Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner

Query Processing on Prefix Trees Live

Erstveröffentlichung in / First published in:

SIGMOD/PODS'13: International Conference on Management of Data, New York 22.-27.06.2013. ACM Digital Library, S. 1105–1108. ISBN 978-1-4503-2037-5

DOI: https://doi.org/10.1145/2463676.2463682

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-804440

SLUB
Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", S. 1105–1108, ISBN 978-1-4503-2037-5
https://doi.org/10.1145/2463676.2463682

Query Processing on Prefix Trees Live

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 demonstration proposal, we present DexterDB, which implements our novel prefix tree-based processing model that makes indexes the first-class citizen of the database system. The core idea is that each operator takes a set of indexes as input and builds a new index as output that is indexed on the attribute requested by the successive operator. With that, we are able to build composed operators, like the multi-way-select-join-group. Such operators speed up the processing of complex OLAP queries so that DexterDB outperforms state-of-the-art in-memory databases. Our demonstration focuses on the different optimization options for such query plans. Hence, we built an interactive GUI that connects to a DexterDB instance and allows the manipulation of query optimization parameters. The generated query plans and important execution statistics are visualized to help the visitor to understand our processing model.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, relational databases*

Keywords

indexing; in-memory query processing; prefix trees;

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 capacity, in-memory database systems become more and more popular in OLAP domain since often the entire data pool fits into main memory. Hence, the superior characteristics of main memory as low latency or high bandwidth are able to be exploited for query processing leading at the same time to a paradigm shift in query processing models.

The traditional tuple-at-a-time processing model [6] 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 [4] or the vector-at-a-time processing model [5], 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.

To completely avoid this reconstruction overhead, we propose to return to row-oriented systems enhanced with a novel up-to-date processing model. In this demonstration, which we will present at SIGMOD, we show our new *indexed table-at-a-time* processing model for in-memory row-stores, which is depicted in Figure 1. The main characteristics of our *indexed table-at-a-time* processing model are:

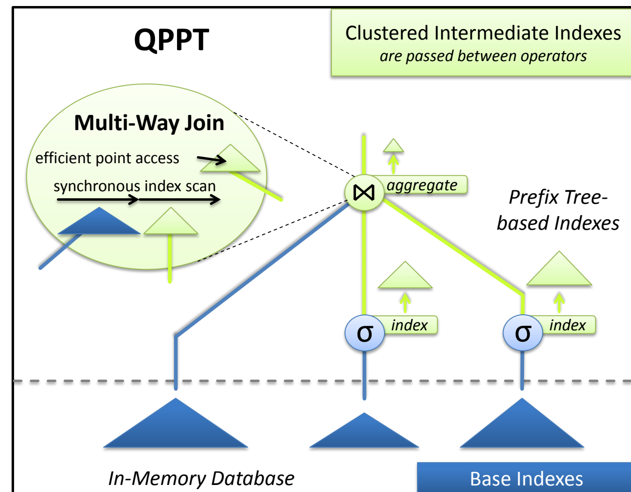

Figure 1: Overview of QPPT's *indexed table-at-a-time* Processing Model.

©2013 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *SIGMOD'13*, June 22–27, 2013, New York, New York, USA.

DOI: https://doi.org/10.1145/2463676.2463682

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", S. 1105–1108, ISBN 978-1-4503-2037-5
https://doi.org/10.1145/2463676.2463682

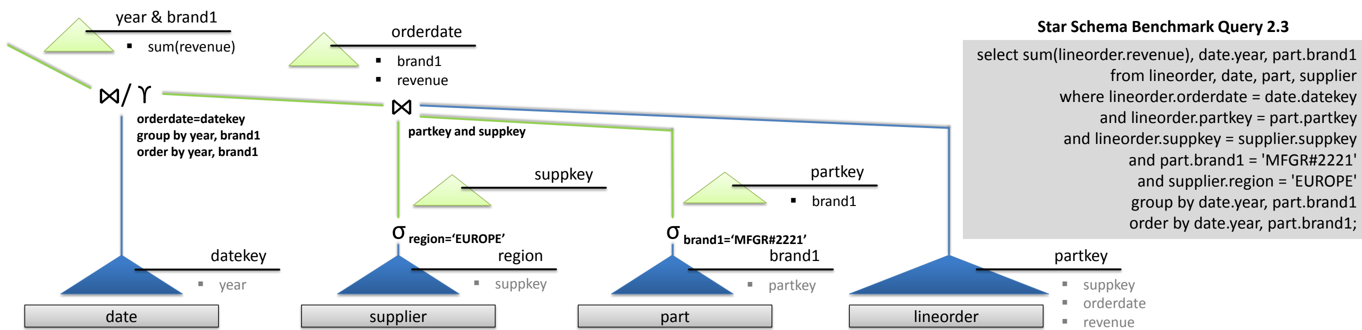

Figure 2: *QPPT* Execution Plan for Star Schema Benchmark Query 2.3.

1. **Intermediate Indexed Tables:** Instead of passing plain tuples, columns, or vectors between individual operators, our 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 model by reducing the number of next calls to exactly one and (ii) allow to process data using highly efficient operators that can exploit the indexed input data.

2. **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, which 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 and thus skips the often unnecessary plain tuple exchange step, since most traditional operators build internal indexes anyway. 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 the successive join operator.

3. **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 data. Also some base operators like sorting and grouping are not necessary anymore, because our intermediate results are always indexed appropriately. Furthermore, our *indexed table-at-a-time* processing model offers efficient implementation and execution of composed (n-ary) operators and each operator automatically does a sort/group or an aggregation on its output data. 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 [9]. Composed operators are core components of our novel processing model.

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 [3] 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 [7] 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. Therefore, we introduced *Query Processing on Prefix Trees (QPPT)* in [8], which is a realization of the *indexed table-at-a-time* processing model. We implemented *QPPT* in our in-memory row-store DexterDB [1] and were able to outperform state-of-the-art commercial column-stores during our experiments with the Star Schema Benchmark [10]. For demonstration purposes, we built an interactive GUI that connects to a DexterDB instance and allows the manipulation of query optimization parameters. The generated query plans and important execution statistics are visualized to help visitors to understand our processing model in full detail.

2. 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, which is depicted on the right of Figure 2 in SQL. A possible *QPPT* execution plan for this query is visualized next to the SQL query. 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).

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. 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 executed. 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 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`

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", S. 1105–1108, ISBN 978-1-4503-2037-5
https://doi.org/10.1145/2463676.2463682

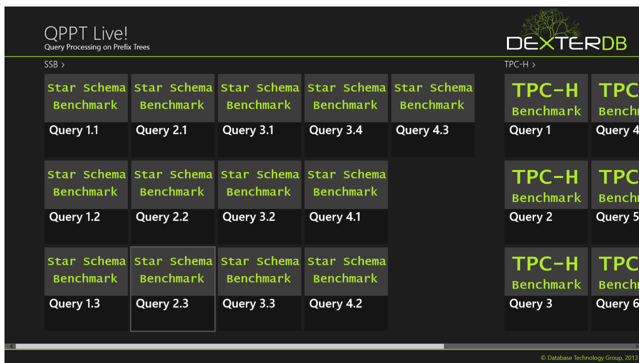

Figure 3: Query Selection View.

and both resulting intermediate indexes of the previous selection operators as input. The reason for doing a 3-way join is to skip the materialization of large intermediate results, which is necessary when using two 2-way joins instead of the 3-way join. In this way, we save memory as well as index insertion costs. Internally, the 3-way join synchronously scans the first two indexes (main indexes) and looks up the appropriate key in the third input index (assisting index). To increase the performance of those lookups, we use batch processing schemes to hide memory latency. The output of the 3-way 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 traditional 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.

3. 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 or transactional isolation. Additionally, we build a demonstration application to interactively help visitors to understand *QPPT* execution plans. This demonstrator includes a set of predefined OLAP queries from the well-known TPC-H [2] and Star Schema Benchmark [10]. After the visitor selects 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 transfered 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 visitor. Figure 3 shows the content of this view. Here, the visitor has the option to select a predefined OLAP query of interest. Those queries will include the entire set of the Star Schema Benchmark queries and the TPC-H benchmark queries. A click on the query of choice opens the query details view. We include the complete set of queries of the respective benchmarks, because each query has different characteristics, which influence query execution performance. For instance, queries of the Star Schema 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 joins.

The query details view for SSB query 2.3 is depicted in Figure 4. On the left-hand side of this view are the individual optimization options located, which can be changed by the visitor. One set of options are the physical plan optimization options. With the first option, the visitor 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 respectively 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 respectively 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 joins, which can be set to 2-way, 3-way, 4-way, or multi-way join. The 2-way join only generates traditional join operators that join two tables with each other, but 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 makes sense again.

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

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", S. 1105–1108, ISBN 978-1-4503-2037-5
https://doi.org/10.1145/2463676.2463682

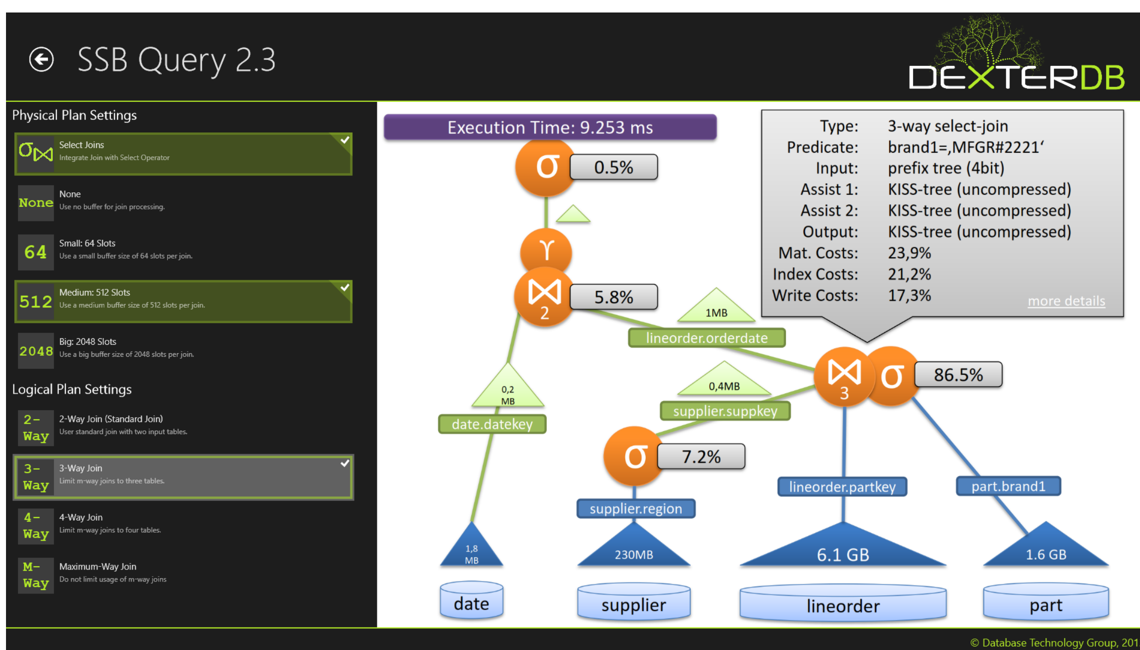

Figure 4: Query Details View.

as an intermediate prefix tree, which 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 runtime statistics comprise: (1) The total execution time of the query and the portion of time that is spend for the individual operators. (2) The size of base indexes and intermediate indexes, which have to be generated during query execution by the single operators. (3) The type of input indexes and the output index. (4) 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 visitor gets a deeper insight into *QPPT* execution plans. Moreover, we can demonstrate which parts of the query plan consume the most time.

4. CONCLUSIONS

In this demo proposal, we gave an overview of our novel query processing model for row-store-based in-memory databases. This processing model relies on indexed tables as intermediate results, which allows the construction of efficient composed operators that avoid the high costs for materializing intermediate results. In DexterDB, we implemented *QPPT* as a realization of this processing model, which relies on prefix tree-based indexes, because of their superior insert/lookup performance compared to other index structures like hash tables. For demonstration purposes, we built an interactive GUI that contains a set of predefined OLAP queries from the TPC-H and Star Schema Benchmark. With the help of this demonstrator, we are able to change different plan optimization settings of DexterDB and to visualize generated execution plans. Moreover, we enrich those query plans with execution statistics to give the visitor a deeper insight into *QPPT* query plans.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] DexterDB. http://wwwdb.inf.tu-dresden.de/dexter.
[2] TPC-H. http://www.tpc.org/tpch/.
[3] 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.
[4] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, Dec. 2008.
[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.
[6] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6:120–135, 1994.
[7] 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.
[8] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. QPPT: Query Processing on Prefix Trees. *CIDR*, 2013.
[9] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.
[10] P. O'Neil, B. O'Neil, and X. Chen. Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden