A Co-Design Approach for Accelerated SQL Query Processing via FPGA-based Data Filtering

Andreas Becher, Daniel Ziener, Klaus Meyer-Wegener, and Jürgen Teich
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
Email: {andreas.becher, daniel.ziener, klaus.meyer-wegener, juergen.teich}@fau.de

***Abstract*—In this paper, we present a novel co-designed architecture for high throughput database query processing. It consists of a highly configurable FPGA-based filter chain with arithmetic operation support and an alignment unit. This feeds the filtered data directly and in a cache-optimized way to embedded processors which are responsible for joining tables and post processing. High throughput interfaces and parallelism of FPGA implementations are thus combined in order to provide reduced and cache-aligned data for optimized processor access. As a key component, we introduce a new highly configurable bloom filter cascade to relieve a processor of time-consuming hash-value computation and to significantly reduce the data for hash joins. It is shown that this unique approach may reduce the amount of data to be processed by the processors in typical data-warehouse applications by several orders of magnitude. The proposed co-design has been implemented on the embedded low-energy system-on-chip (SoC) platform Xilinx Zynq. Performance results for standard benchmarks show an up to 10 x higher throughput compared to a full featured x86-based processor at only a fraction of energy consumption.**

I. Introduction

The amount of data created, captured, or replicated in the IT sector and the amount of energy needed for the processing rises significantly every year. Recent work [1]–[4] has shown that it is worthwhile to investigate the use of FPGAs in the field of database applications in general and for query-execution purposes in particular with the goal of increasing data throughput with lower energy effort [5]. A storage system was proposed in [6] which filters the incoming data with restriction and aggregation operators implemented on an FPGA. Patent [7] describes the usage of Bloom filter close the storage system to reduce the amount of data needed to process. However, the approach is rather static and only few query dependent Bloom filter parameters can be adjusted. To the best of authors knowledge, our proposed work in this paper is the first approach which utilizes *concatenated* high throughput hardware-based Bloom filters combined with restriction and ALU modules for individual query-aware optimized hash-join acceleration.

A good overview of the application field of bloom filters is given in [8]. FPGA-based bloom filters are often used in network processing systems, e.g., for packet inspection and intrusion detection [9], [10].

II. Concept & System Overview

Many data-intensive applications suffer from *limited memory bandwidth* when processed on a conventional microprocessor system, particularly in software-based database query processing, where often a tiny subset of the stored data is needed to produce the result. As a remedy, we propose a reconfigurable data preprocessing approach in FPGA hardware which reduces the incoming data from external memory to the data required for subsequent evaluation on the host processors as much as possible. We introduce a new highly configurable Bloom filter module which accelerate hash based join operations substantially.

978-1-4673-9091-0/15/$31.00 ©2015 IEEE

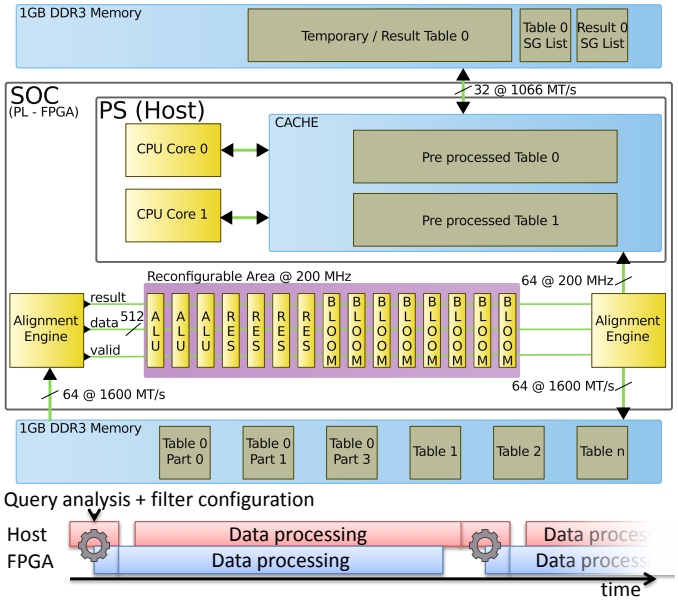

Fig. 1. Overview of the proposed architecture for hardware-accelerated query processing (above). After incoming queries are analyzed and the filter chain configured by parameter adaptation and, if necessary, by structure adaptation through partial dynamic reconfiguration, the data is streamed through the filter chain and concurrently, the filtered data is processed by a hash join implemented in software (see timing diagram below) using an zc706 board.

Figure 1 shows our proposed co-design implemented on a low energy SoC. The data is fetched from external memory and is streamed through the FPGA-based filter chain. The filter chain may contain three types of modules: a *restriction* module (RES) and *ALU* modules which cover compares and, respectively, arithmetic expressions in the *where* clauses. The *bloom filter* modules are responsible for pre-filtering and hash value calculation of the data for a subsequent hash-based join. After the data reduction achieved by these modules, an alignment unit adjusts the filtered data for best possible subsequent processor access, e.g., to be cache-line-aligned and cache-optimized. The aligned remaining data is then processed by the processor system (Host).

As seen in Figure 1, the FPGA-based processing cascade and the processor work in parallel to ensure a maximum data throughput. Furthermore, only stream line operators are used in the pipeline to benefit from the high I/O bandwidth provided when accessing consecutive memory locations. The modules themselves are parameterizable which only requires a reconfiguration if the types of query changes, e.g., search queries and update queries.

Operations which cannot be implemented in such a stream-based manner (e. g., joins) and operations which assume joined tables and thus operate usually on already reduced data sets (e. g., aggregations), are better suited for software processing. Besides this management part, the

software is responsible to implement all not yet covered query operators i.e. (a) the hash-join operator and (b) different kinds of *aggregation* and (c) *group-by* operators.

Incoming queries are analyzed, and if feasible, accelerated in hardware using a proposed accelerator library. A join on multiple tables can be accelerated by configuring the Bloom filters to work on multiple attributes. As fallback, we support to process all modules also in software. Therefore, the database system is still able to process the query even if not all operators can be accelerated. Another task partitioned to software processing is the table management. Rows are aligned to the I/O width of the accelerator and tables are divided into blocks of up to 40 MB of consecutive memory. Scatter-gather lists are generated for each known table plus some result and temporary tables.

The proposed query processing therefore executes as follows: An incoming query is analyzed and the suiting accelerator is chosen. Parameters for each module are determined dependent on the query. One alignment unit is parameterized to extract the proper attributes from the incoming data stream and include placeholders for intermediate results. Another alignment unit at the end of the accelerator pipeline is parameterized to align the data according to certain byte boundaries and discard not needed attributes and tuples. The alignment units work through the scatter gather lists of the corresponding tables in use. Last, the processor calls small software kernels to execute the not yet finished operations on the smaller remaining data of the query. The last step can be done efficiently in software as data is amounts are typically greatly reduced already and hash tables can fit into caches. Also, the higher clock rate compared to the FPGA-logic and a multithreading and/or multiprocessor platform allows for a more efficient processing of this data than in hardware. Here, a central role for achieving a dramatic reduction in the amount of data which has to be processed in software play the proposed join-dependent Bloom filters explained next.

III. Configurable Bloom Filter

A Bloom filter [11] is an essential part in our filter chain and the key enabler for the hash join acceleration. This is achieved by filtering not matching keys before the join operation in software. The concept of Bloom filters was first introduced by Bloom [11] and based on the concept of allowable-error filtering. The main idea is to populate a dictionary based on hashed keys and then to probe this dictionary with the data to filter. The classical way of implementing this kind of hash-based dictionary is a bit vector where the hash-key addresses the bits in this vector. During the *population phase*, the key to be inserted is hashed by one or more hash functions and the bits at the resulting addresses are set to '1'. To test whether a certain key is present in the bit vector in the *probe phase*, the key is hashed with the same hash functions and the bit vector is probed at the generated addresses. For the key to be contained in the set, all probed bits must be '1'.

Due to the nature of hash functions, so-called *false positives* can occur. This means that the Bloom filter shows a key to be contained in the set, but actually it is not. On the other hand, a recall of 100 % is guaranteed, because keys that cannot be found in the dictionary are definitely not contained in the set. This is of great help for hash-based joins in query processing. Conflict resolution for false positives can be expensive, since the actual keys must be compared for every positive result. The Bloom filter can reduce the number of conflict resolutions by reducing the probability of false positives. One way to achieve this is using more than one *individual* hash function. *Individual* here means that all hash functions produce different hash values for the same key.

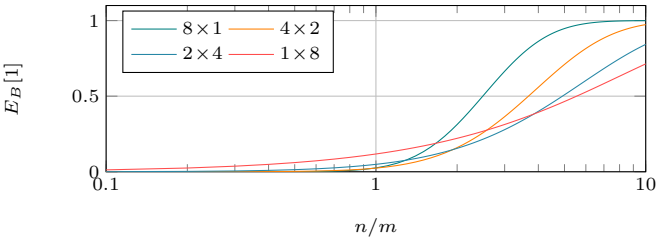

Fig. 2. Bloom Error E_B in respect to the ratio of inserted items $\left(\frac{n}{m}\right)$ and different groups sizes g and number of concatenated filters b for $m = 16318$ and a total of 8 instances. The legends read $b \times g$.

The probability of a false positive is called the *Bloom Error* E_B. It can be calculated from the following three variables: The number of hash functions k, the number of inserted key values n, and the size of the bit vector m. The formula $\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k$ calculates E_B for a shared bit vector which is set by each result of the different hash functions applied to a key.

However, modification of m or k without changing the throughput is only possible at synthesis time and, therefore, very time consuming. To overcome this limitation, our implementation of the Bloom filter works slightly different. Each of our Bloom filter modules has one bitvector (m constant) and one hash function ($k = 1$). In order to decrease E_B, our implementation allows building *Virtual Bloom filters* by *grouping* and *chaining* these simple Bloom filter modules. By grouping multiple Bloom filters, an increase of the size of the bit vector m can be achieved. This is done via a *valid_mask* indicating if a Bloom filter instance is responsible for the calculated hash value. The group variable $g \in 2^a$,where $a \in \mathbb{Z}^+$ describes the multiplication factor of the bit vector m. E.g., $g = 4$ means four of our Bloom filters are grouped to act as a single one with $m_{group} = 4 \cdot m$. The hash function is configured to produce $lb(m_{group})$ bits whereas the bits $[lb(m_{group}) - 1 : lb(m)]$ indicate which instance of the Bloom filter should handle the key. Furthermore, these groups of Bloom filters can be concatenated to decrease E_B even more. Introducing b as the number of Bloom groups which are concatenated, this leads to Equation (1) for E_B.

$$E_B = \prod_{i=0}^{b-1} \left(1 - \left(1 - \frac{1}{g_i \cdot m}\right)^n\right) \tag{1}$$

The impact on the Bloom Error E_B with varying g and b is depicted in Figure 2 for bit vectors with the same size m. One can see the flexibility of our approach as the filter chain can be adopted to the expected amount of inserted tuples on the fly query-dependant. With a given amount of Bloom filter modules in a pre-synthesized module, each individual filter module can be used to further decrease E_B.

Our FPGA implementation of a Bloom filter is shown in Figure 3. The main building block are BRAMs which store the bit vector. Special treatment is needed to change a single bit in the selected column. Therefore, a read-modify-write strategy must be used to keep the bits of the column that are already set. If maximum performance related to lowest timing should be achieved, multiple clock cycles may be needed for this strategy, as BRAM blocks must then implement registers for the address and for the data port to allow for maximum clock frequencies. This can lead to reads from not yet updated columns. To handle this, a small cache is implemented which holds exactly the data not yet written to the BRAM. The cache strategy is a simple write-through. With this technique, and using a dual-port BRAM, the bloom filter can achieve

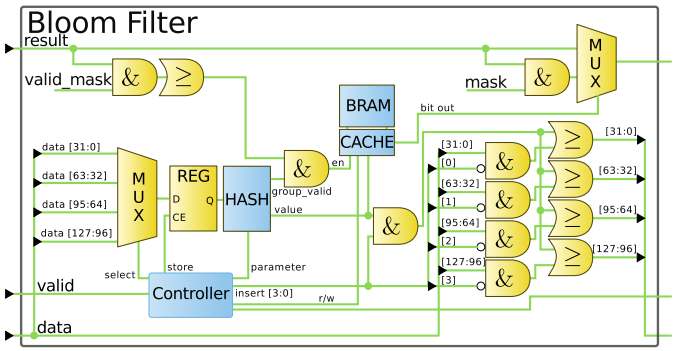

Fig. 3. RTL architecture of a Bloom filter module using 128 bit data bus. The controller, input multiplexers, and registers select the desired operand attribute from incoming chunks (data) for processing. A hash value is calculated from the key attribute in the Bloom filter and is stored or looked up in the bit vector, implemented using BRAMs and a cache block. Finally, the calculated hash value can be inserted into the stream.

a throughput of one update each clock cycle. Therefore, no stalling of incoming data occurs.

The Bloom filter module also utilizes a result bus as Figure 3 depicts. The result bus indicates if a tuple is valid to be used in the *population phase*. During the *probe phase*, it is used for the concatenation of Bloom filters.

As hash function, an algorithm inspired by the Fowler-Noll-Vo (FNV) hash function [12], more precisely, the VNF-1a algorithm has been implemented. Our implementation has been designed to be parametrizable and configurable in order to produce different hash values for same inputs based on run-time configurable *parameters*. In addition to calculating the hash value, our hash entity also produces a *group_valid* signal in order to implement the already introduced group functionality. Perfect performance in relation to throughput is achieved by pipelining the hash function completely.

Last, the generated hash values can be forwarded to the processor by an insert network. This relieves the processor even further in performing the software-based hash join, because recalculation of the hash values can be avoided.

IV. Experiments & Results

The proposed co-design has been prototyped on a Xilinx *zc706 MPSoC* board which includes a Xilinx Zynq-7000 device and two separated DDR3 memories. The throughput from processor attached memory to the dual-core ARM processor is limited by a theoretical total throughput of 3.2 GB/s for reads from and writes[1] to the memory connected to the processor. On the other hand, the DDR memory connected to the FPGA fabric delivers up to 12.8 GB/s. The implemented SoC design was already shown in Figure 1 with the processor attached memory (above) and the memory for the FPGA fabric in which the database tables reside (below).

Table I reports the resources occupied by the implemented design. It can be seen that enough space is available to increase the Bloom filter sizes or extend the ALU module or to simply increase the filter count. *Vivado* took about 23 minutes to generate the bitfiles for this design on a regular desktop.

By utilizing the PCAP interface, the ARM processors are used to reconfigure the partially reconfigurable area of the FPGA with different accelerators. A reconfiguration of the accelerator takes up to 30 ms but can be further increased as demonstated by [13]. As all modules are parameterizable, reconfiguration is often not needed. Parameter adaptations

[1]Given a system clock of 200 MHz inside the FPGA fabric, and $\frac{64}{8}$ Bytes that can be transferred per clock cycle over each of the 2 HP-AXI interfaces.

TABLE I. Post-implementation resource consumption on the xc7z045ffg900-2 SoC.

Resource	Utilization	Available	Utilization [%]
FF	59,171	437,200	13.53
LUT	48,405	218,600	22.14
Memory LUT	9,856	70,400	14.00
I/O	118	362	32.59
BRAM	59	545	10.83

TABLE II. Database used for experimental results generated with *dsdgen* [14] including alignment overhead.

	date_dim	customer	store_sales
Tuples	73,049	100,000	2,880,404
Bytes [Bytes]	14,025,408	25,600,000	368,691,712

of the filter chain can be done in about 0.4 µs, and 10 µs respecting query analysis.

In order to compare (a) our approach, software-only approaches running (b) on the ARM processor of the Zynq SoC and (c) on an x86-based platform, a number of test queries have been processed on all three systems. The software solutions use MariaDB[2] as database software and all tables are configured as *in-memory* tables. Additional memory (heap) for MariaDB is set accordingly not to restrict MariaDB during the execution. The x86 platform consists of an Intel Core i7-3770 CPU running 8 threads at 3.40 GHz (3.9 GHz turbo mode) with 8192 kB cache and two 8 GB DDR3 1600 memory. The ARM platform holds an ARM Cortex-A9 dual core running at 666 MHz with a 32 kB L1 cache, 512 kB L2 cache and 1 GB DDR3 1066 memory.

For our experiments, we have created the tables with the TPC-DS data-generator tool [14] using a scale factor of 1. To demonstrate and compare the performance of each of the three operators for typical queries from that benchmark, we have chosen the largest fact table available, namely *store_sales*, and two different dimension tables, namely *date_dim* and *customer* (see Table II). The dimension tables allow to demonstrate the impact of the Bloom filter, because the join key *date* is distributed linearly in the *date_dim* table, but not in the *customer* table. Note that even the smallest dimension table exceeds the size of the caches of the available ARM processor by far.

To stress the Bloom filters of our design in different ways, two basic queries have been chosen (*q0* and *q1*) as listed after the references. The restrictions and arithmetic operations are processed completely by our reconfigurable hardware. As accelerator, the partially reconfigurable area has been configured as shown in Figure 1. The Bloom filters have been implemented using one 18 kBit BRAM each, so that each has a vector of $m = 16{,}384$ bits. Table III presents the measured results.

A. Performance Evaluation

The shown table sizes indicate the amount of data which is left for the software running on the ARM processor. The worst case was produced by query *q0c0*, where 28.6 MB had to be processed by the ARM. In addition to that, 45 % false positives on a 426 kB dimension table require substantial error handling, which leads to many cache and branch prediction misses and thus reduces performance gains. However, one can see that the x86-based system presumably suffers from the same problem, since the execution time rises to the highest value measured in our experiments. The impact of our flexible Bloom filter design can be seen for query *q0c1*. Reducing the Bloom Error through smart configuration (2x4) gives significant performance benefits.

Overall, the *q0* queries stress the Bloom filters most, as one can tell from the high number of *false positives*. This is due to

[2]Version 10.0.20 on x86 and 5.5.36 on ARM

TABLE III. EXPERIMENTAL RESULTS FROM TWO DIFFERENT QUERIES AND VARIATIONS.

		q0a	q0b0	q0b1	q0c0	q0c1	q1a	q1b	q1c	q1d
Reduced dimension table [Bytes]		11,056	144,848	144,848	425,976	425,976	2,928	248	496	248
Reduced fact table [Bytes]		472,452	9,239,480	4,639,896	28,175,988	19,684,056	6,646,332	1,113,024	1,470,108	111,3024
Bloom configuration (bxg)		3x1	3x1	8x1	8x1	2x4	3x1	3x1	3x1	3x1
False positives [%]		2.72	57.30	14.96	45.40	21.84	0.00	0.00	0.00	0.00
Processing time [ms]		42.63	55.40	44.20	89.10	57.40	45.15	41.25	41.45	41.25
MariaDB on ARM processing time [s]		5.74	6.90	6.90	8.77	8.77	6.75	6.71	8.33	6.79
MariaDB on x86 processing time [s]		0.37	0.42	0.42	0.57	0.57	0.35	0.35	0.42	0.35
Energy hardware acc. [J]		0.18	0.24	0.19	0.38	0.24	0.19	0.17	0.17	0.17
Energy MariaDB on ARM [J]		1.22	1.47	1.47	1.87	1.87	1.44	1.43	1.77	1.45
Energy MariaDB on x86 [J]		4.69	5.33	5.33	7.23	7.23	4.44	4.44	5.33	4.44
Improvement Factors										
Energy-efficiency	vs. MariaDB on ARM	6.66	6.16	7.72	4.87	7.56	7.39	8.04	9.94	8.14
	vs. MariaDB on x86	25.55	22.32	27.97	18.83	29.23	22.82	24.98	29.83	24.98
Processing time	vs. MariaDB on ARM	134.66	124.56	156.12	98.43	152.80	149.49	162.66	200.95	164.60
	vs. MariaDB on x86	8.68	7.58	9.50	6.40	9.93	7.75	8.48	10.13	8.48

the fact that with the number of inserted elements (n) rising from 1,400 to 53,000, the Bloom filters saturate and more false positives are generated. Also, the join attributes have random values, which challenges the hash functions additionally. To demonstrate the effect of b, we have also run query *q0b* with 8 blooms in use. This has reduced the false-positive rate to only about 14 % and leads to an according speedup.

With Query *q1*, we focus more on the number of restrictions in use. First thing to notice is that with three active Bloom filters, no false positives occur and the processor is only checking the key and summing up the values without any cache misses and almost no branch misspredictions, which is completely hidden behind the prefiltering in hardware. One can see that the x86-based system can now exploit its higher I/O bandwidth, without having to deal with many false positives. The impact of the number of restrictions is visible at query *q1c*, with four restriction modules in use.

Furthermore, we ran the query testset also on a commercial in-memory DBMS on the same x86 server with very varying numbers from a slight slowdown of 0.9 × to up to 2.5 × speedup of our approach compared to the commercial DBMS. A noticed dependence between the amount of columns used in the queries and the processing time can be explained by the use of a column-based fetching and processing and aggressive table access optimization.

Finally, we made experiments on the scalability of our approach by introducing a scale factor of 2 for the tables. We can see from the results that our approach scales nicely with the fact table size as long as the filtered dimension table fits the processor cache and the Bloom Error is low.

B. Energy Efficiency

Additionally, we have measured the maximum dynamic *power consumption* of the complete zc706 board while processing data. The measured maximum power consumption is then multiplied with the processing time to calculate an average energy per query value.

For power consumption of the x86 server platform, we have measured the average dynamic wall power consumption of processing the queries. Even though the ratio of one and eight threads is not eight, the I/O bandwidth or cache limit is reached by four queries running simultaneously, and the processing time begins to rise to almost double when the number of queries is increased further. Therefore, we have chosen four threads to be the maximum. Starting at 18.7 W for a single core in use, the consumption rises to a maximum of 49.1 W when processing on four cores which gives about 12.7 W dynamic power for each core in use. On average, the software-only approach on the ARM core consumes about 7.4 × more energy, and the x86-based system about 25 × more energy compared to our proposed architecture.

V. CONCLUSIONS

In this paper, we proposed a hardware/software co-design architecture for SQL query acceleration through hardware-based filtering and software-based table joining. FPGA-based Bloom filters are proposed to be located between the data source and the processor. The concatenated Bloom filters are themselves highly configurable to speed up common database queries using joins significantly. Due to the reduced data volume, this can even be done on low-power embedded processors demonstrating high performance paired with high energy efficiency.

REFERENCES

[1] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM redguide publication, IBM, 2011.

[2] R. Mueller and J. Teubner. FPGA: What's in it for a database? In *Proc. ACM SIGMOD Conf.* 2009. Tutorial.

[3] R. J. Halstead, B. Sukhwani, et al. Accelerating Join Operation for Relational Databases with FPGAs. In *FCCM*, pp. 17–20. IEEE, 2013.

[4] C. Dennl, D. Ziener, et al. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *FCCM*, pp. 45–52. May 2012.

[5] A. Becher, F. Bauer, et al. Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration. In *FPL*, pp. 662 – 669. 2014.

[6] L. Woods, Z. István, et al. Ibex – An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *PVLDB*, 2014.

[7] B. Sukhwani, S. Asaad, et al. Hardware-accelerated relational joins, Aug. 12 2014. US Patent 8,805,850.

[8] S. Tarkoma, C. E. Rothenberg, et al. Theory and practice of bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1):131–155, 2012.

[9] S. Dharmapurikar, P. Krishnamurthy, et al. Deep packet inspection using parallel bloom filters. vol. 24, pp. 52–61. Jan 2004. ISSN 0272-1732.

[10] A. Nikitakis and L. Papaefstathiou. A memory-efficient FPGA-based classification engine. In *FCCM*, pp. 53–62. IEEE, 2008.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM.*, (7):422–426, Jul. 1970. ISSN 0001-0782.

[12] K. V. . D. E. G. Fowler, L. Noll. The FNV Non-Cryptographic Hash Algorithm.

[13] K. Vipin and S. Fahmy. Zycap: Efficient partial reconfiguration management on the xilinx zynq. *Embedded Systems Letters, IEEE*, 6(3):41–44, Sept 2014. ISSN 1943-0663.

[14] TPC BenchmarkDS (TPC-DS): The New Decision Support Benchmark Standard. [accessed 09-April-2015].


```
q0: select sum(s.ss_net_profit) from customer as c, store_sales as s where
(a) c.c_birth_year = 1983
(b) (c.c_birth_year >= 1980 and c.c_birth_year <= 2010)
(c) (c.c_birth_year >= 1955 and c.c_birth_year <= 1995) and s.ss_quantity > 10
and s.ss_customer_sk = c.c_customer_sk;
q1: select sum(s.ss_net_profit) from date_dim as d, store_sales as s where
(a) d.d_year = 2000
(b) d.d_year = 2000 and d.d_moy = 12
(c) (d.d_year = 2000 and d.d_moy = 12) or (d.d_year = 2001 and d.d_moy = 1 )
and s.ss_sold_date_sk = d.d_date_sk;
q1d: select sum(s.ss_list_price - s.ss_sales_price) from
        ... where d.d_year = 2000 and d.d_moy = 12 and s.ss_sold_date_sk = d.d_date_sk;
```