Operating on Hierarchical Enterprise Data in an In-Memory Column Store

Christian Tinnefeld, Bjoern Wagner, Hasso Plattner
Hasso Plattner Institute, University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
christian.tinnefeld@hpi.uni-potsdam.de, bjoern.wagner@hpi.uni-potsdam.de, hasso.plattner@hpi.uni-potsdam.de

***Abstract*—Enterprise data management is currently separated into online transactional processing (OLTP) and online analytical processing (OLAP). This separation brings disadvantages such as the need for costly data replication, maintaining redundant systems or the inability to run data reports on the latest transactional data. Academia and industry are working on a reunification by storing all data in a single columnar in-memory database which is designed to sustain high transactional workloads while simultaneously handle complex analytical tasks as well. Since enterprise data often includes hierarchical data, this paper focuses on modeling, storing, and operating on hierarchical data in an in-memory columnar database. The paper contributes by describing the implementation of the most frequently used hierarchical data operations on such a database while maintaining the ability to execute performant analytical queries on such data as well. A set of benchmarks demonstrates that hierarchical data operations can be executed up to three times faster on an in-memory column store than on an in-memory row store. The paper closes with a discussion which enterprise applications can benefit from this contributions.**

Keywords-hierarchical data; enterprise data management; in-memory column store; SanssouciDB.

I. Introduction

Hierarchies are very common in every day life. Organizational structures of companies, books, websites and most file systems have a natural hierarchical structure. Moreover hierarchies help to abstract complex things. E.g. software engineers use object-oriented decomposition techniques to encapsulate complex applications into independent modules. The object hierarchy and inheritance are two of the key concepts of object-oriented programming. Even one of the first scientific data model proposals, the IBM Information Management System, was based on a hierarchical data structure, long before the rise of relational database management systems [1]. There have been many proposals to store hierarchical data in relational databases as well as in specialized graph databases. This paper explores the opportunities of in-memory columnar databases for the persistence of hierarchical data. These columnar databases are superior in processing speed of huge datasets and provide high scalability especially in parallel scenarios. In addition to that, columnar databases sustain high transactional and analytic workloads at the same time. Therefore, they are used for reuniting transactional and analytical workloads which are currently separated in the context of enterprise data management [2].

The remainder of this paper is organized as follows. Section II describes several enterprise applications that heavily operate on hierarchical data in order to illustrate the need and the requirements towards operating on hierarchical data in the context of enterprise data management. Section III describes the relevant concepts in terms of encoding, data model, and data operations and their implementation in terms of describing the resulting SQL statements. The subsequent Section IV evaluates the performance by performing measurements on SanssouciDB, which is a prototypical in-memory columnar database [3]. Section V closes the paper with a summary of the most important insights and by discussing the implications for enterprise applications.

II. Hierarchcial Data in Enterprise Applications

This section introduces enterprise application domains with a strong focus on hierarchical data. The discussion focuses on central characteristics of the applications and the resulting requirements towards data management.

Supply Chain Management Supply chain management tries to improve the flow of materials, information and financial resources within a company and between different companies [4]. The foundation of supply chain management is information sharing, coordinated planning, scheduling and execution as well as collaborative monitoring and controlling. Supply chain management enables companies to execute just-in-time production, reduce stock levels, provide better forecasting and a shorter time-to-market for new products. The reduction of vertical integration over the last decade requires a more intensive collaboration between suppliers and customers. E.g. in the automotive and high-tech industry supplier coordination is vital to keep the complex production processes up and running. At the same time, the complexity of products and production processes increased rapidly. The National Electronics Manufacturing Initiative, Inc. (NEMI) outlined in [5] that the Bill-of-Materials plays a major role in such collaboration scenarios. The Bill-of-Materials (BoM) (see Figure 1 for an example of a BoM) defines what and how many components are necessary to build a product. Therefore, the BoM is basically a hierarchy of components

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

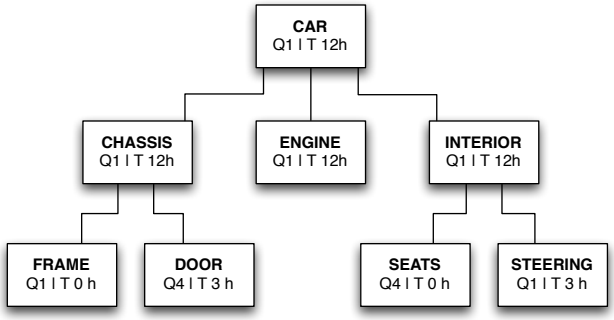


Figure 1. Bill-of-Material for a car containing quantity and assembly time

and sub components of a specific product. Many processes and stakeholders operate on the BoM, such as procurement, planning or production processes. Therefore it is necessary for supply chain management solutions to provide a fast, scalable and reliable BoM implementation. In addition to those primary transactional scenarios, the BoM is also used for analytics. For example, one might wants to know which components cause the most delays in order to improve the procurement process of this component.

Product Lifecycle Management Product lifecycle management (PLM) software is designed to manage the whole lifecycle of a product. This includes the conception, design and manufacturing phase as well as services and disposal. PLM integrates processes, data, people and provides interfaces to other business systems. One key requirement of PLM is the effective management of Bills-of-Information (BoI). Although Bills-of-Material from the previous section are also created and maintained by PLM processes, Bills-of-Information play a more important role. BoIs can be considered as a superset of Bills-of-Materials, because a BoM only contains all information necessary to assemble the product and a BoI holds lots of additional data generated during design, testing, manufacturing, sales and support of the product. A major goal of product lifecycle management is to reduce the time-to-market by centralizing the data organization. This drastically removes search time for component reuse, provides more accurate results and allows a better management as well as traceability of intellectual property assets throughout the whole product life cycle. This brings up two significant technical requirements for the implementation of the BoI hierarchy. First, the BoI hierarchy needs a very fast search interface. Second, the hierarchy has to be update-optimized, because in contrast to the BoM, the BoI changes often within the design iterations.

Project Management Another important application suite based on hierarchical data is project management. Usually, at the beginning of projects at set of high-level tasks is defined. During the project these tasks are divided in more detailed sub tasks. Beside this task hierarchy, employees and resources in general are organized in a hierarchical manner. Project management software uses those structures e.g. to calculate project costs, estimates, and reporting.

Workforce Scheduling The scheduling of a workforce is closely related to project management, but has special requirements concerning the hierarchy implementation. Project management focuses on static calculations and workforce scheduling runs more sophisticated optimizing algorithms. Workforce scheduling is calculated based on a hierarchy of tasks and resources in oder to find an optimal schedule to utilize all available resources. This is critical for utilize the available resources for a project in the most optimal way.

The preceding application domains gave an overview of the broad spectrum of technical requirements for processing hierarchies. It turned out that hierarchy implementations have to tackle read as well as write workloads. In addition to that real-time analytics on the complete hierarchies play an important role e.g. when finding out certain properties of a BoM or to find out how many resources are occupied for a certain task within a project.

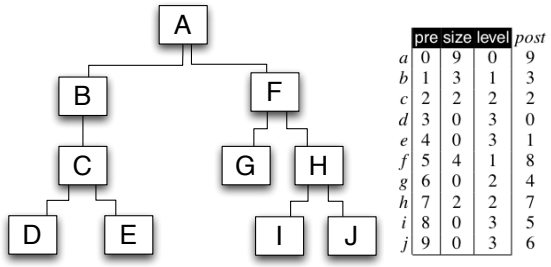
III. Concepts and Implementation

This section focuses on the concepts and their implementation for handling hierarchical data in an in-memory column store. The example of the perviously introduced Bill-of-Material is used as requirement driver throughout this section.

A. Preorder and Postorder Encoding

As mentioned in the previous section, fast execution of queries along the 4 main axes (ancestor, descendant, preceding, following) is crucial for Bill-of-Material processing. According to Grust et al. [6], the 4 main axis of each node in the tree partition the tree in 4 disjunct sets. The union of all those sets with the node itself contains all nodes of the tree exactly once. Grust further defines a mapping of tree nodes to a relational structure that preserves this partitioning and retrieves all the main axes using a simple range query. This mapping is based on so called preorder and postorder tree traversal. A preorder traversal assigns an unique rank to each node in the tree before its children are recursively traversed from left to right. The root node usually has the rank 0. Postorder traversal also assigns an unique rank to each node, but the value is assigned after the children has been traversed recursively.

In [7], Boncz et al. optimize the pre- and postorder encoding by size and level attributes that replace the postorder attribute. The size attribute describes the number of descendants and the level attribute describes the number



	pre	size	level	post
a	0	9	0	9
b	1	3	1	3
c	2	2	2	2
d	3	0	3	0
e	4	0	3	1
f	5	4	1	8
g	6	0	2	4
h	7	2	2	7
i	8	0	3	5
j	9	0	3	6

Figure 2. Example of a Pre- and Postorder table

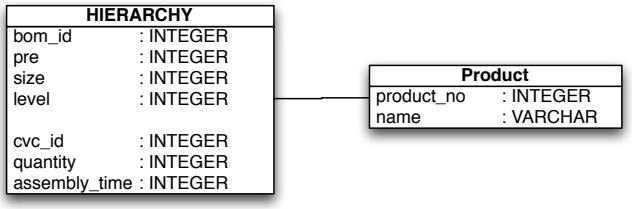


Figure 3. Schema of the pre- and postorder table

of ancestors of a node. The postorder rank is not stored explicitly in this schema anymore, but can be calculated for any node v via post(v) = pre(v) + size(v) - level(v). Figure 2 shows an example of the optimized encoding, Figure 3 depicts the resulting schema.

B. Operations

This subsection describes the implementation of the needed operations. The description is illustrated with SQL queries that refer to the relational schema introduced above. The most important operations are compared in a benchmark in the next section.

1) Number of Nodes in the Tree: The calculation of the number of nodes in a tree can be straightforward or complex depending on the implementation. E.g. if each node is related to one row in a relational database the number of node equals the number of rows that can be retrieved with a simple count() SQL statement as shown in Listing 1, since each node in the tree is represented by a database row.

Listing 1. Query to calculate the number of nodes in the tree

```
SELECT count(pre) FROM TREE
```

2) Number of Leaves in the Tree: A leave is node without any child nodes. Hence, the number of leaves complies with the number of nodes without any children in the tree. The number of leaves in the tree can be calculated using the size attribute of the node with the SQL query of Listing 2.

Listing 2. Query to calculate the number of leaves in the tree.

```
SELECT count(pre) FROM TREE
WHERE      size=0
```

3) Height of the Tree: The height of the tree expresses the maximum distance of a leave to the root node in nodes. The height of the tree can be derived from the maximum level attribute plus 1, because it is zero based. The query can be found in Listing 3.

Listing 3. Query to retrieve the height of the tree.

```
SELECT max(level) + 1 FROM TREE
```

4) Finding the Root Node: The root node is the only node on level 0 and can be found with the following SQL statement as shown in Listing 4

Listing 4. Query to find the root node.

```
SELECT * FROM TREE
WHERE      level=0
```

5) Main Axes Query: The main axes traverse four different kind of axis. The ancestor axis describes the parent nodes of an input node excluding the node itself. The result list is ordered and starts with the root node of the tree. Because of that, the result can also be considered as path from the root node to the input node. The descendant axis is the opposite of the the ancestor axis. It describes the child nodes of the input node and recursively the descendants of those child nodes. As already mentioned, the tree structure is similar to a XML data model. Therefore the best way to illustrate preceding axis, to select all XML elements that have been closed before the context node. The following axis contains all elements that begin after the context element has been closed.

Because the 4 main axes partition the tree in 4 disjunct regions the range queries can be derived from the pre-post-plane. The context_node_pre and context_node_post are named parameters representing the pre- and postorder rank of the context node. The ORDER BY statement is used to ensure document order.

Listing 5. Query for the ancestor axis of the context node.

```
SELECT * FROM TREE
WHERE      pre < :context_node_pre
           AND post > :context_node_post
ORDER BY pre
```

Listing 6. Query for the descendant axis of the context node.

```
SELECT * FROM TREE
WHERE      pre > :context_node_pre
           AND post < :context_node_post
ORDER BY pre
```

Listing 7. Query for the preceding axis of the context node

```
SELECT * FROM TREE
WHERE      pre < :context_node_pre
           AND post < :context_node_post
ORDER BY pre
```

Listing 8. Query for the following axis of the context node.

```
SELECT * FROM TREE
WHERE      pre > :context_node_pre
           AND post > :context_node_post
ORDER BY pre
```

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

6) Subtree matching: Subtree matching is more complex than the operations presented above. First of all, according to [8] there are several kinds of subtree matching. For an ordered rooted tree T with vertex set V and edge set E, an rooted ordered Tree T' is a bottom-up subtree if and only if

1) $V' \subseteq V$
2) $E' \subseteq E$
3) the labeling of V' and E' is preserved in T'
4) if vertex $v \in V$ and $v \in V'$ then all descendants of v must be in V'
5) left-to-right ordering among siblings of T must be preserved in T'

The bottom-up subtree matching defined by Zaki [9] is implemented by transferring the depth encoding tree representation to the relational model. The basic idea behind the subtree matching query is, to pick one tree and match it against all the others by joining the hierarchy table with itself. The query in Listing 9 shows one possibility to implement subtree matching in SQL. It takes the bomId and the number of nodes of the subtree as input parameters. It is possible to retrieve the number of nodes of the subtree by another join with the hierarchy table, but this would also introduce more complexity in this example. Since the subtree matching operates on products, the hierarchy table is joined on the cvcId attribute and where-clause ensures, that subtree isn't matched against itself. But the most important part of this query is the group-by-statement. First of all, the bomId attribute separates the result of different trees. Second, the *tree.pre-subtree.pre* and *tree.post-subtree.post* statements group all nodes of the tree together, that have a similar preorder and postorder rank. To be more exactly, the structure is of the subtree is preserved in the tree, but the position of the root node of subtree in the matching tree doesn't matter. Hence all nodes of a tree that match the structure of the subtree are contained in one group. The last having-clause verifies that all nodes of the subtree can be found in the matching tree. Finally, the select-statement returns the id of the matching tree as well as the preorder and postorder rank of the node that matches the root node of the subtree.

Listing 9. Query for subtree matching

```
SELECT    tree.bom_id, min(tree.pre), max(tree.post)
FROM      hierarchy subtree INNER JOIN hierarchy tree
ON        tree.cvc_id=subtree.cvc_id
WHERE     subtree.bom_id = :subtree_bom_id
          AND tree.bom_id <> :subtree_bom_id
GROUP BY  tree.bom_id, tree.pre-subtree.pre,
          tree.post-subtree.post
HAVING    count(tree.pre) = :subtree_node_count
```

7) Add, Move, Delete Nodes or Subtrees: The following operations change the structure of the tree. An insert operation inserts new nodes or subtrees below an existing node into the tree. Moving a node or a subtree means to change the parent of the node or the root node of the subtree. Deleting a node from a tree detaches it from the hierarchy. If this node has descendants, all descendants are also removed.

The implementations of the read-only statements aren't complex and can be executed fast on a relational database system as seen above. Unfortunately, this is not completely true for structural update operations of the tree. No matter if a single node or a subtree is inserted, moved, or deleted, all pre values following the insert node and all size values of ancestors of the insert node have to be updated. Especially the first update is critical from a performance perspective, because it has to update the half table on average. A highly optimized implementation of the update operation including transaction management was proposed by [7]. Listing 10 illustrates a much simpler query that inserts a node into a tree.

Listing 10. Query to insert a node or subtree

```
UPDATE    Hierarchy SET pre=pre+:subtree_node_count
WHERE     pre > :insert_node_pre
          AND post > :insert_node_post
UPDATE    Hierarchy SET size=size+:subtree_node_count
WHERE     pre < :insert_node_pre
          AND post > :insert_node_post
```

8) Analytical Query: Analytical operations often operate on all nodes of the tree instead of a subset. Path based aggregation traverses the tree or a subtree recursively from root to leaves and applies a arithmetical operation on one attribute of each node on the path.

The analytical query operation flattens the Bill-of-Material hierarchy by processing each component of each product and aggregate the quantity of each node. The result is a simple list of all components necessary to assemble the product from scratch as shown in Listing 11.

Listing 11. Query for flattening data hierarchy

```
SELECT    Hierarchy.bom_id, Cvc.name,
          sum(Hierarchy.quantity)
FROM      Hierarchy INNER JOIN Cvc
ON        Hierarchy.cvc_id=Cvc.id
GROUP BY  Hierarchy.bom_id, Cvc.name
```

IV. BENCHMARKING

This section evaluates the performance of the previously discussed tree implementations by benchmarking the major tree operations and compare the results. The benchmarks are executed on two different database systems: one system is a row-oriented MySQL database running on a RAMDisk, the other database is a column-oriented SanssouciDB [3]. The machine they are running on is an Intel Xeon E5450 Quadcore with 3 GHz and 32GB of RAM.

A. Read Operation

We selected the 4 main axis (ancestor, descendant, preceding, and following) as example query for read operations.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

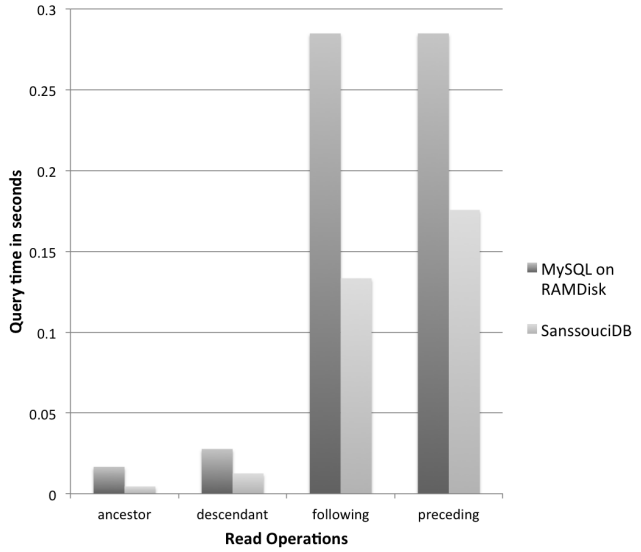


Figure 4. Read operations on a tree with 500.000 nodes

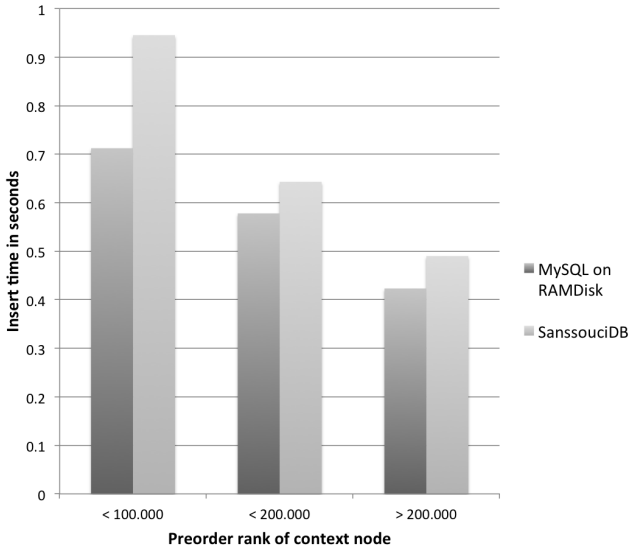


Figure 5. Insert operation on a tree with 500.000 nodes

Figure 4 shows the results for the read operations on a tree with 500.000 nodes. The column store is faster for all 4 read queries. However it occurs that there is a noticeable variance between the different axis on both databases. While the ancestor axis performs very well, the preceding and following axis operation is about ten times slower. This can be explained by the structure of the tree. Because the width of the tree is bigger than the height, the result set of the following axis operation is usually also larger. E.g. the result set of the following axis operation for a node with a small preorder rank can contain almost the complete tree, but the ancestor axis result set is limited to the height of the tree.

B. Write Operation

The write operation we benchmarked adds a new child node to a leaf of the tree. It turns out, the lower the preorder rank of a leaf is, the slower performs SanssouciDB compared to MySQL. Obviously, SanssouciDB is optimized for read-intensive workloads, but this case is compounded by the fact that for lower preorder ranks almost all nodes in the hierarchy table are updated during an insert.

C. Analytic Operation

Finally, the analytic operation flattens the Bill-of-Material hierarchy as described in the previous section. Throughout the different measurements, the number of nodes has been increased, starting with 100.000 nodes up to 1.000.000 nodes. Especially in the latter case, the column store can play off its analytic strengths and passes the MySQL up to a factor three. This is remarkable, because it shows that the described data operations on SanssouciDB provide a performant execution of transactional operations (such as reading and inserting nodes) as well as fast analytics on the same data. Although the non-exponential growth of the response time when adding more nodes indicates that even bigger trees can be analyzed while still maintaining a sub-second response time.

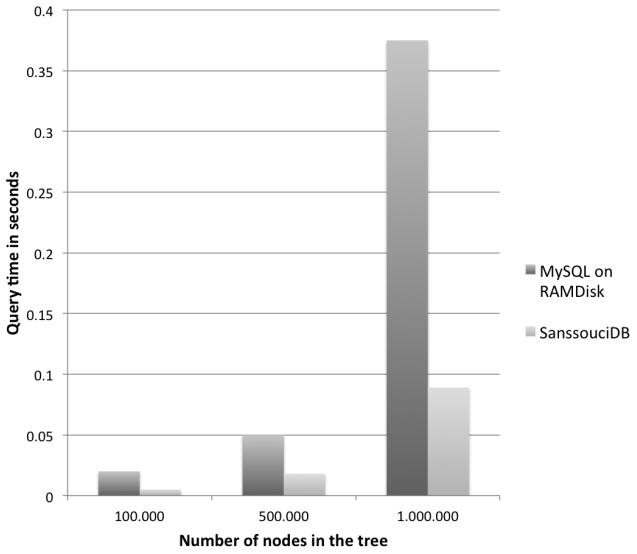


Figure 6. Analytic operation on a tree with varying number of nodes

V. CONCLUSION

This paper shows that the reunification of transactional and analytical workloads on one single database is possible even when operating on hierarchical data. The described tree operations build the foundation for serving enterprise applications such as supply chain management, product lifecycle management, project management or workforce

scheduling. Furthermore, the paper demonstrates that the described operations can be executed in a performant manner on an in-memory column store such as SanssouciDB.

Since this is the technical prerequisite for enabling more analytical functions within applications that work on hierarchical data, the following new application scenarios are within reach: when working on BoMs in the context of supply chain management, the properties of each component of a product can be taken into consideration when doing Supply Network Planning or Production Planning and not only the product as a whole. E.g. when a production plan has to be rescheduled due to a temporary outage of some machines, it can be instantly determined which sub-products can still be produced. Or for example in the field of product lifecycle management, analytics and simulations on every new iteration of a product is possible which leads to quicker validation of new product proposals and allows the product designer to validate that a new product fulfills certain requirements while he is still working on it.

REFERENCES

[1] M. Stonebraker and J. M. Hellerstein, "What Goes Around Comes Around," *Architecture*, 2005. [Online]. Available: http://www.w3.org/TR/xpath/

[2] J. Krueger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner, "A case for online mixed workload processing," in *Proceedings of the Third International Workshop on Testing Database Systems*, ser. DBTest '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/1838126.1838134

[3] H. Plattner and A. Zeier, *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, 2011.

[4] G. Knolmeyer, P. Mertens, A. Zeier, and J. Dickersbach, *Supply Chain Management Based on SAP Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[5] S. Your and S. Chain, "In Search of the Perfect Bill of Materials (BoM)," *Material Interchnage Journal*, no. March, 2002.

[6] T. Grust, "Accelerating XPath location steps," *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*, p. 109, 2002.

[7] P. Boncz, T. Grust, M. V. Keulen, S. Manegold, J. Rittinger, J. Teubner, and C. W. I. Amsterdam, "MonetDB / XQuery : A Fast XQuery Processor Powered by a Relational Engine," 2006.

[8] Y. Chi, "Frequent Subtree Mining — An Overview," *Fundamenta Informaticae*, pp. 1001–1038, 2001.

[9] M. J. Zaki, "Efficiently mining frequent trees in a forest," *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02*, p. 71, 2002.

[10] P. A. Boncz, J. Flokstra, T. Grust, M. van Keulen, S. Manegold, K. S. Mullender, J. Rittinger, and J. Teubner, "Monetdb/xquery-consistent and efficient updates on the pre/post plane," in *EDBT*, ser. Lecture Notes in Computer Science, Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, Eds., vol. 3896. Springer, 2006, pp. 1190–1193.

[11] Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, Eds., *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3896. Springer, 2006.

[12] T. Grust, S. Sakr, and J. Teubner, "Xquery on sql hosts," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 252–263. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316713

[13] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559846

[14] G. Valiente, "Algorithms on trees and graphs," *Security*, 2002.

[15] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann, "Anatomy of a native XML base management system," *The VLDB Journal The International Journal on Very Large Data Bases*, vol. 11, no. 4, pp. 292–314, Dec. 2002.

[16] M. M. Tseng, "Generic Bill-of-Materials-and-Operations for High-Variety Production Management," *Concurrent Engineering*, vol. 8, no. 4, pp. 297–321, Dec. 2000.

[17] X. Shen, X. Papademetris, and R. T. Constable, "Graph-theory based parcellation of functional subunits in the brain from resting-state fMRI data." *NeuroImage*, vol. 50, no. 3, pp. 1027–35, Apr. 2010.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, 2001.

[19] P. Umversay and W. Lafayette, "Pattern Matching in Trees," *Computing*, vol. 29, no. I, pp. 68–95, 1982.

[20] T. Grust, "Staircase Join : Teach a Relational DBMS to Watch its (Axis) Steps," *Evaluation*.

[21] K. Beyer, N. Seemann, T. Truong, B. Van der Linden, B. Vickery, C. Zhang, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, and H. Pirahesh, "System RX," *Proceedings of the 2005 ACM SIGMOD international conference on Management of data - SIGMOD '05*, p. 347, 2005.

[22] M. H. Xiong, S. B. Tor, and L. P. Khoo, "WebATP: a Web-based flexible available-to-promise computation system," *Production Planning & Control*, vol. 14, no. 7, pp. 662–672, Jan. 2004.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4