Query Processing on Prefix Trees Revisited

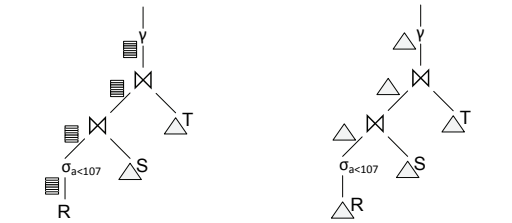
Thomas Kissinger Matthias Boehm Patrick Lehmann Wolfgang Lehner
TU Dresden, Database Technology Group; Dresden, Germany
thomas.kissinger@tu-dresden.de

Abstract

There is a trend towards operational or Live BI (Business Intelligence) that requires immediate synchronization between the operational source systems and the data warehouse infrastructure in order to achieve high up-to-dateness for analytical query results. The high performance requirements imposed by many ad-hoc queries are typically addressed with read-optimized column stores and in-memory data management. However, Live BI additionally requires transactional and update-friendly in-memory indexing due to high update rates of propagated data. For example, in-memory indexing with prefix trees exhibits a well-balanced read/write performance because no index reorganization is required. The vision of this project is to use the underlying in-memory index structures, in the form of prefix trees, for query processing as well. Prefix trees are used as intermediate results of a plan and thus, all database operations can benefit from the structure of the in-memory index by pruning working indices during query execution. While, this is advantageous in terms of the asymptotic time complexity of certain operations, major challenges arise at the same time. In this paper, we sketch our preliminary results. Efficient query processing over huge evolving data sets will enable a broader use of the consolidated enterprise data. Finally, this is a fundamental prerequisite for extending the scope of BI from strategic and dispositive levels to the operational level.

1 Introduction

Advances in information technology combined with rising business requirements lead to an exponentially growing amount of digital information created and replicated worldwide [9]. In addition to this huge amount of data, there is a trend towards Live BI [7, 12, 14] that requires immediate synchronization between the operational source systems and the data warehouse infrastructure in order to achieve high up-to-dateness for analytical query results. Pure data stream management systems can cope with high-rate input streams



(a) Traditional Query Plan (b) Prefix Tree Query Plan

Figure 1. Solution Overview

but have deficits regarding ad-hoc data analysis on historical data. Read-optimized data organization such as column-oriented data management provide efficient ad-hoc data analysis but exhibit drawbacks with regard to write-intensive applications. Recent research therefore focused on hybrid OLTP/OLAP solutions [8, 10]. However, Live BI additionally requires transactional and update-friendly in-memory indexing due to the existence of high update rates. While current research mainly focuses on improving search performance [11], in-memory indexing with prefix trees, in combination with optimistic concurrency control, exhibits a well-balanced read/write performance [13] because no index reorganization is required. In addition, advanced data analytics [5, 6] such as forecasting [1], which go beyond traditional data aggregation and analytical query processing, also require in-memory indexing to enable efficient point and range queries.

The vision of this research project is to use the underlying in-memory prefix tree index structures, which are required for a balanced read/write performance, for efficient query processing as well. As shown in Figure 1, prefix trees are used as intermediate results of a query execution plan and thus, all database operations can benefit from the structure of the in-memory index by pruning working indices during query execution.

Example 1 *Consider the query* $\gamma(\sigma_{a<107}(R) \bowtie S \bowtie T)$. *A traditional query execution plan as shown in Figure 1(a) might also use the underlying index structures, e.g., for index nested loop joins of* $* \bowtie S$ *and* $* \bowtie T$. *In contrast, our idea is to use the underlying index structure as intermediate results of* all *operators rather than just at the leaves of a query plan.*

Operator		Description
Type	Name	
Unary	ixRef	Simple reference to an existing index or a relation.
	ixCopy	Copies the entire index memory, in order to apply changes to it, without actually manipulating the underlying index.
	ixBuild	Builds a temporary, secondary index on a specific attribute of a relation.
	ixSort	Sorts the input prefix tree in ascending or descending order. The operator creates a virtual key, which is a concatenation of the attribute values.
	ixProject	This operator discards a set of attributes. This reduces the tuple size and lowers the memory transfers.
	ixGroupBy*	Groups the source relation by recursively building partitioned indices for each grouping attribute, similar to ixSort. Than, the operator traverses this index and applies an aggregate function like Sum, Average, etc. on those groups. The result will be a new intermediate prefix tree.
	ixGet	Does a simple point query on a prefix tree.
	ixSelectRange	Selects a range of values by pruning the index left to the lower boundary and right to the upper boundary.
	ixSelectMultiple	Selects disjunctive predicates on multiple attributes.
	ixSelectOr	Selects disjunctive predicates on the same attribute. This creates a predicate prefix tree that is joined with the input prefix tree.
	ixLimit	Limits the result set by scanning the defined number of tuples.
	ixEmit*	Transforms the input index to another format, e.g., CSV or text on console.
	ixLoad	Loads a large set of data into the main memory database.
Binary	ixJoinEqui	Realizes an Equi-Join.
	ixUnion	Merges two indexes with each other.
	ixUnionDistinct	Merges two indexes with each other and eliminates duplicates.
	ixDiff	Removes all references from the first index that exist in the second index.
	ixIntersect	Keeps all references in the first index that exist in both indexes.

Table 1. Table of Framework Operators

While, this is advantageous in terms of the asymptotic time complexity of certain operations and thus, allows for efficient query processing on very-large amounts of data, major challenges in terms of memory management, query processing and scalability arises at the same time. The objective of this project is to leverage the HPI Future SOC Lab infrastructure in order to evaluate our prototype in large-scale infrastructures and to adapt our framework accordingly.

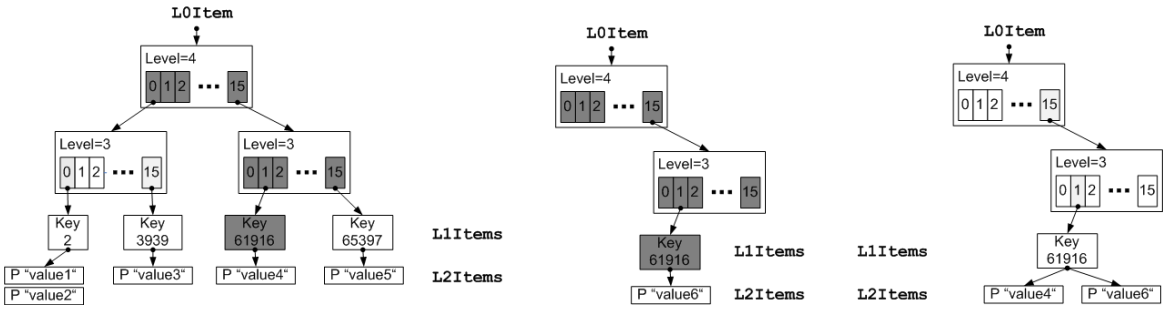
This paper—that extends our previous project report [2] by a detailed explanation of prefix-tree-based operators—is structured as follows. In Section 2, we give an overview of the operators that are included in our framework and we describe selected operators in detail. In Section 3, we summarize the status of our project and give an outlook on our planned future work. Finally, Section 4 concludes the paper.

2 Query Processing Overview

We use the *generalized trie* [3] as our underlying in-memory index structure. It is a prefix tree (trie) with variable prefix length of k' bits, where k' must be a divisor of the maximum key length k. Given an arbitrary data type (mapped to an order-preserving byte sequence) of k bits and a prefix length k', the trie exhibits a fixed maximum height $h = k/k'$ and each node of the trie includes an array of $s = 2^{k'}$ references (node size). The trie path for a given key k_i at level l is then determined with the l^{th} k'-bit prefix of k_i. In addition, we use *trie expansion* such that subtries are only expanded if required (if multiple keys share the same prefix) and *bypass arrays*, where trie levels with prefix zero can be bypassed. This trie exhibits (1) the deterministic property such that any key has exactly one path within the trie and (2) a constant worst-case time complexity of $O(h)$. Due to its deterministic property, this secondary index structure provides very high and predictable performance for read and write operations such that it can be used as a building block for a Live BI infrastructure. In addition, there is huge potential for further optimization (e.g., many-core, HW-architecture-awareness). Combining this in-memory index structure and our vision of query processing on prefix trees, in this section, we present an query processing overview and details on selected operators.

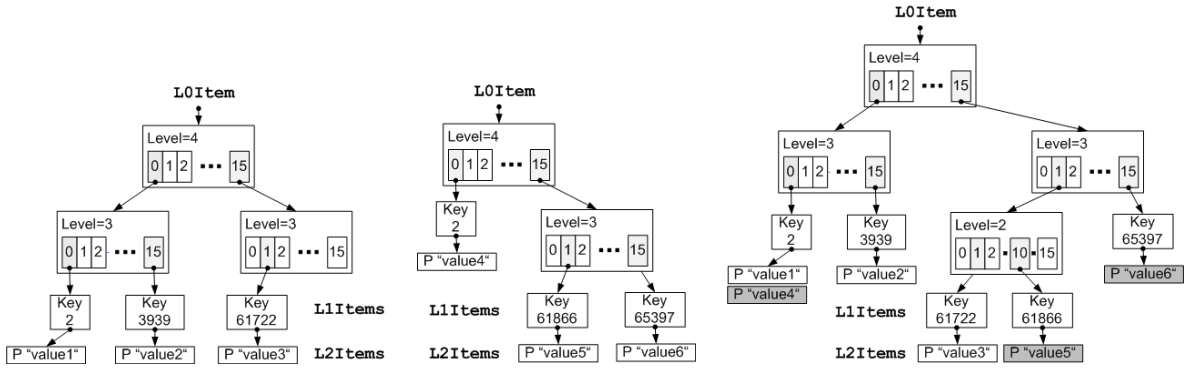
2.1 Selected Operators

In this section, we describe selected operators of our query processing framework. Table 1 shows an



(a) Left-Hand Tree (b) Right-Hand Tree (c) Resulting Tree

Figure 2. Example of an `ixJoinEqui`



(a) Left-Hand Tree (b) Right-Hand Tree (c) Resulting Tree

Figure 3. Example of an `ixUnion`

overview of all operators that are included in this framework including a brief description of operator semantics and realization aspects. These operators basically cover the relational algebra. Additionally, we introduce operators to handle the challenges that arise from our new approach based on in-memory indices. Namely, this is the need for copying the base index, when passing a base index to a destructive operator, creating a new index on another attribute of the relation (preparation step for following operators), and operators that optimize the performance for specific cases. In the following, we describe the `ixJoinEqui`, `ixUnion`, `ixSort` and the `ixSelectOr` in more detail.

ixJoinEqui: This operator takes two prefix trees as input and computes an Equi-Join. Both input trees have to index the comparison attribute. As we pass prefix trees as intermediate results through the query execution plan, we manipulate either the left- or the right-hand tree instead of creating a new temporary tree. The operator synchronously traverses both prefix trees and either prunes those paths, which are not present in both trees, recursively traverses these paths, or combines the tuple references, whenever it encounters a key that is present in both trees. With this approach, we benefit from the deterministic key paths of the underlying prefix trees.

Example 2 *Figure 2 shows an example Equi-Join using prefix trees. The `ixJoinEqui` starts to synchronously traverse the left- (see Figure 2(a)) and the right- (see Figure 2(b)) hand tree. We highlighted all buckets and keys that are accessed by the scan in dark gray. Since the zero-bucket is expanded in the right-hand tree, but not in the left-hand tree, the operator simply cuts off this expansion from the left-hand tree. The 15^{th} bucket on the other hand is expanded in both trees. Therefore, the operator needs to descent both trees synchronously and continue the scan on this level. In the first bucket of the left-hand tree as well as in the right-hand tree, the scan finds an equal value. Hence, it has to add the record pointer from the right-hand tree to the left-hand tree, to create a virtual tuple. The other way would be to materialize the resulting tuple, but this would require more copying and memory management overhead. The scan continues on the third level and also prunes the path in the 15^{th} bucket of the left-hand tree, because of the missing equivalent tuple in the right-hand tree. There is a high potential for massive parallelization of this operator. We can achieve this by dividing both prefix trees in an arbitrary number of disjoint subtrees (given by the deterministic structure of the prefix trees), which can be independently processed by different threads.*

ixUnion: Similar to the `ixJoinEqui` operator, the

`ixUnion` operator takes a left- and a right-hand prefix tree but has to merge the content of both trees. The operator merges the right-hand tree with the left-hand tree by scanning the right-hand tree and inserting each key-value pair it encounters into the left-hand tree. The realization of the `ixUnionDistinct` only differs from the `ixUnion` in a way that it does a conditional insert on the left-hand tree to avoid duplicates.

Example 3 *Figure 3 illustrates an example of an* `ixUnion`*, where Figure 3(a) shows the left-hand tree and Figure 3(b) shows the right-hand tree. The first key, the scan encounters is 2. Since this key is already present in the left-hand tree, it appends the corresponding value to its L2Items list. Key 61866 on the other hand, requires a split of a level 2 node because the bucket is already occupied by another key. Then, we use a simple sequence of inserts into the left-hand tree. The resulting prefix tree is illustrated in Figure 3(c).*

The realization of this `ixUnion` operator is very similar to the `ixDiff` that removes the result set of the right-hand tree from the left-hand tree. For doing this, the operator also scans the right-hand tree and deletes each key-value pair from the left-hand tree. The `ixIntersect`, on the other hand, works more like the `ixJoinEqui` operator, because it has to scan both prefix trees synchronously and just prunes all those keys from the left-hand tree, which are not present in both trees. All of those binary operators have in common, that they manipulate the left-hand tree and that they are able to prune subtrees but still pass an intermediate prefix tree to the following operator.

ixSort: The unary `ixSort` operator is able to sort an input prefix tree by one or more attributes. This can be done either in an ascending or descending order for all attributes. We take advantage from the order preserving characteristic of the prefix tree. At first the operator has to scan the input tree. While doing this, it concatenates all sorting attributes of each tuple. Afterwards, the operator inserts this newly created key into a new prefix tree. This way, the sorting can be achieved with a worst case complexity of $O(kn)$, where k denotes the concatenated key length and n denotes the input cardinality.

ixSelectOr: The operator selects a set if disjunctive predicates on the same attribute. Practically, this is the realization of an *attribute IN (...)* in the where-clause of an SQL statement. We can execute this by simply pruning every path in the prefix tree that does not belong to any predicate in this IN-List.

Example 4 *To have a closer look at this operator, Figure 4 shows an example that filters for the predicates: 2, 61722 and 61866 of an attribute. The* `ixSelectOr` *operator starts scanning the root node of the prefix tree and cuts of every path, for which no appropriate prefix exists in the list of predicates. After this, the operator descents each path that is still there and repeats this procedure until there is no path available anymore. We gain performance improvements if all predicates inside the list share the same prefix. In this case, we can prune long paths at its root and only need to walk down a single path. Another alternative is to map the entire operator to an* `ixJoinEqui` *in the sense of query-data-joins [4]. In this case, we have to create a temporary prefix tree, which contains all the predicates. By computing an Equi-Join on the input tree and the temporary predicate tree, we avoid scanning the entire predicate list multiple times.*

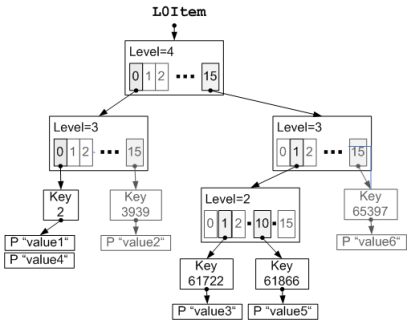


Figure 4. Example of an `ixSelectOr`

3 Project Status and Future Work

In the following, we give a brief overview of the current project status and our planned future work.

3.1 Current Project Status

Based on our conceptual ideas and the initial prototype of the *generalized trie*, we started to build the required prefix-tree-based query processing framework for this project. This includes a runtime for prefix-tree-based operators and query processing, memory and record management as well as meta data management. We also investigated the inter-influences to transaction management and query compilation.

The current status of the project is a partially finished promising initial prototype that allows for preliminary experimental investigation. However, many aspects that offer further optimization potential are not addressed so far such that a first completely finished prototype is expected for the end of 2011.

3.2 Used Future SOC Lab Resources

The objective of this project is to leverage the HPI Future SOC Lab infrastructure in order to evaluate our initial prototype in large-scale infrastructures. Thus, we use the available resources mainly with the aim of experimental evaluation. This includes, in particular (1) the scalability with increasing data size (in-

memory), and (2) the scalability with increasing number of threads. Until now, we used the following HPI Future SOC Lab resources:

- *Fujitsu RX600 S5 1:* CPU: 4 x Xeon (Nehalem EX) E7530, 256 GB RAM, shared,
- *Fujitsu RX600 S5 2:* CPU: 4 x Xeon (Nehalem EX) E7550, 1024 GB RAM, exclusive, and
- *Hewlett Packard DL980 G7-1:* CPU: 8 x Xeon (Nehalem EX) E7560, 2048 GB RAM, exclusive.

The major benefit for us is the possibility to evaluate our in-memory prototype on large-scale data sets due to the available main memory resources. This allows us to adapt our framework with regard to the obtained results, which enables a future-oriented investigation of the idea of query processing on prefix trees.

3.3 Next Steps

Beside the outstanding work for the integrated prototype, there is plenty of future work regarding further optimization potential. This includes the following three major research directions:

- Memory Management (garbage collection, optimized lazy *copy-on-write* for efficient copying),
- Query Processing (recycling intermediates, hybrid row/column storage, physical operator alternatives, cost-based query optimization), and
- Scalability (scalability with increasing data sizes, inter-operator parallelism by pipelining subtries, intra-operator parallelism by task partitioning).

4 Conclusions

Based on the requirements of (1) balanced read/write performance and (2) efficient index support for point and range queries, we proposed the vision of query processing on prefix trees. We sketched the project idea as well as promising preliminary conceptual results on prefix-tree-based operators. Efficient query processing over huge evolving data sets will enable a broader use of the consolidated enterprise data. Finally, this is a fundamental prerequisite in order to (1) extend the scope of BI from strategic and dispositive levels to the operational level, and to (2) enable advanced data analytics that goes beyond traditional data aggregation.

Acknowledgment

We want to thank Peter Benjamin Volk for extensive discussions on the architecture of the initial prototype as well as Frank Dietze, Steve Reiniger, and Thomas Dedek for their efforts on implementing parts of this prefix-tree-based query processing framework.

References

[1] Deepak Agarwal, Datong Chen, Long ji Lin, Jayavel Shanmugasundaram, and Erik Vee. Forecasting High-Dimensional Data. In *SIGMOD Conference*, 2010.

[2] Matthias Boehm, Patrick Lehmann, Peter Benjamin Volk, and Wolfgang Lehner. Query Processing on Prefix Trees. In *Fall 2010 Future SOC Lab Day (HPI Technical Report)*, 2011.

[3] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, 2011.

[4] Sirish Chandrasekaran and Michael J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.

[5] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[6] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD Conference*, 2010.

[7] Umeshwar Dayal, Malú Castellanos, Alkis Simitsis, and Kevin Wilkinson. Data Integration Flows for Business Intelligence. In *EDBT*, 2009.

[8] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2), 2010.

[9] IDC. *The Digital Universe Decade - Are You Ready?* IDC, 2010.

[10] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[11] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*, 2010.

[12] William O'Connell. Extreme Streaming: Business Optimization Driving Algorithmic Challenges. In *SIGMOD Conference*, 2008.

[13] Elizabeth G. Reid. Design and Evaluation of a Benchmark for Main Memory Transaction Processing Systems. Master's thesis, MIT, 2009.

[14] Richard Winter and Pekka Kostamaa. Large Scale Data Warehousing: Trends and Observations. In *ICDE*, 2010.