MESSAGE INDEXING FOR DOCUMENT-ORIENTED INTEGRATION PROCESSES

Matthias Boehm, Uwe Wloka
Dresden University of Applied Sciences, Database Group, 01069 Dresden, Germany
mboehm@informatik.htw-dresden.de, wloka@informatik.htw-dresden.de

Dirk Habich, Wolfgang Lehner
Dresden University of Technology, Database Technology Group, 01062 Dresden, Germany
dirk.habich@inf.tu-dresden.de, wolfgang.lehner@inf.tu-dresden.de

Keywords: Enterprise Application Integration, Message Indexing, MIX, Algorithms, Data Structures, Performance

Abstract: The integration of heterogeneous systems is still an evolving research area. Due to the complexity of integration processes, there are challenges for the optimization of integration processes. Message-based integration systems, like EAI servers and workflow process engines, are mostly document-oriented, using XML technologies to achieve suitable data independence from the different and particular proprietary data representations of the supported external systems. However, such an approach causes large costs for single-value evaluations within the integration processes. At this point, message indexing, adapting extended database technologies, can be applied in order to achieve better performance. In this paper, we introduce our message indexing structure *MIX* and discuss and evaluate immediate as well as deferred indexing concepts.

1 INTRODUCTION

The horizontal integration of systems by message-based communication via middleware products is a widely used method of application integration to ensure a loose coupling of participating systems and applications. In order to reach the required data independence, most integration platforms—like the SAP Exchange Infrastructure (XI) or TransConnect—work in a document-oriented way. There, external data is translated into XML and thus could be processed in a uniform way. Thus, the term *messages* refers to XML messages with additional attributes and binary attachments. This claim is also true for the majority of workflow management systems.

Due to the document-oriented approach, access to single values of a message (e.g., in `ASSIGN` or `SWITCH` operators) is cost-intensive, where those costs increase with the data size. Imagine data-centric integration processes with several attribute evaluations for control-flow determination; here, the document-oriented approach is obviously inefficient. In order to eliminate these disadvantages, message indexing can be used. This leads to a performance improvement which is proportional to the increasing data size. Hence, in this paper, we present a new message indexing approach in order to enable an efficient read and write access to single message values which are required during process execution. Therefore, the paper is structured as follows. In Section 2, we motivate our full approach and give a detailed problem description. Further, we present *MIX*—a transient message indexing structure—and its processing concepts in Section 3. In Section 4, we evaluate our approach with a number of experiments showing the impact of the different concepts and techniques. Finally, in Sections 5 and 6, we give an overview of related work, summarize our results, and conclude the paper. For a full version of the paper—including all algorithms—see (Böhm et al., 2008).

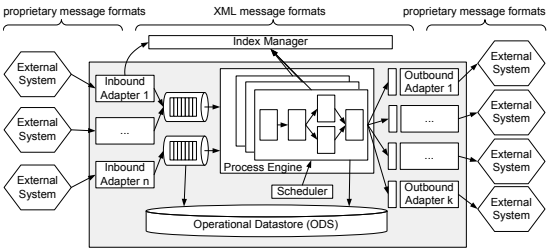

Figure 1: MOM Reference Architecture

2 PROBLEM DESCRIPTION

The indexing of messages for document-oriented integration processes is very different to indexing statically stored data. These approaches often assume that data is statically stored with a low update rate and read optimization is the goal. However, in our application area, high message throughput results in a high update rate. Hence, the write performance is very important.

Architectural Considerations In order to generalize document-oriented integration processes, Figure 1 introduces a Message-Oriented Middleware (MOM) reference architecture. There, proprietary messages are received from external systems with passively listening `inbound adapters`. These `inbound adapters` transform the incoming messages into an internal XML message format. Those messages are put into specific `message queues`, which separate the inbound side from the outbound side. Here, messages are made persistent for scalability and recovery purposes. Further, the core `Process Engine` uses specific `outbound adapters` to actively interact (pull and push) with external systems. Also, the `outbound adapters` realize translations between XML and the proprietary data format of the target system. Thus, the right places of index maintenance for incoming messages are the inbound and outbound adapters because here, all attributes are scanned and transformed into the internal XML format. The indexed attributes are then used within the different integration processes (e.g., in `ASSIGN` and `SWITCH` operators). Fundamentally, the specification of which attributes should be indexed can be explicitly defined by a user or implicitly derived from the integration process workflow description.

Characteristics of Message Indexing Based on the architectural considerations, there are specific characteristics of message indexing, which allow the usage of context knowledge and thus more efficient index maintenance. Each message includes an *sequence-generated message ID*. Thus, the message IDs are an ascending sequence (not clean because of concurrency) of `BIGINT` values and are thus advantageous for the use as index tree key criteria. Further, transformation operations within the process cause the creation of new messages with new message IDs. Thus, index scans must be aware of *dynamic message ID changes* and *dynamic attribute name changes*. Next, a *high update rate* (insert, update and delete) is caused by the fact that indexed attributes are often read only once. Finally, integration processes have a *throughput-oriented optimization goal*, which offers the possibility of asynchronous processing, where latency time is acceptable.

3 MIX: MESSAGE INDEX

The message indexing structure *MIX* uses context knowledge to speed up the index maintenance. In this section, we define the core index structure and describe our techniques for index maintenance.

3.1 Index Structure

The index structure is an extended B^+ tree. Indexed attributes are exclusively stored within the leaf nodes, while all other nodes contain key values. Based on this core model, we define our indexing rules, which differ from the original B^+ tree. The core index tree comprises three types of nodes. First, each *index node* comprises $n-1$ search keys sorted in ascending order and n (node size) pointers to child nodes. Here, all types of nodes could be children of index nodes and the message ID is used as key criterion. Second, all indexed attributes are stored at the lowest tree level within the *leaf data nodes*. Such a node contains a hashmap of several indexed attributes related to one message ID. In this hashmap, (1) the hash values of the XPath expression for the specific attribute and (2) the attribute values are stored. Third, the *Leaf node pointer* is used similar to leaf data nodes, but it does not contain any materialized data but instead it has a pointer to either a leaf data node or another leaf node pointer.

Definition 1 *We define that the* MIX *structure is imbalanced under all circumstances. However, the tree works in a self-organizing way, which lets it tend towards a balanced state. Leaf data nodes are always present on the lowest tree level.*

Definition 2 *Let f be the fill factor of the tree. We define that the root node and all other index nodes of the tree must have at least one pointer to a child node. The index nodes are allowed to have an f of* 1.0 *with n pointers to child nodes. The leaf data nodes must include at least one (XPath expression, value) tuple and the leaf node pointer must reference exactly one leaf data node or leaf node pointer.*

Definition 3 *Let n be the node size; then, an index node may include $n-1$ search keys and n node pointers at the most. We define that n may vary between nodes for adaptively changing n during runtime.*

Definition 4 *We define that single elements or index nodes and thus complete subtrees may be inserted into and deleted from the index tree.*

Figure 2 illustrates the message index macro architecture. Basically, there is the `core index tree`, which works according to the above presented definitions. Furthermore, all accesses to the

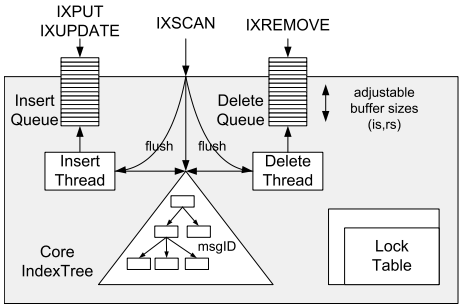

Figure 2: Message Index Macro Architecture

`core index tree` are synchronized using an external `lock table`. For that, only the states `NO_LOCK`, `READ_LOCK` and `WRITE_LOCK` are supported. The locking mechanism is based on hierarchical locks for the specific message ID. Aside from the structural organization, there are four types of interactions with the *MIX*: `IXPUT`(inserting indexed values and subtrees), `IXUPDATE` (updating indexed values), `IXREMOVE` (deleting indexed values and subtrees) and `IXSCAN` (reading indexed values). Here, only `IXSCAN` works in a synchronous and thus blocking way. All other interactions may be used both synchronously (immediate) or asynchronously (deferred). For the case of deferred processing, we provide an `insert queue` as well as a `delete queue`. However, a synchronous `IXSCAN` causes the queues to be flushed. This structure is by design write-optimized (deferred techniques) and allows for dynamic updates of indexed values with low costs.

An example *MIX* index structure is illustrated by Figure 3 and shows a balanced index tree with $n = 3$ and nine leaf nodes. Especially the leaf node pointers are specific to our approach. There is a pointer from leaf node (msgID = 104) to leaf node (msgID = 103). In conclusion, all $\texttt{IXSCAN}_{104}$ will imply an $\texttt{IXSCAN}_{103}$ with very low overhead. Furthermore, there is also an indirect leaf node pointer (msgID=117) that refers to another leaf node pointer.

3.2 Inserting Indexed Values

The insertion of indexed values is realized with the interaction `IXPUT`. Here, we distinguish two techniques: the `immediate IXPUT` and the `deferred IXPUT`. The algorithm for `immediate IXPUT` is realized in the specific node type *Index Node*. The message ID, an XPath expression, an instance value and the current depth (decremented for each index level) are given. Basically, there is a loop over all node pointers of the current index node, searching the right key. If it is found, three different cases may occur. First, the current node pointer could be `NULL`; then, a new child index node is created, depending on the given index depth. Second, the pointer could reference a leaf node. If the msgID is equivalent, the new attribute can simply be inserted; otherwise, the current leaf node has to be reordered or split to a new index node. Third, the pointer is not a leaf node; then the `IXPUT` event is recursively pushed down, or if the child node is already full, it also has to be split.

Especially the sequence-generated message ID in connection with the—not per definition balanced—tree allow for optimization. If the index tree is full, which means that it is balanced and it has a fill factor of 1.0, the next `IXPUT` would result in an index split. In order to use the context of sequence-generated keys, we use the *symmetric index grow-up*. Here, the root index node is checked to see if it is full. If it is, a new index node with n node pointers is created. After that, the whole old root node is inserted into the new root tree. The result of this is that—in case that all message IDs are inserted in sequence (default)—no index node splits will occur any longer, which makes the `IXPUT` operation really efficient.

However, there is further optimization potential. If we know that all inserted values are (nearly) in sequence, the tree path search for each new value can be optimized by a deferred algorithm which reduces the tree path searches. The precondition for that is the introduced `insert queue`, where all insert requests are asynchronously stored. In case the `QUEUE_SIZE` is reached, a specific thread locks the queue and sorts the entries with the quick-sort algorithm. Now, we create a new index node *tmp* with n key slots. Further, we know the last inserted message ID (lastID). For each request, it is tested whether or not the message ID is larger than *lastID*. If it is, the request is inserted into *tmp*; otherwise, it fails and has to be inserted directly into the root index. Finally, the whole subtree *tmp* could be inserted into the root index and the queue can be unlocked. The `deferred IXPUT` algorithm works like the `immediate IXPUT`, except for the fact that complete subtrees can be inserted. It has to be checked if the complete subtree, with given

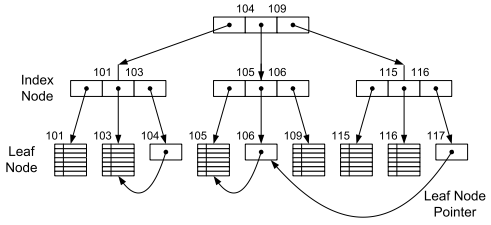

Figure 3: Example Index Tree Structure

minID and *maxID*, can be inserted into the matching node pointer. If so, the algorithm is very similar to `immediate IXPUT`. Otherwise, an original subtree splitting must occur. Due to cost-intensive splitting, all leaf data nodes are simply inserted with `IXPUT`.

3.3 Updating Indexed Values

The update of indexed values is a very specific problem, related to integration processes. There, the update is realized with the interaction `IXUPDATE`. In analogy to the insertion of indexed values, we contribute algorithms for two main techniques: the `immediate IXUPDATE` and the `deferred IXUPDATE`. Basically, an `IXUPDATE` ensures that multiple message IDs may reference one leaf data node. Therefore, the `IXUPDATE` is conceptually separated into an `IXSCAN`, the creation of the leaf node pointer to the scanned node, and an `IXPUT` of the leaf node pointer, which is also a leaf node. Although this could be realized simply on the root index level, this would be inefficient. Our main approach here is that the update request is recursively pushed down along the index tree until the old message ID and the new message ID are not contained within one pointer slot. At this point, the update event is separated into a tree scan and a tree put. Thus, the conceptual separation of events is realized at the lowest possible index level. The algorithm of `immediate IXUPDATE` searches for the right key slot using the old message ID. If a slot is determined, the new message ID is checked to see if it also could be applied to the slot. If so, the update request is recursively applied on the slot pointer. Otherwise, the lowest possible update level in the index has been reached and the separation of events occurs. In addition to the dynamic message ID update, an update request can also initiate dynamic attribute changes (with new and old XPath expressions). The leaf node pointer then also contains a mapping for these attributes. Thus, both types of dynamic updates could be realized with one generic algorithm. The `deferred IXUPDATE` is correlated to the `deferred IXPUT`. There is a unique `insert queue` for both deferred insert and deferred update requests. The enhancement is achieved by the substitution of update requests with insert requests.

3.4 Removing Indexed Values

There are as many deletion requests as insert requests within the index. Thus, an efficient `IXREMOVE` is required. Here, we distinguish as well: the `immediate IXREMOVE` and the `deferred IXREMOVE`. The `deferred IXREMOVE` can be designed to be lazier in order to achieve best performance. The core algorithm of `immediate IXREMOVE` works as follows. It has two parameters: the message ID *msgID* and a Boolean flag $force$. Note, if $force$ is set to false, then this immediate algorithm is only used for setting $delete$ flags as a precondition of the deferred algorithm. Mainly, the algorithm comprises the search for the right key slot. If the right one is found, it is checked whether of not $force$ is specified, whether the child node is a leaf node and whether the child node is unused. This latter check is caused by pending leaf node pointers (back ref counter). In case the check has been successful, the pointer is simply set to `NULL`. Otherwise, the remove event is recursively pushed down. With this algorithm, only leaf nodes are eliminated. All free index nodes have to be deleted by a `deferred IXREMOVE`. The `deferred IXREMOVE` is based on the described flagging of unused data nodes. Figure 4 shows this concept using tree annotations (+,-). So, if a node is flagged as deletable, it gets a '-', otherwise it has a '+'. A parent node is '-' if all its children are set to '-'. The `deferred IXREMOVE` only operates on the root index level, trying to cut off complete subtrees. In conclusion, only a maximum of $n-1$ node pointers and keys have to be shifted to the left. The `deferred IXREMOVE` algorithm is mainly separated into two parts: (1) the removal of subtrees and (2) the reordering of subtrees. So, in conclusion, the current index tree could be seen as a sliding window over all indexed values from past to future.

3.5 Reading Indexed Values

The reading of indexed values can be processed exclusively with an `Immediate IXSCAN`. There, the complexity of scanning an attribute is $O(\log(n))$. It comprises the search for the right key slot. If it is found, an `IXSCAN` is recursively processed on the child node. So, implicitly, three actions may occur. First, if the child node is an index node, the algorithm is called in a recursive way. Second, in case of a leaf data node, the scan is transformed into a simple hashmap lookup using the given XPath expression. Third, in case of a leaf node pointer, the referenced node is scanned.

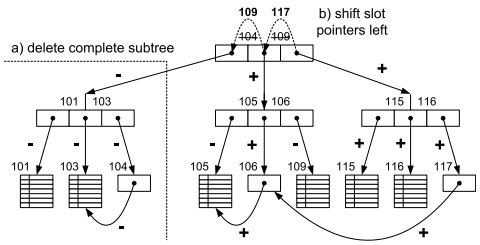

Figure 4: Deferred Index Removal

4 EVALUATION EXPERIMENTS

The message indexing structure *MIX* and its operations `IXPUT`, `IXUPDATE`, `IXREMOVE`, and `IXSCAN` were evaluated with a large set of experiments with different scale factors. Hence, we illustrate only the main experiments. When doing so, we also compare *MIX* with the alternatives, such as a B^+ tree implementation as well as DOM and SAX message scans. The experimental setup for the process type execution comprises a local computer system (CS [Dual Genuine Intel T2400, 1.5GB RAM]) and a Java implementation of the *MIX* index structure, which works in a transient way. This means the single values only reside in the main memory. We chose Java due to its relevance for document-oriented integration systems.

SAX/DOM Comparison: The index usage has to be compared with the direct access to single values of document-oriented messages. The difference is the performance impact when applying the message indexing to integration systems. We conducted experiments to measure the performance of *MIX* as well as of the XML technologies *Document Object Model (DOM)* and *Simple API for XML (SAX)*. The result is shown in Figure 6a. Obviously, using the index, significant performance improvements are possible. The message index has a scanning time independent to the message size. In contrast to this, there is an exponential scanning time for DOM, while the SAX scanning time increases linearly for an increasing message size. The message index is also nearly independent from the used fill factor due to its logarithmic complexity.

B^+ Tree Comparison: Then we compared *MIX* to a default Java B^+ tree implementation. In order to evaluate this issue, we conducted two experiments. The first experiment's focus was on the maintenance of dynamic message ID changes. *MIX* can use the leaf node pointers, while two B^+ trees (values, lookup) are required. The second experiment simply compared the insert, delete and read performance of both index structures. Figure 5 shows the results of both experiments. Obviously, the *MIX* `IXSCAN` is always superior to the B^+ tree `find` and also scales better with an increasing number of pointers. Although the B^+ tree

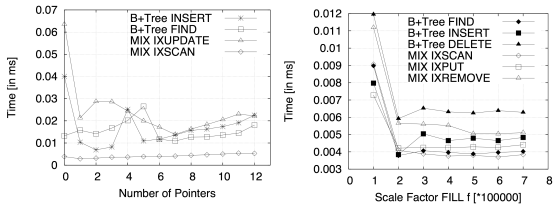

Figure 5: *MIX* / B^+ Tree Comparison

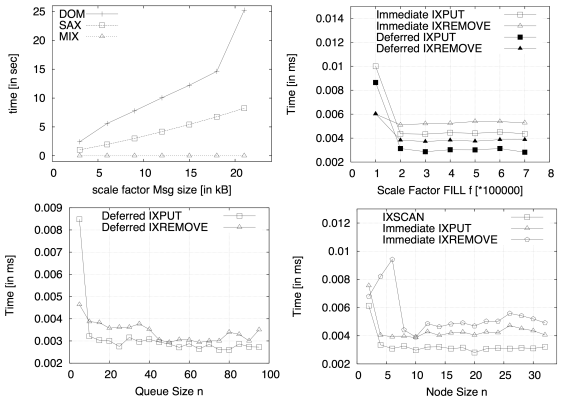

Figure 6: *MIX* Operation Comparisons

`insert` outperforms the *MIX* `IXUPDATE` for a small number of pointers, *MIX* scales much better due to the usage of update request forwarding. In summary, *MIX* is more efficient than the B^+ tree in managing node pointer references. Further, all *MIX* operations are slightly superior to the B^+ tree operations.

Deferred Technique Evaluation In a next step, we compared immediate and deferred techniques, where the latter use context knowledge to achieve much better performance. Thus, we conducted an experiment comparing the immediate and deferred `IXPUT` as well as the immediate and deferred `IXREMOVE`. The deferred techniques both use a queue size of 10. The performance was examined in dependence on the index fill factor. Figure 6b illustrates the results. The deferred techniques are superior in nearly all situations compared to their immediate counterparts. There is a large difference between `immediate IXPUT` and `deferred IXPUT`. Although the difference between `immediate IXREMOVE` and `deferred IXREMOVE` is not as large, it is still a good improvement. However, when using the `deferred IXREMOVE`, performance peaks are possible.

Optimal Configurations The chosen queue size and the node size have measurable impact on the index performance. We conducted two experiments for the *MIX* operations in order to determine optimal configuration properties. First, we experimented with different queue sizes, which are used for the deferred write operations. There, a node size of 3, a fill factor of 100,000 and a sequence factor of 0.99 was used. Figure 6c shows the performance plot. Up from a queue size of 10, there is a pretty constant performance with marginal differences for increasing queue sizes but with no significant trend towards better performance. However, there are performance peaks, caused by the characteristics of deferred processing. Second, we tried to choose the optimal node size with

the immediate operations. The node size was expected to be a compromise between high write performance (large node size) and high read performance (small node size). Figure 6d shows the result. Up from a node size of 8, an almost linear performance trace should be mentioned for increasing node sizes. The performance of read and write operations reacts in the same way for small node sizes. However, up from a node size of 26, the expected trend is shown.

5 RELATED WORK

Basically, we survey four fields of indexing which are close to our approach: XML indexing techniques, B tree optimization, B^+ tree optimization, and existing work in the field of information integration. Due to the fact that we index single message values, identified by XPath expressions, we want to separate us from XML indexing techniques. These could be classified into the groups *structural indexing* ((Chung et al., 2002), (Grust, 2002), (Haustein et al., 2005), (Kaushik et al., 2002), and (Qun et al., 2003)), *value indexing* ((Bruno et al., 2002), (Rao and Moon, 2004)) and *hybrid indexing* (where information retrieval techniques are used). Typically, when applying such indexing techniques, multiple indexes (not applicable in our context) are built, indexing *all* single values of a document. We adopt *MIX* to context knowledge of integration processes. Equal approaches—using workload characteristics—were also used for B tree indexes ((Graefe, 2004), (Graefe, 2006), (Graefe and Larson, 2001), and (Lomet, 2001)). There, specific techniques (e.g., buffering) are provided for optimizing B trees for high update rates or special hardware setups. In contrast to XML indexing techniques and B tree indexing, our index structure is very similar to well-known B^+ tree indexes, where all data reside in the leaf nodes. In particular, we want to point out (Chen et al., 2001) and (Chen et al., 2002), where internal jump-pointers from the current leaf node to the following leaf node are used in order to speed up range scans by pre-fetching. However, due to the semantic context and the type of usage, there are major differences to our approach. In the area of integration of heterogeneous systems, there is only little work on indexing. A very exciting approach is the adaptation of information retrieval methods for indexing dataspaces (Dong and Halevy, 2007). Such an inverted list (like the Hier-ATIL) would also be applicable for message indexing using the message IDs as instance identifiers and the XPath expression as keywords. However, in order to adapt to the context knowledge, a B^+ tree extension is more efficient.

6 SUMMARY AND CONCLUSION

Our intent was to optimize integration processes by applying message indexing using context knowledge about the specific characteristics of message-based and document-oriented integration processes. Therefore, we developed the message indexing structure *MIX*, which is able to handle the *dynamic message ID changes* and *dynamic attribute name changes* in a suitable way. Furthermore, we take advantage of the integration process characteristics, the *sequence-generated message IDs*, the *high update rate* and also the *throughput-oriented optimization goal* by introducing deferred index maintenance techniques.

REFERENCES

Böhm, M., Habich, D., Lehner, W., and Wloka, U. (2008). Message indexing for document-oriented integration processes. Technical report, Dresden University of Applied Sciences.

Bruno, N., Koudas, N., and Srivastava, D. (2002). Holistic twig joins: optimal xml pattern matching. In *SIGMOD*.

Chen, S., Gibbons, P. B., and Mowry, T. C. (2001). Improving index performance through prefetching. In *SIGMOD*.

Chen, S., Gibbons, P. B., Mowry, T. C., and Valentin, G. (2002). Fractal prefetching btrees: optimizing both cache and disk performance. In *SIGMOD*.

Chung, C.-W., Min, J.-K., and Shim, K. (2002). Apex: an adaptive path index for xml data. In *SIGMOD*.

Dong, X. and Halevy, A. Y. (2007). Indexing dataspaces. In *SIGMOD*.

Graefe, G. (2004). Write-optimized b-trees. In *VLDB*.

Graefe, G. (2006). B-tree indexes for high update rates. *SIGMOD Record*, 35(1).

Graefe, G. and Larson, P.-Å. (2001). B-tree indexes and cpu caches. In *ICDE*.

Grust, T. (2002). Accelerating xpath location steps. In *SIGMOD*.

Haustein, M. P., Härder, T., Mathis, C., and Wagner, M. (2005). Deweyids - the key to fine-grained management of xml documents. In *SBBD*.

Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. F. (2002). Covering indexes for branching path queries. In *SIGMOD*.

Lomet, D. B. (2001). The evolution of effective b-tree: Page organization and techniques: A personal account. *SIGMOD Record*, 30(3).

Qun, C., Lim, A., and Ong, K. W. (2003). D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*.

Rao, P. and Moon, B. (2004). Prix: Indexing and querying xml using prüfer sequences. In *ICDE*.