MULTI-PROCESS OPTIMIZATION VIA HORIZONTAL MESSAGE QUEUE PARTITIONING

Matthias Boehm, Dirk Habich and Wolfgang Lehner
Database Technology Group, Dresden University of Technology, Dresden, Germany

Keywords: Integration processes, Multi-process optimization, Message queues, Throughput improvement.

Abstract: Message-oriented integration platforms execute integration processes—in the sense of workflow-based process specifications of integration tasks—in order to exchange data between heterogeneous systems and applications. The overall optimization objective is throughput maximization, i.e., maximizing the number of processed messages per time period. Here, moderate latency time of single messages is acceptable. The efficiency of the central integration platform is crucial for enterprise data management because both the data consistency between operational systems and the up-to-dateness of analytical query results depend on it. With the aim of integration process throughput maximization, we propose the concept of multi-process optimization (MPO). In this approach, messages are collected during a waiting period and executed in batches to optimize sequences of process instances of a single process plan. We introduce a horizontal—and thus, value-based—partitioning approach for message batch creation and show how to compute the optimal waiting time with regard to throughput maximization. This approach significantly reduces the total processing time of a message sequence and hence, it maximizes the throughput while accepting moderate latency time.

1 INTRODUCTION

The scope of data management is continuously changing from the management of locally stored data towards the management of distributed information across multiple heterogeneous applications and systems. In this context, typically, integration processes are used in order to specify and execute complex integration tasks. These integration processes are executed by message-oriented integration platforms such as EAI servers (Enterprise Application Integration) or MOM systems (Message-Oriented Middleware). For two reasons, many independent instances of integration processes are executed over time. First, there is the requirement of immediate data synchronization between operational source systems in order to ensure data consistency. Second, data changes of the operational source systems are directly propagated into the data warehouse infrastructure in order to achieve high up-to-dateness of analytical query results (real-time ETL). Due to this high load of process instances, the performance of the central integration platform is crucial. Thus, optimization is required.

In the context of integration platforms, the major optimization objective is throughput maximization (Lee et al., 2007) rather than the execution time minimization of single process instances. Thus, the goal is to maximize the number of messages processed per time period. Here, moderate latency times of single messages are acceptable (Cecchet et al., 2008).

When optimizing integration processes, the following problems have to be considered:

Problem 1. Expensive External System Access. *The time-expensive access of external systems is caused by network traffic and message transformations from external formats into internal structures. The fact that external systems are accessed with similar queries over time offers potential for optimization.*

Problem 2. Cache Coherency Problem. *One solution to Problem 1 might be the caching of results of external queries. However, this fails, because when integrating highly distributed systems and applications, the central integration platform cannot ensure that the cached data is consistent with the data in the source systems (Lee et al., 2007).*

Problem 3. Serialized External Behavior. *In dependence on the external systems (e.g., caused by referential integrity constraints), we need to ensure the serial order of messages. However, internal out-of-order processing is possible.*

Given these problems, throughput maximization

Boehm M., Habich D. and Lehner W. (2010).
MULTI-PROCESS OPTIMIZATION VIA HORIZONTAL MESSAGE QUEUE PARTITIONING.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Databases and Information Systems Integration*, pages 5-14
DOI: 10.5220/0002862600050014

of integration processes has so far only been addressed by a higher degree of parallelism (Li and Zhan, 2005; Srivastava et al., 2006) or pipelining (Biornstad et al., 2006; Boehm et al., 2009). Although this can significantly increase the resource utilization and thus, optimize the throughput, it does not reduce the work to be executed by the integration platform.

In this paper, we introduce the concept of *Multi-Process Optimization (MPO)* in order to maximize the message throughput. Therefore, we periodically collect incoming messages and execute the whole message batch with one single process instance. The novel idea is to use horizontal (value-based) message queue partitioning as a batch creation strategy and to compute the optimal waiting time. When using this approach, all messages of one batch (partition) exhibit the same attribute value with regard to a chosen partition attribute. Hence, the complexity of several operators is reduced. This yields throughput improvements because the relative processing costs per message decrease with increasing batch size. In detail, we make the following contributions:

- Section 2 presents a typical architecture and execution characteristics of an integration platform.
- In Section 3, we give a solution overview of MPO that leads to the horizontal partitioning approach.
- Then, in Section 4, we define the MPO problem. Here, we also explain the cost estimation and the computation of the optimal waiting time.
- In Section 5, we introduce the concept of a partition tree. We discuss the derivation of partitioning attributes and the rewriting of process plans.
- Afterwards, we illustrate the results of our exhaustive experimental evaluation in Section 6.

Finally, we analyze related work in Section 7 and conclude the paper in Section 8.

2 SYSTEM ARCHITECTURE

A typical integration platform system architecture consists of a set of inbound adapters, multiple message queues, an internal scheduler, a central process execution engine, and a set of outbound adapters. The inbound adapters passively listen for incoming messages, transform them into a common format (e.g., XML) and append the messages to message queues or directly forward them to the process engine. Within the process engine, compiled process plans of deployed integration processes are executed. While executing those processes, the outbound adapters are used as services/gateways in order to actively invoke

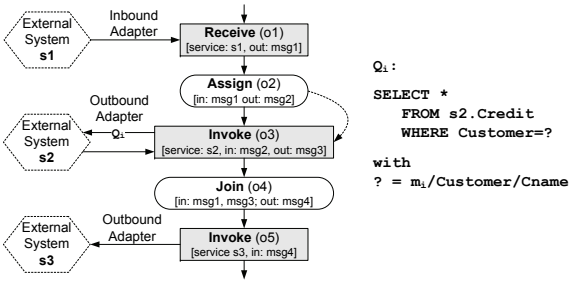


Figure 1: Running Example Process Plan *P*.

external systems. Therefore, they transform the internal format back into the proprietary message representations. This architecture is similar to the architecture of major products such as SAP Process Integration, IBM Message Broker or MS Biztalk Server.

The following example explains the instance-based (step-by-step) process execution, where message queues are used at the inbound server side only.

Example 1. Orders Processing: *Assume a process plan P that consists of an operator sequence o with* $o_i \in (o_1, \ldots, o_5)$ *(Figure 1). In the instance-based case, a new process instance* p_i *is created for each message (Figure 2). The* `Receive` *operator* (o_1) *gets an* orders *message from the queue and writes it to a local variable. Then, the* `Assign` *operator* (o_2) *is used in order to prepare a query with the* customer name *of the received message as a parameter. Subsequently, the* `Invoke` *operator* (o_3) *queries an external system* s_2 *in order to load additional customer data. Here, one SQL query* Q_i *per process instance (per message) is used. The* `Join` *operator* (o_4) *merges the result message with the received message. A final* `Invoke` *operator* (o_5) *sends the join result to system* s_3*. We see that multiple orders from one customer (*`CustA`*:* $m_1 \rightarrow p_1$, $m_3 \rightarrow p_3$*) cause us to pose the same query* (o_3) *multiple times to external system* s_2*.*

At this point, multi-process optimization comes into play, where we optimize the whole sequence of asynchronous process instances.

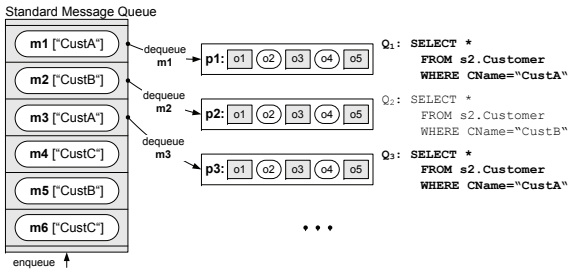


Figure 2: Instance-Based Process Plan Execution of *P*.

3 MULTI-PROCESS OPTIMIZATION

The naïve (time-based) batching approach, as already proposed for distributed queries (Lee et al., 2007), is to collect messages during a waiting time T_W and merge those messages to message batches b_i. Then, we execute a process instance p'_i of the modified process plan P' for the message batch b_i. Due to the simple (time-based) model of collecting messages, there might be multiple distinct messages in the batch according to the attributes used by the operators of P'. It follows that we need to rewrite the queries to external systems. We cannot precisely estimate this influence due to a lack of knowledge about data properties of external systems (Ives et al., 2004). In conclusion, the naïve approach can also hurt performance.

To tackle this problem, we propose a novel concept—the horizontal message queue partitioning approach—that we use in the rest of the paper.

The basic idea is to horizontally partition the inbound message queues according to partitioning attributes ba_i. With such value-based partitioning, all messages of a batch exhibit the same attribute value according to the partitioning attribute. Thus, certain operators of the process plan only need to access this attribute once for the whole partition rather than for each individual message. The core steps are (1) to derive the partitioning attribute from the process specification, (2) to periodically collect messages during a waiting time T_W, (3) to read the first partition from the queue, and (4) to execute the messages of this partition as a batch with an instance p'_i of a modified process plan P'. Additionally, (5) we might need to ensure the serial order of messages at the outbound side.

Example 2. Partitioned Batch-Orders Processing: *Figure 3 reconsiders the running example for partitioned multi-process execution. The incoming messages m_i are partitioned according to the partitioning attribute* customer name *that was extracted with $ba = m_i/Customer/Cname$ at the inbound side. A process instance of the rewritten process plan P' reads the first partition from the queue and executes the single operators for this partition. Due to the equal values of the partitioning attribute, we do not need to rewrite the query to the external system s_2. Every batch contains exactly one distinct attribute value according to ba. In total, we achieve performance benefits for the* `Assign` *as well as the* `Invoke` *operators. Thus, the throughput is improved and does not depend on the number of distinct messages. Note that the incoming order of messages was changed and needs to be serialized at the outbound side.*

The horizontal partitioning has another nice property: Several operators (e.g., `Assign`, `Invoke`, and `Switch`) benefit from partitioned message batch execution. There, partitioning attributes are derived from the process plan specification (e.g., query predicates and switch expressions). The benefit is caused by executing operations on partitions rather than on individual messages, and therefore, is similar to pre-aggregation (Ives et al., 2004) or early-group-by (Chaudhuri and Shim, 1994) in DBMS.

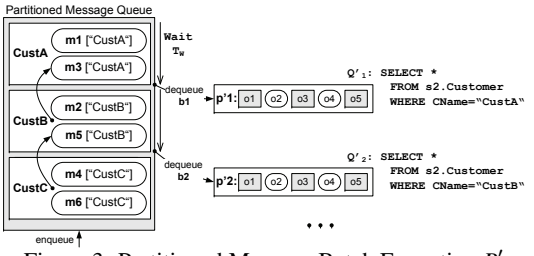


Figure 3: Partitioned Message Batch Execution P'.

Clearly, MQO (Multi-Query Optimization) and OOP (Out-of-Order Processing) (Li et al., 2008) have already been investigated in the context of DBMS and DSMS. However, in contrast to existing work, we present a novel MPO approach that maximizes the throughput by computing the optimal waiting time. Furthermore, this approach is dedicated to the context of integration processes, where such an execution model has been considered for the very first time.

MPO is also related to caching and the recycling of intermediate results (Ivanova et al., 2009). While caching might lead to using outdated data, the partitioned execution might cause us to use data that is more current than it was when the message arrived. However, we cannot ensure strong consistency by using an asynchronous integration technique (message queues). Further, we guarantee that (1) the temporal gap is at most equal to a given latency constraint and that (2) no outdated data is read. In conclusion, caching is advantageous if data of external sources is static, while MPO is beneficial if data of external sources changes dynamically.

Finally, the question arises of how likely it is that we can benefit from MPO. With regard to the experimental evaluation, there are three facts why we benefit from MPO. First, even for 1-message partitions, there is only a moderate runtime overhead. Second, throughput optimization is required if and only if high message load (peaks) exists. In such cases, it is very likely that messages with equal attribute values are in the queue. Third, only a small number of messages is required within one partition to yield a significant speedup for different types of operators.

The major research challenges of MPO via horizontal partitioning are (1) to compute the optimal waiting time and (2) to enable partitioned process ex-

ecution. Both are addressed in the following sections.

4 WAITING TIME COMPUTATION

Based on a formal problem description, we describe how to automatically compute the optimal waiting time w.r.t. maximizing the throughput.

Let M with $m_i \in (m_1, m_2, \ldots)$ be an infinite and ordered stream of messages. We model each message m_i as a (t_i, d_i)-tuple, where $t_i \in \mathbb{N}$ denotes the incoming timestamp of the message and d_i denotes a semi-structured tree of name-value data elements. Each message m_i is processed by an instance p_i of a process plan P, and $t_{out}(m_i) \in \mathbb{N}$ denotes the timestamp when the message has been successfully processed. The latency of a single message $T_L(m_i)$ is given by $T_L(m_i) = t_{out}(m_i) - t_i$. This includes waiting time as well as processing time. Then, the total latency time of a finite message subsequence M' with $M' \subseteq M$ is determined by $T_L(M') = t_{out}(m_{|M'|}) - t_1$.

Definition 1. Multi-Process Optimization Problem (MPO-P): *Maximize the message throughput with regard to a finite message subsequence M'. The optimization objective ϕ is to minimize the total latency time:*

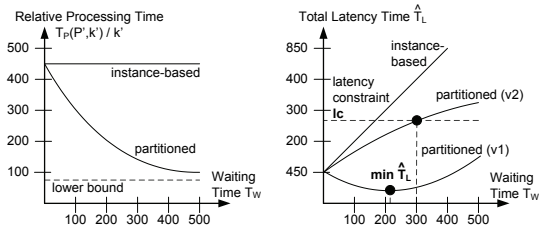
$$\phi = \max \frac{|M'|}{\Delta t} = \min T_L(M'). \quad (1)$$

There, two additional restrictions must hold:

1. *Let lc denote a soft latency constraint that must not be exceeded significantly. Then, the condition $\forall m_i \in M' : T_L(m_i) \leq lc$ must hold.*
2. *The external behavior must be serialized according to the incoming message order, where $\forall m_i \in M' : t_{out}(m_i) \leq t_{out}(m_{i+1})$ must hold.*

In order to solve the MPO-P, we horizontally partition the message queue and periodically compute the optimal waiting time T_W with regard to the current execution statistics. Then, an instance p'_i of a partitioned process plan P' is issued with a period of T_W in order to execute the message partition b_i. In order to avoid temporally overlapping process plan executions and inconsistency between the latency constraint lc and the processing time T_P, we define the *validity condition*: For a given latency constraint lc, there must exist a waiting time T_W such that $(0 \leq T_P \leq T_W) \wedge (0 \leq \hat{T}_L \leq lc)$; otherwise, the constraint is invalid.

Minimizing the total latency requires the cost estimation of a partitioned process plan for specific batch sizes k' with $k' = |b_i|$. First, we monitor the incoming message rate R and the value selectivity $sel \in \mathbb{R}$ with $0 < sel \leq 1$ according to the partitioning attributes. Assuming a uniform distribution function $\mathcal{D}$ of R, the first partition will contain $k' = R \cdot sel \cdot T_W$ messages. For the i-th partition with $i \geq \frac{1}{sel}$, k' is computed by $k' = R \cdot T_W$, independently of the selectivity sel. Second, for reading `Invoke`, `Assign` and `Switch` operators, the costs are independent of k' with $C(o'_i, k') = C(o'_i, 1)$, while for all other operators, costs increase linearly with $C(o'_i, k') = C(o'_i, 1) \cdot k'$. For each operator, a specific cost function is defined based on monitored cardinalities. Then, the costs of a process plan are defined as $C(P', k') = \sum_{i=1}^{m'} C(o'_i, k')$.



(a) $T_W \rightarrow T_P$ Influence (b) $T_W \rightarrow T_L$ Influence

Figure 4: Waiting Time Search Space.

The intuition of our method for computing the optimal waiting time T_W is that the waiting time—and hence, the batch size k'—strongly influences the processing time of single process instances. Then, the latency time mainly depends on that processing time. Figure 4 conceptually illustrates the resulting two inverse influences that our computation algorithm exploits: First, an increasing waiting time T_W causes a decreasing relative processing time T_P/k' for partitioned process execution (Figure 4(a)). Second, an increasing waiting time T_W linearly increases the latency time $\hat{T}_L$ because the waiting time is directly included in $\hat{T}_L$ (Figure 4(b)). The result of these two influences is a non-linear total latency time function that might have a minimum (v1) or not (v2). Given the latency constraint, we can compute the optimal waiting time with regard to latency time minimization and hence, throughput maximization.

In detail, we can compute the waiting time where $\hat{T}_L$ is minimal or where it is equal to lc by

$$T_W = \begin{cases} T_W \text{ with } \min \hat{T}_L(T_W) & 0 \leq \hat{T}_L \leq lc \\ T_W \text{ with } \hat{T}_L(T_W) = lc & otherwise. \end{cases} \quad (2)$$

The estimated total latency time $\hat{T}_L$ is computed by

$$\hat{T}_L = \left\lceil \frac{|M'|}{k'} \right\rceil \cdot T_W + T_P(P', k') \quad \text{with}$$
$$T_P(P', k') = T_P(P) \cdot \frac{C(P', k')}{C(P)} = \sum_{i=1}^{m} T_P(o_i) \cdot \frac{C(o'_i, k')}{C(o_i)}, \quad (3)$$

where $\lceil |M'|/k' \rceil$ denotes the total number of executed partitions. Furthermore, we can substitute k' with $R \cdot T_W$ within T_P and get

$$T_P(P', k') = T_P(P', R \cdot T_W) = \sum_{i=1}^{m} T_P(o_i) \cdot \frac{C(o'_i, R \cdot T_W)}{C(o_i)}. \quad (4)$$

Then, in order to solve the MPO-P, we compute T_W where $\hat{T}_L'(T_W) = 0$ and $\hat{T}_L''(T_W) > 0$. Finally, we check the validity condition and modify the waiting time T_W if required. It can be shown for arbitrary distribution functions $\mathcal{D}$ that the latency time constraint holds.

5 PARTITIONED EXECUTION

In order to enable partitioned process execution, in this section, we introduce the *partition tree* and the related algorithms. The *partition tree* is a multi-dimensional B*-Tree (MDB-Tree) (Scheuermann and Ouksel, 1982), where the messages are horizontally partitioned according to multiple partitioning attributes. Similar to a traditional MDB-Tree, each tree level represents a different partition attribute.

Definition 2. Partition Tree*: The partition tree is an index of h levels, where each level represents a partition attribute ba_i with $ba_i \in (ba_1, ba_2, \ldots, ba_h)$. For each attribute ba_i, a set of batches (partitions) b are maintained. Those partitions are ordered according to their timestamps of creation $t_c(b_i)$ with $t_c(b_{i-1}) \leq t_c(b_i) \leq t_c(b_{i+1})$. Only the last index level ba_h contains the single messages. A partition attribute has a type$(ba_i) \in \{$`value`, `value-list`, `range`$\}$.*

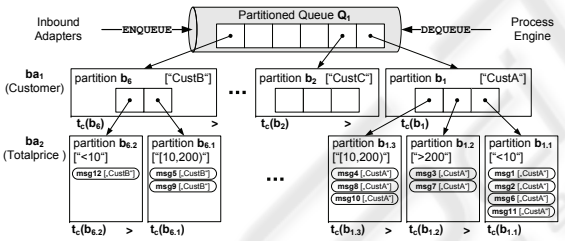


Figure 5: Example Queue Partition Tree ($h = 2$).

Example 3. Partition Tree with $h = 2$: *Assume two partitioning attributes ba_1 (customer, `value`) and ba_2 (total price, `range`) from a process plan P. Then, the partitioned tree exhibits a height of $h = 2$ (see Figure 5). On the first index level, the messages are partitioned according to customer names, and on the second level, each partition is divided according to the range of order total prices.*

There are two essential maintenance procedures of the partition tree: `enqueue` ENQ() and `dequeue` DEQ(). ENQ() is invoked by the inbound adapters for each incoming message, while DEQ() is invoked by the process engine periodically, according to the computed optimal waiting time T_W. The ENQ() function scans over the partitions and determines whether or not a partition with $ba(b_i) = ba(m_i)$ already exists. If so, the message is inserted recursively; otherwise, a new partition is created and added at the last position. The DEQ() function returns the first partition $(\min_{i=1}^{|b|} t_c(b_i))$ of the partition tree.

The partitioning attributes are automatically derived from the single operators $o_i \in P$ that benefit from partitioning. The final partitioning scheme is then created by minimizing the expected number of partitions in the index. Therefore, we order the index attributes according to their selectivities with

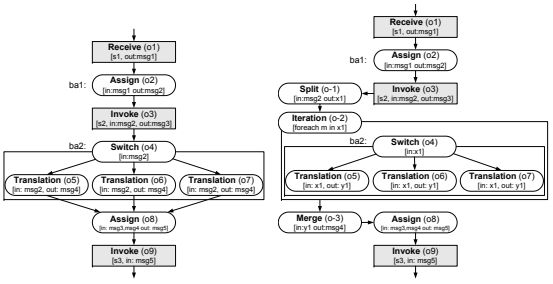
$$\min \sum_{i=1}^{h} |b_j \in ba_i| \text{ with } sel(ba_1) \geq sel(ba_i) \geq sel(ba_h). \quad (5)$$

Thus, we minimized the overhead of queue maintenance and maximized k' of the top-level partitions. The result is the optimal partitioning scheme.

Subsequently, we rewrite P to P' in order to enable partitioned process execution according to this partitioning scheme. Therefore, we use a *split and merge* approach: A process plan receives the top-level partition and executes all operators that benefit from the top-level attribute. Right before an operator that benefits from the next-level partition attribute, we insert a `Split` operator that splits the top-level partition into the $\frac{1}{sel(ba_2)}$ subpartitions (worst case) as well as an `Iteration` operator (foreach). The iteration body is the sequence of operators that benefit from this granularity. Right after this iteration, we insert a `Merge` operator to re-group the resulting partitions if required.

Example 4. Rewriting a Process Plan: *Assume the process plan P shown in Figure 6(a). We receive a message from system s_1, create a parameterized query, and request system s_2. Afterwards, we use an alternative switch path, and finally, we send the result to system s_3. According to Example 3, we have derived the two partitioning attributes ba_1 (customer, `value`) and ba_2 (total price, `range`). If we use the partitioning scheme (ba_1, ba_2), the split and merge approach is applied as shown in Figure 6(b).*

According to the requirement of serialized external behavior, we might need to serialize messages at the outbound side. Therefore, we extended the message structure by a counter c. If a message m_i out-



(a) Original Process Plan P (b) Process Plan P'

Figure 6: Example Rewriting of Process Plans.

runs another message during ENQ() partitioning, its counter $c(m_i)$ is increased by one. Serialization is realized by timestamp comparison, and for each reordered message, the counter is decreased by one. Thus, at the outbound side, we are not allowed to send message m_i until $c(m_i) = 0$. It can be shown that the soft maximum latency constraint is still guaranteed.

6 EXPERIMENTAL EVALUATION

We provide selected results of our exhaustive experimental evaluation. In general, the evaluation shows that (1) significant throughput optimization is reachable and that (2) the maximum latency guarantees hold under experimental investigation.

We implemented the approach of MPO via horizontal partitioning within our java-based workflow process engine (WFPE). This includes the partition tree, slightly changed operators (partition-awareness) and the algorithms for deriving partitioning attributes (DPA), the rewriting of process plans (RPP), and the automatic waiting time computation (WTC) as well as the overall system integration in the sense of an environment for periodical re-optimization.

Subsequently, we ran our experiments on a blade (OS Suse Linux, 32bit) with two processors (each of them a Dual Core AMD Opteron Processor 270 at 1,994 MHz) and 8.9 GB RAM. With regard to repeatability, we used synthetically generated datasets.

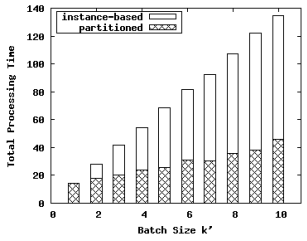
As base integration process, we used our running example ($m = 5$). To scale the number of operators m, we copied those operators and changed the operator configurations slightly. The other scaling factors were set to the following standard parameters: number of messages $|M'| = 100$, message rate $R = 0.005\frac{msg}{ms}$, selectivity according to the partitioning attribute $sel = 0.1$, batch size $k' = 5$, message rate distribution function $\mathcal{D} = uniform$, latency constraint $lc = 10s$, maximum queue size $qmax = 1,000$, and input data size $d = 1$ ($6kb$ messages). Finally, all experiments were repeated 20 times.

First, we investigated the processing time T_P of partitioned execution. Figure 7(a) shows the absolute processing time of k' messages according to the batch size k'. Instance-based execution means that we executed one process instance for each message of the batch as a baseline. Thus, the total processing time linearly increases with increasing batch size. In contrast, we only executed one process instance for the complete batch when using MPO. Here, we can observe that the total processing time increases logarithmically with increasing batch size. Then, Figure 7(d) shows the relative processing time of both execution modes. For partitioned execution, we observe that the relative processing time tends to the lower bound (fraction of costs that linearly depends on the batch size). The total message throughput directly depends on this relative processing time (Little's Law (Little, 1961)). For the used process plan, we observe that partitioned execution improves the maximum throughput by a factor of three. Furthermore, we fixed $k' = 10$ and varied the number of operators m as well as the data size d. Figures 7(b) and 7(e) illustrate the results of these scalability experiments, where the relative improvement stays almost constant when scaling both parameters.

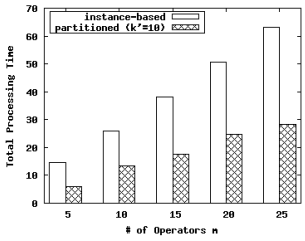
Second, we evaluated the batch size k' according to different message rates R (in $\frac{msg}{ms}$), selectivities sel, and waiting times T_W. We executed $|M'| = 100$ messages and fixed a waiting time of $T_W = 10s$. Figure 7(c) shows the influence of the message rate R on the maximum number of messages in the batch. We can observe (1) that the higher the message rate, the higher the number of messages in the batch, and (2) that the selectivity determines the reachable upper bound. However, the influence of the message rate is independent of the selectivity (see Section 4). Figure 7(f) illustrates the influence of T_W on k', where we fixed $sel = 1.0$. Note that both an increasing waiting time as well as an increasing message rate increases the batch size.

Third, we evaluated the latency influence of partitioned process execution with regard to the maximum latency guarantee. We executed $|M'| = 1,000$ messages with a maximum latency constraint of $lc = 10s$ and measured the latency time $T_L(m_i)$ of single messages m_i. For both $\mathcal{D} = uniform$ (see Figure 8(a)) and $\mathcal{D} =$ `poisson` (see Figure 8(b))—this is typical for arrival processes of workflow instances (Xiao et al., 2006)—the constraint is not significantly exceeded. However, in the latter case, peaks over the latency constraint lc are possible. The constraint also holds for serialized external behavior (SEB), where all messages show more similar latency times (see Figure 8(c), where $\mathcal{D} = uniform$). This is due to serialization at the outbound side. Thus, there is a lower variance of single message latencies. Note that the latency constraint is explicitly a soft constraint, where we guarantee that it is not exceeded significantly. The reason for this is that we compute the waiting time based on our cost estimation. If the real execution costs vary slightly around this estimate, there exist cases where the constraint is slightly exceeded. Thus, a hard latency constraint is impossible.

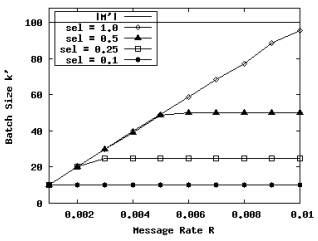
Fourth, we evaluated the algorithm overhead required for horizontally partitioned message execution. The runtime overhead—that includes the wait-



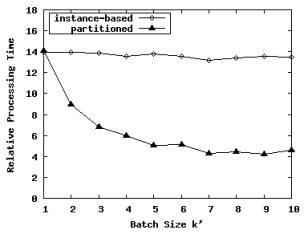
(a) Absolute $T_P(k')$ [in ms]



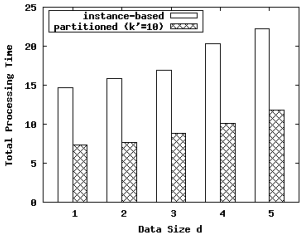
(b) Absolute $T_P(m)$ [in ms]



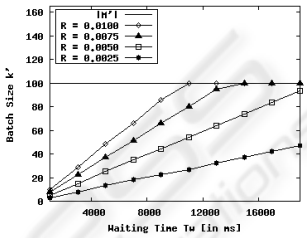
(c) Batch Size $k'(R)$



(d) Relative $\frac{T_P}{k'}(k')$ [in ms]

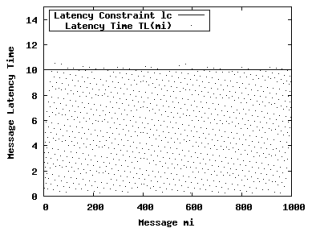


(e) Absolute $T_P(d)$ [in ms]

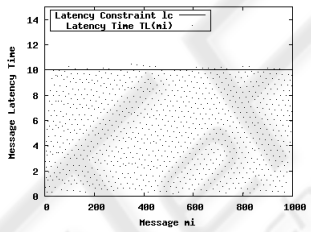


(f) Batch Size $k'(T_W)$

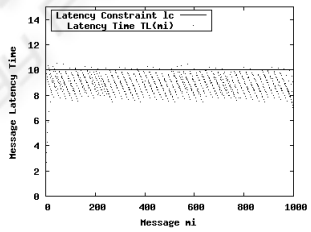
Figure 7: Performance Benefit.



(a) Latency $T_L(uniform)$

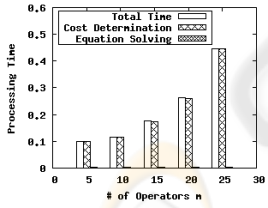


(b) Latency $T_L(poisson)$

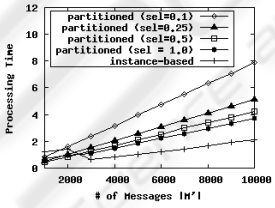


(c) Latency T_L with SEB

Figure 8: Latency of Single Messages [in s].



(a) Computing T_W



(b) Enqueue M'

Figure 9: Algorithm Runtime Overhead [in ms].

ing time computation (WTC, Figure 9(a)) and partitioned queue maintenance (Figure 9(b))—is moderate. Although WTC has a super-linear time complexity, it took less than a millisecond for processes with up to 25 operators. Furthermore, the partitioned enqueue operation clearly depends on the selectivity. The lower the selectivity, the higher the overhead. Anyway, the overhead to enqueue 10,000 messages (even for a selectivity of $sel = 0.1$) was only six milliseconds compared to the standard message queue. The deploy time overhead of partitioning includes the derivation of partitioning attributes (DPA) and the rewriting of process plans (RPP). This overhead is dominated by process plan parsing and unparsing. However, those efforts are required anyway when analyzing and optimizing process plans and they are required only once during deploy time.

Finally, we can state that MPO achieves significant throughout improvement. How much we benefit from MPO depends on the concrete workload. The benefit is caused by two facts. First, even for 1-message partitions, there is only a moderate runtime overhead (Figures 9(a) and 9(b)). Second, only a small number of messages is required within one partition to yield a significant speedup (Figure 7(d)).

7 RELATED WORK

Multi-Query Optimization. The basic concepts of Multi-Query Optimization (MQO) (Roy et al., 2000) are *pipelined query execution* and *data sharing across queries*. Here, a huge body of work exists for lo-

cal environments (Candea et al., 2009; Harizopoulos et al., 2005; Johnson et al., 2007) as well as for distributed query processing (Ives et al., 2004; Kementsietsidis et al., 2008; Lee et al., 2007; Unterbrunner et al., 2009). For example, Lee et al. employed the waiting opportunities within a blocking query execution plan (Lee et al., 2007). Further, Qiao et al. investigated a batch-sharing partitioning scheme (Qiao et al., 2008) in order to allow similar queries to share cache contents. The main difference between MPO and MQO is that MQO benefits from the reuse of results across queries, while for MPO, this is impossible due to disjoint incoming messages. Further, MPO handles dynamic data propagations and benefits from redundant work and acceptable latency time. In addition, MPO computes the optimal waiting time.

Data Partitioning. Horizontal data (value-based) partitioning (Ceri et al., 1982) is strongly applied in DBMS. Typically, this is an issue of physical design (Agrawal et al., 2004). However, there are more recent approaches such as the table partitioning along foreign-key constraints (Eadon et al., 2008). Furthermore, there are interesting approaches where data partitioning is used for distributed tables, such as Yahoo! PNUTS (Silberstein et al., 2008) or Google BigTable (Chang et al., 2006). In the area of data streams, data partitioning was used in the sense of plan partitioning across server nodes (Johnson et al., 2008) or single filter evaluation on tuple granularity (Avnur and Hellerstein, 2000). Finally, there are similarities between our horizontal partitioning approach and partitioning in the area of parallel DBMS. The major difference is that MPO handles infinite streams of messages.

Workflow Optimization. Though there is not much work on optimizing integration processes, there is a data-centric but rule-based approach to optimize BPEL processes (Vrhovnik et al., 2007). In contrast, we already proposed a cost-based optimization approach (Boehm et al., 2008). Anyway, it focuses on execution time minimization rather than on throughput maximization. Furthermore, there are existing approaches (Biornstad et al., 2006; Boehm et al., 2009; Li and Zhan, 2005; Srivastava et al., 2006) that also address the throughput optimization. However, those approaches try to increase the degree of parallelism, while our approach reduces executed work across multiple instances of a process plan.

8 CONCLUSIONS

To summarize, we proposed a novel approach for throughput maximization of integration processes that reduces work by employing horizontal data partitioning. Our exhaustive evaluation showed that significant performance improvements are possible and that theoretical guarantees of optimality and latency also hold under experimental investigation. In conclusion, the MPO approach can seamlessly be applied in a variety of different integration platforms that execute asynchronous integration processes.

Further, the general MPO approach opens many opportunities for further optimizations. Future work might consider (1) the execution of partitions independent of their temporal order, (2) process plan partitioning in the sense of compiling different plans for different partitions, (3) global MPO for multiple process plans, and (4) the cost-based process plan rewriting problem. Finally, it may be interesting (5) to combine MPO with pipelining and load balancing because both address throughput maximization as well.

REFERENCES

Agrawal, S., Narasayya, V. R., and Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*.

Avnur, R. and Hellerstein, J. M. (2000). Eddies: Continuously adaptive query processing. In *SIGMOD*.

Biornstad, B., Pautasso, C., and Alonso, G. (2006). Control the flow: How to safely compose streaming services into business processes. In *SCC*.

Boehm, M., Habich, D., Preissler, S., Lehner, W., and Wloka, U. (2009). Cost-based vectorization of instance-based integration processes. In *ADBIS*.

Boehm, M., Wloka, U., Habich, D., and Lehner, W. (2008). Workload-based optimization of integration processes. In *CIKM*.

Candea, G., Polyzotis, N., and Vingralek, R. (2009). A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1).

Cecchet, E., Candea, G., and Ailamaki, A. (2008). Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*.

Ceri, S., Negri, M., and Pelagatti, G. (1982). Horizontal data partitioning in database design. In *SIGMOD*.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. (2006). Bigtable: A distributed storage system for structured data. In *OSDI*.

Chaudhuri, S. and Shim, K. (1994). Including group-by in query optimization. In *VLDB*.

Eadon, G., Chong, E. I., Shankar, S., Raghavan, A., Srinivasan, J., and Das, S. (2008). Supporting table partitioning by reference in oracle. In *SIGMOD*.

Harizopoulos, S., Shkapenyuk, V., and Ailamaki, A. (2005). Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*.

Ivanova, M., Kersten, M. L., Nes, N. J., and Goncalves, R. (2009). An architecture for recycling intermediates in a column-store. In *SIGMOD*.

Ives, Z. G., Halevy, A. Y., and Weld, D. S. (2004). Adapting to source properties in processing data integration queries. In *SIGMOD*.

Johnson, R., Hardavellas, N., Pandis, I., Mancheril, N., Harizopoulos, S., Sabirli, K., Ailamaki, A., and Falsafi, B. (2007). To share or not to share? In *VLDB*.

Johnson, T., Muthukrishnan, S. M., Shkapenyuk, V., and Spatscheck, O. (2008). Query-aware partitioning for monitoring massive network data streams. In *SIGMOD*.

Kementsietsidis, A., Neven, F., de Craen, D. V., and Vansummeren, S. (2008). Scalable multi-query optimization for exploratory queries over federated scientific databases. In *VLDB*.

Lee, R., Zhou, M., and Liao, H. (2007). Request window: an approach to improve throughput of rdbms-based data integration system by utilizing data sharing across concurrent distributed queries. In *VLDB*.

Li, H. and Zhan, D. (2005). Workflow timed critical path optimization. *Nature and Science*, 3(2).

Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., and Maier, D. (2008). Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1).

Little, J. D. C. (1961). A proof for the queueing formula: $l = \lambda w$. *Operations Research*, 9.

Qiao, L., Raman, V., Reiss, F., Haas, P. J., and Lohman, G. M. (2008). Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1).

Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. (2000). Efficient and extensible algorithms for multi query optimization. In *SIGMOD*.

Scheuermann, P. and Ouksel, A. M. (1982). Multidimensional b-trees for associative searching in database systems. *Inf. Syst.*, 7(2).

Silberstein, A., Cooper, B. F., Srivastava, U., Vee, E., Yerneni, R., and Ramakrishnan, R. (2008). Efficient bulk insertion into a distributed ordered table. In *SIGMOD*.

Srivastava, U., Munagala, K., Widom, J., and Motwani, R. (2006). Query optimization over web services. In *VLDB*.

Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., and Kossmann, D. (2009). Predictable performance for unpredictable workloads. *PVLDB*, 2(1).

Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., and Kraft, T. (2007). An approach to optimize data processing in business processes. In *VLDB*.

Xiao, Z., Chang, H., and Yi, Y. (2006). Optimal allocation of workflow resources with cost constraint. In *CSCWD*.

APPENDIX

A FORMAL ANALYSIS

We additionally provide formal analysis results with regard to the waiting time computation.

A.1 Optimality

First of all, we give an optimality guarantee for $T_P(P',k')$ based on the computed waiting time.

Theorem 1. Optimality of Partitioned Execution: *The horizontal message queue partitioning solves the MPO-P with optimality guarantees of* $T_P(P',k') \cdot \frac{|M'|}{k'} \leq T_P(P',k'-1) \cdot \frac{|M'|}{k'-1} \leq T_P(P,1) \cdot k'$, *where* $k' > 1$.

Proof. The processing time $T_P(P',k')$ is computed by

$$T_P(P',k') = T_P(P) \cdot \frac{C(P',k')}{C(P)}.$$

Further, the costs of a process plan $C(P',k')$ are composed of the costs $C^+(P',k')$ that linearly depend on k' and costs $C^-(P',k')$ that sub-linearly depend on k', with $C(P',k') = C^+(P',k') + C^-(P',k')$. In conclusion, in the worst case, the processing time $T_P(P',k')$ increases linearly when increasing k'. Thus, the relative processing time $\frac{T_P(P',k')}{k'}$ is a *monotonically non-increasing* function with

$$k',k'' \in [1,|M'|] : k' < k'' \Rightarrow \frac{T_P(P',k')}{k'} \geq \frac{T_P(P',k'')}{k''}.$$

If we now fix a certain $|M'|$, it follows directly that

$$T_P(P',k') \cdot \frac{|M'|}{k'} \leq T_P(P',k'-1) \cdot \frac{|M'|}{k'-1} \leq T_P(P,1) \cdot k'.$$

Hence, Theorem 1 holds. □

A.2 Latency Constraint

Furthermore, we guarantee to preserve the given maximum latency constraint for individual messages.

Theorem 2. Soft Guarantee of Maximum Latency: *The waiting time computation ensures that—for a given message rate R, with* $\mathcal{D} = uniform$*—the latency time of a single message* $T_L(m_i)$ *with* $m_i \in M'$ *will not significantly exceed the maximum latency constraint lc with* $T_L(m_i) \leq lc$.

Proof. In the worst case, $\frac{1}{sel}$ distinct messages m_i arrive simultaneously in the system. Hence, the highest possible latency time $T_L(m_i)$ is given by $\frac{1}{sel} \cdot T_W + T_P(P',k')$. Due to our validity condition of $\hat{T}_L \leq lc$, we need to show that $T_L(m_i) \leq \hat{T}_L$ even for this worst case. Further, our *validity condition* ensures that $T_W \geq T_P(P',k')$. Thus, we can write $T_L(m_i) \leq \hat{T}_L(T_W,R)$ as

$$\frac{1}{sel}\cdot T_W + T_P(P',k') \leq \left\lceil \frac{|M'|}{k'} \right\rceil \cdot T_W + T_P(P',k')$$

$$\frac{1}{sel}\cdot T_W \leq \frac{|M'|}{k'}\cdot T_W.$$

We substitute T_W with $\frac{k'}{R}$ and subsequently substitute $|M'|$ by $\frac{k'}{sel}$ (the cardinality $|M'|$ is equal to the number of partitions $\frac{1}{sel}$ times the cardinality of a partition k'), and we get

$$\frac{k'}{R\cdot sel} \leq \frac{|M'|}{R} = \frac{k'}{R\cdot sel}.$$

Thus, for the worst case, $T_L(m_i) = lc$ (more specific, $T_L(m_{|M'|}) = lc$), while for all other cases, $T_L(m_i) \leq lc$ is true. Hence, Theorem 2 holds. □

Note that by hypothesis testing, it can be shown that this guarantee of maximum latency also holds for arbitrary probability distributions of the message rate.

A.3 Lower Bound of Relative Costs

In analogy to *Amdahl's law*, where the fraction of a task (processing time) that cannot be executed in parallel determines the upper bound for the reachable speedup, we compute the lower bound of the relative processing costs. The existence of this lower bound was empirically shown in Section 6. Therefore, let $T_P(P',k')$ denote the absolute processing time using batches of k' messages. Let $C(P)$ denote the costs for $k' = 1$; $C^+(P)$ denotes the costs that linearly depend on k', and $C^-(P)$ denotes the costs that depend sub-linearly on k'. Here, the condition $C(P) = C^+(P) + C^-(P)$ holds. Finally, $\frac{T_P(P',k')}{k'}$ denotes the relative processing time at k'. This relative processing time asymptotically tends to a lower bound.

Theorem 3. *The lower bound of relative processing costs $\frac{T_P(P',k')}{k'}$ is given by $T_P(P)\cdot\frac{C^+(P)}{C(P)}$ as the fraction of costs that linearly depend on k' and of the instance-based costs.*

Proof. Recall that—according to Equation 3—the absolute processing time $T_P(P',k')$ is computed by

$$T_P(P',k') = T_P(P)\cdot\frac{C(P',k')}{C(P)} = T_P(P)\cdot\frac{C^+(P',k')+C^-(P',k')}{C(P)}.$$

Due to the linear dependency of $C^+(P',k')$ on k', we can now write $C^+(P',k') = C^+(P',1)\cdot k' = C^+(P)\cdot k'$. Further, $C^-(P',k')$ has a sub-linear dependency on k' by definition. If we now let k' tend to ∞ with

$$\frac{T_P(P',k')}{k'} = T_P(P)\cdot\frac{C^+(P)\cdot k'}{C(P)\cdot k'} + \frac{C^-(P',k')}{C(P)\cdot k'}$$

$$\lim_{k'\to\infty}\frac{T_P(P',k')}{k'} = T_P(P)\cdot\frac{C^+(P)}{C(P)},$$

we see that $\frac{T_P(P',k')}{k'}$ asymptotically tends to $T_P(P)\cdot\frac{C^+(P)}{C(P)}$. Hence, Theorem 3 holds. □

A.4 Serialized External Behavior

According to the requirement of serialized external behavior, we might need to serialize messages at the outbound side. Therefore, we extended the message structure by a counter c with $c \in \mathbb{N}$ to a (t_i, c_i, d_i)-tuple. If a message m_i outruns another message during ENQ() partitioning, its counter $c(m_i)$ is increased by one. The serialization is realized by timestamp comparison, and for each reordered message, the counter is decreased by one. Thus, at the outbound side, we are not allowed to send message m_i until its counter is $c(m_i) = 0$.

Theorem 4. Serialized Behavior: *The* Soft Guarantee of Maximum Latency *theorem also holds in the case that we have to preserve the serial order of external behavior.*

Proof. Basically, we need to prove that the condition $T_L(m_i) \leq \hat{T}_L \leq lc$ is true even if we have to serialize the external behavior. Therefore, recall the worst case (Theorem 2), where the latency time is given by

$$T_L(m_i) = \frac{1}{sel}\cdot T_W + T_P(P',k').$$

Here, the message m_i has not outrun any other messages. Thus, there is no serialization time required. For all other messages that exhibit a general latency time of

$$T_L(m_i) = \left(\frac{1}{sel} - x\right)\cdot T_W + T_P^*(P',k'),$$

where x denotes the number of partitions after the partition of m_i, this message has outrun at most $x\cdot k'$ messages and its partition is executed in $T_P^*(P',k')$. Thus, additional serialization time of $x\cdot T_W + T_P(P',k')$ is needed. In conclusion, we get

$$\begin{aligned} T_L(m_i) &= \left(\frac{1}{sel} - x\right)\cdot T_W + T_P^*(P',k') && \text{//normal latency} \\ &\quad + x\cdot T_W + T_P(P',k') && \text{//serialization} \\ &= \frac{1}{sel}\cdot T_W + T_P(P',k'). \end{aligned}$$

Thus, $T_L(m_i) \leq \hat{T}_L \leq lc$ is true for the serialized case as well because $T_P^*(P',k')$ is subsumed by $x\cdot T_W$ because the waiting time is longer than the processing time due to the validity condition of $T_W \geq T_P$. Hence, Theorem 4 holds. □

Counting messages that have been outrun also works for CN:CM multiplicities between input and output messages[1]. In fact, the proof works only for sequences of operators.

[1]Messages with counters not equal to zero are ousted by subsequent messages with higher timestamps, and the outbound queues are periodically flushed.