GPU-Based Speculative Query Processing for Database Operations

Peter Benjamin Volk
Technisch Universiät Dresden
peter.benjamin.volk@inf.tu-dresden.de

Dirk Habich
Technisch Universiät Dresden
dirk.habich@tu-dresden.de

Wolfgang Lehner
Technisch Universiät Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

With an increasing amount of data and user demands for fast query processing, the optimization of database operations continues to be a challenging task. A common optimization method is to leverage parallel hardware architectures. With the introduction of general-purpose GPU computing, massively parallel hardware has become available within commodity hardware. To efficiently exploit this technology, we introduce the method of speculative query processing. This speculative query processing works on, but is not limited to, a prefix tree structure to efficiently support heavily used database index operations. Fundamentally, our developed approach traverse a prefix tree structure in a speculative, parallel way instead of a step-by-step traversing. To show the benefits and opportunities of our novel approach, we present an exhaustive evaluation on a graphical processing unit.

1. INTRODUCTION

The increasing availability of structured data managed by relational databases emphasizes the need for fast query processing on large amounts of data. The performance of a database query can be improved in multiple ways and represents a permanent research focus in the database community [1]. The use of indexes is one of the most common and effective methods to enhance the speed of a query in a row-oriented database [15]. In this kind of database, a data tuple contains of multiple columns and the data tuples are stored and processed with all associated column. One common access method realized by such an index is the search for a specific data tuple within a dataset that satisfies certain properties, such as the value of a specific column. For this reason, multiple approaches have been developed and optimized for a variety of different data skews and hardware architectures [16]. Such indexes are one of the most commonly applied data structures, further optimizations of these structures are essential.

Furthermore, indexes can be used as a core component of column-store database systems [9, 19]. While row stores keep all columns physically close to each other, column stores partition the table vertically such that each column of a table is stored in a separate location. To be able to recreate a row, each column contains a row identifier next to its value. A column store is commonly the foundation for database systems performing a large number of analytical queries. These analytical queries access only a few columns but almost all rows of a table. Thus, column stores only access the needed data for this query type, while row stores need to access all columns since they are tightly bound to each other and cannot be skipped.

Therefore, index structures are heavily used in column as well as row-storage database system to efficiently support query processing. The main data structures used for indexes are tree-based structures. The trees store tuples, where each tuple consists of a key and a payload. A key is the identifier for a tuple, and the payload may be a link or the actual value. Multidimensional indexes store a combination of keys to a payload such as $(key_1, .., key_n, payload)$. For specific data skews and requirements, multiple algorithms and structures have been proposed to store and query data most efficiently under specific side conditions [22].

Aside from tree-based structures, prefix trees are a well-known structure in many areas of research. They are used in various ways, e.g., for the translation of virtual to physical addresses in operating systems [12, 21] or within commercial database systems for index compression in an Oracle system. Furthermore, they are widely applied to optimize relational joins of tables, as introduced in [10]. Therefore, it is essential to optimize this kind of structure as far as possible and to explore new ways for query processing with this structure.

Current CPU designs offer an increasing single-digit number of cores creating opportunities for database systems to leverage parallel algorithms. With the introduction of Graphical Processing Units (GPU) as general-purpose processors, a massively parallel architecture has become available to perform database operations in parallel. A common method to speed up the access to data is to partition the data and to perform operations on it in parallel. This can only be done if the underlying data structures contain no data dependencies. To efficiently exploit the technology of a highly parallel system for a prefix tree, we introduce the concept of speculative computing for database operations. With this new concept, we partially pre-create results that can potentially be used to answer database queries. Since these partial results are computed in parallel, the complete runtime of the query may benefit from this approach.

Our Contribution and Outline

In the following section, we briefly review the general idea of prefix trees and the basic structure of Graphical Processing Units (GPU). Then, we introduce our novel developed approach to traverse a prefix tree structure on tightly coupled system in parallel in Section 3. In order to enable this approach, we describe a method to resolve data dependencies for tress structures in Section 3.1. Moreover, we present an analytical study of our approach in Section 3.2. Section 4

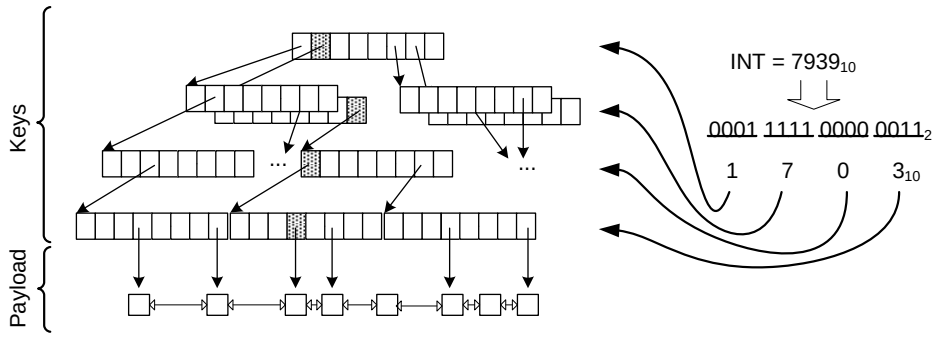


Figure 1: Prefix tree with a depth of 4 and the search for key 7939.

includes a description of how to leverage specific hardware features for our approach. Before we conclude the paper with a short summary in Section 7, we review related work in Section 5 and we present an exhaustive experimental evaluation of our developed approach in Section 6.

2. PREREQUISITES

As a basis for our new method, we use a prefix tree[1]. In this section, we will introduce and discuss the concept of prefix trees and give a detailed description of our hardware model.

2.1 Prefix Tree

Figure 1 shows an example of a prefix tree. The idea is that a tuple consists of a key K_i and a payload P_i, where the key K_i may be of any type of integer (32 or 64 bit) or a string with variable length. Every key in a tree has the same data type. The key is inserted into the tree and the payload is added as a linked list to the leaf nodes of the tree. The path within the tree for a key K_i is defined by the absolute value of the key instead of being defined by the relation of the key to the other keys. A key K_i is split into N equally sized sub-keys K_i^n consisting of $b = \frac{|K_i|}{N}$ bits with $|K_i|$ being the number of bits in the key. Each node contains an array of 2^b child node pointers.

On the $n-th$ tree level, the sub-key K_i^n determines the child node to be used by interpreting the $K_i^n - th$ sub-key as an integer value and accessing the $K_i^n - th$ element within the child node array. Therefore, the tree has a depth of N. Figure 1 illustrates how key 7939 is located within the tree. The elements that are accessed in each node are shaded. With a fixed tree depth, a key lookup can be executed in constant time independent of the tree load. A further benefit of this structure is that the keys are kept in sorted order. Therefore, sorting and filter operations can strongly benefit from this tree type since the operations can be performed directly on the structure of the tree without expensive comparison functions. One of the main points of criticism of this approach is the possible tree depth. Since the tree depth depends directly on the key length, this structure is unsuitable for long keys. Short keys can be very efficiently managed though. Thus, to be able to use this structure as a core component of a database system, it must be optimized with regard to its handling and support of longer keys that are evident in database systems.

A database index is a key component of a database system and widely used to optimize data access. Before a data structure is able to serve as a database index, it has to provide the following primitives:

1. get(key): This operation returns the first payload to a specific key.
2. getNext(): This operation returns the next payload from a position determined by a get(key) function. Once all payloads for a specific key have been returned, the function moves on to the next key in alphabetical order.
3. insert(key, payload): The insertion consists of two steps because the correct position for the key must be found first. This can be resolved by a modified get(key) operation that creates new nodes within the tree if the required nodes in the tree do not yet exist.
4. delete(key, payload): As a reversed operation to the insert, the delete operation removes the key-payload tuple from the index or makes it invisible to the user.

These are the most common and basic operations on an index within a database system [14]. For more enhanced features of a database system, the index may provide more sophisticated access functions, such as methods to determine the number of distinct items or a fill factor. Furthermore, we focus on the get(key) function, since it is used in all other functions, and therefore, an optimization within this function also enhances the speed of others.

2.2 GPU

The original purpose of Graphical Processing Units (GPUs) is to perform mathematical calculations to determine the color of a specific pixel in a picture [13]. With increasing requirements from the game and movie industries, GPUs have become a source for great computing power. This is achieved by highly specialized hardware consisting of tightly coupled parallel processors. Figure 2 illustrates a strongly abstracted architecture view on a GTX285 from NVIDIA. Other vendors' products, such as those from AMD, differ only slightly from this architectural view.

The GPU consists of a two-level hierarchy that can be found on processor and memory level. The GTX285 has 24

[1]The original idea was submitted to the "First Annual SIGMOD Programming Contest – Main Memory Transactional Index" by the DEXTER team and was chosen as one of the 5 fastest submitted implementations.

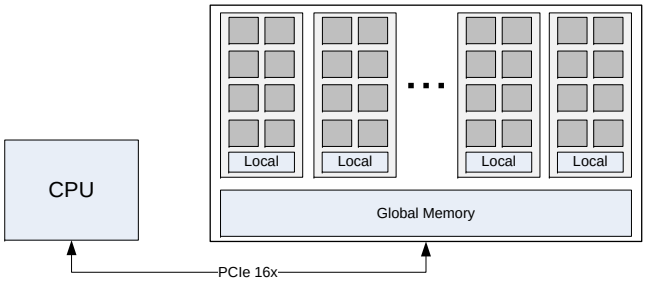


Figure 2: Schematic overview of a GPU architecture.

multiprocessors (MP) and a global memory. Each MP consists of 8 processors and a local memory, resulting in a total of 240 processors. While all data in the global memory can be accessed from all processors, data in the local memory is accessible only from within the same thread. For synchronization purposes, the global memory also includes a shared memory region. This region can be accessed by all threads that are executed within one block. The local and shared memory can be leveraged by software code only and will be flushed when the function being executed on the GPU is complete. Therefore, the shared and local memory represent a software-controlled cache. The size of shared memory is limited to 16kBytes, compared to a maximum of 4GB for the global memory. Data in the global memory is persistent throughout the application runtime. Furthermore, local memory can be accessed within 4-6 cycles, compared to 400-600 cycles of latency on the global memory. With such high latency differences, it is important to leverage the local memory as much as possible. To transfer data, the GPU is connected with a PCIExpress connection. The current theoretical maximum throughput is 8GByte/s (version 2).

To hide the high latency of the global memory, the GPU offers two important features. The first is coalesced memory access. This feature bundles multiple memory requests of processors into one request. This way, high memory bandwidth of 86GByte/s can be used more efficiently. The second feature is fast thread switching. While a thread switch is very costly on the CPU, the GPU can handle thread switching with more ease. It is therefore encouraged to create more threads than are available as physical execution units. This overload can then be used to schedule threads for execution, while others wait for a memory transfer. This is especially important since databases are more likely I/O-bound, not CPU-bound, and it is thus one of the most important features for implementing database operations on the GPU. We will come back to this in later chapters.

The GPU can be employed in multiple ways. Previously, the data has been hidden in structures for graphical processing, such as triangles, and then graphical programming languages like GLSL or DirectX have been used to perform image manipulation functions on the data, resulting in the desired mathematical operation on the data. This has required good knowledge of graphics processing and graphical programming languages. CUDA and Stream are extensions to the C programming language that enable the programmer to write native C code and execute it on the GPU. The methods introduced in this paper are implemented using the CUDA extension exclusively.

3. SPECULATIVE TREE TRAVERSAL

As described in Section 2.1, the performance of a database index structure depends on the efficiency of the $get(key)$ operation. However, this operation normally corresponds to sequential traversals of the underlying tree—from the root to a leaf. To increase the performance and to leverage the high number of available cores on a many-core processor (like a GPU), we introduce our speculative approach for tree traversal. Our goal is not to minimize the number of instructions used to the find the result but to utilize the high number of cores in an efficient and novel way. In this case, we assume that many-core architectures offer enough computation capabilities to allow us to perform some speculative (and sometimes redundant) tasks that we would normally avoid in today's algorithms.

Our concept of speculative tree traversal consists of two steps:

1. Parallel traversal of (all) partitions of the tree, and
2. Aggregation of intermediate results to the final result.

The parallel traversal of the tree uses the high number of cores on a GPU. In its initial implementation, all possible partial results are created. Then, the intermediate results are used to determine the final result. The second step is, again, a traversal of the intermediate results, which is implemented as a serial traversal. To further enhance the performance, we also introduce a method to create the final result in parallel.

3.1 Partition-Based Traversal

To leverage the high number of cores of a GPU, we partition the tree into multiple computational trees. Each partition C_j starts at a specific tree level k and covers m levels. At each leaf of a computational tree, the next computational tree starts, until the leaf nodes of the complete tree have been reached. Each computational tree is assigned to one thread on the GPU. Therefore, each level may consist of one or multiple computational trees, and each thread is responsible for traversing its tree for a given sub-key K_i according to its starting level.

To traverse the tree for a key K, each partition provides one of three different types of results. The result may contain a reference to yet another computational tree. This is the case if the partition is in the middle of the respective tree and if it contains a path for its substring. The result of a computational tree can also be NULL, meaning that the computational tree does not contain any matching path for its substring. The third possible result is a pointer to a payload. This is the case if the computational tree contains leaf nodes of the original prefix tree.

The result of the partition is then entered into a global list for the intermediate results. Since the start addresses of a partition are known, the result of a partition can be used to determine which partition the pointer leads to. Finally, the global list has to be traversed starting from the first partition. Compared to a traversal of the original tree, only a fraction of traversal steps are needed here. Furthermore, the benefits of the computational trees are that they are independent of each other and can be computed in parallel.

Figure 3 illustrates an example for this parallel traversal. It shows a tree for a 12-bit integer. Each node has 8 entries, resulting in a sub-key length of 3 bits and in a tree depth of

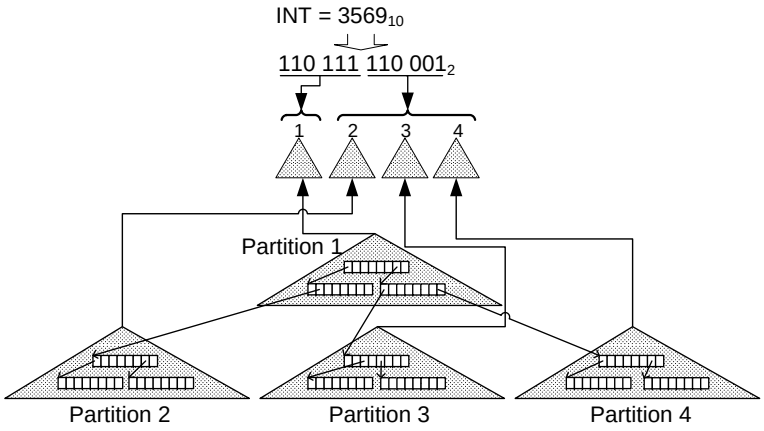


Figure 3: Speculative Traversal of a tree with a depth of 4 for the 12-bit integer 3569.

4 nodes. The tree is first partitioned into 4 computational trees. Note that only those partitions are created that are needed. This can be implemented by keeping a list of start pointers for a specific tree level that is maintained during the insertion of data. Subsequently, each of the 4 computational trees is traversed in parallel, and the intermediate results are entered into a global list, as illustrated in Table 1. To determine the final result, this intermediate result is traversed in two steps. Computational tree no. 3 did not lead to any result compared to trees $1, 2$ and 4. They contain a pointer to a payload or to a subsequent partition. The traversal of the intermediate result starts with the first computational tree, since it contains the root element. Its result is a pointer to the fourth computational tree. Finally, the result of the fourth computational tree is the pointer to the resulting payload. Therefore, we need a total of 4 traversal steps to retrieve the final result.

Sub-Tree	Result
1	4
2	Payload
3	NULL
4	Payload

Table 1: Result of the traversal from Figure 3.

To quantify the possible benefits of this approach, we will analyze it on a theoretical level, showing that for larger trees, this method reduces the number of traversal steps tremendously and, therefore, speeds up all operations on this tree. Further, we will use N as the number of levels within the tree. If all computational trees are executed completely in parallel, the number of tree levels each thread has to pass is equal to m. The number of steps required to build the final result can be determined by $k = \frac{N}{m} - 1$. Therefore, the total number of nodes that need to be traversed is $n = m + k = m + \frac{N}{m} - 1$; the maximum speedup w.r.t the number of nodes traversed is:

$$s = \frac{N}{n} = \frac{N}{m + \frac{N}{m} - 1}. \quad (1)$$

The global maximum of equation 1 is at $m = \sqrt{(N)}$. Therefore, the maximum theoretical speedup compared to a sequential traversal of the tree can be determined by:

$$s_{max} = \frac{N}{(2 * \sqrt{(N)} + 1)}. \quad (2)$$

This maximum speedup can only be reached if memory requests of each thread are coalesced and therefore answered simultaneously. Furthermore, it is necessary that the hardware can hide the remaining memory latency by switching threads into hardware that is not waiting for a memory transfer to be completed. These features are available and very well developed on the NVIDIA GPUs.

The possible speedup is limited by equation 2, by the maximum throughput between memory and processors, and by the number of available processors. Furthermore, the last step of building the final address is executed in serial. This is one of the bottlenecks of this approach, which we will reconsider later on. Furthermore, the possible maximum speedup via equation 2 is only very limited for larger trees and decreases with an increasing number of nodes. Since one of the main goals is the ability to apply a prefix tree to larger keys, it is necessary to perform further optimizations. To increase the speedup of this approach further, more features of the GPU must be exploited, like the local memory with very low latency to the processor. This is especially necessary since equation 1 shows that small computational trees result in a large table with intermediate results that can only be traversed in serial fashion.

3.2 Hierarchical Traversal

To increase the possible speedup further, we adopt the method of parallel traversal to include the fast local memory and to reduce the number of traversal steps on the global memory to build the final result from the intermediate one. To achieve this, we introduce another hierarchy similar to the processor hierarchy on the GPU.

Threads that are executed on the GPU are grouped into blocks. All threads within a block can share data via the shared memory. Furthermore, shared memory can be accessed within 4-6 processor clock cycles, meaning a low latency compared to the global memory. To utilize this shared memory, we group computational trees into blocks. Within each block, the threads write the intermediate results of their computational tree into the shared memory. Then, after all threads of one block have completed their traversal, the in-

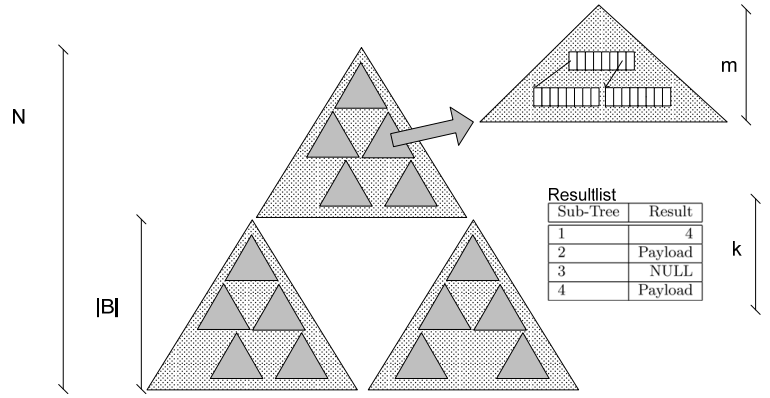


Figure 4: Legend of variables within the tree architecture.

termediate result of the block is traversed. This final result of the block is then entered into the result list in the global memory to build the final result from the intermediate results from the blocks, similar to the previous approach.

This approach results in an additional traversal step but increases the speed by leveraging the fast shared memory of the GPU. The challenge with this additional step is to ensure that the computational trees within one block depend on each other, meaning that no or only very few partitions link to a partition located in a different block. To guarantee this, we split the size of the computational tree and create sub-partitions within this tree. Each sub-partition is then traversed by a separate thread, similar to the previous approach.

To analyze the maximum speedup compared to the previous approach, we need to incorporate more variables into our model. An illustration of the variables is given in Figure 4. The speedup depends on the number of blocks B generated, the height of a block $|B|$ in the number of nodes, the height of computational trees, and the latency relationship between global G and shared memory $R = G/L$.

The number of intermediate results generated in the shared memory to determine final results per block depends on the block depth $|B|$ and can be determined by $|B|/m$. Succeeding this is the number of traversal steps in global memory $= N/(|B|)$. The traversal in global memory is needed to create the final result. One of the limiting factors of the previous approach was the bandwidth between global memory and the processors. As the number of parallel traversals within one block is still equal to m, the bandwidth usage is still a limiting factor. In the following, we will analyze the possible speedup compared to the previous approach. The number of memory accesses $k_{hierarchical}$ of the parallel node traversal must be split into shared and global memory accesses.

$$k^{shared}_{hierarchical} = \frac{|B|}{m} \tag{3}$$

and

$$k^{global}_{hierarchical} = m + \frac{N}{|B|} \tag{4}$$

Access to the global memory is only necessary to determine the final results from the intermediate block results. As a next step in our analysis, we incorporate the latency factor R. Since the latency to the global memory is a lot higher than to the shared memory, we add a penalty to the global memory access.

$$k_{hierarchical} = k^{local}_{hierarchical} + k^{global}_{hierarchical} * R \tag{5}$$

$$= \frac{|B|}{m} + (m + \frac{N}{|B|}) * R \tag{6}$$

Therefore, the speedup of the approach with shared memory compared to the approach without shared memory can be determined by:

$$s = \frac{(m + \frac{N}{m} - 1) * R}{\frac{|B|}{m} + (m + \frac{N}{|B|}) * R}. \tag{7}$$

The maximum speedup can be reached by using $|B| == N$, meaning that the complete tree is traversed within one block. In this case, the final result would be determined completely in the fast local memory. This would result in a maximum speedup compared to the non-hierarchical approach of:

$$s_{max} = R. \tag{8}$$

This theoretical maximum could only be reached if we were able to dispatch unlimited numbers of threads in each block to calculate all computational trees. Since architecture limitations of the GPU prohibit unlimited numbers of threads within one block and since there are limitations on the size of the shared memory, this speedup can only be reached for small trees, as our evaluation will show.

3.3 Parallel result building

As the analysis of the previous section has shown, the determination of the final result is a sequential task and requires $k = \frac{N}{m} - 1$ steps. If this fraction can be parallelized, the total number of sequential steps would be reduced to the number of traversals within a partition $\frac{N}{m}$. In special cases, the number of steps required within the final result can be reduced. If the intermediate result table contains exactly one pointer to a payload, then this pointer is the final result. No other partitions containing leaf nodes have found a path within their containing nodes. Furthermore, keys containing a common beginning, for example, Web URLs, only need to be compared in the last traversal steps.

Therefore, we propose a new method to find the final result. First, we start a thread for each pointer to a payload

Sub-Tree	Result	Payload
1	NULL	NULL
2	NULL	payload
3	NULL	NULL
4	1	payload

Table 2: Reversed intermediate result table from Table 1.

within the result table. Then, each thread checks the entries within the table that reference this entry. This is repeated until no further referencing entry is found or until the referencing entry is the root node. If, during the traversal, all threads except for one are terminated, then the thread must have started at the pointer to the payload that is the final result.

Given the example in Table 1, a thread is created for every entry containing a pointer to a payload. In this example, 2 are created. Further, the threads determine the entries in the table that point to this element. For the payload in partition 2, no parent partition can be found, and therefore, this thread can terminate. Sub-tree 4 is referenced by the root partition. Since the thread from partition 4 is the only thread running and the only one that has found a parent, it must be started at the pointer to the final result.

The challenge with this approach is to find the referencing partitions. The naïve implementation of such a search would be to scan the complete intermediate result table. Since this is very inefficient, we propose to change the intermediate result table to a reversed result table. The entries within it do not point to the successor but to the predecessor. The reversed intermediate result table from Table 1 is illustrated in Table 2. As the reversed table shows, it is necessary to save the pointer to the payload if the result of a partition is a payload.

4. IMPLEMENTATION

Our implementation of speculative pre-computation for the GPU is based on CUDA 3.1 developed by NVidia. Using this technology, it is possible to write c/c++ style functions, called *device* or *global* kernels, that are compiled with *nvcc* and are executed via the runtime environment on the GPU. It is only possible to execute these functions on the GPU. Functions written for the CPU have to be rewritten, such as functions for memory management. Since the insertion of new data requires the allocation of new tree nodes, a memory management must be implemented to function on the GPU. Furthermore, to leverage such functions as coalscaled memory access the data layout must be optimize to avoid conflicts. In this section, we will illustrate the basic architecture of the index system and outline how we optimized the memory layout.

4.1 Basic Architecture

Index systems of database systems do not only consist out of the index but also requires metadata and links to the actual payload. Since the result of the search within an index is used to perform further operations on the result, the data must be located where the next operation will take place. Figure 5 illustrates our main architecture. The dictionary contains metadata to the index, such as name and

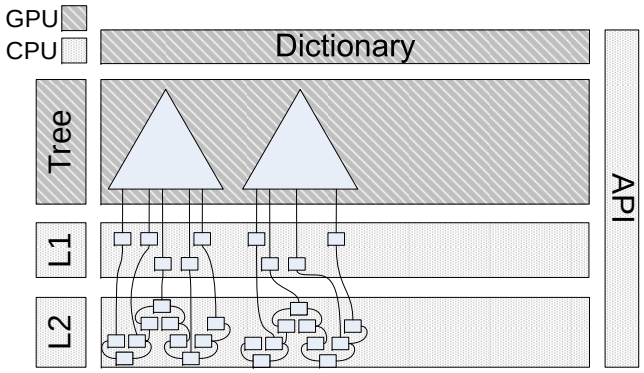


Figure 5: Architecture and location overview.

type. Parts of the dictionary, such as the name, are located on the GPU and on the CPU memory. This makes changes and creation expensive but minimizes data transfer between the host and the device. The metadata on the GPU is used to search for a specific index. If the user opens an index with a specific name the GPU is used to locate it within the system. For this one thread per existing index is created which compares its name with the name that is being searched for. Once the index is found a pointer is returned to the CPU to perform further operations. This is especially useful if many indexes exist within the system.

As Figure 5 shows, the complete index resides on the GPU. The *insert*() or *get*() functions are executed solely on the GPU. Therefore, only minimal data transfer is needed. The limiting factor fur such a design is the amount of available GPU memory. Current systems provide up to 4GB of global memory (NVidia Tesla S1060). Since a prefix tree performs a type of compression we can store strings exceeding the 4GB limit. Furthermore, through our memory layout it is possible to page a complete index from the GPU to CPU ram if necessary.

As depicted in the previous chapter, leafs of the tree contain the payload. These $L1$ items are located in CPU RAM together with the payload list. The leaf nodes on the GPU only contain the address of this $L1$ item within the address space of the CPU RAM. This way only the address of the $L1$ item and the key must be transferred to the GPU when inserting a tuple. The $L1$ Items contain a pointer to the actual payload. To support efficient scan operations, the payloads are within a globally linked list reducing the need to find the next lexical element within the tree.

4.2 Memory and Thread Layout

Finding an efficient memory layout is essential to leverage coalscaled access capabilities of the GPU. The most critical part is the creation of the node structure. Current NVidia GPU can coalscaled memory access from 16 parallel running threads (half warp) on the same multiprocessor. Therefore, it is essential that within a half warp the memory layout is build such that the memory requests can be coalscaled by the hardware. For each partition level we create a multiple of 16 threads and partitions. Each thread traverses one partition of the same level. Each of these partitions is filled in parallel during the insertion of the data into the index. The only difference between the partitions is the value of the leaf node of the partition. Hence each partition has the same

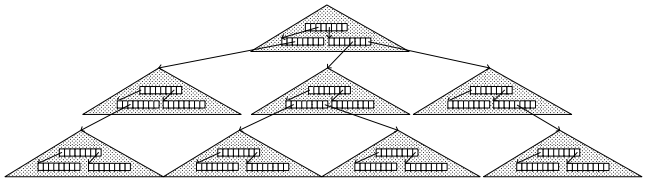


Figure 6: Illustration of parallel filled partitions with the same layout.

number of nodes and the same layout as Figure 6 illustrates. This method creates a higher memory footprint than an allocation of memory only when necessary. If a dataset contains many common substrings with different prefixes then only minimal overhead is created compared to an allocation on request. An allocation on request would randomly create addresses and hence limit coalscaled access.

The only exception is the root partition since there is only exactly one root partition we create only one partition but dispatch 16 threads to traverse it. From these threads 15 terminate immediately. This is necessary to guarantee that all other partition levels are aligned with 16 threads.

Besides coalscaled access this method of allocating the partitions also minimizes the number of divergent Branches. Divergent branches are created when a thread within a half warp has a different execution flow as the other thread. When for example an if condition evaluates $true$ within one thread and $false$ in another thread then the threads are partially serialized reducing the parallelism. Since each thread within a half warp traverses the same path the evaluation of the conditions are the same. Divergent branches create a significant overhead since the scheduler needs to find a new schedule for this new set of operations.

Since dynamic memory allocation can only be performed from the CPU the insertion function requires their own memory management. Memory management is based on pages. Each page consist of G number of bytes. A pointer determines where the next free byte is located within this page. Before an insert operation is executed the system determines the number of free bytes within the current page. If not enough memory is available then another page is allocated and is added to the index. Therefore, the index maintains a list of memory pages it may use. For this setup deletes of nodes are not considered. Each page can be considered as an array of bytes. The actual address that is written into a node of the tree is not the physical address but a reference on into a specific page. With this relative addressing we can move the complete index from and to any GPU device, in a multi device setting, without having to modify the pointers in the tree.

5. RELATED WORK

Multiple approaches to perform relational database operations on a GPU (join) have been proposed in [6, 7]. The approach used to speed up database operations is to partition the data and then to compute each partition in parallel. With this method, strong data dependencies still exist and only a limited parallelization usage of the high number of cores is possible. For spatial databases, similar approaches have been proposed in [2, 3]. Furthermore, all approaches focus on complete operations of databases and not on single elements such as indexes. Index structures for the GPU have found attention in [5, 4]. The goal of [5] is to create a compression and evaluation strategy for bitmap indexes for massively multi-threaded platforms. One of the major points of criticism of this approach is that bitmap indexes are costly to maintain. In [11], the authors use a special layout of a binary tree to optimize the search strategy. Compared to our approach, their concept is update-unfriendly since a complete rebuild of the tree is necessary to insert a tuple. Furthermore, [20] proposes a SIMD-based traversal of a tree to optimize the tree traversal for multiple parallel searches. This differs from our approach, since we use a data-parallel traversal. Furthermore, the previous approaches use a binary tree as a base, compared to our prefix tree using N bits as a prefix. A binary search tree results in a deep tree for long key values, reducing its performance. In contrast, our approach uses multiple bits per node and, therefore, results in shorter trees and a reduction of the total number of traversal steps required to resolve a search query.

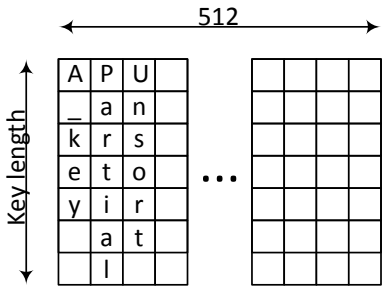


Figure 7: Naive index.

Beyond of the field of databases the method of speculative computation is widely adopted for parallel memory machines as [8, 17, 18] illustrate. Here the authors also compute partial results although they might not contribute to the final result. Precomputation is here applied to a domain containing data dependencies within complex calculations. Distributed memory machines have no global memory and offer different optimization possibilities. The research focus lies on when to push data from what computing node to another. Yet the computing node consist only of a single computing instance and not, like a GPU, of a highly thread parallel system. The GPU furthermore offers other possibilities to optimize the parallel execution and parallel memory access as the previous section illustrates.

6. DISCUSSION AND EVALUATION

We evaluated our proposed approaches against the sequential traversal of the tree, performing N steps on one core on the GPU and CPU, and against a naïve approach. The naïve approach creates a thread for every key within the index. To ensure that the naïve implementation also uses coalesced memory access, we have arranged the data in a matrix containing 512 keys, as Fig. 7 shows. With this organization, each thread accesses a character in ascending address order. Once a matrix is full, another matrix is created. Each matrix is executed as one block on the GPU. All data is unsorted within the matrixes.

For the evaluation, we used a single GTX285 GPU with 240 cores with 1.48GHz per core and a memory clock rate of 1.2GHz running on a Windows 2008 PC and a CUDA driver

of version 3.1. To compare the GPU performance with a CPU implementation, we used an Athlon 64 X2 Dual Core 4200+ with a clock rate of 2.2GHz. To measure the number of serialized blocks, we used NVIDIA's Parallel NSight.

First, we evaluated the performance of the approaches with multiple different artificial data sets containing different data skews in addition to a real dataset. For the following experiments, we use strings as keys with a length of 128 characters. Since the speculative approach creates one thread per partition, the first dataset creates a new partition with every insert. This is the worst case for our speculative approach, since it maxes out the number of threads used. Figure 8(a) illustrates the result of this experiment. As we can see, the speculative approach performs better than the linear traversal and the naïve approach. Since the number of tuples within the index determines the number of partitions, and thus the number of threads used to traverse the tree, the performance degrades. As soon as the GPU hardware has more threads that can fully run in parallel, threads and blocks are serialized.

The second data skew is optimal for the tree construction. Every insert creates a new leaf node by incrementing the least significant byte by one for every new string. With this data skew, only the minimal number of partitions is created, since the tree is only expanded on the very last level. Figure 8(b) illustrates the result of this experiment. Clearly, it shows that the performance is almost always better than with the naïve approach and the linear traversal of the tree. The performances of the naïve approach and the linear traversal are equal to the performance with the previous data skew. This shows that the performances of these approaches are independent of the data skew. Furthermore, it can be seen from the two experiments that the linear tree traversal is independent of the number of tuples compared to the naïve approach. This is due to the tree structure. The number of levels within the tree is independent from the number of tuples within the tree. Since a new thread is created for every new tuple in the naïve approach, it suffers from the same effects as the parallel traversal does. As soon as a specific number of tuples is reached, the GPU hardware starts to serialize the threads and blocks.

The third data skew contains URL-like strings. Their initial segments are the same, but their last segments are very heterogeneous. We used the scaped URL data from the facebook directory. This dataset contains more than 100,000 urls with an average length of 60 characters and a maximum length of 147. It consists out of the Facebook URL, the user name and an integer value being the user ID within Facebook. This type of dataset can be used to show the effects of the parallel traversal of the intermediate result. Figure 8(c) shows the result of this experiment. The performance gain that can be achieved by this additional approach is only very limited but adds another 10% performance increase for this specific data skew. As the previous experiments have shown, the parallel traversal of the intermediate results shows only very little to no performance degradation compared to the sequential traversal of the intermediate result.

Aside from the data skew, the performance of our approach depends on the number of levels within a partition. In the previous examples, we used the theoretical optimum. Figure 8(d) illustrates the performance of our approach with the first dataset with different numbers of levels within a partition. Clearly, the experiment shows that the theoretical optimum is also the optimum for the implementation.

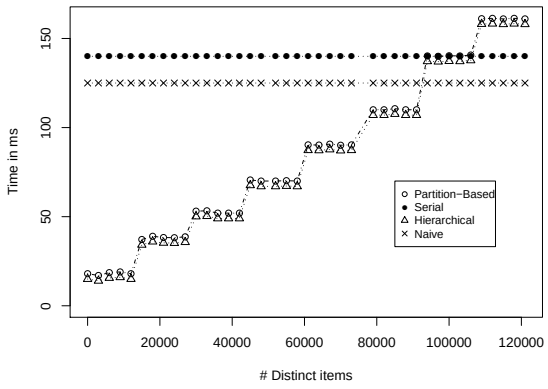
Due to the method how the memory layout is orchestrated, it is important to measure the memory consumption. Figure 8(e) shows an experiment for different dataset sizes from the different data skews from above and compared to a naïve implementation. The naïve implementation allocates data on demand and therefore creates a randomized memory layout minimizing the possibility of coalesced access. The modified memory allocation creates for the Facebook dataset only minimal overhead since the lower partitions are very heterogeneous and new partitions are created very often. The minimal tree in comparison creates a larger overhead since the partitions are completely filled and the parallel partitions for the levels are also filled.

One of the main points of criticism for the usage of the GPU as a co-processor for database operators is the limited bandwidth between CPU and GPU memory, which is currently limited to 4GB/s. Since our approach keeps the data persistent on the GPU's main memory, such criticism only applies to a very limited extent. Figure 8(f) shows another very important observation. In the previous experiments, we only measured the runtime of the kernels executing the search on the GPU. To execute this kernel, it is necessary to leverage the CUDA runtime. This adds significant overhead to a single operation. As Figure 8(f) shows, it is not the data transfer rate that limits the performance but the CUDA runtime overhead, since our approach only transfers very small tuples to the GPU memory. The runtime API is mapped during the compilation to the driver API. This mapping can be enhanced with specific application knowledge, such as that the system runs in a single threaded environment. Since NVidia offers the possibility to use driver API directly to start kernels on the GPU, we implemented our own intermediate management layer to replace the CUDA runtime based on the driver API. This intermediate layer reduces the necessity for specific calls and also minimizes the management overhead. Figure 8(f) shows that our implementation on top of the driver API reduces the overhead significantly.

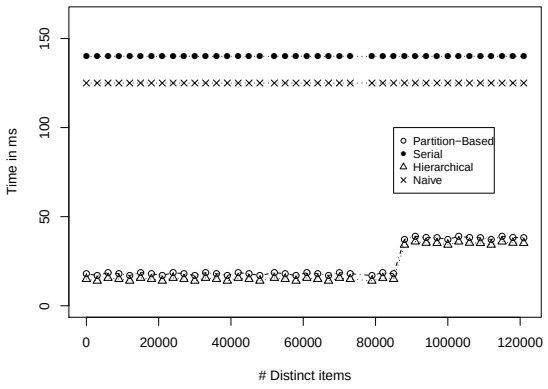
7. CONCLUSION AND OUTLOOK

With an increasing amount of data and user demands for fast query processing, the optimization of database operations continues to be a challenging task. A common optimization method is to leverage parallel hardware architectures. With the introduction of general-purpose GPU computing, massively parallel hardware has become available within commodity hardware. In this paper, we have presented our novel developed approach of speculative processing of database operations for highly parallel systems. As an example, we realized this approach for the search on a prefix index to speed up search queries. Fundamentally, our concept of speculative tree traversal consists of two steps: (i) parallel traversal of (all) partitions of the tree, and (ii) aggregation of intermediate results to the final result. As our exhaustive evaluation showed, our approach scales well for different data skews and sizes. During the implementation, we leveraged the driver API to reduce the overhead generated by the CUDA runtime.

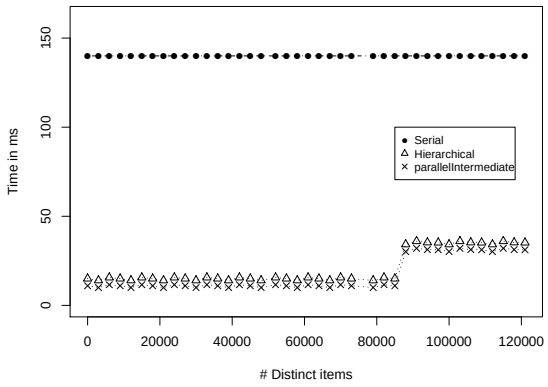
This first implementation of our approach showed the applicability to a small task within a database system. In the future, we are going to apply this speculative idea to other areas within database systems, for example, to joins



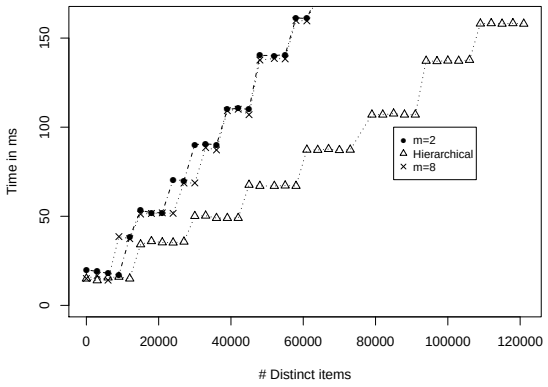
(a) One partition with every new insert.



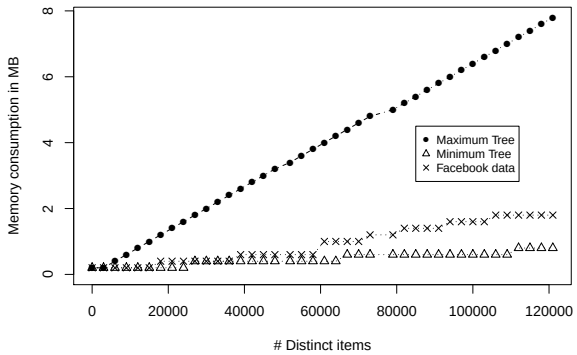
(b) Minimal tree.



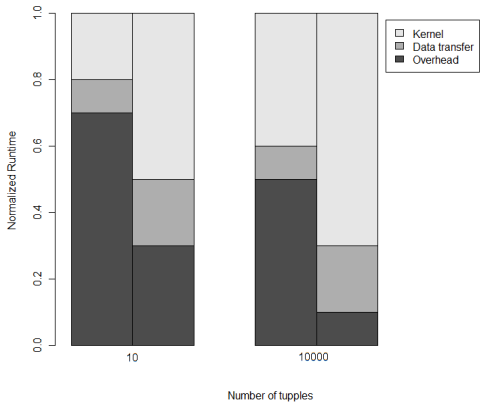
(c) Heterogeneous strings.



(d) Different numbers of levels within a partition.



(e) Memroy consumption for the different datasets



(f) Time consumption of the experiment

Figure 8: Evaluation Results of Our Speculative Approach.

or groupings as intra-operator speculative processing. Aside from the fine-grained speculative processing, this method could also be applied to a complete execution plan of a query by massively using parallel threads to execute operations and then merging the result together in a subsequent step. However, the speculative processing has its limits since the available resources can be exhausted. Hence, it is essential to develop pruning methods, such as the reverse traversal of the intermediate result, to efficiently leverage the hardware for scalable problems.

8. ACKNOWLEDGMENTS

This work has been supported under the terms of a NVidia Proffesor partnership. The source code is available at `http://wwwdb.inf.tu-dresden.de/research/dexter`.

9. REFERENCES

[1] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The claremont report on database research. *SIGMOD Rec.*, 37(3):9–19, 2008.

[2] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1021–1032. VLDB Endowment, 2004.

[3] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1021–1032. VLDB Endowment, 2004.

[4] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther. Density-based clustering using graphics processors. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 661–670, New York, NY, USA, 2009. ACM.

[5] L. J. Gosink, K. Wu, E. W. Bethel, J. D. Owens, and K. I. Joy. Data parallel bin-based indexing for answering queries on multi-core architectures. In *SSDBM*, pages 110–129, 2009.

[6] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proc. of ACM SIGMOD*, pages 215–226. ACM Press, 2004.

[7] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, New York, NY, USA, 2008. ACM.

[8] H. Hong and H. Loidl. Parallel computation of modular multivariate polynomial resultants on a shared memory machine. 854:325–336, 1994.

[9] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 309–320, New York, NY, USA, 2009. ACM.

[10] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *DASFAA*, pages 761–772, 2005.

[11] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD Conference*, pages 339–350, 2010.

[12] J. Liedtke and K. Elphinstone. Guarded page tables on mips r4600 or an exercise in architecture-dependent micro optimization. *SIGOPS Oper. Syst. Rev.*, 30(1):4–15, 1996.

[13] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[14] H. B. Paul, H. J. Schek, and M. H. Scholl. Architecture and implementation of the darmstadt database kernel system. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 196–207, New York, NY, USA, 1987. ACM.

[15] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[16] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 52–60, New York, NY, USA, 2009. ACM.

[17] R. Schröder and W. E. Kluge. Organizing speculative computations in rule-based systems. In H. R. Arabnia, editor, *PDPTA*. CSREA Press, 2000.

[18] A. Sohn. Parallel n-ary speculative computation of simulated annealing. *IEEE Trans. Parallel Distrib. Syst.*, 6(10):997–1005, 1995.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[20] A. D. B. T. Kaldewey, J. Hagen and E. Sedlar. Parallel search on video cards. In *In USENIX Workshop on Hot Topics in Parallelism*, 2009.

[21] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[22] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A parallel hash join algorithm for managing data skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1355–1371, 1993.