPGA only supports SATA II, allowing a theoretical maximum bandwidth of 300 MB/s.

[6]According to the documentation, by recompiling the code, one can set the page size to values ranging from 8 KB to 64 KB. However, this is not officially supported, and for our version of MySQL, we could not compile the INNODB code with pages larger than 16 KB.

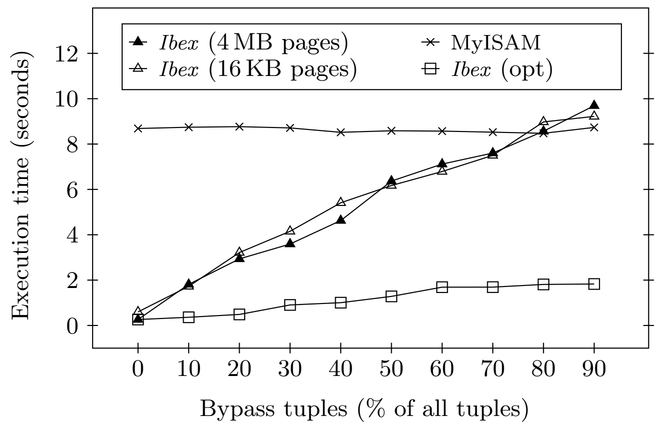

Figure 10: Impact of bypasses on execution time.

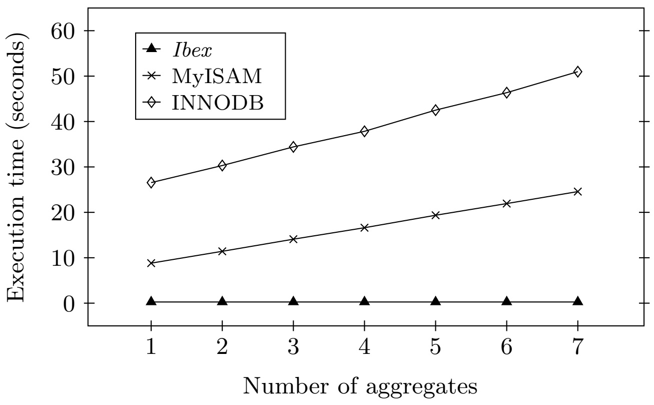

Figure 11: Varying the number of aggregates.

employs a simpler locking mechanism. Since our goal is to compare performance of *Ibex* against the fastest common MySQL storage engines, we sometimes omit INNODB results in the following plots for readability reasons.

Ibex, configured to use 16 KB pages (△), performs significantly better than both MyISAM (✕) and INNODB (◇). However, the throughput is still far below the 300 MB/s of SATA II. Throughput can be increased by using larger database pages. Thus, *Ibex* with 4 MB pages (▲) pretty much saturates the SATA II link, and shows that *Ibex* can sustain SATA II wire speed. Though the maximum bandwidth of SATA II is 300 MB/s, note that actual transfer rates of stored data are around 280 MB/s due to protocol overhead and latencies within the SSD.

6.2.1 The Impact of Bypass Tuples

The previous experiment assumed that the entire GROUP BY aggregation could be off-loaded to the FPGA. However, for other workloads hash collisions may occur, or the predetermined size of the hash table may be chosen too small to hold all groups. Thus, we need to quantify how *bypass* tuples impact performance. For this purpose, we ran query Q_3

```
SELECT col1, SUM ( col2 )
   FROM table                                  (Q3)
GROUP BY col1;
```

on a table from the previous experiment with modified group keys for each run to produce a varying number of collisions and bypass tuples.

If there are bypass tuples we need to do a separate GROUP BY and aggregation step in software. Our first attempt to achieve this was to rewrite query Q_3 as follows for the FPGA case:

```
SELECT col1, SUM ( s )
   FROM (SELECT   col1, SUM ( col2 ) AS s
         FROM     table                        (Q'3)
         GROUP BY col1) AS t1
GROUP BY col1;
```

The inner query is executed by the *Ibex* storage engine, returning a result table that contains bypass tuples, as well as partial aggregates. The outer query, on the other hand, is evaluated completely by the MySQL query processor. The results are depicted in Figure 10. The y-axis shows execution time, and the x-axis displays the percentage of tuples that are bypassed and aggregated in software.

MyISAM. When running query Q_3 with MyISAM, there are of course no bypass tuples but since we modified the group keys for every run, we show a separate measurement for each workload also for MyISAM. Not surprisingly, execution time is relatively constant for all workloads and takes roughly 8.5 seconds (✕).

***Ibex*.** When running query Q'_3 with *Ibex*, the execution time depends on the number of bypass tuples. With no bypasses and 4 MB pages (▲) query execution takes only 0.26 seconds, which is 32 times faster than MyISAM. Execution time then increases linearly with respect to the number of tuples bypassed. At 90% bypassed tuples, the performance of *Ibex* is slightly worse than that of MyISAM since query Q'_3 actually consists of two queries, while query Q_3 is a single query.

We ran query Q'_3 both with 4 MB pages (▲) and 16 KB pages (△). Figure 10 shows that the page size here does not significantly affect execution time. Hence, execution time is dominated by the overhead of software. We confirmed this observation also using the *query profiler* built into MySQL.

***Ibex*: Handling Bypass Tuples Natively.** To avoid the high cost of running the outer query of Q'_3 in MySQL we extended *Ibex* to deal with bypass tuples directly in the software part of the storage engine such that we could run query Q_3 using *Ibex* even with bypass tuples. To this end, we used a software hash table to deal with the bypass tuples, which lead to a significantly more efficient implementation, depicted in Figure 10 (⊟). Thus, even the workload with 90% bypass tuples executed in 1.83 seconds, which results in a speedup of roughly 4.5X, compared to MyISAM. This performance difference is due to known inefficienes (*e.g.*, high tuple interpretation overhead and low IPC efficiency) of Volcano-style database engines such as MySQL [3].

6.2.2 Increasing the Number of Aggregates

In software, not only *grouping* is an expensive operation but also the computation of aggregates itself. This can be seen by running query Q_4 with a varying number of SUM() aggregates.

```
SELECT col1, SUM ( col2 ), SUM ( col3 ), ···
   FROM table                                  (Q4)
GROUP BY col1;
```

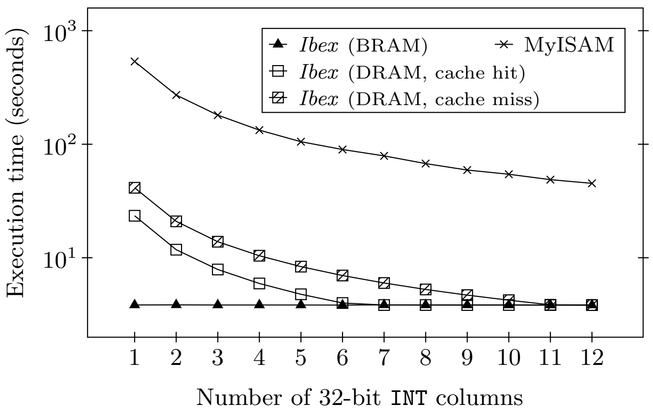

Figure 12: Varying tuple width. Wider tuples allow for better hiding of memory latencies.

We executed this query on a table with eight INT columns and two million rows, and computed between one and seven SUM() aggregates. The results, plotted in Figure 11, show how total execution time significantly increases with every additional aggregate when run with MyISAM (⋇) or INNODB (◇). The added overhead for every additional aggregate is similar for both MyISAM and INNODB, which is expected since the aggregates are computed outside of the storage engines. By contrast, execution time remains constant in the case of *Ibex* (▲) because each aggregate is computed by a separate parallel unit on the FPGA.

A similar effect, though not quite as pronounced, can be observed when several columns are used to form the *group key*, *i.e.*, execution time remains constant with *Ibex*, whereas it increases for MyISAM and INNODB linearly with the complexity of the combined keys.

6.2.3 BRAM- versus DRAM-Hash Table

To analyze the performance characteristics of the BRAM-based hash table versus the DRAM-based hash table, we varied the number of columns of a 1 GB database table and then executed query Q_2 on it. Hence, a 1 GB table with a single-column consisted of 268,435,456 rows (1 GB ÷ 4 bytes), whereas a 1 GB table with eight columns consisted of only 33,554,432 rows (1 GB ÷ 32 bytes).

BRAM Version. As can be seen in Figure 12, execution time of the BRAM version (▲) is independent of the table schema and is determined solely by the amount of data transferred. The execution time is constant, with 3.8 seconds for a 1 GB table corresponding to roughly 280 MB/s.

DRAM Version. The DRAM version behaves differently. For narrow tables, performance is worse than that of the BRAM version. The reason is that DRAM can handle less operations per second than BRAM due to the higher latency of DRAM. For wide tables, on the other hand, our pipelined hash table architecture can completely hide the DRAM latency.

Write-through Cache. Depending on whether there is a cache hit or a cache miss in the *write-through cache* of the hash table, each tuple causes one DRAM access or two, respectively. To measure the impact of the write-through cache we generated a workload with little enough groups

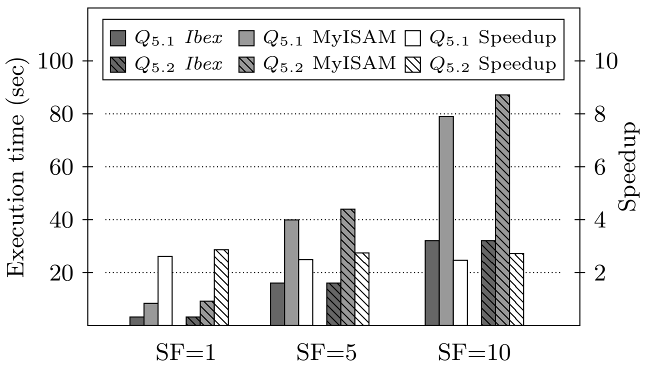

Figure 13: Performance of filter queries on TPC-H lineitem table for scale factors (SF) one, five and ten.

that the cache is always hit (⊟), as well as a workload, where the cache is never hit (⊠), *i.e.*, every tuple needs to first read from DRAM and then write back to it. In the worst case (a single-column table), the query executes in 41.3 seconds when the cache is always missed, and about twice as fast (23.4 seconds) when the cache is always hit. In the former case, line-rate speed is reached at eleven columns (44 bytes wide tuples), whereas in the later case already at seven columns (28 bytes wide tuples).

MyISAM. For comparison, we also plotted MyISAM performance (⋇) for the same query and workloads. The execution time is substantially higher than that of both FPGA versions (BRAM and DRAM). Observe that also for MyISAM the table schema matters. This is because the per-tuple overhead in MySQL is significant [3]. Thus, for the same amount of data, more tuples result in slower execution. With MyISAM, on the single-column table query Q_2 executed in 535.6 seconds and on the 12-column table in 45.2 seconds. Thus, we could always measure a speedup of at least one order of magnitude compared to MyISAM, even for workloads where the write-through cache was ineffective.

6.3 Filter Queries

Running GROUP BY queries with *Ibex* benefits from two effects: *(1)* filtering and *(2)* an efficient GROUP BY implementation in hardware. In this section, we discuss the impact of pure filtering queries. To do so, we ran query $Q_{5.1}$ and query $Q_{5.2}$ below on the TPC-H lineitem table. $Q_{5.1}$ invokes a simple selection filter (a single predicate on a date field), as well as a projection filter that keeps only three out of a total of 16 columns. $Q_{5.2}$, on the other hand, has a more complex WHERE clause but the same projection filter. Both queries also have similar selectivity, *i.e.*, the filtering effect is the same but the evaluation complexity is higher for $Q_{5.2}$.

```
SELECT l_orderkey, l_shipdate, l_linenumber
  FROM lineitem                                  (Q5.1)
 WHERE l_shipdate = '1995-1-17'
```

```
SELECT l_orderkey, l_shipdate, l_linenumber
  FROM lineitem
 WHERE (l_shipdate = '1995-1-17' OR
          l_shipdate = '1995-1-18')              (Q5.2)
       AND
       (l_linenumber = 1 OR l_linenumber = 2)
```

In Figure 13, we show the results. On the y-axis execution time and speedup are displayed and on the x-axis we show the scale factor used to generate workloads of different sizes for both queries. Since *Ibex* evaluates `WHERE` clause expressions at line-rate, independent of their complexity, there is no performance difference between query $Q_{5.1}$ and query $Q_{5.2}$. Conversely, for MyISAM and INNODB the more complex query $Q_{5.2}$ causes a slight decrease in performance because evaluating the complex `WHERE` clause requires more instructions. Nevertheless, for selection and projection it is mainly the amount of data that affects performance and not query complexity. Moreover, note that the speedup of roughly 2.5X is similar for all three scale factors.

6.4 Putting It All Together

While *Ibex* already supports running a wide range of queries with hardware acceleration in MySQL, there are still missing parts in the current version of our prototype (optimizer modifications, indices, etc.) that would allow us to efficiently run all queries of a sophisticated benchmark such as TPC-H. Nevertheless, to give a glimpse of the impact that *Ibex* can have on more complex queries, we used Query 13 of the TPC-H benchmark (cf. Q_{13} below)[7] and handcrafted the missing parts to run this query.

```
1    SELECT c_count, COUNT ( * ) AS custdist
2      FROM
3   ( SELECT c_custkey, COUNT ( o_orderkey ) AS c_count
4      FROM customer LEFT OUTER JOIN orders ON
5           c_custkey = o_custkey AND                    (Q13)
6           o_comment LIKE '%express%packages%'
7  GROUP BY c_custkey ) AS c_orders
8  GROUP BY c_count
9  ORDER BY custdist DESC, c_count DESC;
```

Eager Aggregation. Q_{13} involves two base tables: customer and orders. We store customer as a MyISAM table and orders as an *Ibex* table. By default, *Ibex* will push projection (line 3) and selection (line 6) to the FPGA. The `GROUP BY` clause (line 7), however, would not be pushed because of the preceding join (line 4), which is computed outside of the storage engine. Fortunately, it is possible to evaluate the `GROUP BY` clause before the join by replacing the orders table (line 4) with the following inner query:

```
  ( SELECT o_custkey, COUNT ( o_orderkey ) AS c_count
      FROM orders
     WHERE o_comment LIKE '%express%packages%'
  GROUP BY o_custkey ) AS orders2
```

Furthermore, the `COUNT` (line 3 in Q_{13}) and `GROUP BY` clause (line 7 in Q_{13}) of the outer query can now be removed. This technique is known as *eager aggregation* and has been studied in detail, *e.g.*, in [36].

On-the-fly Index Creation. A storage engine with index support (*e.g.*, MyISAM), will solve the join (line 4 in Q_{13}) using the index given by the foreign key relationship. While *Ibex* does not support indices yet, it is possible to use the hash table that is created on behalf of the `GROUP BY` query as an index for the subsequent join. For this to work, the optimizer needs to be tweaked to use an index-based access path even though an explicit index does not exist. Here we scratch the surface of a topic beyond the scope of this paper: the implications of *Ibex* for the optimizer.

[7]We took the liberty to replace the `NOT LIKE` of the original TPC-H query with `LIKE` in Q_{13} (line 6) to increase the selectivity of the `WHERE` clause.

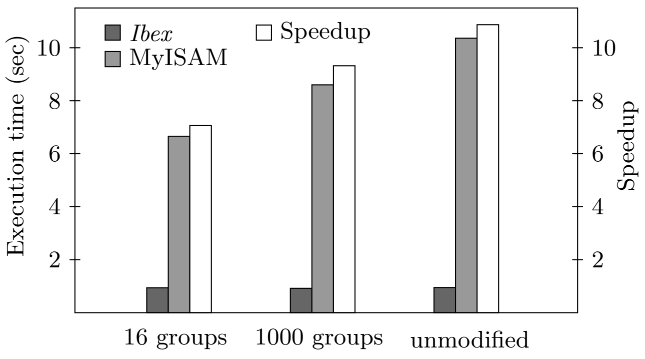

Figure 14: Execution time and speedup for Q_{13} of the TPC-H benchmark (*Ibex* versus MyISAM).

Performance. Figure 14 shows our measurements for Q_{13} and scale factor = 1.0. We executed three experiments: "unmodified" refers to the original orders table (14,730 groups after selection), for "1000 groups" and "16 groups" we modified o_custkey to reduce the number of groups to one thousand and sixteen, respectively. Since there are no bypass tuples here, the performance of *Ibex* is consistent, independent of the number of groups, whereas MyISAM is faster for a smaller number of groups, resulting in speedups between 7X and 11X. The increased speedup compared to Figure 13 can be attributed to the effect of the additional `GROUP BY` pushdown (as opposed to mere selection and projection).

7. ENERGY EFFICIENCY

An important limiting factor in many systems today is energy consumption. A key appeal to design heterogeneous systems thus lies in achieving lower energy consumption and a good performance/watt ratio. Off-loading queries to specialized hardware naturally reduces the load on the host CPU, which in turn may significantly reduce the overall energy consumption of the system. Hardware accelerators, on the other hand, consume only a fraction of the 40–130 watts that commodity processors consume today.

Power Consumption Characteristics. The maximum thermal design power (TDP) of our Intel Quad-Core CPU is 95 watts. By contrast, one Vertex 4 SSD only consumes 2.5 watts of power when active, and 1.3 watts otherwise. We measured the wall power of our entire system with a power meter. The power consumption of the system is **39 watts** when MySQL is *idle* and the FPGA board is *turned off.* With one core under full load the power consumption increases to 54 watts, and with all 4 cores under full load total power consumption amounts to 105 watts.

The FPGA chip itself configured with *Ibex* requires only 2.9 watts.[8] Power consumption of our FPGA board (including the FPGA) is roughly 8 watts, *i.e.*, the system consumes **47 watts** when *idle* with the FPGA board connected and powered on. However, much of the 8 watts are spent on the FPGA board (*i.e.*, an energy in-efficient, general-purpose development board) and not the FPGA chip itself.

[8]FPGA power consumption was estimated using the Xilinx Power Analyzer tool.

Table 1: **Energy and power consumption during query execution of different queries.**

Query / Engine	Energy	Exec Time	∅ Power	Δ Power
Q_2 / MyISAM	3,888 J	66.7 s	58.3 W	19.3 W
Q_2 / *Ibex*	216 J	4.4 s	49.1 W	2.1 W
Q_{13} / MyISAM	576 J	10.4 s	55.4 W	16.4 W
Q_{13} / *Ibex*	47 J	0.95 s	49.8 W	2.8 W

Energy Consumption During Query Execution. We measured energy consumption (joules) of the host system during execution of the following two queries from the previous section: *(i)* Q_2 discussed in Section 6.2, *(ii)* Q_{13} discussed in Section 6.4. The average power consumption (∅ Power) can be computed by dividing the measured energy consumption by the execution time. Δ Power represents the increase of power consumption versus the system being idle (for MyISAM measurements idle power is 39 watts, whereas for *Ibex* it is 47 watts, as explained above). The results are displayed in Table 1.

Our measurements indicate that power consumption increases significantly for commodity storage engines when executing queries, whereas when the queries are off-loaded to the FPGA there is only a minimal increase of power. The overall amount of *energy* consumed is the product of power and time. Since the queries execute much faster on the FPGA, energy consumption is improved even more dramatically, *e.g.*, for the GROUP BY query Q_2, *Ibex* requires only 216 joules, whereas MyISAM consumes 3,888 joules.

Future Outlook. Notice that the energy consumption measurements for *Ibex* include power consumption of the FPGA board, and yet we see significant improvements compared to MyISAM. It is conceivable that in an appliance the FPGA would be integrated directly into the SSD itself for query off-loading, in the spirit of [11], leading to a further reduction in power consumption and as a result even better energy efficiency. Furthermore, since *Ibex* rarely uses the full power of the CPU at all, overall energy consumption is likely to improve when we exchange our high-performance Intel CPU with less powerful but more energy-efficient processor, *e.g.*, an ARM or Atom.

8. RESOURCE CONSUMPTION

In Table 2, we display resource consumption for different circuit components of our complete system. To communicate with the host that runs the MySQL database we used Microsoft's communication framework SIRC [12] and for SATA we used our open source core Groundhog [33]. Together these components consume slightly less than 10 % of the available resources. Furthermore, we distinguish two versions of *Ibex*, one using the BRAM-based hash table for the GROUP BY component, and the other using DRAM to store hash table entries.

***Ibex* (BRAM).** Excluding SIRC and the SATA core, *Ibex* consumes roughly 25 % slices independent of how big the hash table is. The number of BRAM blocks n depends on the size of the hash table. Here the word width was set to 256 bits (8 × 32 bit), *i.e.*, one 36-Kbit BRAM block can hold $k = 144$ entries. However, the actual hash table size is restricted to powers of two, *i.e.*, $k = 2^{\lfloor log_2(n*36) \rfloor + 2}$ entries.

Table 2: **Chip utilization on our Virtex-5 FPGA.**

Module	Slices		BRAMs	
Available	17,280	100.0 %	148	100.0 %
SIRC [12]	1027	5.9 %	11	7.4 %
SATA core [33]	725	4.2 %	2	1.4 %
Ibex (BRAM)	4188	24.2 %	$1 + n$	≥1.4 %
DRAM core [2]	1651	10.0 %	4	2.7 %
Ibex (DRAM)	5047	29.2 %	1	0.7 %

***Ibex* (DRAM).** The advantage of DRAM is that we can support a large number of groups without having to spend precious BRAM blocks for the hash table. Nevertheless, using DRAM incurs other resource consumption costs. The DRAM core [2] consumes additional 10 % of slices, as well as four BRAM blocks. Moreover, the use of DRAM also affects resource consumption of *Ibex* since additional logic such as clock bridges and a more complex control flow are required. Thus, in total, we measured that *Ibex* consumes roughly 40 % of all available slices when using DRAM, *i.e.*, 15 % more than the BRAM version.

9. FUTURE WORK

Ibex opens up several interesting lines of research beyond query processing. One example is online gathering of database statistics. The trade-off between statistics accuracy and maintenance cost is a long-standing database problem. The embedding of *Ibex* in the data path of a DBMS would allow additional hardware units to eavesdrop on the regular traffic, as data is read from secondary storage. While doing so, such units could compute, *e.g.*, relevant histograms on-the-fly, as a side effect of regular query processing without any runtime or CPU overhead (as in [19]).

Moreover, additional functionality could be placed within the processing pipeline of Figure 2, *e.g.*, to perform data (de)compression to increase throughput to and from the FPGA (as in [17]), or to perform encryption/decryption to implement confidentiality and other security features (as in [1]). The only overhead would be additional chip space for the add-on functionality, which is available, even for the small FPGA used in this paper, as we showed in Section 8.

10. CONCLUSION

In this paper, we presented *Ibex*—a prototype of an intelligent storage engine—that uses FPGAs to implement hardware accelerators close to the storage medium, in the data path of a DBMS. We integrated *Ibex* with MySQL but we believe that the concepts presented are more general, and that most existing database systems would benefit from an intelligent storage engine such as *Ibex*. To the best of our knowledge, *Ibex* is the first storage engine that allows combining hardware acceleration with an existing open-source DBMS in a seamless manner. *Ibex* is also the first FPGA-based accelerator that supports complex operators like GROUP BY rather than mere filtering of rows. Finally, we intend to release the source code of *Ibex* in the near future.

Acknowledgments.
This work was supported by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch`).

11. REFERENCES

[1] A. Arasu et al. Orthogonal Security with Cipherbase. In *Proc. 6th CIDR*, Asilomar, CA, USA, 2013.

[2] R. Bittner. The Speedy DDR2 Controller For FPGAs. In *Proc. ERSA*, pages 205–211, Las Vegas, NV, USA, 2009.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. 2nd CIDR*, pages 225–237, Asilomar, CA, USA, 2005.

[4] S. Borkar and A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5), 2011.

[5] C. Computer. Convey HC-2, 2012. http://www.conveycomputer.com.

[6] C. Dennl, D. Ziener, and J. Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *Proc. 20th FCCM*, pages 45–52, Toronto, ON, Canada, 2012.

[7] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration. In *Proc. 21st FCCM*, pages 25–28, Seattle, WA, USA, 2013.

[8] D. Dewitt. DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans. on Comp.*, 28(6):395–406, 1979.

[9] D. Dewitt et al. The Gamma Database Machine Project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[10] U. Dhawan and A. DeHon. Area-Efficient Near-Associative Memories on FPGAs. In *Proc. 21st FPGA*, pages 191–200, Monterey, California, USA, 2013.

[11] J. Do et al. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proc. SIGMOD*, pages 1221–1230, New York, NY, USA, 2013.

[12] K. Eguro. SIRC: An Extensible Reconfigurable Computing Communication API. In *Proc. 18th FCCM*, pages 135–138, Charlotte, NC, USA, 2010.

[13] B. Gold et al. Accelerating Database Operations Using a Network Processor. In *Proc. 1st DaMoN*, Baltimore, MD, USA, 2005.

[14] N. Govindaraju et al. GPUTeraSort: High-Performance Graphics Co-Processor Sorting for Large Database Management. In *Proc. SIGMOD*, pages 325–336, Chicago, IL, USA, 2006.

[15] G. Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[16] B. He et al. Relational Joins on Graphics Processors. In *Proc. SIGMOD*, pages 511–524, Vancouver, BC, USA, 2008.

[17] IBM/Netezza. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics, 2011. White paper: http://www.redbooks.ibm.com/abstracts/redp4725.html.

[18] Z. István et al. A Flexible Hash Table Design for 10GBPs Key-Value Stores in FPGAs. In *Proc. 23rd FPL*, pages 1–8, Porto, Portugal, 2013.

[19] Z. István, L. Woods, and G. Alonso. Histograms as a Side Effect of Data Movement for Big Data. In *Proc. SIGMOD*, Snowbird, UT, USA, 2014.

[20] R. Moussalli et al. Accelerating XML Query Matching through Custom Stack Generation on FPGAs. In *Proc. 5th HiPEAC*, pages 141–155, Pisa, Italy, 2010.

[21] R. Müller, J. Teubner, and G. Alonso. Streams on Wires—A Query Compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.

[22] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible Query Processor on FPGAs. *PVLDB*, 6(12):1310–1313, 2013.

[23] Oracle. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server, 2012. White paper: http://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf.

[24] M. Sadoghi et al. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *PVLDB*, 3(2):1525–1528, 2010.

[25] M. Sadoghi et al. Multi-Query Stream Processing on FPGAs. In *Proc. 28th ICDE*, pages 1229–1232, 2012.

[26] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. 20th VLDB*, pages 510–521, Santiago de Chile, Chile, 1994.

[27] I. Sourdis et al. A Reconfigurable Perfect-Hashing Scheme For Packet Inspection. In *Proc. 15th FPL*, pages 644–647, 2005.

[28] B. Sukhwani et al. Database Analytics Acceleration using FPGAs. In *Proc. 21st PACT*, pages 411–420, Minneapolis, MN, USA, 2012.

[29] T. Takenaka, M. Takagi, and H. Inoue. A Scalable Complex Event Processing Framework for Combination of SQL-based Continuous Queries and C/C++ Functions. In *Proc. 22nd FPL*, pages 237–242, Oslo, Norway, 2012.

[30] J. Teubner, L. Woods, and C. Nie. Skeleton Automata: Reconfiguring without Reconstructing. In *Proc. SIGMOD*, pages 229–240, Scottsdale, AZ, USA, 2012.

[31] T. Thinh, S. Kittitornkun, and S. Tomiyama. Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS. In *ICFPT*, pages 121–128, 2007.

[32] P. Vaidya et al. Symbiote: A Reconfigurable Logic Assisted data Stream Management System (RLADSMS). In *Proc. SIGMOD*, pages 1147–1150, Indianapolis, IN, USA, 2010.

[33] L. Woods and K. Eguro. Groundhog—A Serial ATA Host Bus Adapter (HBA) for FPGAs. In *Proc. 20th FCCM*, pages 220–223, 2012.

[34] L. Woods, Z. István, and G. Alonso. Hybrid FPGA-accelerated SQL Query Processing. In *Proc. 23rd FPL*, page 1, Porto, Portugal, 2013.

[35] L. Woods, J. Teubner, and G. Alonso. Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances. In *Proc. SIGMOD*, pages 1073–1076, New York, NY, USA, 2013.

[36] W. Yan and P.-A. Larson. Eager Aggregation and Lazy Aggregation. In *Proc. 21th VLDB*, pages 345–357, Zurich, Switzerland, 1995.

[37] Y.-H. Yang, W. Jiang, and V. Prasanna. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proc. ANCS*, 2008.