QPPT: Query Processing on Prefix Trees

Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner
Database Technology Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Modern database systems have to process huge amounts of data and should provide results with low latency at the same time. To achieve this, data is nowadays typically hold completely in main memory, to benefit of its high bandwidth and low access latency that could never be reached with disks. Current *in-memory databases* are usually column-stores that exchange columns or vectors between operators and suffer from a high tuple reconstruction overhead. In this paper, we present the *indexed table-at-a-time* processing model that makes indexes the first-class citizen of the database system. The processing model comprises the concepts of intermediate indexed tables and cooperative operators, which make indexes the common data exchange format between plan operators. To keep the intermediate index materialization costs low, we employ optimized prefix trees that offer a balanced read/write performance. The *indexed table-at-a-time* processing model allows the efficient construction of composed operators like the multi-way-select-join-group. Such operators speed up the processing of complex OLAP queries so that our approach outperforms state-of-the-art in-memory databases.

1. INTRODUCTION

Processing complex OLAP queries with low latencies as well as high throughput on ever-growing, huge amounts of data is still a major challenging issue for modern database systems. With the general availability of high main memory capacities, in-memory database systems become more and more popular in the OLAP domain since often the entire data pool fits into main memory. Hence, the superior characteristics of main memory (e.g., low access latency, high bandwidth) could be exploited for query processing, which leads at the same time to a paradigm shift in query processing models.

The traditional tuple-at-a-time processing model [8] was found to be sufficient for row-stores as query processing was limited by the I/O of disks. Newer processing models, e.g., the column-at-a-time [6] or the vector-at-a-time processing model [7], avoid certain drawbacks of the traditional query processing (e.g., massive virtual function calls) and are thus more suitable for modern in-memory column-stores. However, the logical unit of a tuple as present in row-stores is decomposed in columns resulting in an expensive tuple reconstruction overhead. The complexity of the tuple reconstruction procedure grows with an increasing number of attributes involved during query processing.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2013.
6th *Biennial Conference on Innovative Data Systems Research (CIDR '13)* January 6-9, 2013, Asilomar, California, USA.

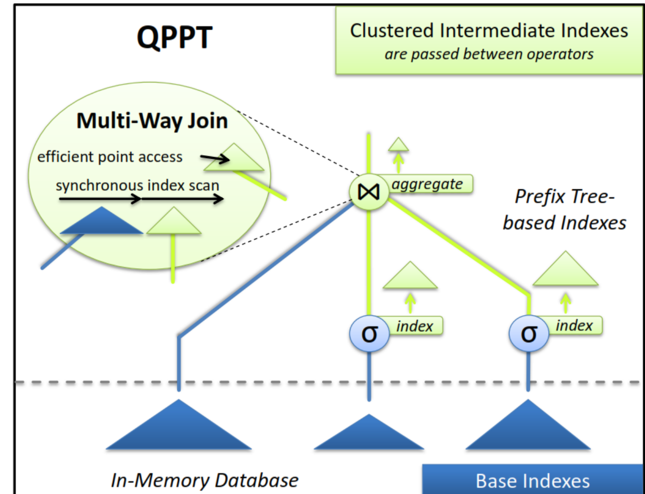
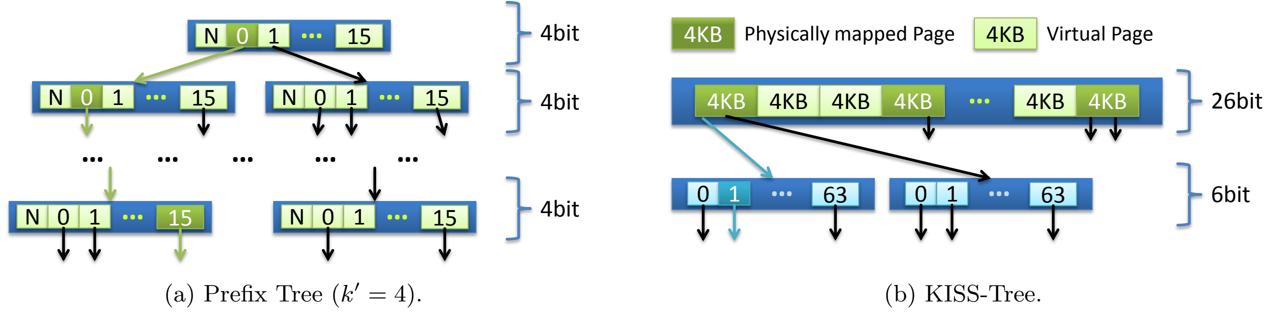


Figure 1: Overview of QPPT's *indexed table-at-a-time* Processing Model.

To avoid the tuple reconstruction overhead, we propose to return to row-oriented systems enhanced with a novel up-to-date processing model. In this paper, we introduce our new *indexed table-at-a-time* processing model for in-memory row-stores, which is depicted in Figure 1. The main characteristics of this processing model are: (1) intermediate indexed tables, (2) cooperative operators, and (3) composed operators.

Intermediate Indexed Tables

Instead of passing plain tuples, columns, or vectors between individual operators, our *indexed table-at-a-time* processing model exchanges clustered indexes (a set of tuples stored within an in-memory index) between operators. In this way, we (i) eliminate the major weakness of the volcano iterator (tuple-at-a-time) model by reducing the number of next calls to exactly one and (ii) enable data processing using highly efficient operators that can exploit the indexed input data.



(a) Prefix Tree ($k' = 4$).

(b) KISS-Tree.

Figure 2: Index Structures Deployed in *QPPT*.

Cooperative Operators

Based on the previous point, each operator takes a set of clustered indexes as input and provides a new indexed table as output. Our second concept is called cooperative operators and enables a more efficient data handling within a query execution plan. An operator's output index is always indexed on the attribute(s) requested by the subsequent operator. Thus, the complete set of indexed intermediate tuples can be transfered from one operator to another by passing a single index handle; exchanging plain tuples, which would be used eventually to build internal indexes in the next operator anyway, is not required anymore. A selection operator, for example, takes a base index as input that is indexed on the operators selection predicate and outputs an intermediate table that is indexed on the join predicate of a successive join operator.

Composed Operators

The set of operators is usually small using the first two concepts. We can omit all non-index operators and keep only operators that can exploit indexes as input, i.e., that make use of efficient range scans and point accesses within the indexed data. Also some base operators like sorting/grouping and aggregation are not necessary anymore, because each operator already indexes its output.

The *indexed table-at-a-time* processing model allows further *composed (n-ary) operators*, which additionally perform sorting/grouping or aggregation on the data being outputted. Using our composed operators, we are able to reduce expensive materialization costs and data exchange between operators to a minimum, which is highly benefical as presented by recent compilation based approaches [12]. For this reason, composed operators form the core components of our novel processing model.

Query Processing on Prefix Trees

One important aspect of our *indexed table-at-a-time* processing model is the output index generation within each operator. It amounts to a large fraction of an operator's execution time and thus must be fast. Previous research revealed that prefix trees [5] offer all the necessary characteristics (i.e., they are optimized for in-memory processing and achieve high update rates) for our approach. Additionally, we employ the KISS-Tree [9] as a more specialized version of the prefix tree, which offers a higher read/write performance than hash tables and allows even more effective batch processing schemes. Based on these data structures, we introduce *Query Processing on Prefix Trees (QPPT)*, which is a realization of the *indexed table-at-a-time* processing model.

Figure 1 shows a *QPPT* overview. Leaf-operators, like the selection, access the base data via a base index and build an intermediate prefix tree as output. The successive multi-way/star join operator uses the intermediate indexes of child operators and one base index. Here, the multi-way/star join as an example of an composed operator executes a *synchronous index scan*, which is an efficient join algorithm that works on unbalanced trees, like the prefix tree. As result, the multi-way/star join builds an index that contains already grouped and aggregated values.

Structure of This Paper

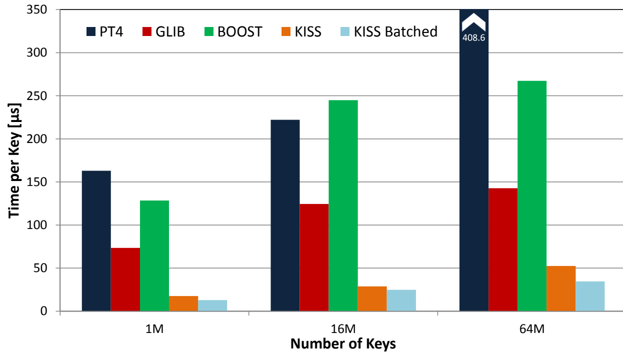
The following section gives a short introduction in prefix trees as prerequisite for our novel query processing model. Moreover, this section demonstrates the advantages of prefix trees over hash tables in terms of insert and lookup performance. A general overview on our novel query processing on prefix trees as a realization of the *indexed table-at-a-time* processing model is presented in Section 3. The aspects of cooperative and composed operators are discussed in Section 4, followed by an exhaustive evaluation in Section 5. The paper closes with related and future work (Section 6 and 7) as well as a conclusion (Section 8).

2. QPPT PREREQUISITES

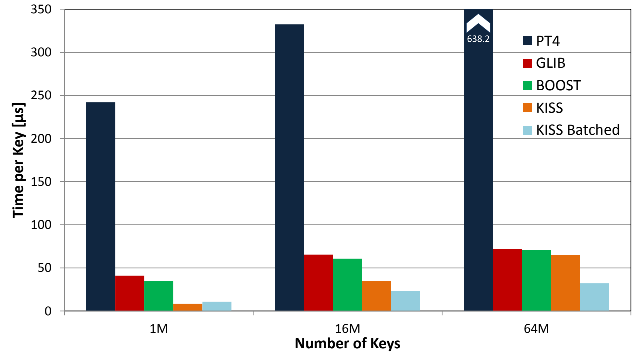
The performance of our *indexed table-at-a-time* processing model heavily depends on the index structures that are deployed for query processing. Traditional index structures like B+-Trees [4] and its main memory optimized version, the CSB+-Tree [14], are not suitable because they achieve only a low update performance. Instead, we employ prefix trees [5], because

1. Prefix trees are optimized for a low number of memory accesses per key, which results in a high key lookup performance and
2. They offer a balanced read/write performance, which is essential for our *indexed table-at-a-time* approach.

Additionally, we deploy a modified KISS-Tree [9], which is a better performing version of the prefix tree, but is limited to 32bit keys. In the following, we briefly discuss both data structures in the context of *QPPT*. Moreover, we introduce the concept of batch processing on prefix trees and an efficient way of handling duplicates to allow an even more



(a) Insert/Update Performance.



(b) Lookup Performance.

Figure 3: Performance of Prefix Trees compared to Hash Tables.

effective query processing on those prefix tree-based structures. Finally, we compare the insert/lookup performance of our indexes to different hash table implementations, because traditional join or group operators usually create hash tables—when its is beneficial—internally.

2.1 Prefix Tree

The employed prefix tree is an order-preserving and—contrary to the B+-Tree—an unbalanced tree[1]. It works by splitting the binary representation of a key into fragments of an equal prefix length k'. The fragments are used to navigate within the tree and thus a key itself has a fixed position in the tree. Figure 2(a) shows the structure of a prefix tree with the parameter $k' = 4$. Each node of the tree consists of $2^{k'} = 16$ *with* $k' = 4$ buckets, each containing a pointer to the next node, which takes the next 4 bits of the key for locating the corresponding bucket inside this node. This tree traversal scheme continues until the point, where the pointer in the bucket points to a content node or zero. Because of the dynamic expansion feature [5] of the prefix tree, the key can not always be reconstructed from the path through the tree. Thus, the content node has to store the complete key for comparison.

The selection of the parameter k' has a huge impact on the insert/lookup performance, which is directly related to the number of memory accesses per key, as well as the memory consumption. Setting k' to a high value like eight, halves the maximum number of memory accesses per key, but increases the memory consumption, if the key distribution is not dense. A lower value of k' causes the opposite effect. Thus, the prefix length should be set according to the key distribution of the indexed data. For the standard case, $k' = 4$ turned out to offer the best trade-off between performance and memory consumption. There are also further optimizations possible like a variable prefix length (per level) and node compression, which are employed in the KISS-Tree.

[1]Unbalanced trees do not require comprehensive tasks for tree balancing, which involves complex locking schemes for parallel operations.

2.2 KISS-Tree

The KISS-Tree is a prefix tree-based index structure for 32bit keys and inherits all of the prefix tree characteristics as presented before. The difference between both is that the KISS-Tree needs less memory accesses for a key lookup. For instance, a 32bit key lookup in a prefix tree issues up to 9 memory accesses, whereas the KISS-Tree requires only 3 memory accesses. The KISS-Tree structure is visualized in Figure 2(b). Here, the key is split in exactly two fragments of 26bit for the first level and 6bit for the second level. The first fragment is used to calculate the bucket position on the first level, which consists of 2^{26} buckets each containing a 32bit compact pointer to the corresponding second level node. To avoid the complete physical allocation of the root node, which consumes 256MB of main memory, it is only allocated virtually. Only if a pointer is written to a bucket, the memory becomes allocated on-demand by the operating system at a 4KB granularity.

While prefix trees support keys of an arbitrary length, the KISS-Tree is limited to 32bit keys, which is mostly sufficient for join attributes. Thus, our $QPPT$ decides at query compile time, which index structure should be used for storing the intermediate result. The original KISS-Tree uses a bit-mask compression for second level nodes to save memory and preserve data locality. When the indexed column contains a dense value range, $QPPT$ disables this compression to avoid the high RCU [11] copy overhead for updates. This optimization increases the update performance and parallelism.

2.3 Batch Processing

As soon as the size of the prefix tree respectively KISS-Tree exceeds the available CPU cache resources, the number of memory accesses per key becomes the most important factor for insert/lookup performance. Since each memory access depends on the previous memory access, a key lookup spends most of the time for waiting on main memory transfers, which wastes CPU resources. A way to hide that memory latency is to process batches of keys to allow software pipelining and memory prefetching. This batch processing approach collects multiple insert/lookup operations before executing them in a single batch operation. Algorithm 1 shows the pseudo code for such a batch lookup operation. The algorithm receives the *batch structure* as input, which

Algorithm 1 Batch Lookup

```
procedure BATCHLOOKUP(batch)
    done ← false
    while not done do
        done ← true
        for all job in batch do
            if job.done then
                continue
            job.node ← GETCHILD(job.node,job.key)
            PREFETCH(job.node)
            if ISCONTENT(job.node) then
                job.done ← true
                ...
            if not ISCONTENT(job.node) then
                done ← false
```

is an array of job structures each consisting of the lookup key, the leaf node, and a field, which indicates that the job is done. To process the batch, the algorithm starts on the root tree level and calculates the memory address of each job's child node, followed by a prefetch instruction of that child node address. This prefetch instruction is important, because it fetches the next node into the L1 cache of the CPU. Thus, the child nodes can be processed without memory latency, when the algorithm moves on to the next level. The level-by-level processing scheme continues until each job found its content/leaf node and performed the associated operation on it.

QPPT makes extensive use of batch processing. On the one hand, join operators buffer their lookups to prefix trees to reduce the function call overhead and to hide memory latency. On the other hand, inserts into intermediate indexes are buffered to achieve the same goals. Moreover, batching prefix tree operations has a positive impact on code and data locality, because operators work a longer amount of time on the same data structures.

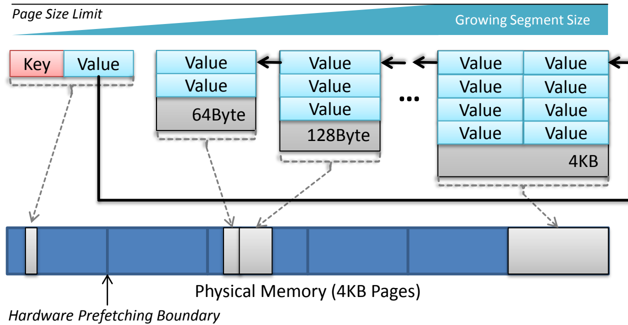


Figure 4: Duplicate Handling.

2.4 Duplicate Handling

Duplicate handling is an important point when processing queries on large indexes. Simply storing duplicates as linked lists usually results in random memory accesses, which add a lot of memory latency to the query execution. To improve the scan performance, duplicates must be stored sequentially in the main memory. This way, the duplicate scan benefits from hardware prefetching mechanisms of modern CPUs. An important characteristic of those hardware prefetchers is that they only work inside of a memory page (usually 4KB), which is related to the virtual to physical address mapping. Figure 4 depicts the duplicate handling that we employed for *QPPT*. The first value for a key is stored in a dedicated memory segment, which includes a pointer to the duplicate list. The duplicate list starts with a memory segment of 64Byte. If this piece of memory is full, the next segment gets allocated, which is twice the size of the previous segment and is put in front of the duplicate list. Memory segments grow only until they reached the page size of 4KB, because it makes no difference for the hardware prefetching, whether 4KB duplicate segments are stored sequentially or not. The memory management subsystem has to take care of the correct alignment to 4KB page boundaries. We decided to use this kind of duplicate handling because it poses a good trade-off between scan performance and memory consumption.

2.5 Performance Evaluation

In this section, we evaluate the insert/update and lookup performance for the prefix tree ($k' = 4$), the GLib hash table, the Boost hash table, and the uncompressed KISS-Tree. For the KISS-Tree, we also evaluated the performance with batch processing. All experiments were run on an Intel i7-3960X, running Ubuntu 12.04 as operating system. Figure 3(a) shows the time it takes to insert/update a key that is randomly picked from a sequential key range for different index sizes (x-axis). As can be seen, the KISS-Tree shows the best performance of all index structures, followed by the hash table implementations. The measurements also reveal that indexing benefits from batch processing, especially for large trees, where operations are memory-bound. The worst performance, we measured for the standard prefix tree, which is related to the high number of memory accesses per key. Configuring the prefix tree with a higher k' and adding batch processing will result in a performance better than the hash tables.

Figure 3(b) shows the corresponding measurements for the lookup performance per key. Here, we observe a behavior similar to the insert/update performance. The major difference is that the performance of the non-batched KISS-Tree gets closer to the performance of the hash table implementations in the memory-bound case. This is a result of the increasing memory latency impact on the KISS-Tree, which can be compensated with batch processing.

2.6 Summary

Prefix Trees [5] including the KISS-Tree [9] are an suitable data structures for our *indexed table-at-a-time* query processing model. As this section showed, the insert/update and lookup performance outperforms hash tables as partially used in various operators today. Furthermore, prefix trees are order-preserving in contrast to regular hash tables. This order-preserving property can be efficiently exploited in almost all database operators (e.g., join, selection, and sorting). Therefore, we decided to design an *indexed table-at-a-time* processing model on top of prefix tree-based index structures. The following section introduces our *Query Processing on Prefix Trees (QPPT)* as a realization of the *indexed table-at-a-time* processing model in more detail.

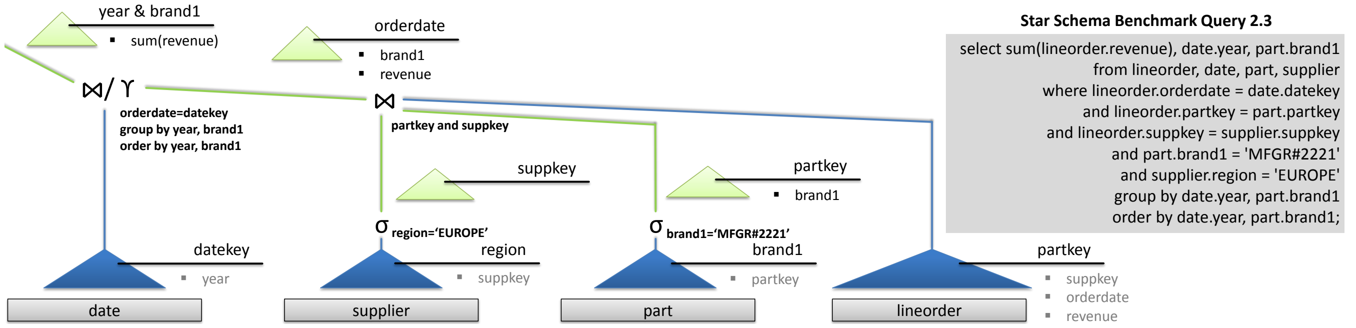


Figure 5: *QPPT* Execution Plan for Star Schema Benchmark Query 2.3.

3. QPPT OVERVIEW

In this section, we give an overview of *QPPT* execution plans. We do this with the help of query 2.3 of the Star Schema Benchmark (SSB) [13], which is depicted on the right of Figure 5 in SQL. This query comprises all important SQL features like selections, joins, grouping, and sorting. The SSB uses a standard star schema; it is derived from TPC-H snowflake schema and consists of the huge fact table `lineorder` surrounded by the dimension tables `part`, `supplier`, `customer` and `date`. Essentially, query 2.3 joins the fact table with the dimension tables `part`, `supplier`, and `date`. Afterwards, a selection is applied to `part.brand1` and `supplier.region`. Finally, the result is grouped and sorted by `date.year` and `part.brand1` with the sum of `lineorder.revenue` as aggregation.

A possible *QPPT* execution plan for SSB query 2.3 is visualized in Figure 5. The starting point of the execution plan are four prefix tree-based indexes—one for each relation (we will refer to them as base indexes). These indexes are either already present or are created once and remain in the data pool for future queries. There are two different options for the kind of payload that is stored inside the base indexes. The first option is to store only the record identifier (rid). This option realizes a pure secondary index. Another way is to additionally store partial records besides the rid in the payload, which realizes a partially clustered index. This approach dramatically increases query execution performance, because operators do not have to randomly access records during processing. Indeed, the costs for random accesses can be reduced with the help of prefetch instructions, but they will never achieve the performance of sequential access. Thus, it is reasonable to store relevant join, selection, and grouping predicates inside the payload of base indexes. However, partially clustered indexes consume more memory and complicate the index selection problem. Moreover, there are also combinations of partially clustered and secondary indexes possible.

The first operator that is executed in query 2.3 is the selection on the `brand1` attribute of the relation `part`. This operator takes the base index that is indexed on the `brand1` column and searches the index for the key `'MFGR#2221'`. The resulting tuples of this index lookup are now inserted in the output index that is indexed on the `partkey` attribute. As already mentioned, a partially clustered base index speeds up the indexation process, because the key for the new intermediate index is already stored inside the payload of the base index. The payloads of this intermediate index store all attributes that are necessary for successive operators of the query plan. As base indexes have to care for transactional isolation, intermediate indexes do not have to, because they are private for the query. In parallel to the selection on table `part`, the second selection on the relation `supplier` is performed. This selection operator works analog to the first selection, but uses a different input index and builds an index on attribute `suppkey` as output.

After both selections finished and created their respective intermediate index, the first join operator is executed. This 3-way/star join operator joins the `part` and the `supplier` dimension tables on the fact table `lineorder`. Therefore, the operator takes the `lineorder` index on attribute `partkey` and both resulting intermediate indexes of the previous selection operators as input. The reason for doing a 3-way/star join is to reduce the tuple cardinality of the output index. In this way, we save memory as well as index insertion costs. We will describe the internal functioning of the join operator in Section 4 in detail. The output of the 3-way/star join operator is an index on the attribute `orderdate` that contains the individual values for the columns `brand1` and `revenue` inside the payload.

Finally, a simple 2-way join operator outputs an already aggregated index. For that, it joins the `date` table on the `orderdate` column. Because the *indexed table-at-a-time* processing model creates indexes as output for each operator, the grouping happens automatically as a side effect. Thus, no separate grouping operator is necessary for the query execution plan. The output index of the join is indexed on the composed key of the attributes `year` and `brand1`. If the insertion of such a composed key detects that the key is already present in the index, it only applies the aggregation function—a sum in this specific case—on the existing value and the new one. The result of the query is an index that is indexed on `year` and `brand1` and stores the sum of `revenue` as payload. Because the resulting index is physically a prefix tree, it is already sorted. At last, the query execution engine has to iterate over the index while transferring the results to the client.

4. COMPOSED OPERATORS

In this section, we describe the internal functioning of *QPPT* plan operators. As already mentioned, *QPPT* uses composed operators as a core component for query processing. Contrary to traditional operators, a composed operator executes the work of multiple logical operators and thus omits high tuple materialization costs. We distinguish between three levels of operator integration:

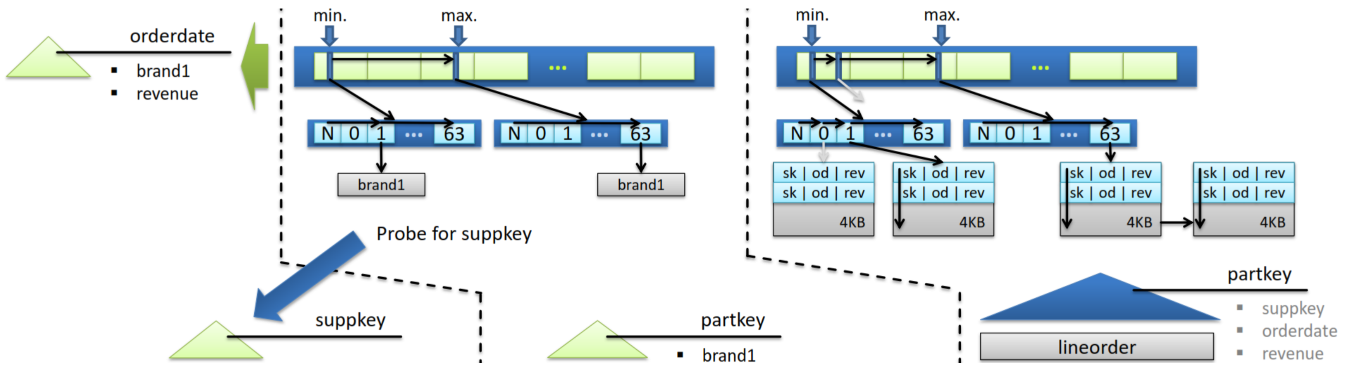


Figure 6: *QPPT*'s 3-Way/Star Join Operator for Star Schema Benchmark Query 2.3.

Level 1: The lowest level of integration is inherent to the *indexed table-at-a-time* processing model, which defines that each *QPPT* operator automatically does simple operations like the sorting/grouping or aggregation when building the output index.

Level 2: On the next level, we combine homogeneous operators. For instance, subsequent join operators are combined to a single multi-way/star join operator.

Level 3: The highest level of optimization composes heterogeneous operators like a multi-way/star join and a selection to a single operator. This way, we are able to execute large parts of a complex OLAP query with only a single composed operator that omits the materialization of large intermediate results.

In the following, we describe operators at the specific integration levels. We start with the selection/having operator, that has the lowest level of integration. Afterwards, we move on to the multi-way/star join operator, which is a representative of composed homogeneous operators. Finally, we discuss the select-join as a composed heterogeneous operator.

4.1 Selection/Having

The logical selection and having operators are physically the same operator. For a simple selection predicate, the operator expects an index on the selection attribute as input. The operator scans the input index for qualifying tuples in the same way as traditional databases do. The main difference is that the selection inserts all the qualifying tuples into a new index on the key requested by the successive operator. To process conjunctive combinations of predicates, the selection operator prefers to operate on a multidimensional index as input. In this case, the processing scheme does not differ from the selection of a simple predicate. Because operators sometimes request multidimensional keys as input, each *QPPT* operator is also capable of producing a multidimensional index as output.

Operators at the beginning of the query plan do not always have multidimensional base indexes available, because it is mostly not feasible to provision all required combinations of attributes as multidimensional indexes. In that case, each predicate of the conjunctive or disjunctive combination is processed by a separate selection operator. Each of those operators builds an index on the record identifier of the qualifying tuples. Afterwards, intersect and/or distinct union operators process those indexes and the last set operator builds a suitable index for the following operator. Set operators use the same efficient *synchronous index scan* as the join operator does; it is introduced in the next section.

4.2 Multi-Way/Star Join

The basic 2-way join operator takes only two indexes (*main indexes*) as input. The multi-way/star join as a composed operator, takes a set of additional indexes (*assisting indexes*) as input. We explain the standard join and the multi-way/star join with the help of the 3-way/star join operator of SSB query 2.3 depicted in Figure 5. Figure 6 gives a closer view of the internal processing of the 3-way/star join. In this specific case, the index on `lineorder.partkey` and the index on `part.partkey` (output of the selection on `part`) are the *main indexes*. The result of the selection on `supplier` is the *assisting index* for this 3-way/star join.

Since both *main indexes* are prefix tree-based (KISS-Trees) and are therefore unbalanced, they are joined using the *synchronous index scan*. This processing approach works by scanning the root nodes of both *main indexes* synchronously from the left to the right. Only if the synchronous scan encounters a bucket that is used in both indexes, the scan suspends on the root node and starts with synchronously scanning both child nodes in the same fashion. As both *main indexes* of our 3-way/star join example are KISS-Trees, the scan on the root level starts at $max(left.min, right.min)$ and ends at $min(left.max, right.max)$. This way, we avoid scanning the entire root nodes, which are each 256MB in size, for dense keys. The main advantage of the *synchronous index scan* is that it only descends to child or content nodes that are in use by both indexes. Thus, we are able to skip a lot of tree descents, which results in a better scan performance. The arrows in Figure 6 show the synchronous scan path for our example. Here, the scan was able to skip the scan of an entire child node and a content node. We use this efficient *synchronous index scan* for all operators that work on two input indexes (i.e., set operators). This processing scheme is also possible for the standard prefix trees.

During our example, the scan encountered two content nodes that are present in both *main indexes*. Each time, such content nodes are hit, the join builds the cross product of the tuples in the left and right hand content node in a nested-loop manner. In the example, the 3-way/star join has one *assisting index* on `suppkey`. Thus, it extracts the corresponding key either from the storage layer or from the index

payload and executes a read operation on the `assisting index`. If the key is not present, the combination is removed from the cross product. Otherwise, all tuples returning from this operation are added to the cross product. For our example, the index on `suppkey` is unique and contains no payload. Therefore, the join just has to probe for the key existence. Finally, each combination of the cross product is materialized and inserted into the output index using the proper key for the next operator. We mainly use those multi-way/star joins, if the expected cardinality of the output can be reduced to avoid a high memory consumption as well as tuple and index materialization costs.

Since multi-way/star joins usually join a high number of indexed tables, we face two problems that have a negative impact on the join performance: First, joining multiple tables is a recursive operation and thus requires a lot of function calls. Second, a high number of point accesses in large indexes adds a lot of memory latency to the execution time of an operator. To circumvent these issues, composed operators use a joinbuffer respectively selectionbuffer. Thus, the operator is able to buffer index lookups, which reduces the number of function calls and allows efficient batch lookups as described in Section 2.3.

4.3 Select-Join

The select-join operator offers a high degree of operator composition. Such operators are usually necessary, if a single select operator would materialize a huge amount of data. In those cases, index creation and tuple materialization costs dominate the overall execution time of the select operator. We are able to reduce these costs with the help of selectionbuffers and batched index insertions, but costs for intermediate result materialization are still to high. For this reason, it is feasible to integrate such a selection operator with the successive operator (e.g., a multi-way/star join). This combination does not allow the application of a *synchronous index scan* which profits from the data locality, because both indexes are sorted on the join predicate. However, prefix trees offer a high point read performance and thus the composition of selection and multi-way/star join operators is very useful. We will demonstrate the impact of the different operator composition levels in Section 5.

5. EVALUATION

To evaluate our *QPPT* plans, we measured the execution times for the entire set of SSB queries. We set the scale factor (SF) for all experiments to 15; we tried also other SFs with similar results. To compare our results to existing DBMSs, we ran the same experiments on MonetDB (column-at-a-time) and a commercial DBMS (vector-at-a-time). For fairness reasons, we ran those DBMSs in single-threaded mode, because our *QPPT* implementation supports currently no intra-operator parallelism. All necessary indexes were created for the specific SSB queries on all systems except for MonetDB, which manages indexes on its own. We implemented the *QPPT* operators as well as the corresponding index structures in our DBMS prototype DexterDB [1]. DexterDB is an in-memory database system that stores tuples in a row-store and uses MVCC [3] for transactional isolation. Our evaluation machine is an *Intel i7-3960X (3.3GHz, 15MB LLC)* CPU equipped with *32GB of main memory*. The running operating system is *Ubuntu Linux 12.04*.

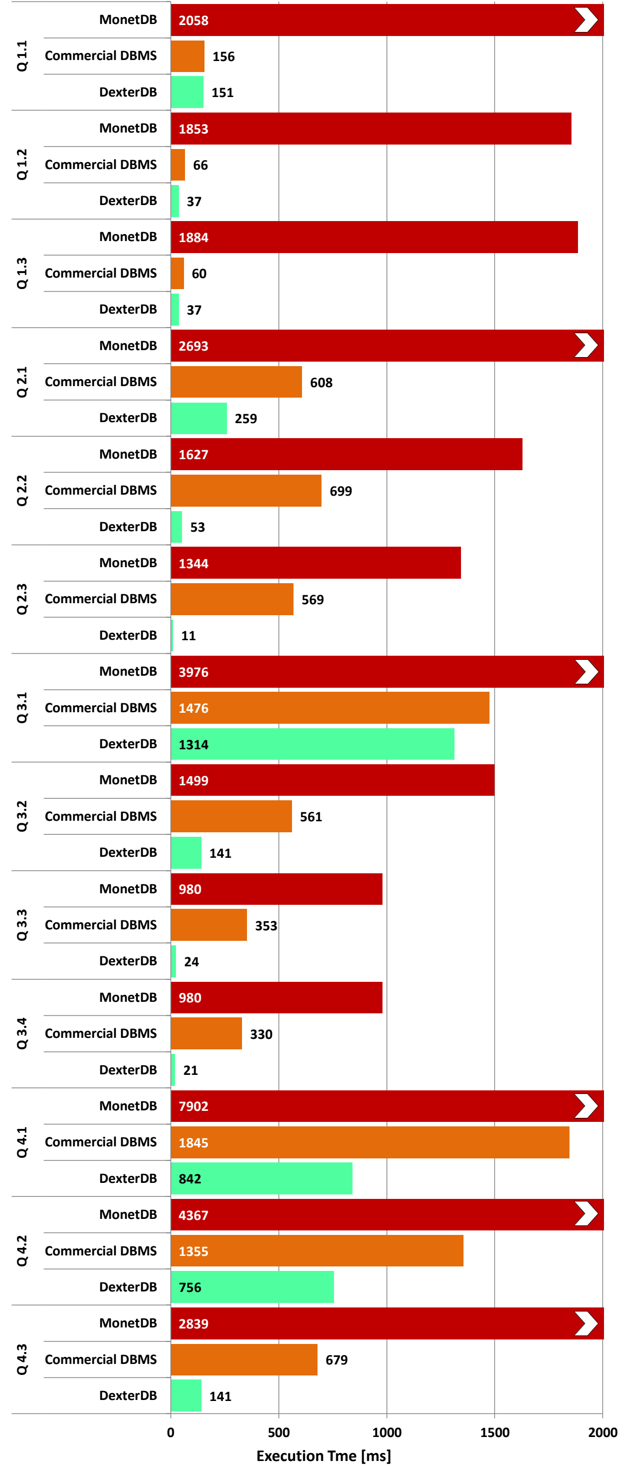


Figure 7: SSB (SF=15) Query Performance.

In Figure 7, we visualize the execution times of all SSB queries on the different DBMSs. As can be seen, DexterDB outperforms MonetDB as well as the commercial database system on each SSB query. The performance gain compared to the other DBMSs highly depends on characteristics of the respective query. For instance, queries 1.x include only a sin-

gle join operation. Here, the results are close to each other, because the query plans are very simple and do not offer a high optimization potential in terms of composing operators. However, those queries are suited for evaluating the index scan and simple join performance of the database systems. DexterDB benefits from its efficient duplicate handling and index lookup performance and is thus able to keep pace with an efficient column-store implementation. Contrary to the simple 1.x queries, the 4.x queries join all five tables of the SSB. When processing such complex queries, column-store-based systems reveal their major weakness: the overhead for tuple reconstruction. With more and more join columns, column-stores have to reconstruct more columns of a tuple, which results in a degrading performance. In a row-oriented DBMS, tuples are stored as physical units and do not need to be reconstrcuted for query processing. However, this concept of passing entire tuples between operators increases output materialization costs for each operator, because tuples have to be unpacked, interpreted, and packed again. *QPPT* omits this materialization overhead with the help of composed operators. Therefore, DexterDB is able to process complex queries more efficiently than a column-oriented database system.

Listing 1: SSB Query 1.1

```
select sum(lo_extendedprice*lo_discount
    ) as revenue from lineorder, ‘date‘
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between 1 and 3
and lo_quantity < 25;
```

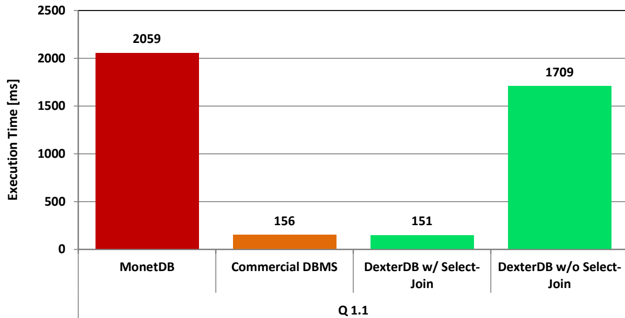


Figure 8: SSB Query 1.1 with and without Select-Join.

Figure 8 shows the execution times of query 1.1 (cf. Listing 1) for two different execution plans of DexterDB as well as both other DBMSs. The first configuration uses a composed select-join-group operator. The second one uses a simple select and a separate join-group operator. This experiment shows the advantage of a composed select-join operator, which skips the costly tuple materialization step for passing the selection result to the join-group operator. About 95% of the execution time of the query plan without a composed operator is spend for the selection operator, which needs the most time for tuple materialization and indexing of its output. However, the successive join-group operator is able to process data more efficiently, because it is provided with two indexes on its input, which avoids random prefix tree lookups. But the advantage is low in this scenario, because the result of the `date` selection fits easily into the CPU cache. Our investigations revealed that it is mostly beneficial to use select-join operators, if the single selection would materialize a large intermediate index as output.

Listing 2: SSB Query 4.1

```
select d_year, c_nation,  sum(
    lo_revenue - lo_supplycost) as
    profit from ‘date‘, customer,
    supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = AMERICA
and s_region = AMERICA
and (p_mfgr=MFGR#1 or p_mfgr=MFGR#2)
group by d_year, c_nation
order by d_year, c_nation
```

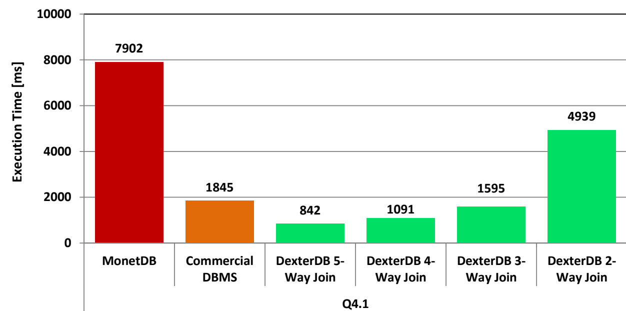


Figure 9: SSB Query 4.1 for Different Multi-Way/Star Join Configurations.

In our final experiment, we quantified the advantage of integrating join operators with each other. Figure 9 shows the respective results of SSB query 4.1 (cf. Listing 2). We measured the worst execution time for the execution plan that consists only of traditional 2-way joins. Building a composed 3-way join operator out of the first two 2-way joins results in much better execution time. Again, this performance increase is a direct result of the skipped tuple materialization and the high lookup performance of prefix trees. The step from the 2-way to the 3-way join is the most beneficial one, because the first join materializes the largest intermediate result. If we integrate more joins to one composed operator, execution time decreases even more, but the difference is not as high as before.

6. RELATED WORK

The related work of this paper can be roughly classified into (1) template-based database system and (2) compile-based database systems.

The *template-based database systems*—where DexterDB belongs to—cover all systems that rely on operator templates used for building query plans. This includes most of the current commercial systems (e.g., VectorVise [17]) and many research prototypes (e.g., C-Store [15], MonetDB [6]). A large number of the template-based systems use the volcano iterator model [8]—possibly including optimizations like buffers between the operators [16] or block-wise processing [2]. As mentioned previously, this tuple-at-a-time

processing model suffers from high costs for copying tuples, interpreting tuples, and virtual function calls. MonetDB [6] avoids most of these costs by employing the column-at-a-time processing model. Each operator in the query plan processes a full column before the next operator is invoked. A query plan itself is composed from a rich set of BAT algebra operators. VectorVise [17] (formerly MonetDB/X100) employs the vector-at-a-time processing model where small batches that fit into the processors caches rather than full columns are processed. Processing takes place using vectorized primitives, which can also be combined to obtain a higher performance. Although the newer processing models allow to exploit the capabilities of modern hardware, but they suffer from disadvantages of the column-wise processing like the expensive tuple reconstruction. Our processing model does not require such a tuple reconstruction but keeps the advantages of the newer models.

Our composed operator approach is related to *compile-based database systems* [12, 10], where query execution plans are translated into compact and efficient machine code. In this approaches, several query compilation strategies (e.g., composing several operators) are proposed to change the operator centric processing to a data centric processing. Generally, the operator boundaries are blurred to reduce expensive intermediate data materialization costs and data exchange between operators to a minimum. The achieved performance gain of the data centric processing compared to the operator centric processing is remarkable. However, the query compilation time massively increases for each single query because an in-depth query execution plan analysis, rewriting, code compilation, and linking have to be conducted. Especially code compilation and linkage costs increase with a growing complexity of the DBMS core functionality. Moreover, using specialized code for each different query, pollutes the CPU's instruction cache when running queries in parallel. Our processing model with the available composed operators avoids those drawbacks.

7. FUTURE CHALLENGES

The transition to the *indexed table-at-a-time* processing model opens up a huge research field. As main challenge, we identified the physical plan optimization, which has lots in common with traditional plan optimization, but offers additional optimization opportunities. For instance, the reduction of the output materialization costs of an operator remains as one of the core optimization targets. To reach this optimization goal, *QPPT* offers additional tools (e.g., batch inserts and composed operators).

When looking at current hardware trends, we identify an increasing degree of parallelism in terms of Hyper-Threading, cores per CPU, and Sockets per Machine, as well as heterogeneous Coprocessors (e.g., GPUs, FPGAs). To scale with the high number of processing units, our *QPPT* operators need to support intra-operator parallelism. We are currently investigating static and dynamic concepts for building such operators and got first promising results. The main advantages of the *QPPT* processing model—in concerns of parallel processing—are the characteristics of the underlying prefix tree index structure. First, the prefix tree is a latch-free index structure that is synchronized via atomic compare-and-swap instructions and second, the deterministic nature of unbalanced trees offers an easy way of splitting the tree into subtrees, which can be assigned to different threads. Contrary, if a balanced tree is modified while it is processed by an operator, the balancing operation may moves already processed data to another threads subtree.

Another challenge is the classic index selection problem. Because *QPPT* relies heavily on base indexes, we have to select those indexes. Additionally, we have to decide which attributes are stored clustered inside an index. Currently, we are working on a fully automatic approach that creates and drops those base indexes on its own. This approach is also able to dynamically grow and shrink indexes depending on the workload. This finer decision granularity offers much more potential compared to binary decisions that either keep an index or not. For instance, if a workload only reads a certain key range of an index, the subtrees outside of this key range are evicted and are restored on-demand.

Finally, we are also looking into the direction of hybrid storages (row- and column-stores) and hybrid processing models, to check whether performance benefits from those hybrid systems.

8. CONCLUSION

OLAP systems have to handle and process large amounts of data that are growing every day. To keep pace with this growing data amounts, in-memory DBMSs start to dominate the analytical market. Compared to traditional disk-based databases, in-memory systems store all data in the main memory to reach higher throughput and lower latency. Those in-memory systems usually use column-stores for OLAP queries, to exploit the sequential main memory access performance. Thus, they also deploy a column-at-a-time processing model, which passes entire columns between operators, instead of a tuple-at-a-time processing model, which was developed for disk-based row-stores. Databases that are considered to be state-of-the-art use a vector-at-a-time approach, which splits the passed column into vectors that fit in the CPU's cache.

In our paper, we introduced the very different row-oriented *indexed table-at-a-time* processing model, which passes entire tables that are stored as clustered indexes between operators. Thus, each operator always gets suitable indexes as input and generates one output table that is indexed on the column(s) requested by the following operator. To minimize the intermediate index materialization costs, we proposed prefix tree-based index structures, which are known to be main memory optimized and offer a balanced read/write performance. Moreover, we presented *QPPT* operators that work very efficiently on prefix trees using the *synchronous index scan*. As another core component, the *indexed table-at-a-time* processing model allows composed operators, which avoid the materialization of large intermediate results and thus dramatically speed up query execution, as our evaluation revealed.

All in all, the *indexed table-at-a-time*-based *QPPT* processing model gave very good results during our SSB experiments. Those results emphasize the high potential of the processing model compared to the existing ones. Since this are only the first results, we have to investigate the performance on the more complex TPC-H benchmark and real world scenarios. Moreover, we work on the complementary technology for intra-operator parallelism, plan optimization, and automatic index management.

9. ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

10. REFERENCES

[1] DexterDB. `http://wwwdb.inf.tu-dresden.de/dexter`.

[2] Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, pages 567–, Washington, DC, USA, 2001. IEEE Computer Society.

[3] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. *ACM Trans. Database Syst.*, 5(2):139–156, June 1980.

[4] R. Bayer and E. McCreight. *Organization and Maintenance of Large Ordered Indexes*, pages 245–262. Software pioneers, New York, NY, USA, 2002.

[5] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, pages 227–246, 2011.

[6] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, Dec. 2008.

[7] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[8] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6:120–135, 1994.

[9] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. *KISS-Tree*: Smart Latch-Free In-Memory Indexing on Modern Architectures. In *DaMoN*, pages 16–23, 2012.

[10] K. Krikellas, S. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *ICDE*, pages 613–624, 2010.

[11] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems.

[12] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[13] P. O'Neil, B. O'Neil, and X. Chen. Star Schema Benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[14] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. *SIGMOD Rec.*, 29:475–486, May 2000.

[15] 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*, pages 553–564. VLDB Endowment, 2005.

[16] J. Zhou and K. A. Ross. Buffering Database Operations for enhanced Instruction Cache Performance. In *SIGMOD*, pages 191–202, 2004.

[17] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

APPENDIX

In the appendix, we give a short proposal for our *QPPT* demonstration that we would like to give at the CIDR conference.

A. DEMONSTRATION DETAILS

We implemented *QPPT* plan operators and prefix tree-based indexes in our row-store-based in-memory database system DexterDB, which uses MVCC for transactional isolation. For demonstration purposes, we build a front-end application to interactively help the audience to understand *QPPT* execution plans. This demonstrator includes a set of predefined OLAP queries from the Star Schema Benchmark. After selecting a specific query, the demonstrator navigates to the query details view where we are able to manipulate query optimization parameters of the connected DexterDB instance. Afterwards, the generated execution plan is visualized in the demonstrator and as soon as query execution finished, execution statistics are transferred from the database to the demonstrator and get visible in the query plan. In the following, we describe the interactive options and visualizations in more detail.

After launching the demonstrator, the query selection view is displayed to the audience. Here, we have the option to select a predefined OLAP query of interest. A click on the query of choice opens the query details view. We include the complete set of queries of the benchmark in our demo, because each query has different characteristics, which influence query execution performance. Queries of the selected benchmark comprise a different number of join operations and different restrictions on the dimension tables and the fact table, which results in a varying size of intermediate results and operator complexities. More simple queries like query 1.x are better suited to demonstrate index scan performance, whereas queries 4.x are better suited for showing the benefit of multi-way/star joins.

The query details view for SSB query 2.3 is depicted in Figure 10. On the left-hand side of this view are the individual optimization options located, which can be changed. One set of options are the physical plan optimization options. With the first option, we can decide whether DexterDB should integrate selection operators with join operators, if possible. Setting this option to off requires that the select operator has to materialize an intermediate index, which mostly has a negative impact on the execution time, but for some queries this option is beneficial. If this option is set to on, the optimizer generates a single select-join operator, which can skip the materialization step of the selection and each tuple of the selection is directly processed by the included join operator. The second physical optimization setting is the joinbuffer or selectionbuffer size. This buffer can be set to size 1(none), 64, 512, or 2048. If the joinbuffer is activated, the database system is able to profit from batch lookups in prefix trees. The other advantage is that the usage of a joinbuffer also enables operators to do batch insert, when materializing the intermediate index. Regarding the size of the joinbuffer, different settings result in a changing execution performance, because a too low or a too high size affects the performance negatively. The last offered setting influences the logical plan optimization. Here, we included an option to limit the generation of multi-way/star joins, which can be set to 2-way, 3-way, 4-way, or multi-way join.

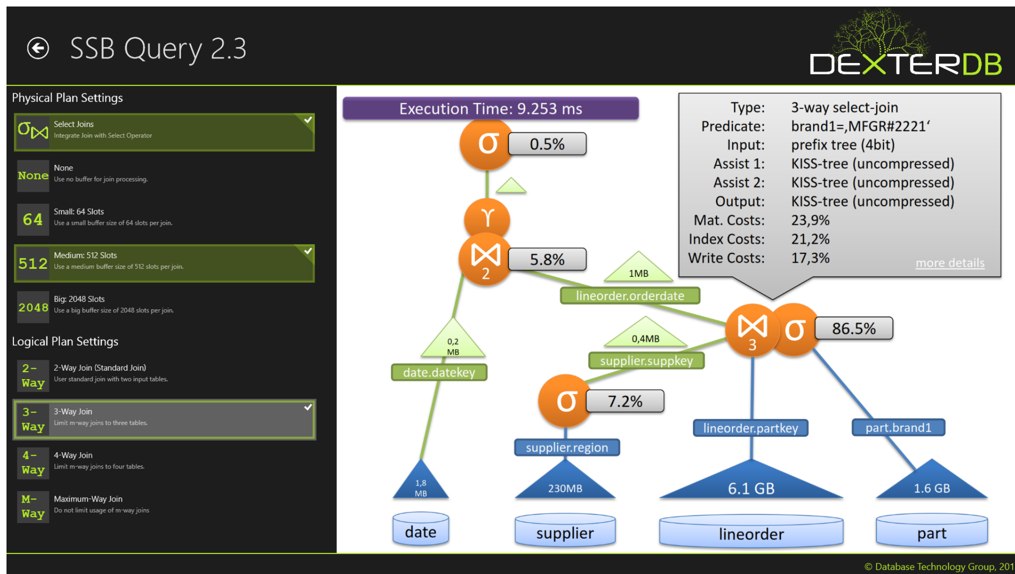


Figure 10: Query Details View.

The 2-way join only generates traditional join operators that join two tables with each other whereas the multi-way join allows the join of multiple tables with a single operator that is able to skip costly materialization steps. When joining dimension tables on fact tables, the size of intermediate results usually gets smaller. Thus, we use this option to show the point where a materialization is again beneficial.

On the right of the setting panel is the query plan visualization located. As soon as DexterDB finished the query execution plan generation, the plan becomes visualized in the demonstrator. The orange circles in the picture are plan operators, which are—based on the chosen optimization settings—either classic operators or composed operators. For instance, the rightmost operator in the figure is a 3-way-select-join that does the selection on `part.brand1` and directly joins results with the `lineorder` and `supplier` table. The result of this composed operator is materialized as an intermediate prefix tree; it is passed to the successive operator, which is a 2-way-join-group. During query execution, DexterDB collects execution statistics, which are visualized in that execution plan right after the execution finished. Those statistics comprise:

- The total execution time of the query and the portion of time that is spend for the individual operators.
- The size of base indexes and intermediate indexes, which have to be generated during query execution by the single operators.
- The type of input indexes and the output index.
- Internal operator statistics that reveal, what amount of time was taken for tuple materialization and the output indexing.

With the help of those execution statistics, the audience gets a deeper insight into *QPPT* execution plans. Moreover, we can demonstrate which parts of the query plan consume the most time.