Matthias Boehm, Dirk Habich, Wolfgang Lehner

On-demand re-optimization of integration flows

Erstveröffentlichung in / First published in:
Information Systems. 2014. 45, S. 382-399. Elsevier. ISSN 0306-4379.
DOI: https://doi.org/10.1016/j.is.2008.06.001

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

On-demand re-optimization of integration flows

Matthias Boehm*, Dirk Habich, Wolfgang Lehner

Technische Universität Dresden, Dresden, Germany

A R T I C L E I N F O

Article history:
Received 10 June 2012
Received in revised form
27 September 2013
Accepted 25 March 2014
Recommended by: Philippe Bonnet
Available online 13 April 2014

Keywords:
Integration flows
Optimization
Workload adaptation

A B S T R A C T

Integration flows are used to propagate data between heterogeneous operational systems or to consolidate data into data warehouse infrastructures. In order to meet the increasing need of up to date information, many messages are exchanged over time. The efficiency of those integration flows is therefore crucial to handle the high load of messages and to reduce message latency. State of the art strategies to address this performance bottleneck are based on incremental statistic maintenance and periodic cost based re optimization. This also achieves adaptation to unknown statistics and changing workload characteristics, which is important since integration flows are deployed for long time horizons. However, the major drawbacks of periodic re optimization are many unnecessary re optimization steps and missed optimization opportunities due to adaptation delays. In this paper, we therefore propose the novel concept of on demand re optimization. We exploit optimality conditions from the optimizer in order to (1) monitor optimality of the current plan, and (2) trigger directed re optimization only if necessary. Furthermore, we introduce the PlanOptimalityTree as a compact representation of optimality conditions that enables efficient monitoring and exploitation of these conditions. As a result and in contrast to existing work, re optimization is immediately triggered but only if a new plan is certain to be found. Our experiments show that we achieve near optimal re optimization overhead and fast workload adaptation.

1. Introduction

Increasing amounts of data as well as technical and organizational issues, like new technologies and pragmatic behavior, fundamentally changed the scope of data management towards distributed data management across numerous heterogeneous systems, applications, and small devices [1]. For this reason, the seamless and efficient integration of these systems becomes more and more crucial for an IT infrastructure and it is seen as one of the most expensive challenges information technology faces today [2].

In order to cope with the high degree of system heterogeneity and complex procedural integration tasks, imperative integration flows are modeled and executed to exchange data between these systems [3]. There are many important application domains such as enterprise information systems, financial messaging, energy data management, telecommunications, or health care management [4]. Integration flows are deployed once and then repeatedly executed by an integration platform, often over months and years. Examples of such platforms are ETL (Extraction Transformation Loading) tools, EAI (Enterprise Application Integration) servers or MOM (Message Oriented Middleware) systems, which have converged more and more in the past [3,5].

In general, there are two major types of use cases for integration flows. First, *horizontal integration* refers to data synchronization between operational systems by EAI or MOM tools. Data exchange is triggered on updates and

* Corresponding author. IBM Research – Almaden, San Jose, CA, USA. Work done while the author was at Technische Universität Dresden.
E-mail address: mboehm@us.ibm.com (M. Boehm).

http://dx.doi.org/10.1016/j.is.2014.03.005

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

hence realized by propagating many small messages [6]. Second, *vertical integration* refers to consolidating data of operational systems into data warehouse infrastructures by ETL tools. There is an increasing demand of operational business intelligence that also requires immediate data synchronization in order to achieve up to date operational reports [7 9]. This immediate synchronization is realized with techniques like increased delta load frequencies or even update driven data propagation (trickle feeds) [7,10].

Both types of use cases lead to the execution of many independent plan instances of integration flows over time. Additionally, often low message latency is required due to the need for up to date information and potentially block ing source systems. The high load combined with low latency requirements inherently leads to a need for opti mization to overcome the performance bottleneck. Exist ing work on optimizing integration flows can be classified as follows into *rule based*, *cost based*, and *adaptive cost based* approaches:

Rule based optimization applies static rewrite rules (e.g., algebraic simplifications) during initial deployment of an integration flow (optimize once) [11 14]. While this approach imposes low optimization overhead, many optimization opportunities cannot be exploited because rewriting often requires dynamic cost based decisions.

Cost based optimization exploits the full optimization potential by applying dynamic rewritings based on estimated costs [10,13,15,16]. Existing approaches are still only executed once during initial plan deployment (optimize once). These works are important steps towards adaptive behavior but they cannot adapt the deployed integration flows to changing workload characteristics [17 19] such as varying cardinal ities, selectivities, or execution times.

Reasons for workload changes are manifold such as varying usage schemes of external systems [20] or varying network properties [18]. For example, consider a web shop, where new orders are immediately propagated to a central system. Time varying numbers of orders cause changing input cardinalities (load shifts). Fluctuating order frequencies per product category reason changing selec tivities (data shifts). Varying utilization of external systems or network speeds especially, in wide area networks influences execution times. Workload characteristics in real productive systems change significantly over time as shown by surveys on eLearning query workloads [21], Website requests [20], and storage workload traces [22]. Missing adaptation to such workload changes is a serious problem because integration flows are deployed for long time horizons. Clearly, existing approaches could easily be extended to an adaptive optimize always model by trig gering optimization for each new plan instance or even permanently. However, due to many short running plan instances, this is inefficient and thus reasons the need for dedicated re optimization models.

Adaptive cost based optimization tries to repeatedly improve the current plan according to the changing workload characteristics. In contrast to traditional adap tive query processing [17], many consecutive instances of deployed integration flows are executed over time. This allows for efficient, asynchronous, inter instance plan re optimization. Here, the state of the art is *periodic re optimization*, where optimization is triggered with fixed optimization interval [4,23]. On the positive side, this simple model ensures full optimization potential and robust workload adaptation with moderate optimization overhead. On the negative side, it has the drawbacks of (1) many unnecessary re optimization steps, (2) adaptation delays, where we miss optimization opportunities, (3) maintenance of statistics that might not be used by the optimizer, and (4) the optimization interval as a high influence parameter.

1.1. Contributions

The main contribution of this paper is the novel concept of *on demand re optimization* for integration flows. This over comes drawbacks of existing techniques and ensures near optimal adaptive re optimization. The core idea is to exploit optimality conditions from the optimizer for *monitoring optimality* of the current plan and triggering *directed re optimization* only if necessary. Furthermore, we make the following detailed contributions that also reflect the struc ture of this paper (an extended and revised version of [4, Chapter 6]):

- First of all, we give a concise background description of integration flows and their optimization in Section 2. Additionally, we present the vision and solution over view of on demand re optimization in Section 3.
- Second, we introduce the monitoring of optimality in Section 4. We define the `PlanOptTree` as a compact representation of optimality conditions as well as we describe algorithms for creating `PlanOptTrees` and for monitoring optimality during statistic maintenance.
- Third, we introduce the directed re optimization in Section 5. This includes algorithms for determining the re optimization search space, directed re optimization for example optimization techniques and updating `PlanOptTrees` after re optimization.
- Fourth, we describe detailed results of our experiments in Section 6, where we achieve near optimal optimiza tion overhead and workload adaptation.
- Finally, we extensively survey related work in Section 7 and conclude the paper in Section 8.

2. Background and preliminaries

As a foundation, we first briefly describe the back ground of integration flows and their optimization. Sub sequently, we systematically analyze drawbacks of existing techniques for adaptive cost based optimization.

2.1. Integration flows

An integration flow is an imperative workflow descrip tion consisting of control flow , data flow , and interaction oriented operators that receive, extract, transform, and propagate data in the form of messages. Often control flow oriented languages such as BPEL (Business Process Execution Language), extended by relational operators, are used to specify these integration flows [4,7,14]. Compared

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

to pure data flows, these imperative control flows additionally express temporal dependencies between operators, e.g., execute operator o_1 *before* o_2. Intermediate results of integration flows are represented as messages, where each message is modeled as $msg_i = (t_i, d_i)$ with $t_i \in \mathbb{Z}^+$ being the message creation timestamp and d_i being a tree of name value data elements. Formally, an integration flow is defined as follows:

Definition 1 (*Integration flow*). A plan P of an integration flow is a sequence of *atomic* or *complex* operators $o = \{o_1, o_2, \ldots\}$, where complex operators recursively contain operator sequences. We denote the total number of operators as $m = |o|$. Each operator o_i may have an arbitrary number of input variables $\{ds_{in_1}(o_i), \ldots, ds_{in_{k1}}(o_i)\}$ and output variables $\{ds_{out_1}(o_i), \ldots, ds_{out_{k2}}(o_i)\}$ spanning a directed graph of (temporal and data) dependencies δ. An instance p_i of a plan P is instantiated in a time based manner or for each received message, and it executes o once. Due to complex operators such as iterations and alternatives some operators may be executed multiple times or not at all.

The following example shows such an integration flow. We use this as a running example throughout the whole paper.

Example 1 (*Integration flow*). New product orders are propagated from a specific ERP system s_1 to a DWH s_2 using plan P as shown in Fig. 1. An ERP adapter instance receives the incoming messages and transforms them into an internal XML representation. Plan instances are initiated and executed for each received message in arrival order (`Receive` operator). We then execute three different `Selection` operators in order to filter special purpose orders, where in this case each `Selection` applies an expensive probe filter. Such a filter probes all returned elements of a given XPath expression against a hashset of disjunctive predicates and discards elements where this probe is unsuccessful. Subsequently, a `Switch` operator redirects the control flow using content based predicates to specific `Translation` operators that apply XML message transformations. Finally, the result is loaded into the DWH using `Assign` and `Invoke` operators as well as a DB adapter instance.

Integration flows such as our example are deployed once into an integration platform and then repeatedly executed.

System architecture: The major commercial integration platforms such as SAP Process Integration, IBM Message Broker, or MS Biztalk Server all exhibit a common system architecture. Inbound adapter instances (e.g., s_1 in Example 1) receive messages from external systems, transform them into an internal presentation (e.g., XML), and append them to persistent inbound message queues. For each received message a plan instance is executed, in serial order of message arrival. Those plans interact (read/write) with external systems via outbound adapter instances (e.g., s_2), which are similar to inbound adapters an abstraction of different types and instances of external systems.

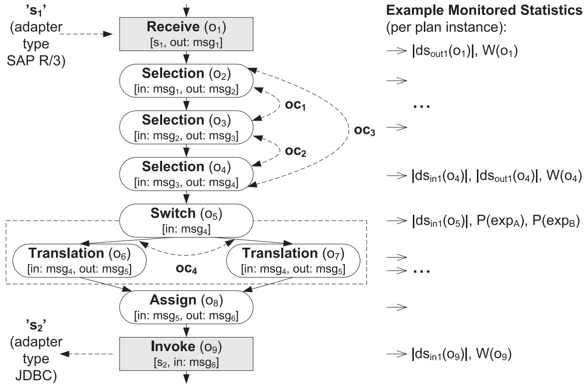

Fig. 1. Running example integration flow.

2.2. Optimization of integration flows

Regarding the existing high performance demands on integration platforms, we now give an overview of adaptive cost based re optimization of integration flows [4,23].

Cost model: As a foundation for cost based optimization and due to the specific characteristics of (1) missing statistics (external and unknown data), (2) arbitrary interactions with external systems (black box), and (3) control flow oriented operators, we employ a *double metric cost model*. Essentially, we monitor and incrementally maintain runtime statistics such as execution times $W(o_i)$ and input/output cardinalities $|ds|$ at an operator level. These statistics are fed into operator type specific cost formulas of cardinality dependent costs $C(o_i)$ (tailor made for known operators, linear for black box interactions) and execution times $W(o_i)$. The execution time of a rewritten subplan P' can then be estimated by an aggregate (e.g., sum for operator sequences) over adjusted operator costs with

$$\hat{W}(o'_i) = \hat{C}(o'_i)/C(o_i) \cdot W(o_i). \tag{1}$$

This time based cost model enables the comparison of all different types of operators, allows one to take parallelization into account, and it is self adjusting to changing workload characteristics.

Optimization algorithm: Our optimization algorithm then uses this cost model as a basis for cost based optimization techniques. We use a transformation based approach in order to preserve semantic correctness of the initially specified, imperative plan. In order to ensure low latency, our basic optimization objective is to minimize the *average* plan execution time with

$$\phi\colon \min \hat{W}(P) \tag{2}$$

but it can be combined with techniques for throughput optimization by taking message waiting times into account. The algorithm essentially iterates over the hierarchy of operator sequences and applies optimization techniques. Finally, it recompiles and exchanges plans. Each optimization technique relies on specific optimality conditions for certain rewriting pattern. We employ techniques from traditional data management, techniques from programming language compilers, and new tailor made techniques for integration flows.

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

Adaptive re optimization: Furthermore, we use *periodic re optimization* for adaptation to changing workloads [23]. The simple yet effective basic idea is to periodically trigger re optimization with an optimization interval Δt to adapt the deployed plan to the current runtime statistics. We use an example to illustrate this re optimization model.

Example 2 (*Periodic re optimization*). Fig. 2 shows the plan execution times in a scenario with two workload shifts. Periodic re optimization triggers re optimization with an optimization interval Δt. After a workload shift occurred (e.g., ws_1), the next re optimization will find the new optimal plan (e.g., P') but there is an adaptation delay until plan rewriting takes place. If no workload shift has occurred during Δt, we do not find a new plan and thus execute unnecessary re optimization steps.

To summarize, periodic re optimization exhibits several drawbacks. First, there are adaptation delays, where we miss optimization opportunities. Second, we might execute many unnecessary full re optimization steps. Third, we might gather statistics that are not used by the optimizer. Fourth, the optimization interval is a parameter that requires tuning. Since the severity of these drawbacks depends on the workload, we systematically quantify the problems (1), (2), and (4) by the following sensitivity analysis.

Example 3 (*Sensitivity analysis*). Assume a scenario, where we serially execute $n = 100{,}000$ instances of plan P. Let $|ws|$ denote the number of workload shifts that occur uniformly distributed every $n/|ws|$ plan instances. Further more, let $W(P) = 0.5$ s be the unoptimized plan execution time, let $W(P') = 0.1$ s be the optimized plan execution time, and let $W(opt) = 2$ s be the optimization time. Fig. 3 then shows the total runtime when using periodic re optimization with optimization interval Δt. Fig. 3(a) shows the influence of $|ws|$ with fixed $\Delta t = 5$ min. For increasing workload dynamics, periodic re optimization degenerates from near optimal performance to the unoptimized case. Fig. 3(b) illustrates the impact of Δt. For small Δt there is high re optimization overhead and for large Δt there are high adaptation delays. In case of almost static workloads ($|ws| = 10$), the overall performance is indeed fairly in sensitive to Δt. However, as workload dynamics increase ($|ws| = 100$ and $|ws| = 1{,}000$) the sensitivity increases and

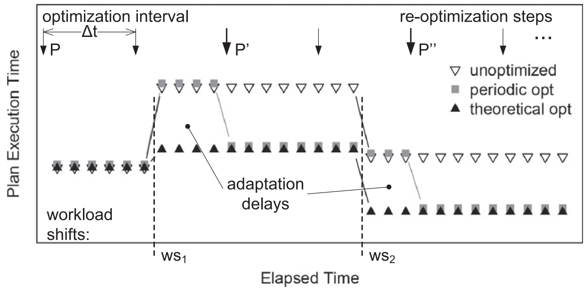

Fig. 2. Example periodic re-optimization.

thus it becomes difficult to find a good Δt. Finally, it is noteworthy that the alternative approach of triggering re optimization if statistics have changed significantly has the same conceptual problem. Due to statistic fluctua tions, a sensitivity parameter is required for deciding on change significance. If it is chosen too high, there are high adaptation delays; if chosen too low, there are many unnecessary re optimization steps.

We conclude that periodic re optimization is a simple indeed effective technique if workload changes are rare events or if the workload is exactly known. On the downside, the overall performance can degenerate to the performance of the unoptimized case.

3. Solution overview

In this paper, we present the novel *on demand re optimization* model that achieves both, robust and near optimal adaptive re optimization. Our core idea is (1) to monitor optimality of the current plan via optimality conditions and (2) to apply directed re optimization if conditions are violated. The following example illustrates these two basic concepts.

Example 4 (*On demand re optimization*). Consider the sub plan $P = (o_2, o_3)$ consisting of two `Selection` operators (see Fig. 1). The search space for this subplan is illustrated as a plan diagram[1] in Fig. 4(a). The plan (o_2, o_3) is optimal

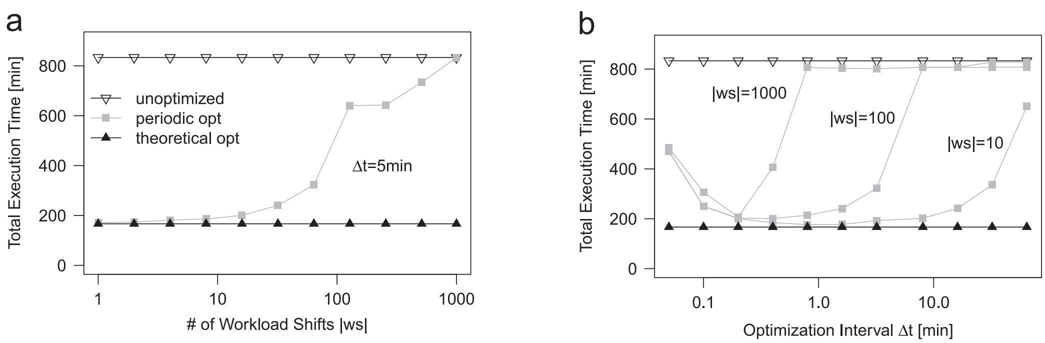

Fig. 3. Sensitivity of periodic re-optimization. (a) Workloads shifts and (b) optimization interval.

[1] Traditional two-dimensional plan diagrams [24] rely on a complete what-if search space analysis, while we model break-even points of plans via multi-dimensional optimality conditions.

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

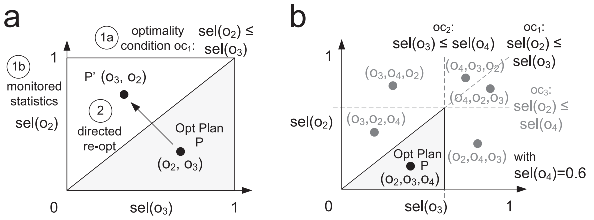

Fig. 4. Plan search space partitioning. (a) Plan diagram $P(o_2, o_3)$ and (b) plan diagram $P(o_2, o_3, o_4)$.

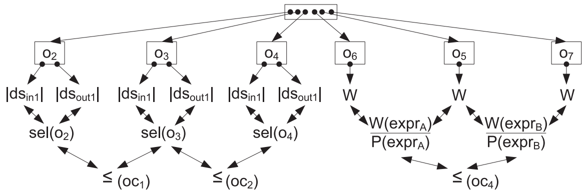

Fig. 5. PlanOptTree of the running example.

as long as $oc_1: sel(o_2) \leq sel(o_3)$ (with $sel(o_i) = |ds_{out}(o_i)|/|ds_{in}(o_i)|$), i.e., the selectivity of o_2 is lower or equal to the selectivity of o_3. Thus, the optimality condition oc_1 models optimality of the current plan (1a). Then, we maintain only necessary statistics that are involved in this optimality condition, i.e., $sel(o_2)$ and $sel(o_3)$, and use them for con tinuously monitoring of optimality (1b). If oc_1 is violated, we use this condition for directed re optimization in order to determine $P' = (o_3, o_2)$ as the new optimal plan (2). As shown for a larger subplan $P = (o_2, o_3, o_4)$ in Fig. 4(b), we maintain only optimality conditions of the current (optimal) plan, i.e., oc_1 and oc_2 instead of all possible plans. For the sake of a clear presentation, we use this simple example throughout the whole paper but as we will exemplify in Section 5.2 the basic concepts apply to arbitrary operators and optimization techniques.

For enabling on demand re optimization, we introduce the PlanOptTree data structure as a compact representation for arbitrary optimality conditions. The PlanOptTree of a current plan indexes monitored and derived statis tics as well as current optimality conditions. We then use it for incremental online statistics maintenance and checking of optimality conditions. This continuous monitoring of opti mality allows us to immediately trigger re optimization if necessary, i.e., if violated conditions guarantee that we will find a plan with lower costs. In addition, the violated condi tions can also be exploited for directed re optimization.

Example 5 (PlanOptTree). The PlanOptTree of our running example (Fig. 1) is shown in Fig. 5. It includes two optimality conditions (oc_1, oc_2) for expressing the ordering of the Selection operators o_2, o_3 and o_4 according to their selectivities (oc_3 is omitted due to transitivity) and the condition oc_4 expressing branch prediction of the Switch operator o_5 regarding its cost weighted path frequencies.

In the following, we explain the monitoring of optimality and directed re optimization using PlanOptTrees in detail.

4. Monitoring optimality

In this section, we formally define the PlanOptTree and show how to create a PlanOptTree during initial plan optimization. Furthermore, we explain how to use it for statistic maintenance and discuss when to trigger re optimization.

4.1. Plan optimality trees

A PlanOptTree, which general structure is shown in Fig. 6, models plan optimality and it is defined as follows:

Definition 2 (PlanOptTree). Let P denote the *optimal* plan with regard to the current statistics, let m be the number of operators, and let s be the number of statistics per operator. Then, the PlanOptTree is defined as a graph of five strata representing all optimality condi tions of P:

- *RNode* (*stratum* 1): A single *root node* refers to m' $(1 \leq m' \leq m)$ operator nodes (ONode).
- *ONode* (*stratum* 2): An *operator node* specifies a unique plan operator via a node identifier *nid* and it refers to s' $(1 \leq s' \leq s)$ statistic nodes (SNode).
- *SNode* (*stratum* 3): A *statistic node* exhibits one of the s atomic statistic types s_j (e.g., input cardinality), where this type must be unique for the parent operator o_i. Each SNode maintains a sliding window of statistic tuples monitored for (o_i, s_j), a single aggregate, as well as references to child sets of complex statistic nodes (CSNode) and optimality condition nodes (OCNode).
- *CSNode* (*stratum* 4): A *complex statistic node* is a mathe matical expression using all referenced parent SNodes or CSNodes as operands, where a CSNode can refer to SNodes of different operators. Further, a CSNode refers to child sets of CSNodes and OCNodes. Hence, arbitrary hierarchies of CSNodes are possible. CSNodes can also be constants or external values.
- *OCNode* (*stratum* 5): An *optimality condition node* is defined as a boolean expression $op_1 \theta op_2$, where θ denotes an arbitrary binary comparison operator and the operands op_1 and op_2 refer to any CSNode or SNode, respectively. The optimality condition is defined as violated if the expression evaluates to false. In addition, each OCNode refers to its source of creation in terms of the originating optimization technique.

References to nodes of strata 1 and 2 are unidirectional, while nodes of strata 3 5 are bidirectional. Furthermore, a PlanOptTree includes a MEMO structure[2] in order to optionally mark subgraphs that have already been evalu ated because nodes might be reachable over multiple paths.

[2] For clarity of presentation, we omit the usage and maintenance of this MEMO structure in our algorithms. From a high-level perspective, it can be understood as a lookup table for memoization in order to reuse results of subgraphs and to prevent redundant computations.

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

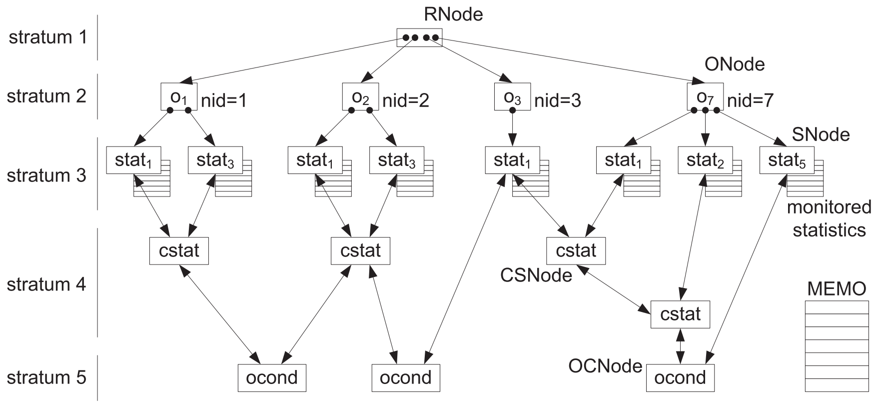

Fig. 6. General `PlanOptTree` structure.

The defined general `PlanOptTree` structure exhibits the following four fundamental properties:

- *Optimality*: Since `PlanOptTree` models plan opti mality, we will only find a plan with lower cost if at least one optimality condition is violated. Thus, there is no need for re optimization until we detect a violation.
- *Transitivity*: If statistics are included in multiple optim ality conditions, we can leverage transitively given optimality conditions. For this reason, potentially only a subset of all relevant optimality conditions are required to monitor the optimal plan.
- *Minimality*: A `PlanOptTree` includes only operators and statistics that are included in optimality condi tions. Thus, we achieve minimal statistics monitoring and minimal condition evaluation (see transitivity) under the requirement of still ensuring optimality.
- *Directed re optimization*: In case of violated optimality conditions, the knowledge of still valid conditions can be exploited for reducing the re optimization search space to a subset of the original search space.

These properties hold for a *complete* `PlanOptTree` that is defined to cover all relevant optimality conditions of a plan. Otherwise, we would have redundancy or miss ing conditions.

4.2. *Creating PlanOptTrees*

During initial deployment of a plan, the full cost based optimization is executed once and an initial `PlanOptTree` is created. Subsequently, we solely rely on incremental and directed re optimization by leveraging this `PlanOptTree`. We now explain how to create the initial `PlanOptTree`.

Our transformation based optimization algorithm (described in Section 2.2) recursively iterates over the hierarchy of operator sequences and changes the current plan by applying optimization techniques. For on demand re optimization, we extended the optimizer in a way that it does not only change the current plan but additionally, each applied optimization technique also returns a *partial* `PlanOptTree` representing optimality conditions for the considered subplan. The use of partial `PlanOptTrees` at the optimizer interface is reasonable because directed re optimization potentially considers only subplans and thus can only return partial `PlanOptTrees`. Creating the initial `PlanOptTree` then reduces to merging all partial `PlanOptTrees` to a minimal representation. In the following, we describe an example and the general case algorithm.

Example 6 (*Merging partial* `PlanOptTrees`). Recall the running example plan P and assume the two partial `PlanOptTrees` shown in Fig. 7(a) and (b). These `PlanOptTrees` have been created by applying selection reordering on opera tors (o_2, o_3, o_4). In detail, they consist of ONodes, SNodes, a CSNode `Selectivity`, and an OCNode. Both partial `PlanOptTrees` include operator o_3 and its selectivity CSNode. Therefore, we add only o_4 and its child nodes from pot_2 to pot_1. When doing so, the dangling reference from the new optim ality condition to $sel(o_3)$ of pot_2 is modified to refer to the existing $sel(o_3)$ of pot_1. The final merged `PlanOptTree` pot is shown in Fig. 7(c).

Algorithm 1. `PlanOptTree` creation (A PC).

```
Require operator op, global PlanOptTree root (initially NULL)
  o ← op.getSequenceOfOperators()
  for i ← 1 to |o| do                                   // for each operator o_i
    if type(o_i) ∈ (Plan, Switch, Fork, Iteration)
    then                                                 // complex
      A − PC(o_i)               // recursive invocation for complex operators
    else                                                 // atomic operators
      ppot ← getPartialOptTree(o_i)
      if root   NULL then
        root ← ppot
      else                                  // merge partial  PlanOptTrees
        for all on ∈ ppot.onodes do                      // for each ONode on
          if root.containsONode(on.nid) then
            eon ← root.getOperator(on.nid)
            for all sn ∈ on.snodes do                    // for each SNode sn
              if eon.containsSNode(sn.type) then
                eson ← eon.getSNode(sn.type)
                modifyDanglingRefs(eon, eson, on, sn)
              else
                eon.snodes.add(sn)
          else
            root.onodes.add(on)                        // add operator subtree
  return root
```

Algorithm A PC (`PlanOptTree` *creation*): The A PC (see Algorithm 1) creates the initial `PlanOptTree` by recursively

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

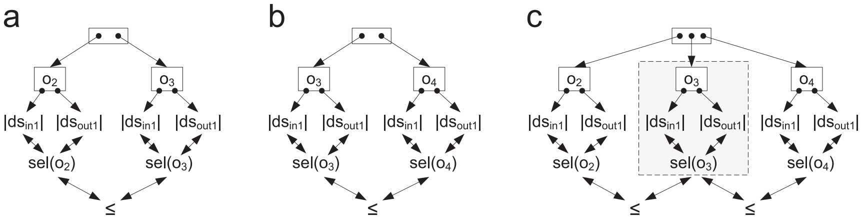

Fig. 7. Merging partial `PlanOptTrees`. (a) Partial pot_1, (b) partial pot_2 and (c) merged pot.

iterating over all operators and merging respective partial `PlanOptTrees`. In case of complex operators, we recursively invoke the A PC, where all subcalls have access to the *root* `PlanOptTree`. At each operator (atomic or complex), we apply all operator type related optimization techniques, where each technique returns resulting partial `PlanOpt Trees`. Some optimization techniques directly optimize whole sequences of operators during the first invocation on an operator of this sequence, where invocations for other operators of this sequences are then ignored. For example, the first invocation of selection reordering on operator o_2 will reorder the sequence (o_2, o_3, o_4) and returns pot_1 and pot_2. If no `PlanOptTree` exists so far, the first partial `PlanOptTree` is used as *root*; otherwise, we merge the partial `PlanOpt Tree` with the existing *root*. When merging, we check the existence of operators as well as statistic nodes, and we add new nodes if required. For complex statistic nodes, we modify dangling references in order to recursively change the refer ences of complex statistics and optimality conditions to the existing `PlanOptTree`. Identical CSNodes are determined by ID or equivalence of input nodes and CSNode class.

Optimizations: Context knowledge of operators can be exploited for optimization. For example, we reuse the output cardinality SNode of operator o_i as input cardinality SNode for operators with data dependencies to o_i (partially applied in our examples) because these statistics are per se equivalent. In general, reusing statistics is beneficial for expensive statistic maintenance approaches.

Complexity analysis: Clearly, the space complexity of a `PlanOptTree` and the time complexities of `PlanOptTree` algorithms depend on the complexity of applied optimization techniques and their optimality conditions. Regarding mon itoring optimality, the `PlanOptTree` is indeed optimal due to its properties of transitivity and minimality. However, for certain plan structures and problems, efficient bounds can be established. For example, Appendix A shows a worst case complexity of $\mathcal{O}(m^2)$ for *local* reordering sequences of operators.

The `PlanOptTree` represents plan optimality via optimality conditions based on current statistics. Hence, only the update of statistics can cause the need for re optimization.

4.3. Updating and evaluating statistics

In order to enable immediate re optimization in case of violated optimality conditions, we use the `PlanOpt Tree` also for incremental online statistic maintenance and continuous monitoring of plan optimality via optimality conditions.

A `PlanOptTree` maintains statistics that are required for monitoring optimality conditions. Atomic statistics are as usual gathered at operator level during plan execution and immediately inserted into the `PlanOptTree`, where unnecessary statistics are declined. Every SNode exhibits a state in terms of *aggregated* atomic statistics (e.g., average input cardinality) because we optimize the *average* case. Different aggregation methods or even time series models can be used. By default, we employ the *simple exponential smoothing*:

$$\mathrm{EMA}_t = \mathrm{EMA}_{t-1} + \alpha(s_t(o_i) \quad \mathrm{EMA}_{t-1}) \quad \text{with } \alpha \in [0,1]. \tag{3}$$

Included statistics $s_t(o_i)$ exhibit exponentially decaying weights due to the recursive computation, where α is used to adjust the smoothing sensitivity. Here, general purpose optimization algorithms such as L BFGS B [25] can be used for parameter estimation. This EMA is suitable for incre mental statistics maintenance with sliding window semantics because (1) it relies on positive incremental maintenance for new values anyway, (2) it can adapt fast to workload changes, and (3) it does not require negative maintenance for expired values due to decaying weights. After incremental statistics maintenance, we also update the hierarchy of relevant com plex statistic measures (CSNodes), and finally we evaluate relevant optimality conditions (OCNodes) as well as trigger re optimization on demand, i.e., only if required because a plan with lower costs exists.

Robustness strategies: Triggering re optimization naïvely for any violated optimality condition ensures immediate adaptation but might cause the problem of *frequently chan ging plans* (instability). There are two potential reasons. First, by assuming independence of monitored conditional selec tivities, data correlation can lead to cyclic re optimizations. Second, if statistics are close to a break even point and fluctuate to some extend for different reasons, we might also have frequent plan changes. We explicitly address these issues of instability with the following strategies:

- *Correlation tables*: Data correlations are addressed by computing conditional selectivities via so called correla tion tables. The idea is to maintain selectivities over multiple versions of a plan, where we store and maintain a row of unconditional and conditional selectivities for each pair of operators with data dependencies in the current plan. Until we are able to monitor and use the unconditional selectivity, we assume statistical indepen dence only. However, using statistics from multiple plan versions prevents us from making wrong decisions multi ple times. Starvation due to outdated statistics is avoided by a time based decay. The integration into the `PlanOpt Tree` is done via a CSNode `ConditionalSelectivity`

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

that maintains and reads the correlation table. This light weight approach is effective for groups of few correlated attributes.

- *Lazy condition violation*: In order to overcome the problem of statistic fluctuations and serialized statistic updates, we trigger re optimization lazily, i.e., only if the condition is violated τ_1 times (lazy count). As a heuristic, we set this count to the number of SNodes of the `PlanOptTree` because often already small lazy counts are sufficient. Furthermore, the condition must be violated at least by a relative cost threshold τ_2 and a true condition evaluation resets the lazy count.
- *Minimal existence time*: As a fall back mechanism for all problems of instability, we introduce Δt as minimal existence time of a plan. We collect statistics but do not evaluate optimality during Δt after the last re optimization. However, Δt is only the minimal interval between re optimizations; afterwards, we continuously monitor optimality and adapt the plan if necessary in order to prevent adaptation delays.

Putting it altogether, we now describe the overall statistic maintenance algorithm and analyze existing parameters.

Algorithm 2. Insert statistics (A IS).

```
Require operator id nid, statistic type, statistic value, lastopt
if (on ← root.getOperator(nid))  NULL or
   (sn ← on.getSNode(type))  NULL then
  return                                  // statistic not required
sn.maintainAggregate(value)
ret ← true
if (time − Δt) > lastopt                  // min existence time
  for all cn ∈ sn.csnodes do              // for each CSNode cn
    ret ← ret and cs.computeStats()
  for all oc ∈ sn.ocnodes do              // for each OCNode oc
    ret ← ret and oc.isOptimal()
if ¬ret then
  A-PTR()                // actively triggering re-opt (start thread)
```

Algorithm A IS (insert statistics): The A IS (see Algorithm 2) is invoked as we measure statistics. It then realizes incremental online statistics maintenance for each operator statistic and triggers re optimization if required. Starting from the root, we search the operator node by *nid*, then search the statistic node of this operator by *type* and finally, if the node exists, maintain the aggregate. Furthermore, if the minimum existence time is exceeded, we check optimality conditions. For this purpose, we recursively compute and check relevant complex statistic measures and optimality conditions that are reachable over child references. During checking optimality, we also update the specific lazy counters. If there is at least one violated optimality condition that reached the lazy count, we trigger re optimization by starting an *asynchronous* re optimization thread.

Asynchronous re optimization: In contrast to synchronous mid query optimization where the remaining plan depends on the optimizer output we do not block plan execution and statistic maintenance. Furthermore, all additional triggers for re optimization are simply rejected as long as the optimizer thread is running. Finally, after successful optimization, we switch plans on the next possible point i.e., just before starting the next plan instance and enable re optimization again.

Parameter analysis: The parameters of on demand re optimization have fairly static influence and thus do not require much tuning. First, if the minimal existence time Δt is smaller than the interval between workload shifts, there are no changes. Otherwise, adaptation delays linearly increase. Second, adaptation delays also linearly increase with increasing lazy count τ_1. However, Δt and τ_1 can be kept low by default as we will show in our evaluation. All other parameters are the same for both on demand and periodic re optimization.

To summarize, the monitoring of plan optimality and re optimization on demand minimizes adaptation delays and at the same time prevents unnecessary re optimization steps.

5. Directed re-optimization

Once re optimization has been triggered by violated optimality conditions during statistic maintenance, we exploit these conditions for directed re optimization of the current plan. We first determine the reduced re optimization search space. Then, we apply directed re optimization instead of full re optimization. Finally, we incrementally update the existing `PlanOptTree` according to the new conditions.

5.1. Re optimization search space

Since directed plan re optimization aims at considering only a subset of the complete search space as already shown in Fig. 4, we first need to determine this reduced re optimization search space. We exploit violated optimality conditions for determining (1) the optimization techniques that produced these conditions and (2) the minimal set of operators that need to be reconsidered by those techniques. Generally speaking, the re optimization search space consists of the set of operators, affected by violated optimality conditions. In case of multiple violated conditions, this is the union of affected operators. In order to guarantee optimality, we also need to take the *transitivity* property of the `PlanOptTree` into account.

In general, we follow a bottom up approach to determine this re optimization search space. We start at each violated optimality condition and traverse all optimality conditions that are reachable over *transitivity connections*. Such a transitivity connection is defined as an atomic or complex statistic node connected with two or more optimality conditions. Transitivity chains of arbitrary length are possible, where the end is given by the lack of transitive connections or by the first condition that is still optimal. Finally, termination is guaranteed because the `MEMO` structure prevents cycles.

Example 7 (*Re optimization search space*). Assume the `PlanOptTree` of Example 6. During initial optimization, selectivities of $sel(o_2) = 0.2$, $sel(o_3) = 0.3$ and $sel(o_4) = 0.4$ resulted in the optimality conditions shown in Fig. 8(a). Re optimization was triggered because the selectivity of

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

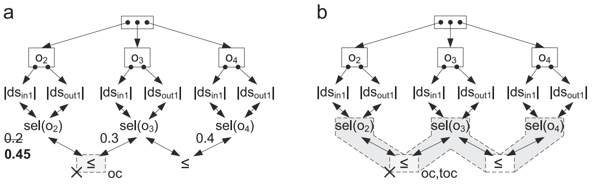

Fig. 8. Re-optimization search space. (a) Violated OCNode and (b) transitive OCNode.

operator o_2 changed to $sel(o_2) = 0.45$ and thus violates $sel(o_2) \leq sel(o_3)$. Hence, we would directly reorder operators o_2 and o_3 during re optimization. Due to the transitivity of optimality conditions, we need to check the selectivity of operator o_4 as well, because we do not know if the implicit optimality condition of $sel(o_2) \leq sel(o_4)$ (given by $sel(o_2) \leq sel(o_3)$ and $sel(o_3) \leq sel(o_4)$) still holds. We traverse the `PlanOptTree` as shown in Fig. 8(b) and see that this transitive condition is violated as well, while the second optimality condition $sel(o_3) \leq sel(o_4)$ is still valid.

Algorithm 3. Trigger re optimization (A TR).

```
Require plan P, invalid optimality conditions C
C' ← C
for all oc ∈ C do                                   // for each OCNode
  for all oc1 ∈ oc.op1.ocnodes do                   // for each OCNode of op1
    if oc1.θ   oc.θ and oc1.op2   oc.op1 then
      if ¬oc1.isOptimal(oc1.op1.agg, oc.op2.agg) then
        C' ← C' ∪ oc1
        rCheckTransitivity (oc, oc1.op1, left)
  for all oc2 ∈ oc.op2.ocnodes do                   // for each OCNode of op2
    if oc2.θ   oc.θ and oc2.op1   oc.op2 then
      if ¬oc2.isOptimal(oc2.op2.agg, oc.op1.agg) then
        C' ← C' ∪ oc2
        rCheckTransitivity(oc, oc2.op2, right)
PPOT ← optimizePlan(P, C')                          // apply directed re-opt
A-UP(PPOT, C')                                      // update PlanOptTree
```

Algorithm A TR (trigger re optimization): The A TR (see Algorithm 3) adds all violated optimality conditions to the set $\mathcal{C}$ and then it recursively traverses transitivity chains for each condition in $\mathcal{C}$ in order to collect transitively violated conditions. There, we check transitivity filters (direction of operands and comparison operator) and transitive optimality filters. Finally, we invoke the optimizer with all violated conditions $\mathcal{C}'$ as re optimization search space. After successful re optimization, we update the `PlanOptTree` (A PPR).

Optimality analysis: Directed re optimization with the reduced search space and full re optimization create equivalent plans. The intuition is as follows. Any rewriting possibility between operators is represented by an explicit or a transitive optimality condition. Arbitrary complex optimality conditions can be used to prevent the starvation in local minima. All operators included in violated explicit or transitive conditions are used by directed re optimization. Hence, directed re optimization guarantees optimality. For a detailed analysis, we refer to Appendix B.

5.2. *Example optimization techniques*

Based on the determined re optimization search space, we apply directed re optimization. As this depends on the concrete optimization techniques, we exemplify the monitoring of optimality and directed re optimization for local selection reordering, heuristic join reordering, and heuristic vectorization. On demand re optimization is applicable for arbitrary operators and optimization techniques because a `PlanOptTree` can represent arbitrary optimality conditions. However, since the `PlanOptTree` complexity depends on the used optimization techniques, it is especially practical for local and heuristic rewrites.

5.2.1. *Selection reordering*

Our running example optimization technique selection reordering applies local rewriting decisions via a variant of bubble sort with an average time complexity of $\mathcal{O}(m^2)$. This exchange based sort algorithm is well suited for rewriting imperative plans because it allows for on the fly correctness checks. The optimality condition is $sel(o_i) \leq sel(o_j)$, where operator o_i is a data flow predecessor of operator o_j. As shown throughout the paper, on demand re optimization then requires m 1 optimality conditions for monitoring optimality of m (or selective) operators. In the best case, the re optimization search space comprises a single violated condition, where we directly reorder the two involved operators with $\mathcal{O}(1)$. In the worst case, all optimality conditions are violated such that we need to sort all m operators with $\mathcal{O}(m^2)$.

5.2.2. *Heuristic join reordering*

Join ordering heuristics are typically used if the number of joins exceed certain limits or if we assume independence of selectivities [26,27]. In the following, we first describe our used heuristic join reordering algorithm and subsequently discuss our extensions for on demand re optimization.

Preliminaries: We consider (1) only left deep join trees (no zig zag trees, no bushy trees), (2) without cross products, and (3) only one join implementation, but we decide on join implementations afterwards. Using these assumptions and our asymmetric cost functions, there exist at most i.e., for clique queries $n!$ alternative plans for joining n datasets [27]. Using our cost model, the costs of, for example, a nested loop join are computed by $C(R \bowtie S) = |R| + |R||S|$ and the join output cardinality can be derived by $|R \bowtie S| = f_{R,S} \cdot |R||S|$ with a join filter selectivity of $f_{R,S} = |R \bowtie S| / (|R||S|)$. Thus, the costs of a left deep join tree $(R \bowtie S) \bowtie T$ are $C((R \bowtie S) \bowtie T) = |R| + |R||S| + f_{R,S} \cdot |R||S| + f_{R,S} \cdot |R||S||T|$.

Algorithm: Our basic algorithm is again an exchange based join reordering heuristic and it relies on binary reordering decisions between subsequent join operators (e.g., $((* \bowtie R) \bowtie S) \bowtie *$ vs. $((* \bowtie S) \bowtie R) \bowtie *$). The underlying observation is similar to Bellman's Principle of Optimality used for dynamic programming that the costs of subplans before and after such a local reordering are independent of that order. Hence, we just compare $C((* \bowtie R) \bowtie S) \leq C((* \bowtie S) \bowtie R)$. The algorithm works as follows: first, we select the input dataset with the smallest cardinality and reorder it with the existing first join operand if possible. Second, we iterate over all joins and reorder adjacent join pairs if possible and if beneficial. This algorithm basically applies to clique and star queries only, while for arbitrary query types, groups of operators are reordered similar to

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

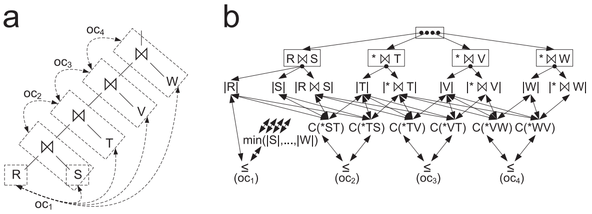

Fig. 9. Example join reordering. (a) Opt. conditions and (b) example `PlanOptTree`.

the IKKBZ algorithm [27]. In general, this heuristic algorithm exhibits an average time complexity of $\mathcal{O}(m^2)$.

On demand re optimization: For monitoring plan optimality of arbitrary left deep join trees, we require m optimality conditions. Fig. 9(a) shows an example join reordering of $R \bowtie S \bowtie T \bowtie V \bowtie W$ ($m = 4$ operators, clique query) and its optimality conditions. First, one condition monitors the minimal cardinality of the first dataset with $oc_1 : |R| \leq \min(|S|, |T|, |V|, |W|)$. Second, there are m 1 conditions monitoring the ordering of join operators. For example, the optimality of executing $* \bowtie S$ before its successor $* \bowtie T$ is given if the following condition holds:

$$\begin{aligned} oc_2 : & |R| + |R||S| + f_{R,S} \cdot |R||S| + f_{R,S} \cdot |R||S||T| \\ & \leq |R| + |R||T| + f_{R,T} \cdot |R||T| + f_{R,T} \cdot |R||T||S|, \end{aligned} \tag{4}$$

where oc_2 can be algebraically simplified. It is possible to monitor all cardinalities $|R|, \ldots, |W|$ but only the conditional selectivities $f_{R,S}, \ldots, f_{*,W}$. Hence, we either assume statistical independence of selectivities with $f_{R,T} = f_{(R \bowtie S),T}$ or we use our correlation table as described in Section 4.3. The `PlanOptTree` of this example is shown in Fig. 9(b). Finally, directed re optimization sorts only operators in violated conditions such that we have a re optimization time complexity of $\mathcal{O}(m^2)$ for the worst case but $\mathcal{O}(1)$ for the best case.

5.2.3. Heuristic vectorization

Plan vectorization is an integration flow specific optimization technique [28]. The core idea is to rewrite instance based plans (operator at a time) into vectorized plans with pipelined message execution (one thread per operator) in order to exploit pipeline parallelism over multiple plan instances. This increases message throughput and still ensures semantic correctness for control flows. We introduced the cost based vectorization that computes the optimal grouping of operators to a minimal number of k execution buckets (threads). This ensures the optimal degree of pipeline parallelism but achieves less message latency and less resource contention (threads, cache). Cost based vectorization is a generalization of execution strategies because instance based plans ($k = 1$) and fully vectorized plans ($k = m$) are specific cases.

Algorithm: The objective is to minimize k under the constraint that bucket execution time of the most expensive operator because it limits the pipeline throughput anyway (convoy effect [29]):

$$\phi : \min_{1 \leq k \leq m} |\forall i \in [1, k] : \sum_{o_j \in b_i} W(o_j) \leq W(o_{max}). \tag{5}$$

Table 1
Analysis of example optimization techniques.

Optimization technique	Traditional algorithm	$\lvert oc \rvert$	Directed reopt	
			Best	Worst
Selection reordering (5.2.1)	$\mathcal{O}(m^2)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(m^2)$
Join reordering (5.2.2)	$\mathcal{O}(m^2)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(m^2)$
Vectorization (5.2.3)	$\mathcal{O}(m^3)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$

This problem exhibits an exponential complexity of $\mathcal{O}(2^m)$. Hence, we apply the following heuristic algorithm: First, we determine the maximum operator execution time. Second, depending on the plan structure (*flow/sequence*), we group operators, similar to bin packing heuristics, in a first fit/next fit manner with a time complexity of $\mathcal{O}(m^2)/\mathcal{O}(m)$. In any case, the total algorithm complexity is dominated by dependency graph creation with $\mathcal{O}(m^3)$.

On demand re optimization: Plan optimality for a *sequence* of operators is then monitored via k ($1 \leq k \leq m$) optimality conditions. First, one condition ensures that each bucket b_i fulfills the constraint that its total execution time does not exceed the execution time of the most time consuming operator with $oc_1 : \max(W(b_i), \ldots, W(b_k)) \leq W(o_{max})$. Second, regarding the next fit approach of our heuristic algorithm, for each bucket except the first one (k 1), one condition monitors if the first operator o_j of this bucket b_i still cannot be assigned to the previous bucket b_{i-1} with $oc_i : W(b_{i-1}) + W(o_i) \geq W(o_{max})$. Directed re optimization starts at the bucket determined by oc_i. Hence, we have a re optimization time complexity of $\mathcal{O}(1)$ for the best case and $\mathcal{O}(m)$ for the worst case, but dependency graph creation is not required.

5.2.4. Discussion of complexity analysis

Table 1 summarizes the complexity analysis of the presented example techniques. We compare the time complexity of the full algorithms, monitoring optimality ($|oc|$) and directed re optimization. Based on these observations we can draw two important conclusions.

First, monitoring optimality is commonly more efficient than full re optimization. Keeping all possible plans is prohibitive as this would result in many cases in factorial time and space complexity for creating all plans. In contrast, our idea relies on monitoring optimality of the current (optimal) plan only. Generally speaking, we benefit from monitoring optimality whenever the complexity of deciding if a solution is optimal is lower than the complexity of finding the optimal solution. This holds for many optimization problems.

Second, there are many cases, where we can benefit from directed re optimization. Commonly, the best case re optimization is constant because a single violated condition can enable to directly infer the new optimal plan, while the worst case (e.g., all conditions violated) is equal to the traditional algorithm. The benefit of directed re optimization is due to (1) a reduced search space per technique, and (2) potentially reconsidering only a subset of techniques. The latter is especially important in scenarios, where we have a mixture of static optimization decisions and high dynamics for other decisions. Furthermore, directed re optimization

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

allows for step wise re optimization, which is beneficial for incrementally learning conditional statistics and sometimes in highly dynamic scenarios, where we have many re optimizations. We refer to Appendix B for an analysis of convergence properties of step wise re optimization.

Finally, there is a trade off between monitoring and re optimization efficiency. So far we realized on demand re optimization for example techniques, including the here presented ones. The specific `PlanOptTree` design, espe cially for complex optimization techniques, constitutes interesting future work.

5.3. Updating PlanOptTrees

Directed re optimization considers only a subset of operators and thus, can only return partial `PlanOptTrees` of rewritten subplans. We therefore incrementally update the existing `PlanOptTree` with new partial `PlanOptTrees` after successful re optimization. The result is equivalent to creating the `PlanOptTree` from scratch (A PC).

Algorithm A UP (Update `PlanOptTree`): The A UP starts bottom up from all violated optimality conditions and removes those OCNodes except for transitively violated conditions because the new partial `PlanOptTree` is aware of them. Then, it recursively removes statistic nodes that do not refer to any child nodes and were affected by re optimization. This removes only nodes included in violated optimality conditions. Finally, we apply the merge of A PC (Section 4.2) for new partial `PlanOptTrees`, copy statistics if necessary, and switch plans as described in Section 4.3.

To summarize, directed re optimization reduces the over head per re optimization step, especially for techniques with large search space. Future work might consider reusing plans and `PlanOptTrees`. Finally, we start over with monitoring optimality during statistic maintenance (Section 4.3).

6. Experimental evaluation

Our experiments study the behavior of on demand re optimization regarding total execution times, optimiza tion times, and workload adaptation properties. We com pare it with periodic re optimization and unoptimized execution (i.e., the initial optimal plan). To summarize, the major results are

- Execution time improvements increase with increasing workload dynamics due to immediate adaptation.
- We gain optimization time improvements for static workloads due to no unnecessary re optimizations.
- Overheads for statistic maintenance, monitoring optim ality, and `PlanOptTree` algorithms are negligible.
- Directed re optimization benefits increase with the plan size and the complexity of optimization.

6.1. Experimental setting

We ran our experiments on an IBM blade LS20 with two AMD Opteron 270 processors, each a 2 GHz Dual Core, and 9 GB RAM, where we used Linux openSUSE 9.1 (32 bit) as the operating system. Our WFPE (workflow process engine) is a prototype integration platform including our cost based optimizer. The WFPE is implemented using Java 1.6 as the programming language and consists of approxi mately 37,000 lines of code. It includes several inbound and outbound adapters for the interaction with external systems, where we currently support files, databases, and Web services.

The test integration flows are four plans with different characteristics: Plan P_1 with $m=9$ is our simple running example shown in Fig. 1. Plan P_2 with $m=19$ is more complex, where we receive messages, load data from four systems, apply schema transformations, join all datasets (clique query type) and finally send the results to another system. The related optimality conditions are shown in Fig. 9. For both plans, the benchmark drivers invoke synchronous inbound adapters and we use file outbound adapters as external systems in order to reduce external influences. Furthermore, we use two additional plans P_3 and P_4 (described in detail later on), where we vary the number of operators up to 100 in order to investigate the influence of plan sizes. All experiments use synthetic data because we want to generate workloads with different selectivities and cardinalities. Any relative time improve ments are specified as $1 \quad t_2/t_1$, where t_1 represent the baseline, and thus are upper bounded by 100%.

Our default parameters are as follows. Both optimiza tion models use `EMA` ($\alpha = 0.5$) for statistics aggregation and a relative cost threshold of $\tau_2 = 0.0$. For clarity of pre sentation, we disabled all optimization techniques except selection and join reordering. The periodic re optimization interval is $\Delta t = 5$ min. On demand uses a minimal exis tence time of $\Delta t = 1$ s, a lazy condition count of $\tau_1 = 10$, and our `MEMO` structure.

6.2. End to end overall comparison

In a first series of experiments, we investigate the major characteristics of on demand and periodic re optimization as well as unoptimized execution in terms of their total execution times and optimization times.

Scenario setup: We use the described plans P_1 and P_2 and execute different workload scenarios, each with $n = 100{,}000$ plan instances but different dynamics in terms of the number of workload shifts ($|ws|$): low ($|ws| = 1$), med ($|ws| = 10$), and high ($|ws| = 100$). We use uniform data distributions for all scenarios. For plan P_1, we use an input data size (per plan instance) of 400 KB and selectiv ities of $\{1.0, 0.8, 0.1\}$ for the three `Selection` operators. A single experiment thus executed P_1 on 38.1 GB of input data. For plan P_2, we use an input data size of 167 KB and cardinalities of {200 KB, 67 KB, 100 KB, 33 KB} for the four loaded datasets in order to make plan execution times comparable to P_1. Workload shifts are realized by shifting the selectivities (P_1) or cardinalities (P_2) round robin to the front (e.g., $sel(o_1) = sel(o_2)$, etc.), done every $n/|ws|$ plan instances. By shifting selectivities/cardinalities $|ws|$ times, the workload dynamics include both different impact and frequency of workload shifts. The results are shown in Fig. 10.

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

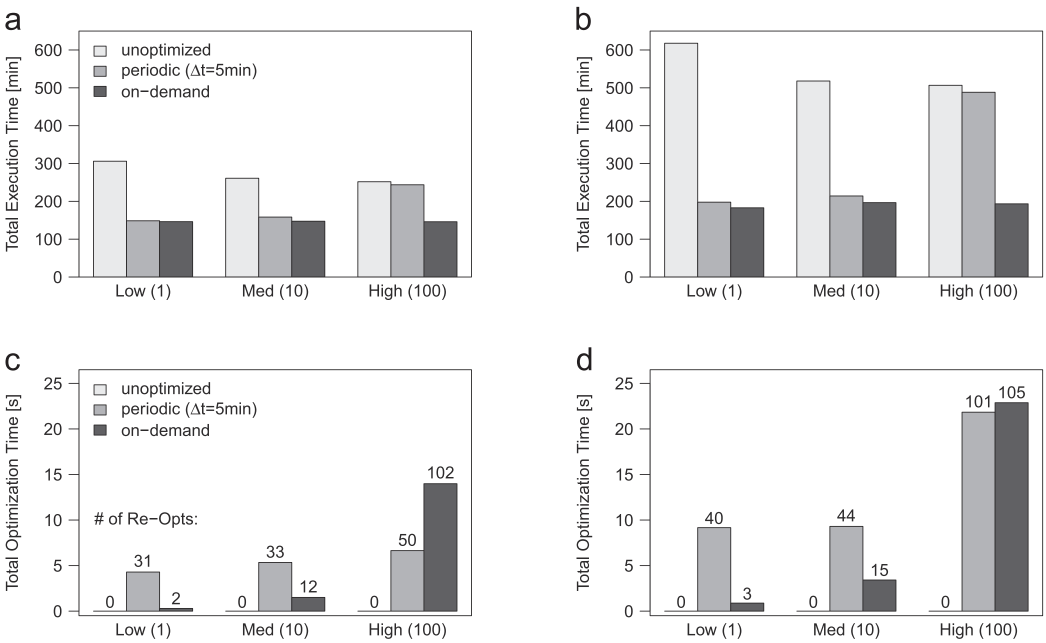

Fig. 10. Overall comparison results with changing workload dynamics. (a) exec plan P_1, (b) exec plan P_2, (c) re-opt plan P_1, and (d) re-opt plan P_2.

Execution time: Fig. 10(a) and (b) presents the most important results in the form of total execution times that already include all monitoring and optimization over heads. The runtime for unoptimized execution is not constant because the initially bad plan becomes optimal[3] from time to time. For static to medium dynamics, on demand achieves only slight improvements because there are only few workload shifts and thus total adaptation delays are low. However, for dynamic workloads, periodic degenerates to unoptimized. In contrast, on demand shows almost constant execution time. The slight increase for high dynamics is reasoned by the lazy count of $\tau_1 = 10$ per workload shift. The relative benefits depend on the optimization techniques and workload characteristics. For example, in our dynamic scenarios, we achieved improve ments of 40.0% for plan P_1 and 60.4% for plan P_2.

Optimization time: In addition, Fig. 10(c) and (d) shows the related total optimization times and number of re optimization steps. Unoptimized does not exhibit these overheads. Although periodic optimization uses a fixed optimization interval Δt, we observe increasing optimiza tion times with increasing workload dynamics because the resulting number of re optimization steps directly depends on the total execution time. For on demand, the number of re optimizations is almost equal to the number of workload shifts. The additional steps are caused by initial optimization and smoothed statistics that led to multiple re optimizations for some workload shifts. In these scenarios, we benefit only slightly from directed re optimization due to a rather small search space. However, we reduced the total optimization time for static workloads by 93.1% for plan P_1 and by 90.4% for plan P_2.

6.3. *Workload adaptation in depth*

In a second series of experiments, we now have a more detailed look at workload adaptation in specific scenarios. The purpose is to quantify adaptation properties rather than an overall performance comparison as discussed before.

6.3.1. *Simple plan scenario*

Scenario setup: Scenario A consists of $n = 100{,}000$ plan instances of plan P_1 and compares periodic and on demand re optimization. We varied the selectivities of the three `Selection` operators as shown in Fig. 11(a). The input data was generated without correlations and we varied its cardinality with $\{1,3,4,5,5,2,1,3,3,3\}$ (in 100 KB). There are four workload shifts (ws_1, ws_2, ws_3, and ws_4), where crossing selectivities cause new optimal plans.

Runtime results: Fig. 11(c) shows the smoothened and sampled execution times of periodic and on demand. Besides the cardinality dependent execution times, we observe adaptation delays for periodic, where we miss optimization opportunities (ws_2 and ws_3). The workload shifts ws_1 and ws_4 have only minor influence due to unchanged minimum operator selectivity (o_2). In contrast, on demand immediately adapts plans, which led to a cumulative execution time improvement of 2.5%. Fig. 11(e) shows the optimization time (incl. recompilation, etc.) per re optimization step. We see that periodic triggers optimization with fixed interval. It took 25 steps in this scenario. In contrast, on demand was only triggered if necessary (at workload shifts) such that we only required

[3] The runtime for unoptimized execution is decreasing with increasing workload dynamics because the initially very bad plan is used less often for dynamic workloads. For example, the initial plan of P_1 is used for 50% of plan instances on low, 36% on med, and 34% on high.

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

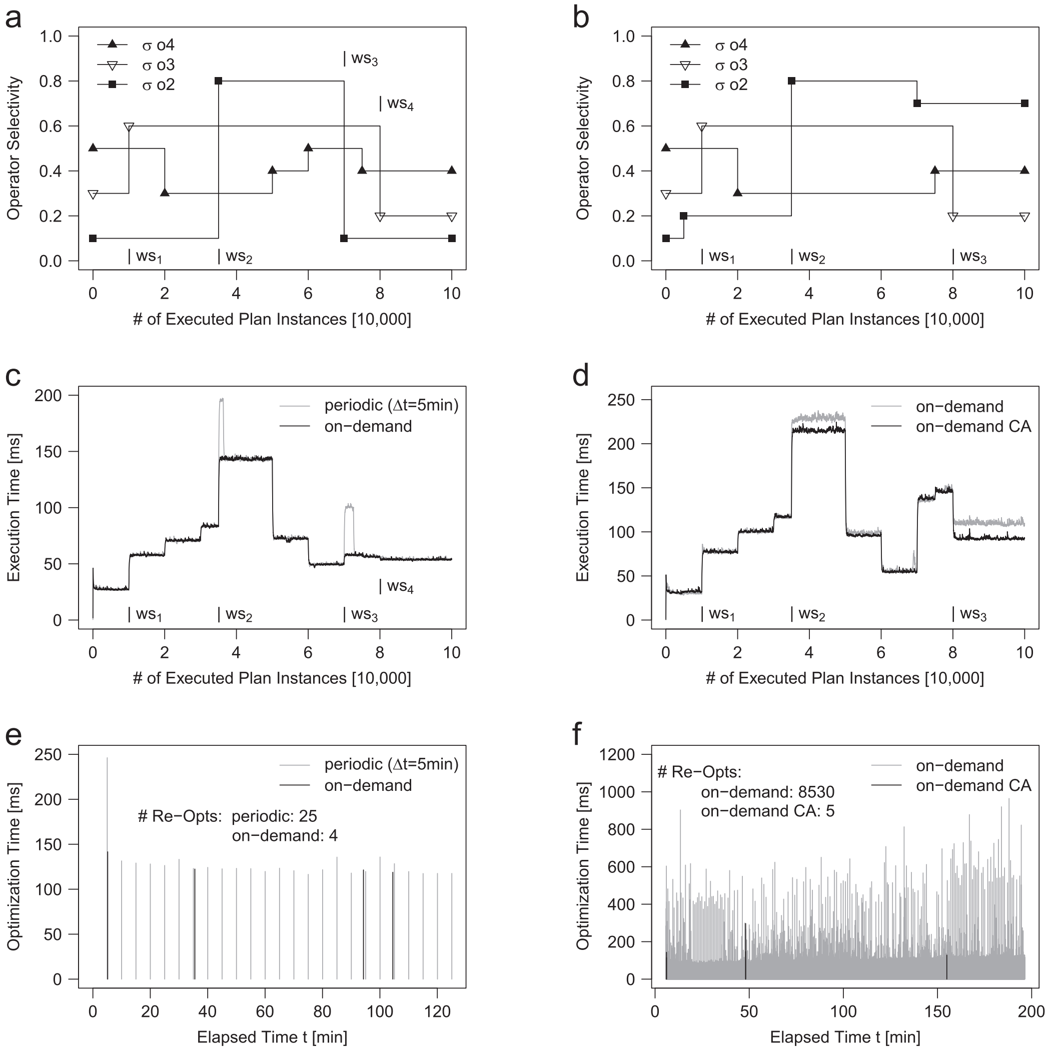

Fig. 11. Simple-plan workload adaptation scenario (without/with correlation). (a) Workload scenario A, (b) workload scenario B, (c) execution time A, (d) execution time B, (e) re-optimization time A, and (f) re-optimization time B.

Table 2
Overhead statistics/condition monitoring.

Traditional	Minimal	POT c_1	POT c_2	POT c_3
159 ms	73 ms	172 ms	314 ms	470 ms

four steps and achieved a cumulative optimization time improvement of 84.3%. Single directed re optimizations are not significantly faster than full re optimizations due to the small search space. The high execution and optimi zation times at the beginning are caused by Java just in time compilation.

Statistic maintenance overhead: Table 2 shows the statis tics maintenance and monitoring overhead of on demand. For that we investigate the algorithm A IS in detail. We use the operator statistic trace from Scenario A that consists of 2,100,000 atomic statistics. Traditional is the baseline (as used for periodic re optimization), where all statistics of all operators are maintained. Minimal refers to a hypothetical scenario, where we know the required statistics and maintain only these. The relative difference between both is the benefit we gain by maintaining only relevant statistics. In contrast, POT monitoring includes the overhead of our `PlanOptTree`. Although the A IS algo rithm declines unnecessary statistics, it is slower than Traditional because for each statistic tuple, we compute the hierarchy of complex statistics and evaluate optimality conditions. We distinguish three configurations: c_1 refers to statistic maintenance without condition evaluation, while c_2 and c_3 (without/with the `MEMO` structure) show the small absolute overhead of continuously monitoring optimality. In this scenario, using the `MEMO` structure is slightly slower due to simple optimality conditions. To summarize, the overhead for statistic maintenance and monitoring at operator granularity is negligible compared to the overall execution time of Scenario A (see Fig. 11(c)).

PlanOptTree algorithm overhead: We also investigated the other `PlanOptTree` algorithms for (1) `PlanOptTree` creation (A PC), (2) triggering re optimization (A TR), and (3)

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

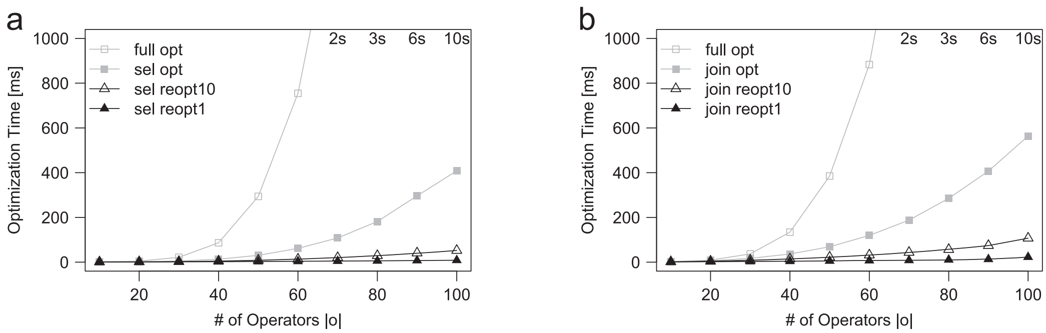

Fig. 12. Directed re-optimization results. (a) Selection reordering P_3 and (b) join reordering P_4.

`PlanOptTree` updating (A UP). For this experiment, we varied the number of `Selection` operators of plan P_1 up to $m = 35$ operators, generated random statistics, and for A TR/A UP, we forced one violated condition. Even for $m = 35$, the mean execution time of 100 repetitions was 0.57 ms (A PC), 0.02 ms (A TR), and 0.21 ms (A UP). Hence, these overheads are also negligible compared to the overall opti mization time in Scenario A (see Fig. 11(e)).

All optimization techniques: On demand, with selection reordering enabled, reduced the total unoptimized execu tion time of Scenario A from 151.7 min to 123.9 min. With all of our cost based optimization techniques enabled (e.g., batched execution, parallel flows, message pipelining), we reduced this total execution time even to 69.2 min. Accordingly, the relative improvements of on demand to periodic re optimization increased almost linearly.

6.3.2. *Simple plan scenario with correlation*

In the interest of a fair evaluation, we now investigate a possible limitation of on demand re optimization: the problem of frequent plan changes caused by correlation.

Scenario setup: Scenario B executes again $n = 100{,}000$ instances of plan P_1. We use the input cardinalities of Scenario A but generated correlated data. Fig. 11(b) shows the condi tional selectivities $sel(o_2)$, $sel(o_3|o_2)$, $sel(o_4|o_2 \wedge o_3)$, while we set $sel(o_3|\neg o_2) = 1.0$ and $P(o_4|\neg o_2 \vee \neg o_3) = 1.0$. Hence, there are strong correlations and the selectivities $sel(o_3)$ and $sel(o_4)$ depend on the operator ordering[4] (only ws_2 and ws_3 are real workload changes). We compare on demand and on demand CA (with correlation table, see Section 4.3).

Runtime results: Fig. 11(d) and (f) shows again the execu tion and optimization times. Without the correlation table selection reordering changes the plan back and forth (up from ws_1), even in case of a constant workload because it wrongly assumes statistical independence. Due to immediate re optimization and the small minimal existence time of $\Delta t = 1$ s, we executed 8,530 re optimization steps. In addi tion, the permanent change between suboptimal and optimal plans led to a degradation of the execution time because non optimal plans were used (e.g., after ws_2 and ws_3). With the correlation table, the number of re optimization steps was reduced to five. There, all three workload shifts have been recognized, where both ws_1 and ws_2 required two re optimizations each. For ws_1, this was due to reverting the wrong decision, while for ws_2, this was due to multiple crossing selectivities, which were learned incrementally via step wise directed re optimization. As a result of the reduced number of re optimization steps and preventing suboptimal plans, we also achieved a 4.4% total execution time improvement.

6.4. *Directed re optimization in depth*

In our third series of experiments, we analyzed the benefit of directed re optimization. We generated plans with varying numbers of `Selection` (P_3) and `Join` (P_4) operators $|o| \in [10, 100]$ and measured full and directed re optimization time. In contrast to previous experiments, the optimizer was tested standalone (without dependency graph maintenance, plan compilation, etc.). We fixed the input cardinalities and randomly generated operator selec tivities. For directed re optimization, we randomly picked $k_1 = 1$ and $k_2 = 10$ operators and generated new statistics for them to violate optimality conditions. All measure ments were repeated 100 times.

Fig. 12 shows the results for the optimization techni ques selection reordering (Fig. 12(a), Section 5.2.1, Plan P_3) and join reordering (Fig. 12(b), Section 5.2.2, Plan P_4). With all optimization techniques enabled, we observe a full optimization time that empirically grows with $\mathcal{O}(n^5)$ (the not shown measurements are annotated at the top right). Here, the optimization time is mainly dominated by the technique of parallel flow rewriting. With all other optimization techniques disabled, the optimization time of both full selection reordering (*sel opt*) and full join reordering (*join opt*) increase still quadratically. For full *re* optimization, these techniques required similar optimi zation times due to dependency checking. In contrast, the directed re optimization times for one (*reopt*1) and ten (*reopt*10) changed operator selectivities (which led to multiple violated optimality conditions) increase almost linearly with the number of plan operators and with the number of violated conditions. Thus, benefits of directed re optimization increase with increasing number of opera tors and with increasing complexity of optimization tech niques. Even for ten changed operators, we gain significant benefits. Comparing selection reordering and join

[4] For example, from ws_3 to the end, we have conditional selectivities of $sel(o_2)$ 0.7, $sel(o_3|o_2)$ 0.2, $sel(o_4|o_2 \wedge o_3)$ 0.4 but total selectivities of $sel(o_2)$ 0.7, $sel(o_3)$ 0.44, $sel(o_4)$ 0.92. This leads to different opera-tor orderings.

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

reordering, the latter took understandably longer but both show the same typical behavior. To summarize, we benefit from the reduced re optimization search space per tech nique and from reconsidering only a subset of techniques.

7. Related work

On demand re optimization belongs to the research field of adaptive query processing (AQP) [17,30] that addresses unknown/mis predicted statistics or changing workloads. AQP also inspired our work but the different runtime model of integration flows requires a different optimization model. We aim at optimizing many consecu tive plan instances of a deployed integration flow, which work on new input data. In the following, we show the relationship of on demand re optimization to important areas and techniques of AQP.

Plan based adaptation in DBMS: Traditional *inter query* optimization aims at optimizing individual query instances. Adaption in this context mainly reduces to the decision when and how to update statistic of the underlying database [17] but optimization happens on granularity of single or multiple concurrent (for sharing opportunities) queries. In contrast, existing work on AQP mainly use *inter operator* or *intra operator* re optimization of single query instances in order to account for mis predicted cardinalities of intermediates. Regarding inter operator, we distinguish *reactive* and *proac tive* approaches [31]. *Reactive*, *inter operator* re optimization uses the traditional optimizer to create a plan, intermediate results are materialized, and if estimation errors are detected, the remaining plan is reactively re optimized. Examples are ReOpt [32] that invokes the optimizer if statistics differ significantly, and Progressive Optimization [33] that uses validity ranges of statistics. In contrast, *proactive*, *inter operator* re optimization proactively creates switchable plans before execution. Rio [31] computes bounding boxes (similar to validity ranges) around all used estimates and then creates robust or switchable plans. During runtime one of three paths can be chosen based on the real statistics (below, estimate, above). Another proactive technique is Parametric Query Optimization (PQO). Due to unknown query para meters during query compile time, PQO [34] and Progressive PQO (PPQO) [35] optimize a query into possible candidate plans and pick the most suitable plan when parameters are bound. *Intra operator* approaches triggered re optimization even during operator runtime. For example, corrective query processing [18] creates new plans for unprocessed data only and the results are combined by stitch up phases. On demand differs in several ways. First, our re optim ization scope is not a single query but the average case of short running plan instances of a deployed plan. Hence, fine grained adaptation (inter/intra operator) based on inter mediate results of a single plan instance is not applicable. Second, we use optimality conditions instead of validity ranges (or bounding boxes). Those validity ranges are defined as absolute cardinalities for subplans, which does not neces sarily mean that the subplan is suboptimal. In contrast, we trigger re optimization only if necessary, i.e., if a new plan is certain to be found and our approach allows for directed re optimization. Furthermore, we do not enumerate alternative plans beforehand but monitor optimality of the current (optimal) plan only.

Adaptation of continuous queries (CQ) in DSMS: CQ based adaptation differs in its re optimization scope of a standing query. Existing work is classified as *routing* or *rewriting based* adaptation. *Routing based* approaches do not rely on predefined plans but route tuples along different stateful operators. An example is Eddies [36,37] with its eddy operator. Dynamic decisions on routing paths via routing policies enable fine grained adaptation [36,38,39] but might incur significant overhead. This overhead can be reduced by routing groups of tuples as done by the self tuning query mesh [19]. For *rewriting based* adaptation, the optimizer requests relevant statistics and re optimization is triggered periodically or on significant changes [30]. Rewriting CQs requires state migration (e.g., tuples in hash tables) [40] to prevent missing tuples, duplicates, or changed tuple orders. Hence, reordering relies on extensive statistic profiling. The Adaptive Greedy algorithm [41] even uses a so called matrix view for conditional selectivity profiling and directed re optimization for the specific technique of reordering stream filters (selection reordering). On demand differs again in several ways. First, the re optimization scope of CQs and integration flows are similar but CQs are stateful that requires state migration on re optimization. Second, in contrast to *passive* structures such as matrix views [41], on demand enables monitoring optimality and directed re optimization for *arbitrary* optimization techniques and actively triggers re optimization that overcomes the need for determining *when* to re optimize.

Adaptation of integration flows: Related work of opti mizing integration flows use *rule based* [11 14], *cost based* [10,13,15,16], and *adaptive cost based* [4,13] approaches. Rule and cost based techniques (optimize once) cannot adapt the plan to changing workloads. Adaptive cost based approaches either use an optimize always model that triggers optimization for each plan instance [13] or peri odic re optimization [4,23] that triggers re optimization with a fixed time interval. On demand achieves near optimal re optimization behavior and thus overcomes the drawbacks of existing adaptive cost based approaches.

8. Conclusions

To summarize, we introduced the concept of on demand re optimization that exploits optimality conditions for re optimization of integration flows. The `PlanOptTree` as a compact representation of optimality conditions enables us to monitor plan optimality during online statistic maintenance and to immediately trigger directed re optimization if the current plan is not optimal. The experiments have shown that on demand re optimization achieves near optimal re optimization behavior in terms of monitoring and re optimization overhead as well as adaptation delays. In conclusion, on demand re optimiza tion perfectly adapts to the dynamics of the current workload, i.e., there are no re optimizations for static but many immediate re optimizations for dynamic workloads. Hence, we benefit from minimized adaptation delays and reduced re optimization efforts. Finally, on demand re optimization is also applicable in other areas. For

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

example, it can be extended for re occurring queries, continuous queries or workflows of MapReduce jobs. In addition, future work might investigate on demand re optimization for specific optimization techniques.

Acknowledgments

We would like to thank Benjamin Schlegel, Ulrike Fischer, Tim Kiefer, and Maik Thiele for valuable comments on earlier versions of this paper.

Appendix A. Analysis of `PlanOptTree` complexity

For certain plan structures and problems, worst case complexity bounds can be guaranteed. Here, we focus on local operator reordering for a sequence of operators. For m operators, there are $m!$ alternative plans.

Proposition 1 (*Local operator reordering* `PlanOptTree` *Complexity*). *Monitoring local optimality of a sequence of m operators has a worst case* `PlanOptTree` *space complexity of* $\mathcal{O}(m^2)$ *nodes and accordingly, the algorithms A IS, A TR, and A UP have a worst case time complexity of* $\mathcal{O}(m^2)$.

Proof. Assume a plan P of a sequence of m operators o. A minimal `PlanOptTree` has at most m ONodes, $m \cdot s$ SNodes, $2 \cdot |oc|$ CSNodes (two complex statistics nodes for each binary optimality condition), and $|oc|$ OCNodes. Each operator can be included at most in one local optimality condition per dependency (in case of a data dependency this subsumes any temporal dependency). Then, an arbi trary operator o_i with $1 \leq i \leq m$ can in the worst case be the target of i 1 dependencies δ_i^-, and it can be the source of m i dependencies δ_i^+. Based on the equivalence of $\delta^- = \delta^+$ and thus, $|\delta^-| = |\delta^+|$, the maximum number of optimality conditions is inherently given by

$$|oc| = \sum_{i \quad 1}^{m} (i \quad 1) = \frac{m(m \quad 1)}{2}. \tag{A.1}$$

The total number of nodes is therefore $m + s \cdot m + 3 \cdot m(m \quad 1)/2$. Since s is a constant, we have $\mathcal{O}(m^2)$ nodes in the worst case. Processed nodes are memoized such that the algorithms A IS, A TR, and A UP access at most $\mathcal{O}(m^2)$ nodes per invocation. Hence, Proposition 1 holds. □

Appendix B. Analysis of directed re-optimization

In the general case, but depending on the `PlanOpt Tree` design per optimization technique, we can give correctness guarantees for directed re optimization. Here, we focus on (1) equivalence to full re optimization and (2) convergence of step wise re optimization, for reordering sequences of operators.

B.1. Directed re optimization

Proposition 2 (*Directed operator recordering equivalence*). *Directed re optimization (re ordering) of all operators* $o' \in P$ *included in violated optimality conditions* $\mathcal{C}'$ *of a* `PlanOptTree` *(for a sequence of m operators o) is equiva lent to the full re optimization of all operators* $o \in P$.

Proof. Assume all dependencies between operators o of plan P to be a directed graph $G = (V, E)$ of vertexes (operators) and edges (dependencies). Then, the re optimization of P is a graph homomorphism $f: G \rightarrow H$. In order to prove Proposition 2, we show that

$$\forall o_i \notin o' : (v_{pre(o_i)} \in G \equiv v_{pre(o_i)} \in H) \wedge (v_{suc(o_i)} \in G \equiv v_{suc(o_i)} \in H), \tag{B.1}$$

where $v_{pre(o_i)}$ denotes the set of predecessors of operator o_i and $v_{suc(o_i)}$ denotes the set of successors of o_i. (1) If there exists a homomorphism $f: G \rightarrow H$ such that

$$v_j \prec o_i \in G \wedge o_i \prec v_j \in H, \tag{B.2}$$

then, the order $v_j \prec o_i$ is represented by an optimality condition oc with $o_i, v_j \in oc$ or by a transitive optimality condition toc with $o_i, v_j \in toc$. The same is true for success of o_i. (2) The `PlanOptTree` allows for arbitrary optimality conditions between operators and input statistics. Hence, during re optimization, $f: G \rightarrow H$, the globally optimal solution will be found. (3) Further, all operators o' included in violated optimality conditions $\forall o_i \in oc'$ with $oc' \in \mathcal{C}'$ or transitive optimality conditions $\forall o_i \in toc'$ with $toc' \in \mathcal{C}'$ are used by $f: G \rightarrow H$. As a result,

$$\nexists (o_i \notin o' \wedge ((v_{pre(o_i)} \in G \neq v_{pre(o_i)} \in H) \vee (v_{suc(o_i)} \in G \neq v_{suc(o_i)} \in H))), \tag{B.3}$$

such that both directed re optimization and full re optimi zation results in the same plan. Hence, Proposition 2 holds. □

B.2. Step wise directed re optimization

Proposition 3 (*Step wise directed re optimization conver gence*). *For optimization problems* $\min \hat{W}(P)$ *with a single minimum, step wise directed re optimization converges to the same plan* P' *as directed re optimization if the workload* ω *is static in the time interval* $[T_1, T_2]$ *with* $\nexists ws_i \in [T_1, T_2]$.

Proof. Assume a finite plan search space $\mathcal{S}$ and exact runtime statistics. (1) For an optimization problem with a single minimum, we have

$$P' = \arg \min_{\forall P \in \mathcal{S}} \hat{W}(P) = \operatorname*{opt}_{\forall P \in \mathcal{S}} (P), \tag{B.4}$$

independent of the optimization start point plan P because for problems with a single minimum, there is by definition only one local optimum and hence it is also the global optimum. (2) Given a static workload ω in the time interval $[T_1, T_2]$ with $\nexists ws_i \in [T_1, T_2]$ directly implies that the optimal plan $P' = \arg\min_{\forall P \in \mathcal{S}} \hat{W}(P)$ is constant in $[T_1, T_2]$. (3) By definition of the `PlanOptTree`, any partial re optimization step addresses at least one optimality condition oc_i and all of its transitive optimality conditions $toc(oc_i)$. Each partial re optimization $P'_{oc_i} = \text{opt}_{oc_i}(P)$ reduces the plan costs with $\hat{W}(P'_{oc_i}) < \hat{W}(P)$. Thus, no cycles are possible. Re optimization steps are triggered as long as at least one optimality condition oc_i is violated. Without loss of generality, assume $T_1 = 0$ and $T_1 = \infty$. Then, we can conclude that step wise directed re optim;ization in the

Final edited form was published in "Information Systems". 45. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

finite search space $\mathcal{S}$ converges to $P' = \mathrm{argmin}_{\forall P \in \mathcal{S}} \hat{W}(P)$. Hence, Proposition 3 holds. □

Given our monotonic operator cost function (see Section 2.2), any plan optimization problem $\min \hat{W}(P)$ can be transformed into an optimization problem with a single minimum by adding new optimality conditions, i.e., by conceptually transforming it into a higher dimensional space. However, this property clearly depends on the specific `PlanPlanOptTree` design.

Appendix C. Supplementary data

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.is.2014.03.005.

References

[1] L.M. Haas, Beauty and the beast: the theory and practice of information integration, in: ICDT, 2007, pp. 28–43.
[2] P.A. Bernstein, L.M. Haas, Information integration in the enterprise, Commun. ACM 51 (9) (2008) 72–79.
[3] A.Y. Halevy, N. Ashish, D. Bitton, M.J. Carey, D. Draper, J. Pollock, A. Rosenthal, V. Sikka, Enterprise information integration: successes, challenges and controversies, in: SIGMOD, 2005, pp. 778–787.
[4] M. Boehm, Cost-based optimization of integration flows (Ph.D. thesis), TU Dresden, Available at ⟨http://wwwdb.inf.tu-dresden.de/boehm/diss_final.pdf⟩, 2011.
[5] M. Stonebraker, Too much middleware, SIGMOD Rec. 31 (1) (2002) 97–106.
[6] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2004.
[7] U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson, Data integration flows for business intelligence, in: EDBT, 2009, pp. 1–11.
[8] W. O'Connell, Extreme streaming: business optimization driving algorithmic challenges, in: SIGMOD, 2008, pp. 13–14.
[9] R. Winter, P. Kostamaa, Large scale data warehousing: trends and observations, in: ICDE, 2010, p. 1.
[10] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, QoX-driven ETL design: reducing the cost of ETL consulting engagements, in: SIGMOD, 2009, pp. 953–960.
[11] A. Behrend, T. Joerg, Optimized incremental ETL jobs for maintaining data warehouses, in: IDEAS, 2010, pp. 1–9.
[12] M. Bhide, M. Agarwal, A. Bar-Or, S. Padmanabhan, S. Mittapalli, G. Venkatachaliah, XPEDIA: XML processing for data integration, Proc. VLDB Endow. 2 (2) (2009) 1330–1341.
[13] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: VLDB, 2006, pp. 355–366.
[14] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, T. Kraft, An approach to optimize data processing in business processes, in: VLDB, 2007, pp. 615–626.
[15] A. Simitsis, P. Vassiliadis, T.K. Sellis, Optimizing ETL processes in data warehouses, in: ICDE, 2005, pp. 564–575.
[16] A. Simitsis, K. Wilkinson, U. Dayal, M. Castellanos, Optimizing ETL workflows for fault-tolerance, in: ICDE, 2010, pp. 385–396.
[17] A. Deshpande, Z.G. Ives, V. Raman, Adaptive query processing, Found. Trends Databases 1 (1) (2007) 1–140.
[18] Z.G. Ives, A.Y. Halevy, D.S. Weld, Adapting to source properties in processing data integration queries, in: SIGMOD, 2004, pp. 395–406.
[19] R.V. Nehme, E.A. Rundensteiner, E. Bertino, Self-tuning query mesh for adaptive multi-route query processing, in: EDBT, 2009, pp. 803–814.
[20] P. Bodík, A. Fox, M.J. Franklin, M.I. Jordan, D.A. Patterson, Characterizing, modeling, and generating workload spikes for stateful services, in: SoCC, 2010, pp. 241–252.
[21] T. Rabl, A. Lang, T. Hackl, B. Sick, H. Kosch, Generating shifting workloads to benchmark adaptability in relational database systems, in: TPCTC, 2009, pp. 116–131.
[22] S. Kavalanekar, B.L. Worthington, Q. Zhang, V. Sharda, Characterization of storage workload traces from production windows servers, in: IISWC, 2008, pp. 119–128.
[23] M. Boehm, D. Habich, W. Lehner, U. Wloka, Workload-based optimization of integration processes, in: CIKM, 2008, pp. 1479–1480.
[24] N. Reddy, J.R. Haritsa, Analyzing plan diagrams of database query optimizers, in: VLDB, 2005, pp. 1228–1240.
[25] R.H. Byrd, P. Lu, J. Nocedal, C. Zhu, A limited memory algorithm for bound constrained optimization, SIAM J. Sci. Comput. 16 (5) (1995) 1190–1208.
[26] N. Bruno, C.A. Galindo-Legaria, M. Joshi, Polynomial heuristics for query optimization, in: ICDE, 2010, pp. 589–600.
[27] G. Moerkotte, Building Query Compilers, Available at ⟨http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf⟩, 2009.
[28] M. Boehm, D. Habich, S. Preissler, W. Lehner, U. Wloka, Cost-based vectorization of instance-based integration processes, Inf. Syst. 36 (1) (2011) 3–29.
[29] M.W. Blasgen, J. Gray, M.F. Mitoma, T.G. Price, The convoy phenomenon, SIGOPS Operat. Syst. Rev. 13 (2) (1979) 20–25.
[30] S. Babu, P. Bizarro, Adaptive query processing in the looking glass, in: CIDR, 2005, pp. 238–249.
[31] S. Babu, P. Bizarro, D.J. DeWitt, Proactive re-optimization, in: SIGMOD, 2005, pp. 107–118.
[32] N. Kabra, D.J. DeWitt, Efficient mid-query re-optimization of sub-optimal query execution plans, in: SIGMOD, 1998, pp. 106–117.
[33] V. Markl, V. Raman, D.E. Simmen, G.M. Lohman, H. Pirahesh, Robust query processing through progressive optimization, in: SIGMOD, 2004, pp. 659–670.
[34] Y.E. Ioannidis, R.T. Ng, K. Shim, T.K. Sellis, Parametric query optimization, in: VLDB, 1992, pp. 103–114.
[35] P. Bizarro, N. Bruno, D.J. DeWitt, Progressive parametric query optimization, IEEE Trans. Knowl. Data Eng. 21 (4) (2009) 582–594.
[36] R. Avnur, J.M. Hellerstein, Eddies: continuously adaptive query processing, in: SIGMOD, 2000, pp. 261–272.
[37] S. Madden, M.A. Shah, J.M. Hellerstein, V. Raman, Continuously adaptive continuous queries over streams, in: SIGMOD, 2002, pp. 49–60.
[38] P. Bizarro, S. Babu, D.J. DeWitt, J. Widom, Content-based routing: different plans for different data, in: VLDB, 2005, pp. 757–768.
[39] F. Tian, D.J. DeWitt, Tuple routing strategies for distributed eddies, in: VLDB, 2003, pp. 333–344.
[40] Y. Zhu, E.A. Rundensteiner, G.T. Heineman, Dynamic plan migration for continuous queries over data streams, in: SIGMOD, 2004, pp. 431–442.
[41] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, J. Widom, Adaptive ordering of pipelined stream filters, in: SIGMOD, 2004, pp. 407–418.