Cost-Based Optimization of Integration Flows

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Wirt.-Inf. (FH) Matthias Böhm
geboren am 25. Juni 1982 in Dresden

Gutacher: **Prof. Dr.-Ing. Wolfgang Lehner**
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr.-Ing. habil. Bernhard Mitschang
Universität Stuttgart
Fakultät Informatik, Elektrotechnik
und Informationstechnik
Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
70569 Stuttgart

Tag der Verteidigung: 15. März 2011

Dresden im April 2011

Abstract

Integration flows are increasingly used to specify and execute data-intensive integration tasks between several heterogeneous systems and applications. There are many different application areas such as (near) real-time ETL (Extraction Transformation Loading) and data synchronization between operational systems. For the reasons of (1) an increasing amount of data, (2) typically highly distributed IT infrastructures, and (3) high requirements for data consistency and up-to-dateness, many instances of integration flows—with rather small amounts of data per instance—are executed over time by the central integration platform. Due to this high load as well as blocking synchronous source systems or client applications, the performance of the central integration platform is crucial for an IT infrastructure. As a result, there is a need for optimizing integration flows. Existing approaches for the optimization of integration flows tackle this problem with rule-based optimization in the form of algebraic simplifications or static rewriting decisions during deployment. Unfortunately, rule-based optimization exhibits two major drawbacks. First, we cannot exploit the full optimization potential because the decision on rewriting alternatives often depends on dynamically changing costs with regard to execution statistics such as cardinalities, selectivities and execution times. Second, there is no re-optimization over time and hence, the adaptation to changing workload characteristics is impossible. In conclusion, there is a need for adaptive cost-based optimization of integration flows.

This problem of cost-based optimization of integration flows is not as straight-forward as it may appear at a first glance. The differences to optimization in traditional data management systems are manifold. First, integration flows are reactive in the sense that they process remote, partially non-accessible data that is received in the form of message streams. Thus, proactive optimization such as dedicated physical design is impossible. Second, there is also the problem of missing knowledge about data properties of external systems because, in the context of loosely coupled applications, statistics are non-accessible or do not exist at all. Third, in contrast to traditional declarative queries, integration flows are described as imperative flow specifications including both data-flow-oriented and control-flow-oriented operators. This requires awareness with regard to semantic correctness when rewriting such flows. Additionally, further integration-flow-specific transactional properties such as the serial order of messages, the cache coherency problem when interacting with external systems, and the compensation-based rollback must be taken into account when optimizing such integration flows. In conclusion, the cost-based optimization of integration flows is a hard but highly relevant problem in today's IT infrastructures.

In this thesis, we introduce the concept of cost-based optimization of integration flows that relies on incremental statistics maintenance and inter-instance plan re-optimization. As a foundation, we propose the concept of periodical re-optimization and present how to integrate such a cost-based optimizer into the system architecture of an integration platform. This includes integration-flow-specific (1) prerequisites such as the dependency analysis and a cost model for interaction-, control-flow- and data-flow-oriented operators as well as (2) specific statistic maintenance strategies, optimization algorithms and optimization techniques. While this architecture was inspired by cost-based optimizers

of traditional data management systems and the included optimization techniques focus on execution time minimization only, we additionally introduce two novel cost-based optimization techniques that are tailor-made for integration flows, which both follow the optimization objective of throughput maximization. First, we explain the concept of cost-based vectorization of integration flows in order to optimally leverage pipeline parallelism of plan operators and thus, increase the message throughput. Second, we discuss the concept of multi-flow optimization via horizontal message queue partitioning that increases throughput by executing operations on message partitions instead of on individual messages and thus, it reduces work of the integration platform such as the costs for querying external systems. Finally, the major drawbacks of periodical re-optimization are (1) many unnecessary re-optimization steps, where we find a new plan, only if workload characteristics have changed, and (2) adaptation delays after a workload change, where we use a suboptimal plan until re-optimization and miss optimization opportunities. Therefore, we refine the re-optimization approach from periodical re-optimization to on-demand re-optimization, where only necessary statistics are maintained and re-optimization is immediately triggered only if a new plan is certain to be found.

The positive consequences of the cost-based optimization of integration flows are, in general, (1) the continuous adaptation to dynamically changing workload characteristics and (2) performance improvements in the sense of minimizing execution times and maximizing message throughput by exploiting the full optimization potential of rewriting decisions. In particular, the parameterless on-demand re-optimization achieves a fast but robust adaptation to changing workload characteristics with minimal overhead for incremental statistics maintenance and directed re-optimization. Finally, this cost-based optimization framework of integration flows can be used for investigating additional integration-flow-specific optimization techniques. Those optimizations are strongly needed in order to meet the continuously increasing performance requirements on integration platforms.

Acknowledgments

First of all, I'd like to thank my advisor Wolfgang Lehner for always being a challenging counterpart. Wolfgang accomplished to create an inspiring twofold working atmosphere. On the one side, this atmosphere was coined by independence and freedom of research. On the other side, by playing the role of an advocatus diaboli, he constantly motivated me to always doing a bit more than necessary. Although sometimes I felt like being thrown in at the deep end, eventually, I've grown with the given challenges. Putting it all together, I'm really thankful for his supervision.

I'm also deeply grateful to my co-advisor Dirk Habich. He always had time for discussions concerning my research and all the sideway problems. I've learned a lot from the joint writing of many research papers. Most importantly, he was just a great colleague. Beside the mentioned scientific supervision, I'm also thankful to Uwe Wloka who made the initial cooperative research project, between TU Dresden and HTW Dresden, possible by creating the contact to Wolfgang Lehner and by getting the initial funds. Thanks for always trusting me and for giving me the opportunity of independent research. Finally, I also want to thank Bernhard Mitschang for co-refereeing this thesis.

One of the most important aspects in workaday life at the university were great colleagues at the database technology group. In particular, I thank Ulrike Fischer, Benjamin Schlegel, Philipp Rösch, Frank Rosenthal, Simone Linke, and Steffen Preißler, with which I had the pleasure to share the office (in reverse chronological order), for extensive discussions, collaborations and simply for being great colleagues and friends. Special thanks go to Benjamin, Ulrike and Dirk for invaluable comments, suggestions and proof-reading this thesis. Furthermore, I'd like to express my gratitude to Simone who helped me a lot in revising many research papers regarding spelling and grammar as well as for proof-reading parts of this thesis. I'm also thankful to the other colleagues Anja, Bernd, Bernhard, Eric, Gunnar, Hannes, Henri, Henrike, Ines, Katrin, Lars, Maik, Martin, Peter, Philipp, Rainer, Thomas, and Tim for a good atmosphere at the research group. Furthermore, I thank my students who were involved in related research projects. Beside the colleagues from the university, I also want to thank Jürgen Bittner, Torsten Uhr, and Holger Rehn from the SQL Project AG. During my diploma thesis and a subsequent employment at the SQL Project AG, they introduced me into the field of enterprise application integration (EAI) and gave me great insights into the design and implementation of the commercial integration platform TransConnect®. Thanks for many discussions and suggestions.

Finally but probably most importantly, I am deeply thankful to my family and friends. My parents Roswitha and Hans-Hendrik as well as my siblings Claudia and Michael always supported me and my goals. Thanks to Hans-Hendrik and Claudia for thoughtful suggestions and proof-reading this thesis. Without my family and a group of close friends who have shown great understanding of this time full of privations, the time of writing would have been much harder if not impossible.

Matthias Böhm
Dresden, December 3, 2010

Contents

1 Introduction

I can't change the direction of the wind,
but I can adjust my sails to always reach my destination.

— Jimmy Dean

Advances in information technology combined with rising business requirements lead to a rapidly growing amount of digital information created and replicated worldwide. For example, recent studies [IDC08, IDC10] conducted by the market research organization IDC, report a yearly information increase by over 60% to 1.2 zettabyte (ZB) in 2010. Beside this huge amount of digital information, due to technical and organizational issues, the information is distributed across numerous heterogeneous systems, applications and small devices [IDC08, Haa07]. For these reasons, the integration of heterogeneous systems and applications becomes more and more crucial for an IT infrastructure. In another recent study [Gar09], the market research organization Gartner reveals that the revenue of the worldwide application infrastructure and middleware market increased by 6.9% from 14.17 billion USD in 2007 to 15.15 billion USD in 2008. In conclusion, the integration of heterogeneous systems is seen as one of the biggest and most cost-extensive challenges information technology faces today [BH08].

Historically, the predominant integration approaches were materialized and virtually integrated systems [DD99], where both types provide a homogeneous global view over data of several source systems. Nowadays, we observe emerging requirements of complex integration tasks that (1) stretch beyond simple read-only applications, (2) involve many types of heterogeneous systems and applications, and (3) require fairly complex procedural aspects. To meet these requirements, typically, imperative integration flows are modeled and executed in order to exchange data between the heterogeneous systems and applications of an IT infrastructure [HAB+05]. There are plenty of application examples for this type of imperative integration flows, including, for instance, enterprise information systems [BH08], health care management [CHX08, GGH00], energy data management [SAP03], financial messaging [MS09], telecommunications [ACG+08], and context-aware mobile applications [CEB+09]. For example, large health care management system infrastructures include up to 20-120 different applications and systems [Mic07] and rely on domain-specific standards for data exchange such as HL/7 (Health Level 7) [hl707] or DICOM (Digital Imaging and Communications in Medicine) [dic09]. The deployed integration flows are repeatedly executed by an integration platform. Examples of such platforms are ETL tools (Extraction Transformation Loading) [KC04], EAI servers (Enterprise Application Integration) [Lin99] or MOM systems (Message-Oriented Middleware) [HW04], which have converged more and more in the past [Sto02, HAB+05] and this trend is expected to continue in the future as well [HAB+05].

From a business perspective, we classify all information systems of an enterprise according to the three levels of an information system pyramid [Sch97] as operational systems, dispositive systems, or strategical systems. In this context, typically, two major use cases

for integration flows exist [Sch01]. First, the horizontal integration describes the integration of systems within one level. A typical example is the integration of operational systems by EAI servers (adapter-based integration of arbitrary systems and applications) or MOM systems (efficient message transport via messaging standards), where every update within an operational system can initiate data synchronization with other operational systems and hence, data is exchanged by transferring many small messages [HW04]. Second, the vertical integration describes the integration across the levels of the information pyramid. The most typical example is the integration of data from the operational systems into the dispositive and strategical systems (data warehouses) by ETL tools. In this context, there is a trend towards operational BI (Business Intelligence) that requires immediate synchronization between the operational source systems and the data warehouse in order to achieve high up-to-dateness for analytical query results [DCSW09, O'C08, WK10]. This requirement is typically addressed with a near-real-time approach [DCSW09], where the frequency of periodical delta load is simply increased, or with data-driven ETL flows, where data changes of the operational source systems are directly propagated to the data warehouse infrastructure as so-called trickle-feeds [DCSW09, SWCD09].

As a result of both horizontal and vertical integration, many independent instances of integration flows are executed over time. In addition to this high load of flow instances, the performance of source systems depends on the execution time of synchronous data-driven integration flows, where the source systems are blocked during execution. For these reasons, there are high performance demands on integration platforms in terms of minimizing execution and latency times. Furthermore, from an availability perspective in terms of the average response times, the performance of synchronous integration flows has also direct monetary influences. For example, Amazon states that just $0.1s$ increase in average response times will cost them 1% in sales [Bro09]. Similarly, Google recognized that just $0.5s$ increase in latency time caused the traffic to drop by a fifth [Lin06]. In consequence, optimization approaches are required.

Existing optimization approaches of integration flows are mainly rule-based in the sense that a flow is only optimized once during the initial deployment. Thus, only static rewriting decisions can be made. Further, the optimization of integration flows is a hard problem with regard to the characteristics of imperative flow specifications with interaction-, control-flow- and data-flow-oriented operators as well as specific transactional properties such the need for preserving the serial order of incoming messages. The advantage is low optimization overhead because optimization is only executed once. However, rule-based optimization has two major drawbacks. First, many optimization opportunities cannot be exploited because rewriting decisions can often only be made dynamically based on costs with regard to execution statistics such as operator execution times, selectivities and cardinalities. Second, it is impossible to adapt to changing workload characteristics, which commonly vary significantly over time [IHW04, NRB09, DIR07, CC08, LSM+07, BMM+04, MSHR02]. This would require rewriting an integration flow in a cost-based manner according to the load of flow instances and specific execution statistics.

Contributions

In order to address the high performance demands on integration platforms and to overcome the drawbacks of rule-based optimization, we introduce the concept of *cost-based optimization of integration flows* with a primary focus on typically used *imperative* in-

tegration flows. In detail, we make the following more concrete contributions that also reflect the structure of this thesis.

- As a prerequisite, Chapter 2 analyzes existing techniques and introduces the notation and examples used throughout this thesis. The literature survey is twofold. On the one side, we review common integration approaches and, more specifically, we discuss the modeling, execution and optimization of integration flows. On the other side, we discuss adaptive query processing techniques to illustrate the state-of-the-art of cost-based optimization in other system categories such as DBMS (Database Management Systems) or DSMS (Data Stream Management Systems).
- In Chapter 3, we explain the novel fundamentals for the cost-based optimization of integration flows that enable arbitrary optimization techniques. The specific characteristics of integration flows in terms of missing statistics, changing workload characteristics, imperative flow specification (prescriptive) and transactional properties have led to the need for fundamentally new concepts. In order to take into account both data-flow- and control-flow-oriented operators, we explain the necessary dependency analysis as well as a hybrid cost model. We introduce the inter-instance, periodical re-optimization that includes the core transformation-based optimization algorithm as well as specific approaches for search space reduction, workload adaptation sensibility, and the handling of correlated data. Finally, we present selected concrete optimization techniques to illustrate the rewriting of flows.
- Subsequently, in Chapter 4, we present the cost-based vectorization of integration flows that is a tailor-made, control-flow-oriented optimization technique. Based on the problems of low resource utilization and specific transactional requirements, this technique computes the optimal grouping of operators to multi-threaded execution buckets in order to achieve the optimal degree of pipeline parallelism with a minimal number of buckets and hence, it maximizes message throughput. We present context-specific rewriting techniques to ensure transactional properties and discuss exhaustive and heuristic computation approaches for certain settings and constraints.
- As a tailor-made data-flow-oriented optimization technique, we introduce the concept of multi-flow optimization in Chapter 5. It is based on the problem of expensive access of external systems. The core idea is to horizontally partition the inbound message queues and to execute flows for partitions of multiple messages. Due to the decreased number of queries to external systems as well as cost reductions for local operators, this results in throughput improvements. In detail, we introduce the partition tree that is used as a physical message queue representation, an approach for creating and incrementally maintaining such partition trees, and related flow rewriting techniques. Furthermore, we extend the defined cost model and present an approach to periodically compute the optimal waiting time for collecting messages in order to achieve the highest throughput, while ensuring maximum latency constraints of individual messages.
- In order to decrease the overhead for statistics monitoring and re-optimization but to adapt to changing workloads as fast as possible, in Chapter 6, we introduce the novel concept of on-demand re-optimization. This includes (1) to model optimality of a plan by its optimality conditions using a so-called Plan Optimality Tree rather than considering the complete search space, (2) to monitor only statistics that are included in these conditions, and (3) to use directed re-optimization if conditions are

violated. We present this concept separately for two reasons: first, the contributions of Chapters 3-5 are easier to understand using the simple model of periodical re-optimization; second, although the concept of on-demand re-optimization has been designed for integration flows, other research fields can also benefit from this general re-optimization approach.

Finally, Chapter 7 concludes this thesis with a summary of achieved results and a discussion of open research aspects.

Structure

Figure 1.1 illustrates the resulting overall structure of the thesis and shows how the mentioned chapters are partitioned into three major parts.

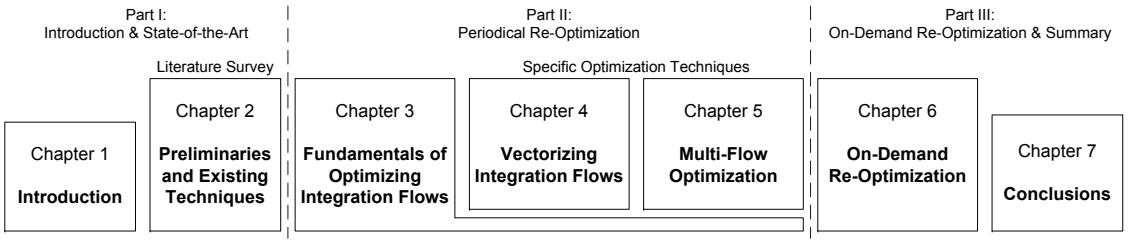


Figure 1.1: Overview of the Structure of this Thesis

In the first part, we give the necessary background in terms of a literature survey. Subsequently, in the second part, we explain the foundations of periodical cost-based re-optimization and discuss two integration-flow-specific optimization techniques in detail. Finally, the third part introduces the advanced concept of on-demand, cost-based re-optimization and how the presented approaches from Part II benefit from this concept.

2 Preliminaries and Existing Techniques

As preliminaries, in this chapter, we survey existing techniques in order to give a comprehensive overview of the state-of-the-art of optimizing integration flows. We start with a classification of integration approaches. Subsequently, we generalize the system architecture of typical integration platforms and review the modeling, execution and optimization of integration flows. Moreover, we also classify approaches of cost-based optimization and adaptive query processing in different system categories. Furthermore, we introduce the used notation of integration flows including their transactional requirements and we define the running example integration flows of this thesis.

2.1 Integration Flows

Integration (from lat. integer = complete) refers to the assembly of many parts to a single composite. In computer science, integration is used in the sense of combining local integration objects (systems, applications, data or functions) with a certain integration technology. For several reasons, which we will reveal in this section, the integration and interoperability of heterogeneous and distributed components, applications, and systems is one of the broadest and most important research areas in computer science.

Historically, database technology itself was introduced with the goal of integrated enterprise data management in the sense of so-called enterprise databases [Bit05]. Unfortunately, the comprehensive and future-oriented database design over multiple application programs has not been achieved. As a result, the current situation is that enterprise data is inherently distributed across many different systems and applications. In this context, we often observe the problems of (1) non-disjointly distributed data across these systems, (2) heterogeneous data representations, and (3) data propagation workflows that are implicitly defined by the applications. Despite the permanent goal of homogeneity and the trend towards homogeneous services and exchange formats, those problems will also remain in the future due to the diversity of application requirements, performance overheads for ensuring homogeneity (e.g., XML exchange formats), continuous development of new technologies that always cause heterogeneity with regard to legacy technologies, as well as organizational aspects such as autonomy and privacy. In consequence, data must be synchronized across those distributed and heterogeneous applications and systems.

This historically reasoned situation of enterprise data management by itself leads to the major goals of integration. First, the distributed and heterogeneous data causes the requirement of *data consolidation and homogeneity* in order to provide a consistent global view over all operational systems. Second, non-disjointly distributed data and implicit data propagation workflows reason the need for *interoperability* in terms of interactions and data synchronization between autonomous systems and applications. Third, in order to achieve *availability* (high availability and disaster recovery) as well as *performance and scalability* (location-based access, load balancing, and virtualization) technically distributed subsystems must be integrated and synchronized as well.

Due to these goals of integration that are based on different application use cases, a variety of integration approaches and realizations emerged in the past. In order to precisely separate the focus of this thesis, we classify those integration approaches, where integration flows are only one category among others.

2.1.1 Classification of Integration Approaches

There exist several, partially overlapping, classification aspects of integration approaches, which include the (1) application area, (2) data location, (3) time aspect and consistency model, (4) event model, (5) topology, and (6) specification method. We explicitly exclude orthogonal integration aspects such as schema matching, model management, master data management, lineage tracing and the integration of semi-structured and unstructured data because techniques from these areas are typically not specifically designed for a certain integration approach.

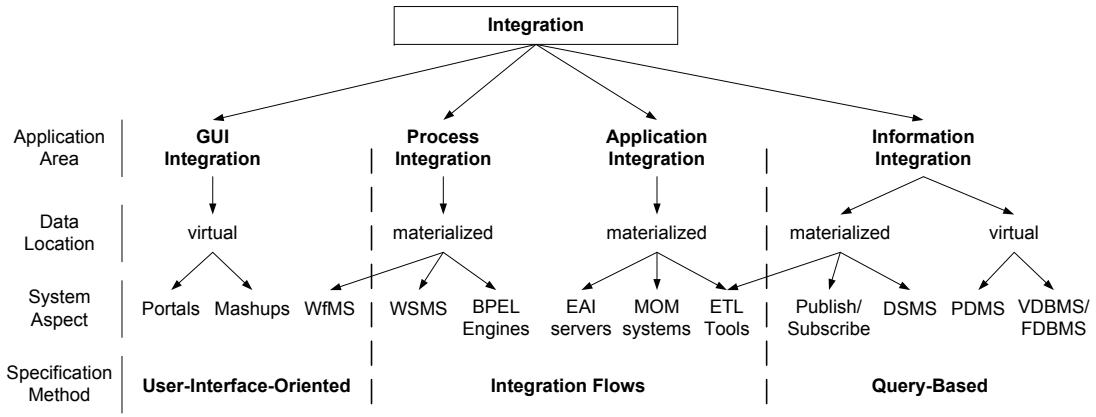


Figure 2.1: Classification of Integration Approaches

Our overall classification that is shown in Figure 2.1 comprises the aspects (1) *application area*, (2) *data location*, and (6) *specification method*. In addition, we put the major types of integration systems (*system aspect*) into the context of this classification. Regarding the application area, we distinguish between information integration (data and function integration), application integration, process integration and GUI integration.

Information integration refers to the area of data-centric integration approaches, where huge amounts of data are integrated. In this context, we typically distinguish between virtual and materialized integration [DD99] based on the data location. First, for virtual integration, a global (virtual) view over the distributed data sources is provided and data is not physically consolidated. Examples for this type of integration approach are virtual DBMS (VDBMS), federated DBMS (FDBMS), and Peer Database Management Systems (PDBMS). The difference is that VDBMS/FDBMS use a hierarchical topology, while PDBMS use a peer-to-peer topology. Typically, both system types provide an event model of ad-hoc queries in order to allow dynamic integration. Second, for materialized integration, the data is physically stored or exchanged with the aim of data consolidation or synchronization. In this context, we mainly distinguish two types of system categories based on their event model. On the one side, DSMS (Data Stream Management Systems) and Publish/Subscribe systems follow a data-driven model where tuples are processed by standing queries or subscription trees, respectively. While DSMS execute many rather

complex queries, Publish/Subscribe systems usually execute huge numbers of fairly simple queries. On the other side, there are ETL tools that follow a time-based or data-driven event model. All three system categories conceptually use a hub-and-spoke or bus topology, where in both cases a central integration platform is used. Finally, all system types of the application area of information integration are *query-based* in the sense of the specification method for integration task modeling. A special case is given by ETL tools that often use integration flows as the specification method as well.

The categories of *application integration* and *process integration* refer to a more loosely coupled type of integration, where *integration flows* are used as specification method and message-oriented flows are hierarchically composed. The main distinction between both is that application integration refers to the integration of heterogeneous systems and applications, while process integration refers to a business-process-oriented integration of homogeneous services (e.g., Web services). Thus, both are classified as materialized integration approaches because messages are propagated and physically stored by the target systems. However, application integration is data-centric in terms of efficiently exchanging data between the involved applications, while process integration is more focused on procedural aspects in the sense of controlling the overall business process and its involved systems. In this context, we see many different facets of system types such as (near) real-time ETL tools (that use the data-driven event model), MOM systems (that use standard-messaging infrastructures such as Java Message Service), EAI systems, BPEL Engines (Business Process Execution Language), and Web Service Management Systems (WSMS). Note that those system categories have converged more and more in the past [Sto02, HAB+05] in the form of overlapping functionalities [Sto02]. For example, standards from the area of process integration such as BPEL are also partially used to specify application integration tasks.

Finally, *GUI integration* (Graphical User Interface) describes the unique and integrated visualization of (or the access to) heterogeneous and distributed data sources. Portals provide a unique system for accessing heterogeneous data sources and applications, where data is only integrated for visualization purposes. In contrast, mashups dynamically compose Web content (feeds, maps, etc.) for creating small applications with a stronger focus on content integration. See the classification by Aumueller and Thor [AT08] for a detailed classification of existing mashup approaches. Both portals and mashups are classified as virtual integration approaches that use a hierarchical topology, and the specification is mainly *user-interface-oriented*.

The exclusive scope of this thesis is the category of integration flows. As a result, the proposed approaches can be applied for process integration, application integration, and partially information integration as well.

2.1.2 System Architecture for Integration Flows

The integration of highly heterogeneous systems and applications that require fairly complex procedural aspects makes it almost impossible to realize these integration tasks using traditional techniques from the area of distributed query processing [Kos00] or replication techniques [CCA08, PGVA08]. In consequence, those complex integration tasks are typically modeled and executed as imperative integration flow specifications.

Based on the specific characteristics of integration flows, a typical system architecture has evolved in the past. This architecture is commonly used by the major EAI products such as SAP eXchange Infrastructure (XI) / Process Integration (PI) [SAP10], IBM

Message Broker [IBM10], MS Biztalk Server [Mic10], or Oracle SOA Suite [Ora10]. We generalize this widely-accepted[1] common architecture to a reference architecture for integration flows in order to survey existing work with regard to modeling, executing, and optimizing integration flows. Furthermore, this system architecture—which is illustrated in Figure 2.2—represents the foundation of this thesis from a system architecture perspective. In detail, two major components are relevant: (1) a modeling component for integration flow specification and (2) a runtime execution component that executes deployed integration flows.

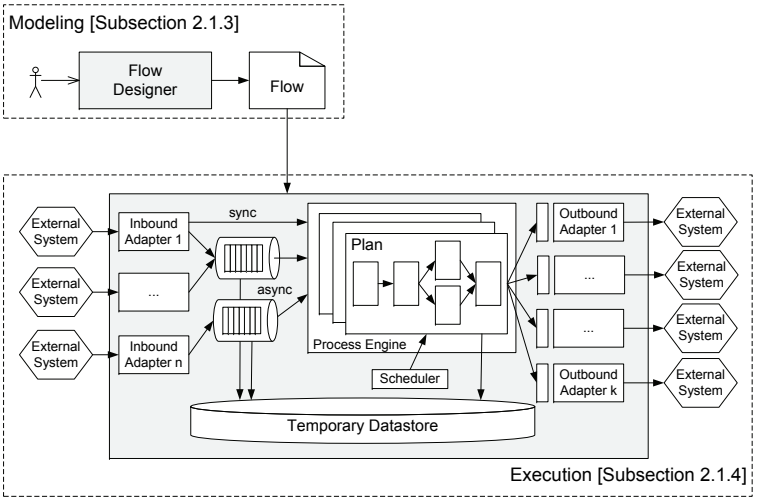


Figure 2.2: Reference System Architecture for Integration Flows

The *modeling component* is used for specifying complex integration tasks by integration flows in the form of graphs or hierarchies of operators and edges. Typically, the configuration and administration of the execution environment is done via this modeling component as well. From a research perspective, we concentrate only on the specification of integration flows and discuss a classification as well as related approaches and languages in Subsection 2.1.3.

The *execution component* consists of a set of inbound adapters, multiple message queues, an internal scheduler (for time-based integration flows), a central process execution engine, a set of outbound adapters, and a temporary data store (persistence of messages and configuration meta data). The inbound adapters listen for incoming messages, transform them into a common format (e.g., XML), and either append the messages to inbound queues (in case of asynchronous flow execution) or directly forward them to the process engine (in case of synchronous flow execution). Typically, those inbound adapters are realized as decoupled listener threads (e.g., a TCP Listener) or standard APIs (e.g., an HTTP implementation). Existing integration platforms provide many different technical and application-level adapters for domain-specific data exchange standards such as HL/7 (Health Level 7) [hl707] or DICOM (Digital Imaging and Communications in Medicine) [dic09] and proprietary system APIs (e.g., SAP R/3). Within the process engine, compiled plans of deployed integration flows are executed. For this purpose, this engine includes a flow parser for the deployment of integration flows and a runtime environment for the

[1] There are plenty of existing application integration platforms. For a detailed survey of the major products, we refer the interested reader to a related study conducted by the market research organization Gartner [TSN+08] and to a survey on SQL support in workflow products [VSRM08].

execution. During execution of these flows, the outbound adapters are used as services (gateways) in order to actively invoke external systems, where they transform the internal format back to the proprietary message representations. The inbound and outbound adapters hide syntactic heterogeneities of external systems, while structural heterogeneities are addressed by schema transformation as part of the plan execution within the process engine. Note that each supported proprietary application, data format or standard protocol requires such specific inbound and outbound adapters. This concept of inbound and outbound adapters in combination with the common internal message format (in terms of XML or other tree-structured data representations) is crucial in order to reduce the complexity of such an execution component for two reasons. First, for n incoming formats and m outgoing formats, we reduced the number of required transformation services (e.g., for transforming an XML message to a binary EDIFACT message) from $n \cdot m$ to $n + m$ because we only need individual mappings (transformations) between the internal format and the sets of incoming and outgoing formats. Second, using the common format, all operators of the process engine need only to be realized once, according to this internal format, and the specific characteristics of external systems are hidden from the process engine by a generic outbound adapter interface.

The deployment of integration flows finally combines both the modeling and execution components. Typically, an XML representation of the modeled integration flow is used in order to transfer the flow specification to the execution component, where the flow parser is used in order to create an internal plan of the integration flow. All analysis and optimization procedures then work on this plan representation. Finally, the plan is compiled into an executable form. In this context, we distinguish several different execution models and approaches that we will discuss in detail in Subsection 2.1.4.

As already mentioned, there are plenty of application examples for imperative integration flows [BH08, CHX08, GGH00, SAP03, MS09, ACG+08, CEB+09]. Due to (1) the high number of heterogeneous systems and applications and (2) the continuous change of technology that reasons heterogeneity, the need for efficient integration will constantly remain in the future.

2.1.3 Modeling Integration Flows

With regard to the overall system architecture, one important aspect is how to model integration flows. Essentially, we classify existing approaches of specification models using the two criteria (1) *flow structure* and (2) *flow semantics*. Figure 2.3 illustrates the resulting classification of integration flow modeling approaches. In contrast to declarative queries that are specified with descriptive languages, imperative integration flows are typically specified using prescriptive languages.

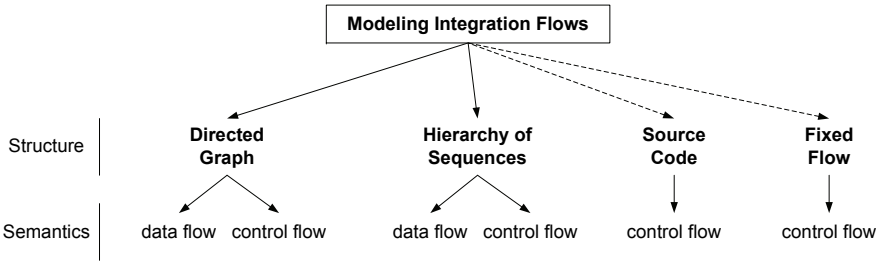
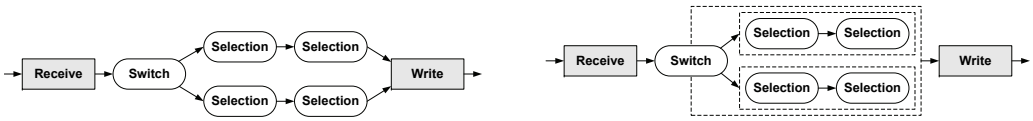


Figure 2.3: Classification of Modeling Approaches for Integration Flows

From the perspective of structural representation, we distinguish four major types. First, there are directed graphs, where the integration flow is modeled with $G = (V, A)$ as sets of vertexes V (operators) and arcs A (dependencies between operators). Typically, such a graph is restricted to directed acyclic graphs (DAG), and XMI (XML Metadata Interchange) [OMG07] is used as technical representation. Common examples of this type are UML (Unified Modeling Language) activity diagrams [OMG03] and BPMN (Business Process Modeling Notation) process specifications [BMI06]. However, many proprietary representations exist as well. Van der Aalst et al. identified common workflow patterns for imperative workflows with directed graph structures in order to classify the expressive power of concrete workflow languages [vdABtHK00, vdAtHKB03]. Second, hierarchies of sequences are a common representation. There, the flow is modeled as a sequence of operators, where each operator can be an atomic or complex (composite of subsequences) operator. Thus, arbitrary hierarchies can be modeled. As the technical representation XML is used due to its suitable hierarchical nature. This is the most common type for representing integration flows because this specification is more restrictive, which allows the automatic compilation into an executable form and all algorithms working on this representation can be realized as simple recursive algorithms. Examples are BPEL [OAS06] processes and XPDL (XML Process Definition Language) [WfM05] process definitions.

Example 2.1. *In order to make this distinction between directed graphs and hierarchies of sequences more understandable, Figure 2.4 shows an example integration flow with both different flow structures. Figure 2.4(a) illustrates an example plan with a directed graph structure, where we receive a message, execute two filters depending on an expression evaluation and finally send the result to an external system. Figure 2.4(b) illustrates the same example with a hierarchy-of-sequences structure. When comparing both, we see that the directed graph uses only atomic operators and allows for arbitrary temporal dependencies between operators, while the hierarchy of sequences require complex operators (e.g., the* `Switch` *operator) that recursively contain other operators and therefore is more restrictive.*



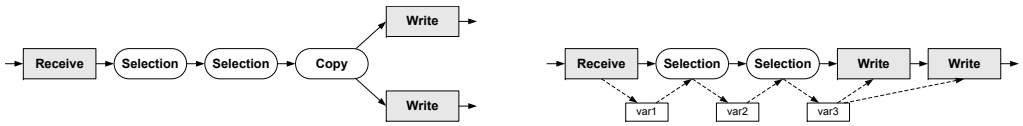
(a) Directed Graph (b) Hierarchy of Sequences

Figure 2.4: Integration Flow Modeling with Directed Graphs and Hierarchies of Sequences

In contrast to directed graphs and hierarchies of sequences, there is a third and a fourth type, both of which are less common nowadays. Some integration platforms allow to model (to program) integration flows on source code level using specific APIs. Examples of this type are JPD (Java Process Definition) [BEA08], BPELJ (BPEL extension for Java) [BEA04], and script-based integration platforms such as pygrametl [TP09]. As an advantage, arbitrary custom code can be used, while the flow designer is confronted with high modeling and optimization efforts. Finally, some platforms use fixed integration flows. A fairly simple example is the concept of so-called message channels, where messages are received, transformed and finally sent to a single external system. Concrete integration flows are modeled by configuring these fixed flows. Although this structure allows for

efficient processing of simple integration tasks, complex integration tasks require many indirections and thus, cannot be realized efficiently or not modeled at all. In addition, from the perspective of flow semantics, we distinguish between data-flow- and control-flow-oriented modeling [MMLW05].

Example 2.2. *Figure 2.5 shows an example integration flow with both different flow semantics (data flow and control flow): Figure 2.5(a) illustrates an example plan with data-flow semantics, where we receive a message, execute two filters and finally, send the result to two external systems. Here, the edges describe data dependencies between operators, while temporal dependencies cannot be specified. For example, we cannot specify in which temporal order to execute the two final writes. A specific characteristic is that if two or more operators require a certain intermediate result, this intermediate result must be copied in order to send it to both operators. In contrast, Figure 2.5(b) illustrates the same example plan using the control-flow semantics. Here, the edges describe temporal dependencies, while data dependencies are implicitly given by input and output variables (for clarity, illustrated as dashed edges). Thus, the semantic of this integration flow additionally includes the execution order of operators and does not require copy operations.*



(a) Data-Flow Semantics (b) Control-Flow Semantics

Figure 2.5: Integration Flow Modeling with Directed Graphs

Examples of integration flow modeling with directed graphs and control-flow semantics are UML activity diagrams [OMG03] and BPMN process specifications [BMI06]. In contrast, directed graphs in combination with data-flow semantics are commonly used by traditional ETL tools. To summarize, control-flow semantics specify an integration flow more precisely than pure data-flow semantics because the control-flow includes the data flow and additional temporal dependencies. However, note that the implicit data flow specification (beside the primary temporal dependencies) can cause semantic data flow modeling errors such as lost or inconsistent data [TvdAS09].

In addition to this classification, further aspects of modeling integration flows—that we will reveal in the following—are currently discussed in the literature. This includes (1) the combination of control-flow and data-flow modeling (hybrid flow semantic), (2) the combination of hierarchical and source code structure (hybrid flow structure), (3) the model-driven development of integration flows, and (4) the declarative flow modeling.

A. Hybrid Flow Semantic

The strict distinction between data-flow and control-flow semantics has been considered as a problem, especially, in the context of data-intensive integration flows that also require rather complex procedural aspects. In consequence, two projects have addressed the combination of data flow and control flow using hybrid modeling semantics, where the data flow is modeled explicitly rather than only by input and output variables.

First, there is the concept of BPEL/SQL, where specific SQL activities can be used within BPEL process specifications in order to combine the advantages of data-flow and

control-flow modeling [MMLW05]. In detail, `SQL` activities, `Retrieve Set` activities, and `Atomic SQL Sequence` activities (as transactional boundary for other `SQL` activities) have been introduced. For example, this concept was implemented as the BPEL extension ii4BPEL (information integration for BPEL) [IBM05b] within the IBM Business Integration Suite that is based on the WebSphere application server. See the survey by Vrhovnik et al. [VSRM08] for a detailed comparison of SQL support by workflow products. To summarize, this BPEL/SQL approach provides hybrid modeling semantics in terms of control flow and data flow. In addition, it enables the efficient execution of data-intensive processes due to potentially reduced transferred data. We will revisit this approach from an optimization perspective in Subsection 2.1.5.

In contrast to the BPEL/SQL activities, the BPEL-DT (Data Transitions) [HRP+07, Hab09] approach uses the traditional control-flow semantics enriched with so-called data transitions. It is based on the concept of data-grey-box Web services [HPL+07] that represent a specific data framework in order to propagate data between Web services by reference. Data transitions are used in terms of data-intensive data dependencies between external Web service interactions within a BPEL process. Such a data transition is modeled and executed as an integration flow using arbitrary integration platforms such as ETL tools. In conclusion, arbitrary combinations of the control-flow-oriented BPEL specification with data-flow-oriented integration flows can be realized.

However, both approaches are still control-flow-oriented because the control-flow-oriented BPEL is enriched with data-centric constructs, but still exhibits temporal dependencies, which is the major classification criterion between data-flow and control-flow semantics.

B. Hybrid Flow Structure

In addition to the mentioned hybrid modeling semantics, there are also arguments for a hybrid modeling structure. For example, when using directed graphs or the hierarchy of sequences, aspects like complex control flow branches, specific variable initialization or the preparation of complex queries to external systems are difficult and complex to model. In contrast, when using source code structures, the overall process specification is hidden and too fine-grained. For these reasons, a combination of both aspects is advantageous for certain integration flows.

BPELJ (BPEL for Java) [BEA04] enables such a hybrid modeling structure by combining the hierarchy of sequences and source code. Therefore, arbitrary Java code snippets (expressions, or small blocks of Java code) can be included in BPEL process specifications. This is done via so-called `bpelj:snippet` activities, where the complete code block is included in this activity and the activity can be used within the overall process specification.

C. Model-Driven Development of Integration Flows

In the context of complex integration flows, we observe a trend towards applying model-driven development techniques from the area of software technology.

First steps towards model-driven development of integration flows were made by Simitsis et al. in the sense of separating ETL flow modeling into conceptual [VSS02a], logical [VSS02b] and physical [TVS07] models as well as using transformation rules [Sim05] to describe the transition from one model to another.

The Orchid project [DHW+08] enables the generation of ETL jobs from declarative schema mapping specifications. A so-called Operator Hub Model (OHM) is used in order

to automatically specify the semantics of ETL jobs in a platform-independent manner. The Orchid project is restricted to the generation of ETL flows. In addition, Jörg and Deßloch addressed the generation of incremental ETL jobs [JD08, JD09] and subsequently, they also investigated the optimization of these incremental ETL jobs [BJ10].

In contrast, the METL project by Albrecht and Naumann focuses on the combination of the generation and model management of ETL flows [AN08]. There, platform-independent and tool-independent operators for model management were introduced. In contrast to generic model management [BM07] for schema mappings, the order, type, and configuration of data transformation steps must be taken into account. These requirements are addressed with specific high-level operators and an ETL management platform [AN09].

In addition to these ETL flow approaches, our GCIP framework (Generation of Complex Integration Processes) [BHLW09a] addresses the modeling of platform-independent integration flows [BHLW08e], the flow generation for arbitrary integration platforms (e.g., FDBMS, EAI, ETL) as well as the application of optimization techniques [BHLW08f, BHLW08g, BBH+08a, BBH+08b] during model-driven generation and finally, the deployment of these generated integration flows [BHLW09b]. Therefore, a hierarchy of platform-independent, platform-specific, and tool-specific models is used.

While all of these approaches address at most two aspects of integration flows, namely the flow specification and schema definitions, Mazon et al. defined the multi-dimensional model-driven architecture for the development of data warehouses [MTSP05]. There, the aspects (1) data sources, (2) ETL flows, (3) multi-dimensional data warehouse design, (4) customization code, and (5) application code as well as their inter-influences are taken into account. This general framework was recently extended by data merging and data customization aspects for data mining [KZOC09].

D. Declarative Integration Flow Modeling

In contrast to the usually used imperative integration flows that are modeled in a prescriptive manner, the Demaq project [BMK08] models declarative integration flows in a descriptive manner by using dependable XML message queue definitions (basic, time-based, gateway), where the processing logic is described in terms of a declarative rule language. As a foundation, a declarative Queue Definition Language (QDL) and a Queue Manipulation Language (QML) based on the XQuery update facility are used [BKM07]. Furthermore, Demaq introduced the concept of slices that are specifications of logical message groups in the form of virtual queues. Despite the concept of dependable and time-based queues, complex procedural aspects, i.e., temporal dependencies, and complex control-flows are hard to model. However, for message-centric integration flows, reasonable performance, scalability and transactional properties are achieved [BK09].

While Demaq describes the overall flow with dependencies between queues, the Orchid project [DHW+08] uses declarative schema mappings between source and target schemes and generates imperative ETL flows. To this extent, Orchid also uses declarative modeling semantics despite the resulting imperative ETL flow specifications.

Finally, the modeled integration flows are deployed into the execution environment using pre-defined exchange formats (e.g., XML). From this point in time on, the deployed integration flows are identified by logical names and executed many times. However, there are several side-effects from modeling to execution.

2.1.4 Executing Integration Flows

When deploying an integration flow, we transform the logical flow into an executable plan. Therefore, we distinguish two major plan representations. First, there are interpreted plans, where we use an object graph of operators and interpret this object graph during execution of a plan instance. Second, there are compiled plans, where code templates are used in order to generate and compile physical executable plans. As a first side-effect from the modeling perspective, (1) directed graphs are typically interpreted, while (2) hierarchies of sequences, source code and fixed flows are commonly executed as compiled plans.

Moreover, as a second side effect, flows with data-flow modeling semantics are typically also executed with data-flow semantics. The same is true for control-flow semantics. Hence, we use the flow semantics, as our major classification criterion of execution approaches. With regard to the data granularity as the second classification criterion of plan execution, we traditionally distinguish between two fundamental execution models:

- *Iterator Model:* The Volcano iterator model [Gra90, Gra94] is the typical execution model of traditional DBMS (row stores). Each operator implements an interface with the operations `open()`, `next()` and `close()`. The operators of a plan call their predecessors, i.e., the top operator determines the execution of the whole plan (pull principle). In addition, each operator can be executed by an individual thread (and thus, adheres to the pipes-and-filter execution model), where each operator exhibits a so-called iterator state (tuple buffer) [Gra90]. The advantages of this model are extensibility with additional operators as well as the exploitation of vertical parallelism (pipeline parallelism or data parallelism) and horizontal parallelism (parallel pipelines). The disadvantages are the high communication overhead between operators and the predominant applicability for row-based (tuple-based) execution.
- *Materialized Intermediates:* The concept of materialized intermediates is the typical execution model of column stores [KM05, MBK00]. Operators of a plan are executed in sequence (one operator at a time), where the result of one operator is completely materialized (as variable) and then used as input of the next operator (push principle). This reduces the overhead of operator communication and is particularly advantageous for column stores, where operators work on (compressed) columns in the form of continuous memory (arrays). This concept offers additional optimization opportunities such as vectorized execution within a single operator, or the recycling of intermediate results [IKNG09] across multiple plans.

Integration flows are typically executed as independent plan instances. Here, we distinguish between data-driven integration flows, where incoming data conceptually initiates a new plan instance, and scheduled integration flows, where such an instance is initiated by a time-based scheduler. If strong consistency is required, data-driven integration flows are executed synchronously, which means that the client systems are blocked during execution. In contrast, if only weak (eventual) consistency is required, data-driven integration flows can also be executed asynchronously using inbound queues. Note that scheduled integration flows are per se asynchronous and thus, ensure only weak consistency. We use this integration-flow-specific characteristic of independent instances to refine the classification criterion of data granularity. Therefore, we introduce the notion of instance-local (data/messages of one flow instance) and instance-global (data/messages of multiple flow instances) data granularity.

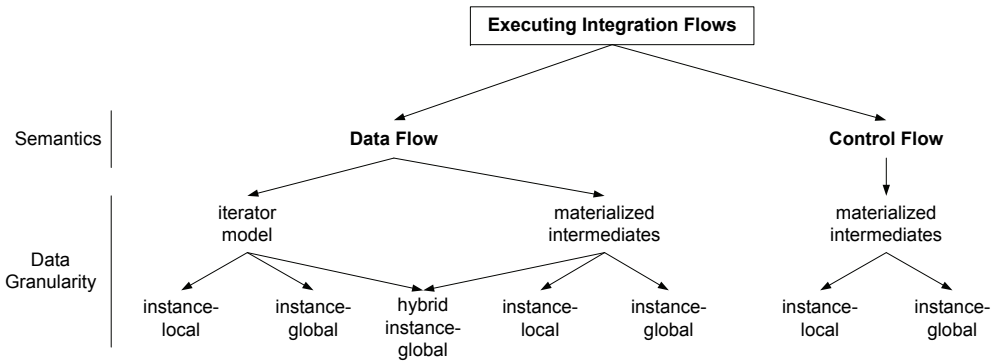


Figure 2.6: Classification of Execution Approaches for Integration Flows

Putting it all together, Figure 2.6 illustrates the overall classification. Using control-flow-oriented execution semantics (with temporal dependencies) directly implies the use of materialized intermediates in the form of variables. In this context, both instance-local and instance-global processing is possible. Typically, EAI servers and BPEL engines use this execution approach. Hence, this thesis uses the execution model of control-flow semantics and materialized intermediates as conceptual foundation. Furthermore, data flow execution semantics allow for both the iterator model and materialized intermediates as well as both instance-local and instance-global data granularity. Examples for the use of materialized intermediates are Demaq [BMK08] (instance-global) and some ETL tools (instance-local). The iterator model requires a more fine-grained classification. Iterator, instance-global refers to a tuple-based processing over data of multiple instances, where punctuations are used to distinguish data from the different instances. An example for this model is the stream-based Web service approach [PVHL09a, PVHL09b]. In contrast, iterator, instance-local refers to a tuple-based processing over data of a single instance, which is the typical execution model of ETL tools. In addition, an iterator, hybrid instance-global model can be used, where the single materialized intermediates of multiple instances are executed in a pipelined fashion and thus, with the iterator model.

Finally, we will use this classification of execution approaches in order to position the different optimization approaches as well as the results of this thesis.

2.1.5 Optimizing Integration Flows

Based on the different modeling and execution approaches, we now focus on the optimization of deployed integration flows. The main scope is the optimization of integration flows with *control-flow* execution semantics.

Due to the early state of the research area of integration flow optimization, an exhaustive classification of optimization approaches for integration flows does not exist. However, typically, integration-flow-specific characteristics are exploited for optimization. First, the expensive access of external systems is tackled with approaches that speed up the external data transfer. Second, the control-flow-oriented execution is optimized by parallel execution of subplans of an integration flow or by operator reordering. Thus, we use these two categories in order to classify the existing approaches.

A. Data Transfer Optimization

Essentially, the problem of expensive access to external systems (by actively invoking the existing outbound adapters) is tackled with data transfer optimization. In this context, we distinguish between (1) streaming transfered data (where the amount of exchanged data is almost unchanged) and (2) reducing the amount of transfered data.

Streaming Transfered Data. The standard invocation model of external systems or Web services is the transfer of materialized messages, i.e. the complete message is serialized (converted to a byte stream) or, more generally, marshalled and transfered to the external system, where the message is unmarshalled and processed. In contrast, the streaming invocation of Web services for each tuple achieves higher throughput by leveraging pipeline parallelism, but it also causes overhead, incurred by parsing SOAP/XML headers and network latencies. Thus, there is a trade-off between pipeline parallelism achieved by streaming and the existing overhead. There are approaches that exploit this trade-off by passing batches of tuples (chunks of messages) in the form of individual messages to the external system. Srivastava et al. introduced the adaptive data chunking [SMWM06], where the chunk size k was determined—based on the measured response time of a batch $c_i(k_i)$—by minimizing the response time per tuple $c_i(k_i)/k$. This concept was refined by Gounaris et al. to the use of an online extremum-control approach [GYSD08b, GYSD08a] that was adapted from control theory. Both approaches use a combination of control-flow semantics with iterator, instance-local execution. In contrast, Preißler et al. introduced the concept of stream-based Web services [PVHL09a, PVHL09b] over multiple process instances, which is thus classified as control-flow semantics with iterator, instance-global execution. Here, splitting rules are defined and message buckets are determined by the message content according to those splitting rules. These buckets are then streamed to and from the external Web service without the overhead (network latency) of passing individual messages. The concept of stream-based Web services has the advantage that both the data transfer (similar to the previously mentioned approaches, but with less latency) and the processing within the Web service (that is able to statefully work on the defined subtrees of XML messages) are optimized. In addition, streaming subtrees of messages is also advantageous for local processing steps [PHL10]. Although some of these approaches are first steps towards the adaptive behavior of integration flows, they neglect local processing costs in the sense of optimizing only calls to external systems or applying static (rule-based) optimizations only.

Reducing Transfered Data. The aforementioned approaches use a stream-based data transfer between the integration platform and the external systems. This does not affect the amount of exchanged data (payload). In contrast, there are also approaches that can reduce the amount of transfered data. First, BPEL-DT and BPEL/SQL achieve this by using references to data sets instead of physically exchanging data. BPEL-DT [HRP+07, Hab09] reduces the transfered data by passing references to a data layer and using ETL tools when possible. This reduction is achieved by using proprietary exchange formats instead of XML. Furthermore, Vrhovnik et al. introduced the rule-based optimization of BPEL/SQL processes [VSES08, VSS+07], where several rewrite rules were defined in order to condense sequences of SQL statements and to push down certain operations to the DBMS. In particular, the tuple-to-set rewrite rules reduce the amount of transfered data. Second, Subramanian and Sindre introduced the rule-based optimization of ActiveXML workflows [SS09a, SS09b]. An ActiveXML document is an XML document that contains several Web service calls in order to load dynamic external content. In this approach,

they group many independent service calls of an ActiveXML document that target the same Web service into one service call and thus, in most cases, they reduce the amount of transfered data. Note that all of these approaches for reducing the transfered data are rule-based in the sense that rewriting is only applied once during the initial deployment of an integration flow or the execution model was changed.

B. Control Flow Parallelization and Operator Reordering

In contrast to approaches that optimize the data transfer between the integration platform and the external systems, further optimization approaches use control flow parallelization in order to speed up integration flows by explicitly parallelizing processing steps or operator reordering to reduce the size of intermediate results.

In general, all of those approaches analyze dependencies between tasks of a workflow and then rewrite tasks with no dependencies between them to parallel subflows. For example, Li and Zhan determine the critical path of a workflow with regard to the execution time and then optimize only this part of the workflow [LZ05]. Typically, such rewritings are made only once during the initial deployment (*optimize-once* model [BBD09]) and thus, they are rule-based optimization approaches, where rewriting rules are often defined in terms of algebraic equivalences [HML09, YB08]. For example, Behrend and Jörg proposed to use the known Magic Sets technique for the rule-based optimization of incremental ETL flows [Beh09, BJ10]. There, selection constants gathered in a subflow are propagated to other subflows with common attributes in order to filter tuples as early as possible. In addition, the XPEDIA system [BABO+09] achieves partitioned parallelism by partitioning large XML documents into multiple parts, evaluating the parts in parallel, and finally, merging the results. This concept is a hybrid approach of control flow and data flow parallelism because parallel subflows are used to evaluate independent data partitions.

Furthermore, Srivastava et al. proposed an algorithm for finding the best plan of Web service calls (control-flow semantics) with regard to the highest degree of parallelism and thus, lowest total execution time [BMS05, SMWM06]. In addition to the exploitation of parallelism, they also include the selectivity of services as a metric for arranging these services. However, costs of local processing steps are neglected and the optimal plan is computed for each incoming ad-hoc Web service query (*optimize-always* optimization model). Similar to this, Simitisis et al. proposed an optimization algorithm for ETL flows [Sim04, SVS05] by modeling this optimization problem as a state-space search problem. In addition to the parallelization of operators, they used techniques for merging, splitting and reordering operators. Although they use a cost model, the approach is mainly rule-based due to optimization on logical level and optimization is triggered whenever a flow instance is requested. This *optimize-always* model [BBD09] is advantageous for long-running flow instances. However, when executing many instances with rather small amounts of data—as it is the case for typical workloads of integration flows—the optimize-always model falls short due to the predominant optimization costs.

The BPEL/SQL approach uses, in addition to SQL-activity-specific rewriting rules, parallelization techniques for the rewriting of SQL activities [Rei08]. This is an example of a hybrid approach, where aspects of data transfer optimization and control flow parallelization are combined in order to achieve highest performance.

To summarize, existing techniques of optimizing integration flows mainly try to decrease the costs for accessing external systems and to increase the degree of parallelism. Thereby,

none of the existing approaches addresses the basic characteristic of integration flows that they are deployed once and executed many times. On the one side, there are approaches that follow the *optimize-once* model, where rule-based optimization is applied during the initial deployment of a plan only. These approaches cannot adapt to changing workload characteristics, which may lead to poor plans over time, and many optimization techniques that require cost-based decisions cannot be applied at all. On the other side, there are approaches that follow the *optimize-always* model, where optimization is triggered whenever an instance of an integration flow is executed. This might lead to tremendous optimization overhead if many instances with rather small amounts of data are executed over time because in such scenarios, the optimization time can be even higher than the execution time of a single instance. In conclusion, we observe the lack of a tailor-made cost-based optimization approach for integration flows that allows the continuous adaptation to changing workload characteristics and that exploits the specific characteristics of integration flows in the form of being deployed once and executed many times.

2.2 Adaptive Query Processing

In contrast to the mainly rule-based optimization of integration flows, there is a large body of work on the cost-based optimization in various system categories. With regard to the aim of adaptation to changing workload characteristics as well as to unknown and misestimated statistics, the literature refers to this field as *Adaptive Query Processing* (AQP). In this section, we classify and discuss the existing techniques. For this purpose, we present an extended AQP classification that combines the known, system-oriented classification of Babu and Bizarro [BB05] with the time-oriented classification of Deshpande et al. [DHR06] and extend it by the category of integration flows. We focus only on the main characteristics and drawbacks but refer to the surveys [BB05, DIR07] and the tutorials [DHR06, IDR07] for a more detailed analysis of the individual categories.

2.2.1 Classification of Adaptive Query Processing

From a system perspective [BB05], we distinguish between (1) the plan-based adaptation of ad-hoc queries in DBMS, (2) the adaptation of deployed integration flows (see Section 2.1) in integration platforms, (3) the adaptation of continuous queries (CQs) in DSMS, and (4) tuple routing as a specific execution model for CQs. We use this system-oriented classification when surveying existing techniques in the following subsections. All of those different system categories exhibit specific characteristics that are reflected by the specific optimization approaches.

In addition to this system-oriented classification, we use a time-based classification in the sense of when re-optimization is triggered. Figure 2.7 illustrates the resulting overall classification. According to the spectrum of adaptivity [DHR06], there are essentially four types when re-optimization can be triggered. First, the coarse-grained *inter-query* optimization refers to the standard optimization model of DBMS as established in System R [SAC+79], where each query is optimized during compile-time and thus, before execution (optimize-always). For OLTP systems with rather short query execution times, plan or QEP (query execution plan) caching [ZDS+08, BBD09, Low09] is a widely used approach in order to reduce the optimization time by compiling a new QEP only if it does not exist or if statistics have changed significantly (optimize once). Second, *late binding* uses

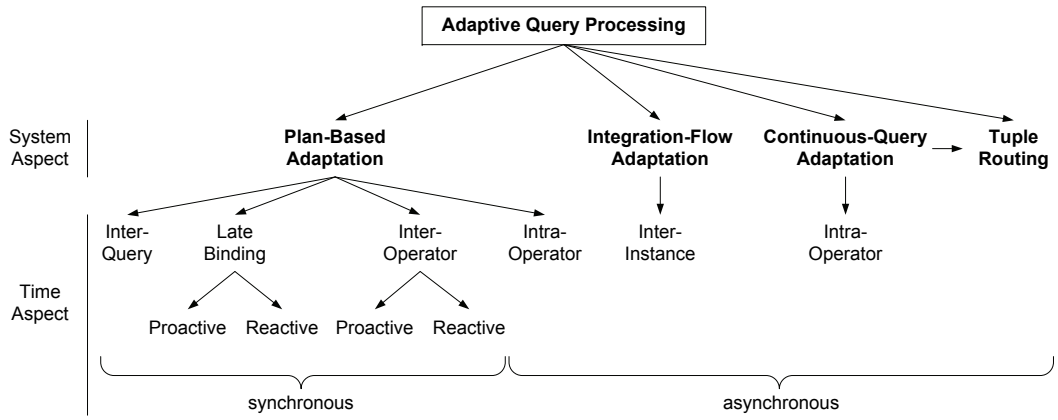


Figure 2.7: Classification of Adaptive Query Processing

natural blocking operators such as `Sort` and `Group By` in order to evaluate statistics and to re-optimize the remaining query plan during runtime. Third, *inter-operator* adaptation can re-optimize a plan at any possible point between operators by inserting artificial materialization points (`Checkpoint` operators). Fourth, the fine-grained *intra-operator* adaptation allows for re-optimization during the runtime of a single operator by data partitioning across concurrent subplans (and thus, concurrent instances of an operator). From a time perspective, we additionally use the notation of *reactive* (react to suboptimalities) and *proactive* (create switchable plans before execution) re-optimization [BBD05a]. Furthermore, we distinguish between *synchronous* and *asynchronous* re-optimization. For synchronous re-optimization (late-binding and inter-operator), execution is blocked because the current execution process requires the optimization result (in terms of the optimal remaining subplan), while for asynchronous optimization (intra-operator), execution is not blocked and plans are exchanged or merged at the next possible point.

Putting it all together, in the context of plan-based adaptation in DBMS, there exist approaches for inter-query optimization, late-binding, inter-operator and intra-operator re-optimization. Here, only intra-operator re-optimization can be executed asynchronously, while all other approaches require synchronous re-optimization because the remaining plan is optimized. Furthermore, the integration flow adaptation is classified as inter-instance adaptation, which is similar to inter-query adaptation. In the opposite, continuous query adaptation refers to intra-operator adaptation because theoretically, an infinite stream of tuples is processed (and thus, we cannot block execution until all tuples are read by any operator). Finally, the tuple routing strategies, as an alternative for continuous query adaptation, represent by themselves the most fine-grained time scheme for re-optimization, where the optimization decisions in the sense of routing paths over operators can be made for each individual tuple. In the following, we use this general classification to put existing techniques into the context of adaptive query processing.

2.2.2 Plan-Based Adaptation

For *reactive*, *inter-operator* re-optimization, the traditional optimizer is used to create a plan. During execution, intermediate results are materialized, and if estimation errors are detected, the current plan is re-optimized. In ReOpt [KD98], the optimizer is invoked if statistics differ significantly, regardless of whether or not this will produce another

plan. Further, progressive optimization [HNM+07, MRS+04, EKMR06] uses checkpoint operators to determine if validity ranges of subplans are violated and thus, can avoid unnecessary re-optimization. The major drawback is that validity ranges are defined as black boxes for subplans. Hence, directed re-optimization is impossible and it cannot be guaranteed that the current plan is not optimal. In addition, there might be trashing of intermediates, and the use of materialization points might be too coarse-grained. The latter problems were addressed by corrective query processing [IHW04] (*reactive*, *intra-operator*) that uses data partitioning across plans. Here, new plans are used only for new data and stitch-up phases combine the results of different plans. This is disadvantageous if there are large intermediate results in combination with large operator states because these results must be post-processed and then merged with a union.

In contrast to these reactive re-optimization approaches, *proactive*, *inter-operator* re-optimization in Rio [BBD05a, BBD05b] computes bounding boxes around all used estimates to express their uncertainty before optimization. The bounding boxes are then used to create robust or switchable plans. During execution, a switch operator can choose between three (low, estimate, high) different remaining plans based on a random sample of its input. However, those bounding boxes are used as black boxes with regard to single estimates. Again, this makes directed re-optimization impossible and the suboptimality of the current plan cannot be guaranteed.

To summarize, all plan-based adaptation approaches rely on the assumption of long-running queries, where mid-query re-optimization can significantly improve the query execution time. Optimization is triggered synchronously at materialization points or asynchronously in the case of intra-operator re-optimization.

2.2.3 Continuous-Query-Based Adaptation

In contrast to plan-based adaptation in DBMS, the adaptation of continuous queries in DSMS is typically realized with another approach: The optimizer specifies which statistics to gather, requests them from the monitoring component, and re-optimization is triggered periodically or whenever significant changes have occurred [BB05]. CQ-specific aspects are the extensive profiling of stream characteristics [BMM+04] and the state migration (e.g., tuples in hash tables) during re-optimization [ZRH04] in order to prevent missing tuples or duplicates and to ensure the tuple order. Examples for this optimization model are CAPE [ZRH04, RDS+04, LZJ+05], NiagaraCQ [CDTW00], StreaMon [BW04], and PIPES [CKSV08, KS09, KS04]. There exist high statistics monitoring overhead and the mentioned problem of when to trigger re-optimization.

In order to tackle the problem of state migration and to allow for fine-grained adaptation as well as load balancing, also tuple routing strategies can be applied. The routing-based adaptation does not use any optimizer but combines optimization, execution, and statistics gathering. The most prominent example of such a system is Eddies [AH00, MSHR02]. An eddy operator is used to route single tuples along different operators rather than using predefined plans. Due to the dynamic evaluation of applicable operators as well as the decisions on routing paths by routing policies [AH00, TD03, BBDW05], there can be significant overhead compared to plan-based adaptation [Des04]. This problem was weakened by the self-tuning query mesh [NRB09, NWRB09] that uses a concept drift approach to route groups of tuples instead of single tuples.

Both approaches of continuous-query-based adaptation and tuple routing strategies rely on the assumption that continuous queries process infinite tuple streams. Specific charac-

teristics are non-blocking operators and the need for state-awareness (e.g., state migration on plan rewriting). Due to processing infinite streams, re-optimization is by definition intra-operator or per-tuple routing, where re-optimization can be done asynchronously.

2.2.4 Integration Flow Optimization

As we have shown in Subsection 2.1.5, existing techniques for the optimization of integration flows are mainly rule-based (optimize-once) [LZ05, VSS+07, BJ10] and thus, do not address re-optimization, or they follow an optimize-always model [SVS05, SMWM06]. However, similar to the categories of plan-based adaptation in DBMS and continuous-query-based adaptation in DSMS, integration flows exhibit certain specific characteristics that could be exploited for a more efficient re-optimization approach.

First, integration flows are deployed once and executed many times. Due to the execution of many instances with rather small amounts of data (that stands in contrast to plan-based adaptation in DBMS), there is no need for inter-operator or intra-operator re-optimization. Second, in contrast to CQ-based adaptation in DSMS, many independent instances of an integration flow are executed over time. Due to this execution model of independent instances, there is no need for state migration because a plan can be exchanged between two subsequent instances with low costs.

Based on the specific characteristics, integration flows require *incremental* statistic maintenance and *inter-instance* (inter-query) re-optimization. The advantages would be (1) the asynchronous optimization independent of executing certain instances, (2) the fact that all subsequent instances rather than only the current query benefit from re-optimization, and (3) the inter-instance plan change without the need of state migration.

To summarize, we infer that the specific characteristics of DBMS, DSMS and integration platforms require tailor-made optimization approaches. While there exist sophisticated approaches for plan-based adaptation in DBMS and continuous-query-based adaptation in DSMS, to the best of our knowledge, there does not exist any optimization approach of integration flows that allows the continuous adaptation to changing workload characteristics. This lack of a tailor-made cost-based optimization approach for integration flows is addressed in this thesis.

2.3 Integration Flow Meta Model

Based on the literature review of integration flows, we now define the integration flow meta model that is used as our formal foundation throughout the whole thesis. On the one side, we introduce the basic notation of integration flows including the message meta model and the flow meta model as well as the execution semantics of integration flows. On the other side, we discuss specific transactional properties of integration flows that must be ensured when rewriting such flows.

2.3.1 Notation of Integration Flows

As the basic notation of integration flows, essentially, we use an extension of the so-called Message Transformation Model (MTM) [BWH+07, BHW+07]. This integration flow meta model consists of two vital parts. First, the message meta model describes the structural aspects, that is, the structure of data objects (materialized intermediates) processed by an

integration flow. Second, the flow meta model describes the operational aspects in terms of the control flow and the data flow.

Message Meta Model

Assume a sequence[2] of incoming messages $M = \{m_1, m_2, \ldots, m_n\}$ that is processed by an integration flow. We model each message m_i as a (t_i, d_i, a_i)-tuple, where $t_i \in \mathbb{Z}^+$ denotes the incoming timestamp of the message, d_i denotes a semi-structured (self-describing) tree of name-value data elements that represents the payload of the message, and a_i denotes a list of additional, atomic name-value attributes, which is used for meta data and message protocol information.

Example 2.3 (Order Message)**.** *We assume a simple order message that has been received from a Web shop in the form of a proprietary binary message. This message has been transformed into an internal XML representation by an inbound adapter:*

```
ti =  61238178981000 //Fr Jun 25 7:16:21 2010
di = "<Order>
      <Orderkey>7000000</Orderkey>
      <Custkey>1001</Custkey>
      <Orderdate>2010-06-25</Orderdate>
      <Totalprice>398.80</Totalprice>
      <Orderlines>
        <Orderline>
          <Productkey>109</Productkey> <Quantity>4.0</Quantity>
          <Lineprice>49.95</Lineprice> <Shipmode>1</Shipmode>
          <Comment>Color: blue</Comment>
        </Orderline>
        <Orderline>
          <Productkey>57</Productkey> <Quantity>1.0</Quantity>
          <Lineprice>199.00</Lineprice> <Shipmode>1</Shipmode>
        </Orderline>
      </Orderlines>
      </Order>"
ai = msgtype,    "web_order";
     RcvPort,    "5010"';
     RcvAdapter, "WSO";
```

We received this message at timestamp $t_1 = 61238178981000$ *(Fr Jun 25 7:16:21 2010). The content of the message is represented by the semi-structured tree of data elements* d_i*, which includes the general order information as well as two detailed orderlines. Finally, the list* a_i *of key/value pairs is used to hold meta data such as the message type that has been annotated by the configured inbound adapter instance.*

Using the tree of name-value data elements d_i, we can represent heterogeneous data formats and all operators can be realized according to this internal representation.

[2] We use the term sequence for finite and infinite sequences (or synonymously streams) of incoming messages. In case of infinite sequences, we have $n = \infty$.

Flow Meta Model

According to our classification of integration flows, our flow meta model defines a plan P of an integration flow as a *hierarchy of sequences* with *control-flow semantics* that uses *instance-local, materialized intermediates*.

In detail, the flow meta model exhibits an instance-based execution model, which means that each incoming message m_i or scheduled time event e_i (in case of plans that are initiated by an internal scheduler) conceptually initiates a new instance p_i of a plan P. This instance is then executed within a single master thread (except for modeled parallel subflows) such that no queues between operators are required. As a consequence of this execution model, a plan instance p_i has a local state, while a plan P is stateless by definition, which stands in contrast to standing queries in DSMS. With these control-flow semantics, both the data flow and the control flow can be appropriately described because data dependencies as well as temporal dependencies are expressible. The single operators use local messages as variables (data dependencies), while edges between operators describe the temporal order of these operators. Similar types of instance-based process models are typical for workflow-based integration systems [BPA06, OAS06, WfM05]. Putting it all together, a plan of an integration flow is defined as follows:

Definition 2.1 (Plan)**.** *A plan P of an integration flow is defined with $P = (o, c, s)$ as a 3-tuple representation. Let $o = \{o_1, o_2 \ldots, o_m\}$ denote a sequence of operators, let c denote the context of P as a set of local message variables, and let s denote a set of services $s = \{s_1, s_2 \ldots, s_l\}$ that represent outbound adapters. An instance p_i of a plan P, with $P \Rightarrow p_i$, executes the sequence of operators exactly once. Each operator o_j has a specific type as well as a unique node identifier nid. We distinguish between* atomic *and* complex *operators, where complex operators recursively contain sequences of operators with $o_j = \{o_{j,1}, o_{j,2} \ldots, o_{j,m'}\}$. A single operator can have multiple input variables and multiple output variables. Each service s_i contains a type, a configuration and a set of operations.*

Our flow meta model includes specific operator types for integration flows in order to describe local processing steps and the interaction with external systems. In detail, this flow meta model defines a set of interaction-, control-flow-, and data-flow-oriented operators by combining common process description languages with the relational algebra. We use the min-max notation [KE09] to describe the minimal and maximal number of input and output messages of each specific operator type. In addition, we identify complex operators that recursively contain arbitrary other operators.

The interaction-oriented operators describe the interaction between the integration platform and the external systems. These operators use adapters in order to encapsulate the

Table 2.1: Interaction-Oriented Operators

Name	Description	Input	Output	Complex
`Invoke`	Actively write/read data to/from external systems	(0,1)	(0,1)	No
`Receive`	Passively receive a message from an invoking client system (listener)	(0,0)	(1,1)	No
`Reply`	Send a result to the invoking client system	(1,1)	(0,0)	No

Table 2.2: Control-Flow-Oriented Operators

Name	Description	Input	Output	Complex
`Switch`	Content-based alternative with multiple paths (if-elseif-else semantics)	(1,1)	(0,0)	Yes
`Fork`	Execution of multiple parallel subflows (forklanes)	(0,0)	(0,0)	Yes
`Iteration`	Loop over a sequence of operators (foreach, while)	(0,1)	(0,0)	Yes
`Delay`	Wait until a specific timestamp or for a specified time, respectively	(0,0)	(0,0)	No
`Signal`	Initiation of a signal, which could be caught by a specific signal handler	(0,0)	(0,0)	No

Table 2.3: Data-Flow-Oriented Operators

Name	Description	Input	Output	Complex
`Assign`	Value assignment of atomic or complex objects (language: XPath)	(0,*)	(1,1)	No
`Translation`	Execution of schema mappings, in the sense of message transformations (e.g., XSLT, STX, XQuery)	(1,1)	(1,1)	No
`Selection`	Filtering of tuples depending on content-based conditions	(1,1)	(1,1)	No
`Projection`	Selection of attributes depending on a specific attribute list	(1,1)	(1,1)	No
`Join`	Join between two input messages w.r.t. a join condition and a join type	(2,2)	(1,1)	No
`Setoperation`	Set operations (`union`, `intersection`, `difference`) of two input messages	(2,2)	(1,1)	No
`Split`	Decomposition of a large message into multiple smaller messages	(1,1)	(0,*)	No
`Orderby`	Sorting of tuples according to a specified attribute	(1,1)	(1,1)	No
`Groupby`	Partitioning of tuples w.r.t. grouping attributes and aggregate functions	(1,1)	(1,1)	No
`Window`	Partitioning of tuples for ranking and correlations, without grouping	(1,1)	(1,1)	No
`Validate`	Validation of one input message depending on a specific condition	(1,1)	(0,0)	No
`Savepoint`	User-defined storage of UNDO/ REDO images of the input variable	(1,*)	(0,0)	No
`Action`	Execution of arbitrary code snippets	(0,1)	(0,1)	No

access to heterogeneous systems and applications. There, the proprietary external messages and data representations are transformed into the described internal message meta model. In detail, the group of interaction-oriented operators include the operators shown in Table 2.1. In contrast to this, the data-flow- and control-flow-oriented operators are used as local processing steps within the integration platform in the sense that they do not perform any interactions with external systems. Both groups of operators are shown in Table 2.2 and Table 2.3, respectively.

The instance-based plan execution has several implications for all operators. First, the operators use materialized intermediate results in the sense of local message variables. Second, the data flow is implicitly given by those input and output variables.

Moreover, we distinguish between external integration flow descriptions (e.g., BPEL), internal plans (logical representation) and internal compiled plans (physical representation). Here, the term plan is a shorthand for internal plans. In Section 2.4, we present several use cases that are used as example plans throughout the whole thesis.

2.3.2 Transactional Properties

There are several transactional properties of integration flows that must be guaranteed under all circumstances. In this section, we discuss different problems that can occur while executing an integration flow as well as how they are typically addressed in integration platforms and what we can imply for the cost-based optimization of integration flows. For more details, see our analysis of problem categories [BHLW08a].

The most important risk of executing integration flows is the problem of *message lost* when using a simple *send and forget* execution principle.

Problem 2.1 (Message Lost)**.** *Assume that the stream of incoming messages M is collected using transient (in-memory) inbound message queues. If a server breakdown of the integration platform has occurred, all messages not sent to the target systems will be lost. This is a problem because often the messages cannot be restored by the source systems.*

As shown in Section 2.1.2, this problem is typically addressed by persistently storing all incoming messages at the inbound server side of the integration platform. The resulting implication is that all integration platforms follow a *store and forward* principle in order to guarantee that each received message will be successfully delivered to the external systems. Thus, if a failure occurs, the stored messages are used to resume the state of execution. A failure or server breakdown can occur at an arbitrary point in time. Thus, there might be operators and interactions with external systems that have already been successfully finished, while other operators have not. When re-executing the complete integration flow, the following problem arises.

Problem 2.2 (Message Double Processing)**.** *Assume that a server breakdown during execution of plan instance p_1 has occurred. Recall that typically, each interaction with an external system is an atomic transaction. Thus, there might be successfully finished operators and currently unfinished operators. Furthermore, if the external system does not support transactional behavior, there might be partially successful interactions with external systems. If we re-execute the plan instance p_1 with p_1', we might send the same message twice to an external system.*

In order to tackle this problem, specific recovery models for integration flows are used, where we distinguish between two types. First, there is the compensation-based transac-

tion model [OAS06], where compensation flows are modeled by the user (e.g., the compensation of an `INSERT` would be a `DELETE` with the appropriate identifier). These compensations are executed for successfully executed parts of an integration flow. As a result, the compensated parts are rolled back (compensated) and completely re-executed after that. With regard to arbitrary external systems and applications, there might exist operations where no compensation exists at all. Second, there is the recovery-based transaction model [BHLW08a, SWDC10] that tries to address the problem of missing compensations. Here, REDO-images—in the sense of output messages of successfully executed operators—are stored in order to resume integration flows after the last successful operator. In conclusion, the problems of *message lost* and *message double processing* are typically addressed with persistent message storage and a tailor-made recovery model. Thus, the contract of an integration platform can be extended from store-and-forward to a form that guarantees that each received message will be successfully delivered exactly once to the external systems.

Beside these data-related guarantees also temporal guarantees must be ensured. From the perspective of integration flow optimization, we would consider executing subsequent plan instances in parallel. Unfortunately, the problem of *message outrun* would arise.

Problem 2.3 (Message Outrun)**.** *Assume two messages m_1 and m_2, where m_1 arrives earlier at the integration platform than m_2, with $t_1 < t_2$. If we execute the two resulting plan instances p_1 and p_2 in parallel, an outrun of messages in terms of changed sequential order of messages at the outbound side might take place and the result of p_2 is sent to the external system s_1 before the result of p_1. For example, if customer master data is propagated to the external system s_1 with the customer's first order, a message outrun can result in a referential integrity conflict within the target system s_1. Additional examples from the area of financial messaging that also require serialization are financial statements and stock exchange orders.*

To tackle this problem, typically, inbound message queues are used in combination with single-threaded plan execution. This serialized execution of plan instances guarantees that no message outrun can take place. This is comparable to snapshot isolation in DBMS [LKPMJP05, CRF08]. Hence, internal out-of-order processing would be possible, because we only need to ensure the *serialized external behavior* in the sense that the inbound order is equivalent to the outbound order of messages. More formally, eventual consistency [Vog08] with the property of monotonic writes (serialize the writes of the same plan), and thus, with convergence property, must be guaranteed. In addition, also monotonic reads with regard to individual data objects must be ensured.

The mentioned transactional properties have several implications for the cost-based optimization of integration flows. First, when rewriting plans during optimization, we must be aware of the problems of *message lost*, *message double processing*, and *message outrun*. Second, the contract of an integration platform with any client application or system is that each received message must be successfully delivered, in arrival-order (monotonic writes), with monotonic reads from external systems, exactly once to the external systems.

2.4 Use Cases

From a business perspective, we distinguish between horizontal and vertical integration of information systems [Sch01]. In this section, we illustrate an example scenario for both use cases, including concrete integration flows that we will use as running examples and

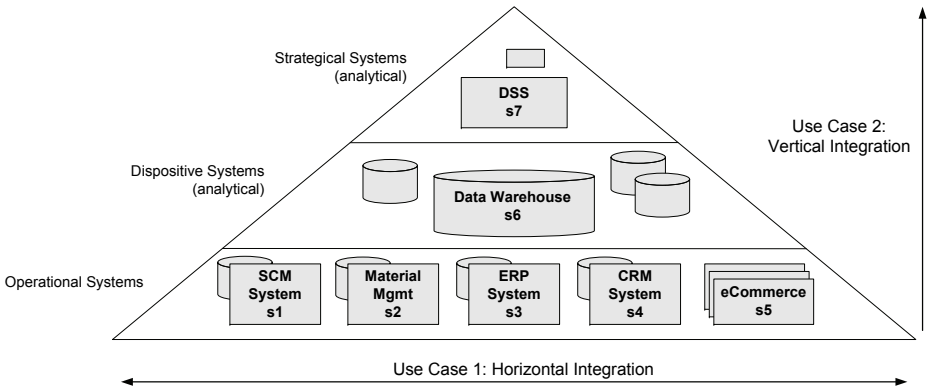
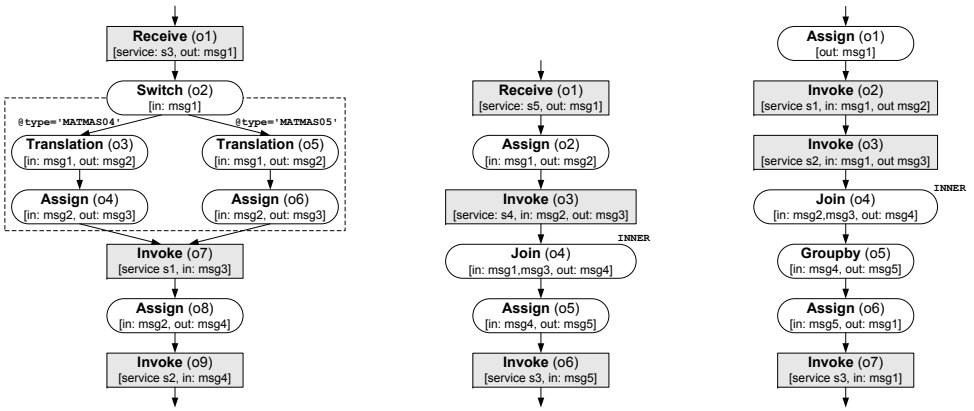


Figure 2.8: Use Cases of the Example Scenario w.r.t. the Information System Pyramid

test cases for our experimental evaluation throughout the whole thesis. Note that parts of this scenario are adapted from our DIPBench specification (Data-Intensive Integration Process Benchmark) [BHLW08b, BHLW08c]. While these flows and related workloads adhere to common characteristics of real-world integration flows, we explicitly do not use real-world data sets and workload characteristics because for evaluation purposes we want to generate arbitrary selectivities, cardinalities and temporal variations in order to cover a broad spectrum of application scenarios. The context of this example scenario is supply chain management (SCM) [MDK+01] within an enterprise. This includes the planning, execution, control, and monitoring of all supply chain activities in order to automatically synchronize customer demands with supply. Figure 2.8 illustrates an information system pyramid including all information systems affected by our simplified SCM scenario.

In this scenario, several operational, dispositive and strategical information systems exist. At the level of operational systems, various systems have to be distinguished. First, an eCommerce Web shop s_5 is used by the customers in order to submit orders. Second, the master data of all customers is maintained by a specific CRM system (Customer Relationship Management) s_4. Third, the ERP system (Enterprise Resource Planning) s_3 is the leading information system that is used for all core business activities. Fourth, all materials in terms of basic material as well as created products are managed using a material management system s_2. Fifth, and finally, there is a specific SCM system s_1, which is used to automatically submit orders to suppliers of the required basic material. At the level of dispositive systems, a global data warehouse s_6 and context-specific data marts are used in order to consolidate data from the operational systems and to enable arbitrary, analytical ad-hoc queries. Finally, at the strategical level, a DSS (Decision Support System) s_7 is used for long-term planning.

There are strong dependencies between the different heterogeneous systems and applications. Thus, integration is crucial for the automatic supply chain management, where all those systems and applications need to interact with each other. In this scenario, we distinguish two types of integration use cases. First, horizontal integration refers to the integration of operational systems, where data must be synchronized immediately based on business events such as a submitted order in the eCommerce Web shop. Second, vertical integration refers to the physical consolidation of data of all operational systems into the data warehouse infrastructure. Then, the strategical systems can be used in order to monitor, analyze, and plan all supply chain activities. In the following, we describe the two use cases in more detail and introduce example integration flows for both.



(a) Plan P_1 (b) Plan P_2 (c) Plan P_3

Figure 2.9: Example Horizontal Integration Flows

2.4.1 Horizontal Integration: EAI

The use case of horizontal integration addresses the integration of the operational systems (SCM, Material, ERP, CRM, and eCommerce) of our example scenario. This use case is typical for EAI (Enterprise Application Integration) scenarios, where updates within an operational system trigger immediate data synchronization in the sense of message exchange by data-driven integration flows. Due to the OLTP character in terms of many short business transactions in the operational systems, many instances of integration flows with rather small amounts of data per instance are executed over time. Within the scope of this use case, various integration flows exist. We pick four concrete examples with different characteristics and define these integration flows in detail.

Example 2.4 (Material Synchronization)**.** *Figure 2.9(a) illustrates an integration flow (plan P_1) that is used to synchronize material (basic material and products) master data from the ERP system to the SCM system as well as to the Material system. Essentially, when new material has been created within the ERP system, a proprietary material message is automatically sent to the integration platform. This asynchronously initiates a plan instance of P_1. The* `Receive` *operator (o_1) writes the internal message to an instance-local variable. Subsequently, a* `Switch` *operator o_2 evaluates the message type (*`MATMAS04` *or* `MATMAS05` *for inter-version compatibility) and decides which alternative* `Switch` *path is executed. After specific schema transformations (*`Translation`*) that differ according to the present message type, queries are prepared (*`Assign`*) in order to send (*`Invoke`*) the data to the external systems s_1 (SCM) and s_2 (Material).*

Example 2.5 (Orders Processing)**.** *The integration flow (plan P_2) shown in Figure 2.9(b) propagates order messages that have been submitted within our eCommerce Web shop to the central ERP system. During execution of an instance of this plan, additional master data of the customer are read from the CRM database and joined to this message. In detail, the* `Receive` *operator (o_1) gets an* orders *message from the queue and writes it to a local variable. Then, the* `Assign` *operator (o_2) is used in order to prepare a query with the* customer name *of the received message as a parameter. Subsequently, the* `Invoke` *operator (o_3) queries the external system s_4 (CRM system) in order to load master data information for that customer. Here, one SQL query Q_i per plan instance (per received*

message) is used. The **`Join`** *operator (o_4) merges the result message with the received message (with the* customer key *as join predicate). Finally, an* **`Assign`** *operator (o_5) and an* **`Invoke`** *operator (o_6) are used to sent the join result to system s_3 (ERP system).*

Example 2.6 (Material Inventory Update)**.** *The ERP system is the leading information system in our example scenario. Hence, the current material inventory is periodically updated within the ERP system. Therefore, the integration flow (plan P_3) that is shown in Figure 2.9(c) is periodically executed. Essentially, this plan specifies that after a query has been prepared, the external systems s_1 (SCM system) and s_2 (Material system) are queried. The result sets are joined (o_4) and aggregated (o_5) per material. Finally, the aggregated material inventory is sent to the ERP system s_3.*

Example 2.7 (Product Cost Estimate)**.** *Using the eCommerce Web shop, a customer can order user-defined products. Therefore, one can pose a cost estimate bid. In order to answer such a cost estimate bid, data from several operational systems is required. Therefore, the synchronous integration flow (plan P_4) shown in Figure 2.10 is used.*

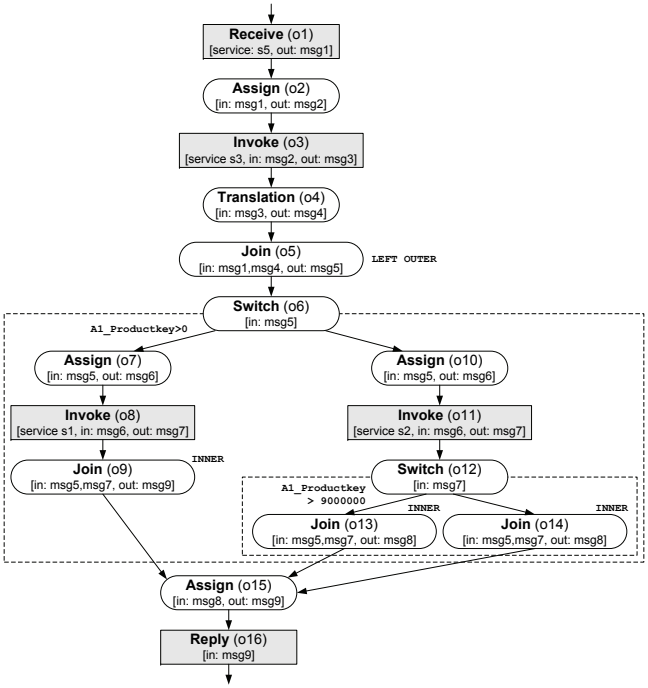


Figure 2.10: Example Integration Flow – Plan P_4

We receive a message from the external system s_5 (eCommerce). Subsequently, we query the ERP system in order to determine if this user-defined product has already been sold. If this is the case, we query the SCM system and join those possible products with the basic information. Otherwise, we query the Material system in order to determine the costs of a newly created product. Finally, we prepare the answer in the sense of a cost estimate and send a reply message to the customer (who was blocked during execution).

To summarize, the plans P_1, P_2, and P_4 are data-driven integration flows, while the plan P_3 is a periodically (time-based) initiated integration flow. Furthermore, the plans P_1, P_2, and P_3 are executed asynchronously (where for P_1 and P_2 inbound queues are used), while the plan P_4 is executed synchronously (where the client system is blocked).

2.4.2 Vertical Integration: Real-Time ETL

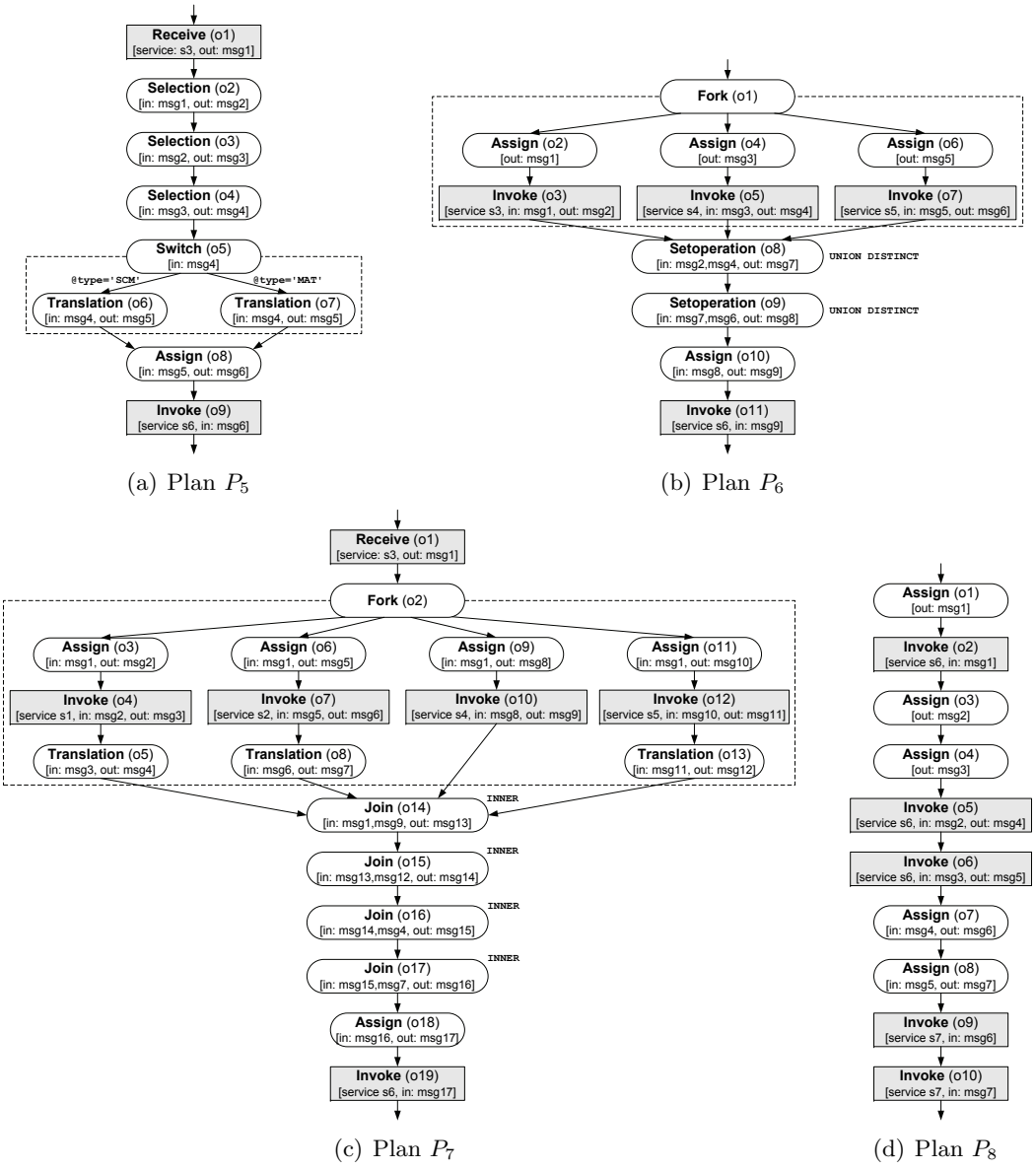
In contrast to horizontal integration, the use case of vertical integration addresses the consolidation of data from the operational source systems into dispositive and strategical systems. In this context, typically, data-centric ETL (Extraction Transformation Loading) flows are used. However, there is a trend towards operational BI where changes in the source systems are directly propagated to the data warehouse infrastructure in order to achieve high up-to-dateness of analytical query results [DCSW09, O'C08, WK10]. This is typically realized with (1) near real-time ETL flows, where data is loaded periodically but with high frequency, or with (2) real-time ETL flows, where data is loaded based on business transactions. As a result of both strategies, many instances of integration flows with rather small amounts of data are executed over time. Although this is similar to horizontal integration, we use selected ETL integration flows in order to demonstrate their specific characteristics as well.

Example 2.8 (Real-Time Standard Orders Loading)**.** *If a new order of standard products (not user-defined) is created using the ERP system, the data is directly propagated to the data warehouse infrastructure. Therefore, the integration flow (plan P_5) shown in Figure 2.11(a) is used. A plan instance is asynchronously initiated by receiving a data set from the ERP system, and it executes three different* `Selection` *operators (according to different attributes) in order to filter orders that are maintained within dedicated data marts. Subsequently, a* `Switch` *operator routes the incoming tuples—using content-based predicates—to specific schema mapping* `Translation` *operators (specific to the referenced material). Finally, the result is loaded into the data warehouse s_6.*

Example 2.9 (Near-Real-Time Customer Loading)**.** *Non-disjoint (overlapping) customer master data from the eCommerce Web shop, the CRM system as well as the ERP system is loaded into the data warehouse in a near real-time fashion. Plan instances of the integration flow (plan P_6)—that is illustrated in Figure 2.11(b)—are periodically initiated and executed. Essentially, this plan creates three parallel subflows, where the customer master data is loaded from the ERP system s_3, from the CRM system s_4, and from the eCommerce Web shop s_5. After the subflows have been temporally joined, two subsequent* `Setoperation` *operators (type* `UNION DISTINCT`*) are executed in order to eliminate duplicates. Finally, the resulting customer data is loaded into the data warehouse s_6.*

Example 2.10 (Real-Time Customized Orders Loading)**.** *Incoming orders of customized products that are registered within the central ERP system are also directly propagated to the data warehouse architecture using the integration flow (plan P_7) that is shown in Figure 2.11(c). Initiated by those incoming order messages, four parallel subflows are executed, where we load the related supplier data from the SCM system s_1, product information from the Material system s_2, customer data from the CRM system s_4, and transaction information from the eCommerce Web shop s_5. Subsequently, a left-deep-join-tree—consisting of four* `Join` *operators (chain query type)—merges the received order message with the loaded data. Finally, the result is sent to the data warehouse s_6.*

Example 2.11 (DSS Data Provision)**.** *The consolidated data of the data warehouse is partially provided for strategical planning within the DSS. In order to synchronize the data warehouse with the DSS, the integration flow (plan P_8) is used as shown in Figure 2.11(d). Essentially, instances of this plan are initiated periodically. First, a procedure is called by the* `Invoke` *operator o_2 that executes several data cleaning and aggregation operations on*



(a) Plan P_5 (b) Plan P_6

(c) Plan P_7 (d) Plan P_8

Figure 2.11: Example Vertical Integration Flows

the collected orders information. Furthermore, two queries are prepared in order to read orders and orderline information from the data warehouse. After those data sets have been extracted from the data warehouse, two `Invoke` *operators are used in order to load the data into the DSS.*

To summarize, the plans P_5 and P_7 are data-driven integration flows, while the plans P_6 and P_8 are scheduled and thus, time-based initiated integration flows. Only plan P_5 is initiated synchronously, while the plans P_6, P_7, and P_8 are all initiated asynchronously.

We will use these eight integration flows from the use cases of horizontal and vertical integration as running examples throughout the whole thesis. All of these flows exhibit different characteristics that can be exploited for optimization purposes.

2.5 Summary and Discussion

To summarize, we classified existing work of specifying integration tasks, where we mainly distinguish query-based, integration-flow-based and user-interface-oriented approaches. Due to the emerging requirements of complex integration tasks that (1) stretch beyond simple read-only applications, (2) involve many types of heterogeneous systems and applications, and (3) require fairly complex procedural aspects, imperative integration flows are increasingly used. Hence, we further classified the modeling, execution and optimization of these integration flows in detail according to a generalized reference system architecture of an integration platform for integration flows. Typically, an integration flow is modeled as a hierarchy of sequences with control-flow semantics. The control-flow semantics subsumes also implicit data-flow semantics by using instance-local, materialized intermediates in the form of variables. With regard to the optimization of such integration flows, we can summarize that mainly rule-based optimization approaches (optimize-once) have been proposed so far. This optimization model has two major drawbacks. First, adaptation to changing workload characteristics is impossible because the flow is only optimized once during the initial deployment. Second, many cost-based optimization decisions cannot be made statically in a rule-based fashion.

In contrast to the rule-based optimization of integration flows, there are numerous approaches of adaptive query processing in different application areas. However, these approaches are tailor-made for specific system types and their underlying assumptions of execution characteristics. For example, plan-based adaptation in DBMS is based on the assumption of long running queries over finite data sets, while continuous-query-based adaptation in DSMS relies on the assumption of continuous queries over infinite tuple streams. In contrast to these system types, integration flows exhibit the specific characteristics of being deployed once and executed many times, where many independent instances—with rather small amounts of data per instance—are executed over time. In conclusion, the major research question is if we can exploit context knowledge of integration flows in order to design a tailor-made optimization approach that takes into account these specific characteristics of integration flows.

As a formal foundation, we defined the basic notation in the form of a meta model for integration flows, including a message meta model that covers all static data aspects and a flow meta model that precisely defines the plan execution characteristics as well as interaction-, control-flow-, and data-flow-oriented operators. This meta model reflects the common modeling and execution semantics of integration flows as well as their specific transactional requirements and thus, all results of this thesis can be seamlessly applied to other meta models as well. Furthermore, we specified example integration flows within the context of the two major use cases of horizontal and vertical integration. These example flows represent the main characteristics and different facets of integration flows and hence, they are used as running examples throughout the whole thesis.

Putting it all together, there are existing approaches for query-based, integration-flow-based and UI-oriented integration. From the perspective of optimization, there exist tailor-made techniques for adaptive query processing. In contrast, the optimization of integration flows is mainly rule-based. Thus, the focus and novelty of this thesis is the cost-based optimization of integration flows that is strongly required in order to address the high performance demands when executing integration flows.

3 Fundamentals of Optimizing Integration Flows

In this chapter, we introduce the fundamentals of a novel optimization framework for integration flows [BHW+07, BHLW08f, BHLW08g, BHLW09a] in order to enable arbitrary cost-based optimization techniques. This framework is tailor-made for integration flows with control-flow execution semantics because it exploits the major integration-flow-specific characteristic of being deployed once and executed many times. Furthermore, it tackles the specific problems of missing statistics, changing workload characteristics, and imperative flow specifications, while ensuring the required transactional properties as well.

The core idea of the overall cost-based optimization framework for integration flows is incremental statistics maintenance in combination with asynchronous, inter-instance plan re-optimization. In order to take into account both, data-flow- and control-flow-oriented operators, we specify the necessary dependency analysis as well as a novel hybrid cost model. Furthermore, we introduce the periodical re-optimization that includes the core transformation-based optimization algorithm as well as specific approaches for search space reduction (such as a join reordering heuristic), influencing workload adaptation sensibility, and handling of correlated data. Subsequently, we present selected concrete optimization techniques (such as the reordering/merging of switch paths, early selection application, or the rewriting of sequences and iterations to parallel flows) to illustrate the rewriting of plans. Finally, the evaluation shows that significant performance improvements are possible with fairly low optimization overhead.

3.1 Motivation and Problem Description

The motivation for designing a tailor-made optimization approach for integration flows is the specific characteristic of being deployed once and executed many times that can be exploited for efficient re-optimization. Moreover, integration flows are specified with control-flow semantics (imperative) in order to enable the execution of complex procedural integration tasks.

Problem 3.1 (Imperative Integration Flows)**.** *When rewriting imperative flow specifications, the data flow* and *the control flow (in the sense of restrictive temporal dependencies) must be taken into account in order to ensure semantic correctness. Here, semantic correctness is used in the sense of preventing the external behavior (data aspects and temporal order) from being changed.*

The majority of existing flow optimization approaches [LZ05, VSS+07, BJ10, BABO+09] apply rule-based optimizations only (optimize-once) using, for example, algebraic equivalences. There, rewriting decisions are statically made only once during the initial deployment of an integration flow. However, this is inadequate due to the following problem:

Problem 3.2 (Changing Workload Characteristics)**.** *In the context of integration flows that integrate distributed systems and applications, the workload characteristics—in the form of statistics such as selectivities, cardinalities, and execution times—can change significantly over time [IHW04, NRB09, DIR07, CC08, LSM+07, BMM+04, MSHR02]. This can be caused by unpredictable workloads of the external systems (e.g., number of update transactions, amount of data to be integrated, waiting times for external systems) or temporal variations of infrastructural properties (e.g., network traffic or bandwidth). These influences lead to changing workload characteristics of the integration platform in the sense of different numbers of plan instances, different cardinalities and selectivities as well as different execution times when accessing external systems.*

As a result of Problem 3.2, rule-based optimized plans, where no execution statistics are used for optimization, fall short and may perform inefficiently over time. For the same reason, also manually optimized plans, where a fixed (hand-crafted) plan is specified by an administrator, cannot be applied [Win03]. Cost-based optimization, using data properties and execution statistics, has the potential to overcome this deficit because optimal execution plans are generated with regard to the current statistics. By keeping these statistics up-to-date, adaptation to changing workload characteristics is possible. However, for integration flows, there is the additional problem of missing statistics:

Problem 3.3 (Missing Knowledge about Statistics of External Systems)**.** *One of the main problems of integration flow optimization is the lack of knowledge about data characteristics (e.g., cardinalities, selectivities, ordering) and execution statistics (execution times, bandwidth) of the different data sources [IHW04]. Due to the integration of autonomous (loosely-coupled) source systems, the integration platform usually has no access to statistical information of the external systems—if they exist at all.*

However, execution statistics are required for cost-based optimization. Due to the missing statistics, the central integration platform has to incrementally maintain the workload characteristics and execution statistics by itself. The combination of Problem 3.2 and Problem 3.3 leads to the need for a cost-based optimization approach that incrementally maintains execution statistics and re-optimizes given plans if necessary. Unfortunately, existing cost-based approaches [SMWM06, SVS05] follow an optimize-always model, where optimization is synchronously trigged for each plan instance before it is executed. This optimize-always model falls short in the presence of many short-running plan instances, where the optimization time might be even higher than the execution time of an instance. In addition, these existing approaches do not consider an overall cost-based optimization framework but only investigate selected cost-based optimization techniques in isolation.

In consequence, we introduce the general concept of cost-based optimization of imperative integration flows. As a starting point, we follow the optimization objective of minimizing the average execution time of a plan, which implicitly increases the message throughput as well. The core concept is (1) to incrementally monitor workload characteristics and execution statistics, and (2) to periodically re-optimize given plans using a set of cost-based optimization techniques. As a result, this approach enables cost-based rewriting decisions and it achieves a suitable adaptation to changing workload characteristics, while it requires less optimization overhead than the optimize-always model.

The cost-based optimization of integration flows differs from cost-based optimization for (1) programming languages or (2) data management systems for several reasons. First, although optimizers of programming language compilers optimize imperative programs,

they do not consider any data-intensive operators (such as `Join`, `Groupby`, etc) and related optimization techniques as well as they are typically not aware of interactions with external systems. Second, existing cost-based optimizers for DBMS or DSMS optimize data-flow graphs rather than imperative control-flow graphs. Similarly to existing approaches of integration flow optimization, both categories use the optimize-once or optimize-always models and thus, do not take into account the major characteristic of integration flows in the form of being deployed once and executed many times.

We use the introduced system architecture of typical integration platforms (see Subsection 2.1.2) in order to sketch the required architectural extensions for enabling the cost-based optimization of integration flows. Figure 3.1 illustrates this extended reference system architecture including the novel cost-based optimization component.

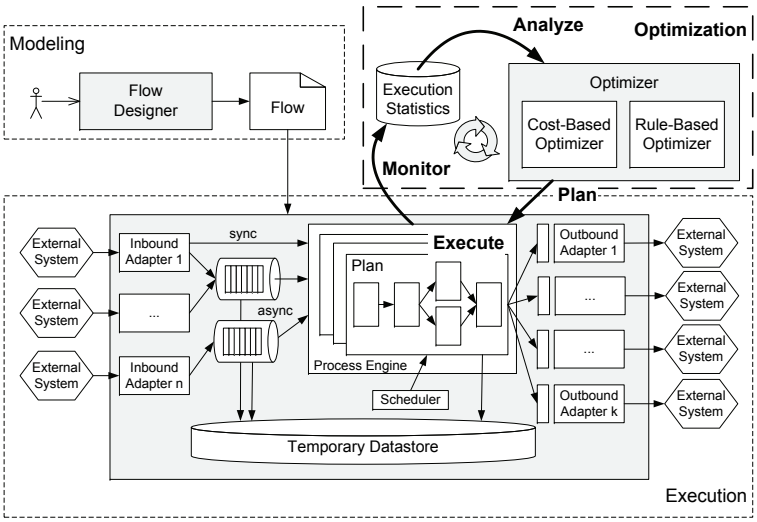


Figure 3.1: Extended Reference System Architecture

In order to address the problem of imperative integration flows, the deployment process is modified such that a dependency analysis of operators is executed during the initial deployment. Furthermore, the rule-based optimization, where we do not require any execution statistics, is executed once during this deployment as well. We described several rule-based optimization techniques for integration flows [BHW+07], but in this thesis, we omit the details for the sake of clarity of presentation. From this point in time, the plan is executed many times and the optimizer is used to continuously adapt the current plan to changing workload characteristics. The goal of plan optimization is to rewrite (transform) a given plan into a semantically equivalent plan that is optimal in the average case with regard to the estimated costs. Therefore, we use a feedback loop according to the general MAPE (Monitor, Analyze, Plan, Execute) concept [IBM05a]. During plan execution, statistics are gathered (*Monitor*). Then, the optimizer periodically analyzes the given workload (*Analyze*) in order to optimize the current plan. Subsequently, the rewritten plan is deployed (*Plan*) and used for execution (*Execute*).

We use this general feedback loop as the overall structure of this chapter. First, we present a self-adjusting cost model for integration flows and explain the cost estimation of plans, using this cost model (Monitor → Analyze; Subsection 3.2.2). Second, we illustrate the optimization problem as well as the overall periodical re-optimization algorithm (Analyze → Plan; Section 3.3). Third, we demonstrate the rewriting of plans using selected

cost-based optimization techniques (Plan → Execute; Section 3.4). Fourth, selected experimental evaluation results are presented in order to illustrate the achieved execution time improvement as well as the required optimization overhead (Execute → Monitor; Section 3.5). Finally, we summarize the results of this chapter and discuss advantages and disadvantages of this approach in Section 3.6.

3.2 Prerequisites for Cost-Based Optimization

Based on the problem of imperative integration flows, the prerequisites of cost-based optimization are two-fold. On the one side, the operator dependency analysis is required in order to ensure correctness of plan rewriting. On the other side, an accurate cost model is required in order to allow for precise cost estimation when comparing alternative plans of an integration flow. In this section, we address both fundamental requirements.

3.2.1 Dependency Analysis

When rewriting plans, we have to preserve *semantic correctness*. Here, semantic correctness is used in the sense of preventing the external behavior from being changed. This is comparable to snapshot isolation in DBMS [CRF08] or in replication scenarios [DS06, LKPMJP05]. We introduce the dependency analysis for integration flows that resembles similar dependency models from areas like compiler construction for programming languages [Muc97] and computational engineering.

The *dependency analysis* (based on the analysis of *control-flow* and *data-flow*) is executed once during the initial deployment of a plan in order to generate the so-called dependency graph $DG(P)$ of a plan P. All of our optimization techniques use these operator dependencies in order to determine whether or not rewriting is possible. In detail, we distinguish three dependency types:

- *Data Dependency* $o_j \xrightarrow{\delta^D_{m_1}} o_i$: Operator o_j depends on (reads as input) the message m_1 that has been modified or created by operator o_i (*read after write*).
- *Output dependency* $o_j \xrightarrow{\delta^O_{m_1}} o_i$: Both, operator o_j and operator o_i, write their results to message m_1, where operator o_j is a temporal successor of o_i (m_1 is overwritten multiple times, *write after write*). However, the variable might be read in between.
- *Anti-dependency* $o_j \xrightarrow{\delta^{-1}_{m_1}} o_i$: Operator o_j modifies the message m_1, while operator o_i—as a predecessor of o_j—depends on this message (before m_1 is written, it is referenced, *write after read*).

Based on this distinction of dependency types, the dependency graph $DG(P)$ is constructed using three basic rules. First, a data dependency is created if the output variable of an operator is one of the input variables of a following operator. Second, if two operators have the same output variable and if this variable is not written in between, an output dependency is created between the two operators. Third, an anti-dependency is created if the output variable of an operator is the input variable of a previous operator and if this variable is not written in between. Hence, an anti-dependency can only occur if an output dependency exist. It follows that, for example, output dependencies are subsumed by other output dependencies, i.e., one operator is involved at most in one output dependency per data object.

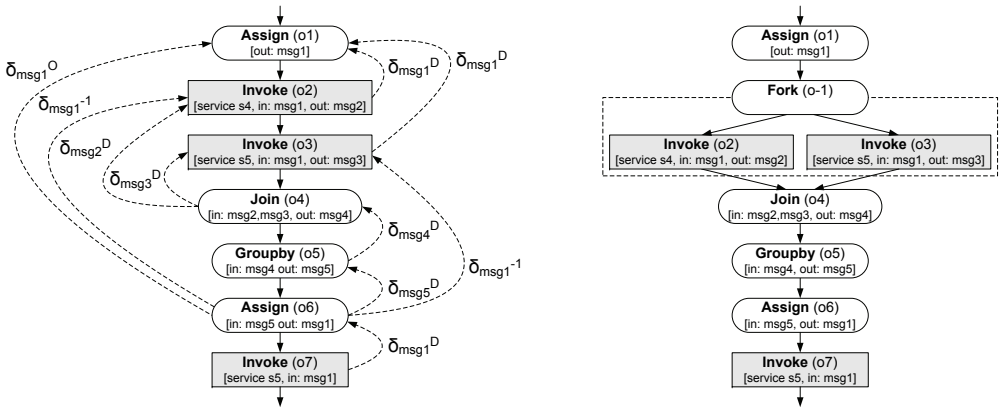
In [LRD06], semantic constraints are modeled *locally* for each individual plan by the user in order to preserve semantic correctness by using application knowledge. There, constraints between operators are explicitly specified in order to exclude these operators from any plan rewriting. In contrast to this approach, we define semantic correctness of plans with *global* constraints—that are independent of specific plans and thus, reduce the required modeling and configuration efforts for a user—as follows:

Definition 3.1 (Semantic Correctness)**.** *Let P denote an original plan and let P' denote a plan that was created by rewriting P. Then, semantic correctness of P' refers to the semantic equivalence of $P \equiv P'$. This equivalence property is given if the following constraints hold:*

1. *There are no dependencies δ between operators of concurrent subflows (parallel subflows of a* `Fork` *operator).*
2. *If there is a dependency δ between an interaction-oriented operator and another operator, the temporal order of them must be equivalent in P and P'.*
3. *If there are two interaction-oriented operators, where at least one performs a write operation and both refer to the same external system, the temporal order of them must be equivalent in P and P'.*
4. *If there exists an anti-dependency between two operators, the temporal order of these operators must be equivalent in P and P'.*
5. *If there is a data dependency between two operators, the sequential order of these operators must be equivalent in P and P' or the applied optimization technique must guarantee semantic correctness (equivalent results) of the changed sequential order.*

According to Rule 5, the specific optimization techniques decide whether or not operators, with dependencies between these operators, can be reordered. This is necessary because the reordering decision must be made based on the concrete involved operators and their parameterizations. For example, two `Selection` operators can be reordered, while this is impossible for a sequence of `Selection` and `Projection` operators if the selection attribute is removed by the projection. In case there was a dependency δ between two operators and if their sequential order (and thus, also the temporal order) was changed when rewriting P to P', the parameters of the two operators (and hence, the new data flow) must be changed accordingly. Thus, when rewriting a plan, incremental maintenance (transformation) of the dependency graph is applied as well. Due to the importance of this dependency graph, we use Example 3.1 to illustrate its core concepts.

Example 3.1 (Dependency Graph)**.** *We use the plan P_3 from Example 2.6. Figure 3.2(a) shows the related dependency graph. Consider the dependency $\delta^D_{msg_3}$. It is a data dependency (D) over the message msg_3 from o_4 to o_3; i.e., operator o_4 reads the result of o_3 as one of its join operands. Hence, o_4 depends on o_3. The dependency graph is used to determine rewriting possibilities. For example, since there are no dependencies between operators o_2 and o_3 and none of them is a writing interaction, we can insert a* `Fork` *operator and execute those as parallel subflows (Figure 3.2(b)). If there are data dependencies between local operators (no interaction-oriented operators), we can yet exchange their sequential order (e.g., o_4 and o_5 by the optimization technique eager group-by). However, we are not allowed to exchange o_6 and o_7 because this would change the external behavior. Further, the output and anti dependencies determine that we are not allowed to exchange the order of the involved operators (e.g., o_3 and o_6).*



(a) Dependency Graph $DG(P_3)$ (b) Plan P_3'

Figure 3.2: Example Dependency Graph and its Application

While our approach focuses on the optimization of complete plans, Vrhovnik et al. use a so-called *Sphere Hierarchy*—in addition to a dependency analysis—to determine optimization boundaries that must not be crossed [VSS+07]. Thus, they independently optimize partitions (spheres) of a plan. For example, the operator subsequence of a complex operator (e.g., each subflow of the `Fork` operator) is optimized only locally. However, since this is BPEL-specific [OAS06] (`scope` activity) and reduces the optimization potential, we do not restrict ourselves to these boundaries. Another approach [WPSB07] generates a minimal dependency set. For this purpose, they merge and optimize explicitly modeled dependencies. In contrast, we do not use explicitly modeled dependencies but analyze implicit dependencies (given by the data flow) in order to ensure semantic correctness. Finally, our automatic dependency-awareness reduces the development effort of integration flows and it is an essential prerequisite for both rule-based and cost-based optimization.

3.2.2 Cost Model and Cost Estimation

Referring back to the problem of changing workload characteristics (Problem 3.2), a tailor-made cost model reflecting these workload characteristics is required as a foundation of cost-based plan optimization. Due to the problem of missing statistics (Problem 3.3), execution statistics must be incrementally maintained as input for the defined cost model. In this context, the problem is to determine costs (cardinalities as well as execution times) for rewritten parts of a plan, where no statistics exist so far. Furthermore, the challenge of representing (1) data-flow-, (2) control-flow-, and (3) interaction-oriented operators arises (Problem 3.1), where the different operator categories are described by different execution statistics. While for the data-flow- and interaction-oriented operators, cardinality is a widely-used metric, this is not applicable for the control-flow-oriented operators because the costs of those operators are mainly described by means of execution times. In addition, concrete costs of the interaction-oriented operators strongly depend on the involved external systems and their individual performance. Hence, also for interaction-oriented operators the execution time rather than the cardinality should be used as the metric. In consequence of the aforementioned characteristics, in this subsection, we propose the *double-metric* cost model [BHLW09c] for enabling precise cost estimation of a plan.

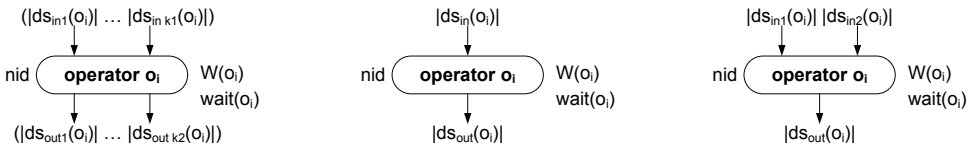
Cost models in other domains typically follow a different approach. A widely used

concept is to use an *empirical* cost model. There, physical operators are empirically analyzed (static or dynamic/leaning-based) with regard to the execution costs when varying the input parameters such as the input cardinalities. The results of these experimental measurements are then used in order to determine continuous cost functions that approximately describe the cost of the evaluated operator. This is done for all physical operators and then, it is generalized to cost functions for logical operators. These continuous cost functions are tailor-made for a given hardware platform or they include parameters to fine-tune and adjust the cost models to different hardware platforms, respectively. Empirical cost models are known to produce precise results in local settings, where operators and data structures are known in advance and where all operations are executed locally. For example, such empirical cost models exist for object-relational DBMS [Mak07], native XML DBMS [WH09, Wei11], and computational-science applications [SBC06]. Unfortunately, due to the integration of heterogeneous systems and applications, an empirical cost model is inadequate for integration flows or distributed data management in general because the dynamic adaptation to external costs (execution times of external queries or remote data access) is required [ROH99, JSHL02, NGT98, GST96, RZL04, SW97]. Furthermore, learning-based approaches can lead to non-monotonic cost functions.

In contrast to empirical cost models in DBMS, our *double-metric* cost model relies on a fundamentally different concept. The core idea is twofold. First, we define abstract cost functions according to the asymptotic time complexity of the different operator implementations using the monitored cardinalities—similar to empirical cost models—as input parameter. Second, we additionally use the monitored execution times in order to weight the computed costs. This ensures—similar to cardinality estimation in the Leo project [SLMK01, ML02, AHL+04]—that the cost model is self-adjusting by definition, which means that it does not require any knowledge about the involved external systems, and that we are able to compare costs of interaction-, control-flow- and data-flow-oriented operators by a combined metric of cardinalities and execution times.

Statistics Monitoring and Cost Model

Execution statistics are monitored at the granularity of single operators in order to enable our double metric cost estimation approach. Figure 3.3 illustrates the conceptual model of operator statistics.



(a) Generic Operator (b) Unary Operator (c) Binary Operator

Figure 3.3: General Model of Operator Execution Statistics

An operator o_i is described, in the general case (see Figure 3.3(a)), by cardinalities of arbitrary input data sets $|ds_{in_i}|$, a node identifier *nid*, the total execution time $W(o_i)$, the subsumed waiting time $wait(o_i)$ (e.g., time of external query execution, where local processing is blocked) with $0 \leq wait(o_i) \leq W(o_i)$, and cardinalities of arbitrary output data sets $|ds_{out_j}|$. Naturally, unary (see Figure 3.3(b)) and binary ((see Figure 3.3(c)) operators with a single output are most common. With regard to efficient monitoring, we use the

size of messages as good indicator for the cardinalities. Based on these monitored atomic statistics, derived more complex statistics such as the relative frequencies of alternative paths $P(path_i)$ or operator selectivities $sel(o_i) = |ds_{out_1}(o_i)|/|ds_{in_1}(o_i)|$ are computable. In order to allow cost estimation for integration flows with control-flow semantics, in the following, we define a double-metric cost model.

Definition 3.2 (Double-Metric Cost Estimation)**.** *The costs of a plan P are defined as an aggregate of operator costs, where these operator costs are defined by two metrics:*

1. Abstract costs $C(o_i)$ *are defined for all data-flow- and interaction-oriented operators in the form of their time complexity using cardinalities as the metric.*
2. Execution times $W(o_i)$ *are then used as the second metric in order to weight the abstract costs. For control-flow-oriented operators, we only use execution times.*

Both types of input statistics (cardinalities and execution times) are used in the form of aggregates over the monitored atomic statistics of executed plan instances.

For multiple deployed integration flows, it can be necessary to normalize the monitored execution statistics (in particular execution times) when aggregating them. We presented detailed cost normalization algorithms [BHLW09c] that we omit here for the sake of being focused on the core cost model.

Based on Definition 3.2, Tables 3.1-3.3 show the double-metric costs of all operators of our flow meta model (see Subsection 2.3.1). In the following, we substantiate the different cost formulas according to the two steps of our cost estimation approach.

1: Abstract Costs: In a first step, we consider the mentioned abstract costs $C(P)$ that are based on the cardinality metric. Costs for interaction-oriented operators include the costs for the operators `Receive`, `Reply` and `Invoke`. The costs for the `Receive` operator are determined as $|ds_{out}|$, i.e., by the cardinality of the received data set ds_{out} and they comprise costs for transport, protocol handling, format conversion as well as decompression. The costs for the related `Reply` operator are similarly computed with $|ds_{in}|$. Finally, the costs of the `Invoke` operator are computed by $|ds_{in}| + |ds_{out}|$ because it actively sends and receives data. A data set $|ds_{out}|$ might be `NULL` (e.g., in case of pure write interactions); in that case, we assume a cardinality of 0. Control-flow-oriented operators have no abstract costs. An exception to this is the `Switch` operator because it evaluates expressions over input data sets, which requires costs of $|ds_{in}|$ for a single path expression and due to the if-elseif-else semantic total costs of $\sum_{i=1}^{n} (P(path_i) \cdot i \cdot |ds_{in}|)$ for all n paths, where $P(path_i)$ denotes switch path probabilities (relative frequencies). For computing the abstract costs of the data-flow-oriented operators, we partly adapted a cost model from a commercial RDBMS [Mak07] to the specific characteristics of integration flows. With regard to physical operator alternatives, we excluded hash-based algorithms because they

Table 3.1: Double-Metric Costs of Interaction-Oriented Operators

Operator Name	**Abstract Costs** $C(o_i)$	**Execution Time** $W(o_i)$
`Invoke`	$\|ds_{in}\| + \|ds_{out}\|$	$W(o_i)$
`Receive`	$\|ds_{out}\|$	$W(o_i)$
`Reply`	$\|ds_{in}\|$	$W(o_i)$

Table 3.2: Double-Metric Costs of Control-Flow-Oriented Operators

Operator Name	**Abstract Costs** $C(o_i)$	**Execution Time** $W(o_i)$
`Switch`	$\lvert ds_{in}\rvert$	$\sum_{i=1}^{n}\left(P\left(path_i\right)\cdot\left(\sum_{j=1}^{i}W\left(expr_{path_j}\right)+\sum_{k=1}^{m_i}W\left(o_{i,k}\right)\right)\right)$
`Fork`	-	$\max_{i=1}^{n}\left(\sum_{j=1}^{m_i}W\left(o_{i,j}\right)+i\cdot W\left(\text{Start\_Thread}\right)\right)$
`Iteration`	-	$r\cdot\sum_{i=1}^{n}W\left(o_i\right)$
`Delay`	-	$W(o_i)$
`Signal`	-	$W(o_i)$

Table 3.3: Double-Metric Costs of Data-Flow-Oriented Operators

Operator Name	**Abstract Costs** $C(o_i)$	**Execution Time** $W(o_i)$
`Assign`	$\lvert ds_{in}\rvert+\lvert ds_{out}\rvert$	$W(o_i)$
`Translation`	$\lvert ds_{in}\rvert+\lvert ds_{out}\rvert$	$W(o_i)$
`Selection`	$\lvert ds_{in}\rvert$ `sorted:`$\lvert ds_{in}\rvert/2$	$W(o_i)$
`Projection`	$\pi_{\texttt{distinct}}:\lvert ds_{in}\rvert\cdot\lvert ds_{out}\rvert/2$ $\pi_{\texttt{all}}/\texttt{sorted}:\lvert ds_{in}\rvert$	$W(o_i)$
`Join`	$\bowtie_{NL}$: $\lvert ds_{in1}\rvert+\lvert ds_{in1}\rvert\cdot\lvert ds_{in2}\rvert$ `sorted:`$\lvert ds_{in1}\rvert+\lvert ds_{in2}\rvert$	$W(o_i)$
`Setoperation`	$\cup,\setminus,\cap:\lvert ds_{in1}\rvert+\lvert ds_{in2}\rvert\cdot\lvert ds_{out}\rvert/2$ `sorted:`$\lvert ds_{in1}\rvert+\lvert ds_{in2}\rvert$ $\cup_{\texttt{all}}:\lvert ds_{in1}\rvert+\lvert ds_{in2}\rvert$	$W(o_i)$
`Split`	$\lvert ds_{in}\rvert+\sum_{j=1}^{n_i}\lvert ds_{out_j}\rvert$	$W(o_i)$
`Orderby`	$\lvert ds_{in}\rvert\cdot\log_2\lvert ds_{in}\rvert$	$W(o_i)$
`Groupby`	$\lvert ds_{in}\rvert+\lvert ds_{in}\rvert\cdot\lvert ds_{out}\rvert/2$	$W(o_i)$
`Window`	$\lvert ds_{in}\rvert\cdot\lvert ds_{out}\rvert/2$	$W(o_i)$
`Validate`	$\lvert ds_{in}\rvert$	$W(o_i)$
`Savepoint`	$\lvert ds_{in}\rvert$	$W(o_i)$
`Action`	$\lvert ds_{in}\rvert+\lvert ds_{out}\rvert$	$W(o_i)$

do not introduce additional requirements on plan rewriting and they are not applicable without extensions for persistent messages that do not fit into main memory. Additionally to the relational operators, the operators `Assign` and `Translation` both exhibit abstract costs of $\lvert ds_{in}\rvert+\lvert ds_{out}\rvert$ because they get an input message and transform it to an output message, where the output can be smaller or larger than the input. Furthermore, the `Split` operator decomposes a message into multiple smaller messages. Hence, the sum of cardinalities for the input message $\lvert ds_{in}\rvert$ and of the set of output messages $\sum_{j=1}^{n_i}\lvert ds_{out_j}\rvert$ is a representative indicator. Both the `Validate` and the `Savepoint` operator have costs of $\lvert ds_{in}\rvert$ because they do not create any output message. Integrating the `Savepoint` into the cost model enables, for example, to include the overhead for recoverability [SWDC10]

into optimization. Finally, the `Action` operator exhibits abstract costs in the form of $|ds_{in}| + |ds_{out}|$. However, the `Action` operator will not be included in any rewriting (except for parallelism) because it executes arbitrary code snippets and thus, is treated as a black box by the optimizer.

2: Execution Times: In a second step, we monitor statistics (e.g., execution times and cardinalities) in order to weight the mentioned abstract costs of interaction- and data-flow-oriented operators. With the aim to estimate the costs for a newly created plan P', we aggregate the costs $C(o_i')$ and $C(o_i)$ of the single operators weighted with the execution statistics $W(o_i)$ of the current plan P. Thus, we estimate missing statistics with

$$\hat{W}(o_i') = \frac{C(o_i')}{C(o_i)} \cdot W(o_i). \tag{3.1}$$

For control-flow-oriented operators, we directly estimate the execution time. The costs for the complex control-flow-oriented `Switch` operator can be computed by

$$\sum_{i=1}^{n} \left(P\left(path_i\right) \cdot \left(\sum_{j=1}^{i} W\left(expr_{path_j}\right) + \sum_{k=1}^{m_i} W\left(o_{i,k}\right) \right) \right), \tag{3.2}$$

where we require switch path probabilities $P\left(path_i\right)$ (relative frequencies) for all n paths, weighted costs for path expression evaluation $W\left(expr_{path_j}\right)$ because the evaluation of these expressions (e.g., XPath) can be cost-intensive as well as weighted costs for the m_i operators of each path. Here, the second summation goes only up to $j = i$ because the evaluation is aborted if we find a true condition due to the if-elseif-else semantic of this operator. Similar, the costs for the complex `Fork` operator (concurrent subflows of arbitrary operators) are computed by the most time-consuming subflow:

$$\max_{i=1}^{n} \left(\sum_{j=1}^{m_i} W\left(o_{i,j}\right) + i \cdot W\left(\text{Start\_Thread}\right) \right), \tag{3.3}$$

where $W(Start\_Thread)$ denotes a constant, used to represent the required time for creation and start of a thread. When computing the costs for the `Iteration` operator, with

$$r \cdot \sum_{i=1}^{n} W\left(o_i\right), \tag{3.4}$$

the average number of iteration loops r is required as well. Further, the waiting time of the `Delay` operator is also taken into account. Finally, the `Signal` operator has to be mentioned, where costs (needed for raising an exception) are represented as a constant.

Putting it all together, this cost model has several fundamental properties. Some of these properties are used by different chapters of this thesis.

- *Self-Adjustment:* Due to weighting with monitored execution times, the cost model is self-adjusting with regard to the behavior of different operators according to changing workload characteristics. Thus, the cost model adjusts itself to the present environment (hardware platform, behavior of external systems). Especially, this behavior of external systems or different queries to these systems and thus, also of network properties could not be taken into account by an empirical cost model that is only based on cardinalities.

- *Comparability of Control- and Data-Flow-Oriented Operators:* The double-metric cost model enables the comparison of data-flow and control-flow-oriented operators by their normalized execution time. In contrast to empirical cost models, the double metric cost model includes interaction- and control-flow-oriented operators as well.
- *Plan Cost Monotonicity:* The Picasso project [RH05] has shown that the assumption of *Plan Cost Monotonicity* (PCM) holds for *most* queries in commercial DBMS even over *all* plans of a plan diagram [HDH07]. In contrast, this assumption *always* holds for our cost model of integration flows with regard to a *single* plan. As a result, the costs of a plan are monotonically non-decreasing with regard to any increasing influencing parameter such as selectivities, cardinalities, or execution times.
- *Asymmetric Cost Functions:* The cost model exhibits asymmetric cost functions, i.e., the ordering of binary operator inputs has influence on the computed costs. Thus, commutativity of inputs must be considered during optimization.
- *No ASI Property:* Finally, the cost model does not exhibit the ASI (Adjacent Sequence Interchange) property [Moe09]. This property is given if and only if there is a ranking function of data sets such that the ordering of ranks is the optimal join ordering. Due to (1) possibly correlated external data sets and (2) different join implementations including the merge-join (which is known to not having the ASI property [Moe09]) our cost model does not exhibit this property.

Cost Estimation

In the following, we illustrate the cost estimation, using the known data-flow-oriented optimization technique of eager group-by [CS94] (type invariant group-by), as an example rewriting technique.

Example 3.2 (Cost Estimation)**.** *Assume the plan P_3 with monitored execution statistics and a plan P_3' that has been created by rewriting P_3 during optimization (invariant group-by due to N:1-relationship between data sets, e.g., given by message schema descriptions). There are no statistics available for P_3'. The plans are shown in Figure 3.4. The statistics of subplans that are equivalent in P_3 and P_3' can be reused. Thus, only the new output cardinality $|ds_{out}(o_5')|$ and the execution times $W(o_5')$ and $W(o_4')$ have to be estimated. Assuming a monitored join selectivity (where f is a shorthand for filter selectivity) of*

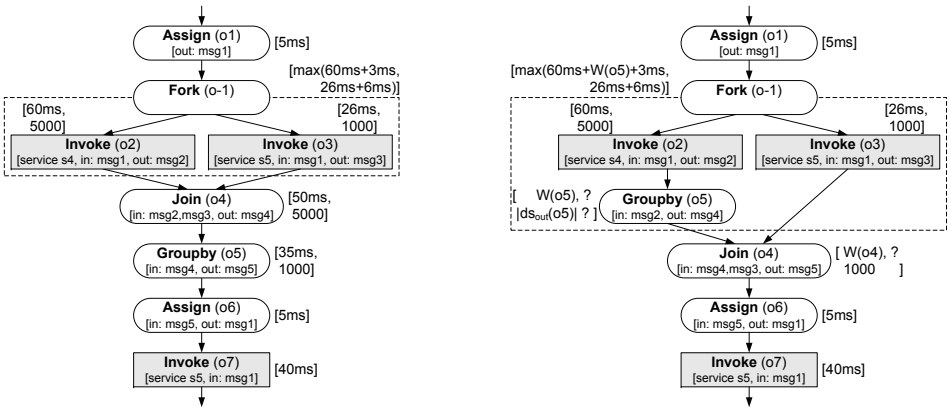
$$f_{ds_{out}(o_2),ds_{out}(o_3)} = \frac{|ds_{out}(o_2) \bowtie ds_{out}(o_3)|}{|ds_{out}(o_2)| \cdot |ds_{out}(o_3)|} = \frac{|ds_{out}(o_4)|}{|ds_{out}(o_2)| \cdot |ds_{out}(o_3)|} = \frac{5{,}000}{5{,}000{,}000} = \frac{1}{1{,}000}$$

and a group-by selectivity of

$$f_{\gamma ds_{out}(o_4)} = \frac{|\gamma ds_{out}(o_4)|}{|ds_{out}(o_4)|} = \frac{|ds_{out}(o_5)|}{|ds_{out}(o_4)|} = \frac{1{,}000}{5{,}000} = \frac{1}{5}.$$

The selectivities are used in order to compute the output cardinalities of new or reordered operators. Due to the invariant placement of the group-by operator, we can compute the new group-by output cardinality and directly set the new join output cardinality by

$$|\hat{ds}_{out}(o_5')| = f_{\gamma ds_{out}(o_4)} \cdot |ds_{out}(o_2)| = \frac{1}{5} \cdot 5{,}000 = 1{,}000$$
$$|\hat{ds}_{out}(o_4')| = 1{,}000.$$



(a) Plan P_3 (b) Plan P_3'

Figure 3.4: Plan Cost Estimation Example

Furthermore, we estimate the missing execution times using the monitored statistics of P_3 *and the defined abstract costs of* P_3 *and* P_3' *as follows:*

$$\hat{W}(o_5') = \frac{|ds_{in}(o_5')| + |ds_{in}(o_5')| \cdot \frac{|ds_{out}o_5'|}{2}}{|ds_{in}(o_5)| + |ds_{in}(o_5)| \cdot \frac{|ds_{out}(o_5)|}{2}} \cdot W(o_5) = \frac{2{,}505{,}000}{2{,}505{,}000} \cdot 35\,\text{ms} = 35\,\text{ms}$$

$$\hat{W}(o_4') = \frac{|ds_{in1}(o_4')| + |ds_{in1}(o_4')| \cdot |ds_{in2}(o_4')|}{|ds_{in1}(o_4)| + |ds_{in1}(o_4)| \cdot |ds_{in2}(o_4)|} \cdot W(o_4) = \frac{1{,}001{,}000}{5{,}005{,}000} \cdot 50\,\text{ms} = 10\,\text{ms}.$$

Finally, we can use the computed cost estimates, aggregate the plan costs, and compare these costs as follows:

$$\begin{aligned} W(P_3) &= 5\,\text{ms} + \max(60\,\text{ms} + 3\,\text{ms}, 26\,\text{ms} + 2 \cdot 3\,\text{ms}) + 50\,\text{ms} + 35\,\text{ms} + 5\,\text{ms} + 40\,\text{ms} \\ &= 198\,\text{ms} \\ \hat{W}(P_3') &= 5\,\text{ms} + \max(60\,\text{ms} + 35\,\text{ms} + 3\,\text{ms}, 26\,\text{ms} + 2 \cdot 3\,\text{ms}) + 10\,\text{ms} + 5\,\text{ms} + 40\,\text{ms} \\ &= 158\,\text{ms}. \end{aligned}$$

In our example, we would choose P_3' *as execution plan because, it is optimal, on average, under the assumption of precise monitored statistics. Note that although* o_2 *and* o_7 *are defined with equal abstract costs (`Invoke`), we adapt to the concrete workload characteristics by weighting those costs with monitored execution times. In addition, the double metric cost model enables us to use one single metric (the execution time) for data-flow-oriented operators (e.g., `Groupby`), interaction-oriented operators (e.g., `Invoke`), and control-flow-oriented operators (e.g., `Fork`).*

To summarize, we proposed the first complete cost model for integration flows. This double-metric cost model is self-adjusting because we weight the abstract costs with monitored execution statistics. For this reason, over time, the estimates converge to the real costs of the concrete application environment and hence, this cost model enables the adaptation to changing workload characteristics. Further, the two metrics enable to integrate the interaction-, control-flow-, and data-flow-oriented operators into a unique cost model and thus, enable the comparison of plans with control-flow semantics.

3.3 Periodical Re-Optimization

Based on the presented prerequisites, we now explain the core algorithm for the cost-based optimization of imperative integration flows. First, we formally define the *periodic plan optimization problem* including the existing parameters and show that this problem is NP-hard. Due to the complexity of this optimization problem, we additionally introduce two search space reduction approaches. Then, we describe how the parameters can be leveraged in order to influence the sensibility of workload adaptation. Finally, we sketch how to handle conditional probabilities and correlation without the knowledge about data characteristics (e.g., value distributions) of external systems.

The core optimization algorithm is independent of any concrete optimization technique. For that reason, it can be extended with arbitrary new techniques. However, we use selected optimization techniques in order to illustrate the properties and the behavior of our optimization algorithm. In Section 3.4, we will explain various concrete optimization techniques in more detail.

3.3.1 Overall Optimization Algorithm

Existing approaches of integration flow optimization, which also take execution statistics into account [SMWM06, SVS05], use the *optimize-always* model, where the given plan is optimized for each initiated plan instance. While this is advantageous for changing workload characteristics in combination with long running plan instances, it fails under the assumption of many plan instances with rather small amounts of data because in this case the optimization time might be even higher than the execution time of the plan. In consequence, we introduce an optimization algorithm that exploits the integration-flow-specific characteristics of (1) being deployed once and executed many times as well as (2) the presence of an initially given imperative integration flow.

Optimization Problem

The goal of plan optimization is to rewrite (transform) a given plan into a semantically equivalent plan that is optimal in the average case with regard to the estimated costs. The major differences to DBMS are (1) the *average* case optimization of a deployed plan and (2) the *transformation-based* plan rewriting that takes into account the specified control-flow semantics of imperative integration flows. As a first step, we define the optimal plan as follows:

Definition 3.3 (Optimal Plan)**.** *A plan $P = (o, c, s)$ is* optimal *at timestamp T_k with respect to a given workload $W(P, T_k)$ if no plan $P' = (o', c', s)$ with lower estimated execution time $\hat{W}(P') < W(P)$ exists. Thus, the optimization objective ϕ of any optimization algorithm is to minimize the estimated average execution time of the plan with:*

$$\phi = \min \hat{W}(P). \tag{3.5}$$

The plan P is optimal according to the monitored statistics at the timestamp of optimization T_k. In case of changing workload characteristics, over time, the plan P might loose this property of optimality. For this reason, we require periodical re-optimization if we do not want to employ an optimize-always model. Hence, as a second step, we formally define this time-based optimization problem as follows:

Definition 3.4 (Periodic Plan Optimization Problem (P-PPO[3]))**.** *The P-PPO describes the periodical creation of the* optimal *plan P at timestamp T_k with the period Δt (optimization interval). The workload $W(P, T_k)$ is available for a sliding time window of size Δw. Optimization required an execution time T_{Opt} with $T_{Opt} = T_{k,1} - T_{k,0}$ and $T_k = T_{k,0}$. When solving the* P-PPO *at timestamp $T_{k,1}$, the result is the* optimal *plan w.r.t. the statistics available at timestamp $T_{k,0}$.*

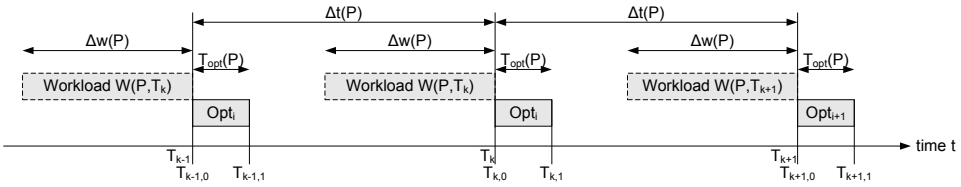


Figure 3.5: Temporal Aspects of the P-PPO

Figure 3.5 illustrates the temporal aspects of periodical re-optimization in more detail. We optimize the given plan at timestamp T_k with a period of Δt. During this optimization, we evaluate costs of a plan by double-metric cost estimation, where only statistics in the interval of $[T_k - \Delta w, T_k]$ are taken into account. The parameters optimization interval Δt and workload sliding time window size Δw are set individually for each plan P. We will revisit the influence of these parameters in Subsection 3.3.3. The following example illustrates this concept of periodical re-optimization and shows how the introduced cost estimation approach is used in this context.

Example 3.3 (Periodical Re-optimization)**.** *Assume a plan P with instances p_i, for which execution statistics have been monitored. This plan comprises a* `Receive` *operator o_1 and a following* `Invoke` *operator o_2.*

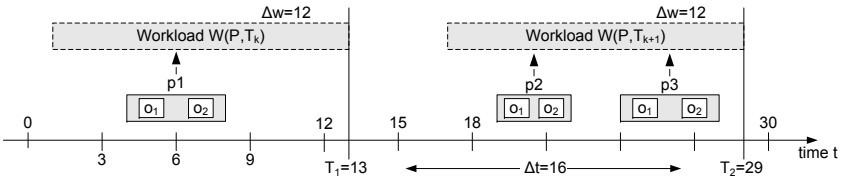


Figure 3.6: Example Execution Scenario

Figure 3.6 illustrates a situation where three plan instances, p_1, p_2 and p_3, are executed in sequence. Hence, nine statistic tuples are stored and normalized: two operator tuples and a single plan tuple, for every plan instance. Table 3.4 illustrates example execution statistics. Assume an optimization interval $\Delta t = 16\,\text{ms}$, a sliding time window size $\Delta w = 12\,\text{ms}$, and the moving average (`MA`*) over this window as workload aggregation method (method used to aggregate statistics over the time window). As a result, we estimate the costs of plan P at timestamps $T_k = \{T_1 = 13, T_2 = 29\}$ as follows (simplified on plan granularity):*

$$\hat{W}_{T_1}(P) = \frac{\sum_{i=1}^{n} W(p_i \in [1,13])}{|W_P|} = \frac{W_3}{1} = \frac{4\,\text{ms}}{1} = 4\,\text{ms}$$

$$\hat{W}_{T_2}(P) = \frac{\sum_{i=1}^{n} W(p_i \in [17,29])}{|W_P|} = \frac{W_6 + W_9}{2} = \frac{3\,\text{ms} + 4\,\text{ms}}{2} = 3.5\,\text{ms}.$$

[3]We use the prefix P- as indicator for problems throughout the whole thesis. As an example, P-PPO is the abbreviation for the Periodic Plan Optimization problem.

Table 3.4: Example Execution Statistics

PID	NID	OType	Start	End	...	W [ms]
1 (p_1)	1	`Receive`	4.1	5.7	...	1.6
1 (p_1)	2	`Invoke`	5.9	7.7	...	1.8
1 (p_1)		`Plan`	4.0	8.0	...	4.0
2 (p_2)	1	`Receive`	19.1	20.3	...	1.2
2 (p_2)	2	`Invoke`	20.4	21.7	...	1.3
2 (p_2)		`Plan`	19.0	22.0	...	3.0
3 (p_3)	1	`Receive`	24.2	26.1	...	1.9
3 (p_3)	2	`Invoke`	26.1	27.9	...	1.8
3 (p_3)		`Plan`	24.0	28.0	...	4.0

Basically, the optimization algorithm is triggered with period $\Delta t = 16\,\text{ms}$. *At those timestamps, only statistics of plan instances* $p_i \in [T_k - \Delta w, T_k]$ *are used for cost estimation. Hence, at* $T_1 = 13$, *only statistics of* p_1 *are included, while at* $T_2 = 29$ *(*$T_1 + \Delta t$*), statistics of* p_2 *and* p_3 *are used.*

As a result, we are able to periodically estimate the costs of a plan with the aim to optimize this plan according to the current workload characteristics.

In the following, we discuss the complexity of this approach. Essentially, the periodic plan optimization problem that includes the creation of the *optimal* plan is NP-hard. This claim is justified by the known complexity of two subproblems (concrete optimization techniques). First, the merging of parallel flows (see `Fork` operator) to a minimal number of parallel flows with a maximum constraint on the total costs of such a flow is reducible to the NP-hard bin packing problem. Second, also the subproblem of join enumeration is, in general, NP-hard but requires a more detailed argumentation:

- A plan is a hierarchy of sequences (Definition 2.1) with control-flow semantics (that subsume the data-flow semantics). Hence, all types of join queries are possible.
- The join enumeration is NP-hard in general (if all types of join queries are supported) [Neu09]. For a comprehensive analysis of known results, see [Moe09].
- If a cost model has the ASI property, join enumeration can be computed with polynomial time. There, ranks are assigned to relations and the sequence of ordered ranks is optimal [IK84, KBZ86, CM95].
- Our cost model does not exhibit the ASI property (see Subsection 3.2.2).

As a result, the *periodic plan optimization problem* belongs to the complexity class of NP-hard problems. Obviously, the analogous problem of cost-based query optimization in DBMS is also NP-hard. However, in [SMWM06], it was shown that the optimal Web service query plan can be computed in $O(n^5)$, where n is the number of Web services. The difference is caused by their assumption of negligible local processing costs (including joins and other data-flow-oriented operators) such that no join enumeration has been used. In contrast, our optimization objective is to minimize the average total execution costs including local processing steps. In order to ensure efficient periodical re-optimization, we will introduce tailor-made search space reduction heuristics in Subsection 3.3.2.

Optimization Algorithm

According to the defined integration flow optimization problem, we now explain the overall optimization algorithm including the two aspects of (1) when and how to trigger re-optimization of a plan and (2) how to re-optimize the given plan using the set of available cost-based optimization techniques. The naïve algorithm for solving the P-PPO comprises three subproblems: (1) the complete creation of alternative plans (the search space), (2) the periodical cost evaluation of each created plan (search space evaluation), and (3) the choice of the plan with minimal costs. In contrast to this generation-based approach, we exploit the specific characteristic of an initially given imperative plan, by using an iterative (transformation-based) optimization algorithm. In the following, we describe in detail how to trigger re-optimization and how to re-optimize the given plan.

Algorithm 3.1 Trigger Re-Optimization (A-TR)

Require: plan identifier *ptid*, optimization interval Δt, workload window size Δw, aggregation method *method*, optimization algorithm *algorithm*

```
1:  while true do
2:    sleep(Δt)
3:    P ← getPlan(ptid)
4:    DG ← getDependencyGraph(ptid)
5:    Estimator.aggregateStatistics(P, Δw, method)      // see Subsection 3.3.3
6:    ret ← Optimizer.optimizePlan(P, DG, algorithm)
7:    if ret.isChanged() then
8:      P ← ret.getPlan()
9:      putPlan(ptid, P)
10:     putDependencyGraph(ptid, ret.getDG())
11:     Parser.recompilePlan(ptid, P)
12:     Runtime.exchangePlans(ptid, P)
```

Algorithm 3.1[4] illustrates when and how re-optimization is triggered. Essentially, this algorithm is started as a background thread for each deployed integration flow and periodically issues plan re-optimization with period Δt (line 2). Therefore, monitored execution statistics are aggregated with a certain aggregation method (line 5) and re-optimization is initiated with one of our optimization algorithms (line 6). If the current plan has been changed during this re-optimization, we recompile the logical plan into an executable physical plan (line 11) and exchange the plan at the next possible point between two subsequent plan instances (line 12). When triggering re-optimization, the optimization algorithm is selected. There, `patternMatchingOptimization` (A-PMO) is the default algorithm, while several additional heuristic algorithms can be used for search space reduction.

Algorithm 3.2 illustrates our transformation-based optimization algorithm A-PMO. This algorithm is invoked for the complete plan, where it recursively iterates over the hierarchy of operator sequences (of the current plan) and applies optimization techniques according to the operator types (the included comments show the abbreviations of applied optimization techniques, which we partly discuss in Section 3.4). There are four types of optimization techniques. First, we apply all techniques, which need to be executed on top level of a plan and before all other optimization techniques (line 2). For example, the join

[4]Similar to the naming scheme of problems, we use the prefix A- to indicate names of algorithms.

Algorithm 3.2 Pattern Matching Optimization (A-PMO)

Require: operator *op*, dependency graph *DG*

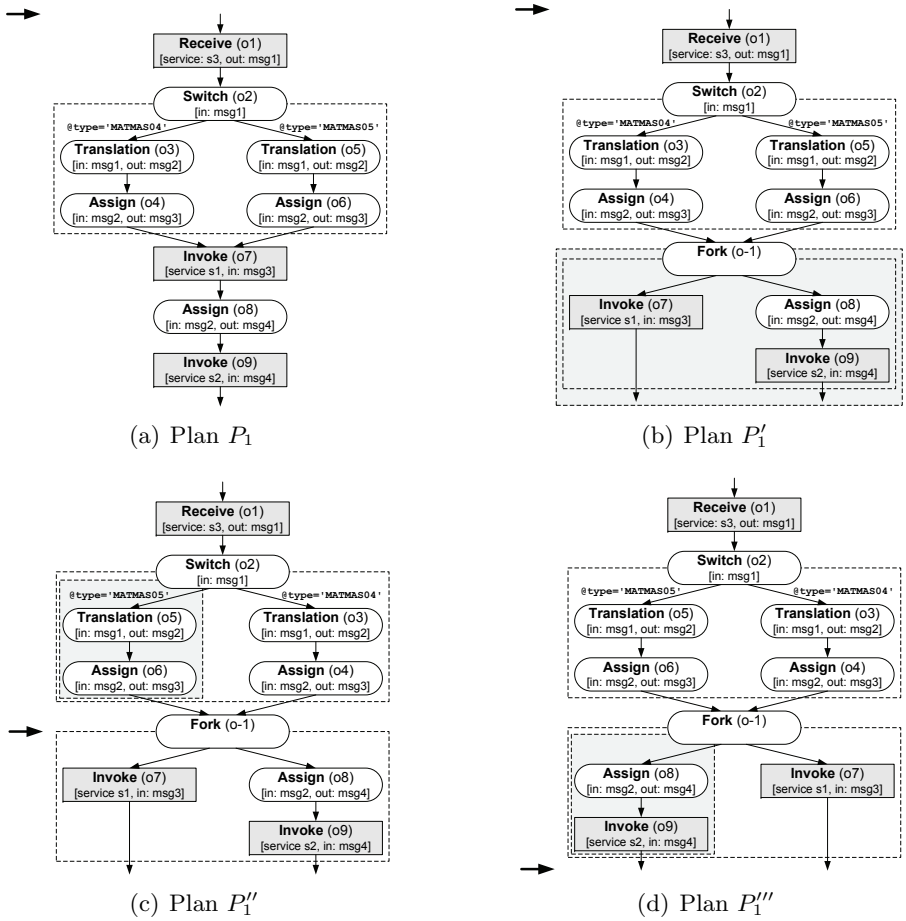
```
1:  if type(op) is Plan then
2:    apply Plan techniques on op (Before)          // WD10 (DPSize), WD13, MFO
3:  o ← op.getSequenceOfOperators()
4:  apply WC2 on o                                  // apply technique for all sequences
5:  for i ← 1 to |o| do                             // for each operator of the sequence
6:    if type(o_i) ∈ (Plan, Switch, Fork, Iteration, Undefined) then   // complex
7:      o_i ← A-PMO(o_i, DG)
8:      apply operator techniques on o_i            // WC1, WC4, WD1, WD2, WM1, WC3
9:    else                                          // atomic
10:     apply operator techniques on o_i            // WM1, WM2, WM3, WD3, WD4, WD5,
                                                    // WD6, WD8, WD9, WD11, WD12
11: if type(op) is Plan then
12:   apply Plan techniques on op (After)           // Vect, HLB
```

enumeration is only executed once for the complete plan. Within this full optimization algorithm, we use the DPSize [SAC+79, Moe09] join enumeration algorithm. Another example is the optimization technique *multi-flow optimization* (see Chapter 5). Second, for complex operators (line 6), we recursively invoke this algorithm and subsequently, apply available optimization techniques. Third, we apply operator-type-specific techniques for the individual atomic operators (line 9). Note that from each operator, parent nodes and all other operators are reachable. From the perspective of a single optimization technique, however, only successors (following operators) are considered (forward-only) in order to avoid to consider the same operator multiple times. Fourth, we apply all techniques, which need to be executed on top level of a plan and after the operator-type-specific techniques. Among others, we invoke the optimization technique *vectorization* (see Chapter 4). In contrast to the full plan enumeration with dynamic programming approaches, this iterative, transformation-based algorithm preserves the control-flow semantics of the given plan and it iteratively improves the current solution. Thus, it can be aborted between applying optimization techniques without loss of intermediate optimization results.

The worst-case time complexity of the A-PMO is given by the optimization technique with the highest individual complexity. In our case this is the optimization technique join enumeration, where we use the DPSize join enumeration algorithm. According to the complexity analysis of DPSize [MN06] by Moerkotte and Neumann, it is given by $O(n^4)$ for chain and cycle queries as well as by $O(c^n)$ for star and clique queries, where n denotes the number of joined input data sets.

Example 3.4 (Optimization Algorithm)**.** *Recall the plan P_1 that is shown in Figure 3.7(a). The A-PMO recursively iterates over all operators and applies available optimization techniques. We start at the top-level sequence of operators, where we apply the rewriting of sequences to parallel flows*[5] *because no data dependencies exists between o_7 and o_8. The resulting plan P_1' is shown in Figure 3.7(b). Then, we iterate over the individual operators.*

[5] Rule-based optimization techniques are applied during the initial deployment of a plan. As an example consider the operators o_4 and o_6, which would be detected as redundant work and thus merged to a single operator after o_2. This operator would have been included in the parallel flow of operator o_7. For simplicity of presentation, we did not apply rule-based optimization techniques in the example.



(a) Plan P_1 (b) Plan P_1'

(c) Plan P_1'' (d) Plan P_1'''

Figure 3.7: Example Execution of the Optimization Algorithm

At operator o_1 we find no optimization technique. At operator o_2, we recursively invoke the algorithm for the operator sequences (o_3, o_4) and (o_5, o_6) but do not find a different plan. Afterwards, the reordering of switch paths according to their cost-weighted path probabilities is applied, where the resulting plan P'' is shown in Figure 3.7(c). Further, we apply the rescheduling of parallel flows in order to start the most time-consuming flow first (Figure 3.7(d)). Note that the application order of different optimization techniques influences the resulting plan. For this reason, we predefined this application order for existing optimization techniques. Finally, we recursively invoke the algorithm for the sequences (o_7) and (o_8, o_9) but do not find another plan.

This overall cost-based optimization algorithm represents the framework for applying arbitrary optimization techniques. We describe selected techniques in Section 3.4 but one can easily extend the set of used techniques with additional ones.

3.3.2 Search Space Reduction Heuristics

Our core optimization algorithm produces the globally optimal plan with regard to set of used optimization techniques and the monitored statistics. Additionally, we now provide heuristic optimization algorithms. Although these heuristics reduce the costs of pe-

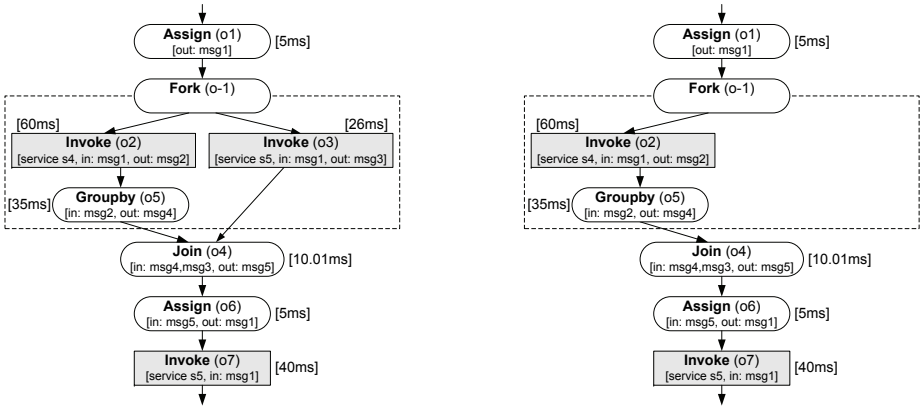
riodical re-optimization, we cannot guarantee to find the globally optimal plan. However, often good plans are produced by these heuristics. Essentially, we discuss the `criticalPathOptimization` algorithm (A-CPO) as well as the `heuristicPatternMatchingOptimization` algorithm (A-HPMO), which are both based on our standard A-PMO.

Critical Path Optimization

The application of our first heuristic, *critical path optimization*, promises optimization time reduction but does not change the asymptotic worst-case time complexity of the optimization algorithm. The critical path $cp(P)$ of the plan is determined in the form of operator execution times [LZ05]. We then apply the A-PMO only for this critical path and thus we might reduce the number of operators evaluated by the optimization algorithm from m to m', where m' denotes the number of operators included in the critical path with $m' = |o_i \in cp(P)|$ and $m' \leq m$.

The intuition of this heuristic is that the execution time of a plan instance does not depend on parallel subflows, which execution time is subsumed by more time-consuming subflows. As a result, we only benefit from this heuristic if the plan includes a `Fork` operator. This heuristic exploits the control-flow semantics of our flow meta model. In the following, we illustrate the concept of critical path optimization with an example.

Example 3.5 (Critical Path Optimization)**.** *Recall the plan P_3' from Example 3.2. Clearly, the critical path of a plan can only be different to this plan if and only if parallel subflows (`Fork` operator) exist. Then, we determine the most time-consuming subflow of each `Fork` operator as part of the critical path, while all other subflows are subsumed and hence, are marked as not to be optimized. Figure 3.8 illustrates the determined critical path of the running example P_3, where (o_3) is subsumed by (o_2, o_5). Hence, we have reduced the number of operators that are included in the optimization.*



(a) Plan P_3' (b) Plan $cp(P_3')$

Figure 3.8: Example Critical Path Optimization

This algorithm is a heuristic because we cannot guarantee to find the globally optimal plan for two reasons. First, the critical path might change during the optimization. As a consequence of not considering certain operators of the new critical path, we might get suboptimal plans. Second, we do not consider data-flow-oriented operators not included

within the critical path. When reordering selective operators (e.g., joins), this can have tremendous impact for following operators (that are included in the critical path). For example, we might exclude a join operator from the optimization because it is not within the critical path, which might lead to a suboptimal join order.

In conclusion, there are three cases where this critical-path approach is advantageous. First, it can be used if the optimization interval is really short and thus, optimization time is more important than in other cases. Second, it is beneficial if the parallel subflows are fully independent of the rest of the plan because then, we do not miss any global optimization potential. Third, its application is promising if there is a significant difference in the costs of the single subflows; otherwise, the critical path might change.

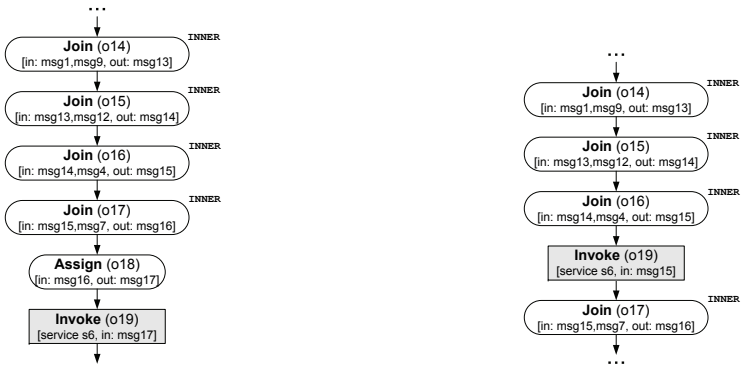
Heuristic Join Enumeration

Since the complexity of the A-PMO is dominated by the complexity of join enumeration, we typically use our tailor-made heuristic optimization algorithm, if the number of join operators of a plan exceed a certain number. In contrast, for the second problem with high complexity (merging parallel flows), we apply a heuristic by default because the plan cost influence of join enumeration is much higher than the number of parallel flows. Thus, the A-HPMO is essentially equivalent to the A-PMO except that we do not apply the full join enumeration but the heuristic described here.

Similar concepts are also used in DBMS, where existing approaches typically fall back to some kind of greedy heuristics or randomized algorithms if a certain optimization time is exceeded [Neu09]. In contrast, we use—similar to selected DBMS such as Postgres—the number of joins as an indicator when to use the heuristic because otherwise, intermediate results of join enumeration cannot be exploited when using the DPSize (bottom-up dynamic programming) join enumeration algorithm and thus, the elapsed optimization time would be wasted if we have to fall back to the heuristic.

Before discussing the join enumeration heuristic, we need to define the join enumeration restrictions that must be taken into account when reordering joins in order to preserve the semantic correctness. Most importantly, Rule 2 from Definition 3.1 applies. Thus, if there is a dependency between an interaction-oriented operator and another operator, the temporal order of them must be equivalent in P and P'. In addition to this, the input data of interaction-oriented operators (data that is sent to external systems) must also be equivalent in P and P' (preventing the external behavior from being changed). This has influence of applicable join re-orderings. We use an example to illustrate the consequences of these join enumeration restrictions.

Example 3.6 (Join Enumeration Restrictions)**.** *Recall the example plan P_7 and a slightly different plan P_7' (where we changed the initial order of the* `Invoke` *operator o_{19}) that are illustrated in Figure 3.9. While for plan P_7 (Figure 3.9(a)) a full reordering is applicable (in case of a clique query type, where all data sets are directly connected), for P_7' (Figure 3.9(b)), we are not allowed to reorder all* `Join` *operators of this plan. The plan P_7' is an example, where we might change the external behavior if we consider full join enumeration. The reason is, that the external system s_6 requires the result of operators o_{14}, o_{15} and o_{16}. During full join reordering, we might use operator o_{17} earlier in this chain of* `Join` *operators and thus, we would be unable to produce the required result in case of selective joins (or we need at least a combination of* `Selection` *and* `Projection` *operators in order to hide additional data). In conclusion, only partial join reordering*



(a) Plan P_7 (full reordering possible) (b) Plan P_7' (partial reordering possible)

Figure 3.9: Join Enumeration Example Plans

including the operators o_{14}, o_{15} and o_{16} (independently of the operator o_{17}) is possible. In contrast, for join enumeration in DBMS, the temporal order of table accesses does not matter when considering only the final query result because all joins can be considered by simply evaluating the connectedness of quantifiers (data sets).

In order to take into account the described join enumeration restrictions as well as the control-flow semantics of an integration flow, we introduce a tailor-made, transformation-based join enumeration heuristic. For the sake of clarity, we require some notation before discussing the join enumeration heuristic. For our heuristic join reordering, we do only consider (1) left-deep join trees (no composite inners [OL90] in the sense of bushy trees), (2) without cross-products, and (3) only one join implementation (nested loop join). Note that after join re-ordering, we still decide between different join operator implementations. Using these assumptions in combination with our asymmetric cost functions, there exist $n!$ alternative plans for joining n data sets. For example, assume a left-deep join tree $(R \bowtie S) \bowtie T$ $(n = 3)$ with the following $n! = 6$ possible plans:

$$\begin{array}{llllll} P_a(opt): & (R \bowtie S) \bowtie T & P_c: & (R \bowtie T) \bowtie S & P_e: & (S \bowtie T) \bowtie R \\ P_b: & (S \bowtie R) \bowtie T & P_d: & (T \bowtie R) \bowtie S & P_f: & (T \bowtie S) \bowtie R. \end{array}$$

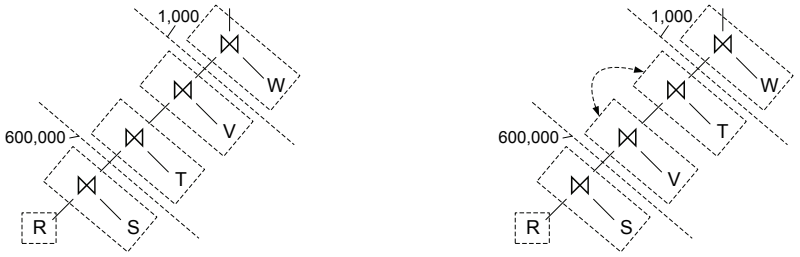
The join selectivity $f_{R,S}$ (filter selectivity) of $R \bowtie S$ is given by

$$f_{R,S} = \frac{|R \bowtie S|}{|R| \cdot |S|} \text{ with } f_{R,S} \in [0,1] \tag{3.6}$$

and the costs of the nested loop join are computed by $C(R \bowtie S) = |R| + |R| \cdot |S|$ (asymmetric, in order to take into account commutativity of join inputs). Further, the join output cardinality can be derived with $|R \bowtie S| = f_{R,S} \cdot |R| \cdot |S|$. Thus, the costs of the complete plan $(R \bowtie S) \bowtie T$ are given by

$$C((R \bowtie S) \bowtie T) = |R| + |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| \cdot |T|. \tag{3.7}$$

The core idea of our heuristic join reordering is to transform the full join enumeration into binary re-ordering decisions between subsequent join operators. This is possible because we restricted ourself to left-deep-join trees and nested loop joins only. We then can observe that the costs before and after a binary reordering decision are independent of



(a) Plan P (before reordering) (b) Plan P' (after reordering)

Figure 3.10: Heuristic Join Reordering Example

the order of these two operators as shown in Figure 3.10. For example, the costs of the `Join` operator $* \bowtie W$ are independent of the order of previous join operators because the size of the intermediate result (1,000) before this operator is constant and hence, these operator costs are also constant. This concept of binary cost-based reordering decisions take into account cardinality and selectivity of join operators and thus generalized the existing ranking functions *minSel*, *minCard*, and *minSize* [BGLJ10].

Algorithm 3.3 illustrates the heuristic join reordering in detail. Essentially, it uses a transformation-based approach, where the inputs of join operators are reordered. First, we select the input data set with the smallest cardinality (line 1) and reorder it with the existing first join operand (line 2). Second, we incrementally reorder subsequent join operands by iterating over all joins (line 3) and comparing the costs of the overall plan under the hypothesis that we reorder the join operand with a subsequent operand. Assuming variable selectivities and cardinalities, the cost comparison of subplans for arbitrary left-deep join trees are specified as follows. First, fix two input data sets with the commutativity optimality condition of $|R| \leq |S|$. Second, the optimality of executing $R \bowtie S$ before $* \bowtie T$ is given if the following optimality condition holds:

$$\begin{aligned} |R| + |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| \cdot |T| \\ \leq |R| + |R| \cdot |T| + f_{R,T} \cdot |R| \cdot |T| + f_{R,T} \cdot |R| \cdot |T| \cdot |S| \\ |S| + f_{R,S} \cdot |S| + f_{R,S} \cdot |S| \cdot |T| \\ \leq |T| + f_{R,T} \cdot |T| + f_{R,T} \cdot |T| \cdot |S|. \end{aligned} \tag{3.8}$$

Algorithm 3.3 Heuristic Join Reordering (A-HJR)

Require: set of input data sets R (with $n = |R|$)
1: $R_k \leftarrow \min_{R_i \in R} |R_i|$ *// determine smallest input*
2: `reorder`(R_1,R_k)
3: **for** $i \leftarrow 3$ **to** n **do** *// for each input*
4: **for** $j \leftarrow i - 1$ **to** 2 **do** *// for each predecessor input*
5: **if** $\neg$ `connected`(R_i, R_{j-1}) **then** *// is reordering impossible*
6: **break**
7: **if** $\neg$ `optimal`(R_i, R_j) **then** *// is reordering not meaningful*
8: **break**
9: `reorder`(R_j,R_i)
10: **return** R

We can monitor all cardinalities $|R|$, $|S|$, and $|T|$ but only the selectivities $f_{R,S}$ and $f_{(R \bowtie S),T}$. To estimate $f_{R,T}$, we need to derive it with $f_{R,T}\ \theta\ f_{(R \bowtie S),T}$, where θ is a function representing the correlation. If we assume statistical independence of selectivities, we can set $f_{R,T} = f_{(R \bowtie S),T}$. However, in general, the selectivities are derived from the present conjunction of join predicates. This cost comparison is applied for each subset of join operators of a plan that can be fully reordered. Note that this heuristic does not necessarily requires that the cost model exhibits the ASI property [Moe09]. In addition, simple sorting of join operands is not applicable due to the use of arbitrary correlation functions and the need for evaluating if join inputs are connected.

Although this algorithm often produces good results, obviously, it does not guarantee to find the optimal join order. This is reasoned by (1) the restrictions of considering only nested loop joins and left-deep-join trees, (2) the selection of the first join operand based on minimum cardinality, and (3) reordering only directly connected join operands rather than partial subtrees.

In contrast to our transformation-based join reordering heuristic, recent approaches of heuristic query optimization [BGLJ10] use merge-based techniques, where ranked subplans are merged iteratively to an overall plan. While this bottom-up approach is advantageous for declarative queries, our transformation-based reordering is more advantageous for imperative integration flows with regard to the characteristic of an initially specified plan.

In general, this heuristic join reordering algorithm exhibits a quadratic time complexity of $O(m^2)$, where m denotes the number of operators with $m = n - 1$ and n denotes the number of joined input data sets. This is reasoned as follows. First, we iterate over all n input data sets in order to determine the minimum cardinality. Second, we iterate over all input data sets and for each, compare the costs assuming a reordering with its predecessors (similar to selection sort). Thus, in total, we execute at most

$$n + \sum_{i=3}^{n} (i-2) = \frac{n^2 - n}{2} + 1 \tag{3.9}$$

iterations during this algorithm. Finally, note that—except the awareness of temporal dependencies (join enumeration restrictions)—this heuristic join reordering algorithm can be applied in data management systems as well. As a result, the use of this heuristic join enumeration algorithm (combined with an extended heuristic *first fit* algorithm for merging parallel flows that we will describe in Section 3.4) reduces the overall complexity of the periodic plan optimization problem to polynomial time, where most of our other optimization techniques exhibit a linear or quadratic time complexity.

3.3.3 Workload Adaptation Sensibility

The core optimization algorithm can be influenced by a number of parameters. We can leverage these parameters in order to adjust the sensibility of adaptation to changing workload characteristics. For our core estimation approach, workload statistics of the current plan P are monitored. If an alternative plan P' has been created, we estimate the missing operator statistics with $\hat{W}(o'_i) = C(o'_i)/C(o_i) \cdot W(o_i)$ using our cost model, where workload statistics of P are aggregated over the sliding time window. In this context, the following three parameters influence the sensibility of workload adaptation:

Workload Sliding Time Window Size Δw (time interval used for statistic aggregation): Monitored statistics of plan instances p_i with $p_i \in [T_k - \Delta w, T_k]$ are included in the

current sliding window used for cost estimation. Increasing the sliding window size Δw results in a slower adaptation because statistics are computed over long time intervals. For example, when using average-based aggregation methods, a long sliding time window causes a slow adaptation because the overall influence of the most recent statistic tuples is fairly low. As a result, the parameterization is a trade-off between robustness of estimates and fast adaptation to changing workload characteristics. A short time window leads to high influence of single outliers (not robust but fast adaptation), while a long time window takes long histories into account (robust but slow adaptation).

Optimization Interval Δt: An increasing Δt will also cause a slower adaptation because no re-optimization (and thus, no re-estimation) is initiated during this optimization interval. The longer the optimization interval, the longer we rely on historic estimates. This parameter only affects the number of estimation points rather than influencing the estimation itself. Thus, this is also a trade-off between the costs of re-optimization and the fast adaptation to changing workload characteristics.

Workload Aggregation Method (method used to aggregate statistics over the sliding window): The choice of the workload aggregation method also influences the adaptation sensibility. For workload aggregation over a sliding time window of length Δw, which contains statistics (equi-distant time series) of n plan instances, our statistic estimator uses the following four aggregation methods in order to compute the one-step-ahead forecast at timestamp t and we assume that this estimate stays constant for the next optimization interval. As an example, we illustrate the aggregation of operator execution times $W(o_j)$:

- Moving Average (`MA`):

$$\mathrm{MA}_t = \frac{1}{n}\sum_{i=1}^{n} W_i\left(o_j\right) \tag{3.10}$$

- Weighted Moving Average (`WMA`):

$$\mathrm{WMA}_t = \left(\left(\sum_{i=1}^{n}\left(w_i \cdot W_i\left(o_j\right)\right)\right) / \sum_{i=1}^{n} w_i \text{ with } w_i = \frac{i}{2}\right) = \frac{\sum_{i=1}^{n}\left(w_i \cdot W_i\left(o_j\right)\right)}{\frac{n\cdot(n+1)}{4}} \tag{3.11}$$

- Exponential Moving Average (`EMA`):

$$\begin{aligned} &\mathrm{EMA}_1 = W_i\left(o_j\right) \\ &\mathrm{EMA}_t = \mathrm{EMA}_{t-1} + \alpha \cdot \left(W_i\left(o_j\right) - \mathrm{EMA}_{t-1}\right) \text{ with } \alpha = 0.05, 1 \leq i \leq n \end{aligned} \tag{3.12}$$

- Linear Regression (`LR`):

$$\begin{aligned} &\mathrm{LR}_t = a + b \cdot x \text{ with } x = n+1 \\ a =& \frac{1}{n}\sum_{i=1}^{n} W_i(o_j) - b \cdot \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\sum_{i=1}^{n} W_i\left(o_j\right) - b \cdot \frac{n+1}{2} \\ b =& \frac{n\sum_{i=1}^{n}\left(i \cdot W_i(o_j)\right) - \sum_{i=1}^{n} i \cdot \sum_{i=1}^{n} W_i(o_j)}{n\sum_{i=1}^{n} i^2 - \left(\sum_{i=1}^{n} i\right)^2} = \frac{\sum_{i=1}^{n}\left(i \cdot W_i\left(o_j\right)\right) - \frac{(n+1)}{2}\sum_{i=1}^{n} W_i\left(o_j\right)}{\frac{1}{12}(n^3 - n)}. \end{aligned} \tag{3.13}$$

The MA causes the slowest adaptation because of the simple average, where all items are equally weighted, while WMA (linear weights) and EMA (exponential weights) support a faster adaptation due to the highest influence of the latest items. However, LR achieves the fastest adaptation because the estimate is extrapolated from the last items. Unfortunately, LR tends to over- and underestimate on abrupt changes. Further, full re-computation with all methods can be realized with linear complexity of $O(n)$. A detailed explanation of the different adaptation parameters with regard to monitored execution statistics is given within the experimental evaluation. We also experimented with Polynomial Regression (PR) with up to a degree of four. Due to high over- and underestimation on abrupt changes, we do not use this aggregation method. Apart from these aggregation methods, one can use forecast models types [DB07, BJR94] in order to detect reoccurring patterns and thus, increase the accuracy of estimation. However, doing so is a trade-off between estimation overhead and benefit achieved by more accurate estimation. Experiments have shown that the simple exponential moving average (EMA) robustly achieves the highest accuracy with low estimation overhead such that we use this method as our default workload aggregation strategy. There, we do not use any automatic parameter estimation. However, the automatic evaluation and adaptation of this parameter could be seamlessly integrated into the periodical re-optimization framework.

Recall the parameters optimization interval Δt and sliding time window size Δw. If $\Delta t \geq \Delta w$, statistic tuples are only used at one specific optimization timestamp. In this case, we simply compute those statistics from scratch using the workload aggregation methods. However, if $\Delta t < \Delta w$, this is an inefficient approach because we aggregate portions of statistic tuples multiple times. In such a case, incremental maintenance of workload statistics—in the sense of updating the aggregate with new statistics—is required. In general, incremental statistics maintenance is possible for all of these aggregation methods. However, note that MA and LR require negative (implicitly removed tuples based on time) and positive maintenance (new tuples) according to the sliding time window size Δw (and thus, atomic statistics must be stored), while EMA and WMA does not require negative maintenance due to the increasing weights, where the influence of older tuples can be neglected. In conclusion, we use the EMA as default workload aggregation method.

Finally, the workload aggregation can be optimized. Similar to the *basic counting algorithm* [DGIM02], approximate incremental statistics maintenance over the sliding window—with summarizing data structures—is possible. Further, according to *workload shift detection* approaches [HR07, HR08], it might be possible to minimize the optimization costs by deferring the cost re-estimation until predicted workload shifts (anticipatory re-optimization).

3.3.4 Handling Correlation and Conditional Probabilities

In the context of missing knowledge about data properties of external systems, the main problem of cost estimation is correlation and conditional probabilities (more precisely, relative frequencies). Assume two successive Selection operators σ_A and σ_B. In fact, we are only able to monitor $P(A)$ and $P(B|A)$. A naïve approach would be to assume that all predicates and the resulting selectivities and cardinalities are independent. However, this can lead to wrong estimates [BMM+04] that would result in non-optimal plans or changing a plan back and forth.

Example 3.7 (Monitored Selectivities)**.** *Assume two successive* Selection *operators* σ_A

and σ_B and the following gathered statistics:

$$\begin{aligned}
\sigma_A: \quad & |ds_{in}| = 100, |ds_{out}| = 40 \qquad \sigma_B: \quad |ds_{in}| = 40, |ds_{out}| = 8 \\
\Rightarrow & P(A) = 0.4 \qquad \textit{//monitored} \\
& P(B) = P(B|A) = 0.2 \qquad \textit{//computed assuming independence} \\
& P(A \wedge B) = 0.08. \qquad \textit{//monitored}
\end{aligned}$$

Assuming statistical independence, we set $P(B) = P(B|A)$ and based on the comparison of $P(A) > P(B)$, we reorder the sequence (σ_A, σ_B) to (σ_B, σ_A). Using the new plan, we gather the following statistics:

$$\begin{aligned}
\sigma_B: \quad & |ds_{in}| = 100, |ds_{out}| = 68 \qquad \sigma_A: \quad |ds_{in}| = 68, |ds_{out}| = 8 \\
\Rightarrow & P(B) = 0.68 \qquad \textit{//monitored} \\
& P(A) = P(A|B) \approx 0.12 \qquad \textit{//computed assuming independence} \\
& P(A \wedge B) = 0.08. \qquad \textit{//monitored}
\end{aligned}$$

Clearly, $P(A)$ and $P(B)$ are strongly correlated ($P(\neg A \wedge \neg B) = 0$). Due to the simplifying assumption of independence, we would assume that $P(B) > P(A)$. Hence, we would reorder the operators back to the initial plan. As a result, even in the presence of constant statistics, we would reorder the plan back and forth and thus, produce inefficient plans.

Our approach of *conditional selectivities* explicitly takes those conditional probabilities into account in order to overcome that problem when reordering selective data-flow-oriented operators (e.g., `Selection`, `Projection` with duplicate elimination, `Join`, `Groupby`, `Setoperation`) or the paths of a `Switch` operator. Essentially, we maintain selectivity statistics over multiple versions of a plan, independently of the sliding time window statistics. Therefore, for each pair of data-flow-oriented operators with direct data dependency within the current plan, we maintain a row of selectivities:

$$(o_1, o_2, P(o_1), P(o_2), P(o_1 \wedge o_2)), \tag{3.14}$$

where both operators o_1 and o_2 are identified, and we store the selectivity as well as the conjunct selectivity of both operators. The approach works similar for binary operators and reordered `Switch` paths. Due to the binary comparison approach of only two operators at-a-time, the overhead is fairly low. In the worst case, there are m^2 selectivity tuples, where m denotes the number of operators. We revisit the example in order to explain that concept in detail.

Example 3.8 (Monitored Conditional Selectivities)**.** *Assume the same setting as in Example 3.7. When reordering σ_A and σ_B at timestamp T_1, we create a new statistic tuple as shown in Table 3.5.*

Table 3.5: Conditional Selectivity Table

	$\underline{o_1}$	$\underline{o_2}$	$P(o_1)$	$P(o_2)$	$P(o_1 \wedge o_2)$
T_1	σ_A	σ_B	0.4		0.08
T_2	σ_A	σ_B	0.4	0.68	0.08

We only include probabilities that are known not to be conditional (the first operator and the combination of both operators). If we evaluate the temporal order of those operators

at T_2, we enrich the selectivity tuple with the missing information (due to reordering, we now have independent statistics for both operators). Over time, we use the exponential moving average (`EMA`) to adapt those statistics independently of the sliding time window. Finally, we see that the sequence of (σ_A, σ_B) is the best choice because $P(A) < P(B)$. We might make wrong decisions at the beginning but the probability estimates converge to the real statistics over time. Hence, we do not make wrong decisions twice as long as the probability comparison and the data dependency between both operators do not change.

As long as a selectivity tuple contains a missing value for an operator o_2, we make the assumption of statistical independence and hence compute $P(o_2) = P(o_1 \wedge o_2)/P(o_1)$. Furthermore, we compute the conditional probabilities for a given operator—based on the described selectivity tuple—as follows:

$$P(o_2|o_1) = \frac{P(o_1 \wedge o_2)}{P(o_1)}$$
$$P(o_1|o_2) = \begin{cases} \dfrac{P(o_1 \wedge o_2)}{P(o_2|o_1)} = P(o_1) & P(o_2) = \texttt{NULL} \\ \dfrac{P(o_1 \wedge o_2)}{P(o_2)} & \text{otherwise.} \end{cases} \tag{3.15}$$

Thus, we explicitly use the monitored conditional probabilities if available. It is important to note that we can only maintain the probability of the first operator as well as the joint probability of both operators. Hence, the additional problem of starvation in reordering decisions might occur if the real probability of the second operator decreases because we cannot monitor this effect. In order to tackle this problem of starvation in certain rewriting decisions, we use an aging strategy, where the probability of the second operator is slowly decreased over time. This prevents starvation because over time we reorder both operators due to the decreased probability and can monitor the actual probability of this second operator. Although this can cause suboptimal plans in case of no workload changes, it prevents starvation and converges to the real selectivities and probabilities.

For arbitrary chains of operators, the correlation table is used recursively along operators that are directly connected by data dependencies such that an operator might be included in multiple entries of the correlation table. For such chains, we allow only reordering directly connected operators. After a reordering, all entries of the correlation table that refer to a removed data dependency are removed as well and new entries for new data dependencies are created.

In conclusion, the adaptive behavior of our condition selectivity approach ensures more robust and stable optimizer decisions for conditional probabilities and correlated data, even in the context of changing workload characteristics. Note that there are more sophisticated, heavyweight approaches (e.g., by using a maximum entropy approach [MHK+07, MMK+05] or by using a measure of clusteredness [HNM03]) for correlation-aware selectivity estimation in the context of DBMS. We could also maintain adaptable multi-dimensional histograms [BCG01, AC99, LWV03, MVW00, KW99]. However, in contrast to DBMS, where data is static and statistics are required for arbitrary predicates, we optimize the average plan execution time with known predicates but dynamic data. Hence, the introduced lightweight correlation table can be used instead of heavyweight multi-dimensional histograms, where we would need to maintain exact or approximate frequencies with regard to arbitrary conjunct predicates [Pol05, TDJ10]. However, other approaches can be integrated as well. Due to the problem of missing statistics about

data properties of external systems, we use the described lightweight concept for explicitly taking into account conditional probabilities and correlation at the same time using a incremental reordering approach that is tailor-made for integration flows.

3.4 Optimization Techniques

In this section, we discuss specific optimization techniques that are used within the described core optimization algorithm. For selected techniques, we present the core idea, the optimality conditions, possible execution time reduction, the rewriting algorithm and its time complexity, as well as possible side effects to other techniques.

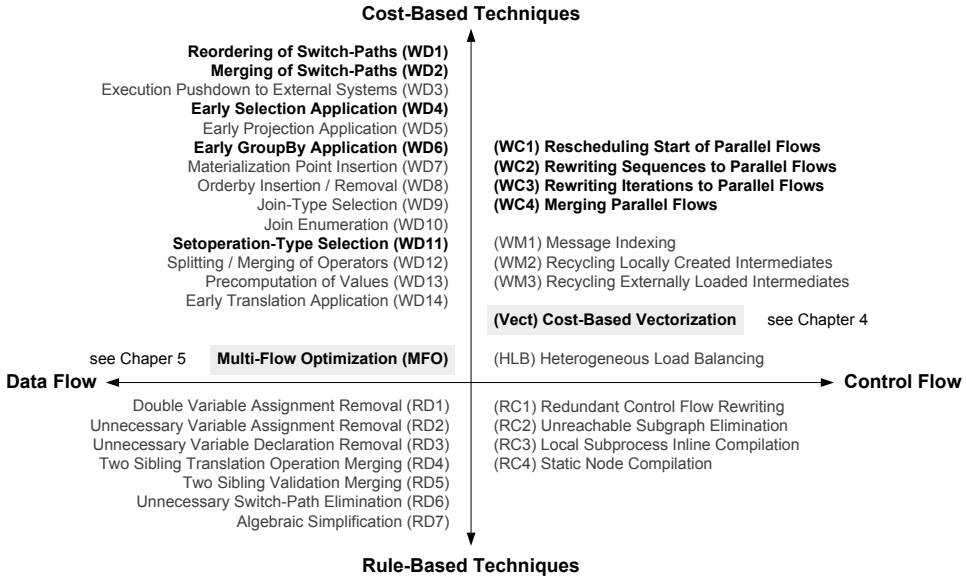


Figure 3.11: Cost-Based Optimization Techniques

Figure 3.11 distinguishes the used cost-based optimization techniques into control-flow-oriented and data-flow-oriented optimization techniques and emphasizes (with bold font) the techniques that we will describe in detail. All optimization techniques presented in this section follow the optimization objective of minimizing the average execution time of a plan. In addition to these techniques, we will discuss in very detail the cost-based vectorization in Chapter 4 and the multi-flow optimization in Chapter 5, which both follow the optimization objective of maximizing the message throughput. Apart from these techniques, we refer the interested reader to our details on message indexing [BHW+07, BHLW08d] and rule-based optimization techniques [BHW+07] that are omitted in this thesis. The latter includes, for example, relational algebraic simplifications as described by Dadam [Dad96].

3.4.1 Control-Flow-Oriented Techniques

Control-flow-oriented optimization techniques address the interaction- and control-flow-oriented operators of our flow meta model. These techniques try to exploit the specific characteristics of operators like alternatives (`Switch` operator), loops (`Iteration` operator), and parallel subflows (`Fork` operator, constrained by `Invoke` operators).

One of the core concepts is to leverage parallelism of operator execution in order to minimize the execution time. Due to typically low CPU utilization reasoned by (1) single-threaded, instance-based plan execution, (2) significant waiting times for external systems, and (3) IO-bottlenecks for message persistence, these decisions are made with cost-based optimality conditions rather than statically during the initial deployment.

Rescheduling the Start of Parallel Flows

The technique *WC1: Rescheduling the Start of Parallel Flows* rewrites existing `Fork` operators. The execution time of the `Fork` operator o is determined by its most time-consuming subflow r_i with

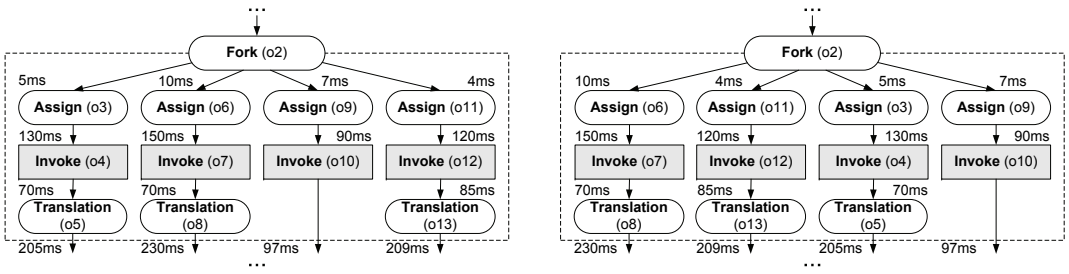
$$W(o) = \max_{i=1}^{|r|} \left(\sum_{j=1}^{m_i} W\left(o_{i,j}\right) + i \cdot W\left(\text{Start\_Thread}\right) \right) \tag{3.16}$$

because the individual subflows are started in sequence and temporally joined (synchronized) at the end of this operator.

The core principle of this technique is to reduce waiting time by rewriting the concurrent subflows such that the subflows are started in descending order of their execution time. In other words, the start sequence of parallel subflows is optimal if the condition $\hat{W}(r_i) \geq \hat{W}(r_{i+1})$ holds, where $\hat{W}(r_i) = \sum_{j=1}^{m_i} \hat{W}(o_{i,j})$. The execution time can be reduced by $(|r|-1) \cdot W(Start\_Thread)$, where $W(Start\_Thread)$ denotes the constant costs for creation and start of a thread.

The rewriting algorithm essentially consists of two steps. First, for each subflow, we recursively sum up the costs of all individual operators. Second, we check if the optimality condition holds and order the subflows according to the estimated costs if required. The time complexity of this algorithm is given by $O(|r| \cdot \log|r|)$.

Example 3.9 (Rescheduling Parallel Flows)**.** *Assume our example plan P_7 and the monitored execution times shown in Figure 3.12(a). Furthermore, assume the cost for starting of a parallel subflow (thread) to be $W(Start\_Thread) = 3\,\text{ms}$. The estimated costs of the* `Fork` *operator are then given by $\hat{W}(o_2) = 2 \cdot 3\,\text{ms} + 230\,\text{ms} = 236\,\text{ms}$. Rescheduling the parallel flows yields the alternative plan shown in Figure 3.12(b). Using this plan, the costs are reduced to $\hat{W}(o_2) = 3\,\text{ms} + 230\,\text{ms} = 233\,\text{ms}$.*



(a) Plan P_7 (b) Optimized Plan P_7'

Figure 3.12: Example Rescheduling Parallel Flows

Clearly, the benefit of this optimization technique is limited but the potential increases with increasing number of subflows. This technique should be used after WC2 and WC3

(parallelizing sequences and iterations) because otherwise we might miss optimization opportunities. Furthermore, this technique should be used also after WC4 (merging parallel flows) because otherwise, we might perform unnecessary optimization efforts.

Rewriting Sequences to Parallel Flows

The technique *WC2: Rewriting Sequences to Parallel Flows* is used to optimize sequences of operators. Such sequences can be found as implicit children of each `Plan`, `Switch` path, `Fork` subflow, and `Iteration`. The costs of a sequence of operators o is given by the sum of their execution times with $W(o) = \sum_{i=1}^{m} W(o_i)$.

The core concept is to rewrite a sequence of operators to parallel subflows of a `Fork` operator by analyzing the dependencies between the single operators. Recall that the execution time of the `Fork` operator is determined by the subflow with highest cost. For analyzing the optimality of such a rewriting, we take into account the number of logical processors (hardware threads) k as well as the CPU utilization of the involved operators. There, the CPU utilization of an operator with regard to a single logical processor is computed by $(W(o_i) - wait(o_i))/W(o_i)$, where the waiting time can be monitored (e.g., waiting time for external systems); otherwise we assume $wait(o_i) = W(o_i) \cdot 0.05$ as a heuristic. Such a rewriting of a sequence to $|r|$ parallel subflows is advantageous if

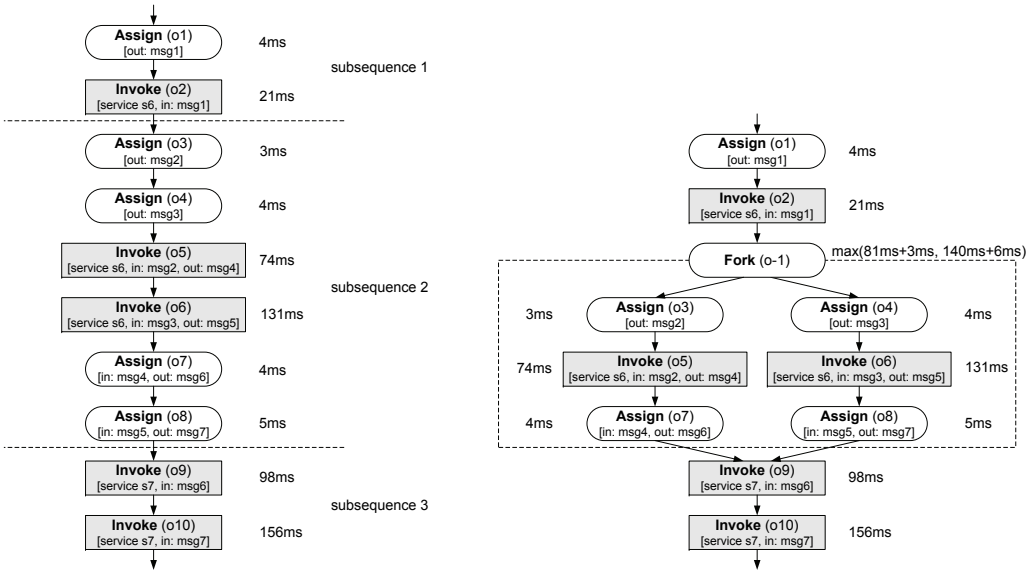
$$\max_{i=1}^{|r|} \left(\sum_{j=1}^{m_i} \hat{W}\left(o_{i,j}\right) + i \cdot W\left(\text{Start\_Thread}\right) \right) < \sum_{i=1}^{m} \hat{W}(o_i) \tag{3.17}$$

$$with\ \hat{W}\left(o_{i,j}\right) = \frac{|r|}{\min(|r|, k)} \cdot \left(W(o_{i,j}) - wait(o_{i,j})\right) + wait(o_{i,j}),$$

which means that the estimated most time-consuming parallel subflow must have lower costs than the plan sequence of operators. There, the costs of operators within parallel subflows are estimated by the waiting time plus the execution time that depends on the number of logical processors k and the number of parallel subflows $|r|$. Intuitively, this represents the increased execution time if parallel subflows share hardware resources. This is a worst-case consideration because for an exact model, the temporal overlap of waiting times and execution times would be required as well.

Rewriting sequences to parallel flows is realized with the following algorithm. First, we split the given sequence into disjoint subsequences according to Rule 3 of Definition 3.1 (preserve temporal order of writing interactions to the same external system). Second, for each of these subsequences, we create a new `Fork` operator and partition the individual operators, where we iterate over the operators and determine if they depend on other operators of the same subsequence. If so, we add this operator to the existing subflow, where the operator referred by the dependency exists; otherwise, we create a new subflow and add the operator as a child. If an operator depends on multiple operators from different subflows, this operator splits the subsequence into two subsequences. Third, if a `Fork` operator contains only one subflow, we rewrite it back to a simple sequence. Fourth, and finally, we check the optimality condition for each `Fork` operator. The time complexity of this algorithm is $O(m^2)$ due to the dependency checking for each operator. In the following, we use an example to illustrate this rewriting concept in more detail.

Example 3.10 (Rewriting Sequences to Parallel Flows)**.** *Recall our example plan P_8 that is essentially a sequence of operators and assume the monitored execution times shown in*



(a) Plan P_8 (b) Optimized Plan P_8'

Figure 3.13: Example Rewriting Sequences to Parallel Flows

Figure 3.13(a). The costs are estimated as $\hat{W}(P_8) = 500\,\text{ms}$. *Figure 3.13(b) illustrates the optimized plan* P_8'. *During rewriting, the operator sequence was split by the writing interactions* o_2 *(after* o_2 *because it is the first interaction) and* o_9 *(before* o_9 *in order to avoid executing it in parallel to* o_6*) into three subsequences due to Rule 3 of Definition 3.1 (no parallel interactions with the same external system if one is a writing interaction). Furthermore, for the second subsequence, we created a* `Fork` *operator with two subflows. If we assume* $k = 2$ *logical processors and a thread start time of* $W(\textit{Start\_Thread}) = 3\,\text{ms}$, *we estimate the costs of the* `Fork` *operator as follows:* $W(o_{-1}) = \max((81\,\text{ms}+3\,\text{ms}), (140\,\text{ms}+2 \cdot 3\,\text{ms})) = 146\,\text{ms}$ *As a result, we reduced the estimated plan costs to* $\hat{W}(P_8') = 425\,\text{ms}$, *while ensuring semantic correctness at the same time. There, the* `Invoke` *operators* o_5 *and* o_6 *can be executed in parallel because they are both reading interactions.*

This optimization technique offers high optimization opportunities. With regard to maximal optimization opportunities, this technique should be applied before the techniques WC1 (rescheduling parallel flows) and WC4 (merging parallel flows). In order to prevent the special case of local suboptima, we directly evaluate the techniques WC1 and WC4 for the resulting plan (e.g., P_8' from Example 3.10) if the parallel execution exhibits higher estimated costs than the serial execution.

Rewriting Iterations to Parallel Flows

Similar to rewriting sequences, the technique *WC3: Rewriting Iterations to Parallel Flows* rewrites an `Iteration` with r loops to a `Fork` operator with r concurrent subflows. This is applicable if there are no dependencies between operators of different iteration loops. However, *foreach* semantics without such dependencies are typical for integration flows.

The core idea is to use, similar to loop unrolling of programming language compilers for parallel environments, the estimated number of iterations in order to determine the

number of parallel subflows. The rewriting of iterations to parallel flows is beneficial if

$$\max_{i=1}^{|r|}\left(\sum_{j=1}^{m_i}\hat{W}\left(o_{i,j}\right)+i\cdot W\left(\text{Start\_Thread}\right)\right)<r\cdot\sum_{j=1}^{m_i}W\left(o_{i,j}\right) \tag{3.18}$$

$$with\ \hat{W}\left(o_{i,j}\right)=\frac{|r|}{\min(|r|,k)}\cdot(\hat{W}(o_{i,j})-wait(o_{i,j}))+wait(o_{i,j}).$$

Due to *foreach* semantics, r must be estimated in terms of the frequency of iterations. Rewriting is realized by the following algorithm. We check if there are no dependencies between iterations and if we are allowed to change the temporal order. In case of independence, we compile all operators of the iteration body to r subflows plus one additional subflow. Each subflow references a specific data partition of the inbound data set, while the last subflow is used for all partitions that exceed the estimated number of iterations.

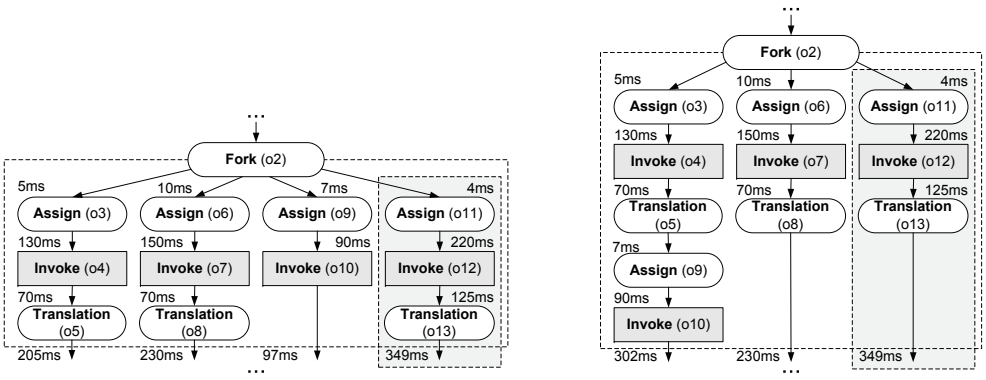
In conclusion, this optimization technique offers—similar to the rewriting of sequences—high optimization opportunities. In contrast to sequences, the rewriting relies heavily on the estimated number of iterations. The possibility of arbitrary input data sets might result in unused subflows (overestimation) or in executing multiple partitions with the last subflow (underestimation). One can further enhance this by using dynamic runtime scheduling of parallel subflows (e.g., guided self-scheduling [PK87], or factoring [HSF91]). In addition, this technique can be combined with rewriting sequences (the operators of one iteration) to parallel flows and this technique should be applied before the techniques WC1 (rescheduling parallel flows) and WC4 (merging parallel flows).

Merging Parallel Flows

Recall the costs of a `Fork` operator that are determined by the most time-consuming subflow. The idea of the technique *WC4: Merging Parallel Flows* is that if the costs of the subflow with maximum costs subsume the costs of two or more other subflows, the subsumed subflows can be rewritten to one subflow in order to reduce the costs by $W(Thread)$ that denotes the costs for thread creation, starting, and monitoring (in contrast to the previously used $W(Start\_Thread)$ that denotes the time required for thread creation only). Therefore, all `Fork` operators with more than two subflows have to be considered. We achieve an execution time reduction because less threads are required, while the most time-consuming subflow is unchanged.

In general, this problem of a given maximum constraint per partition and the optimization objective of minimizing the number of partitions, is reducible to the offline *bin packing* problem that is known to be an NP-hard problem. Therefore, we use an extension of the heuristic *first fit* algorithm [Joh74] that works as follows. First, we determine the subflow with maximum costs. Second, for each old subflow, we check if it can be merged with an existing new subflow. If this is possible, we merge the subflow with the *first fit* subflow by temporally concatenating the sequences of operators; otherwise, we create a new subflow. In the worst case, for each old subflow, we check each new subflow. Thus, the time complexity is given by $O(m^2)$. In the following, we use an example to illustrate this heuristic algorithm.

Example 3.11 (Merging Parallel Flows)**.** *Recall our example plan P_7 from Example 3.9. Further, assume changed execution times as shown in Figure 3.14(a). First, we determine the fourth subflow (*349 ms*) as upper bound for our extended first fit algorithm. Second,*



(a) Plan P_7 (b) Optimized Plan P_7'

Figure 3.14: Example Merging Parallel Flows

we iterate over all subflows in order to minimize the total number of subflows. There, for the first old subflow, we create a new subflow. Then, we check if the second old subflow fits into the first. Due to $205\,\text{ms} + 230\,\text{ms} > 349\,\text{ms}$*, we create a second new subflow. Subsequently, we repeat this with the third old subflow and observe that we can merge it with the first new subflow. Finally, the fourth old subflow, obviously, cannot fit into an existing subflow and hence, we create a third new subflow. Figure 3.14(b) illustrates the result of this algorithm.*

As a result, this optimization technique reduces the number of required threads without sacrificing the estimated execution time of a `Fork` operator. Due to less synchronization and thread monitoring overhead, this results in execution time reductions. Note that this technique should be applied after WC2 and WC3 (rewriting sequences and iterations to parallel flows), but before WC1 (rescheduling parallel flows).

Apart from these purely control-flow-oriented optimization techniques, there are several optimization techniques that can be classified as hybrid techniques because they combine data-flow- and control-flow-oriented aspects in order to achieve lower execution time. We will discuss them within the context of data-flow oriented optimization techniques.

3.4.2 Data-Flow-Oriented Techniques

The data-flow-oriented optimization techniques address the interaction-oriented operators and the data-flow-oriented operators of our flow meta model. This includes various techniques from traditional query processing, hybrid techniques combining data-flow and control-flow-oriented aspects, and techniques that are tailor-made for integration flows.

Reordering and Merging of Switch Paths

Programming language compilers and modern processors use only path probabilities for static and dynamic branch prediction in order to support speculative path computation. In contrast, expression evaluation within integration flows is typically more expensive due to the expression evaluation on messages (e.g., XPath expressions on XML messages).

The techniques *WD1: Reordering of Switch Paths* and *WD2: Merging of Switch Paths* are applied to the `Switch` operator depending on the workload characteristics. For this

purpose, the path probabilities $P(path_i)$ (as relative frequencies over the sliding window) and the absolute costs for evaluation of a path expression $W(expr_{path_i})$ are needed in order to compute the relative costs for accessing a `Switch` path with $W(expr_{path_i})/P(path_i)$.

As the core concept of *WD1*, we reorder `Switch` path expressions according to their relative costs for expression evaluation. When applying this technique we need to ensure the semantic correctness, where the structure of an expression is assumed to be a set of predicates *attribute θ value*. We define that only independent expressions (e.g., annotated within the flow specification) can be reordered, while conditional expressions prevent any reordering. This reordering of `Switch` paths is optimal (in the average case) if `Switch` paths are sorted in ascending order of their relative costs, such that the following optimality condition holds:

$$\frac{W\left(expr_{path_i}\right)}{P\left(path_i\right)} \leq \frac{W\left(expr_{path_{i+1}}\right)}{P\left(path_{i+1}\right)}. \tag{3.19}$$

With such a reordering, an execution time reduction of

$$\Delta W(path_i, path_{i+1}) = P\left(path_i\right) \cdot W\left(expr_{path_{i+1}}\right) - P\left(path_{i+1}\right) \cdot W\left(expr_{path_i}\right) \tag{3.20}$$

is possible when reordering two paths $path_{i+1}$ and $path_i$.

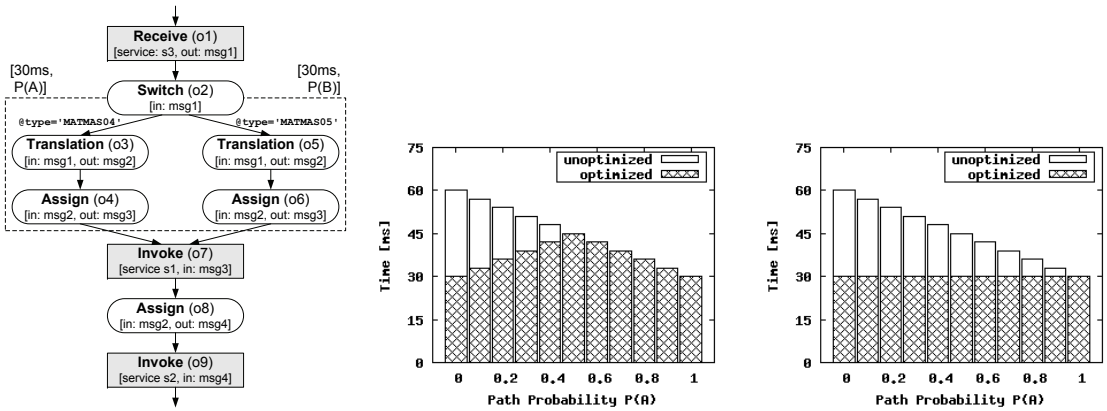
In contrast to the reordering of independent expressions, for any expressions that refer to the same attribute, the technique *WD2* can be applied. There, the concept is to merge expressions with equivalent attribute to a *compound switch path* in order to extract the single value only once and to evaluate it multiple times. With such a merged path evaluation, an execution time reduction of

$$\Delta W\left(path_i, path_{i+1}\right) = P\left(path_{i+1}\right) \cdot W\left(expr_{path_{i+1}}\right) \tag{3.21}$$

can be achieved. The *compound path* can be reordered similar to normal `Switch` paths. In consequence, the technique WD2 should be applied before WD1.

The following rewriting algorithm applies the reordering and merging of `Switch` paths. First, we partition the expressions, according to the attributes (e.g., represented by XPath expressions). If a partition contains multiple paths, we apply the merging by replacing the two paths with one compound path that writes the extracted attribute value to an operator-local cache and evaluates it multiple times. Therefore, all subpaths of the compound path are annotated as compound. Second, we compute the relative costs $W(expr_{path_i})/P(path_i)$ for each path and reorder the path according to the given optimality condition. In total, this rewriting algorithm exhibits a complexity of $O(m^2) = O(m^2 + m \cdot \log m)$ due to partitioning and sorting of `Switch` paths. We use an example to illustrate the resulting execution time when using these techniques.

Example 3.12 (Reordering and Merging Switch Paths)**.** *Recall our example plan P_1 that is shown in Figure 3.15(a). Further, assume that the costs for accessing each of the two* `Switch` *paths has been monitored as $W(expr) = 30\,\text{ms}$. We analytically investigate the influence of varying path probabilities $P(A) \in [0, 1]$ with $P(A) + P(B) = 1$. Figure 3.15(b) illustrates the influence of reordering the switch paths (assuming independent expressions, e.g., $A : var1 = x$ and $B : var2 = y$), where the costs are computed by $P(A) \cdot W(expr) + (P(A) \cdot W(expr) + P(B) \cdot W(expr))$ due to the ordered if-elseif semantic. Based on the equivalence of costs for evaluating the expressions, we benefit from reordering if $P(A) < P(B)$. This means, for $P(A) < 0.5$, we reorder (A, B) to (B, A) and thus, achieve the shown benefits. Furthermore, Figure 3.15(c) illustrates the influence of merging switch*



(a) Plan P_1 (b) Reordering Switch Paths (c) Merging Switch Paths

Figure 3.15: Example Reordering and Merging of Switch Paths

paths (assuming non-disjoint expressions, e.g., $A : var1 < x$ and $B : var1 < y$), where the total costs are independent of the path probabilities because the XPath expression is only evaluated once. Therefore, we benefit from merging if $P(A) < 1$.

Finally, note that the monitored path probabilities are conditional probabilities due to the ordered if-elseif-else semantics of the `Switch` operator. For example, we monitor the relative frequency of $P(path_1)$ but the conditional frequency of $P(path_2|path_1)$. Please, refer to Subsection 3.3.4 on how to estimate conditional probabilities in this context.

Selection Reordering

Similar to traditional query processing, reordering of selective operators such as `Selection`, `Projection` (distinct), `Groupby`, `Join`, and `Setoperation` (distinct) is important in order to find the optimal plan that reduces the amount of processed data as early as possible. In contrast to existing approaches, in the context of integration flows, the control-flow semantics must be taken into account when evaluating selective operators. Essentially, this control-flow awareness applies to all selective data-flow-oriented operators. However, we use the technique *WD4: Early Selection Application* in order to explain this control-flow-awareness.

The core idea of selection reordering is to reduce the amount of processed data by reordering `Selection` operators by their selectivity $f_{o_i} = |ds_{out}|/|ds_{in}|$, where $f_{o_i} \in [0, 1]$. The costs of a single `Selection` operator is given by $|ds_{in}|$. Thus, the costs of a sequence of `Selection` operators are determined by

$$C(P) = \sum_{i=1}^{m} |ds_{in}(o_i)| = \sum_{i=1}^{m} \left(\prod_{j=1}^{i-1} f_{o_j} \cdot |ds_{in}(o_1)| \right). \tag{3.22}$$

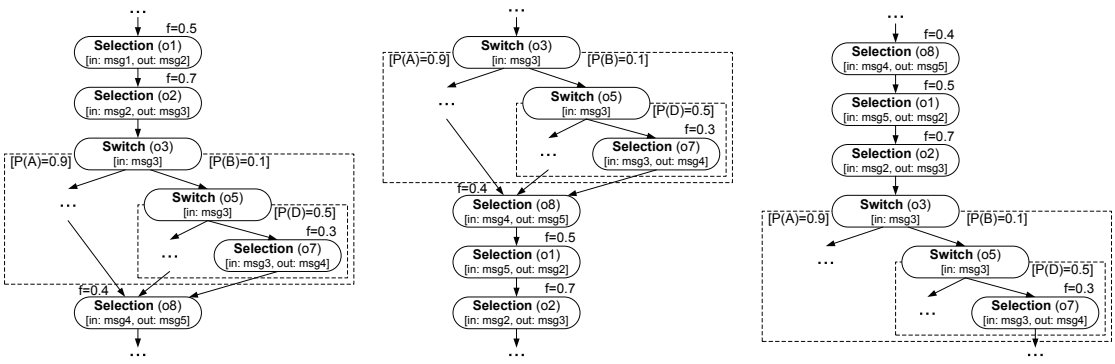
This implies that the order of `Selection` operators is optimal if $f_{o_i} \leq f_{o_{i+1}}$. Due to the problem of data correlation, the first optimization of a plan orders the `Selection` operators according to this optimality condition, while all subsequent optimization steps use the introduced correlation table for correlation-aware incremental re-ordering.

In order to achieve control-flow awareness, we additionally need to take into account the path probabilities of (possibly hierarchically structured) `Switch` operators in the form of probabilities $P(o_i)$ that an operator is executed. As a result, the costs of a sequence of `Selection` operators are determined by

$$\begin{aligned} C(P) &= \sum_{i=1}^{m} \left(P(o_i) \cdot |ds_{in}(o_i)| \right) \\ &= \sum_{i=1}^{m} \left(\prod_{j=1}^{i-1} \left(P(o_j) \cdot f_{o_j} + (1 - P(o_j)) \right) \cdot |ds_{in}(o_1)| \right), \end{aligned} \tag{3.23}$$

where $(P(o_i) \cdot f_{o_i} + (1 - P(o_i)))$ denotes the effective operator selectivity. The order of `Selection` operators is still optimal if $f_{o_i} \leq f_{o_{i+1}}$ holds for all selective operators due to the conditional operator execution with $P(o_i)$. However, the effective operator selectivity must be taken into account when evaluating following operators. If a `Switch` operator contains multiple `Selection` operators (in different paths), we reorder only the complete `Switch` operator according to the total effective operator selectivity $P(o_i) \cdot f_{o_i} + P(o_j) \cdot f_{o_j}$. The actual rewriting algorithm works as follows. First, we determine the selectivities of each individual operator. Second, we reorder the operators according to the given optimality condition. Third, we evaluate if the increased costs of additional non-selective operators (such as the `Switch` operator) are amortized by applying the selective operator earlier. While the normal selection reordering exhibits a time complexity of $O(m \log m)$ due to the simple sorting according to the selectivities, the control-flow aware selection reordering exhibits a time complexity of $O(m^2) = O(m^2 + m \log m)$ because after sorting, for each operator, we additionally evaluate all subsequent operators. In the following, we use an example to illustrate the importance of this control-flow aware rewriting.

Example 3.13 (Selection Reordering)**.** *Assume a subplan P as illustrated in Figure 3.16(a) that contains four different* `Selection` *operators. Furthermore, assume the given monitored selectivities, path probabilities and an input cardinality of $|ds_{in}(o_1)| = 1{,}000$. For sake of simplicity the costs of a (hierarchically structured)* `Switch` *operator are computed by $|ds_{in}|$ and we use the abstract costs rather than the weighted costs for cost comparison. Then, the costs are determined as follows: $C(P) = 1{,}000+500+350+0.05 \cdot 350+0.965 \cdot 350 =$*



(a) Plan P (b) Normal Selection Reordering (c) Aware Selection Reordering

Figure 3.16: Example Control-Flow-Aware Selection Reordering

2,205.05, *where for example, the cost of the* `Switch` *operator* o_3 *are determined with* $C(P) = 1{,}000 \cdot 0.5 \cdot 0.7 = 350$ *by the input data size and the selectivity of previous operators. If we would reorder the* `Selection` *operators traditionally, we would get the operator sequence* (o_7, o_8, o_1, o_2) *shown in Figure 3.16(b). The costs of this sequences (using Equation 3.23) are given by* $C(P') = 1{,}000 + 0.05 \cdot 1{,}000 + 1{,}000 \cdot 0.965 + 1{,}000 \cdot 0.965 \cdot 0.4 + 1{,}000 \cdot 0.965 \cdot 0.4 \cdot 0.5 = 2{,}594$ *and thus, the costs are higher than the initial costs. In contrast, if we reorder* `Selection` *operators in a control-flow-aware manner, we get the operator sequences* (o_8, o_1, o_2, o_7) *shown in Figure 3.16(c). The costs of this sequence are computed by* $C(P'') = 1{,}000 + 1{,}000 \cdot 0.4 + 1{,}000 \cdot 0.4 \cdot 0.5 + 200 + 1{,}000 \cdot 0.4 \cdot 0.5 \cdot 0.7 \cdot 0.05 = 1{,}807$. *As a result, control-flow-aware selection reordering reduced the costs from* 2,594 *to* 1,807.

To summarize, the control-flow aware selection reordering exhibits a slightly worse time complexity than the traditional selection ordering. However, the opportunity of a significant plan cost reduction justifies the application of this technique. Finally, note that the same concept of effective operator selectivities $(P(o_i) \cdot f_{o_i} + (1 - P(o_i)))$ is in general, also applicable for all selective operators such as `Groupby`, `Join`, `Setoperation`, and `Projection`. For the sake of a clear presentation, we will not mention this during the discussion of the following data-flow-oriented optimization techniques.

Eager Group-By and Pre-Aggregation

Similar to reordering selective operators, it can be more efficient to apply specific operators as early as possible in order to reduce the cardinalities of intermediate results, where the earliest possible position can be determined using the dependency graph. The core concept is to reduce the cardinality of intermediate results and thus, to improve the execution time of the following operators.

We concentrate only on *WD6: Early Groupby Application*, which was considered for DBMS [CS94] (with complete [YL95] or partial [Lar02] aggregation) and for EII (Enterprise Information Integration) frameworks (adjustable partial window pre-aggregation [IHW04]). For early group-by application, a sequence of `Join` and `Groupby` is rewritten to a construct of `Groupby` and `Join` (invariant group-by also known as eager group-by) or to a construct of `Groupby`, `Join`, and `Groupby` (pre-aggregation). The assumption is that it can be more efficient—with respect to the monitored cardinalities and selectivities—to compute the `Join` on pre-aggregated partitions rather than on the single tuples. The precondition is that the list of grouping attributes G contains the `Join` predicate jp.

First of all, we need some additional notation, where we use the relational algebra for simplicity of presentation. Assume a join of n data sets (with arbitrary multiplicities) and a subsequent group-by, where the join predicate and group-by attributes are equal with $\gamma_{F(X);A_1}(R \bowtie_{R.A_1=S.A_1} S)$. For left-deep join trees, without cross products, and only one join implementation, there are then $4n!$ possible plans because for each join operator, four possibilities exist to apply the `Groupby` operator (for invariant group-by, the final γ in P_c-P_f can be omitted):

$$\begin{array}{llllllll} P_a(opt): & \gamma(R \bowtie S) & P_c: & \gamma((\gamma R) \bowtie S) & P_e: & \gamma(R \bowtie (\gamma S)) & P_g: & (\gamma R) \bowtie (\gamma S) \\ P_b: & \gamma(S \bowtie R) & P_d: & \gamma(S \bowtie (\gamma R)) & P_f: & \gamma((\gamma S) \bowtie R) & P_h: & (\gamma S) \bowtie (\gamma R). \end{array}$$

Without loss of generality, we assume $n = 2$ and concentrate on the four possibilities to arrange group-by and join for a given join order. In addition to the join costs

of $C(\bowtie) = |R| + |R| \cdot |S|$, the group-by costs are given by $C(\gamma) = |R| + |R| \cdot |R|/2$. Furthermore, the output cardinality in case of a single group-by attribute A_i is defined as $1 \leq |\gamma R| \leq |D_{A_i}(R)|$, while for an arbitrary number of group-by attributes it is $1 \leq |\gamma R| \leq \prod_{i=1}^{|A|} |D_{A_i}(R)|$, where D_{A_i} denotes the domain of an attribute A_i. Further, we denote the group-by selectivity with $f_{\gamma R} = |\gamma R|/|R|$. Then, the plan P_a is optimal if the following four optimality conditions hold. First, the commutative join order is expressed with $|R| \leq |S|$. Second, there is one optimality condition for each single join input (in order to determine if pre-aggregation is advantageous):

$$\begin{aligned} C(\gamma(R \bowtie S)) \leq{}& \left(|R| + |R|^2/2\right) + \left(f_{\gamma R} \cdot |R| + f_{\gamma R} \cdot |R| \cdot |S|\right) \\ &+ \left(f_{(\gamma R),S} \cdot f_{\gamma R} \cdot |R| \cdot |S| + \left(f_{(\gamma R),S} \cdot f_{\gamma R} \cdot |R| \cdot |S|\right)^2/2\right) \\ \text{with } C(\gamma(R \bowtie S)) ={}& \left(|R| + |R| \cdot |S|\right) + \left(f_{R,S} \cdot |R| \cdot |S| + \left(f_{R,S} \cdot |R| \cdot |S|\right)^2/2\right), \end{aligned} \tag{3.24}$$

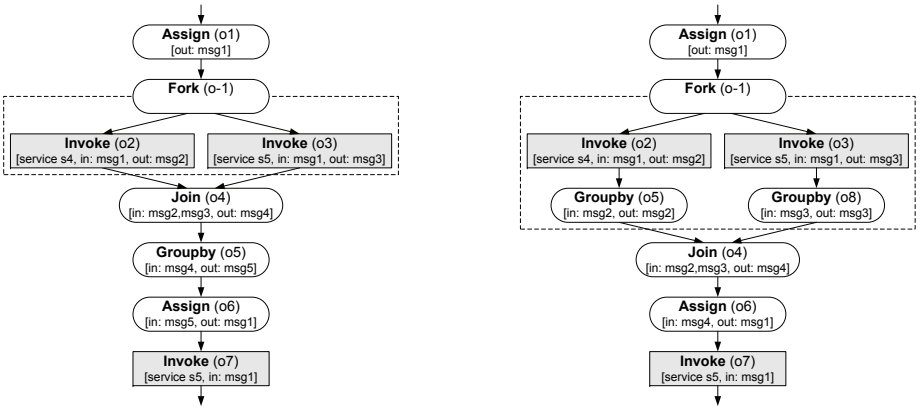
$$C(\gamma(R \bowtie S)) \leq \left(|S| + |S|^2/2\right) + \left(|R| + |R| \cdot f_{\gamma S} \cdot |S|\right) + \left(|R| + \left(f_{R,(\gamma S)} \cdot f_{\gamma S} \cdot |R| \cdot |S|\right)^2/2\right), \tag{3.25}$$

and one condition for all join inputs:

$$C(\gamma(R \bowtie S)) \leq \left(|R| + |R|^2/2\right) + \left(|S| + |S|^2/2\right) + \left(f_{\gamma R} \cdot |R| + f_{\gamma R} \cdot |R| \cdot f_{\gamma S} \cdot |S|\right). \tag{3.26}$$

These conditions are necessary due to the characteristic of missing knowledge about data properties. For example, we do not know the multiplicities of join inputs that can be exploited for defining simpler optimality conditions in advance. The algorithm for realizing this technique is invoked for each `Groupby` operator. Then, we check by the use of the dependency graph if this operator can be reordered with predecessor `Join` operators, where for each join there are four optimality conditions. As a result, this algorithm exhibits a linear time complexity of $O(m)$. We use an example to illustrate this concept.

Example 3.14 (Eager Group-By)**.** *Recall our running example plan P_2 as shown in Figure 3.17(a). Assume arbitrary join multiplicities and monitored statistics. Based on the given optimality condition, the plan has been rewritten to P_2' as shown in Figure 3.17(b). Essentially, we observed that the full eager-group-by before the join causes lower costs than the join-group-by combination. Note that the* `Fork` *operator is taken into account by*



(a) Plan P_2 (b) Plan P_2'

Figure 3.17: Example Eager Group-By Application

determining only the most time-consuming subflow rather than all subflows. A detailed cost estimation example of a rewritten subgraph (applying this technique) was given with Example 3.2 (Cost Estimation) in Subsection 3.2.2

Finally, this technique should be applied after the join enumeration WD10 because it requires the optimal join order as the basis for its rewriting algorithm but it should be applied before WD9 because the join type selection might change according to the estimated cardinalities. Interdependencies to join enumeration are deferred to the next invocation of the optimizer.

Set Operations with Distinctness

For the `Setoperation` operator, we distinguish different types, among others the `UNION DISTINCT` and the `UNION ALL`. While the `UNION ALL` can be computed with low costs of $|ds_{in1}| + |ds_{in2}|$, the costs of `UNION DISTINCT` are computed by

$$|ds_{in1}| + |ds_{in2}| \cdot \frac{|ds_{out}|}{2}. \tag{3.27}$$

We transfer the first data set completely into the result ($|ds_{in1}|$) and then, for each tuple of the second data set ds_{in2}, we need to check whether or not this tuple is already in the result. In the average case, this causes costs of $|ds_{out}|/2$ for each tuple. As a result, `UNION DISTINCT` has a quadratic time complexity of $O(N^2)$.

The core idea of *WD11: Setoperation-Type Selection* in combination with the *WD8: Orderby Insertion / Removal* is to sort both input data sets by their distinct key in order to enable the application of an efficient merge algorithm for ensuring distinctness. Hence, only costs of $|ds_{in1}| + |ds_{in2}|$ (similar to a `UNION ALL`) would be necessary to compute the `UNION DISTINCT`. Including the costs for sorting, the result is a time complexity of $O(N \log N)$. Despite the internal order-preserving XML representation, sorting of message content is applicable in dependence on the source adapter types of these messages.

In the following, we consider the required optimality conditions. There are three alternative subplans for a union distinct $R \cup S$. First, there is the normal union distinct operator with costs that are given by $C(R \cup S) = |R| + |S| \cdot |R \cup S|/2$ (two plans due to asymmetric costs), where $|R| \leq |R \cup S| \leq |R| + |S|$ holds. Second, we can sort both input data sets and apply a merge algorithm (third plan), where the costs are computed by

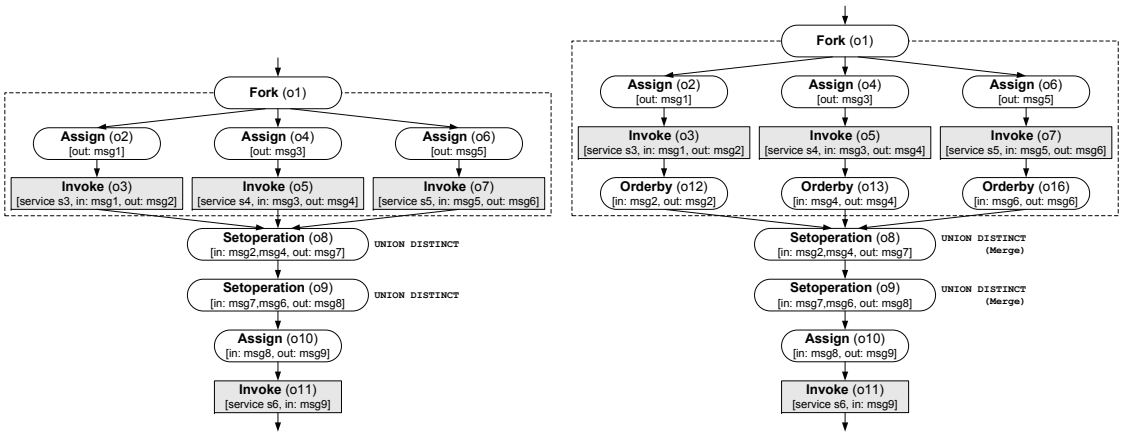
$$C\left(\texttt{sort}(R) \cup_M \texttt{sort}(S)\right) = |R| + |S| + |R| \cdot \log_2|R| + |S| \cdot \log_2|S|. \tag{3.28}$$

In conclusion, for arbitrary cardinalities, the optimality conditions are $|R| \geq |S|$ and

$$\begin{aligned} |R| + |S| \cdot \frac{|R \cup S|}{2} \leq\ & |R| + |S| + |R| \cdot \log_2|R| + |S| \cdot \log_2|S| \\ |R \cup S| \leq\ & 2 + \frac{2 \cdot |R| \cdot \log_2|R|}{|S|} + 2 \cdot \log_2|S|. \end{aligned} \tag{3.29}$$

We see that this decision depends on the union output cardinality and both input cardinalities. If one input is known to be sorted, the corresponding `Orderby` operator is omitted and the optimality conditions are modified accordingly.

Example 3.15 (Setoperation-Type Selection)**.** *Assume our example plan P_6 (see Figure 3.18(a)) that includes two `Setoperation` operators of type `UNION DISTINCT`. Using*



(a) Plan P_6 (b) Plan P_6'

Figure 3.18: Example Setoperation Type Selection

the techniques orderby insertion and setoperation type selection, we created the rewritten plan P_6' shown in Figure 3.18(b). Here, we use the efficient merge algorithm for both `Setoperation` *operators and hence, require to sort all three input data sets. Sorting the result of the first* `Setoperation` *operator is not required because the output of the merge algorithm is already ordered. Consider two cases with different statistics for input and output cardinalities of the* `Setoperation` *o_8. Figure 3.19 (left) shows the abstract costs of the two possible subplans—$P_6 : (o_8)$ versus $P_6' : (o_{12}', o_{13}', o_8')$—in both cases.*

	Statistics	$C(o_8)$	$C(o_{12}', o_{13}', o_8')$
case 1	$\lvert ds_{in1}(o_8)\rvert = 1{,}000$ $\lvert ds_{in2}(o_8)\rvert = 1{,}000,$ $\lvert ds_{out}(o_8)\rvert = 1{,}000$	501,000	<u>21,932</u>
case 2	$\lvert ds_{in1}(o_8)\rvert = 1{,}000,$ $\lvert ds_{in2}(o_8)\rvert = 10,$ $\lvert ds_{out}(o_8)\rvert = 1{,}000$	<u>6,000</u>	11,009

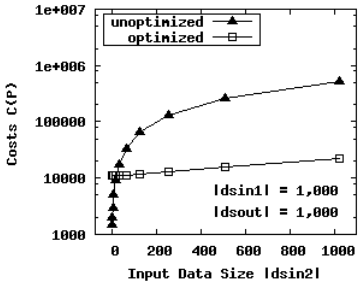


Figure 3.19: Example Setoperation Cost Comparison

We observe that the optimality of these subplans depends on current workload characteristics, while the subplan (o_{12}', o_{13}', o_8') is more robust[6] over arbitrary statistic ranges than the subplan (o_8) as shown in Figure 3.19 (right).

Finally, this optimization technique is also applicable for other operators such as `Projection` with duplicate elimination or for forcing a merge-based join algorithm (WD9). Thus, in general, the technique WD8 (orderby insertion) should be applied before selecting different physical types of an operator.

To summarize, we presented selected control-flow- and data-flow-oriented optimization

[6]Robustness is an alternative optimization objective, which is beyond the scope of this thesis. However, in contrast to existing work [ABD+10], we would identify these robust (insensitive to input statistics) plans by simply choosing one of the plans with lowest asymptotic time complexity with regard to their overall abstract cost functions of all plan operators.

techniques. Traditional techniques can be adapted also for the optimization of integration flows. In addition to this, there are hybrid techniques, where the control flow must be taken into account for data-flow-oriented techniques as well. Finally, there are also techniques that are tailor-made for integration flows. In conclusion, we observe the presence of many optimization opportunities, where ensuring semantic correctness is a major challenge with regard to concrete optimization techniques. Therefore, our transformation-based optimization algorithm iteratively applies optimization techniques. Many of these techniques are independent but for dependable techniques, we need to prevent local suboptima. Note that applying techniques independently (1) reduces the optimization overhead due to a reduced search space, and (2) less development effort for new techniques. The presented general optimization framework can then be extended with arbitrary additional optimization techniques.

3.5 Experimental Evaluation

In this section, we present results of our exhaustive experimental evaluation with regard to the three evaluation aspects: (1) optimization benefits and scalability, (2) optimization overheads, as well as (3) workload adaptation. In general, the evaluation shows that:

- Significant performance improvements can be achieved by periodical re-optimization in the sense of minimizing the average execution time of a plan. According to Little's Law [Lit61], this has also direct influence on the message throughput improvement. Scalability experiments showed that the benefit increases with increasing amount of input data as well as with increasing plan complexity.
- The overhead for statistic maintenance and periodical re-optimization is moderate. Thus, this overhead is typically subsumed by the achieved execution time reduction. Even in the worst-case, where the initial plan constantly exhibits the optimality property, this additional runtime overhead is moderate.
- The right choice of parameterization (workload aggregation, optimization interval, sliding time window size) in combination with correlation awareness can ensure an accurate but still robust adaptation to changing workload characteristics. In detail, even after specific workload changes, the self-adjusting cost model consistently converges to the real costs.

In conclusion of these major experimental findings, the periodical re-optimization can be applied by default. The available parameters of the optimization algorithm can additionally be used to fine-tune the adaptation sensibility (and thus, influence the execution time reduction) and the optimization overhead.

The detailed description of our experimental results is structured as follows. First, we explain the end-to-end comparison of no-optimization versus periodical re-optimization for all plans of our use cases. This already includes the optimization overhead. Second, we analyze this optimization overhead in more detail with regard to statistics monitoring and re-optimization. Third, we evaluate the cost model according to the adaptation to changing workload characteristics as well as how to set the existing parameters.

Experimental Setting

The experimental setup comprises an IBM blade with two processors (each a Dual Core AMD Opteron Processor 270 at 2 GHz) and 9 GB RAM, where we used Linux openSUSE 9.1 (32 bit) as the operating system. Our WFPE (workflow process engine) realizes the extended reference system architecture (described in this chapter) including our optimization component. The WFPE is implemented using Java 1.6 as the programming language and includes approximately 36,000 lines of code. It currently includes several adapters (inbound and outbound) for the interaction with files, databases, and Web services. However, the external systems, used by our example integration flows, have been simulated with file adapters in order to minimize the influence of used systems on the measured experimental results (reproducibility). In order to use arbitrary workload scenarios with different cardinalities and selectivities (workability), we executed all experiments on synthetic XML data generated using our DIPBench toolsuite [BHLW08c].

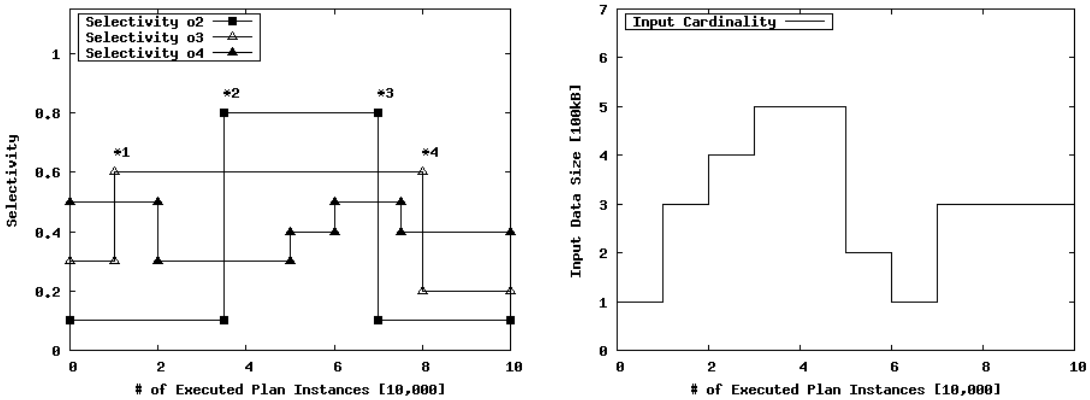
There are several parameters that influence plan execution. Essentially, we analyzed two groups of parameters. First, there is the group of optimization parameters, where we used different sliding window sizes Δw (default: 5 min), different optimization intervals Δt (default: 5 min), and different workload aggregation methods *Agg* (default: `EMA`). Second, there is the group of workload characteristics. Here, we used different numbers of plan instances n (executed instances), different plans with certain numbers of operators m, and different input data sizes d (default: $d = 1$, which stands for 100 kB input messages) and different selectivities. With regard to applied optimization techniques, we used all cost-based optimization techniques, except *message indexing* and *heterogeneous load balancing* (both not discussed in this thesis) as well as *vectorization* (see Chapter 4) and *multi-flow optimization* (see Chapter 5). Furthermore, we disabled all rule-based optimization techniques in order to focus on the benefit achieved by cost-based optimization because these techniques either did not apply (e.g., algebraic simplifications) for the used plans or they achieved a constant absolute improvement (e.g., static node compilation) for both the unoptimized and the cost-based optimized execution.

End-to-End Comparison and Optimization Benefits

In a first series of experiments, we compared the end-to-end performance of no-optimization versus the periodical re-optimization. These experiments already include all optimization overheads (such as statistics maintenance and periodical re-optimization). As a result, these experiments show the overall benefit achieved by periodical re-optimization.

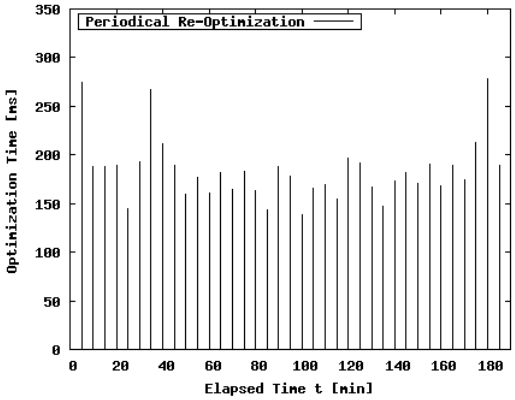
First, we compared the periodical re-optimization with no-optimization. The periodical re-optimization was realized as asynchronous inter-instance optimization approach. We executed 100,000 instances of our example plan P_5 for the non-optimized plan as well as for the optimized plan and measured re-optimization and plan execution time. For periodical re-optimization, the plan execution time already includes the synchronous statistic maintenance. During execution, we varied the input cardinality (see Figure 3.20(b)) and selectivities of the three selection operators (see Figure 3.20(a)). Here, the input data was generated without correlations between different attributes. With regard to re-optimization, there are four points ($*1$, $*2$, $*3$, and $*4$) where a workload change (shown as intersection points between selectivities) reasons the change of the optimal plan.

For periodical re-optimization, we used an optimization interval of $\Delta t = 5$ min, a sliding window size of $\Delta w = 5$ min and `EMA` as the workload aggregation method. Figure 3.20(c)

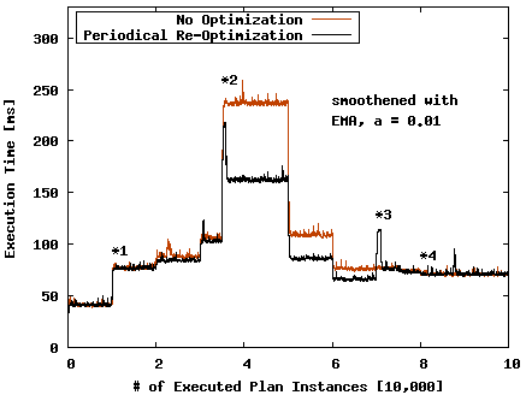


(a) Selectivity Variations

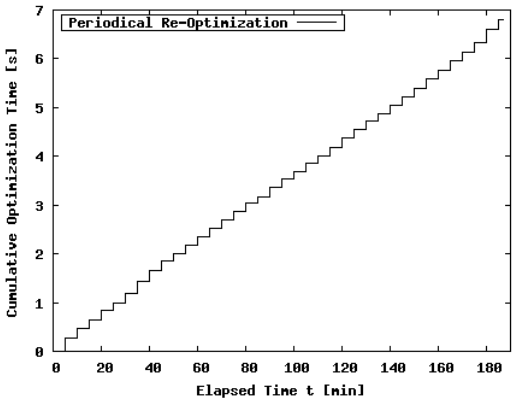
(b) Input Data Size Variations



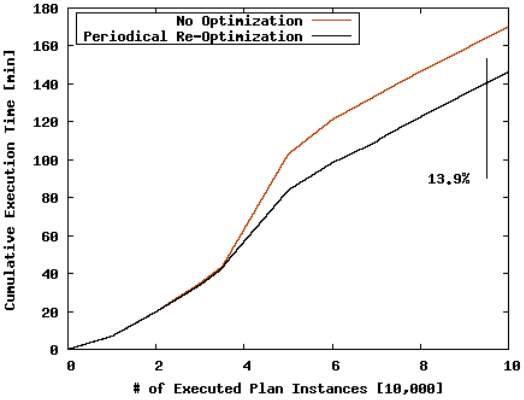
(c) Optimization Time



(d) Execution Time



(e) Cumulative Optimization Time



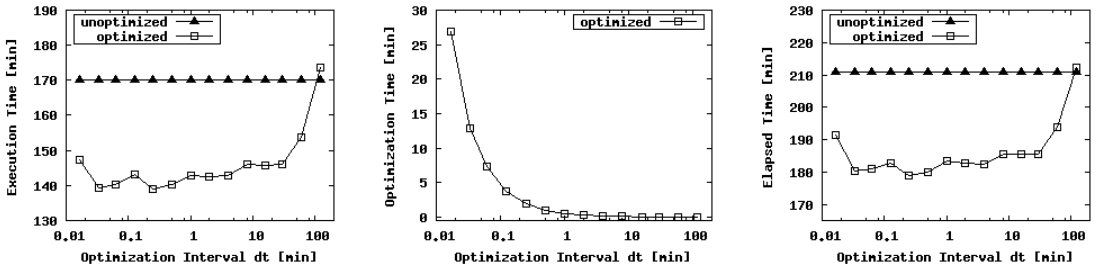
(f) Cumulative Execution Time

Figure 3.20: Comparison Scenario Periodical Re-Optimization

shows the single re-optimization times and Figure 3.20(e) illustrates the cumulative time required for re-optimization during this scenario. Note that we used the elapsed scenario time as the x-axis in order to illustrate the characteristics of periodical re-optimization. We see that periodical re-optimization requires many unnecessary optimization steps. However, each single optimization step requires only a fairly low optimization time. Note that

for this example plan, the re-optimization time is dominated by the physical plan compilation and the waiting time for the next possible exchange of plans. The generation of physical plans regardless of whether or not a new plan was found has been reasoned by optimization techniques (such as switch path re-ordering), which directly reorder operators and thus, they always signal that recompilation is required. Clearly, for periodical re-optimization, we could use a larger Δt and thus, would reduce the cumulative total optimization time. However, in that case, we would use suboptimal plans for a longer time and hence, we would miss more optimization opportunities.

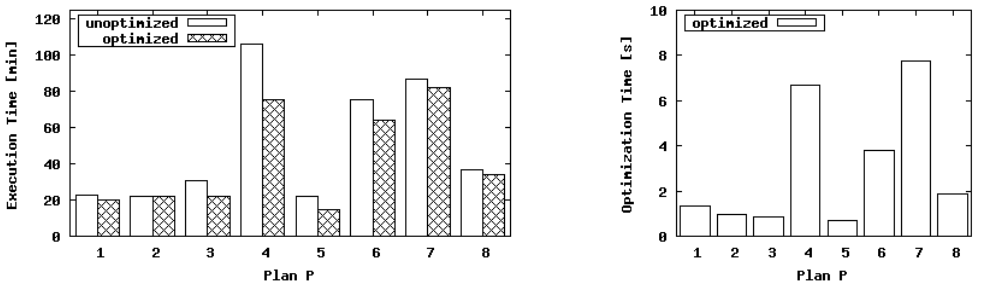
Furthermore, Figures 3.20(d) and 3.20(f) show the execution time using periodical re-optimization compared to the non-optimized execution. The different execution times are caused by the changing workload characteristics in the sense of different input cardinalities as well as selectivities of the different operators. When using periodical re-optimization, the often re-occurring small peaks are caused by the numerous asynchronous re-optimization steps. Further, a major characteristic of periodical re-optimization is that after a certain workload shift, there is an adaptation delay until periodical re-optimization is triggered. During this time, the execution time of the current plan is much longer than the execution time of the optimal plan. In order to reduce these adaptation delays, a small Δt is required. However, this would significantly increase the total re-optimization time. To summarize, over time, significant execution time reductions are yielded by periodical re-optimization due to the adaptation to changing workload characteristics.



(a) Cumulative Execution Time (b) Cumulative Opt. Time (c) Scenario Elapsed Time

Figure 3.21: Influence of Optimization Interval Δt

Second, we used the introduced comparison scenario in order to investigate the influence of the parameter Δt in more detail. We re-executed this with different optimization periods $\Delta t \in \{1\,\text{s}, 2\,\text{s}, 3.75\,\text{s}, 7.5\,\text{s}, 15\,\text{s}, 30\,\text{s}, 60\,\text{s}, 120\,\text{s}, 240\,\text{s}, 480\,\text{s}, 960\,\text{s}\, 1800\,\text{s}, 3600\,\text{s}, 7200\,\text{s}\}$. Figure 3.21(a) illustrates the resulting cumulative execution time for optimized execution compared to the unoptimized execution. We observe that the higher the optimization period, the higher the cumulative execution time because we miss optimization opportunities after a workload shift due to deferred adaptation. However, it is important to note that if the optimization period is too small, the execution time gets worse again. This is reasoned by small benefits reached by immediate asynchronous re-optimization in combination with increasing optimization costs as shown in Figure 3.21(b). While, so far we used only the cumulative execution time (sum of plan execution times) as indicator, now, we also discuss the elapsed time (time required for executing the sequence of plan instances, which includes the workflow engine overhead and time during exchange of plans). Figure 3.21(c) shows that for small optimization intervals Δt—where we often exchange plans—the elapsed time increase faster than the cumulative execution time. The reason is



(a) Cumulative Execution Time (b) Cumulative Optimization Time

Figure 3.22: Use Case Comparison of Periodical Re-Optimization

asynchronous optimization but synchronous exchange of plans, where execution is blocked. As a result both cumulative execution time and elapsed time might have different optimal Δt configurations. Thus, this parameter is a possibility to fine-tune the optimizer.

Third, with regard to *workability*, we observed fairly similar results for our other example plans and statistic variations. Here, we compared the periodical re-optimization with no-optimization once again. In detail, we executed 20,000 plan instances for each example plan (P_1,P_2,P_3,P_4,P_5,P_6,P_7,P_8) and for each execution model. There, we fixed the cardinality of input data sets to $d = 1$ (100 kB messages) and used a well-balanced workload configuration (without correlations and without workload changes). Furthermore, we fixed an optimization interval of $\Delta t = 5$ min, a sliding window size of $\Delta w = 5$ min and `EMA` as the workload aggregation method.

To summarize, we consistently observe execution time reductions (see Figure 3.22(a)). In the following, we describe in detail how these benefits have been achieved:

- P_1: This plan was affected by three different optimization techniques. First, the technique WD1 reordered the two paths of `Switch` operator o_2. Furthermore, the operator sequence (o_7,o_8,o_9) has been rewritten to parallel subflows (o_7) and (o_8,o_9). Finally, the technique WC1 rescheduled the start of both subflows in order to start the most time-consuming subflow (o_8,o_9) first.
- P_2: No optimization technique affected this plan.
- P_3: Similar to plan P_1, the techniques WC2 and WC1 have been applied on the operator sequence (o_2,o_3) in order to rewrite this sequence to parallel subflows and to reschedule the start of these subflows. In addition, the technique WD6 has been applied in order to pushdown the invariant group-by and thus, exchanged the temporal order and data dependencies of operators o_4 and o_5.
- P_4: For this rather complex plan, only the optimization technique WD9 was applied. In detail the `Join` operator o_9 was rewritten from a nested loop join to a subplan of two (concurrent) `Orderby` operators and one merge join.
- P_5: Similar to our first end-to-end comparison scenario, the technique WD4 was applied for the plan P_5 with the aim of reordering the sequence of `Selection` operators (o_2,o_3,o_4).
- P_6: This plan was affected by a set of techniques. First, the initially given `Fork` operator was rescheduled by WC1. Furthermore, the techniques WD11 and WD8 rewrote the two subsequent `Setoperation` (`UNION DISTINCT`) operators to a sub-

plan with `Orderby` and `Setoperation` (`UNION DISTINCT MERGE`) operators. The sorted output order between the two `Setoperation` operators was exploited by the optimizer in order to reduce the number of required `Orderby` operators. Finally, the `Setoperation` and `Orderby` operators were parallelized by WC2.

- P_7: Similar to P_6, this plan was also affected by WC1 in the sense of rescheduled subflows of the existing `Fork` operator. Furthermore, the techniques WD10 and WD8 changed the `Join` operators o_{14}, o_{15} and o_{16} from nested loop joins to subplans of `Orderby` and merge join operators. Finally, note that join enumeration did not resulted in a new join order.
- P_8: This plan was mainly affected by control-flow oriented optimization techniques. In detail, the operator sequence (o_3-o_9) was rewritten into two parallel subflows of a `Fork` operator. The last operator o_{10} was not included because both o_9 and o_{10} are two writing interactions to the same external system. Finally, the technique WC1 was applied once again for rescheduling the created subflows.

In addition to these consistent optimization benefits, Figure 3.22(b) shows the required cumulative optimization time. The significant differences between the optimization times of different plans are caused by two facts. First, the different total execution time influences the number of periodical re-optimization steps required in this scenario because these optimization steps are triggered periodically. Second, different techniques (with different time complexity) are applied according to the specific operator types used in the concrete plan. For example, plans P_4 and P_7 are dominated by the costs for join enumeration, where we did not found different join orders due to ensuring semantic correctness (P_4) and the chain query type (P_7).

Putting it all together, we can conclude that execution time reductions are possible, while only a fairly low overhead is required by periodical re-optimization.

Scalability

In addition to the presented comparison of optimized and unoptimized execution, scalability is one of the most important aspects. Hence, we conducted a series of experiments that examines the scalability with regard to increasing number of operators as well as with regard to increasing input data size.

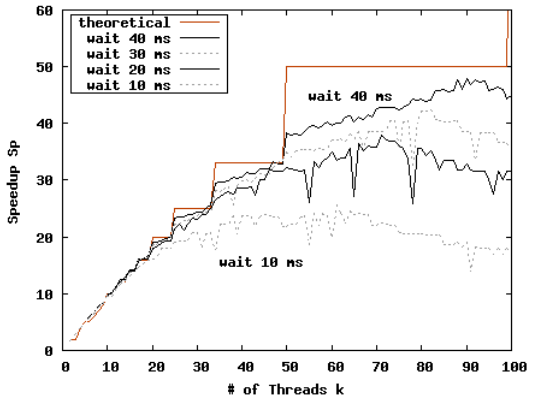


Figure 3.23: Speedup of Rewriting Sequences to Parallel Flows

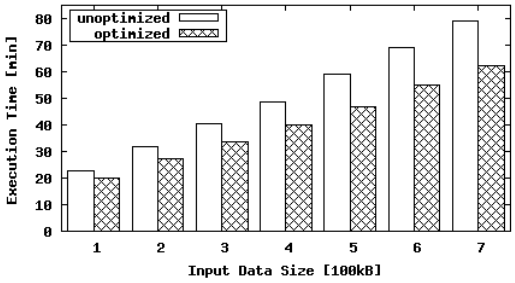
First, in order to investigate the benefits of rewriting patterns to parallel subflows in more detail, we executed a speedup experiment. Figure 3.23 shows the results of this experiment on the rewriting of sequences to parallel flows (WC2). We used a plan that contains a sequence of $m = 100$ independent `Delay` operators, and we explicitly varied the number of threads k (concurrent subflows) used for parallelism (`Fork` operator) in order to evaluate the speedup. Due to the distribution of m operators to k forklanes, a theoretical speedup of $m/\lceil m/k \rceil$ is possible. Then, we varied the waiting time of each single `Delay` operator from 10 ms to 40 ms in order to simulate different network delays and waiting times for external systems, respectively. This experiment was repeated ten times. As a result, an increasing maximum speedup (at an increasing number of threads) was measured with increasing waiting time (note that the fall-offs were caused by the Java garbage collector). This strong dependence of multi-tasking benefit on the waiting time of involved operators justifies the decision to use the waiting time as main cost indicator when rewriting sequences and iterations to parallel flows.

Second, we used our running example plans in order to investigate the scalability of optimization benefits with increasing input data size. Based on the experimental results shown in Figure 3.22, we re-used the workload configuration and all plans except plan P_2 because for this plan no optimization technique could be applied. In detail, we executed 20,000 plan instances for each running example plan and compared the periodical re-optimization with no-optimization varying the input data size $d \in \{1, 2, 3, 4, 5, 6, 7\}$ (in 100 kB messages), which resulted in a total processed input data size of up to 13.35 GB (for $d = 7$). Note that we varied this input data size only for the initially received message of a plan, while for plans P_3, P_6 and P_8, we changed the cardinality of externally loaded data sets because these plans are time-based initiated. Further, we fixed an optimization interval of $\Delta t = 5$ min, a sliding window size of $\Delta w = 5$ min and `EMA` as the workload aggregation method. For all investigated plans, we observe that the relative benefit of optimization increases with increasing data size as shown in Figure 3.24. Essentially, the same optimization techniques were applied and thus, the optimization time is unaffected by the used input data size. The highest benefits were reached by the data-flow oriented optimization techniques. For example, the unoptimized versions of plans P_4, P_6, P_7 show a super-linearly increasing execution time with increasing data size, while the optimized versions shows almost a linear increasing execution time. Thus, the optimized versions exhibit a better asymptotic behavior, which is for example, caused by rewriting nested loop `Joins` to combinations of `Orderby` and merge `Join` subplans. With this in mind, it is clear that arbitrarily high optimization benefits can be reached with increasing input data size (input cardinalities).

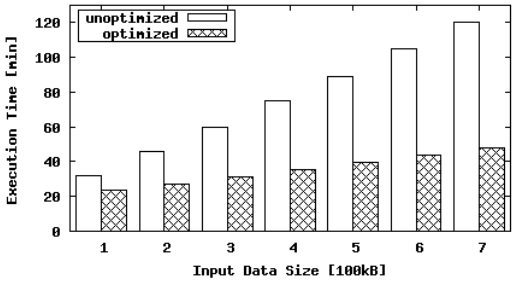
Optimization Overhead

In addition to the evaluation of performance benefits and the scalability with increasing input cardinalities and increasing number of operators, we evaluated the optimization overhead in more detail. Therefore, we conducted a series of experiments, where (1) we compared the optimization overhead of our exhaustive optimization approach versus the heuristic optimization approach and (2) we analyzed the overhead of statistics monitoring and aggregation in detail.

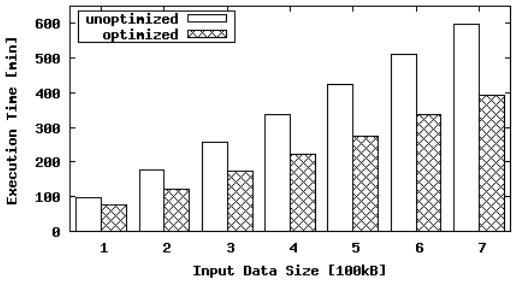
First, we evaluated the influence of using the exhaustive optimization algorithm A-PMO or the heuristic A-HPMO. We already discussed when it is applicable to use the second heuristic A-CPO and therefore did not include it into the evaluation. Essentially, the ex-



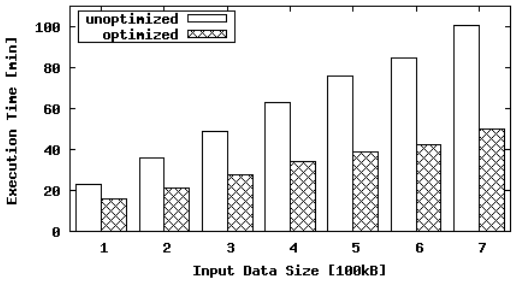
(a) Plan P_1



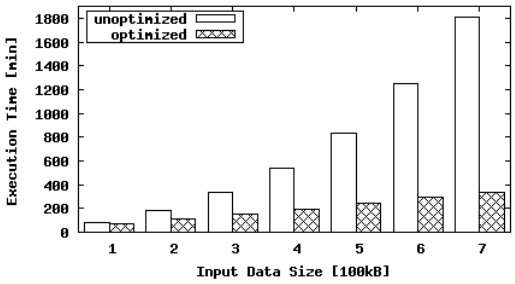
(b) Plan P_3



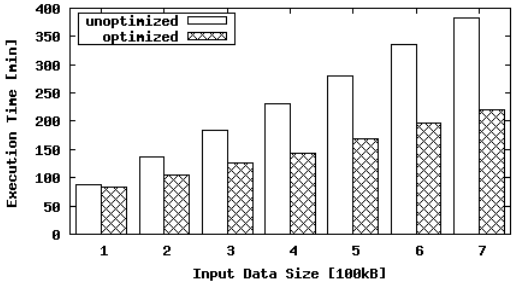
(c) Plan P_4



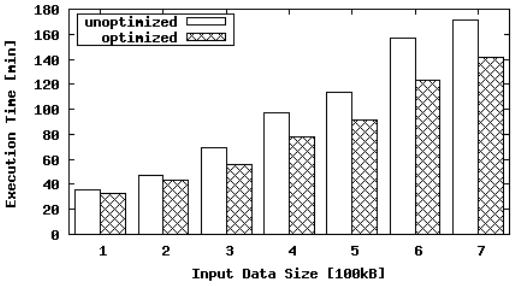
(d) Plan P_5



(e) Plan P_6



(f) Plan P_7



(g) Plan P_8

Figure 3.24: Use Case Scalability Comparison of Periodical Re-Optimization

haustive and heuristic optimization approach differ only in the join enumeration algorithm used. Thus, we evaluated the execution time of the technique join enumeration for different plans that included different numbers of `Join` operators ($m \in \{2, 4, 6, 8, 10, 12, 14\}$) as a clique query (where all quantifiers are directly connected). Here, we compared the optimization time of (1) the exhaustive join enumeration using the standard DPSize algo-

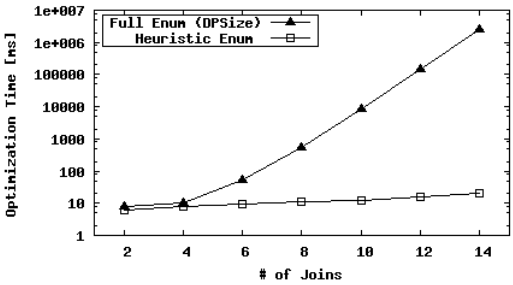


Figure 3.25: Optimization Overhead of Join Enumeration

rithm [Moe09] and (2) our join reordering heuristic with quadratic time complexity that we have described in Subsection 3.3.2. Before optimization, we randomly generated statistics for input cardinalities and join selectivities. The experiment was repeated ten times. Figure 3.25 illustrates the results of this experiment using a log-scaled y-axis. The optimization time of the full join enumeration increases exponentially, while for the heuristic re-optimization, the optimization time increases slightly super-linear. However, we observe acceptable absolute optimization time of exhaustive join enumeration until eight `Join` operators, where the clique query is the worst-case for the DPSize algorithm. This justifies our algorithm selection rule of using the full optimization algorithm until eight joins and to use the heuristic for larger numbers of joins. As a result, we can guarantee that (1) the time required by our optimization algorithm will not increase exponentially with the complexity of plans and (2) the algorithm will find the optimal plan in the presence of small numbers of `Join` operators.

Second, we analyzed the overhead of statistics monitoring and statistics aggregation. If statistics monitoring is enabled, each operator propagates at least three ($|ds_{in}|$, $|ds_{out}|$, and $W(o_i)$) statistics to the Estimator. However, there are operators that propagate more statistics such as `Switch` path frequencies, number of iterations and the cardinality of multiple input and output data sets. Thus, the efficiency of statistics monitoring and workload aggregation is important in order to achieve moderate re-optimization overheads.

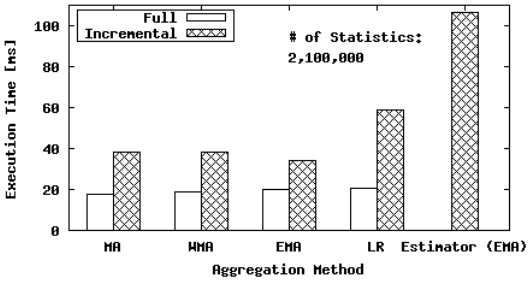


Figure 3.26: Cumulative Statistic Maintenance Overhead

We used the statistic trace from our first comparison scenario (see Figure 3.20), where we executed $n = 100{,}000$ plan instances of P_5. We used three statistics (execution time, input and output cardinalities) from seven of nine operators of this plan that results in a test set of 2,100,000 statistic tuples. All sub experiments were repeated 1,000 times. Figure 3.26 illustrates the results for different aggregation strategies. Full aggregation refers to a single estimation using all statistics, which is applicable if the optimization

period is longer than the sliding time window size ($\Delta t \geq \Delta w$). In detail, we aggregated 700,000 statistics (execution times $W(o_i)$ only) and we observed that all statistics were aggregated in less than 20 ms. The single aggregation methods differ only slightly in their execution time, where `MA` is the fastest method but only minor differences are observable. If $\Delta t < \Delta w$ or no Δw is used, incremental statistics maintenance is required. Thus, we repeated the experiment with our incremental aggregation methods. When comparing full and incremental maintenance, we see that the incremental methods are a factor of 1.5 to 3 slower than the full methods because they require additional computation efforts for producing valid intermediate results and for many method invocations. `EMA` is the fastest incremental method based on its incremental nature. Our Estimator comprises all of these aggregation methods and some additional infrastructural functionalities, where we use the incremental `EMA` as default aggregation method. The maintenance of all three statistics ($|ds_{in}|$,$|ds_{out}|$, and $W(o_i)$) for all plan instances of the test set (2,100,000 statistic tuples) using our Estimator is illustrated as `Estimator (EMA)`. In conclusion, the overhead for statistics maintenance during the full comparison scenario was 106 ms. This is negligible compared to the cumulative execution time of 140 min in the optimized case.

Workload Adaptation

Due to changing workload characteristics, the sensibility of workload adaptation has high importance. According to Subsection 3.3.3, there are three possibilities to influence the sensibility of workload adaptation: (1) the workload sliding time window size Δw, (2) the optimization period Δt, and (3) the workload aggregation method Agg. We evaluated their influence in the following series of experiments.

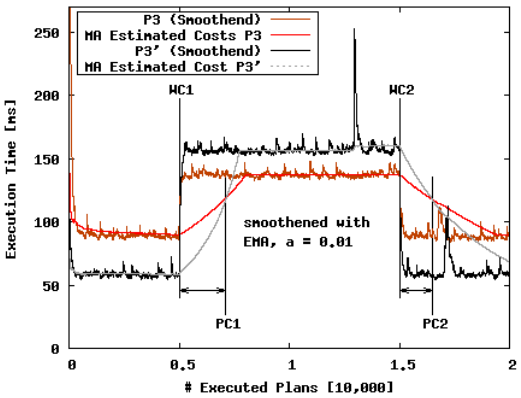
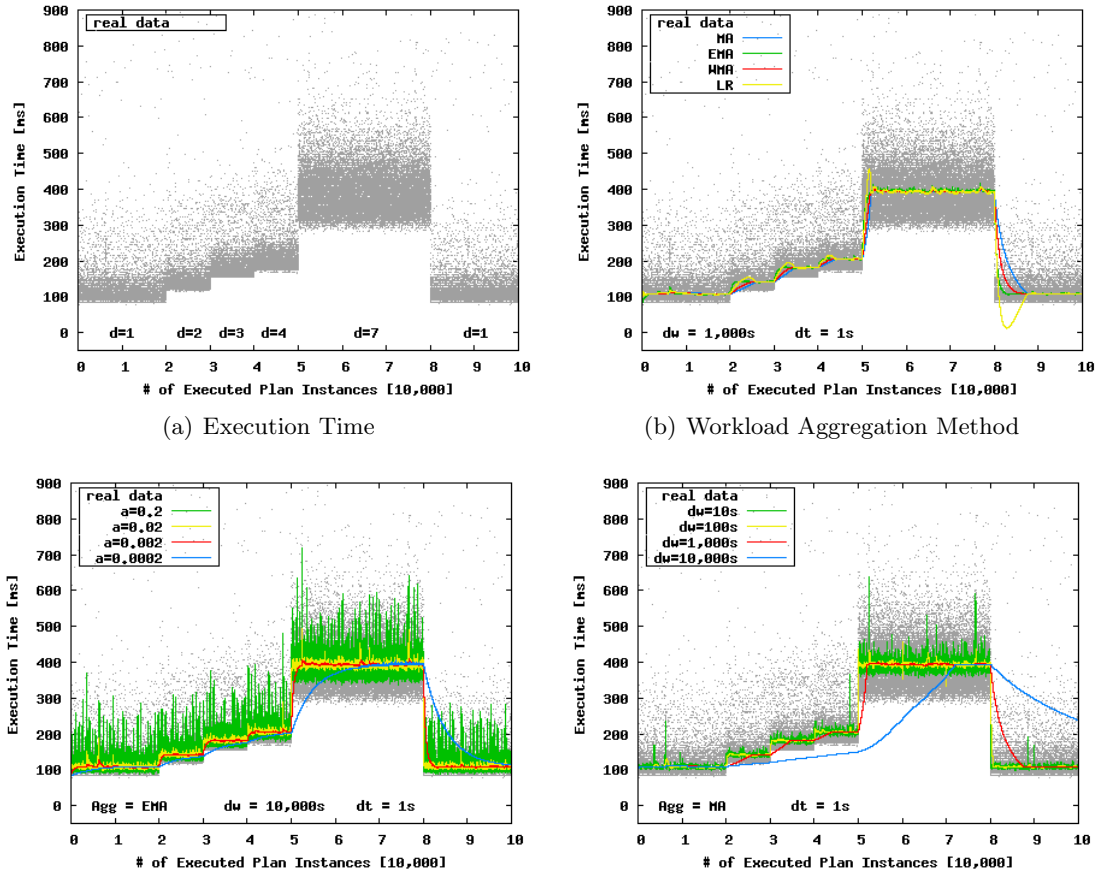


Figure 3.27: Workload Adaptation Delays

Figure 3.27 shows the results of an experiment, where we executed $n = 20{,}000$ instances of plan P_3 and a modified plan P_3' (with eager group-by) with disabled periodical re-optimization. After $n = 5{,}000$ and $n = 15{,}000$ instances, we changed the cardinality of one of two input data sets (workload changes WC1 and WC2). While in the first part, the eager group-by was most efficient, the simple join and group-by performed better after WC1. We fixed a sliding window size of $\Delta w = 5{,}000$ s and `MA` as the workload aggregation method. It took 2,100 plan instances to adapt to the workload shift and the plan changed (PC1 at break even point between estimated plan costs). This adaptation delay depends on the used sliding time window size Δw and the aggregation method.

We conducted further experiments where we executed $n = 100{,}000$ instances of the plan P_8 and varied the scale factor data size d (in 100 kB) in order to simulate changing workload characteristics. Figure 3.28(a) illustrates the monitored plan execution times with annotated input data sizes d. Based on these statistics, we evaluated the influences of workload aggregation methods, their parameters, and of the sliding time window size in detail.



(a) Execution Time

(b) Workload Aggregation Method

(c) `EMA` Smoothing Parameter α

(d) Sliding Time Window Size Δw

Figure 3.28: Influence of Parameters on the Sensibility of Workload Adaptation

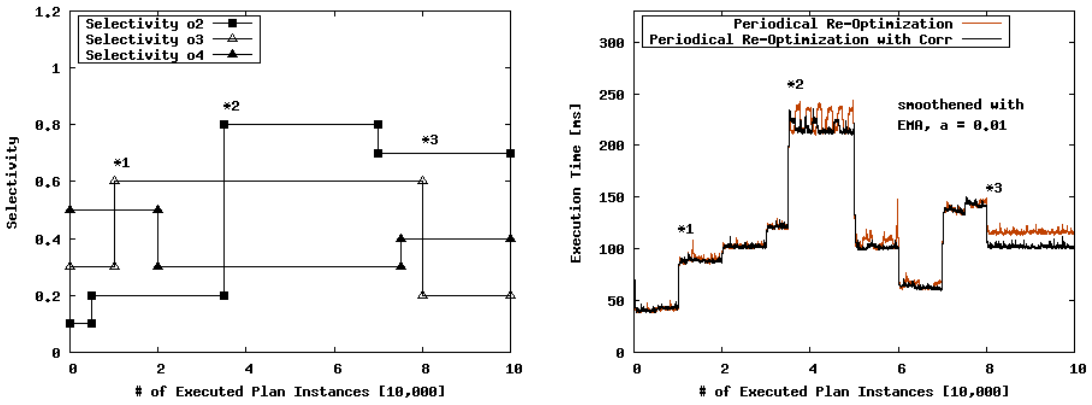
First, Figure 3.28(b) shows the influence of the workload aggregation method, where we fixed $\Delta w = 1{,}000\,\text{s}$ and illustrate the estimated costs continuously ($\Delta t = 1\,\text{s}$). The real execution times contain several outliers and a major skew. Obviously, `MA` causes the slowest adaptation, while `WMA` and `EMA` cause faster adaptation. This is reasoned by the fact that `MA` (constant weights) computes a linear average, while `WMA` (linear weights) and `EMA` (exponential weights) compute weighted averages over the sliding window, where the latest items have higher influence. Further, `LR` causes the fastest adaptation due to extrapolation. However, it is important to note that `LR` tends to strongly over- and underestimate on abrupt workload shifts such as at the 80,000$^{\text{th}}$ plan instance. The results support our decision to use the exponential moving average (`EMA`) as default aggregation method due to fast but robust workload adaptation. There, our default setting of the `EMA` parameter $\alpha = 0.002$ was set empirically based on the observed variance of plan execution times

for typical workloads. Second, we evaluate the parameters of these workload aggregation methods. Figure 3.28(c) illustrates the influence of the parameter `EMA` smoothing constant α. We used a sliding time window size of $\Delta w = 10{,}000\,\text{s}$ and illustrated the estimated costs continuously ($\Delta t = 1\,\text{s}$). Clearly, an decreasing parameter α causes slower adaptation and therefore more robust estimation. However, for typical parameter settings of 0.05 to 0.001 very fast but still robust adaptation can be achieved. Note that for $\alpha \in \{0.2, 0.02, 0.002\}$ we obtained similar results for the sliding window size of $\Delta w = 1{,}000\,\text{s}$ from the previous experiment. However, for $\alpha = 0.0002$ (with $\Delta w = 1{,}000\,\text{s}$) the estimates varied significantly which was caused by too few statistics in the time window in combination with a low smoothing factor such that the estimated values were significantly determined by the initial value (first statistic in the window) because the adaptation took too long. In order to analyze the influence of the sliding window size Δw in general, we conducted an additional experiment. Figure 3.28(d) illustrates the influence of the sliding time window size, where we fixed $Agg = \texttt{MA}$ and varied Δw from 10 s to 10,000 s. Clearly, the adaptation slows down with an increasing Δw. However, both extremes can lead to wrong (large error) estimations. The choice of the window size should be made based on the specific plan because, for example, a long-running plan or a infrequently used plan need a longer time window than plans with many instances per time period. The `EMA` method, typically, does not need sliding window semantics due to the time-decaying character where older items can be neglected. However, if a sliding window is used, the sliding window size Δw should be set according to the plan and the used smoothing constant α such that enough statistics are available as already discussed. Furthermore, the optimization interval influences the re-estimation granularity. With $\Delta t = 1\,\text{s}$, we get a continuous cost function, while an increasing Δt causes a slower adaptation because this influences the maximal delay of Δt until re-estimation. Obviously, parameter estimators, which minimize the error between forecast values and real values could be used to determine optimal parameter values for Δt and Δw. However, when and how to adjust these parameters is a trade-off between additional statistic maintenance overhead and cost estimation accuracy that is beyond the scope of this thesis.

With regard to precise statistic estimation, handling of correlated data and conditional probabilities are important. Therefore, we conducted an experiment in order to evaluate our lightweight correlation table approach in detail. We reused our end-to-end comparison scenario (see Figure 3.20), where we executed 100,000 instances of our example plan P_5 and compared the resulting execution time when using periodical re-optimization with and without the use of our correlation table. In contrast to the original comparison scenario, we generated correlated[7] data. Figure 3.29(a) illustrates the conditional selectivities $P(o_2)$, $P(o_3|o_2)$, and $P(o_4|o_2 \wedge o_3)$ of the three `Selection` operators, where we additionally set $P(o_3|\neg o_2) = 1$ and $P(o_4|\neg o_2 \vee \neg o_3) = 1$. As a result, o_3 strongly depends on o_2 as well as o_4 strongly depends on o_2 and o_3.

Figure 3.29(b) illustrates the resulting execution time with and without the use of our correlation table. We observe that without the use of the correlation table, the optimization technique selection reordering assumes statistical independence and thus, changed the plan back and forth, even in case of constant workload characteristics. This led to the periodic use of suboptimal plans, where the optimization interval $\Delta t = 5\,\text{min}$ prevented more frequent plan changes. In contrast, the use of the correlation table ensured robustness by

[7] We did not use the Pearson correlation coefficient and known data generation techniques [Fac10] in order to enable the exact control of unconditional and conditional selectivities.



(a) Selectivity Variations (b) Execution Time

Figure 3.29: Comparison Scenario of Periodical Re-Optimization with Correlation

maintaining conditional probabilities over multiple versions of a plan. Due to the involvement of three correlated operators, incremental optimization required deleting records from this correlation table. As a result, there are also some plan switches to suboptimal plans (e.g., after workload shift *2, we observe three wrong plan switches). However, over time, the conditional selectivity estimates converge to the real selectivities, which reasons a 10% improvement with regard to the cumulative execution time. In conclusion, the use of our correlation table ensures robustness in the presense of correlated data or conditional probabilities, while the overhead is negligible (in this comparison scenario, we maintained three entries in this correlation table).

The details of our exhaustive evaluation have shown that significant performance improvements can be achieved by periodical re-optimization in the sense of minimizing the average execution time of a plan, while only moderate overhead is imposed by statistics monitoring and periodical re-optimization. Even in the case, where no optimization techniques could be applied, no significant performance penalty was measured. Most importantly, the optimized plans show a better scalability than unoptimized plans. Thus, typically, the relative performance improvements increase with an increasing input data size or an increasing number of operators. Finally, with the right choice of parameters the self-adjusting cost model in combination with correlation awareness enables a fast but still robust adaptation to changing workload characteristics.

3.6 Summary and Discussion

To summarize, in this chapter, we introduced the *cost-based* optimization of *imperative* integration flows to overcome the major problem of inefficiently performing integration flows in the presence of changing workload characteristics. The incremental maintenance of execution statistics addresses missing knowledge about data properties when integrating heterogeneous and highly distributed systems and applications. In the area of integration flows, cost-based re-optimization has been considered for the first time. Our solution comprises the dependency analysis and details on the monitoring of workload and execution statistics as well as the definition of the double-metric cost model. Based on these foundations, we discussed the NP-hard *Periodic Plan Optimization Problem (P-PPO)*, including

our transformation-based optimization algorithm, approaches for search space reduction and adjusting the sensibility of workload adaptation as well as a lightweight concept for handling conditional probabilities and correlation. Further, we explained selected optimization techniques that are specific to integration flows because they exploit both the data flow *and* the control flow in a combined manner. Our evaluation shows significant performance improvements with moderate overhead for periodical re-optimization.

In conclusion, our cost-based optimization approach can be integrated seamlessly into the major products in the area of integration platforms. Based on the observation of many independent instances of integration flows, this approach of periodical cost-based re-optimization is tailor-made for integration flows. In detail, the advantages of periodical re-optimization are (1) the asynchronous optimization independently of executing certain instances, (2) the fact that all subsequent instances rather than only the current query benefit from re-optimization, and (3) the inter-instance plan change without the need of state migration. This general optimization framework can be used as foundation for further rewriting techniques and optimization approaches.

Apart from these re-optimization advantages, the optimization framework presented so far has still several shortcomings. First, only the optimization objective of minimizing the average plan execution time was considered. This is not always a suitable optimization objective because in high load scenarios often the major optimization objective is throughput maximization, while moderate latency times are acceptable. Therefore, in the following, we will present two integration-flow-specific optimization techniques that have the potential to significantly increase the message throughput. In detail, we present the *cost-based vectorization* (a control-flow-oriented optimization technique) in Chapter 4 and the *multi-flow optimization* (a data-flow-oriented optimization technique) in Chapter 5. Second, also the periodical re-optimization algorithm itself has several drawbacks. This includes the generic gathering of statistics for all operators that causes the maintenance of statistics that might not be used by the optimizer. While for the evaluated workload aggregation methods, this overhead was negligible, there might be performance issues when using more complex forecast models. In addition, there is the problem of periodically triggered re-optimization, where a new plan is only found if workload characteristics have changed. Otherwise, we trigger many unnecessary invocations of the optimizer that evaluates the complete search space. Depending on the used optimization techniques this can have notable performance implications. However, if a workload change occurs, it takes a while until re-optimization is triggered. During this adaptation delay, we thus use a suboptimal plan and miss optimization opportunities. Finally, the parameter Δt (optimization period) has high influence on optimization and execution times and hence, parameterization requires awareness of changing workloads. These four drawbacks are addressed with the concept of on-demand re-optimization that we will present in Chapter 6. However, the periodical re-optimization already provides a reasonable optimization framework including many fundamental concepts and thus, is used as the conceptual basis of this thesis.

4 Vectorizing Integration Flows

Based on the general cost-based optimization framework, in this chapter, we present the vectorization of integration flows [BHP+09a, BHP+09b, BHP+11] as a control-flow-oriented optimization technique that is tailor-made for integration flows. This technique tackles the problem of low CPU utilization imposed by the instance-based plan execution of integration flows. The core idea is to transparently rewrite instance-based plans into vectorized plans with pipelined execution characteristics in order to exploit pipeline parallelism over multiple plan instances. Thus, this concept increases the message throughput, while it still ensures the required transactional properties. We call this concept vectorization because a vector of messages is processed at-a-time.

In order to enable vectorization, we first describe necessary flow meta model extensions as well as the rule-based plan vectorization that ensures semantic correctness; i.e., the rewriting algorithm preserves the serialized external behavior. Furthermore, we present the cost-based vectorization that computes the optimal grouping of operators to multi-threaded execution buckets in order to achieve the optimal degree of pipeline parallelism and hence, maximize message throughput. We present exhaustive, heuristic, and constrained computation approaches. In addition, we also discuss the cost-based vectorization for multiple deployed plans and we sketch how this rather complex optimization technique is embedded within our periodical re-optimization framework. Finally, the experimental evaluation shows that significant throughput improvements are achieved by vectorization, with a moderate increase of latency time for individual messages. The cost-based vectorization further increases this improvement and ensures robustness of vectorization.

4.1 Motivation and Problem Description

In scenarios with high load of plan instances, the major optimization objective is often throughput maximization, where moderate latency times are acceptable [UGA+09]. Unfortunately, despite the optimization techniques on parallelizing subflows, instance-based plans of integration flows, typically, do not achieve a high CPU utilization.

Problem 4.1 (Low CPU Utilization)**.** *The low CPU utilization is mainly caused by (1) significant waiting times for external systems (for example, the plan instance is blocked, while executing external queries), (2) the trend towards multi- and many-core architectures, which stands in contrast to the single-threaded execution of instance-based integration flows, and (3) the IO bottleneck due to the need for message persistence to enable recoverability of plan instances.*

In conclusion of Problem 4.1 in combination with the existence of many independent plan instances, there are optimization opportunities with regard to the message throughput, which we could exploit by increasing the degree of parallelism. Essentially, we could leverage four different types of parallelism to overcome that problem, where we additionally use the classification [Gra90] of horizontal (parallel processing of data partitions) and vertical parallelism (pipelining):

- *Intra-Operator (Horizontal):* First, the operator implementations can be changed such that they split input messages, work on data partitions of input messages in parallel, and finally merge the results. For example, this type of parallelism is used in the XPEDIA system [BABO+09]. In the context of processing tree-structured messages (XML), the benefit is limited due to a high serial fraction required for splitting and merging of messages.
- *Inter-Operator (Horizontal):* Second, there are the already mentioned techniques on rewriting sequences and iterations to parallel subflows (Subsection 3.4.1), where the scope is intra-instance (or synonymously inter-operator) only. However, in the absence of iterations and due to restrictive dependencies between operators, the benefit of these techniques is limited. Nevertheless, they are valid because they can be applied to all types of flows, while the following two techniques are limited to data-driven integration flows.
- *Inter-Instance (Horizontal):* Third, we could execute multiple plan instances in parallel. Unfortunately, due to the need for logical serialization of plan instances (Section 2.3.2), simple multi-threaded plan execution is not applicable because expensive global serialization would be required at the outbound side. If no serialization is required, this technique has the highest optimization potential.
- *Inter-Operator/Inter-Instance (Vertical):* Fourth, there is the possibility of pipeline parallelism with a scope that stretches across multiple instances of a plan.

In order to overcome the problem of low CPU utilization, we introduce the vectorization of integration flows that uses the fourth possibility by applying pipelining to integration flows. Instance-based plans (that use an one-operator-at-a-time execution model) are transparently rewritten into pipelined plans (that use the pipes-and-filters execution model, where multiple operators represent a pipeline and thus, are executed in parallel). This concept of vectorization applies to data-driven integration flows (that include at least one `Receive` operator). More specifically, it is designed for asynchronous data-driven integration flows, where the client source systems are not blocked during execution of a plan instance. However, it is also extensible for synchronous data-driven integration flows by employing call-back mechanisms in combination with the concept of correlation IDs [OAS06], where waiting clients are logically mapped to asynchronously processed messages. Due to the control-flow semantics of integration flows and the need for ensuring semantic correctness, transparent pipelining as an internal optimization technique (not visible to the user) poses a hard challenge.

With regard to this challenge, in Section 4.2, we introduce the full vectorization of plans, where each operator is executed as a single thread. There, we discuss the rewriting algorithm as well as the rewriting with control-flow-awareness in detail. While, this typically already increases throughput, the full vectorization exhibits a fundamental problem: namely, this execution model might require a high number of threads.

Problem 4.2 (High Number of Required Threads)**.** *If each operator is executed as a single thread, the number of operators determine the number of threads required for a vectorized plan. Thus, the throughput improvement of a concrete plan achieved by vectorization strongly depends on the number of operators of this plan. In dependence on the concrete workload (runtime of a single operator) and the available parallelism of the used hardware platform, a number of threads that is too high can also hurt performance due to additional*

thread monitoring and synchronization efforts as well as increased cache displacement. This problem is strengthened in the presence of multiple deployed plans because there, the total number of operators and thus, the number of threads is even higher. In addition, the higher the number of threads, the higher the latency time of single messages.

To tackle this problem of a possibly high number of required threads, in Section 4.3, we introduce the cost-based vectorization of integration flows that assigns groups of operators to execution buckets and thus, to threads. This reduces the number of required threads and achieves higher throughput. In addition, Section 4.4 discusses how to compute the optimal assignment of threads to multiple deployed plans. Finally, Section 4.5 illustrates how this cost-based vectorization of integration flows, as an optimization technique for throughput maximization, is embedded into our overall optimization framework that was described in Chapter 3.

In contrast to related work on throughput optimization by parallelization of tasks in DBMS [HA03, HSA05, GHP+06], DSMS [SBL04, BMK08, AAB+05] or ETL tools [BABO+09, SWCD09], our approach allows the rewriting of procedural integration flows to pipelined (vectorized) execution plans. Further, existing approaches [CHK+07, CcR+03, BBDM03, JC04] that also distribute operators across a number of threads or server nodes, compute this distribution in a static manner during query deploy time. In contrast, we compute the cost-optimal distribution during periodical re-optimization in order to achieve the highest throughput and to allow the adaptation to changing workload characteristics.

4.2 Plan Vectorization

As a prerequisite, we give an overview of the core concepts of our plan vectorization approach [BHP+09b]. We define the vectorization problem, explain the required modifications of the integration flow meta model, sketch the basic rewriting algorithm, describe context-specific rewriting rules and analyze the costs of vectorized plans.

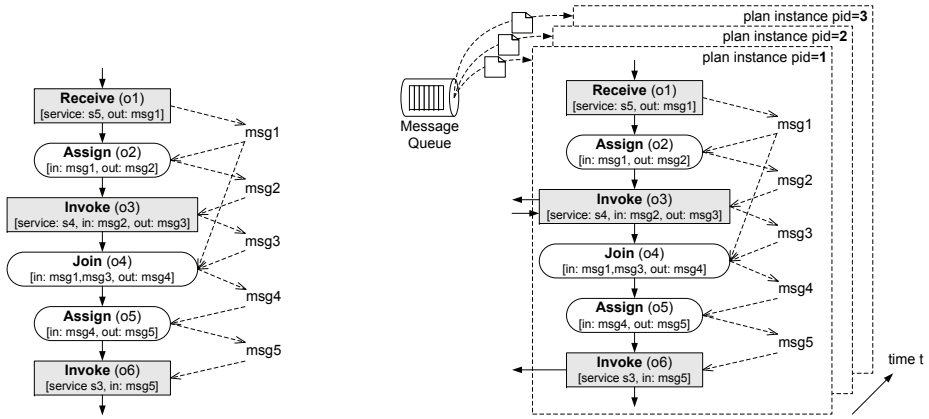
4.2.1 Overview and Meta Model Extension

The general idea of plan vectorization is to transparently rewrite the instance-based plan—where each instance is executed as a thread—into a vectorized plan, where each operator is executed as a single *execution bucket* and hence, as a single thread. Thus, we model a standing plan of an integration flow. Due to different execution times of the single operators, transient inter-bucket message queues (with constraints[8] on the maximum number of messages) are required for each data flow edge. With regard to our classification of execution approaches, we change the execution model from control-flow semantics with instance-local, materialized intermediates to data-flow semantics with a hybrid instance-global data granularity (pipelining of materialized intermediates from multiple instances), while still enabling complex procedural modeling. We illustrate this idea of plan vectorization with an example.

Example 4.1 (Full Plan Vectorization)**.** *Assume the instance-based example plan P_2 as shown in Figure 4.1(a). Further, Figure 4.1(b) illustrates the typical instance-based plan*

[8] Queues in front of cost-intensive operators include larger numbers of messages. In order to overcome the high memory requirements, typically, the (1) maximal number of messages or (2) the maximal total size of messages per queue is constrained. It is important to note that finally, this constraint leads to a work-cycle of the whole pipeline that is dominated by the most time-consuming operator.

execution, where each incoming message initiates a new plan instance. All operators of one instance are executed before the next instance is started.



(a) Example Plan P_2 (b) Instance-Based Plan Execution of P_2

Figure 4.1: Example Instance-Based Execution of Plan P_2

In contrast, Figure 4.2 shows the fully vectorized plan, where each operator is executed within an execution bucket. Note that we also emphasized the changed operator parameters.

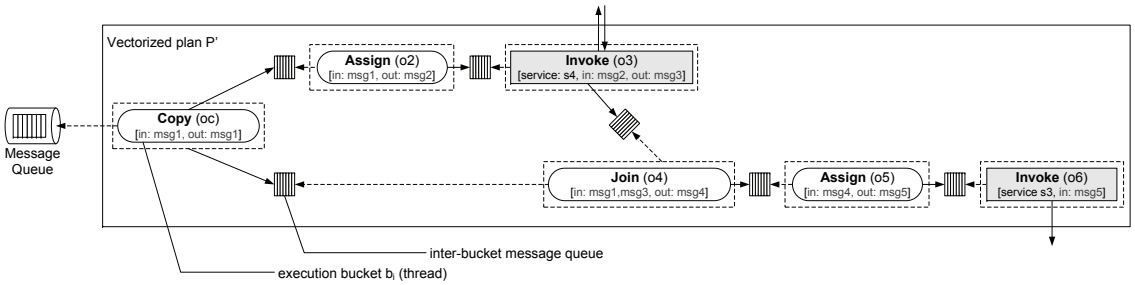
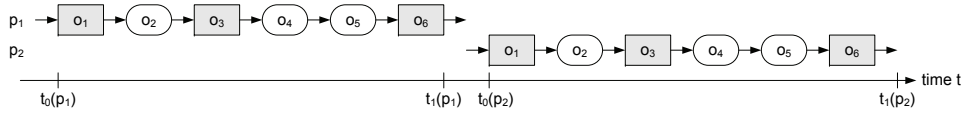


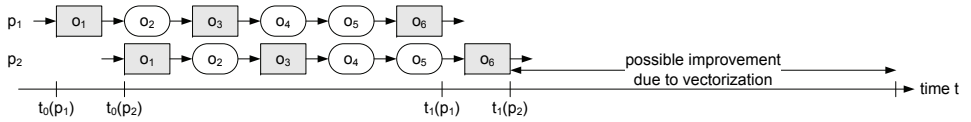
Figure 4.2: Example Fully Vectorized Execution of Plan P_2'

We can leverage pipeline parallelism (within a single pipeline) and parallel pipelines. In this model, each edge of a data flow graph includes a message queue for inter-operator communication. Dashed arrows represent dequeue (read) operations, while normal arrows represent enqueue (write) operations. Additional operators (e.g., the `Copy` *operator for data flow splits) are required, while the* `Receive` *operator is not needed anymore.*

Major challenges have to be tackled when transforming P into P' in order to preserve the control-flow semantics and prevent the external behavior from being changed. Based on the mentioned requirement of ensuring semantic correctness in the form of serialized external behavior, we now formally define the plan vectorization problem. Figure 4.3(a) illustrates the temporal aspects of the example instance-based plan (assuming a sequence of operators). In this case, different instances of this plan are serialized in incoming order. Such an instance-based plan is the input of our vectorization problem. In contrast to this, Figure 4.3(b) shows the temporal aspects of a vectorized plan for the best case. Here, only the external behavior (according to the start time t_0 and the end time t_1 of plan and operator instances) must be serialized. Such a vectorized plan is the output of the vectorization problem. The plan vectorization problem is then defined as follows.



(a) Instance-Based Execution of Plan P



(b) Fully Vectorized Execution of Plan P'

Figure 4.3: Temporal Aspects of Instance-Based and Vectorized Plans

Definition 4.1 (Plan Vectorization Problem (P-PV))**.** *Let P denote a plan, and $p_i \in \{p_1, p_2, \ldots, p_n\}$ denotes the plan instances with $P \Rightarrow p_i$. Further, let the plan P contain a sequence of atomic or complex operators $o_i \in \{o_1, o_2, \ldots, o_m\}$. For serialization purposes, the plan instances are executed in sequence, where the end time t_1 of a plan instance is lower than the start time t_0 of the subsequent plan instance with $t_1(p_i) \leq t_0(p_{i+1})$. Then, the P-PV describes the search for the vectorized plan P' (with data flow semantics) that exhibits the highest degree of parallelism for the plan instances p'_i such that the constraint conditions $(t_1(p'_i, o_i) \leq t_0(p'_i, o_{i+1})) \wedge (t_1(p'_i, o_i) \leq t_0(p'_{i+1}, o_i))$ hold and the semantic correctness (see Definition 3.1) is ensured.*

The same rules of ensuring semantic correctness as used for inter-operator parallelism in Chapter 3 also apply for vectorized plans. For example, this requires synchronization of writing interactions. However, we assume independence of plan instances, which holds for typical data-propagating integration flows. This means that we synchronize, for example, a reading interaction with a subsequent writing interaction of plan instance p_1 but we allow executing the reading interaction of p_2 in parallel to the writing interaction of p_1. Nevertheless, monotonic reads and writes are ensured. We will revisit this issue of intra-instance synchronization when discussing the rewriting algorithm.

Based on the P-PV, we now reveal the static cost analysis of the best case (full pipelines), where cost denotes the total execution time. Let P include an operator sequence o with constant operator costs $W(o_i) = 1$, the costs of n plan instances are

$$\begin{aligned} W(P) &= n \cdot m && \text{// instance-based} \\ W(P') &= n + m - 1 && \text{// fully vectorized} \\ \Delta(W(P) - W(P')) &= (n-1) \cdot (m-1), \end{aligned} \tag{4.1}$$

where m denotes the number of operators. This is an idealized model only used for illustration purposes. In practice, the improvement depends on the most time-consuming operator o_{max} with $W(o_{max}) = \max_{i=1}^{m} W(o_i)$ of a vectorized plan P' because the work-cycle of the whole data flow graph depends on this operator due to filled queues (with maximum constraints) in front of this operator. We will revisit this effect when discussing the cost-based vectorization in Section 4.3. The costs are then specified by:

$$\begin{aligned} W(P) &= n \cdot \sum_{i=1}^{m} W(o_i) && \text{// instance-based} \\ W(P') &= (n + m - 1) \cdot W(o_{max}). && \text{// fully vectorized} \end{aligned} \tag{4.2}$$

Over time and hence, with an increasing number of plan instances n, the performance improvement regarding the total execution time grows linearly. We use the term performance in the sense of high throughput and low execution time of a finite message sequence M'. For this case of a finite sequence, where the incoming order of this sequence must be preserved, (1) the execution time $W(P, M')$, (2) the message throughput $|M'|/\Delta t$, and (3) the latency time $T_L(M')$ are correlated. According to Little's Law [Lit61], the rationale for this is the waiting time within the system because instances of one plan must not be executed in parallel. In more detail, decreasing total execution time of the message subsequence decreases the waiting time, increases the message throughput and thus finally decreases the total latency time of this message sequence:

$$W(P, M') \propto \frac{1}{|M'|/\Delta t} \propto T_L(M'). \tag{4.3}$$

However, the latency of single messages can be higher for vectorized plans compared to instance-based plans.

Message and Flow Meta Model

In order to allow for transparent vectorization as an internal optimization technique, the control-flow semantics must be preserved when vectorizing a plan. Therefore, we extended the *Message Transformation Model (MTM)* (Subsection 2.3.1) in order to make it applicable for vectorized plans, where we refer to it as *VMTM*.

In the VMTM, we extend the message meta model from a triple to a quadruple with $m_i = (t_i, d_i, a_i, c_i)$, where the context information c denotes an additional map of atomic name-value attribute pairs with $c_{ij} = (n_j, v_j)$. This extension is necessary due to processing of multiple messages within one single standing (vectorized) plan instead of independent plan instances. Thus, instance-related context information such as local variables (e.g., counters or extracted attribute values) must be stored within the messages.

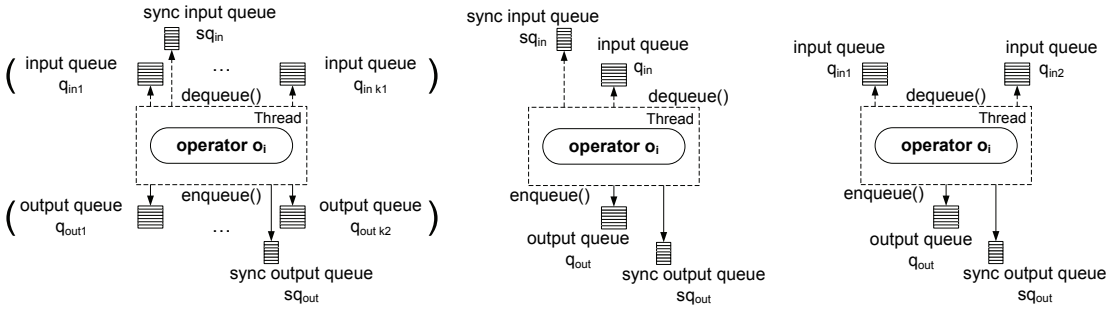
In contrast to the *MTM* flow meta model, in the *VMTM*, the flow relations between operators o_i do not specify the control flow (temporal dependencies) but the explicit data flow in the form of message streams. Additionally, the `Fork` operator is removed because in the vectorized case, operators are inherently executed in parallel. Finally, we introduce the additional operators `And` and `Xor` for synchronization of operators in order to preserve

Table 4.1: Additional Operators of the VMTM

Name	Description	Input	Output	Complex
`And`	Reads a synchronization ID and a single message and forwards the read message.	(2,2)	(1,1)	No
`Xor`	Reads a synchronization ID and/or multiple messages and outputs all messages, which synchronization IDs have already been seen. Thus, this operator has an intra-operator state of IDs and messages.	(1,*)	(0,*)	No
`Copy`	Gets a single message, then copies it $n-1$ times and puts those messages into the n output queues.	(1,1)	(2,*)	No

the control-flow semantics of an integration flow as well as the `Copy` operator for data flow splits. The semantics of these operators are described in Table 4.1.

In order to realize the pipes-and-filter execution model, operators are executed in so-called multi-threaded executed buckets. Figure 4.4 illustrates the conceptual model of such an execution bucket.



(a) Generic Operator (b) Unary Operator (c) Binary Operator

Figure 4.4: Conceptual Model of Execution Buckets

In general, Figure 4.4(a) shows the generic model of an execution bucket that contains a set of input message queues, a set of output message queues, a single so-called input synchronization queue, a single output synchronization queue, and an operator. The bucket is a thread with an endless loop, where in each iteration it dequeues from all input message queues, dequeues from the synchronization queue, executes the operator, and enqueues the results into all output queues. However, with regard to the defined flow meta model, unary (see Figure 4.4(b)) and binary (see Figure 4.4(c)) operators with one or two inputs and a single output are most common. Note that only the unary operators `Xor` and `And` require synchronization input queues, while each operator can have a synchronization output queue, which depends on its position within the plan and the need for serialization. Apart from the synchronization queues, similar operator models are commonly used in the context of data stream management systems [GAW+08] and related system categories [CEB+09, CWGN11].

4.2.2 Rewriting Algorithm

In this subsection, we first describe the core rewriting algorithm and second, we specify the control-flow-specific rewriting techniques, which preserve the external behavior. Both aspects are required in order to enable plan vectorization as an optimization technique.

Core Algorithm

The basic rewriting algorithm that is described in the following can be applied for all types of operators of our integration flow meta model.

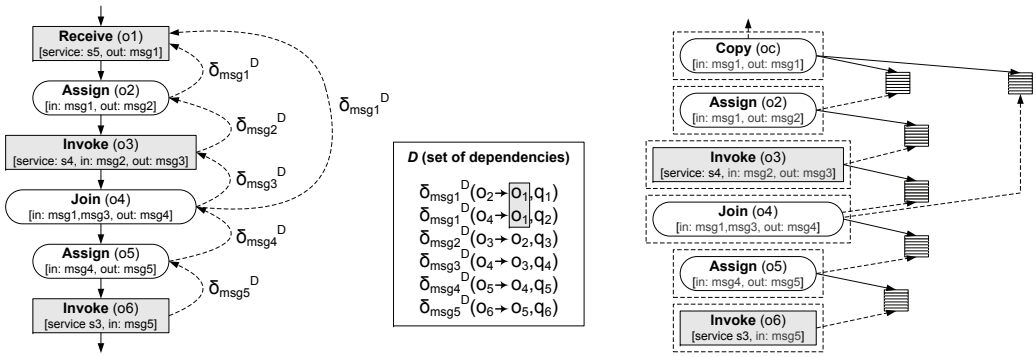
In detail, Algorithm 4.1 consists of two parts. In a first part, the dependency analysis is performed by determining all data dependencies between operators. There, we create a queue instance for each data dependency between two operators (the output message of operator o_i is the input message of operator o_j). Internally, our optimizer reuses the existing dependency graph that was described in Subsection 3.2.1. In a second part, we

Algorithm 4.1 Plan Vectorization (A-PV)

Require: operator sequence o
1: $B \leftarrow \emptyset$, $D \leftarrow \emptyset$, $Q \leftarrow \emptyset$
2: **for** $i \leftarrow 1$ **to** $|o|$ **do** *// for each operator o_i*
3: *// Part 1: Dependency Analysis*
4: **for** $j \leftarrow i$ **to** $|o|$ **do** *// for each following operator o_j*
5: **if** $\exists o_j \xrightarrow{\delta} o_i$ **then**
6: $Q \leftarrow Q \cup q$ with $q \leftarrow$ `createQueue()`
7: $D \leftarrow D \cup d\langle o_j, q, o_i\rangle$ with $d\langle o_j, q, o_i\rangle \leftarrow$ `createDependency()`
8: *// Part 2: Graph Creation*
9: **if** $o_i \in$ `Switch`, `Iteration`, `Validate`, `Signal`, `Savepoint`, `Invoke` **then**
10: *// see rules on rewriting context-specific operators*
11: **else**
12: $b(o_i) \leftarrow$ `createBucket`(o_i)
13: **for** $k \leftarrow 1$ **to** $|D|$ **do** *// for each dependency d*
14: $d\langle o_y, q, o_x\rangle \leftarrow d_k$
15: **if** $o_i \equiv o_x$ **then**
16: $b(o_i)$.`addOutputQueue`(q)
17: **else if** $o_i \equiv o_y$ **then**
18: $b(o_i)$.`addInputQueue`(q)
19: **if** $b(o_i)$.`countOutputQueues()` ≥ 2 **then**
20: `createCopyOperatorAfter`$(b(o_i), B, D, Q)$
21: **else if** $b(o_i)$.`countOutputQueues()` $= 0$ **then**
22: `createLogicalQueue`$(b(o_{i-1}), b(o_i), B, D, Q)$
23: $B \leftarrow B \cup b(o_i)$
24: **return** B

create the graph of execution buckets. This part contains the following three steps. First, we create an execution bucket for each operator. Second, we connect each operator with the referenced input queues. Clearly, each queue is referenced by exactly one operator, but each operator can reference multiple queues. Third, we connect each operator with the referenced output queues. If one operator must be connected to n output queues with $n \geq 2$ (its results are used by multiple following operators), we insert a `Copy` operator after this operator in order to send the message to all dependency sources. Within the `createCopyOperatorAfter` function the new operator, execution bucket and queue are created as well as the dependencies and queue references are changed accordingly. Furthermore, if an operator is connected to zero input queues, we insert an artificial dummy queue (where empty message objects are passed) between this operator and its former temporal predecessor. Based on this basic rewriting concept, additional rewriting rules for context-specific operators (e.g., `Switch`, `Iteration`) and for serialization and recoverability are required (line 10). We use an example to illustrate the A-PV in more detail.

Example 4.2 (Automatic Plan Vectorization)**.** *Recall our example plan P_2 as well as the vectorized plan P_2' of Example 4.1. Figure 4.5(a) and 4.5(b) illustrate the core aspects when rewriting P_2 to P_2'. First, we determine all direct data dependencies δ and add those to the set of dependencies D. If an operator is referenced by multiple dependencies (e.g., operator o_1 in this example), we need to insert a* `Copy` *operator just after it. Furthermore,*



(a) Dependency Graph $DG(P_2)$ of Plan P_2 (b) Vectorized Plan P_2'

Figure 4.5: Example Plan Vectorization

we create a queue q_i for each dependency and connect the operators with these queues. Finally, we simply remove all temporal dependencies and get P'.

Although the A-PV is only executed once during the initial deployment of an integration flow, it is important to note its time complexity with increasing number of operators m.

Theorem 4.1. *The A-PV exhibits a cubic worst-case time complexity of $O(m^3)$.*

Proof. Basically, we prove the complexity for the two algorithm parts: the dependency analysis and the graph creation. Assume an operator sequence o. We fix the number of operators m with $m = |o|$. Then, an arbitrary operator o_i with $1 \leq i \leq m$ can—in the worst case—be the target of $i-1$ data dependencies δ_i^-, and it can be the source of $m-i$ data dependencies δ_i^+. Based on the equivalence of $\delta^- = \delta^+$ and thus, $|\delta^-| = |\delta^+|$, there are at most

$$\sum_{i=1}^{m}(i-1) = \sum_{i=1}^{m-1} i = \frac{m \cdot (m-1)}{2} \tag{4.4}$$

dependencies between arbitrary operators of this sequence. Hence, the dependency analysis is computable with quadratic complexity of $O(m^2)$.

When evaluating operator o_i, there are at most $\sum_{i=1}^{m-1} i = m \cdot (m-1)/2$ (for cases $i = m-1$ and $i = m$) dependencies in D. During the graph creation, for each operator, each dependency $d \in D$ must be evaluated in order to connect queues and execution buckets. In summary, for all operators, we must check at most

$$\begin{aligned}\sum_{i=1}^{m}\sum_{j=1}^{i}(m-j) &= m \cdot \frac{m \cdot (m+1)}{2} - \sum_{i=1}^{m} i \cdot (m-i-1) \\ &= \frac{m^2 \cdot (m+1)}{2} + \sum_{i=1}^{m} i^2 - \sum_{i=1}^{m} m \cdot i + \sum_{i=1}^{m} i = \frac{m^3 - m}{3}\end{aligned} \tag{4.5}$$

dependencies for graph creation. Hence, the algorithm part of graph creation is computed with cubic complexity of $O(m^3)$. In summary, the plan vectorization algorithm exhibits a cubic worst-case time complexity of $O(m^3) = O(m^3 + m^2)$. Hence, Theorem 4.1 holds. □

Note that this is the complexity analysis of our A-PV algorithm, while the P-PV problem can be solved with quadratic time complexity of $O(m^2)$. For example, one can use

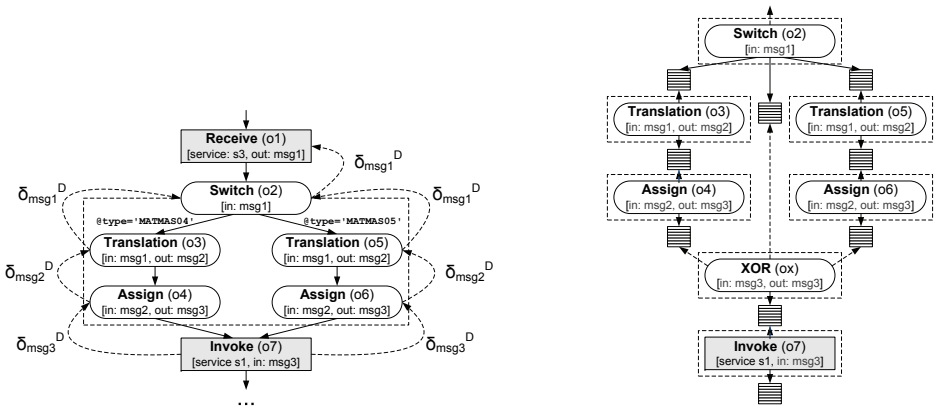
two additional hash maps as secondary index structures for source operators and target operators over the set of data dependencies D. Then, for any operator o_i we only need to iterate over the lists of its own source and target dependencies. This leads to a quadratic worst-case time complexity for the algorithm part of graph creation. However, for the general case, where we need to iterate over all dependencies, we have cubic complexity. Furthermore, for low numbers of data dependencies, the overhead of the A-PV is moderate and therefore, we use the more general form.

Rewriting Context-specific Operators

In addition to the core rewriting algorithm, we now discuss rewriting rules for the context-specific operators `Switch`, `Iteration`, and `Savepoint` as well as specific situations, where multiple writes to external systems must be synchronized (`Invoke`) in order to guarantee semantic correctness when applying plan vectorization.

Rewriting `Switch` operators. When rewriting `Switch` operators, we must be aware of their ordered if-elseif-else semantics. Here, message sequences are routed along different switch-paths, which will eventually be merged. Assume a message sequence of m_1 and m_2, where m_1 is routed to path A, while m_2 is routed to path B. If $W(A) \geq W(B) + W(Switch_B)$, m_2 arrives earlier at the merging point than m_1 does. Hence, a *message outrun* has taken place. In order to overcome this problem, we could use timestamp comparison at the merging point. Therefore, we introduced the `XOR` operator that is inserted just before the single switch paths are merged. It reads from all queues, compares the timestamps of read messages and forwards the oldest. Due to the possibility of message starvation (we are not allowed to forward a message until we read a younger message from all other switch paths) in combination with possibly nested `Switch` operators, we use a so-called synchronization queue that represents the temporal order of messages and thus, by comparing the message source IDs with the read synchronization IDs, we overcome the problem of starvation because we can output messages according to this sequence of IDs. The dedicated synchronization queue is required due to arbitrarily nested `Switch` operators, where the assumption of a cohesive sequence of source IDs does not hold.

Example 4.3 (Rewriting `Switch` Operators)**.** *Assume the dependency graph $DG(P_1)$ of*



(a) Dependency Graph $DG(P_1)$ of Subplan P_1 (b) Vectorized Subplan P_1'

Figure 4.6: Rewriting `Switch` Operators

a subplan of our example plan P_1 shown in Figure 4.6(a). The vectorized plan P_1' is the output of the A-PV and it is shown in Figure 4.6(b). There, the `Switch`*-specific rewriting technique has been applied, where we have created two pipeline branches (one for each switch-path) and changed the data flow such that messages are passed through the* `Switch` *operator. In order to avoid message outrun, we inserted the* `XOR` *operator and the synchronization queue. Note that the full plan of P_1 required additional* `Copy` *and* `And` *operators for the last* `Invoke` *operator because it depends directly on the output of the* `Translation` *operator. This serialization concept will be discussed separately.*

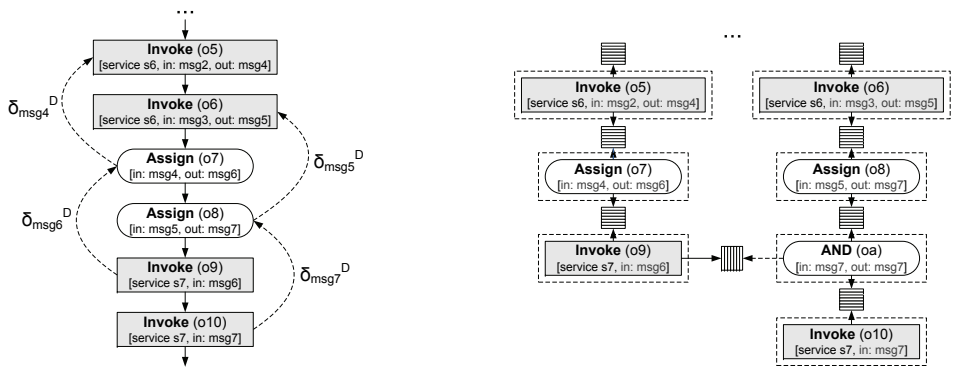
Rewriting `Iteration` *operators.* When rewriting `Iteration` operators, the main problem is also the *message outrun*. We must ensure that all iteration loops (for a message) have been processed before the next message enters. Basically, a *foreach* `Iteration` is rewritten to a sequence of (1) one `Split` operator, (2) operators of the `Iteration` body and (3) one `Setoperation` (`UNION ALL`) operator. Using this strategy, inherently leads to the highest degree of parallelism, while it requires only moderate additional costs for splitting and merging. In contrast to this, iterations with *while* semantics are not vectorized (one single execution bucket) because we cannot guarantee semantic correctness.

Rewriting `Savepoint` *operators.* Within the instance-based model, we can use message-specific and context-specific savepoints. Due to the missing global context, we need to reduce the savepoint semantics to the storage of messages, where context information needs to be stored via specific messages. However, in order to ensure the semantic correctness, we require two different rewriting methodologies. The message-specific savepoint is simply vectorized in a standard manner. In contrast to this, the context-specific savepoint, that stores all current messages at a certain plan position, must be rewritten in a more complex way. Here, we need to insert one savepoint into each parallel data flow branch with respect to the operator position in the instance-based case.

Rewriting `Invoke` *operators.* In order to realize the serialization of external behavior (precondition for transparency), we must ensure that explicitly modeled sequences of writing interactions (`Invoke` operators) are serialized (see Rule 3 of Definition 3.1). Hence, we use the `And` operator for synchronization purposes. If (1) two `Invoke` operators have a temporal dependency within P, (2) they perform a writing interaction to the same external system, and (3) they are included in different pipelines in P', we insert an `And` operator right before the second `Invoke` operator as well as a synchronization queue between the first `Invoke` operator and the `And` operator. The `And` operator reads from the synchronization queue and from the original queue and synchronizes the external behavior by deferring all messages until its source message ID is available in the synchronization queue. We use an example to illustrate this concept.

Example 4.4 (Serialization of External Behavior)**.** *Assume a dependency graph $DG(P_8)$ of a subplan of our example plan P_8 (see Figure 4.7(a)) to be part of a data-driven integration flow. If we vectorize this subplan to P_8' (see Figure 4.7(b)) with two pipeline branches, we need to ensure the serialized external behavior. We inserted an* `And` *operator, where the first* `Invoke` *sends synchronizing source message IDs to this operator using the introduced synchronization queue. Only in the case that the* `Assign` *as well as the first* `Invoke` *have been processed successfully, the payload message of the right pipeline branch is forwarded to the second* `Invoke`*.*

In summary, when rewriting instance-based plans to vectorized plans, we guarantee semantic correctness for context-specific operators as well.



(a) Dependency Graph $DG(P_8)$ of Subplan P_8 (b) Vectorized Subplan P_8'

Figure 4.7: Rewriting `Invoke` Operators

4.2.3 Cost Analysis

We already discussed the cost analysis of sequences of operators, where each operator o_i has a single data dependency with the previous operator o_{i-1}. In addition, we investigate the costs with regard to the specific rewriting results. Therefore, we reuse the idealized model where each operator exhibits constant costs with $W(o_i) = 1$. Similar to the case of operator sequences, this can be extended to the case of arbitrary operator costs.

Parallel data flow branches. In the case of different data flow branches, messages are processed by $|r|$ concurrent pipelines within one vectorized plan, where a single pipeline r_i contains $|r_i|$ operators. Examples for this type of branches are simply overlapping data dependencies, as well as the `Switch` and the `Fork` operator. In this case of multiple branches with $|r| \geq 2$, the idealized costs for processing n messages are

$$\begin{aligned} W(P) &= \begin{cases} n \cdot m & \text{instance-based (unoptimized)} \\ n \cdot \max_{i=1}^{|r|}(|r_i|) & \text{instance-based (optimized, if applicable)} \end{cases} \\ W(P') &= n + \max_{i=1}^{|r|}(|r_i|) - 1 \quad \text{fully vectorized,} \end{aligned} \tag{4.6}$$

where $\max_{i=1}^{|r|}(|r_i|)$ denotes the longest branch. The benefit compared to inter-operator (horizontal) parallelism depends on the optimization techniques that could be applied on the instance-based representation because not all operators without data dependencies can be parallelized (e.g., sequence of writing interactions). Furthermore, in the case of $|r| = 1$ and thus, $|r_1| = m$, the general cost analysis stays true. The improvement is caused by the higher degree of parallelism. However, the presence of parallel data flow branches may also cause overhead for vectorized plans with regard to splitting and merging those branches. An example for the splitting of branches is the `Copy` operator that is used for multiple dependencies on one message. Further, examples for the merging of branches are the `And` operator for synchronizing external writes as well as the `Xor` operator for synchronizing the `Switch` operator.

Rolled-out `Iteration`. Further, the rewriting of `Iteration` operators with *foreach* semantics needs some consideration. Here, we split messages according to the *foreach* condition and execute the iteration body as inner pipeline without cyclic dependencies. Finally, the processed sub messages are merged using the `Setoperation` operator (`UNION`

`ALL`). In fact, in the instance-based case, the costs of processing n messages are determined by $W(P) = n \cdot r \cdot m$, where r denotes the number of iteration loops for each message and m denotes the number of operators in the iteration body. When rewriting the `Iteration` operator to parallel flows, in the best case, the costs are reduced to $W(P) = n \cdot m$ because all iteration loops are executed in parallel. In contrast, due to the sub pipelining of a vectorized plan, we can reduce the costs to $W(P') = n \cdot r + m - 1 + 2$, where r denotes the number of sub-messages. We see that costs of 2 must be added to represent the costs for message splitting (`Split`) and merging (`Setoperation`). Furthermore, the optimality of vectorized execution is given if $r \leq m$.

Finally, note that this is a best-case consideration using an idealized static cost model supposed for illustration purposes. This does not take into account a changed number of operators during vectorization or additional costs for synchronization. However, in the following sections, we will revisit these issues.

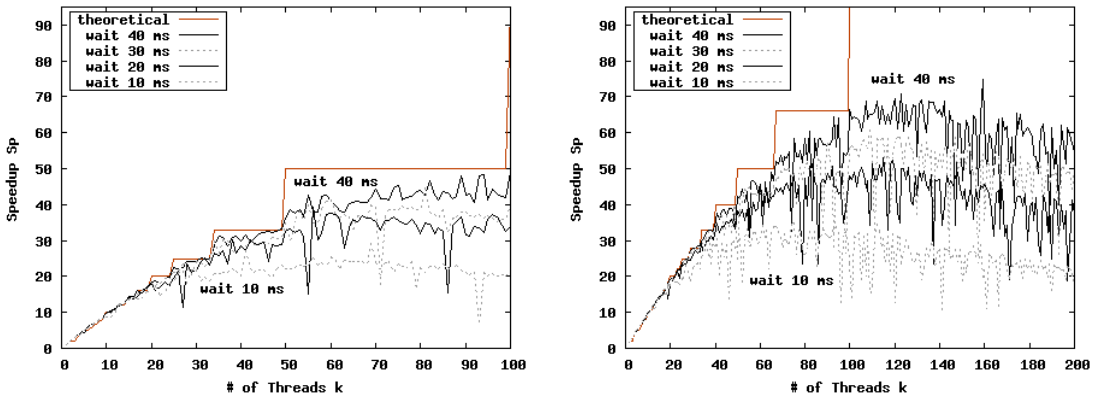
In conclusion, plan vectorization strongly increases the degree of parallelism and thus, may lead to a higher CPU utilization. In this section, we introduced the basic vectorization approach and the required meta model extensions. In addition, we described the core rewriting algorithm as well as the specific rewriting rules that are necessary in order to guarantee semantic correctness.

4.3 Cost-Based Vectorization

Plan vectorization rewrites an instance-based plan (one execution bucket per plan) into a fully vectorized plan (one execution bucket per operator), which solves the P-PV. However, the approach of full vectorization has two major drawbacks. First, the theoretical performance and latency of a vectorized plan mainly depends on the performance of the most time-consuming operator. The reason is that the work cycle of a whole data-flow graph is given by the longest running operator because all queues after this operator are empty, while queues in front of it reach their maximum constraint. Similar theoretical observations have also been made for task scheduling in parallel computing environments [Gra69], where Graham described bounds on the overall time influence of task timing anomalies, which quantify the optimization potential vectorized plans still exhibit. Second, the practical performance also strongly depends on the number of operators because each operator requires a single thread. Depending on the concrete workload (runtime of operators), a number of threads that is too high can also hurt performance due to (1) additional thread monitoring and synchronization efforts as well as (2) cache displacement because the different threads work on different intermediate result messages of a plan.

Figure 4.8 shows the results of a speedup experiment from Chapter 3, which was re-executed for two plans with $m = 100$ and $m = 200$ `Delay` operators, respectively. Then, we varied the number of threads ($k \in [1, m]$) as well as the delay time in order to simulate the waiting time of `Invoke` operators for external systems. Furthermore, we computed the speedup by $S_p = W(P, 1)/W(P', k)$. The theoretical maximum speedup is $m/\lceil m/k \rceil$. As a result, we see that the empirical speedup increases, but decreases after a certain maximum. There, the maximum speedup and the number of threads, where this maximum speedup occurs depends on the waiting time, i.e., the higher the waiting time, the higher the reachable speedup and the higher the number of threads, where this maximum occurs.

In conclusion, an enhanced vectorization approach is required that takes into account the execution statistics of single operators. In this section, we introduce a generalization



(a) P with $m = 100$ (b) P with $m = 200$

Figure 4.8: Speedup Test with Varying Degree of Parallelism

of the P-PV and present our cost-based vectorization approach [BHP+09a, BHP+11] that overcomes the two drawbacks of vectorization. The instance-based plan and the fully vectorized plan are then specific cases of this more general solution. This cost-based vectorization directly relies on the foundations of our general cost-based optimization framework in the form of a cost-based, control-flow-oriented optimization technique.

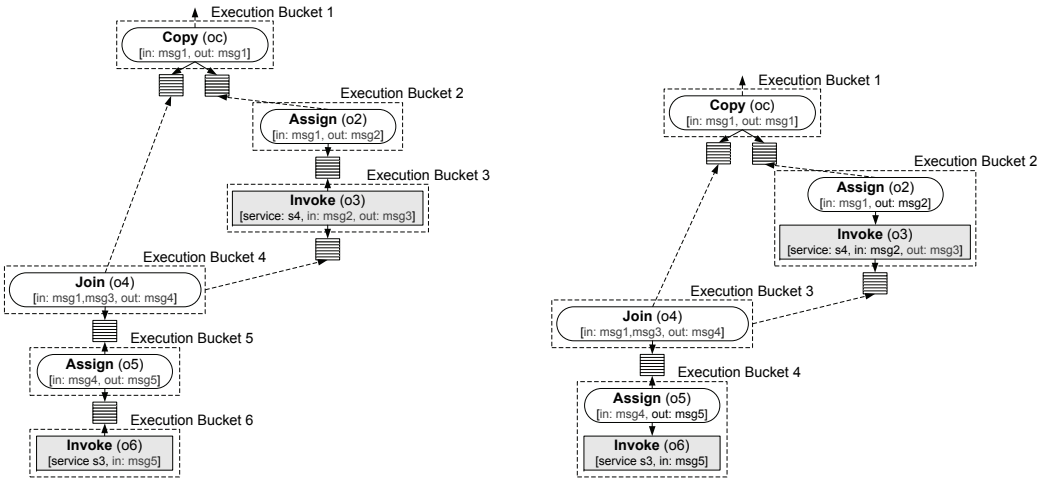
4.3.1 Problem Generalization

The core idea of this problem generalization is to rewrite an instance-based plan to a cost-based vectorized plan with a minimal number of execution buckets, where each bucket can contain multiple operators. All operators of a single execution bucket are executed instance-based, while the set of execution buckets use the pipes-and-filter execution model.

Example 4.5 (Cost-Based Vectorization)**.** *Recall the vectorized plan P_2' (shown in Figure 4.9(a)) from Example 4.1. If we now use cost-based vectorization, we search for the cost-optimal plan P_2'' with k execution buckets. Figure 4.9(b) illustrates an example of such a cost-based vectorized plan. There, $k = 4$ execution buckets are used, while buckets 2 and 4 include two operators. The individual operators of each bucket are executed with the instance-based execution model. As a result, we require only four instead of six threads for this plan.*

The input (instance-based plan) and the output (vectorized plan) of the P-PV are extreme cases of this generalization. In order to compute the cost-optimal vectorized plan, we generalize the P-PV to the Cost-Based P-PV:

Definition 4.2 (Cost-Based Plan Vectorization Problem (P-CPV))**.** *Let P denote a plan, and $p_i \in \{p_1, p_2, \ldots, p_n\}$ denotes the implied plan instances with $P \Rightarrow p_i$. Further, let each plan P comprise a sequence of atomic and complex operators $o_i \in \{o_1, o_2, \ldots, o_m\}$. For serialization purposes, the plan instances are executed in sequence with $t_1(p_i) \leq t_0(p_{i+1})$. The P-CPV describes the search for the derived cost-optimal plan P'' according to the optimization objective ϕ with $k \in [1, m]$ execution buckets $b_i \in \{b_1, b_2, \ldots, b_k\}$, where each bucket contains l operators $o_i \in \{o_1, o_2, \ldots, o_l\}$. Here, the constraint conditions $(t_1(p_i'', b_i) \leq t_0(p_i'', b_{i+1})) \wedge (t_1(p_i'', b_i) \leq t_0(p_{i+1}'', b_i))$ and $(t_1(b_i, o_i) \leq t_0(b_i, o_{i+1})) \wedge$*



(a) Vectorized Plan P_2' (b) Cost-Based Vectorized Plan P_2''

Figure 4.9: Example Cost-Based Plan Vectorization

$(t_1(b_i, o_i) \leq t_0(p''_{i+1}, b_i))$ *must hold. We define that* $(l_{b_i} \geq 1) \wedge (l_{b_i} \leq m)$ *and* $\sum_{i=1}^{|b|} l_{b_i} = m$ *and that each operator* o_i *is assigned to exactly one bucket* b_i*.*

An instance-based plan P is a specific case of the cost-based vectorized plan P'', with $k = 1$ execution buckets. Similarly, the fully vectorized plan P' is also a specific case of the cost-based vectorized plan P'', with $k = m$ execution buckets, where m denotes the number of operators. Figure 4.10 illustrates the resulting spectrum of cost-based vectorization.

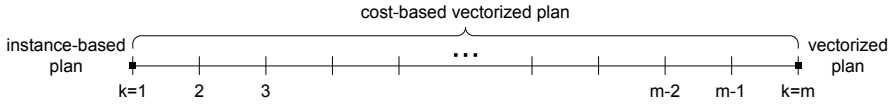
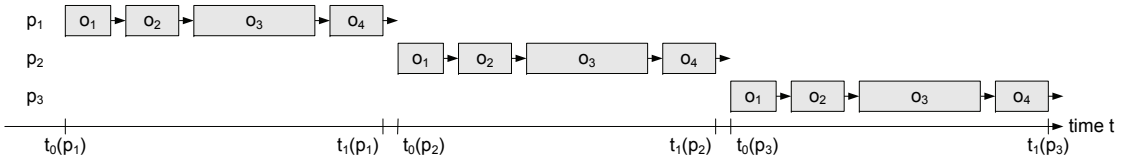


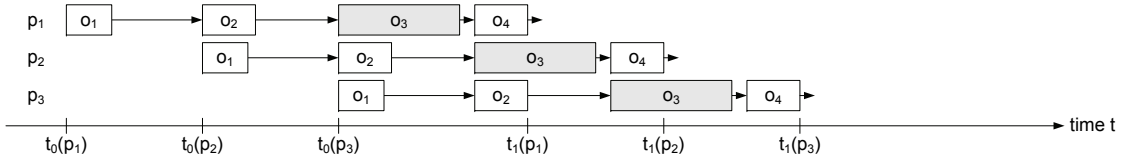
Figure 4.10: Spectrum of Cost-Based Vectorization

At this point, we need to define the optimization objective ϕ, where in general, arbitrary objectives could be used. However, the goal of the vectorization optimization technique is message throughput improvement. Thus, the optimization objective is to reach the highest degree of pipeline parallelism with a minimal number of threads.

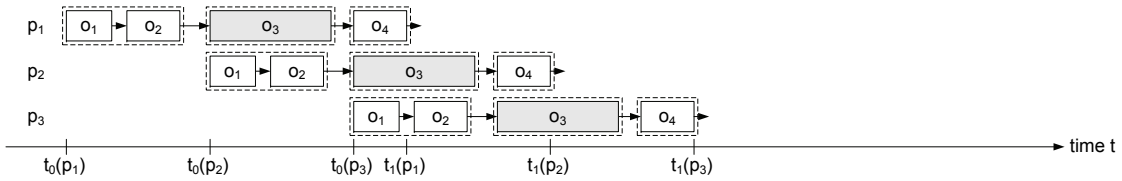
The core idea of this objective is illustrated in Figure 4.11. In case of an instance-based plan, all operators, except for parallel subflows, are included in the critical path that is shown as gray-shaded operators. In contrast, in case of a vectorized plan that includes a sequence of operators o with data dependencies between these operators, the execution time mainly depends on the most time-consuming operator o_{max} with $W(o_{max}) = \max_{i=1}^{m} W(o_i)$. The reason is that queues in front of this most time-consuming operator reach their maximum constraints—in case of full system utilization—and thus, the costs of a vectorized plan are computed with $W(P') = (n + m - 1) \cdot W(o_{max})$. This execution characteristic, that the work-cycle of a pipeline depends on its most time-consuming subtask, is known from other research areas (e.g., databases and operating systems) as the *convoy effect* [Ros10, BGMP79]. As a result of this effect, the work cycle of the vectorized plan is given by $W(o_{max})$ with the time period between the start of two subsequent plan instances $W(o_{max}) = t_0(p_{i+1}) - t_0(p_i)$. For example, operator o_3 dominates the work cycle



(a) Instance-Based Plan P



(b) Fully Vectorized Plan P'



(c) Cost-Based Vectorized Plan P''

Figure 4.11: Work Cycle Domination by Operator o_3

of plan P' in Figure 4.11(b). In conclusion, we leverage the waiting time during work cycles of the data flow graph and merge operators into execution buckets if applicable. Formally, this optimization objective is defined as follows:

$$\phi = \min_{k=1}^{m} k \quad | \quad \forall i \in [1,k] : \left(\sum_{j=1}^{l_{b_i}} W(o_j) \right) \leq W(o_{max}) \tag{4.7}$$

The goal is to find the minimal number of execution buckets k under the restriction that the execution time of each bucket b_i (sum of execution times of the l_{b_i} operators of this bucket) does not exceed the execution time of the most time-consuming operator. As a result, we achieve the highest degree of parallelism with a minimal number of threads. Further advantages of this concept are reduced latency time for single messages and robustness in the case of many plan operators but limited thread resources. The special case of the P-CPV with optimization objective ϕ, where all operators are independent (no data dependencies), is reducible to the NP-hard offline *bin packing* problem [Joh74].

Typically, the optimization objective ϕ allows to find a scheme that exploits the highest pipeline parallelism but requires fewer threads than the full vectorization. However, in special cases such as (1) where all operators exhibit almost the same execution time or (2) where a plan contains too many operators, the problem of a large number of required threads still exist. In order to overcome this general problem, we extend the P-CPV by a parameter to allow for higher robustness. In detail, this extended optimization problem is defined as follows:

Definition 4.3 (Constrained P-CPV)**.** *With regard to the P-CPV, find the minimal number of k buckets and an assignment of operators o_j with $j \in [1,m]$ to those execution buckets*

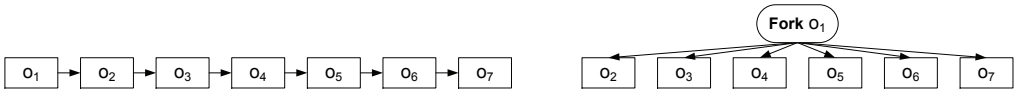
b_i with $i \in [1,k]$ according to the constrained optimization objective

$$\phi_c = \min_{k=1}^{m} k \quad | \quad \forall i \in [1,k] : \left(\sum_{j=1}^{l_{b_i}} W(o_j) \right) \leq W(o_{max}) + \lambda, \tag{4.8}$$

where λ (with $\lambda \geq 0$) is a user-defined absolute parameter to control the cost constraint. There, $\lambda = 0$ leads to the highest meaningful degree of parallelism, while higher values of λ lead to a decrease of parallelism.

Essentially, one can configure the absolute parameter λ in order to influence the number of threads. The higher the value of λ, the more operators are assigned to single execution buckets, and thus, the lower the number of buckets and the lower the number of required threads. However, the decision on merging operators is still made in a cost-based manner.

Based on the defined cost-based vectorization problems, we now investigate the resulting search space. Essentially, both problems exhibit the same search space because they only differ in the optimization objective ϕ, where the worst-case time complexity depends on the structure of a given plan. Figure 4.12 illustrates the best case and the worst case.



(a) Best-Case Plan P_b (b) Worst-Case Plan P_w

Figure 4.12: Plan-Dependent Search Space

The best case from a computational complexity perspective is the sequence of operators (see Figure 4.12(a)), where each operator has a data dependency to its predecessor. Here, the order of operators must be preserved when assigning operators to execution buckets. In contrast, the worst-case is a set of operators without any dependencies between operators (see Figure 4.12(b)) because there, we could create arbitrary combinations of operators. We use an example to illustrate the resulting plan search space for the best case.

Example 4.6 (Operator Distribution Across Buckets)**.** *Assume a plan P with a sequence of four operators ($m = 4$). Table 4.2 shows the possible plans for the different numbers of buckets k.*

Table 4.2: Example Operator Distribution

	$\|b\|$	b_1	b_2	b_3	b_4
Plan 1	$k=1$	o_1, o_2, o_3, o_4	-	-	-
Plan 2	$k=2$	o_1	o_2, o_3, o_4	-	-
Plan 3		o_1, o_2	o_3, o_4	-	-
Plan 4		o_1, o_2, o_3	o_4	-	-
Plan 5	$k=3$	o_1	o_2	o_3, o_4	-
Plan 6		o_1	o_2, o_3	o_4	-
Plan 7		o_1, o_2	o_3	o_4	-
Plan 8	$k=4$	o_1	o_2	o_3	o_4

We can distinguish eight different ($2^{4-1} = 8$) plans, where Plan 1 is the special case of an instance-based plan and Plan 8 is the special case of a fully vectorized plan.

In the following, we formally analyze the time complexity for exhaustively solving this problem. For this purpose, we analyze the complexity of the best and worst case and combine this to the general result.

Lemma 4.1. *The cost-based plan vectorization problem exhibits an exponential time complexity of $O(2^m)$ for the best-case plan of an operator sequence.*

Proof. The distribution function $\mathcal{D}$ of the number of possible plans over k is a symmetric function according to *Pascal's Triangle*[9], where the condition $l_{b_i} = l_{b_{k-i+1}}$ with $i \leq m/2$ holds. Based on Definition 2.1, a plan contains m operators. Due to Definition 4.2, we search for k execution buckets b_i with $l_{b_i} \geq 1 \wedge l_{b_i} \leq m$ and $\sum_{i=1}^{|b|} l_{b_i} = m$. Hence, different numbers of buckets $k \in [1, m]$ have to be evaluated. From now on, we fix m' as $m' = m-1$ and k' as $k' = k-1$. In fact, there is only one possible plan for $k = 1$ (all operators in one bucket) and $k = m$ (each operator in a different bucket), respectively:

$$|P|_{k'=0} = \binom{m'}{0} = 1 \quad \text{and} \quad |P'|_{k'=m'} = \binom{m'}{m'} = 1 \ . \tag{4.9}$$

Now, without loss of generality, we fix a specific m. The number of possible plans for a given k is then computed with

$$|P''|_k = \binom{m'}{k'} = \binom{m'-1}{k'-1} + \binom{m'-1}{k'} = \prod_{i=1}^{k'} \frac{m'+1-i}{i}. \tag{4.10}$$

In order to compute the total number of possible plans, we have to sum up the possible plans for each k, with $1 \leq k \leq m$:

$$|P''| = \sum_{k'=0}^{m'} \binom{m'}{k'} \text{ with } k' = k-1 \text{ and } m' = m-1. \tag{4.11}$$

Finally, $\sum_{k=0}^{n} \binom{n}{k}$ is known to be equal to 2^n. Hence, by changing the index k from $k' = 0$ to $k = 1$, we can write:

$$|P''| = \sum_{k'=0}^{m'} \binom{m'}{k'} = \sum_{k=1}^{m} \binom{m-1}{k-1} = 2^{m-1}. \tag{4.12}$$

In conclusion, there are 2^{m-1} possible plans that must be evaluated. Due to the linear complexity of $O(m)$ for determining the costs of a plan, the cost-based plan vectorization problem exhibits an exponential best-case overall time complexity of $O\left(2^m\right) = O\left(m \cdot 2^{m-1}\right)$. Hence, Lemma 4.1 holds. □

Lemma 4.2. *The cost-based plan vectorization problem exhibits an exponential time complexity of $O(2^m)$ for the worst-case plan of a set of operators.*

[9] As an alternative to Pascal's Triangle, we could also consider the $m-1$ virtual delimiters between operators. Due to the binary decision for each delimiter to be set or not, we get 2^{m-1} different plans. However, we used Pascal's Triangle in order to be able to determine the number of plans for a given number of execution buckets k.

Proof. The problem of finding all possible plans of m operators that are not connected by any data dependencies is reducible to the known problem of finding all possible partitions of a set with m members, where the Bell's numbers B_m [Bel34a, Bel34b] represent the total number of partitions. Note that similar to this known problem, we exclude the plan with zero operators ($B_0 = B_1 = 1$). Thus, the number of possible plans can be recursively computed by

$$|P''| = B_m = \sum_{j=0}^{m-1} \binom{m-1}{j} \cdot B_j. \tag{4.13}$$

Furthermore, each Bell number is the sum of Stirling numbers of the second kind [Jr.68]. As a result, we are able to determine the number of plans $|P''|_k$ for a given k by

$$\begin{aligned} B_m &= \sum_{k=0}^{m} S(m,k) \text{ with } S(m,k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^m \\ |P''|_k &= \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^m \end{aligned} \tag{4.14}$$

In addition, many asymptotic limits for Bell numbers are known [Lov93]. However, in general, we can state that the Bell numbers grow in $O(2^{cm})$, where c is a constant factor. Due to the linear complexity of $O(m)$ for determining the costs of a plan, the cost-based plan vectorization problem exhibits an exponential worst-case overall time complexity of $O\left(2^m\right) = O\left(m \cdot 2^{cm}\right)$. Hence, Lemma 4.2 holds. □

Now, we can combine the results for the best and the worst case to the general result.

Theorem 4.2. *The cost-based plan vectorization problem exhibits an exponential time complexity of $O(2^m)$.*

Proof. The cost-based plan vectorization problem exhibits an exponential time complexity of $O(2^m)$ for both the best-case plan (Lemma 4.1) and the worst-case plan (Lemma 4.2). Hence, Theorem 4.2 holds. □

4.3.2 Computation Approach

So far, we have analyzed the search space of the P-CPV. We now explain how the optimal plan is computed with regard to the current execution statistics. In detail, we present an exhaustive computation approach (thus, with exponential time complexity) as well as a heuristic with linear time complexity that is used within our general cost-based optimization framework. For simplicity of presentation, we use the sequence of operators. However, the general version of the exhaustive and heuristic computation approaches use recursive algorithms that contain many specific cases for arbitrary combinations of subplans (sequences and sets) as well as cases for complex operators.

Exhaustive Computation Approach

The exhaustive computation approach has the following three steps:

1. Scheme Enumeration: Enumerate all 2^{m-1} possible plan distribution schemes for the sequence of operators.

2. Schema Evaluation: Evaluate all distribution schemes according to the optimization objective ϕ or ϕ_c.
3. Plan Rewriting: Rewrite the plan according to the distribution scheme.

In the following, we briefly describe each of those steps with more technical depth.

1: Scheme Enumeration: In order to enumerate all possible distribution schemes of a plan P with m operators, we recursively use Algorithm 4.2. As a first step, we create a `MEMO` table with m columns. In a second step, for each $k \in [1, m]$, we create a record of length k and invoke the recursive A-EDS. Conceptually, this algorithm varies the number of operators of bucket 1 (line 5) and recursively invokes itself in order to distribute the remaining operators across buckets 2 to k. It then varies the number of operators of bucket 2 and so on. Finally, if the remaining operators should be distributed across the last bucket, we insert the tuple into the `MEMO` structure but we could also directly evaluate the enumerated scheme. As a result, the `MEMO` structure holds all 2^{m-1} candidate distribution schemes. Note that this approach is used recursively for complex operators and it contains different loop conditions for the case of sets of operators.

Algorithm 4.2 Enumerate Distribution Schemes (A-EDS)

Require: number of operators m, number of buckets k, record r, position pos

```
1: if k = 1 then
2:    r.pos[1] ← m
3:    insert r into MEMO
4: else
5:    for i ← 1 to m − k + 1 do                    // for each operator o_i
6:       r.pos[pos] ← i
7:       A-EDS(m − i, k − 1, r, pos + 1)    // recursively enumerate distribution schemes
```

2: Scheme Evaluation: Having enumerated all candidates, we can now iterate over the `MEMO` structure and evaluate those schemes in order to determine the optimal scheme according to the optimization objectives ϕ or ϕ_c. Recall the problem definition of cost-based vectorization, i.e., the overall performance of vectorized plans depends on the most time-consuming operator. Here, the costs of a bucket are defined as the sum of all operators in that bucket. We then determine the bucket with maximum costs. The overall optimization objective ϕ is to minimize the number of buckets under the condition of lowest possible maximum bucket costs. In general, all 2^{m-1} candidate schemes need to be evaluated. However, we could prune schemes, where (1) we already determined that a bucket exceeds the maximum execution time and (2) the number of buckets exceeds the minimum number of buckets seen so far. These pruning techniques can be realized on-the-fly during scheme enumeration or with skip-list structures as known from other research areas such as join enumeration [HKL+08] or time series analysis [GZ08].

3: Plan Rewriting: Finally, we use the optimal scheme in order to rewrite the given plan P. For that, the A-PV can be reused with minor changes. Here, we do not create an execution bucket for each operator but we consider the computed k. All operators of one bucket can be copied as a subplan, while data dependencies across execution buckets are replaced by queues. The general model of execution buckets (see Subsection 4.2) is reused as it is with the difference of arbitrary subplans instead of single operators.

Similar to the analysis of Section 4.2, the rewriting algorithm still has a worst-case complexity of $O(m^3)$ for the case of $k = m$. Furthermore, the evaluation of a single distribution

scheme has a linear time complexity of $O(m)$. As a result, the overall complexity of the exhaustive computation is still dominated by the enumeration of candidate distribution schemes, and hence, it has an exponential time complexity of $O(2^m)$.

Heuristic Computation Approach

Due to this exponential complexity of the P-CPV, a search space reduction approach for determining the (near) cost-optimal solution for the P-CPV is required. Therefore, we present a heuristic algorithm that solves the P-CPV and the Constrained P-CPV with linear complexity of $O(m)$. The core idea is to use a first fit (next fit) approach of merging operators into execution buckets until the maximum constraint is reached.

Algorithm 4.3 Cost-Based Plan Vectorization (A-CPV)

Require: operator sequence o
1: $A \leftarrow \emptyset$, $B \leftarrow \emptyset$, $k \leftarrow 0$
2: $max \leftarrow \max_{i=1}^{m} W(P', o_i) + \lambda$
3: **for** $i \leftarrow 1$ **to** $|o|$ **do** *// for each operator o_i*
4: **if** $o_i \in A$ **then**
5: **continue** 3
6: $k \leftarrow k + 1$
7: $b_k(o_i) \leftarrow$ create bucket over o_i
8: **for** $j \leftarrow i + 1$ **to** $|o|$ **do** *// for each following operator o_j*
9: **if** $\left(\sum_{c=1}^{|b_k|} W(o_c) + W(o_j)\right) \leq max$ **then**
10: $b_k \leftarrow$ add o_j to b_k
11: $A \leftarrow A \cup o_j$
12: **else**
13: **break** 9
14: $B \leftarrow B \cup b_k$
15: **return** B

Algorithm 4.3 illustrates the concept of the cost-based plan vectorization algorithm. The operator sequence o is required. First, we initialize two sets A and B as empty sets. Thereafter, we compute the maximal costs of a bucket max with $max = \max_{i=1}^{m} W(o_i) + \lambda$ followed by the main loop over all operators. If the operator o_i belongs to A (operators already assigned to buckets), we can proceed with the next operator. Otherwise, we create a new bucket b_k and increment the number of buckets k accordingly. After that, we execute the inner loop in order to assign operators to this bucket such that the constraint $\sum_{c=1}^{|b_k|} W(o_c) \leq max$ holds. This is done by adding o_j to b_k and to A. Here, we can ensure that each created bucket has at least one operator assigned. Finally, each new bucket b_k is added to the set of buckets B.

The heuristic character is reasoned by merging subsequent operators. This is similar to the first-fit (next fit) algorithm [Joh74] of the bin packing problem but with the difference that the order of operators must be preserved. Thus, there are cases, where we do not find the optimal scheme. However, this algorithm often leads to good or near-optimal results. In conclusion, we use this heuristic as default computation approach, which motivates a more detailed complexity and cost analysis, which we discuss in the following.

Theorem 4.3. *The cost-based plan vectorization algorithm solves (1) the P-CPV and (2) the constrained P-CPV with linear time complexity of $O(m)$. There, the cost constraints hold but the number of execution buckets might not be minimized.*

Proof. Assume a plan that comprises a sequence of m operators. First, the maximum of a value list (line 2) is known to exhibit a linear time complexity of $O(m)$. Second, we see that the bucket number is at least 1 (all operators assigned to one bucket) and at most m (each operator assigned to exactly one bucket). Third, in both cases of $k = 1$, and $k = m$, there are at most $2m - 1$ possible operator evaluations. If we assume constant time complexity for all set operations, we can now conclude that the cost-based plan vectorization algorithm exhibits a linear complexity with $O(m) = O(3m - 1)$. However, due to the importance of the concrete order of operator evaluations, we might require a higher number of execution buckets k than optimal. Hence, Theorem 4.3 holds. □

We use an example to illustrate this heuristic cost-based plan vectorization algorithm and the influence of the λ parameter regarding the constrained optimization objective ϕ_c.

Example 4.7 (Heuristic Cost-Based Plan Vectorization)**.** *Assume a plan with $m = 6$ operators shown in Figure 4.13. Each operator o_i has assigned execution times $W(o_i)$. The maximum operator execution time is given by $W(o_{max}) = \max_{i=1}^{m} W(o_i) = W(o_3) = 5\,\text{ms}$.*

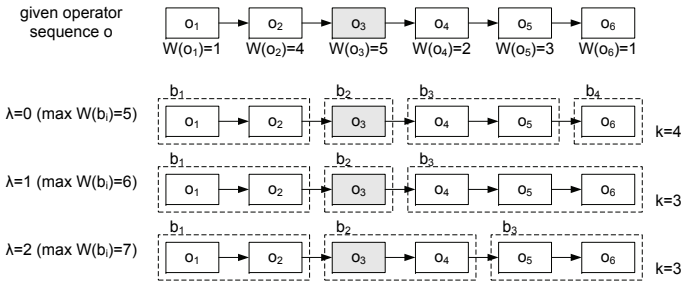


Figure 4.13: Bucket Merging with Different λ

The Constrained P-CPV describes the search for the minimal number of execution buckets, where the cumulative costs of each bucket must not be larger than the determined maximum plus a user-defined cost increase λ. Hence, we search for those k buckets whose cumulative costs of each bucket are, at most, equal to five. If we increase λ, we can reduce the number of buckets by increasing the allowed maximum and hence, the work cycle of the vectorized plan. This example also shows the heuristic character of the algorithm. For the case of $\lambda = 2\,\text{ms}$, we find a scheme with $(k = 3, W(b_1) = 5\,\text{ms}, W(b_2) = 7\,\text{ms}, W(b_3) = 4\,\text{ms})$, while our exhaustive approach would find the more balanced scheme $(k = 3, W(b_1) = 5\,\text{ms}, W(b_2) = 5\,\text{ms}, W(b_3) = 6\,\text{ms})$.

In conclusion, this heuristic approach ensures the maximum benefit of pipeline parallelism and often minimizes the number of execution buckets and hence, reduces the number of threads as well as the length of the pipeline at the same time. Cammert et al. introduced a similar optimization objective in terms of a stall-avoiding partitioning of continuous queries [CHK+07] in the context of data stream management systems. In contrast to their approach, our algorithm is tailor-made for integration flows with control flow semantics and materialized intermediate results within an execution bucket. In addition, (1) we presented exhaustive and heuristic computation approaches as well as

the related complexity analysis, (2) we introduced the parameter λ in order to adjust the computation approaches, and (3) we integrated this heuristic algorithm into our overall cost-based optimization framework.

Optimality Analysis

As already mentioned, the optimality of the vectorized plan depends on (1) the costs of the single operators, (2) the CPU utilization of each operator and (3) the available resources (possible parallelism). However, the A-CPV only takes the costs from (1) into consideration. Nevertheless, we can give optimality guarantees for this heuristic approach.

The algorithm can be parameterized with respect to the hardware resources (3). If we want to force the single-threaded execution, we simply set λ to $\lambda \geq \sum_{i=1}^{m} W(o_i) - \max_{i=1}^{m} W(o_i)$. If we want to force the highest meaningful degree of parallelism (this is not necessarily a full vectorization), we simply set $\lambda = 0$.

Now, assuming the given λ configuration, the question is, which optimality guarantee we can give for the solution of the cost-based plan vectorization algorithm[10]. For this purpose, $R_e(o_i)$ denotes the empirical CPU utilization (measured with a specific configuration) of an operator o_i with $0 \leq R_e(o_i) \leq 1$, and $R_o(o_i)$ denotes the maximal resource consumption of an operator o_i with $0 \leq R_o(o_i) \leq 1$. Here, $R_o(o_i) = 1$ means that the operator o_i exhibits an average CPU utilization of 100 percent. In fact, the condition $\sum_{i=1}^{m} R_e(o_i) \leq 1$ must hold.

Obviously, for an instance-based plan P, we can write $R_e(o_i) = R_o(o_i)$ because all operators are executed in sequence and thus, do not influence each other. When we vectorize P to a fully vectorized plan P', with a maximum of $R_e(o'_i) = 1/m$, we have to compute the costs with $W(o'_i) = R_o(o_i)/R_e(o'_i) \cdot W(o_i)$. When we merge two execution buckets b'_i and b'_{i+1} during cost-based plan vectorization, we compute the effective CPU utilization $R_e(b''_i) = 1/|b|$, the maximal CPU utilization $R_o(b''_i) = (W(b'_i) \cdot R_o(b'_i) + W(b'_{i+1}) \cdot R_o(b'_{i+1}))/(W(b'_i) + W(b'_{i+1}))$, and the cost

$$W(b''_i) = \begin{cases} \dfrac{R_e(b'_i)}{R_e(b''_i)} \cdot W(b'_i) + \dfrac{R_e(b'_{i+1})}{R_e(b''_{i+1})} \cdot W(b'_{i+1}) & R_e(b''_i) \leq R_o(b''_i) \\ \dfrac{R_o(b'_i)}{R_o(b''_i)} \cdot W(b'_i) + \dfrac{R_o(b'_{i+1})}{R_o(b''_{i+1})} \cdot W(b'_{i+1}) & \text{otherwise.} \end{cases} \tag{4.15}$$

We made the assumption that each execution bucket gets the same maximal empirical CPU utilization $R_e(b''_i)$, that resources are not exchanged between those buckets, and we do not take the temporal overlap into consideration. However, we can give the following guarantee, while optimality cannot be ensured due to the heuristic character of the A-CPV (see Theorem 4.3).

Theorem 4.4. *The A-CPV solves (1) the P-CPV, and (2) the constrained P-CPV with guarantees of $W(P'') \leq W(P)$ and $W(P'') \leq W(P')$ under the restriction of $\lambda = 0$.*

Proof. As a precondition, it is important to note that, for the case of $\lambda = 0$, the A-CPV cannot result in a plan with $k = 1$ (although this is a special case of the P-CPV) due to the maximum rule of $\sum_{c=1}^{|b_k|} W(o_c) + W(o_j) \leq max$ (Algorithm 4.3, line 10). Hence, in

[10] For this optimality analysis, we use a slightly different notation of operators o_i and execution buckets b_i in order to clearly distinguish the three different execution models. Let o_i denote instance-based execution, o'_i denote full vectorized execution, and o''_i denote cost-based vectorized execution.

order to prove the theorem, we need to prove the two single claims of $W(P'') \leq W(P)$ and $W(P'') \leq W(P')$.
For the proof of $W(P'') \leq W(P)$, assume the worst case, where $\forall o_i : R_o(o_i) = 1$. If we vectorize this to P'', we need to compute the costs by $W(b_i'') = (R_o(b_i''))/(R_e(b_i'')) \cdot W(o_i)$ with $R_e(b_i'') = 1/|b|$. Due to the vectorized execution, $W(P'') = \max_{i=1}^{m} W(b_i'')$, while $W(P) = \sum_{i=1}^{m} W(o_i)$. Hence, we can write $W(P'') = W(P)$ if the condition $\forall o_i : R_o(o_i) = 1$ holds. This is the worst case. For each $R_o(o_i) < 1$, we get $W(P'') < W(P)$.
In order to prove $W(P'') \leq W(P')$, we fix $\lambda = 0$. If we merge two buckets b_i and b_{i+1}, we see that $R_e(b_i'')$ is increased from $1/|b|$ to $1/(|b|-1)$. Thus, we re-compute the costs $W(b_i'')$ as mentioned before. In the worst case, $W(b_i'') = W(b_i')$, which is true iff $R_e(b_i') = R_o(b_i')$ because then we also have $R_e(b_i'') = R_e(b_i')$. Due to $W(P'') = \max_{i=1}^{m} W(b_i'')$, we can state $W(P'') \leq W(P)$. Hence, Theorem 4.4 holds. □

In conclusion, we cannot guarantee that the result of the A-CPV is the global optimum because we cannot efficiently evaluate the effective resource consumption. However, we can guarantee that each merging of execution buckets when solving the P-CPV with $\lambda = 0$ (where the costs of each bucket are lower than or equal to the highest operator costs) improves the performance of the plan P.

4.3.3 Cost-Based Vectorization with Restricted Number of Buckets

Due to dynamically changing workload characteristics, we recommend using the cost-based vectorization approach. However, there might exist scenarios where an explicit restriction of k and thus, of the number of threads, is advantageous. Hence, in this subsection, we discuss the necessary changes of the exhaustive and heuristic computation approaches when using this constraint.

Exhaustive Computation Approach

With regard to the exhaustive cost-based computation approach (see Subsection 4.3.2), only minor changes are required when restricting k. Due to the restricted number of execution buckets, $k = |b|$, the search space is smaller than for the previously described P-CPV. As already stated, for an operator sequence (best case), there are

$$|P''|_k = \prod_{i=1}^{k-1} \frac{m-i}{i} \tag{4.16}$$

different possibilities, while for sets of operators, there are

$$|P''|_k = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^m \tag{4.17}$$

possibilities to distribute the m operators of plan P across k buckets. Hence, the enumeration of candidate distribution schemes can be reused by simply invoking the recursive Algorithm 4.2 only once for the given k. In addition, we change the optimality condition for evaluating those candidates to

$$\phi = \min \left(\max_{i=1}^{|b|=k} \left(\sum_{j=1}^{l_{b_i}} W(o_j) \right) \right), \tag{4.18}$$

where we determine the minimum costs only for the given k. Hence, the optimization objective is to keep the maximum costs of execution buckets as low as possible. Due to the explicitly given k, the constrained objective ϕ_c is not applicable. Finally, the rewriting algorithm can be reused as it is.

Heuristic Computation Approach

In contrast to the exhaustive computation, the heuristic approach requires major changes when restricting k because we cannot exploit the maximum capacity of a bucket that would result in any k and thus, would stand in conflict to the fixed k constraint. Hence, we require an alternative heuristic to solve this optimization problem.

The heuristic algorithm (A-RCPV) for a restricted number of execution buckets k works as follows. In a first step, we distribute the m operators uniformly across the given k buckets, where the first $m - k \cdot \lfloor m/k \rfloor$ buckets get $\lceil m/k \rceil$ operators and all other operators get $\lfloor m/k \rfloor$ operators assigned to them. In a second step, for each bucket, we check if the performance can be improved by assigning its first operator to the previous bucket or its last operator to the next bucket. There, the optimization objective ϕ (Equation 4.18) is used to determine the influence in the sense of lower maximum bucket costs. Finally, we do this for each operator until no more operators are exchanged during one run over all operators. Due to the direct evaluation with ϕ, cycles are impossible and hence, the algorithm terminates and it exhibits a linear time complexity of $O(m)$. We illustrate this using an example.

Example 4.8 (Heuristic Computation with Fixed k)**.** *Assume a fixed number of buckets, $k = 3$. Figure 4.14 uses the plan and statistics from Example 4.7 and illustrates the heuristic approach for fixed k.*

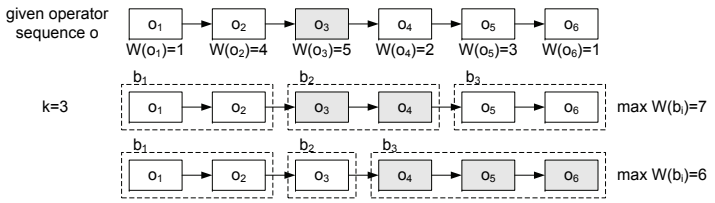


Figure 4.14: Heuristic Operator Distribution with Fixed k

We distribute the six operators uniformly across the three execution buckets. As already mentioned, the performance of the plan depends on the most time-consuming bucket. In our example, this is bucket 2, with $W(b_2) = 7\,\text{ms}$. Now, we exchange operators. First, at bucket 1, no operator is exchanged because transferring o_2 from b_1 to b_2 would increase the maximum bucket costs. The same is true for a transfer of o_3 from b_2 to b_1. However, we can transfer o_4 from b_2 to b_3 and reduce the maximum costs to $W(b_3) = 6\,\text{ms}$. Finally, we require a final run over all buckets to check the termination condition.

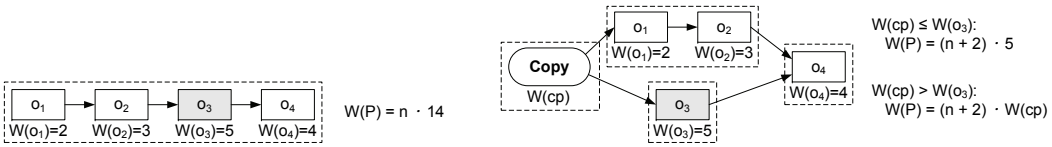
As a result, one can solve both the P-CPV as well as the constrained P-CPV under the restriction of a fixed number of execution buckets and thus, also with a fixed number of threads. With this approach we can guarantee to overcome the problem of a possibly large number of required threads.

4.3.4 Operator-Aware Cost-Based Vectorization

Although the cost-based vectorization described so far significantly improves performance, it has one drawback. When rewriting an instance-based plan into a cost-based vectorized plan, we take only the costs of originally existing operators into account. Thus, the optimization objective is to minimize the number of execution buckets with lowest possible maximum bucket execution costs. However, we neglected the overhead of additional operators such as `Copy`, `And` or `Xor` that are only used within vectorized plans. In conclusion, an operator-aware rewriting approach is required in order to improve the standard cost-based vectorization approach.

In detail, we use *explicit cost comparisons* for operators that are only used for vectorized plans. With this concept, we can ensure that the performance is not hurt by costs of those additional operators. We explain this using the `Copy` operator as an example.

Example 4.9 (Operator-Aware Cost Comparison)**.** *Assume the instance-based subplan illustrated in Figure 4.15(a) and the given operator costs.*



(a) Instance-Based Subplan (b) Cost-Based Vectorized Subplan

Figure 4.15: Example Operator Awareness

The costs of processing n messages are then determined by $W(P) = n \cdot 14\,\text{ms}$*. In contrast, the cost-based vectorized subplan that is shown in Figure 4.15(b) uses* $k = 3 + 1$ *execution buckets and hence, it usually increases the throughput. For the exact cost analysis, we need to distinguish two cases. First, if the costs of the* `Copy` *operator* $W(cp)$ *are lower than the maximum operator costs* $W(o_3)$*, we compute the total costs by* $W(P) = (n + 2) \cdot 5\,\text{ms}$*. Second, if* $W(cp) > W(o_3)$*, we need to compute the costs by* $W(P) = (n + 2) \cdot W(cp)$*. While we always benefit from vectorization in the first case, we have a break-even point for the vectorization benefit in the second case. In detail, if* $n \to \infty$ *and* $W(cp) > 10\,\text{ms}$ *(the costs for executing this subplan in an instance-based manner), it is advantageous to execute the subplan* (o_1, o_2, o_3) *as a single execution bucket because the costs of the* `Copy` *operator are not amortized by the vectorization benefit.*

We use those explicit cost comparisons (optimality conditions) between costs of additional operators and the instance-based execution of such a subplan whenever we determine parallel pipelines. Therefore, only the subplan—from the beginning of those parallel pipelines to the temporal join at the end—is used for comparison.

In conclusion, the operator-aware cost-based vectorization is used within both the exact and the heuristic computation approach. We use explicit cost comparisons in case of subplans consisting of parallel pipelines because those require additional operators, which we can now take into account as well. If statistics are available, this is a binary decision for each subplan and hence, it can be efficiently computed. As a result, this approach adds awareness of specific cases, where vectorization should not be applied for a subplan. This ensures robustness of the cost-based vectorization of a single plan, in the sense that it will never be slower than the instance-based execution.

4.4 Cost-Based Vectorization for Multiple Plans

So far, we have described how to compute the cost-optimal vectorized plan for a *single* deployed plan. While this approach significantly improves the performance of this plan, the cost-based vectorization can also hurt the overall performance in case of *multiple* deployed plans (independent) due to a possibly high number of execution buckets regarding all plans. In conclusion, it is not appropriate to simply use the cost-based vectorization approach or the fixed number of execution buckets for the set of all plans as well. In this section, we present an approach that takes into account executions statistics of all deployed plans. As a result, this approach achieves robustness in terms of the overall performance and hence, allows for more predictable performance of the integration platform.

4.4.1 Problem Description

In many real-world scenarios, multiple independent plans $P_i \in \{P_1, \ldots, P_h\}$ are deployed within an integration platform that executes instances of these plans concurrently. Cost-based vectorization overcomes the problems of full vectorization, i.e., the number of required threads and the work-cycle domination by single operators with regard to a single deployed plan. When executing multiple cost-based vectorized plans concurrently, a similar problem arises. Here, the number of threads required by all h plans depends on the number of plans. In detail, it is upper-bounded by $\sum_{i=1}^{h} m_i$, where m_i denotes the number of operators of plan P_i.

In order to overcome this problem in case of a high number of deployed plans, we define an extended vectorization problem. The core idea is to restrict the maximum number of threads by K in the sense of a user-defined parameter. Then, we compute the fairest distribution of all operators of the h plans across the K execution buckets according to the current workload characteristics and execution statistics. First of all, we formally define the extended cost-based vectorization problem for multiple plans.

Definition 4.4 (Cost-Based Multiple Plan Vectorization Problem (P-MPV))**.** *Let P with $P_i \in \{P_1, \ldots, P_h\}$ denote a set of h plans. The P-MPV then describes the problem of finding a restricted cost-optimal plan P_i'' with k_i execution buckets for each $P_i \in P$ according to the P-CPV. There, the constraint of the maximum overall number of execution buckets of $\sum_{i=1}^{h} k_i \leq K$ must hold.*

Obviously, when simply solving the standard cost-based optimization problem for each single plan P_i, we might exceed the maximum number of execution buckets with $\sum_{i=1}^{h} k_i > K$. The following example illustrates this problem.

Example 4.10 (Problem when Solving the P-MPV)**.** *Assume three plans P_a, P_b and P_c with different numbers of operators and monitored costs as shown in Figure 4.16. We set the maximum total number of execution buckets to $K = 7$. Further, the P-CPV is solved for each single plan using the heuristic computation approach. For this example, we observe that we get $\sum_{i=1}^{h} k_i = 9$ execution buckets and hence, we exceed the maximum constraint of $K = 7$.*

As a result, the P-MPV cannot be solved by applying the P-CPV for each single plan. In contrast to restricting k, (see Subsection 4.3.3), the given maximum constraint K is only an upper bound and therefore we have to consider more solution candidates. In addition, we might not fully use the optimization potential if we simply use K/h buckets

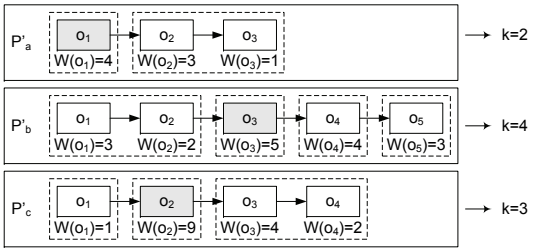


Figure 4.16: Problem of Solving P-CPV for all Plans

for each plan and compute the restricted vectorized plan accordingly. The major challenge of solving the P-MPV is to determine the best distribution of all operators of the h different plans across the maximum number of K execution buckets such that we get the highest overall performance and do not exceed the maximum constraint. In the next subsection, we present a computation approach that addresses this challenge.

4.4.2 Computation Approach

The core idea of computing the optimal operator distribution across buckets for multiple plans is to use the costs of each plan in order to assign more execution buckets to more cost-intensive plans. In detail, we use those costs to weight the maximum constraint K of execution buckets and determine a local maximum constraint K_i for each plan P_i. We use the plan execution time $W(P_i)$ as well as the message arrival rate R_i (that can be monitored as the number of plan instances per time period).

In a first step, we determine the local maximum constraint K_i for each plan. If all h plans would exhibit the same message arrival rate and execution times, we could compute it by $K_i = \lfloor K/h \rfloor$. However, based on the monitored statistics, we determine the constraint by

$$K_i = 1 + \left\lfloor \frac{R_i \cdot W(P_i)}{\sum_{j=1}^{h} R_j \cdot W(P_j)} \cdot (K - h) \right\rfloor, \tag{4.19}$$

where K_i is at least one execution bucket and at most K execution buckets. Due to the lower bound, a constraint of $K < h$ (smaller than the number of plans) is invalid.

In a second step, after we have determined the local maximum constraints, we use a heuristic computation approach to determine the cost-based operator distribution for each plan. If the maximum constraint is not exceeded, we use the computed scheme. If the used number of execution buckets k_i is smaller than the local constraint, in a third step, we further redistribute the $K_i - k_i$ open execution buckets across the remaining plans, where the local constraint is exceeded.

In detail, Algorithm 4.4 illustrates our heuristic approach. First, we determine the local maximum constraints and number of free buckets according to the statistics (lines 2-4). Second, for each plan, we execute the heuristic cost-based plan vectorization (lines 5-10) using the A-CPV. If the determined number of buckets is smaller than or equal to the local maximum constraint, we accept this plan and add the remaining buckets to the free buckets. Third, after all plans have been processed, we redistribute the remaining free buckets according to the monitored statistics of the individual plans (lines 11-12) in an upper-bounded fashion. Fourth, we use the heuristic cost-based plan vectorization but restrict it to the extended maximum constraints (line 13-19) using the A-RCPV. Here, we

Algorithm 4.4 Heuristic Multiple Plan Vectorization (A-HMPV)

Require: plans P, maximum constraint K

1: $P' \leftarrow \emptyset$, $free \leftarrow K$
2: **for** $i \leftarrow 1$ **to** h **do** *// determine local maximum constraints for each plan*
3: $K_i \leftarrow 1 + \left\lfloor \frac{R_i \cdot W(P_i)}{\sum_{j=1}^{h} R_j \cdot W(P_j)} \cdot (K - h) \right\rfloor$ *// $1 \leq K_i \leq K$*
4: $free \leftarrow free - K_i$
5: **for** $i \leftarrow 1$ **to** h **do** *// cost-based plan vectorization for each plan*
6: $P'_i \leftarrow$ `A-CPV` (P_i)
7: **if** $k'_i \leq K_i$ **then**
8: $P' \leftarrow P' \cup P'_i$
9: $P \leftarrow P - P_i$
10: $free \leftarrow free + (K_i - k')$
11: **for** $i \leftarrow 1$ **to** $|P|$ **do** *// distribute remaining threads across remaining plans*
12: $K_i \leftarrow 1 + \left\lceil \frac{R_i \cdot W(P_i)}{\sum_{j=1}^{h} R_j \cdot W(P_j)} \cdot (free - |P|) \right\rceil$
13: **for** $i \leftarrow 1$ **to** $|P|$ **do** *// plan vectorization restricting k' for each remaining plan*
14: **if** $K_i \leq free$ **then**
15: $P'_i \leftarrow$ `A-RCPV` (P_i, K_i) *// restricted cost-based plan vectorization*
16: $free \leftarrow free - K_i$
17: **else**
18: $P'_i \leftarrow$ `A-RCPV` $(P_i, free)$ *// restricted cost-based plan vectorization*
19: $free \leftarrow 0$
20: **return** P'

check that the number of free elements and hence, the global constraint, is not exceeded. We use the following example to illustrate this algorithm.

Example 4.11 (Multiple Plan Vectorization)**.** *Recall Example 4.10 with the maximum constraint of $K = 7$ and assume the following monitored execution statistics:*

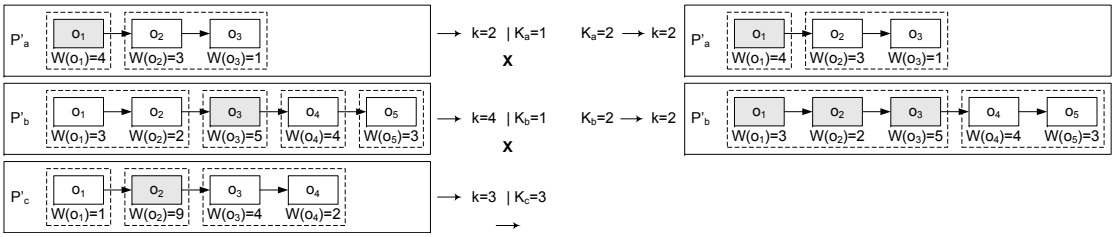
$$\begin{aligned} P_a: &\quad W(P_a) = 8\,\text{ms}, &\quad R_a = 5 &\quad \text{msg/s} \\ P_b: &\quad W(P_b) = 17\,\text{ms}, &\quad R_b = 2.5 &\quad \text{msg/s} \\ P_c: &\quad W(P_c) = 16\,\text{ms}, &\quad R_c = 7 &\quad \text{msg/s}. \end{aligned}$$

In a first step, we compute the local maximum constraint for the individual plans:

$$P_a : K_a = 1 + \left\lfloor \frac{40}{194.5} \cdot 4 \right\rfloor = 1, \;\; P_b : K_b = 1 + \left\lfloor \frac{42.5}{194.5} \cdot 4 \right\rfloor = 1, \;\; P_c : K_c = 1 + \left\lfloor \frac{112}{194.5} \cdot 4 \right\rfloor = 3.$$

In a second step, we compute the cost-based vectorized plan of each deployed plan and check that the resulting number of execution buckets k_i does not exceed K_i. Figure 4.17(a) shows the result of this step. In detail, only P_c is accepted. Hence, we reduce the number of free buckets from $free = 7$ to $free = 4$. Furthermore, in a third step, we distribute the remaining buckets with

$$P_a : K_a = 1 + \left\lceil \frac{40}{82.5} \cdot 2 \right\rceil = 2, \qquad P_b : K_b = 1 + \left\lceil \frac{42.5}{82.5} \cdot 2 \right\rceil = 3.$$



(a) Cost-Based Plan Vectorization (b) Restricted Cost-Based Plan Vectorization

Figure 4.17: Heuristic Multiple Plan Vectorization

Finally, we execute the restricted cost-based plan vectorization with K_i as a parameter. Figure 4.17(b) shows the result of this step, where $K_a = 2$ was used, but $K_b = 2$ because for P_b, the local constraint exceeded the number of free buckets. As a result, we ensured that the global maximum constraint was not exceeded.

This is a heuristic computation because (1) we directly assign free buckets (independent of relative cost improvements) and (2) the order of considered plans might influence the resulting distribution. However, it often solves this problem adequately. An exact computation approach would use a while loop and redistribute free buckets as long as no more changes are made. Just after this, we would restrict k explicitly. However, an exact computation approach would require using the exhaustive cost-based computation.

Our heuristic algorithm has a time complexity of $O(h \cdot m)$ because we call h times the A-CPV that has a time complexity of $O(m)$ (see Theorem 4.3). Furthermore, we might call at most h times the heuristic restricted plan vectorization algorithm that has also a linear time complexity of $O(m)$.

In conclusion, the multiple plan vectorization takes into account the workload characteristics in order to restrict the maximum number of used execution buckets. There, plans with high load are preferred and get more execution buckets assigned to them. At the same time, this algorithm applies the cost-based vectorization and hence, can ensure near-optimal performance even in the case of large numbers of plans.

4.5 Periodical Re-Optimization

We have shown how to compute cost-based vectorized plans in case of both single and multiple deployed plans. Those exhaustive and heuristic algorithms rely on continuous gathering of execution statistic and periodical re-optimization in order to be aware of changing workload characteristics. Finally, the whole vectorization approach is embedded as a specific optimization technique into our general cost-based optimization framework. Note that our transformation-based optimization algorithm (A-PMO, Subsection 3.3.1) applies the cost-based vectorization after all other rewriting techniques, where techniques for rewriting patterns to parallel flows are not used if vectorization is enabled.

With regard to the whole feedback loop of our general cost-based optimization framework, there are two additional challenges that need to be addressed for vectorized plans. First, if vectorization produces a plan that differs from the currently deployed plan, we have to evaluate the re-optimization potential in the sense of the benefit of exchanging plans. Second, if an exchange is beneficial or one of the other optimization techniques

changed the current plan, there is a need for dynamic rewriting of vectorized plans due to the required state migration of loaded message queues and intra-operator states. Both challenges are addressed in the following.

Evaluating Re-Optimization Potential

If the current logical plan has been changed during optimization, we need to evaluate the benefit of rewriting the physical plan during runtime. This is required because a vectorized plan exhibits a state in the sense of all messages that are currently within the standing process (operators and loaded queues) and thus, we cannot simply generate a new physical plan. Hence, there is a trade-off between the overhead of exchanging the plan and the benefit yielded by the newly computed best plan. In general, the same is true for the general optimization framework as well. However, in contrast to the efficient inter-instance plan change, this trade-off has much higher importance when rewriting vectorized plans due to the need for state migration or flushing of pipelines.

The intuition behind our evaluation approach is to compare the costs of flushing the pipelines of the current plan, with the estimated benefit we gain by using P''_{new} instead of P''_{cur} for the next period Δt. We restrict the cost comparison to Δt because at the next evaluation timestamp, we might revert the plan change due to changed execution statistics and hence, we cannot estimate the benefit for a period longer than Δt. Although we might miss optimization opportunities in case of a constant workload, we use this approach in order to ensure robustness of optimizer choices in terms of plan stability.

In detail, the costs of flushing the pipelines are affected by the number of messages in the queues q_i and the execution time of the most time-consuming operator. We determine the queue cardinalities and compute the costs by

$$W_{flush}(P''_{cur}) = W(b_x) \cdot \sum_{i=1}^{x} |q_i| + \sum_{i=x}^{|b|} W(b_i) \quad \text{with} \quad W(b_x) = \max_{j=1}^{k} \sum_{l=1}^{l_{b_i}} W(o_l). \tag{4.20}$$

These costs are given by the number of messages in front of the most time-consuming execution bucket multiplied by the costs of this bucket $W(b_x)$ plus the costs for the remaining buckets after b_x. Those are the approximated costs for flushing the whole pipeline. Note that incremental rewriting (merging and splitting) is possible as well.

For computing the benefit of dynamic rewriting, we use the message rate R to compute the number of processed messages by $n = R \cdot \Delta t$. Then, the benefit of exchanging plans is given by

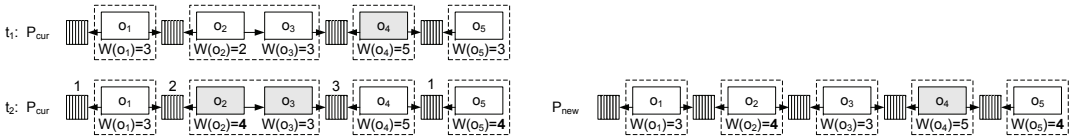
$$W_{change}(P'') = (n + |b|_{P''_{new}} - 1) \cdot W_{P''_{new}}(b_{x1}) - (n + |b|_{P''_{cur}} - 1) \cdot W_{P''_{cur}}(b_{x2}), \tag{4.21}$$

where $W_{change} < 0$ holds by definition because the optimizer will only return a new plan P_{new} if $W(P_{new}) < W(P_{cur})$. Finally, we would change plans if

$$W_{flush} + W_{change} \leq 0. \tag{4.22}$$

We illustrate this evaluation approach with the following example.

Example 4.12 (Evaluating Rewriting Benefit)**.** *Assume the current plan shown in Figure 4.18(a). The figure also shows the statistics that were present when creating this plan (t_1) as well as the current state (t_2) in the form of the numbers of messages in queues. During the period $\Delta t = 10\,\text{s}$, average execution times changed ($W(o_2) = 4$ and $W(o_5) = 4$).*



(a) Deployed Plan P_{cur} (b) New Plan P_{new}

Figure 4.18: Example Periodical Re-Optimization

Hence, we created the new plan shown in Figure 4.18(b). First, we determine the costs for flushing the current pipeline, assuming the new statistics with

$$W_{flush}(P''_{cur}) = 3 \cdot W(b_2) + 5\,\text{ms} + 4\,\text{ms} = 30\,\text{ms}.$$

Then, we compute the benefit of changing the plan by

$$W_{change} = (n + 5 - 1) \cdot 5\,\text{ms} - (n + 4 - 1) \cdot (4\,\text{ms} + 3\,\text{ms}) = -2\,\text{ms} \cdot n - 1\,\text{ms}.$$

Subsequently, we use the monitored message rate $R = 10.7\,\text{msg/s}$ *and the optimization period* $\Delta t = 10\,\text{s}$ *as estimation for the number of processed messages* $n = 10.7\,\text{msg/s} \cdot 10\,\text{s} = 107$ *during the next period and compare the costs with the benefit, by assuming full system utilization, as follows:*

$$(W_{flush} + W_{change} = 30\,\text{ms} + (-2\,\text{ms} \cdot 107 - 1\,\text{ms})) \leq 0.$$

Finally, we decide to exchange plans because in the next period Δt*, we will yield an improvement of* 185 ms*, including the overhead for rewriting.*

If the evaluation of the rewriting benefit resulted in the decision to exchange plans, we need to dynamically rewrite the existing plan during runtime. In the following, we explain this step in more detail.

Dynamic Plan Rewriting

The major problem when rewriting a vectorized plan during runtime is posed by loaded queues. One approach would be explicit state migration and state re-computation [ZRH04]. However, re-computation might is impossible for integration flows due to interactions with external systems that have to be executed exactly once. Therefore, plan rewriting is realized by stopping execution buckets and flushing of intermediate queues.

For example, in order to merge two execution buckets b_i and b_{i+1} with a queue q_{i+1} in between, we need to stop the execution bucket b_i, while bucket b_{i+1} is still working. Over time, we flush q_{i+1} and wait until it contains zero messages. We then merge the execution buckets to b_i, which contains an instance-based subplan with all operators of the merged subplans, and simply remove q_{i+1}. This concept can be used for bucket merging and splitting, respectively and we never loose a message during dynamic plan rewriting.

Putting it all together, we introduced the general concept of vectorization as a control-flow-oriented optimization technique that aims to improve the message throughput. Furthermore, we generalized this concept to the cost-based plan vectorization and explained how to take multiple deployed plans into account as well. Finally, we described how this technique is embedded into our general cost-based optimization framework. Although, this

approach was designed for multi- and many-core systems, in general, it can be extended to the distributed case as well. There, extensions with regard to the communication costs between several nodes as well as heterogeneous hardware (different execution times on different server nodes) would be required. However, in this distributed setting, the *cost-based* vectorization would be even more important because the number of involved server nodes could be reduced without sacrificing the degree of parallelism.

4.6 Experimental Evaluation

In this section, we provide experimental evaluation results for both the full vectorization and the cost-based vectorization of integration flows. The major perspectives of this evaluation are (1) the performance in terms of message throughput, (2) the influence on latency times of single messages, as well as (3) the optimization overhead and influences of parameterization. In general, the evaluation shows that:

- Significant performance improvements in the sense of increased message throughput are achievable. The benefit of vectorization increases with increasing number of operators, with increasing data size, and with increasing number of plan instances.
- The latency of single messages is moderately increased by vectorization. However, due to Little's Law [Lit61], in case of high load of messages, the total latency time (including waiting time) is reduced by vectorization.
- The deployment and optimization overhead imposed by vectorization is moderate as well. In addition to the influence of the parameters of periodical optimization, also the cost-constraint-parameter λ has high influence on the resulting performance. Typically, the default setting of $\lambda = 0$ leads to highest performance.
- The cost-based optimization typically finds the global optimal plan according to the number of execution buckets. Thus, the number of buckets should not be restricted.

Finally, we can state that the cost-based vectorization achieves significant throughput improvements, while accepting moderate additional latency for single messages. In conclusion, this concept can be applied by default if a high load of plan instances exists and moderate latency time is acceptable. The theoretical optimality and latency guarantees also hold under experimental performance evaluation.

The evaluation is structured as follows. First, we present the end-to-end comparison of unoptimized and vectorized execution using our running example plans. Second, we use a set of additional template plans in order to evaluate the aspects throughput improvement, latency time, and optimization overhead in more detail on plans with variable number of operators. Third, we present evaluation results with regard to multiple deployed plans.

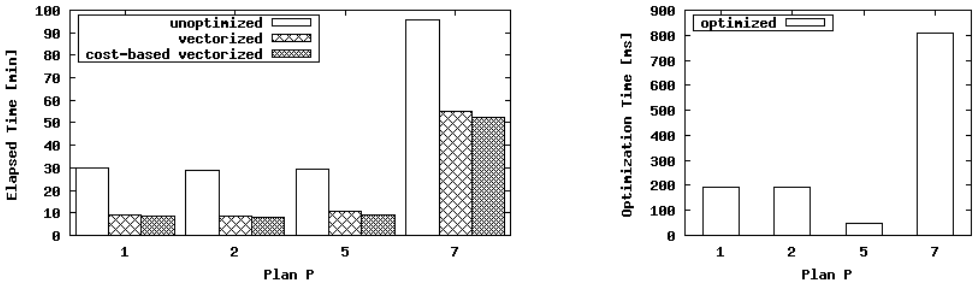
Experimental Setting

We implemented the presented approach within our `WFPE` (workflow process engine). In general, the `WFPE` uses compiled plans and an instance-based execution model. Then, we integrated components for the full vectorization (`VWFPE`) and for the cost-based vectorization (`CBVWFPE`). For this purpose, new deployment functionalities have been introduced and several changes in the runtime environment were required because those plans are executed in an interpreted fashion.

We ran our experiments using the same platform as described in Section 3.6. Further, we executed all experiments on synthetically generated XML data (using our DIPBench toolsuite [BHLW08c]) due to only minor influence of the data distribution of real data sets on the benefit achieved by vectorization because it is a control-flow-oriented optimization technique. However, there are several aspects with influences on vectorization. In general, we used five scale factors for all three execution approaches: the data size d of input messages, the number of operators m, the time interval t between two arriving messages, the number of plan instances n, and the maximum constraint of messages in a queue q.

End-to-End Comparison and Scalability

Similar to the general comparison experiment of optimized and unoptimized plan execution, which results are shown in Figure 3.22, we first evaluated the impact of vectorization and cost-based vectorization compared to the unoptimized execution for our example use case plans. In detail, we executed 20,000 plan instances for all asynchronous, data-driven example plans (P_1, P_2, P_5, and P_7) and for each execution model. We fixed the cardinality of input data sets to $d = 1$ (100 kB messages) and used the same workload configuration (without workload changes and without correlations) as in the mentioned experiment of Chapter 3. Note that the normal cost-based plan rewriting is orthogonal to vectorization, where vectorization achieves additional improvements except for the effects of rewriting patterns to parallel flows. In order to be focused on vectorization, we disable all other optimization techniques. Furthermore, we fixed an optimization interval of $\Delta t = 5$ min, a sliding window size of $\Delta w = 5$ min and `EMA` as the workload aggregation method. To summarize, we consistently observe significant total execution time reductions (see Figure 4.19(a)) of 71% (P_1), 72% (P_2), 69% (P_5), and 55% (P_7). In contrast to Chapter 3, we measured the scenario elapsed time (the latency time of the message sequence) because for vectorized execution, the execution times of single plan instances cannot be aggregated due to overlapping message execution (pipeline semantics).



(a) Scenario Elapsed Time (b) First Optimization Time

Figure 4.19: Use Case Comparison of Vectorization

First, the full vectorization approach leads to a significant reduction of the total elapsed time for execution of the sequence of 20,000 plan instances. We achieved a speedup of factor three for the plans P_1, P_2, and P_5, while for the plan P_7 we achieved a speedup of factor two. Furthermore, the cost-based vectorization further improved the full vectorization by about 10%. However, there are cases, where the cost-based vectorization caused only a minor improvement because plans such as P_1 are too restrictive with regard to merging execution buckets (e.g., the combination of a `Switch` operator with specific paths is not

allowed). For the cost-based vectorization, we used the introduced heuristic computation approach with $\lambda = 0$, which changed the number of buckets (includes additional VMTM operators) as follows: P_1 : from 10 to 8 buckets, P_2 : from 6 to 3 buckets, P_5 : from 9 to 6 buckets, and P_7 : from 18 to 3 buckets. Due to the absence of changing workload characteristics, only the first invocation of the optimizer requires noteable optimization time for merging buckets and flushing pipelines, while all subsequent re-optimization steps take much less optimization time. Figure 4.19(b) shows this optimization time. Clearly, the plan P_7 required the highest optimization time but still takes less than a second, which is negligible compared to the achieved total execution time reduction of over 40 min.

Second, scalability experiments have shown that the absolute improvement increases with increasing data size but the relative improvement with increasing data size depends on the used plan. There are plans with constant relative improvement (e.g., P_1) and plans, where the relative improvement decreases with increasing data size (e.g., P_7). Further, the relative improvement of both vectorization approaches increases with an increasing number of executed plan instances until the point of full pipeline utilization is reached. From thereon, the relative improvement stays constant.

In conclusion, we achieve performance improvements in the form of an increase of message throughput. In addition, we observe that the absolute benefit increases with increasing number of plan instances and with an increasing data size as well.

Performance and Throughput

In order to evaluate both vectorization approaches in more detail, we use a template plan that can be extended to arbitrary numbers of operators. Essentially, we modeled a simple sequence of six operators as shown in Figure 4.20.

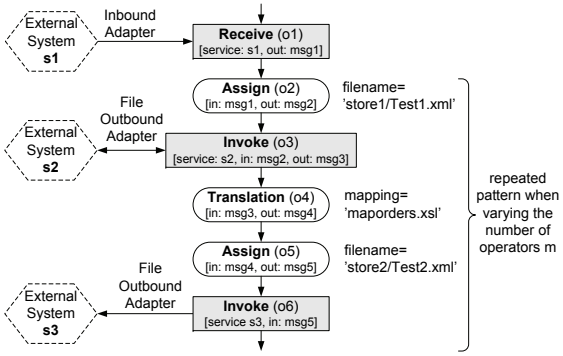
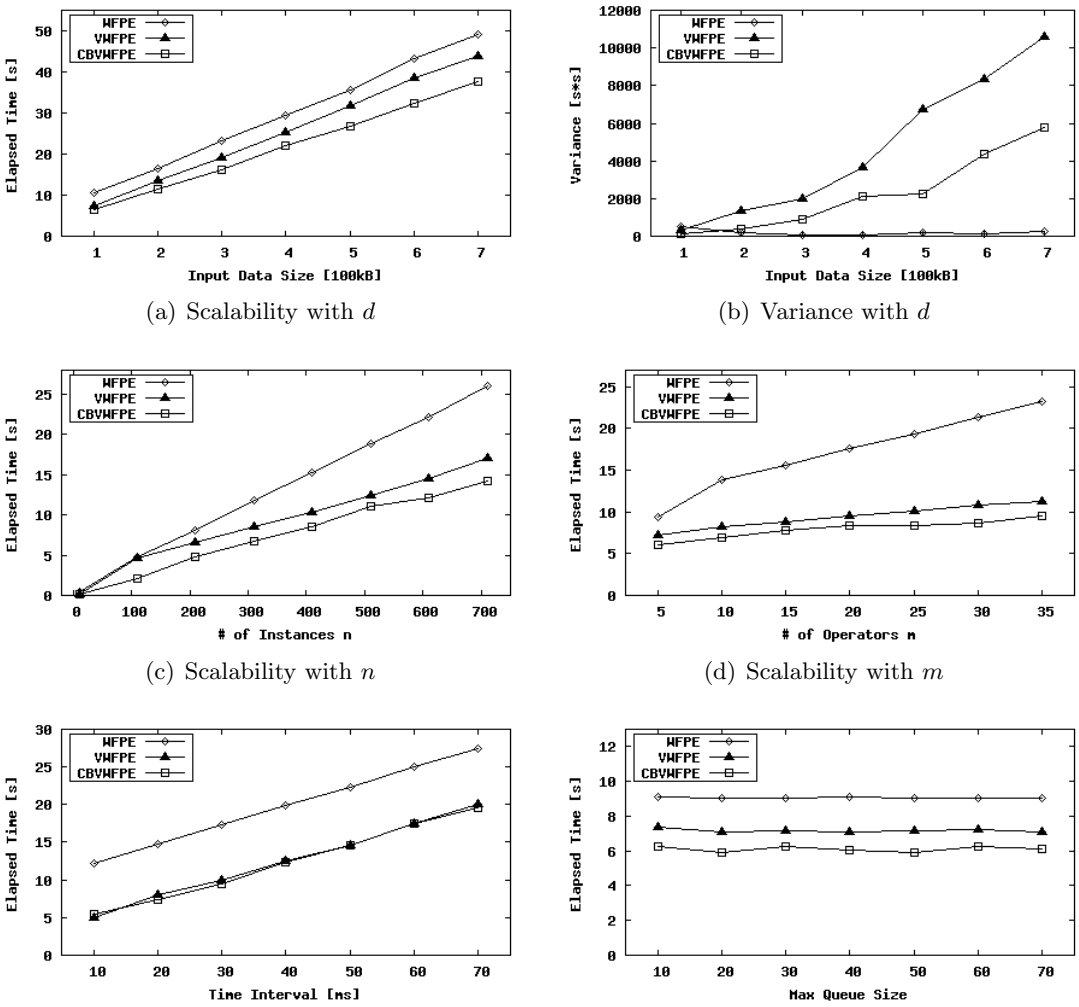


Figure 4.20: Evaluated Example Plan P_m

A message is received (`Receive`), prepared for a writing interaction (`Assign`), which is then executed with the file outbound adapter (`Invoke`). Subsequently, the resulting message (contains Orders and Orderlines) is modified by a `Translation` operator and finally, the message is written to a specific directory (`Assign`, `Invoke`). We refer to this as $m = 5$ because the `Receive` operator is removed during vectorization. When scaling m up to $m = 35$, we simply copy the last five operators and reconfigure them as a chain of m operators with direct data dependencies. All of the resulting `Invoke` operators refer to different directories. We ran a series of five experiments (each repeated 20 times) according to the already introduced scale factors. The results of these experiments are shown in Figure 4.21.



(a) Scalability with d (b) Variance with d

(c) Scalability with n (d) Scalability with m

(e) Scalability with t (f) Scalability with q

Figure 4.21: Scalability Comparison with Different Influencing Factors

In Figure 4.21(a), we scaled the data size d of the XML input messages from 100 kB to 700 kB and measured the execution time (elapsed time) of 250 plan instances ($n = 250$) needed by the three different execution models. There, we fixed $m = 5$, $t = 0$, $n = 250$ and $q = 50$. We observe that all three execution models exhibit a linear scaling according to the data size and that significant improvements can be achieved with vectorization. There, the absolute improvement increases with increasing data size. Further, in Figure 4.21(b), we illustrated the variance over all 20 repetitions of this sub-experiment. The variance of the instance-based execution is minimal, while the variance of both vectorized models is worse due to the unpredictable influence of thread scheduling by the operating system. Cost-based vectorization exhibits a significantly lower variance than full vectorization because we use fewer threads and therefore reduce the thread scheduling influence.

Now, we fix $d = 1$ (lowest improvement in 4.21(a)), $t = 0$, $n = 250$ and $q = 50$ in order to investigate the influence of m. In Figure 4.21(d), we vary m from 5 to 35 operators, as already mentioned for the experimental setup. Interestingly, not only the absolute but also

the relative improvement of vectorization increases with increasing number of operators.

Figure 4.21(e) shows the impact of the time interval t between the initiation of two plan instances. For that, we fixed $d = 1$, $m = 5$, $n = 250$, $q = 50$ and we varied t from 10 ms to 70 ms. There is almost no difference between the full vectorization and the cost-based vectorization. However, the absolute improvement between instance-based and vectorized approaches decreases slightly with increasing t. The explanation is that the time interval has no impact on the instance-based execution. In contrast, the vectorized approach depends on t because the highest improvement is achieved with full pipeline utilization.

Further, we analyze the influence of the number of instances n as illustrated in Figure 4.21(c). Here, we fixed $d = 1$, $m = 5$, $t = 0$, $q = 50$ and we varied n with $n \in \{10, 100, 200, 300, 400, 500, 600, 700\}$. Basically, we observe that the relative improvement between instance-based and vectorized execution increases when increasing n, due to parallelism of plan instances. However, it is interesting to note that the fully vectorized solution performs slightly better for small n. However, when increasing n, the cost-based vectorized approach performs optimal because there the maximum queue constraint q is reached and we observe the influence of the already mentioned convoy effect.

Figure 4.21(f) illustrates the influence of the maximum queue size q, which we varied from 10 to 70. Here, we fixed $d = 1$, $m = 5$, $t = 0$ and $n = 250$. An increasing q slightly decreases the execution time of vectorized plans. This is reasoned by (1) less request notifications of waiting threads and (2) better load balancing by the thread scheduler.

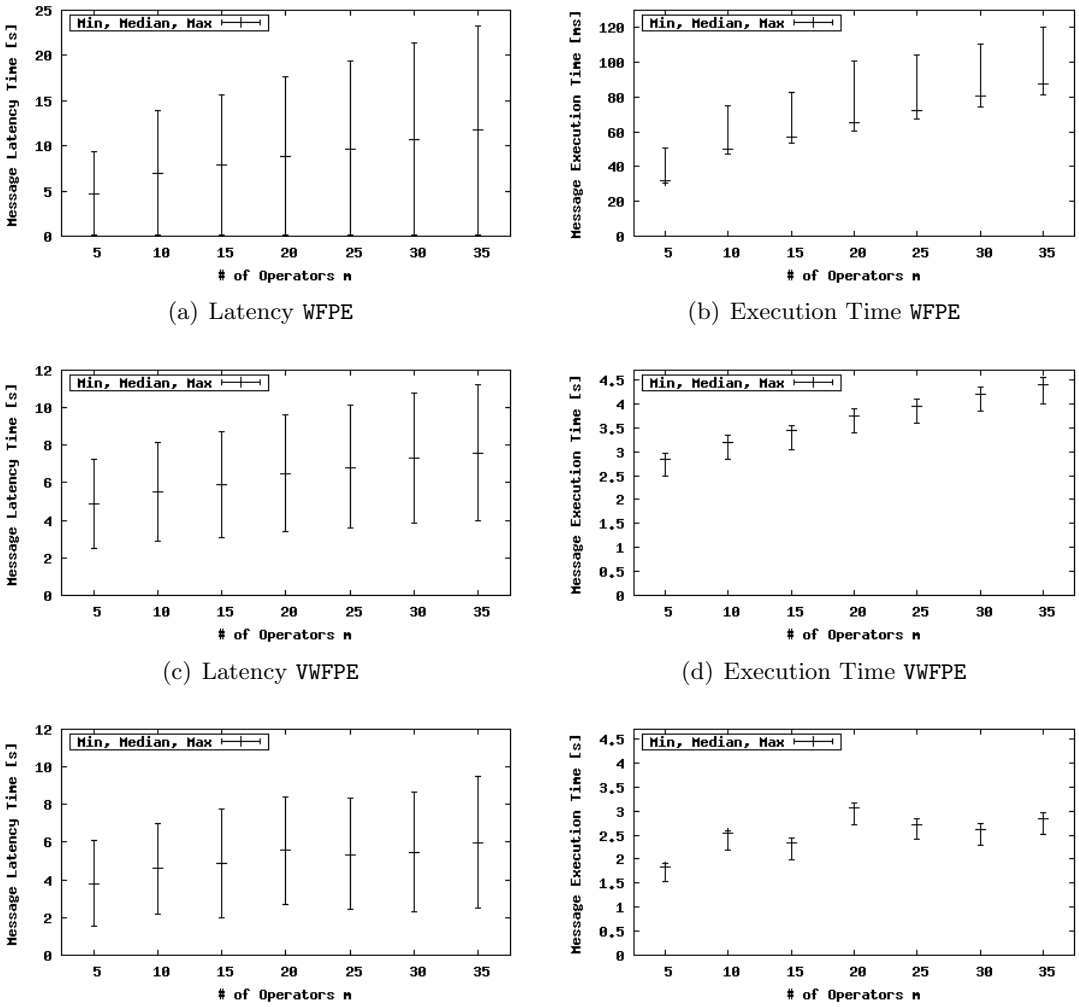
Message Latency

Vectorization is a trade-off between message throughput improvement and increased latency time of single messages. Therefore, we now investigate the latency of single messages.

Figure 4.22 illustrates the differences of the three execution models *instance-based* (`WFPE`), *vectorized* (`VWFPE`) and *cost-based vectorized* (`CBVWFPE`) according to the latency of single messages (including inbound waiting time) and the execution time of single messages (without inbound waiting time). Therefore, we fixed $d = 1$, $t = 0$, $q = 50$, and we varied the number of operators m, similar to Figure 4.21(a). All results are illustrated as error bars using the minimum, median (50% quartile) and maximum latency/execution time, respectively. In this experiment, all $n = 250$ messages arrive simultaneously in the system. The latency time includes the waiting time and execution time in the sense of end-to-end latency. In contrast, the execution time shows how long it takes to process a single message, without waiting time at the server inbound message queues.

First, we observe that the instance-based execution allows for lowest minimum latency (first processed message), while both the vectorized as well as the cost-based vectorized execution requires higher initial time for processing the first messages. This is caused by queue management and synchronization between threads. It is important to note that the cost-based vectorized model exhibit lower initial latencies. Further, we see that the median and maximum latencies are higher for the instance-based execution model because it is directly influenced by the reached throughput.

Second, it is important that the execution time of a single message is much smaller when using the instance-based execution model rather than the vectorized model. The reason is that the vectorized execution is dominated by the most time-consuming operator and requires additional effort for thread queue management and synchronization. In addition, for the used setting of a queue size of $q = 50$, most messages wait just in front of this



(a) Latency WFPE (b) Execution Time WFPE

(c) Latency VWFPE (d) Execution Time VWFPE

(e) Latency CBVWFPE (f) Execution Time CBVWFPE

Figure 4.22: Latency Time and Execution Time of Single Messages (for $n = 250$)

most time-consuming operator. This waiting time at inter-bucket queues is included in the execution time of the whole plan. Note that the cost-based vectorized execution is faster than the full vectorization but with non-linear scaling because the number of execution buckets changed with increasing number of operators. Finally, the median of instance-based execution is close to the minimum, while for vectorized execution, it is close to the maximum. The substantiation is that there are only few messages (the first messages of a sequence) without any waiting time within inter-bucket queues when executing vectorized plans, while for all other messages, execution times include these waiting times.

Deployment and Maintenance

Furthermore, we evaluated the deployment and vectorization overhead with increasing number operators in a static manner (without dynamic plan rewriting).

First, we measured the costs for the plan vectorization algorithm (A-PV) and the periodically invoked cost-based plan vectorization algorithm (A-CPV). Figure 4.23 shows those

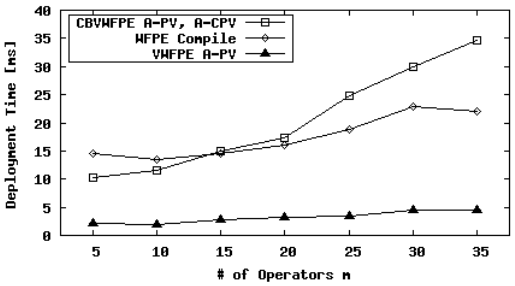


Figure 4.23: Vectorization Deployment Overhead

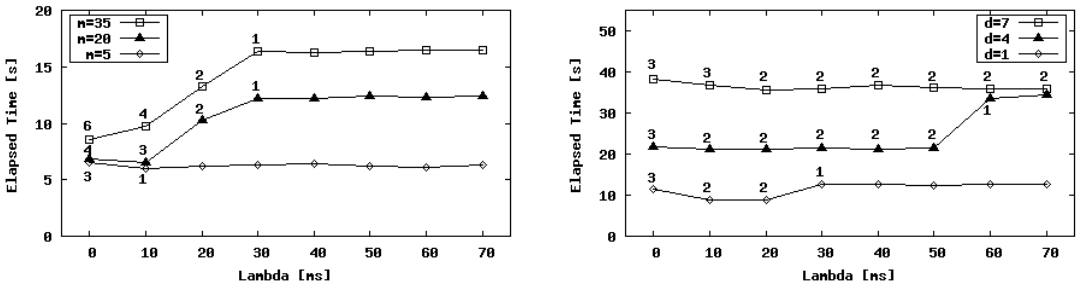
results, where we varied the number of operators m because all other scale factors do not influence the deployment and maintenance costs. In general, there is a high performance improvement using vectorization with a factor of up to seven. It is caused by the different deployment approaches. The `WFPE` uses a compilation approach, where Java classes are generated from the integration flow specification. In contrast to this, the `VWFPE` as well as the `CBVWFPE` uses interpretation approaches, where plans are built dynamically with the A-PV. The `VWFPE` always outperforms `CBVWFPE` because both use the A-PV but `CBVWFPE` additionally uses the A-CPV in order to find the optimal number of execution buckets k. Note that the additional costs for the A-CPV (that cause a break-even point with the standard WFPE) occur periodically during runtime. Here, we excluded the costs for flushing the pipelines because it depends mainly on the maximum constraint of the queues and on the costs of the most time-consuming operator. In conclusion, the vectorization of integration flows shows better runtime as well as often better deployment time performance with regard to plan generation. Even in the case where a deployment overhead exists, it is negligible compared to the runtime improvement we gain by vectorization.

In conclusion, the deployment and maintenance overhead is moderate compared to the yielded performance improvement. Recall the evaluation results from Figure 4.21. It is important to note that the presented performance of the cost-based vectorized execution model already includes the costs for periodical re-optimization and statistics maintenance.

Parameters of Periodic Optimization

The resulting performance improvement of vectorization in the presence of changing workload characteristics depends on the periodic re-optimization. This re-optimization can be influenced by several parameters including the workload aggregation method, the sliding time window size Δw, the optimization period Δt, and the maximum cost increase λ. In this subsection, we evaluate the influence of λ with regard to the cost-based vectorization, while the other parameters have already been evaluated in Chapter 3. Therefore, we conducted an experiment, where we measured how increasing maximum costs influence the number of execution buckets and thus, indirectly influence the elapsed time as well.

In a first sub-experiment, we fixed $d = 1$, $t = 0$, $q = 50$ and executed $n = 250$ messages with different λ and for different plans (with different numbers of operators m). Figure 4.24(a) shows the influence of λ on the number of execution buckets k as well as on the execution time. It is obvious that the number of execution buckets (annotated at the top of each point) decreases with increasing λ because for each bucket, the sum of operator costs must not exceed $max + \lambda$ and hence, more operators can be executed by a single bucket. Clearly, when increasing λ, the number of execution buckets cannot increase. In



(a) Number of Operators m (b) Input Data Size d

Figure 4.24: Influence of λ with Different Numbers of Operators and Data Sizes

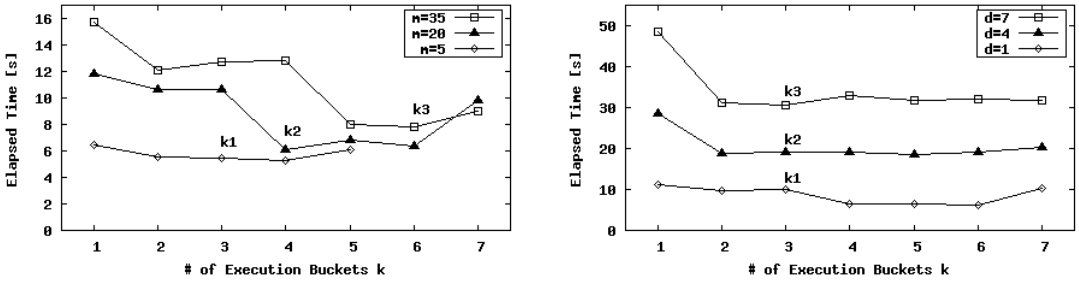
detail, we see the near-optimal solution with $\lambda = 0$. Typically, when increasing the number of execution buckets from this point, the elapsed time increases. However, there are cases such as the plan with $m = 5$, where we observe that the instance-similar execution (with one bucket for all operators) performs better. The difference to traditional instance-based execution is caused by (1) reused operator instances (e.g., pre-parsed XSLT stylesheets of `Translation` operators), and (2) pipelined inbound processing. Further, for $m = 20$, we see the optimal total execution time at $\lambda = 10$. Note that even in this case of $k = 1$, we reach better performance than in the instance-based case because there are no synchronous (blocking) calls through the whole engine but only within the single plan. In addition, for small numbers of processed messages n, the instance-based execution model performs worse due to the overhead of just-in-time compilation of generated plans. Furthermore, we observe that the higher the number of operators m, the higher the influence of the parameter λ. In conclusion, we typically find a very good solution with $\lambda = 0$, but when required, this parameter can be used to easily adjust the degree of parallelism.

Furthermore, in the second sub-experiment, we used a single plan with $m = 20$, we fixed $t = 0$, $q = 50$ and we executed $n = 250$ messages with different λ and for different data sizes $d \in \{1, 4, 7\}$ (in 100 kB). Figure 4.24(b) shows the results with regard to the execution time as well as the number of execution buckets (annotated at the top of each point) when varying λ. In general, we see similar behavior as in Figure 4.24(a) (for $m = 20$). The different numbers of execution buckets for $d = 1$ and $\lambda \in (0, 10)$ are caused by dynamically monitored operator costs, which varied slightly. The major difference when comparing the influence of varying the data size with the previous sub-experiment is that the data size significantly increases the execution time of single operators. As a result, we observe that we require higher values of λ to reduce the number of execution buckets. In conclusion, λ should be configured with context knowledge about current workload characteristics. We could overcome this workload dependency with a relative value of λ according to the maximum operator costs. However, with λ as an absolute value, we can explicitly determine the maximum work-cycle increase of the data flow graph.

In conclusion, there are several parameters with significant influence on the total execution time and on the behavior of cost-based vectorization. As a general heuristic, one should use a maximum costs increase of $\lambda = 0$. This simplest configuration typically results in near-optimal throughput. However, if more context knowledge about the workload is available, the described parameters can be used as tuning knobs.

Plan with Restricted k

In order to reveal the characteristics of vectorizing multiple plans, we further evaluated the influence of restricting the number of execution buckets. This is applied if the cost-based vectorization exceeds this computed maximum number of execution buckets. All other aspects of vectorization for multiple plans is a combination of already presented effects.



(a) Number of Operators m (b) Input Data Size d

Figure 4.25: Restricting k with Different Numbers of Operators and Data Sizes

The first sub-experiment analyzes the influence of the number of execution buckets on the execution time of a message sequence with regard to varying number of plan operators m. We fixed $d = 1$, $t = 0$, $q = 50$ and explicitly varied the number of execution buckets k. Figure 4.25(a) shows the resulting execution time for a message sequence of $n = 250$. We observe that, in a first part, an increasing number of execution buckets leads to decreasing execution time. In a second part, a further increase of the number of execution buckets led to an increasing execution time. As a result, there is an optimal number of execution buckets, which increases depending on the number of operators. We annotated with $k1$, $k2$ and $k3$ the numbers of execution buckets that our A-CPV computed without restricting k. Note that for $m = 5$, at most $k = 5$ execution buckets can be used.

In addition to this, we also analyzed the influence of the number of execution buckets on the execution time with regard to different data sizes. Hence, we used the plan $m = 20$, we fixed $t = 0$, $q = 50$ and we varied the data size $d \in \{1, 4, 7\}$ (in 100 kB). Figure 4.25(b) shows these results. We observe that the first additional execution bucket significantly decrease the execution time, while after that, the execution time varies only slightly for an increasing number of execution buckets. However, there is also an optimal point, where the optimal number of execution buckets decreases with increasing data size due to increased cache displacement. Again, we annotated the resulting number of execution buckets of our A-CPV.

We might use randomized algorithms as heuristics for determining the number of buckets k and for assigning operators to buckets. However, the presented experiments (Figure 4.25(a) and Figure 4.25(a)) show an interesting characteristic that prohibit such randomized heuristics. While our cost-based vectorization approach finds the near-optimal solution, a randomly chosen k might significantly decrease the performance, where the influence of determining the best k increases with increasing number of operators.

In conclusion, the cost-based vectorization typically computes schemes, where k is close to the optimal number of execution buckets with regard to minimal execution time of a message sequence. Thus, we recommend using cost-based vectorization without restricting k. However, for multiple deployed plans it is required to ensure the maximum constraint.

4.7 Summary and Discussion

In this chapter, we introduced the control-flow-oriented optimization technique of plan vectorization with the aim of throughput optimization for integration flows. We use the term vectorization as an analogy to executing a vector of messages at a time by a standing plan. Due to the dependency on the dynamic workload characteristics, we introduced the cost-based plan vectorization as a generalization, where the costs of single operators are taken into account and operators are merged to execution buckets. In detail, we presented exhaustive and heuristic algorithms for computing the cost-optimal plan. Furthermore, we showed how to use those algorithms in the presence of multiple deployed plans and how this concept is embedded into our general cost-based optimization framework.

Based on our evaluation, we can state that significant throughput improvements are possible. In comparison to full vectorization, the cost-based vectorization achieves even better performance, reduces the latency of single messages, and ensures robustness in the sense of minimizing the number of required threads. In conclusion, the concept of plan vectorization is applicable in many different application areas. It is important to note that the benefit of vectorization and hence, also cost-based vectorization, will increase with the ongoing development of modern many-core processors because the gap between CPU performance and main memory, IO, and network speed is increasing (Problem 4.1).

The main differences of our approach to prior work are (1) that we vectorize *procedural* integration flows (imperative flow specifications) and that we (2) *dynamically* compute the *cost-optimal* vectorized plan within our *periodical re-optimization* framework. This enables the dynamic adaptation to changing workload characteristics in terms of the operator execution times. Despite the focus on procedural plans, the cost-based vectorization approach, in general, can also be applied in the context of DSMS and ETL tools.

However, the vectorization approach has also some limitations that must be taken into account when applying this optimization technique. First, vectorization is a trade-off between throughput improvement and additional latency time. Thus, it should only be applied if the optimization objective is throughput improvement or minimizing the latency in the presence of high message rates rather than minimizing the latency time of single messages. Second, for plans with complex procedural aspects, vectorization requires additional operators for synchronization and handling of the explicit data flow. This aspect is explicitly taken into account by cost-based vectorization but might reduce the achievable throughput improvement. Third, low cost plans with many less time-consuming operators might also not benefit from vectorization due to the higher relative overhead of queue management as well as thread synchronization and monitoring. Despite these general limitations of vectorization, the cost-based vectorization can be applied by default due to its hybrid model characteristics (full spectrum between instance-based and vectorized execution) that takes the execution statistics into account. As already mentioned, the concept of cost-based vectorization can also be extended to a distributed setting, where operators are executed by different server nodes rather than only by different threads.

While the vectorization of integration flows is a control-flow-oriented optimization technique, the next chapter will address a data-flow-oriented optimization technique. However, both techniques reduce the execution time of message sequences and thus, increase the message throughput, where the benefit of both techniques can be combined.

5 Multi-Flow Optimization

Similar to the vectorization of integration flows, in this chapter, we introduce the multi-flow optimization [BHL10, BHL11] as a data-flow-oriented optimization technique that is tailor-made for integration flows. This technique tackles the problem of expensive external system access as well as it exploits the optimization potential that equivalent work (e.g., same queries to external systems) is done multiple times. The core idea is to horizontally partition inbound message queues and to execute plan instances for message batches rather than for individual messages. Therefore, this technique is applicable for asynchronous data-driven integration flows, where message queues are used at the inbound side of the integration platform. As a result, the message throughput is increased by reducing the amount of work (external system access and local processing steps) done by the integration platform. We call this technique multi-flow optimization because sequences of messages that would initiate multiple plan instances are processed together.

In order to enable multi-flow optimization, in Section 5.1, we introduce the batch creation via horizontal message queue partitioning. Essentially, two major challenges arise in the context of multi-flow optimization. In Section 5.2, we discuss the challenge of plan execution on batches of messages. Furthermore, in Section 5.3, we describe how this optimization technique is embedded within the periodical re-optimization framework and we address the challenge of computing the optimal waiting time with regard to message throughput maximization. In addition, we provide formal analysis results such as optimality and latency guarantees in Section 5.4. Finally, the experimental evaluation, which is presented in Section 5.5, shows that significant performance improvements in the sense of an increased message throughput are achieved by multi-flow optimization.

5.1 Motivation and Problem Description

In the context of integration platforms, especially in scenarios with huge numbers of plan instances, the major optimization objective is throughput maximization [LZL07] rather than the execution time minimization of single plan instances. The goal is (1) to maximize the number of messages processed per time period, or synonymously in our context, (2) to minimize the total execution time of a sequence of plan instances. Here, depending on the application area, moderate latency times of single messages, in the orders of seconds to minutes, are acceptable [UGA+09]. When addressing this general optimization objective, the following concrete problems have to be considered:

Problem 5.1 (Expensive External System Access)**.** *External system access can be really time-consuming caused by network latency (minimal roundtrip time), external query processing, network traffic, and message transformations from external formats into internal structures. Depending on the involved external systems and on the present infrastructure, the fraction of these influences with regard to the required total access time may vary significantly. However, in particular when accessing custom applications and services, data*

cannot be processed as a stream such that these influences are additive components rather than being subsumed by the most time-consuming influence.

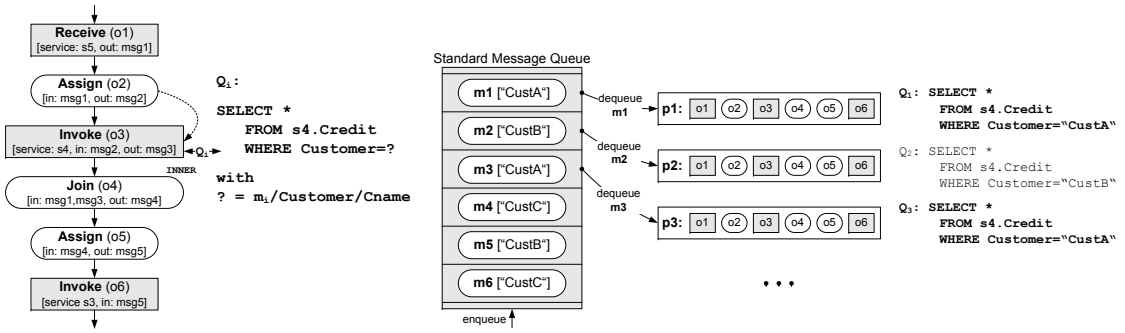
Problem 5.2 (Cache Coherency Problem)**.** *One solution to Problem 5.1 might be the caching of results of external queries. However, this fails due to the integration of heterogeneous and highly distributed systems and applications (loosely coupled without any notification mechanisms). In such distributed environments, caching is not applicable because the central integration platform cannot ensure that the cached data is consistent with the data in the source systems [LZL07]. A similar problem is also known for caching proxy servers, which might break client cache directives (RFC 3143) [CD01].*

Due to this problem, other projects, such as the MT-Cache, use currency bounds (maximum time of caching certain data objects) [GLRG04]. However, they can only ensure weak consistency, while for integration flows eventual consistency is needed as described in Subsection 2.3.2. Caching (without semantic cache invalidation) cannot ensure the properties of (1) read-your-writes (consistency between a writing `Invoke` and a subsequent reading `Invoke` of the same data object) and (2) session consistency (consistency between multiple reading `Invoke` of the same data object) of eventual consistency [Vog08].

Problem 5.3 (Serialized External Behavior)**.** *Depending on the involved external systems, we need to ensure the serial order of messages (see Problem 2.2 Message Outrun). For example, this can be caused by referential integrity constraints within the target systems. Thus, we need to guarantee monotonic reads and writes for individual data objects.*

Given these problems, the optimization objective of throughput maximization has so far only been addressed by leveraging a higher degree of parallelism, such as (1) intra-operator, horizontal parallelism (data partitioning, see [BABO+09]), (2) inter-operator, horizontal parallelism (explicit parallel subflows, see Chapter 3, [LZ05, SMWM06]), and (3) inter-operator, vertical parallelism (pipelining of messages and message parts, see Chapter 4, [PVHL09a, PVHL09b]). Although these techniques can significantly increase the resource utilization and thus, increase the throughput, they do not reduce the executed work. We use an example to illustrate the problem of expensive external system access and how multi-flow optimization addresses this problem.

Example 5.1 (Instance-Based Orders Processing)**.** *Assume our example plan P_2 (Figure 5.1(a)). The instance-based execution model initiates a new plan instance p_i for each*



(a) Example Plan P_2 (b) Instance-Based Plan Execution of P_2

Figure 5.1: Example Instance-Based Plan Execution

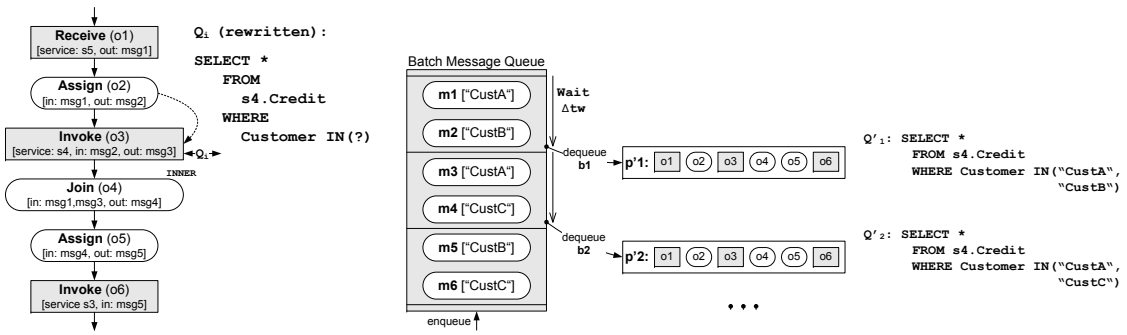
incoming message as shown in Figure 5.1(b). The **`Receive`** *operator (o_1) reads an* order *message from the queue and writes it to a local variable. Then, the* **`Assign`** *operator (o_2) extracts the* customer name *of the received message via XPath and prepares a query with this parameter. Subsequently, the* **`Invoke`** *operator (o_3) queries the external system s_4 in order to load credit rating information for that customer. An isolated SQL query Q_i per plan instance (per message) is used. The* **`Join`** *operator (o_4) merges the result message with the received message (with the* customer key *as join predicate). Finally, the pair of* **`Assign`** *and* **`Invoke`** *operators (o_5 and o_6) sends the result to system s_3. We see that multiple orders from one customer (`CustA`: m_1, m_3) cause us to pose the same query (`Invoke` operator o_3) multiple times to the external system s_4.*

Due to the serialized execution of plan instances, we may end up with work done multiple times, for all types of operators (interaction-oriented, data-flow-oriented as well as control-flow-oriented operators). At this point, multi-flow optimization comes into play, where we consider optimizing the sequence of plan instances. Our core idea is to periodically collect incoming messages and to execute whole message batches with single plan instances. In the following, we give an overview of the naïve, time-based approach as well as the horizontal (value-based) message queue partitioning as batch creation strategies.

Naïve Time-Based Batch Creation

The underlying theme of the naïve (time-based) batching approach, as already proposed in variations for distributed queries [LZL07, LX09], scan sharing [QRR+08], operator scheduling strategies in DSMS [Sch07], and web service interactions [SMWM06, GYSD08b, GYSD08a], is to periodically collect messages (that would initiate plan instances p_i) using a specific waiting time Δtw and merge those messages to message batches b_i. We then execute a plan instance p'_i of the modified (rewritten) plan P' for the message batch b_i. In the following, we revisit our example and illustrate that naïve (time-based) approach.

Example 5.2 (Batch-Orders Processing)**.** *Figure 5.2(b) shows the naïve approach, where we wait for incoming messages during a period of time Δtw and execute the collected messages as a batch. For this purpose, P_2 is rewritten to P'_2 (see Figure 5.2(a)), where in this particular example only the prepared query has been modified.*



(a) Example Plan P'_2 (b) Message Batch Plan Execution of P'_2

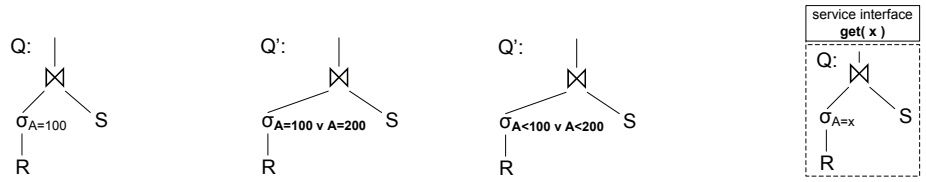
Figure 5.2: Example Message Batch Plan Execution

In this example, the first batch contains two messages (m_1 and m_2) and the second also contains two messages (m_3 and m_4). In order to make use of batch processing, we extend

the query of o_3 (`Invoke`) such that additional information for all messages of the batch is loaded from the external system, which led to an improvement due to fewer plan instances.

The naïve (time-based) approach mainly depends on the waiting time. An increasing waiting time causes a larger number of messages in every batch and therefore it causes a decreasing number of batch plan instances. As a result, the relative execution time per message might decrease and message throughput increases. However, the naïve approach has the following major drawback:

Problem 5.4 (Distinct Messages in the Batch)**.** *Due to the simple (time-based) model of collecting messages, there might be multiple distinct messages in the batch according to the attributes used by the operators of P'. It follows that we need to rewrite the queries to external systems. Figure 5.3 illustrates common cases of those queries.*



(a) Original (b) Disjunctive Query (w/o and w/ overlap) (c) Split Queries

Figure 5.3: Common Cases of Rewritten Queries to External Systems

Distinct messages in a batch implies two major problems. First, this might have negative performance influence on the overall execution time. Suppose the external query shown in Figure 5.3(a), where the attribute value $x = 100$ was extracted from a message. If we have multiple distinct message according to this attribute in a batch, we insert a disjunctive predicate into the query (Figure 5.3(b)). For equality predicates this leads to an increased cardinality of intermediate results and thus the execution time of following operators suffers. Due to possibly superlinear complexity (e.g., join or group-by) of following operators, the execution time of a rewritten external query might be even higher than the total execution time of independent queries. In addition, due to the disjunctive predicate, the data transfer over the network is unchanged. For range predicates, the rewritten queries are faster if there is a partial overlap. In this case, also the amount of transferred data is reduced.

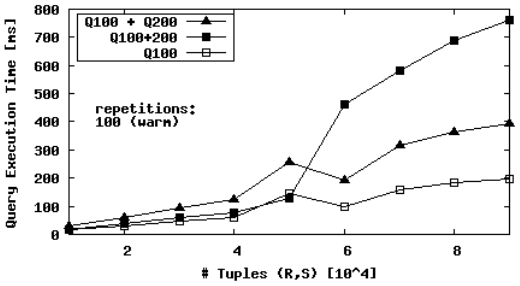


Figure 5.4: Query Execution Times

Figure 5.4 shows the results of an experiment (warm) that investigates the influence of rewritten queries on the (local) query execution time using a local (embedded) Derby instance. It compares the execution times of a single original query (Q100), a batch of two

original queries (Q100 + Q200) and the rewritten query with non-overlapping disjunctive predicate (Q100+200). In addition, for both types of disjunctive queries the integration platform must post-process the received data in order to ensure correctness by assigning only partial results to the individual messages. Second, the rewriting of queries might not be possible at all for certain service interfaces or custom applications such that a single query for each distinct message in the batch must be used. For example, Figure 5.3(c) shows a service interface with a single parameter, where the service implementation always uses the same query template such that the query cannot be rewritten by an external client. Thus, the possible throughput improvement strongly depends on the number of distinct items in the batch. We cannot precisely estimate this influence of rewritten queries due to missing knowledge about data properties of involved external systems [IHW04].

In conclusion, the naïve approach can hurt performance. Furthermore, it requires that queries to external systems can be rewritten according to the different items in the batch. This is not always possible when integrating arbitrary systems and applications.

Batch Creation via Horizontal Queue Partitioning

Due to Problem 5.4, we propose the concept of horizontal message queue partitioning[11] as batch creation strategy. The basic idea is to horizontally partition the inbound message queues according to specific partitioning attributes *ba*. With such value-based partitioning, all messages of a batch exhibit the same attribute value according to the partitioning attribute. Thus, certain operators of the plan only need to access this attribute once for the whole partition rather than for each individual message. The core steps are (1) to derive the partitioning attributes from the integration flow, (2) to periodically collect messages during an automatically computed waiting time Δtw, (3) to read the first partition from the queue and (4) to execute the messages of this partition as a batch with an instance p'_i of a rewritten plan P'. Additionally, (5) we might need to ensure the serial order of messages at the outbound side. In order to illustrate the core idea, we revisit our example.

Example 5.3 (Partitioned Batch-Orders Processing)**.** *Figure 5.5 reconsiders the example for partitioned multi-flow execution. The rewritten plan P'_2 is equivalent to the instance-based plan (Figure 5.1(a)) because (1) external queries require no rewriting at all, and (2) plan rewriting is only required for multiple partitioning attributes.*

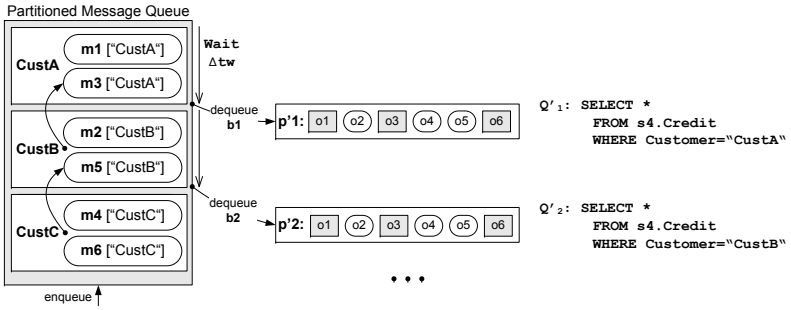


Figure 5.5: Partitioned Message Batch Execution P'_2

[11] Horizontal data partitioning [CNP82] is strongly applied in DBMS and distributed systems. Typically, this is an issue of physical design [ANY04], where a table is partitioned by selection predicates (value).

The incoming messages m_i are partitioned according to the partitioning attribute customer name *that was extracted with $ba = m_i/Customer/Cname$ at the inbound side. A plan instance of the rewritten plan P_2' reads the first partition from the queue and executes the single operators for this partition. Due to the equal values of the partitioning attribute, we do not need to rewrite the query to the external system s_4. Every batch contains exactly one distinct attribute value according to ba. In total, we achieve performance benefits for the* `Assign`*, as well as the* `Invoke` *operators. Thus, the throughput is further improved because the execution time of such a batch does not include any distinct messages anymore. Note that the incoming order of messages was changed (arrows in Figure 5.5) and therefore needs to be serialized at the outbound side.*

It is important to note that beside external queries (`Invoke` operator), also local operators (e.g., `Assign` and `Switch`) can directly benefit from horizontal partitioning. Partitioning attributes are derived from the plan (e.g., query predicates, or switch expressions). This benefit is caused by operation execution on partitions instead of on individual messages. A similar underlying concept is also used for pre-aggregation [IHW04] or early-group-by [CS94]. In addition, all operators that work on partitions of equal messages (e.g., loaded once from an external system) also need to be executed only once.

Our cost-based optimizer realizes the optimization objective of throughput maximization by monitoring several statistics of the stream of incoming messages and by periodical re-optimization, where the optimal waiting time Δtw is computed. In case of low message rates (no full utilization of the integration platform), the waiting time is decreased in order to ensure low latency of single messages. As the message rate increases, the waiting time is increased accordingly to increase the message throughput by processing more messages per batch, while preserving maximum latency constraints for single messages.

MQO (Multi-Query Optimization) and OOP (Out-of-Order Processing) [LTS+08] have already been investigated for other system types. In contrast to existing work, we present the novel MFO approach that is tailor-made for integration flows and that maximizes the throughput by employing horizontal message queue partitioning and computing the optimal waiting time. MFO is also related to caching and the recycling of intermediate results [IKNG09]. While caching might lead to use of outdated data, the partitioned execution might cause reading more recent data of different objects. However, we cannot ensure strong consistency by using asynchronous integration flows (decoupled from clients with message queues) anyway. Furthermore, with regard to eventual consistency [Vog08], we guarantee (1) monotonic writes, (2) monotonic reads with regard to individual data objects[12], (3) read-your-writes/session consistency, (4) semantic correctness as defined in Definition 3.1, (5) that the temporal gap of up-to-dateness is at most equal to a given latency constraint, and (6) that no outdated data is read. In contrast, caching cannot guarantee read-your-writes/session consistency and that no outdated data is read. In conclusion, caching is advantageous if data of external sources is static and the amount of data is rather small, while MFO is beneficial if data of external sources changes dynamically.

Finally, the major research challenges of MFO via horizontal partitioning are (1) to enable plan execution of horizontally partitioned message batches and (2) to compute the optimal waiting time Δtw during periodical re-optimization. We address these two challenges in Section 5.2 and Section 5.3, respectively.

[12] Both caching and MFO cannot ensure monotonic reads over multiple data objects due to different read times of certain data objects.

5.2 Horizontal Queue Partitioning

In order to enable plan execution of message partitions, several preconditions are required. First, we describe the tailor-made message queue data structure *partition tree*. It enables horizontal partitioning according to multiple attributes because a single plan might include several predicates, where we can benefit from partitioning. Second, we explain changes of the deployment process, which include (1) the derivation of partitioning attributes from a given plan, (2) the derivation of the optimal partitioning scheme in case of multiple attributes, and (3) the rewriting of plans according to the chosen partitioning scheme.

5.2.1 Maintaining Partition Trees

As the foundation for multi-flow optimization, we introduce the *partition tree* as a partitioned message queue data structure. Essentially, this tree is a simplified multi-dimensional B*-Tree (MDB-Tree) [SO82] with specific extensions, where the messages are horizontally (value-based) partitioned. Similar to a traditional MDB-Tree, each tree level represents a different partitioning attribute. The major difference to a traditional MDB-Tree is that the partitions are sorted according to their timestamps of creation rather than according to the key values of index attributes. This is reasoned by queuing semantics that imply a temporal order. Thus, at each tree level, a list of partitions, unsorted with regard to the attribute values, is stored. Formally, the partition tree is defined as follows:

Definition 5.1 (Partition Tree)**.** *The partition tree is an index for multi-dimensional attributes. It contains h levels, where each level represents a partition attribute $ba_i \in \{ba_1, ba_2, \ldots, ba_h\}$. For each attribute ba_i, a list of batches (partitions) b are maintained. Those partitions are ordered according to their timestamps of creation $t_c(b_i)$ with $t_c(b_{i-1}) \leq t_c(b_i) \leq t_c(b_{i+1})$. Only the last index level ba_h contains the queued messages. A partition attribute has a type with type$(ba_i) \in \{$`value`, `value-list`, `range`$\}$.*

Such a partition tree is used as our message queue representation. Similar to usual message queues, it decouples the inbound adapters from the process engine in order to enable receiving incoming messages even in the case of overload situations. For example, such situations might be caused by (1) workload peaks, or (2) temporarily unavailable target systems. Additionally, the partition tree realizes the horizontal queue partitioning. We use an example in order to explain the structure of this partition tree more clearly.

Example 5.4 (Partition Tree)**.** *Assume two partitioning attributes ba_1 (customer, `value`) and ba_2 (total price, `range`) that have been derived from a plan P. Figure 5.6 shows this*

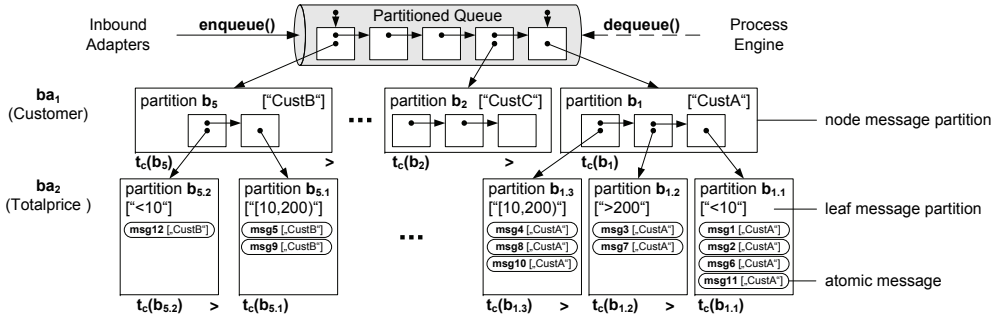


Figure 5.6: Example Partition Tree

example partition tree of height $h = 2$. On the first index level, the messages are partitioned according to customer names $ba_1(m_i)$, and on the second level, each partition is divided according to the range of order total prices $ba_2(m_i)$. Horizontal partitioning with the temporal order of partitions reason the outrun of single messages. However, the messages within an individual partition are still sorted according to their incoming order.

Based on the partition tree data structure, there are two fundamental maintenance procedures of such a horizontally partitioned message queue:

- The `enqueue` operation is invoked by the inbound adapters whenever a message was received and transformed into the internal representation. During message transformation, partitioning attributes are extracted with low cost as well. Algorithm 5.1 describes the `enqueue` operation. We use a thread monitor approach for synchronization of those enqueue and dequeue operations in order to avoid any busy waiting. Subsequently, if a partition with $ba_l(m_i) = ba(b_i)$ already exists, the message is inserted; otherwise, a new partition is created and added at the end of the list. The recursive insert operation depends on the partition type, where we distinguish node partitions (index levels 1 to $h-1$) and leaf partitions (last index level that contains the messages). In case of a node partition, the algorithm part line 3 to line 13 is used recursively, while in case of a leaf partition, the message is simply added to the end of the list of messages in this partition.
- The process engine then periodically invokes the `dequeue` operation according to the computed waiting time Δtw. This `dequeue` operation removes and returns the first top-level partition with $b^- \leftarrow b_1$. The removed partition exhibits the property of being the oldest partition within the partition tree with $t_c(b_1) = \min_{i=1}^{|b|} t_c(b_i)$, which ensures that starvation of messages is impossible.

The operations `enqueue` and `dequeue` are invoked for each individual message. Therefore, it is meaningful to discuss the worst-case complexity for both operations. Let $sel(ba_i)$

Algorithm 5.1 Partition Tree Enqueue (A-PTE)

Require: message m_i, partitioning attribute values $ba(m_i)$, index level l
initialized counter $c \leftarrow 0$, boolean SEB (serialized external behavior)

```
1:  while ∑_{j=1}^{|b|} |b_j| > q do                    // ensure maximum queue constraint q
2:    WAIT                          // wait for notifying dequeue() (without busy waiting)
3:  for j ← |b| to 1 do                                          // for each partition
4:    if ba_l(m_i) = ba(b_j) then                          // if partition already exists
5:      b^+ ← b_j
6:      break
7:    else if SEB then
8:      c ← c + |b_j|                    // count outrun messages for later serialization
9:  if b^+ = NULL then                                       // if partition does not exist
10:   b^+ ← create partition with t_c(b^+) ← t_i(m_i)
11:   b_{|b|+1} ← b^+                                                // add as last partition
12: m_i.c ← c              // zero in case of newly created partitions or for disabled SEB
13: b^+.insert(m_i, ba(m_i), l − 1)                                        // recursive insert
14: NOTIFY                                                 // notify waiting dequeue()
```

denote the average selectivity of a single partitioning attribute. Thus, $1/sel(ba_i)$ represents the number of distinct values of this attribute, which is equivalent to the worst-case number of partitions at index level i. These partitions are unsorted according to the partitioning attribute. Thus, due to the resulting linear comparison of partitions at each index level, the `enqueue` operation exhibits a worst-case time complexity of $O(\sum_i^h 1/sel(ba_i))$. In contrast to this, the `dequeue` operation exhibits a constant worst-case time complexity of $O(1)$ because it simply removes the first top-level partition. In conclusion, there is only moderate overhead for maintaining partition trees if the selectivity is not too low.

However, in order to ensure robustness with regard to arbitrary selectivities, we extend the basic structure of a partition tree to the *hash partition tree*. Figure 5.7 illustrates an example of its extended queue data structure. Essentially, a hash partition tree is a partition tree with a hash table as a secondary index structure over the partitioning attribute values (primarily applicable for attribute types `value` and `range`).

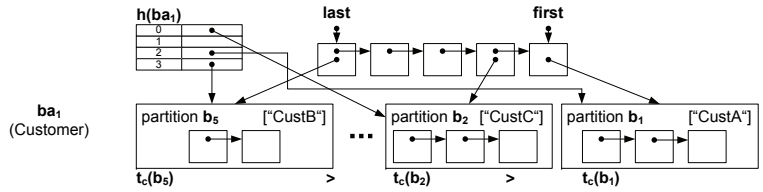


Figure 5.7: Example Hash Partition Tree

This hash table is used in order to probe (`get`) if there already exist a value of a partitioning attribute. If so, we insert the message into the corresponding partition; otherwise, we create a new partition, append it at the end of the list and `put` a reference to this partition into the hash table as well. Accordingly, the `dequeue` operation still gets the first partition from the list but additionally `removes` the pointer to this partition from the hash table. As a result, we reduced the complexity for both operations, `enqueue` and `dequeue`—for the case, where no serialized external behavior (SEB) is required—to constant time complexity of $O(1)$. Despite, this probe possibility, for SEB, we additionally need to determine the number of messages that have been outrun if a related partition already exist. As a result, for the case, where SEB is required, the worst-case complexity is still $O(\sum_i^h 1/sel(ba_i))$ but we benefit if the partition does not exist already.

The requirement of serialized external behavior (SEB) implies the synchronization of messages at the outbound side. Therefore, we extended the message structure by a counter c with $c \in \mathbb{Z}^+$ to $m_i = (t_i, c_i, d_i, a_i)$. If a message m_i outruns another message m_j during the `enqueue` operation, its counter c_i is incremented by one. The serialization at the outbound side is then realized by comparing source message IDs similar to the serialization concept of Chapter 4, and for each reordered message, the counter is decremented by one. Thus, at the outbound side, we are not allowed to send message m_i until $c_i = 0$. This counter-based serialization concept[13] is required in addition to the concept of `AND` and `XOR` serialization operators (introduced in Chapter 4) in order to allow out-of-order execution rather than just parallel execution of concurrent operator pipelines. Despite this serialization concept, we cannot execute message partitions in parallel because this would

[13] Counter-based serialization works also for CN:CM multiplicities between input and output messages, where locally created messages get the counter of the input messages. For example, N:C multiplicities arise if there are writing interactions in paths of a `Switch` operator. This is addressed by periodically flushing the outbound queues, where all counters of messages that exceed lc are set to zero.

not allow a discrete counting of outrun messages due to the possible outrun of partitions.

5.2.2 Deriving Partitioning Schemes

The partitioning scheme in terms of the optimal layout of the partition tree is derived automatically. This includes (1) deriving candidate partitioning attributes from the given plan and (2) to find the optimal partitioning scheme for the overall partition tree.

Candidate partition attributes are derived from the single operators o_i of plan P. We realize this by searching for attributes that are involved in predicates, expressions and dynamic parameter assignments. This is a linear search over all operators with $O(m)$. Due to different semantics of those attributes, we distinguish between the following three types of partitioning attributes, which have been introduced in Definition 5.1:

1. *Value:* This scheme causes a data partitioning by exact value. Thus, for all $1/sel$ distinct values of this attribute, a partition is used. An example for this type is an equality predicate of a query to an external system (`Assign` and `Invoke` operator).
2. *Value List:* Due to disjunctive predicates of external queries or local `Switch` operators, we can also use a list of exact values with or-semantics.
3. *Range:* According to range query predicates or inequalities range partitioning is used. Examples for this type are expressions of `Switch` operators or range predicates of queries to external systems (`Assign` and `Invoke` operator).

After having derived the set of candidate partitioning attributes and the type of each partitioning attribute, we need to select candidates that are advantageous to use. First, we remove all candidates, where a partitioning attribute refers to externally loaded data because these attribute values are not present for incoming messages at the inbound side of the integration platform. Second, we compare the benefit of using a partitioning attribute (see Section 5.3) with a user-specified cost reduction threshold τ and remove all candidates that are evaluated as being below this threshold.

Based on the set of partitioning attributes, we create a concrete partitioning scheme for the partition tree. For h partitioning attributes, there are $h!$ different partitioning schemes. Due to this factorial complexity, we use a heuristic for finding the optimal scheme. The intuition is to minimize the number of partitions in the index. Assuming no correlations[14], we order the index attributes according to their selectivities with

$$\min \sum_{i=1}^{h} |b \in ba_i| \quad \text{iff} \quad sel(ba_1) \geq sel(ba_i) \geq sel(ba_h). \tag{5.1}$$

Hence, finding the best partitioning scheme exhibits a complexity of $O(h \log h)$ due to the requirement of sorting the h partitioning attributes. Another approach would be to take the additional costs of rewritten plans into account. However, the costs of additional operators (splitting and merging of partitions) have shown to be negligible.

Example 5.5 (Minimizing the Number of Partitions)**.** *Recall Example 5.4 and assume the partitioning attributes* ba_1 *(customer,* `value`*) and* ba_2 *(total price,* `range`*) as well*

[14]The MFO approach works also for correlated partitioning attributes, where queue management might become more expensive due to mis-estimated selectivities. Alternatively, the correlation table introduced in Subsection 3.3.4 could be used for correlation-aware ordering of partitioning attributes.

as the monitored average value selectivities $sel(ba_1) = 1/3$ *and* $sel(ba_2) = 1/10$*. This results in two possible partitioning schemes with maximum partition numbers of* $|ba_1, ba_2| = 3 + 3 \cdot 10 = 33$ *and* $|ba_2, ba_1| = 10 + 10 \cdot 3 = 40$*. Thus, we select* (ba_1, ba_2) *as the best partitioning scheme because* $sel(ba_1) \geq sel(ba_2)$*. For a given message subsequence, this results in three instead of ten plan instances.*

Having minimized the total number of partitions, we minimized the overhead of queue maintenance and more importantly maximized the number of messages per top-level partition, which reduces the number of required plan instances. The result is the optimal partitioning scheme with regard to relative execution time and thus message throughput.

5.2.3 Plan Rewriting Algorithm

With regard to executing hierarchical message partitions, only slight changes of physical operator implementations are necessary. All other changes are made on logical level when rewriting a plan P to P' during the initial deployment or during periodical re-optimization.

For the purpose of plan rewriting, several integration flow meta model extensions are required. First, the message meta model is extended in such a way that an abstract message can be either an atomic message or a message partition, where the latter is described by a partitioning attribute ba_i as well as the type and the values of this partitioning attribute. In addition, the message partition can be a node partition, which references child partitions, or a leaf partition, which references atomic messages. All operators that benefit from partitioning (e.g., `Invoke`, `Assign`, or `Switch`) are modified accordingly, while all other operators transparently split the incoming message partition into all atomic messages, are executed for each message, and then they repartition the messages after execution. Second, the flow meta model is extended by two additional operators that are described in Table 5.1. They represent inverse functionalities as shown in Figure 5.8.

Based on these two additional operators, the logical plan rewriting is realized with the so-called *split and merge* approach. From a macroscopic view, a plan receives the top-level partition, dequeued from the partition tree. Then, we can execute all operators that benefit from the top-level partitioning attribute. Just before an operator that benefits from a lower-level partition attribute, we need to insert a `PSplit` operator that splits the top-level partition into the $1/sel(ba_2)$ subpartitions (worst case) as well as an `Iteration` operator (foreach) that iterates over these subpartitions. The sequence of operators that benefit from this granularity are used as iteration body. After this iteration, we insert a `PMerge` operator in order to merge the resulting partitions back to the top-level partition if required (e.g., if a subsequent operator benefit from higher level partitioning attributes).

Table 5.1: Additional Operators for Partitioned Plan Execution

Name	Description	Input	Output	Complex
`PSplit`	Reads a message partition, splits this partition into the next level partitions, and returns a directly accessible array of abstract messages.	(1,1)	(1,*)	No
`PMerge`	The inverse operation to a `PSplit` operator reads an array of message partitions and groups this messages into a single partition.	(1,*)	(1,1)	No

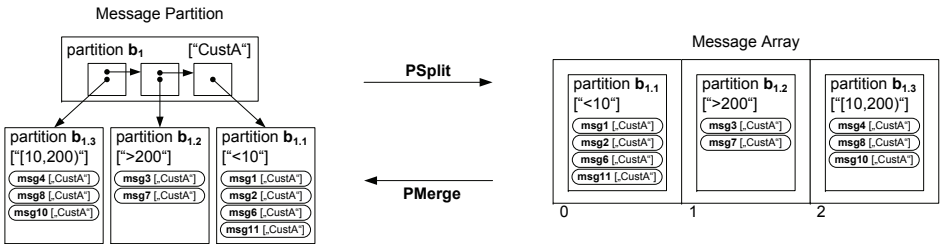
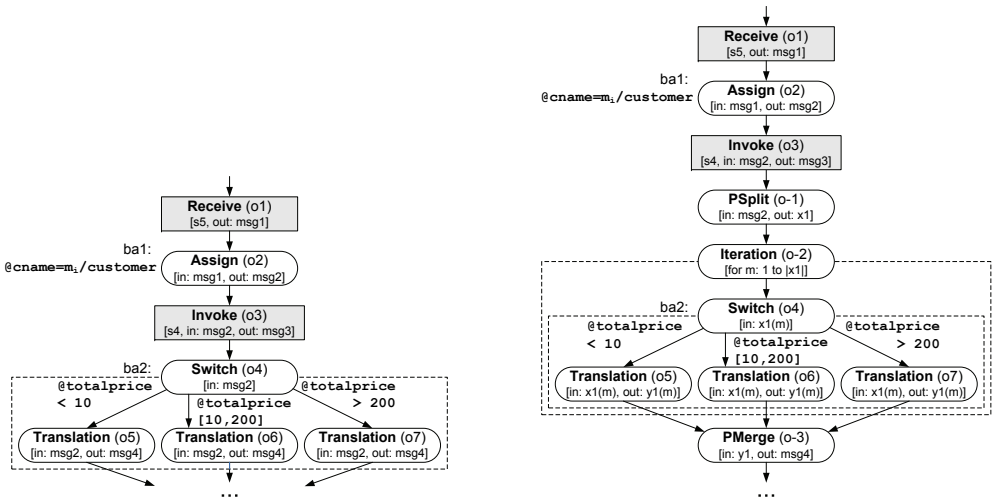


Figure 5.8: Inverse Operators `PSplit` and `PMerge`

Recall that if we have only one partition attribute, we do not need to apply *split and merge* because each operator can work on a single partition level. We use an example to illustrate the concept of split and merge.

Example 5.6 (Plan Rewriting)**.** *Assume a subplan P as shown in Figure 5.9(a). Here, we receive a message from system s_5, create a parameterized query, and request system s_4. Afterwards, we use an alternative switch path depending on an expression, and finally, proceed with the rest of the plan. According to Example 5.5, we have derived the two partitioning attributes ba_1 (customer, `value`) and ba_2 (total price, `range`).*



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

Figure 5.9: Example Plan Rewriting Using the Split and Merge Approach

If we use the partitioning scheme (ba_1, ba_2), we rewrite the plan as shown in Figure 5.9(b). First, we insert a `PSplit` and an `Iteration` operator just before the `Switch` operator. As a result, o_2 and o_3 benefit from the top-level partition, while o_4 is executed for each sub-partition of the top-level partition. Second, right after this `Switch` operator, a `PMerge` is inserted as well, because subsequent operators benefit from the top level partition.

Algorithm 5.2 illustrates the plan rewriting. For simplicity of presentation, we use a linear representation of the algorithm, while the realization of this is recursive according to the hierarchy of sequences of complex and atomic operators. Essentially, this algorithm consists of two parts. First, there is the dependency analysis that we have already described in Subsection 3.2.1. Second, plan rewriting requires as input the operator sequence

Algorithm 5.2 MFO Plan Rewriting (A-MPR)

Require: operator sequence o, partitioning scheme ba

```
1:  D ← ∅
2:  for i ← 1 to |o| do                                          // for each operator o_i
3:    // Part 1: Dependency Analysis
4:    for j ← i to |o| do                                        // for each following operator o_j
5:      if ∃o_j --δ--> o_i then                                  // existing data dependency over data object var
6:        D ← D ∪ d⟨o_j, var, o_i⟩ with d⟨o_j, var, o_i⟩ ← create dependency
7:    // Part 2: Plan Rewriting
8:    for k ← 1 to |D| do                                        // for each dependency d
9:      d⟨o_y, q, o_x⟩ ← d_k
10:     if o_i ≡ o_y then                                        // o_i is target of data dependency
11:       if ba(o_y, var) < ba(o_x, var) then                    // o_i benefits from lower-level partition
12:         o_x.insertAfter(o_s) with o_s ←createPSlit
13:         o_y.insertBefore(o_iter) with o_iter ←createIteration
14:         ba_cur ← ba(o_y, var)
15:         for j ← i to |o| do                                  // for each following operator o_j
16:           if ba(o_j) = ba_cur or ba(o_j) = NULL then
17:             o.remove(o_j)
18:             o_iter.insertLast(o_j)
19:             D.modify(o_iter, o_j)                          // change the old dependencies
20:           else if ba(o_j) < ba_cur then
21:             recursive plan rewriting (lines 12-24)
22:           else                                               // ba(o_j) > ba_cur
23:             o_j.insertBefore(o_m) with o_m ←createPMerge
24:             break
25: return o
```

o, the derived partitioning scheme ba and the created set of dependencies D. For each operator o_i in o, we iterate over all dependencies in D followed by the main rewriting rules. If the partitioning attribute of operator o_x (source of the data and thus, target of data dependency) is higher than the partitioning attribute of the source of the data dependency o_y, we insert a `PSplit` operator as well as an `Iteration` operator. We then iterate over all following operators (including o_y) and evaluate if they can be included into the iteration body by comparing the partitioning attributes with $ba(o_j) = ba_{cur}$. If the required level is a lower-level attribute, we recursively invoke the insertion of `PSplit` and `Iteration` operators. In contrast, if we determined an operator with a higher-level attribute, we found the end of the current `Iteration` operator and insert a `PMerge` operator. For binary operators (e.g., `Join` operator), there might be a difference between partitioning attributes (for one side an `Iteration` was inserted). In this case, we do not use a combined iteration but a direct positional lookup at the higher-level partition. A single `PSplit` or `PMerge` operator can bridge multiple levels of the partitioning scheme such that for plan rewriting, the lower, equal, and higher comparison is sufficient.

This algorithm is used whenever the partitioning scheme (derived from the plan) changes during periodical re-optimization. Thus, although the algorithm is not executed during each re-optimization step, it might be used during runtime and hence, it is worth to

mention its complexity. The A-MPR exhibits—similar to the plan vectorization algorithm (A-PV, Subsection 4.2.2)—a cubic worst-case time complexity of $O(m^3)$ according to the number of operators m. The rationale for this is the already analyzed dependency checking. Note that the additional inner loop over following operators do not change this asymptotic behavior because each operator is assigned only once to an inserted `Iteration` operator.

The split and merge approach realizes the transparent plan rewriting and thus enables the execution of message partitions even in the case of multiple partitioning attributes. The rewritten plan mainly depends on the cost-based derived partitioning scheme, which neglects any additional costs of `PSlit` and `PMerge` operators. The reason for this optimization objective is that the ordering of partitioning attributes has a higher influence on the overall performance (partitioned queue maintenance and benefit by partitioning) than the additional operators, because `PSplit` and `PMerge` are low-cost operators with linear scalability with regard to the number of messages due to the efficient hash partition tree data structure. In addition to the throughput improvement achieved by executing operations on partitions of messages, the inserted `Iteration` operators offer further optimization potential. In detail, the technique *WC3: Rewriting Iterations to Parallel Flows*, described in Subsection 3.4.1, can be applied after the A-MPR in order to additionally achieve a higher degree of parallelism that further increases the throughput.

To summarize, we discussed the necessary preconditions in order to enable the horizontal message queue partitioning and the execution of operations on these message partitions. In detail, we introduced the (hash) partition tree as a message queue data structure that allows the hierarchical partitioning of messages according to certain partitioning attributes. Furthermore, we introduced basic algorithms (1) for deriving candidate partitioning attributes from a plan, (2) for deriving the optimal partitioning scheme of attributes, and (3) for rewriting the plan according to this scheme. Only minor changes of operators and the execution environment are necessary, while all other aspects are issues of logical optimization and therefore, fit seamlessly into our cost-based optimization framework. Multi-flow optimization now reduces to the challenge of computing the optimal waiting time.

5.3 Periodical Re-Optimization

The cost-based decision of the multi-flow optimization technique is to compute the optimal waiting time Δtw in order to adjust the trade-off between message throughput and latency times of single messages according to the current workload characteristics. In this section, we define the formal optimization objective, we explain the extended cost model and cost estimation, we discuss the waiting time computation and finally, show how to integrate this optimization technique into our cost-based optimization framework.

5.3.1 Formal Problem Definition

As described in Section 2.3.1, we assume a message sequence $M = \{m_1, m_2, \ldots, m_n\}$ of incoming messages, where each message m_i is modeled as a (t_i, d_i, a_i)-tuple, where $t_i \in \mathbb{Z}^+$ denotes the incoming timestamp of the message, d_i denotes a semi-structured tree of name-value data elements, and a_i denotes a list of additional atomic name-value attributes. Each message m_i is processed by an instance p_i of a plan P, and $t_{out}(m_i) \in \mathbb{Z}^+$ denotes the timestamp when the message has been successfully executed. Here, the latency of a single message $T_L(m_i)$ is given by $T_L(m_i) = t_{out}(m_i) - t_i(m_i)$.

With regard to the optimization objective of throughput maximization, we need some additional notation. The execution characteristics of a hypothetical finite message subsequence M' with $M' \subseteq M$ are described by two major statistics:

- *Total Execution Time $W(M')$:* The total execution time of a finite message subsequence is determined by $W(M') = \sum W(p')$ as the sum of the execution time of all partitioned plan instances required to execute all messages $m_i \in M'$.
- *Total Latency Time $T_L(M')$:* The total latency time of a finite message subsequence is determined by $T_L(M') = t_{out}(m_{|M'|}) - t_i(m_1)$ as the time between receiving the first message until the last message has been executed (both with regard to the subsequence M'). This includes overlapping execution time and waiting time.

Following these prerequisites, we now formally define the multi-flow optimization problem.

Definition 5.2 (Multi-Flow Optimization Problem (P-MFO))**.** *Maximize the message throughput with regard to a hypothetical finite message subsequence M'. The optimization objective ϕ is to execute M' with minimal latency time:*

$$\phi = \max \frac{|M'|}{\Delta t} = \min T_L(M'). \tag{5.2}$$

There, two additional restrictions must hold:

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

Finally, the P-MFO describes the search for the optimal waiting time Δtw with regard to ϕ and the given constraints.

Based on the horizontal partitioning of message queues, Figure 5.10 illustrates the basic temporal aspects that need to be taken into account, for a scenario with partitioning attribute selectivity of $sel = 1.0$.

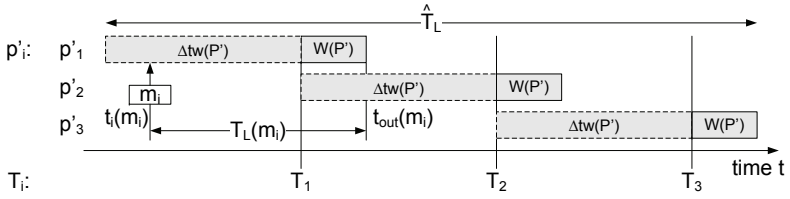


Figure 5.10: P-MFO Temporal Aspects (with $\Delta tw > W(P')$)

Essentially, an instance p'_i of a rewritten plan P' is initiated periodically at T_i, where the period is determined by the waiting time Δtw. A message partition b_i is then executed by p'_i with an execution time of $W(P')$. Finally, we want to minimize the total latency time $\hat{T}_L(M')$ in order to solve the defined optimization problem. Comparing Δtw and $W(P')$, we have to distinguish the following three cases:

- **Case 1:** $\Delta tw = W(P')$: We wait exactly for the period of time that is required for executing partition b_i before executing b_{i+1}. This is the simplest model and results in the full utilization by a single plan. As we will show this specific case can be one desirable aim with regard to the optimization objective ϕ.

- **Case 2:** $\Delta tw > W(P')$ (Figure 5.10): The execution time is shorter than the waiting time. This can improve the throughput and is advantageous for multiple plans.
- **Case 3:** $\Delta tw < W(P')$: The waiting time is shorter than the execution time. Thus, the result would be (1) temporally overlapping plan instances for different partitions or (2) growing message queue sizes (increasing k').

Problem 5.5 (Waiting Time Inconsistency)**.** *When computing the waiting time Δtw, we must prevent case 3 ($\Delta tw < W(P')$) because it would lead to the parallel execution of plan instances, where we could not ensure serialization. Furthermore, there might be an a-priori violated latency constraint with $T_L(M') \geq lc$ or the latency constraint lc might be set to impossible values with regard to the execution time $W(P')$.*

In order to overcome Problem 5.5, we define the *validity condition*: For a given latency constraint lc, there must exist a waiting time Δtw such that $(0 \leq W(P') \leq \Delta tw) \wedge (0 \leq \hat{T}_L(M') \leq lc)$; otherwise, the constraint is invalid. In other words, (1) we avoid case 3, and (2) we check if the worst-case latency of a single message—in the sense of the estimated total latency of $1/sel$ distinct partitions—fulfills the latency constraint lc.

5.3.2 Extended Cost Model and Cost Estimation

In order to estimate the costs of plan instances for specific batch sizes k' with $k' = |b_i|$, which is required for computing the optimal waiting time, we need to adapt the used cost model for integration flows that has been described in Subsection 3.2.2. Basically, only operators that benefit from partitioning require extensions.

The cost model extensions address the abstract costs as well as the execution time because interaction-, control-flow-, and data-flow-oriented operators are affected by partitioning. Table 5.2 shows these extended costs $C(o_i', k')$ and execution times $W(o_i', k')$ in detail. The extended costs of operators that directly benefit from partitioning, namely `Invoke`, `Receive`, `Switch`, and `Assign`, are independent of the number of messages k'. For example, the `Invoke` operator executes an external query once for the whole batch. Further, also the `Switch` operator evaluates its path expressions on the partitioning attribute and thus, for the whole batch rather than on individual messages. However, the `Switch` operator recursively contains the extended costs of arbitrary child operators that might benefit from partitioning or not. In contrast, the costs of all other operators (that do not benefit from partitioning) linearly depend on the number of messages in the batch k'. This is also true for `Invoke`, `Receive`, `Switch`, and `Assign` operators if they do not

Table 5.2: Extended Double-Metric Operator Costs for Message Partitions

Operator Name	**Abstract Costs** $C(o_i', k')$	**Execution Time** $W(o_i', k')$
`Invoke`	$\lvert ds_{in}\rvert + \lvert ds_{out}\rvert$	$W(o_i)$
`Receive`	$\lvert ds_{out}\rvert$	$W(o_i)$
`Switch`	$\lvert ds_{in}\rvert$	$\sum_{i=1}^{n} \left(P\left(path_i\right) \cdot \left(\sum_{j=1}^{i} W\left(expr_{path_j}\right) + \sum_{l=1}^{m_i} W\left(o_{i,l}', k'\right) \right) \right)$
`Assign`	$\lvert ds_{in}\rvert + \lvert ds_{out}\rvert$	$W(o_i)$
other	$C(o_i) \cdot k'$	$W(o_i) \cdot k'$

benefit from a partitioning attribute (e.g., for writing interactions of an `Invoke` operator). However, in some cases (e.g., operations on externally loaded data), all operators can benefit from partitioning as well because they are inherently executed only once and just expanded to the batch size if required (e.g., by the first binary operator that receives a message partition as one of its inputs). For example, as we load data for a partition of messages, we can execute any subsequent transformation of this loaded data also only once.

For operators that do not benefit from partitioning, the abstract costs are computed by $C(o_i', k') = C(o_i) \cdot k'$ and the execution time can be computed by $W(o_i', k') = W(o_i) \cdot k'$ or by $W(o_i', k') = W(o_i) \cdot C(o_i', k')/C(o_i)$. Finally, if $k' = 1$, we get the instance-based costs with $C(o_i', k') = C(o_i)$ and $W(o_i', k') = W(o_i)$. Thus, the instance-based execution is a specific case of the execution of horizontally partitioned message batches. As a result, theoretically, partitioning cannot cause any performance decrease of an operator.

In addition to the mentioned operators that can benefit from partitioning, there are further operators that might also benefit from partitioning. Examples for these are the `Join`, `Selection`, and `Groupby` operators. However, due to the partitioning of complete messages (with tree-structured data) partitioning applies only to specific cases, where a message has only a single tuple (to which the value of the partitioning attribute refers). Hence, we do not consider these operators because the possible benefit is strongly limited. Nevertheless, these operators could be included with benefit if streaming of message parts (e.g., a part for each tuple) [PVHL09a, PVHL09b] is applied because, we could execute `Join`, `Selection`, and `Groupby` operators efficiently on whole batches of these parts. We use our example plan P_2 in order to illustrate the overall cost estimation in detail.

Example 5.7 (Extended Cost Estimation)**.** *Recall the rewritten plan P_2' (Figure 5.5) and assume a number of k' messages per message partition. Using the extended cost model, we can estimate the execution time $W(P_2', k')$. The monitored average execution times $W(o_i)$ are shown in the table in Figure 5.11. Now, we compute $W(P_2', k')$ as follows:*

$$W(P_2', k') = \sum_{i=1}^{m} W(o_i', k') = W(o_1) + W(o_2) + W(o_3) + W(o_4) \cdot k' + W(o_5) \cdot k' + W(o_6) \cdot k'$$
$$= W(o_1) + W(o_2) + W(o_3) + (W(o_4) + W(o_5) + W(o_6)) \cdot k'$$

The operators o_1, o_2, and o_3 benefit from partitioning and hence, we assign costs that are independent of k', while costs of operators o_4, o_5, and o_6 increase linearly with k'. Using this cost function of P_2 we can estimate the execution time for an arbitrary number

Operator o_i	**Execution Time** $W(o_i)$
o_1	0.01 s
o_2	0.015 s
o_3	0.3 s
o_4	0.055 s
o_5	0.02 s
o_6	0.13 s
P	0.53 s

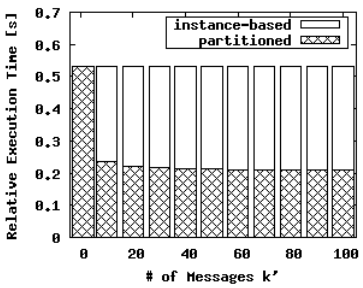


Figure 5.11: Relative Execution Time $W(P_2', k')/k'$

k'. Figure 5.11 illustrates the resulting relative execution time per message (analytical consideration) with varying batch size $k' \in \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. We observe a decreasing relative execution time per message $W(P'_2, k')/k'$ with increasing k', and that this relative execution time asymptotically tends to a lower bound.

Using the double-metric cost model in combination with its extension for message partitions, we now can compute the total execution time $W(M', k')$ and the total latency time $T_L(M', k')$ of finite message subsequences M' (with undefined length) by assuming $\lceil |M'|/k' \rceil = 1/sel$ instances of a partitioned plan (rounded up to numbers of instances).

- *Total Execution Time $W(M')$:* The estimated total execution time $\hat{W}(M')$ of a message subsequence is computed as the product of the estimated costs per instance times the number of executed plan instances with

$$\hat{W}(M', k') = \hat{W}(P', k') \cdot \left\lceil \frac{|M'|}{k'} \right\rceil, \tag{5.3}$$

  where $\hat{W}(P', k')$ is computed as sum over the sequence of extended operator costs with $\hat{W}(P', k') = \sum_{i=1}^{m} \hat{W}(o', k')$.

- *Total Latency Time $T_L(M')$:* In contrast, the estimated total latency time $\hat{T}_L(M')$ of a message subsequence is composed of the waiting time Δtw and the execution time $\hat{W}(P', k')$. Thus, we compute it depending on the comparison between Δtw and $\hat{W}(P', k')$ with

$$\hat{T}_L(M', k') = \begin{cases} \left\lceil \frac{|M'|}{k'} \right\rceil \cdot \Delta tw + \hat{W}(P', k') & \Delta tw \geq W(P', k') \\ \Delta tw + \left\lceil \frac{|M'|}{k'} \right\rceil \cdot \hat{W}(P', k') & \text{otherwise.} \end{cases} \tag{5.4}$$

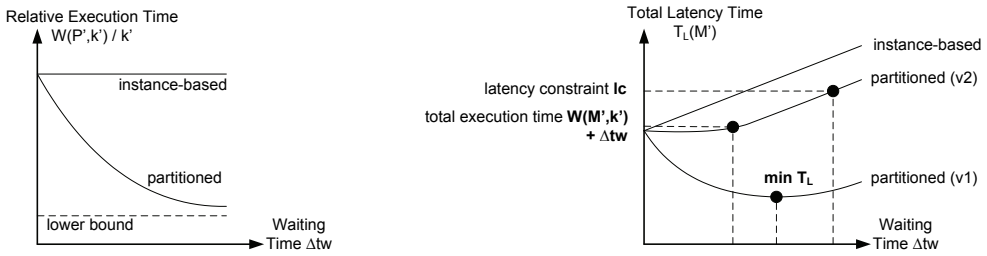
  Due to our defined validity condition, $\Delta tw < W(P', k')$ is invalid. Hence, in the following we use only the first case of Equation 5.4.

It follows directly that $\hat{W}(M', k') \leq \hat{T}_L(M', k')$, where $\hat{W}(M', k') + \Delta tw = \hat{T}_L(M', k')$ is the specific case at $\Delta tw = W(P', k')$. This means, the estimated total latency time cannot be lower than the estimated total execution time because the latency time additionally includes the waiting time. Finally, the optimization objective ϕ of the P-MFO is to minimize this total latency time function $\hat{T}_L$ under all existing constraints. In the following, we will explain how to compute the optimal waiting time Δtw that achieves this minimization for the general case of arbitrary cost models.

5.3.3 Waiting Time Computation

The intuition of computing the optimal waiting time Δtw is that the waiting time—and hence, the batch size k'—strongly influences the execution time of single plan instances. The latency time then depends on that execution time. Figure 5.12 conceptually illustrates the resulting two inverse influences that our computation algorithm exploits:

- Figure 5.12(a): For partitioned plans, an increasing waiting time Δtw causes decreasing relative execution time per message $W(P', k')/k'$ and total execution time $W(M', k')$, which are both non-linear functions that asymptotically tend towards a lower bound. In contrast, Δtw has no influence on instance-based execution.



(a) $\Delta tw \rightarrow W(P')/k'$ Influence (b) $\Delta tw \rightarrow T_L(M')$ Influence

Figure 5.12: Search Space for Waiting Time Computation

- Figure 5.12(b): On the one side, an increasing waiting time Δtw linearly increases the latency time $\hat{T}_L$ because the waiting time is directly included in $\hat{T}_L$. On the other side, an increasing Δtw causes a decreasing relative execution time and thus, indirectly decreases $\hat{T}_L$ because the execution time is included in $\hat{T}_L$. The result of these two influences, in the general case of arbitrary extended cost functions, is a non-linear total latency time function that has a local minimum (v1) or not (v2). In any case, due to the validity condition, the total latency function is defined for the closed interval $T_L(M', k') \in [W(M', k') + \Delta tw, lc]$ and hence, both global minimum and maximum exist. Given these characteristics, we compute the optimal waiting time with regard to latency time minimization and hence, throughput maximization.

For waiting time computation, we need some additional notation. We monitor the incoming message rate $R \in \mathbb{R}$ and the value selectivity $sel \in (0, 1]$ according to the partitioning attributes. For the sake of a simple analytical model, we only consider uniform value distributions of the partitioning attributes. However, it can be extended via histograms for value-based selectivities as well. The first partition will contain $k' = R \cdot sel \cdot \Delta tw$ messages. For the i-th partition with $i \geq 1/sel$, k' is computed by $k' = R \cdot \Delta tw$, independently of the selectivity sel. A low selectivity implies many partitions b_i but only few messages per partition per time unit ($|b_i| = R \cdot sel$). However, the high number of partitions b_i forces us to wait longer ($\Delta tw/sel$) until the start of execution of such a partition such that the batch size only depends on message rate R and the waiting time Δtw.

Based on the relationship between the waiting time Δtw and the number of messages per batch k', we can compute the waiting time, where $\hat{T}_L$ is minimal by

$$\Delta tw \quad | \quad \min \hat{T}_L(\Delta tw) \quad \wedge \quad \hat{W}(M', \Delta tw) \leq \hat{T}_L(M', \Delta tw) \leq lc \tag{5.5}$$

Using Equation 5.4 and assuming a fixed message rate R with $1/sel$ distinct items according to the partitioning attribute, we can substitute k' with $R \cdot \Delta tw$ and get

$$\begin{aligned} \hat{T}_L(M', k') &= \left\lceil \frac{|M'|}{k'} \right\rceil \cdot \Delta tw + W(P', k') \\ \hat{T}_L(M', R \cdot \Delta tw) &= \left\lceil \frac{|M'|}{R \cdot \Delta tw} \right\rceil \cdot \Delta tw + W(P', R \cdot \Delta tw). \end{aligned} \tag{5.6}$$

Finally, we set the cardinality of the hypothetical message subsequence to the worst case by $M' = k'/sel = (R \cdot \Delta tw)/sel$ (execution of all $1/sel$ distinct partitions, where each

partition contains k' messages), use Equation 5.6 and compute $\min \hat{T}_L$ with

$$\Delta tw \quad | \quad \min \hat{T}_L(M', R \cdot \Delta tw) \quad \text{with} \quad \begin{array}{l} \hat{T}'_L(M', R \cdot \Delta tw)_{\Delta tw} = 0 \\ \hat{T}''_L(M', R \cdot \Delta tw)_{\Delta tw \Delta tw} > 0, \end{array} \tag{5.7}$$

where $\hat{T}'_L$ and $\hat{T}''_L$ are the first and second partial deviation of the total latency function $\hat{T}_L$ with repect to Δtw. If such a local minimum exists, we check the validity condition of $(0 \leq W(P', k') \leq \Delta tw) \wedge (0 \leq \hat{T}_L \leq lc)$. If $\Delta tw < W(P', k')$, we search for $\Delta tw = W(P', k')$. Further, if $\hat{T}_L > lc$, we search for the specific Δtw with $\hat{T}_L(\Delta tw) = lc$. If such a local minimum does not exist, we use the lower border of the function interval to compute Δtw with

$$\Delta tw \quad | \quad \min \hat{T}_L(M', R \cdot \Delta tw) \quad \text{with} \quad \hat{T}_L(M', R, \Delta tw) = W(M', \Delta tw \cdot R) + \Delta tw, \tag{5.8}$$

where this lower border of $W(M', k') + \Delta tw$ is given at $\Delta tw = W(P', k')$, where $W(P', k')$ depends itself on Δtw. In dependence on the load situation, there might not be a valid $\Delta tw = W(P', k')$ with $\Delta tw \geq 0$. In this overload situation, we compute the maximum number of messages per batch k'' by

$$k'' \quad | \quad W(M', k'') + \Delta tw = T_L(M', k'') = lc. \tag{5.9}$$

In this case, the waiting time is exactly $\Delta tw = W(P'', k'')$ and thus we have a full utilization. However, due to the overload situation, we do not execute the partition with all collected messages k' but only with the k'' messages, while the $k' - k''$ messages are reassigned to the end of the partition tree (and with regard to serialized external behavior, the outrun counters are increased accordingly). This ensures lowest possible latency of single messages. Thus, we achieve highest throughput, but the input queue size increases. We use an example to illustrate this whole computation approach.

Example 5.8 (Waiting Time Computation)**.** *Recall the Example 5.7 and assume a fixed message rate* $R = 4\,\text{msg/s}$ *(overload situation for instance-based execution due to* $W(P_2) = 0.53\,\text{s}$ *), a latency constraint* $lc = 200\,\text{s}$*, and a selectivity of* $sel = 0.01$*. In order to compute* Δtw*, we use the following* $\hat{T}_L$ *function (w.r.t. Equation 5.6):*

$$\begin{aligned} \hat{T}_L(M', R \cdot \Delta tw) = &\left\lceil \frac{|M'|}{R \cdot \Delta tw} \right\rceil \cdot \Delta tw \\ &+ W(o_1) + W(o_2) + W(o_3) + (W(o_4) + W(o_5) + W(o_6)) \cdot R \cdot \Delta tw. \end{aligned}$$

We then set $M' = (R \cdot \Delta tw)/sel$ *to the worst-case of* $k' = R \cdot \Delta tw$ *messages for all* $1/sel$ *distinct partitions and thus get*

$$\begin{aligned} \hat{T}_L(M', R \cdot \Delta tw) = &\left\lceil \frac{1}{sel} \right\rceil \cdot \Delta tw \\ &+ W(o_1) + W(o_2) + W(o_3) + (W(o_4) + W(o_5) + W(o_6)) \cdot R \cdot \Delta tw. \end{aligned}$$

Based on this obtained function, we see that the deviation according to Δtw *will lead to a constant function and thus, no local minimum exist. Hence, we determine* Δtw *at* $\Delta tw = W(P'_2, k')$ *with*

$$\begin{aligned} \Delta tw &= W(P'_2, R \cdot \Delta tw) \\ \Delta tw &= W(o_1) + W(o_2) + W(o_3) + (W(o_4) + W(o_5) + W(o_6)) \cdot R \cdot \Delta tw \\ \Delta tw &= \frac{1 - (W(o_4) + W(o_5) + W(o_6)) \cdot R}{W(o_1) + W(o_2) + W(o_3)} \approx 1.8056\,\text{s}. \end{aligned}$$

Based on this waiting time, we compute the hypothetical total latency time of

$$\hat{T}_L(M', R, \Delta tw) = \left\lceil \frac{1}{sel} \right\rceil \cdot \Delta tw + W(P', R \cdot \Delta tw)$$
$$= 100 \cdot 1.8056\,\text{s} + \left(0.325 + 0.205 \cdot 4\frac{1}{\text{s}}\right) \cdot 1.8056\,\text{s} \approx 182.3656\,\text{s}.$$

Finally, we check the validity conditions of $(0 \leq W(P_2', k') \leq \Delta tw) \wedge (0 \leq \hat{T}_L \leq lc)$*, where we observe that all constraints are given*[15]*. Comparing instance-based and partitioned execution with regard to executing for example* $|M| = 1{,}000$ *messages, we reduced the total latency time from* $\hat{T}_L(M) = 1{,}000 \cdot 0.53\,\text{s} = 530\,\text{s}$ *(simplified due to overload situation) to* $\lceil 1{,}000/(4\frac{1}{\text{s}}) \cdot 1.8056\,\text{s} \rceil \cdot 1.8056\,\text{s} + 1.8056\,\text{s} = 252.784\,\text{s}$ *and the total execution time from* $530\,\text{s}$ *to* $250.978\,\text{s}$ *(*$W(M, k') = \hat{T}_L(M, k') + \Delta tw$ *because we computed* Δtw *according to the lower border of the defined interval).*

The whole computation approach is designed for the general case of arbitrary cost functions. Based on the observation that our concrete extended cost model only uses two categories of operator costs (operators that are independent of k' and operators that depend linearly on k'), we use an efficient tailor-made waiting time computation algorithm as a simplification of this computation approach. In contrast to the general solution of computing the optimal waiting time, this algorithm does not require the derivation of the latency time function.

Algorithm 5.3 Waiting Time Computation (A-WTC)

Require: rewritten plan P', message rate R, latency constraint lc, selectivity sel
1: $\Delta tw \leftarrow 0$, $k' \leftarrow 0$, $W^-(P') \leftarrow 0$, $W^+(P') \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** m **do** *// for each operator* o_i
3: **if** $W(o_i')$ is independent of k' **then**
4: $W^-(P') \leftarrow W^-(P') + W(o_i')$
5: **else**
6: $W^+(P') \leftarrow W^+(P') + W(o_i')$
7: $\Delta tw \leftarrow \frac{W^-(P')}{1 - W^+(P') \cdot R}$ *//* $\Delta tw = W(P', \Delta tw \cdot R) = W^-(P') + W^+(P') \cdot \Delta tw \cdot R$
8: $T_L \leftarrow \lceil \frac{1}{sel} \rceil \cdot \Delta tw + W(P', R \cdot \Delta tw)$
9: $k' \leftarrow R \cdot \Delta tw$
10: **if** $\neg(0 \leq W(P', k') \leq \Delta tw) \vee \neg(0 \leq \hat{T}_L \leq lc)$ **then**
11: $k' \leftarrow k' \mid W(M', k') + \Delta tw = T_L(M', k') = lc$ *// overload situation*
12: **return** $\Delta tw, R, k'$

Algorithm 5.3 illustrates this automatic waiting time computation approach for a given message rate R, the latency constraint lc, and number of distinct items sel. First, there are some initialization steps (line 1). Then, we iterate over the operators of the rewritten plan P' (lines 2-6) in order to compute the costs $W^-(P)$ that are independent of k' and the costs $W^+(P)$ that depend linearly on k'. Using a simplified Equation 5.8, we compute

[15] In this example, the result depends on the message arrival rate R. For example, $R = 10\,{}^{\text{msg}}/_{\text{s}}$ would result in an invalid waiting time $\Delta tw \approx -0.3095\,\text{s}$. In this case we would try to compute $\hat{T}_L(M', R \cdot \Delta tw) = W(M', \Delta tw \cdot R)$ but would observe an overload situation and therefore, would proceed accordingly and compute the maximum batch size k''.

the waiting time at the lower border of the defined T_L function interval (line 7). Finally, we compute the total latency time, the resulting number of messages per batch k' and check the validity condition in order to react on overload situations (line 8-11). Due to the linear classification of operator costs and the analytical determination of Δtw, this algorithm exhibits a linear complexity of $O(m)$ according to the number of operators m.

Despite the described simplification of waiting time computation, where we compute the special case $\Delta tw = W(P', k')$, it would not be sufficient to simply collect messages for the execution time of the current message partition. Most importantly, this is reasoned by horizontal (value-based) partitioning that leads to internal out-of-order execution. For high message rates in combination with many distinct partitions, the average message latency as well as the effective system output rate would degrade due to message synchronization at the outbound side. By computing the waiting time or maximal partition size, respectively, we flush oldest messages out of the system such that better average message latency times and output rates are achieved and we also reduce the effort for outbound message synchronization. Finally, we can also compute the global minimum for the general case of arbitrary cost models.

In this subsection, we have described how to compute the optimal waiting time with regard to minimizing the total latency time under a maximum latency constraint. This maximizes the message throughput with regard to a single deployed plan. With regard to multiple deployed plans this approach is applicable as well. However, changing the optimization objective to minimizing the total execution time under the maximum latency constraint can lead to an even higher throughput because this minimizes potentially overlapping execution. However, the computation approach is realized similarly as for the case of a single deployed plan despite the difference that we always try to determine the upper border (lc) of the defined T_L function interval rather than the lower border. Finally, the waiting time computation is a central aspect of the multi-flow optimization technique and of its integration into our general cost-based optimization framework.

5.3.4 Optimization Algorithm

Putting it all together, we now describe how the multi-flow optimization technique is integrated into our general cost-based optimization framework. This includes changes of the deployment process as well as the cost-based re-optimization.

The deployment process is modified such that it now additionally includes the automatic derivation of partitioning attributes, as described in Subsection 5.2. Apart from that, most aspects of the multi-flow optimization technique are integrated into the feedback loop of our cost-based optimization framework. The major issues are the derivation of partitioning schemes, the rewriting of plans and the computation of the optimal waiting time. First, according to the monitored selectivities of partitioning attributes that have been derived during the initial deployment, we derive the optimal partitioning scheme in case of multiple partitioning attributes. Second, if there are at least two attributes and if we have found a new partitioning scheme during re-optimization, we rewrite the plan according to this partitioning scheme in order to enable the execution of operations on horizontally partitioned message batches. For a single partitioning attribute, rewriting is not required because all operators can work transparently on a single partition level as described in Subsection 5.2.3. This rewriting includes also the requirement of dynamic state migration in the sense of transforming an existing partition tree that indexes collected messages from one partitioning scheme into another scheme. In order to ensure the

latency constraint and to avoid additional overhead for this transformation, we do not use state migration but use two partition trees (with the two schemes) and the two plans in parallel. New messages are enqueued into the new partition tree, while we execute message partitions from the old partition tree with the old plan until this partition tree is empty. When this termination condition is reached, we exchange plans and execute partitions from the new partition tree. Third, during cost-based optimization, we estimate the costs and compute the optimal waiting time with regard to minimizing the total latency time. Note that there are several side effects between MFO and the overall cost-based optimization. For example, there are bidirectional influences on cost estimation and plan rewriting. Therefore, during optimization the MFO technique is applied before the operator specific rewriting techniques. This is a worst-case scenario (no other optimizations applied) consideration with regard to the latency time and we benefit from subsequently applied techniques (e.g., rewriting iterations, which have been introduced by MFO, to parallel flows). Finally, the computed waiting time Δtw is used in order to asynchronously issue instances of the rewritten plan.

To summarize, we discussed how to enable the execution of horizontally partitioned message batches and how to compute the optimal waiting time with regard to the total latency time minimization that implies message throughput maximization. Furthermore, we explained how the overall multi-flow optimization technique is seamlessly integrated into the general cost-based optimization framework.

5.4 Formal Analysis

In this section, we now formally analyze our solution of horizontal message queue partitioning and waiting time computation according to the defined multi-flow optimization problem (P-MFO, see Definition 5.2). This includes (1) the optimality analysis with regard to latency time minimization as well as the two additional restrictions of (2) the maximum soft latency constraint for single messages and (3) the serialized external behavior.

5.4.1 Optimality Analysis

First of all, we analyze the optimality with regard to the computed waiting time Δtw. In detail, we discuss (1) the monotonically non-increasing execution time function that (2) asymptotically tends towards a lower bound as shown in Figure 5.13.

Monotonically Non-Increasing Relative and Total Execution Time

Based on the extended cost model, we can give an optimality guarantee for $W(P', k')$ with regard to the computed waiting time.

Theorem 5.1 (Optimality of Partitioned Execution)**.** *The horizontal message queue partitioning solves the P-MFO with optimality guarantees of* $\min T_L(M')$ *and monotonically non-increasing total execution time of*

$$\frac{W(P', k')}{k'} \cdot |M'| \leq \frac{W(P', k'-1)}{k'-1} \cdot |M'| \leq W(P) \cdot M', \quad \text{where } k' > 1. \tag{5.10}$$

Proof. We compute the waiting time Δtw, where $\min T_L(M')$ under the given restrictions. This determines the batch size $k' = R \cdot \Delta tw$ and ensures optimal throughput because the

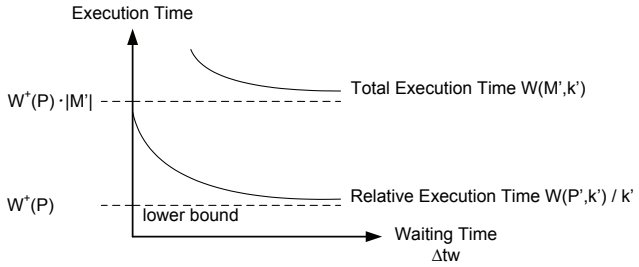


Figure 5.13: Monotonically Non-Increasing Execution Time with Lower Bound

total execution time is decreased as long as we benefit from it with regard to the total latency time. The execution time $W(P', k')$ is computed by

$$W(P', k') = W^-(P') + W^+(P') \cdot k', \tag{5.11}$$

where $W^+(P')$ denote the costs (execution time) of operators that do not benefit from partitioning, while $W^-(P')$ denotes the costs of operators that benefit from partitioning (independent of k'). As a result, in the worst case, the execution time $W(P', k')$ increases linearly with increasing k'. Thus, the relative execution time $W(P', k')/k'$ is a *monotonically non-increasing* function with

$$\forall k', k'' \in [1, |M'|] : k' < k'' \Rightarrow \frac{W(P', k')}{k'} \geq \frac{W(P', k'')}{k''}. \tag{5.12}$$

If we now fix a certain $|M'|$, this also implies that the total execution time $W(M', k')$ is a monotonically non-increasing function with

$$W(M', k') = \frac{W(P', k')}{k'} \cdot |M'|. \tag{5.13}$$

Then, it follows directly that

$$\begin{aligned} \frac{W(P', k')}{k'} &\leq \frac{W(P', k'-1)}{k'-1} \leq W(P) \\ \frac{W(P', k')}{k'} \cdot |M'| &\leq \frac{W(P', k'-1)}{k'-1} \cdot |M'| \leq W(P) \cdot |M'|. \end{aligned} \tag{5.14}$$

Hence, Theorem 5.1 holds. □

This result of monotonic non-increasing relative and total execution time functions is illustrated in Figure 5.13 as (strictly) monotonically decreasing function. However, due to (1) integer batch sizes k' and (2) a constant function in the case, where no operator benefits from partitioning, the functions are, in general, monotonic non-increasing.

Lower Bound of Relative and Total Execution Time

In analogy to *Amdahl's law*, where the fraction of a task (execution time) that cannot be executed in parallel determines the upper bound for the reachable speedup, we compute the lower bound of the relative and total execution times. The existence of this lower bound was already discussed in Section 5.3. Now, we investigate this lower bound analytically.

Theorem 5.2. *The lower bound of relative and total execution times $W(P', k')/k'$ and $W(M', k')/k'$ is given by $W^+(P')$ and $W^+(P') \cdot |M'|$, respectively, as the costs that linearly depend on k'.*

Proof. Recall that the execution time $W(P', k')$ is computed by

$$W(P', k') = W^-(P) + W^+(P) \cdot k', \tag{5.15}$$

where, $W^-(P', k')$ is independent of k' by definition. If we now use the relative execution time $W(P', k')/k'$ and let k' tend to ∞ with

$$\begin{aligned} \lim_{k' \to \infty} \frac{W(P', k')}{k'} &= W^+(P') \text{ with } \frac{W(P', k')}{k'} = \frac{W^-(P)}{k'} + \frac{W^+(P) \cdot k'}{k'} \\ \lim_{k' \to \infty} \frac{W(M', k')}{k'} &= W^+(P') \cdot |M'| \\ \text{with } \frac{W(M', k')}{k'} &= \left(\frac{W^-(P')}{k'} + \frac{W^+(P') \cdot k'}{k'} \right) \cdot |M'|, \end{aligned} \tag{5.16}$$

we see that $W(P', k')/k'$ asymptotically tends to $W^+(P)$ and thus, it represents the lower bound of the relative execution time, while the lower bound of the total execution time is $W^+(P) \cdot |M'|$. Hence, Theorem 5.2 holds. □

These lower bounds are shown in Figure 5.13 and they are the reason why minimizing the total latency time leads to maximal message throughput because the additional benefit in terms of lower execution time decreases with increasing waiting time.

5.4.2 Maximum Latency Constraint

Beside the property of optimality with regard to the message throughput, our waiting time computation ensures that the restriction of the maximum latency constraint holds, in expectation, at the same time as well.

Theorem 5.3 (Soft Guarantee of Maximum Latency)**.** *The waiting time computation ensures that—for a given fixed message rate R—the latency time of a single message $T_L(m_i)$ with $m_i \in M'$ will, in expectation, and for non-overload situations, not exceed the maximum latency constraint lc with $T_L(m_i) \leq lc$.*

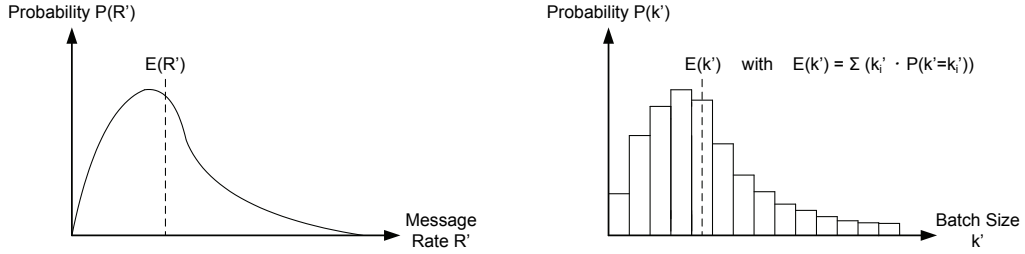
Proof. In the worst case, $1/sel$ distinct messages m_i arrive simultaneously in the system. Hence, the highest possible latency time $T_L(m_i)$ is given by the total latency time $1/sel \cdot \Delta tw + W(P', k')$. Due to our validity condition of $\hat{T}_L(M') \leq lc$ with $|M'| = k'/sel$, we need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

$$\begin{aligned} \frac{1}{sel} \cdot \Delta tw + W(P', k') &\leq \left\lceil \frac{|M'|}{k'} \right\rceil \cdot \Delta tw + W(P', k') \\ \frac{1}{sel} \cdot \Delta tw &\leq \frac{|M'|}{k'} \cdot \Delta tw. \end{aligned} \tag{5.17}$$

If we now substitute $|M'|$ by k'/sel (in the sense that the cardinality $|M'|$ is equal to the number of partitions $1/sel$ times the cardinality of a partition k'), we get

$$\frac{1}{sel} \cdot \Delta tw \leq \frac{|M'|}{k'} = \frac{1}{sel} \cdot \Delta tw. \tag{5.18}$$

Thus, for the worst case, $T_L(m_i) = lc$, while for all other cases, $T_L(m_i) \leq lc$ is true. Hence, Theorem 5.3 holds. □



(a) Message Arrival Rate R' (b) Batch Size k'

Figure 5.14: Waiting Time Computation With Skewed Message Arrival Rate

So far, we have assumed a fixed message rate R. Now, we formally analyze the influence of the assumption of a stochastic arrival rate R' that exhibits a potentially skewed probability distribution $\mathcal{D}$ such as the Poisson distribution. Figure 5.14(a) illustrates the probability of a specific arrival rate (for a continuous arrival rate distribution function), while Figure 5.14(b) illustrates the resulting batch sizes k'. Here, $\mathcal{D}$ can be a discrete or continuous function, where the expected value $E(R')$ is computed as the probability-weighted integral of continuous values or as the probability-weighted sum of discrete values, respectively. For example, the discrete Poisson distribution is relevant because Xiao et al. argued that the arrival process of workflow instances is typically Poisson-distributed [XCY06]. The main difference is that we now include uncertainty—in the form of an arbitrary message rate—into the $\min \hat{T}_L$ computation because until now, we have used $k' = R \cdot \Delta tw$ as batch size estimation. For $\mathcal{D} = poisson$, the fixed arrival rate R is substituted with an uncertain arrival rate R' such that

$$k' = R' \cdot \Delta tw \text{ with } P_R(R' = r) = \frac{R^r}{r!} \cdot e^{-R}, \tag{5.19}$$

with an expected value of $E(R') = R$. Due to the introduced uncertainty, we need to extend Theorem 5.3 to skewed probability distributions functions.

Theorem 5.4 (Extended Soft Guarantee of Maximum Latency)**.** *The waiting time computation ensures that—for a given uncertain message rate R' with skewed probability distribution function $\mathcal{D}$—the latency time of a single message $T_L(m_i)$ with $m_i \in M'$ will, in expectation, and for non-overload situations, not exceed the maximum latency constraint lc with $T_L(m_i) \leq lc$.*

Proof. Recall the worst case of $T_L(M') = lc$ with

$$T_L(M') = \left\lceil \frac{|M'|}{k'} \right\rceil \cdot \Delta tw + W(P', k'). \tag{5.20}$$

There, the over- and underestimation of batch size k' does affect the execution time $W(P', k')$. Hence, the worst case is given, where $\Delta tw = W(P', k')$. With regard to the uncertain arrival rate, the average discrete over- and under-estimation of k' is, in expectation, equal with

$$\sum_{k'_i \leq E(k')} \left(k'_i \cdot P(k' = k'_i)\right) = \sum_{k'_i \geq E(k')} \left(k'_i \cdot P(k' = k'_i)\right) \tag{5.21}$$

because $E(k') = \sum_{i=1}^{\infty} k'_i \cdot P(k' = k'_i)$. In order to show that the latency constraint lc is, in expectation, not exceeded by skewed arrival rate distributions, we need to show that the effects of overestimation $W(P', k' - 1) - W(P', k')$ (lower execution time) amortize the effects of underestimation $W(P', k' + 1) - W(P', k')$ (higher execution time). The composed effect $\Delta W(P')$ is computed by

$$\begin{aligned}\Delta W(P') &= \big(W(P', k' - 1) - W(P', k')\big) + \big(W(P', k' + 1) - W(P', k')\big) \\ &= \big(W^-(P) + W^+(P) \cdot (k' - 1) - (W^-(P) + W^+(P) \cdot k')\big) \\ &\quad + \big(W^-(P) + W^+(P) \cdot (k' + 1) - (W^-(P) + W^+(P) \cdot k')\big) = 0\end{aligned} \tag{5.22}$$

Thus, for the worst case, the condition $T_L(M') = lc$ holds in expectation. Hence, Theorem 5.4 holds. □

This extended soft guarantee of maximum latency holds for both right-skewed and left-skewed distributions functions due to the equal average over- and under-estimation. However, it is important to note that these guarantees of maximum latency are soft constraints that hold, in expectation, and for non-overload situations, while changing workload characteristics in combination with a long optimization interval Δt might lead to temporarily exceeding the latency constraint.

5.4.3 Serialized External Behavior

According to the requirement of serialized external behavior, we additionally might need to serialize messages at the outbound side. We analyze once again the given maximum latency guarantee with regard to arbitrary serialization concepts.

Theorem 5.5 (Serialized Behavior)**.** *Theorems 5.3 (Soft Guarantee of Maximum Latency) and 5.4 (Extended Soft Guarantee of Maximum Latency) also hold for serialized external behavior.*

Proof. We need to prove that the condition $T_L(m_i) \leq \hat{T}_L(M') \leq lc$ is true even in the case, where we have to serialize the external behavior. Therefore, recall the worst case (Theorem 5.3), where the latency time is given by

$$T_L(m_i) = \frac{1}{sel} \cdot \Delta tw + W(P', k'). \tag{5.23}$$

In that case, the message m_i has not outrun other messages such that no serialization time is required. For all other messages that exhibit a general latency time of

$$T_L(m_i) = \left(\frac{1}{sel} - x\right) \cdot \Delta tw + W(P', k'), \tag{5.24}$$

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

$$\begin{aligned}T_L(m_i) &= \left(\frac{1}{sel} - x\right) \cdot \Delta tw + W^*(P', k') + x \cdot \Delta tw + W(P', k') \\ &= \frac{1}{sel} \cdot \Delta tw + W(P', k').\end{aligned} \tag{5.25}$$

Thus, $T_L(m_i) \leq \hat{T}_L(M') \leq lc$ is true due to the subsumtion of $W^*(P', k')$ by Δtw because the waiting time is longer than the execution time, for the valid case of $\Delta tw \geq W(P', k')$. This is also true for arbitrary serialization concepts. Hence, Theorem 5.5 holds. □

Based on this formal analysis, we can state that the introduced waiting time computation approach (1) optimizes the message throughput by minimizing the total latency time of a message subsequence, and ensures the additional restrictions of (2) a maximum latency time constraint for single messages, and (3) the serialized external behavior.

5.5 Experimental Evaluation

In this section, we present experimental evaluation results with regard to the three evaluation aspects of (1) optimization benefits/scalability, (2) optimization overheads, and (3) latency guarantees under certain constraints. In general, the evaluation shows that:

- Message throughput improvements are yielded by minimizing the total latency time of message sequences. Compared to the unoptimized case, the achieved relative improvements decrease with increasing data sizes for some plans, while they stay constant for other plans. In contrast, the relative improvement increases with increasing batch sizes.
- The runtime optimization overhead for deriving partitioning schemes, rewriting plans, and computing the optimal waiting time is moderate. In addition, the overhead for horizontal partitioning of inbound message queues is, for moderate numbers of distinct items, fairly low compared to commonly used transient message queues.
- Finally, the theoretical maximum latency guarantees for arbitrary distribution functions also hold under experimental investigation. This stays true under the constraint of serialized external behavior (additional serialization at the outbound side) as well.

The detailed description of our experimental findings is structured as follows. First, we evaluate the end-to-end throughput improvement as well as the overheads of periodical reoptimization. Second, we present scalability results with regard to increasing data sizes as well as increasing batch sizes. Third, we analyze in detail the execution time with regard to influencing factors such as message rate, selectivities, waiting time, and batch sizes. Fourth, we present the influences on the latency time of single messages with and without serialized external behavior and with arbitrary message rate distribution functions. Fifth and finally, we discuss the runtime overhead of horizontal message queue partitioning.

Experimental Setting

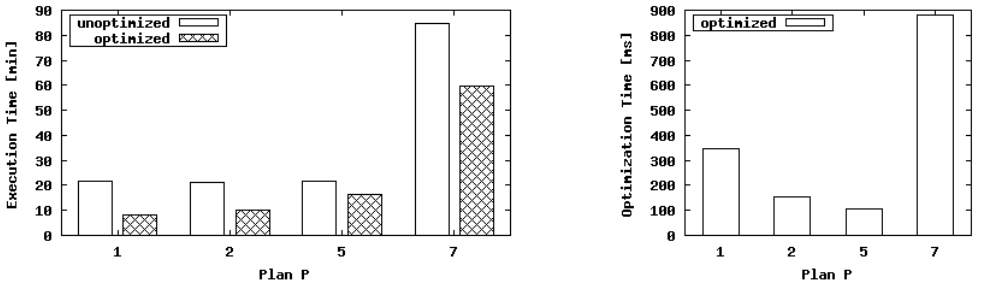
We implemented the approach of MFO via horizontal partitioning within our Java-based `WFPE` (workflow process engine) and integrated it into our general cost-based optimization framework. This includes the partitioned message queue (partition tree and hash partition tree), slightly changed operators (partition-awareness) as well as the algorithms for deriving partitioning attributes (A-DPA), rewriting of plans (A-MPR) and automatic waiting time computation (A-WTC).

We ran our experiments on the same platform as described in Section 3.5 and we used synthetically generated data sets. As integration flows under test, we use all asynchronous,

data-driven integration flow use cases (plans P_1, P_2, P_5, and P_7), which have been described in Section 2.4. Furthermore, we used the following scale factors: the number of messages $|M|$, the message rate R, the selectivity according to the partitioning attribute sel, the batch size k', the message rate distribution function $\mathcal{D}$, the maximum latency constraint lc, and the data size d of input messages (in 100 kB).

End-to-End Comparison and Optimization Benefits

First of all, we investigate the end-to-end optimization benefit achieved by multi-flow optimization and the related optimization overhead. We compared the multi-flow optimization with no-optimization, while all other optimization techniques have been disabled. Similar to the use case comparison in Section 3.5 and 4.6, we executed 20,000 plan instances for each asynchronous, data-driven example plan (P_1, P_2, P_5, and P_7) and for both execution models. We reused the same workload configuration as already presented (without correlations and without workload changes). Furthermore, we fixed the cardinality of input data sets to $d = 1$ (100 kB messages), an optimization interval of $\Delta t = 5$ min, a sliding window size of $\Delta w = 5$ min and `EMA` as the workload aggregation method. With regard to multi-flow optimization, we did not use the computed waiting time but directly restricted the batch size to $k' = 10$ in order to achieve comparable results across the different plans.



(a) Cumulative Execution Time (b) Cumulative Optimization Time

Figure 5.15: Use Case Comparison of Multi-Flow Optimization

Figure 5.15(a) shows the resulting total execution times. To summarize, we consistently observe significant execution time reductions that have been achieved as follows:

- P_1: The plan P_1 benefits from MFO in several ways. First, the `Switch` operator o_2 is executed once for a message batch because the switch expression attribute `/material/type` is used as the only partitioning attribute. Furthermore, the `Assign` operators o_4, o_6, and o_8 are also executed only once because the result is exclusively used by the partition-aware `Invoke` operators o_7, and o_9. These writing `Invoke` operators show additional benefit because a single operator instance is used to process all messages of a batch. Overall, this achieves a throughput improvement of 62%.
- P_2: The plan P_2 mainly benefits from executing the `Invoke` operator o_3 and the predecessor `Assign` operator o_2 only once for a whole partition. There, the predicate part `/resultsets/resultset/row/A1_Custkey` is used as the partitioning attribute. Additional benefit is achieved by the final `Assign` and `Invoke` operators o_5 and o_6. In total, an improvement of 53% has been achieved.

- P_5: The plan P_5 shows the lowest benefit. There, the `Switch` operator o_5 is executed only once for a batch because the switch expression attribute `/resultsets/resultset/row/A1_Orderdate` is used as partitioning attribute. Additional benefit comes from the `Assign` and `Invoke` operators o_8 and o_9. However, we achieved only an improvement of 25% because the `Selection` operators in front of the operators that benefit from partitioning consume most of the time and significantly reduces the cardinality of intermediate results.
- P_7: In contrast to the other plans, plan P_7 does not contain any partitioning attribute candidate. Therefore, a system partitioning with $sel = 1.0$ is used (this case is similar to the time-base batch creation strategy but without the drawback of distinct messages in a batch). Many operators benefit from partitioning. First, the queries to external systems are prepared only once (`Assign` operators o_3, o_6, o_9, and o_{11}). Second, also the external queries and the subsequent schema mapping is only executed once (`Invoke` operators o_4, o_7, o_{10}, and o_{12} as well as `Translation` operators o_5, o_8, o_{13}). Third, additional benefit is achieved by the final `Assign` and `Invoke` operators o_{18} and o_{19}. In total, this led to an improvement of 30%.

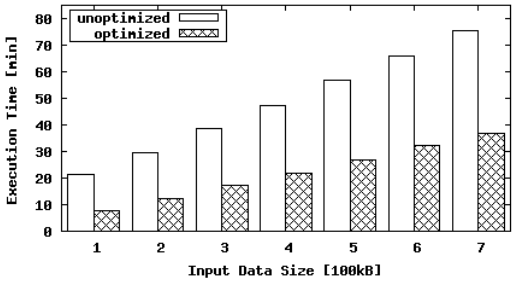
Figure 5.15(b) illustrates the optimization overhead imposed by the cost-based multi-flow optimization. This includes the derivation of partitioning attributes, the creation of a partitioning scheme, the plan rewriting, as well as the continuous waiting time computation. Essentially, we observe that the overhead is moderate, where the differences are mainly reasoned by the different numbers of operators.

Finally, based on the observed results, we can conclude that the multi-flow optimization technique can be used by default (if small additional latency for single messages is acceptable) because the throughput improvements clearly amortize the optimization overhead. This is true for arbitrary asynchronous, data-driven integration flows because each flow has at least one combination of writing `Assign` and `Invoke` operators.

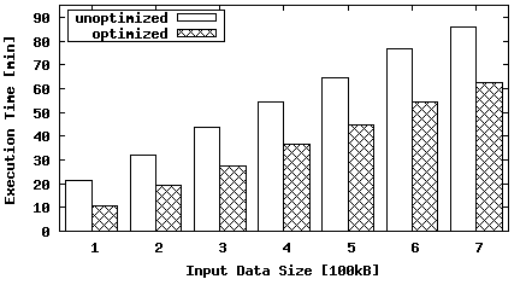
Scalability

We now investigate the scalability of plan execution, which includes (1) the scalability with increasing input data sizes and (2) the scalability with increasing batch sizes.

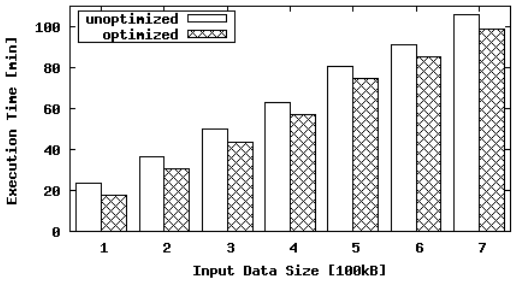
First, we used our example plans in order to investigate the scalability of optimization benefits with increasing input data size. We reused the scalability experiment with increasing data size from Section 3.5. In contrast to the already presented scalability results, we now disabled all optimization techniques except multi-flow optimization. In detail, we executed 20,000 plan instances for the plans P_1, P_2, P_5, and P_7 and compared the optimized plans with their unoptimized counterparts varying the input data size $d \in \{1, 2, 3, 4, 5, 6, 7\}$ (in 100 kB). Again, we varied the input data size of these plans (the size of the received message) only but did not change the size of externally loaded data. Further, we fixed a batch size of $k' = 10$, an optimization interval of $\Delta t = 5\,\text{min}$, a sliding window size of $\Delta w = 5\,\text{min}$ and `EMA` as the workload aggregation method. The results are shown in Figure 5.16. In general, the plans scale with increasing data size but with a decreasing relative improvement. With regard to the different plans, we observe different scalability behavior. The plan P_1 scales best with increasing data size and shows almost constant relative improvement because this plan mainly benefits from reduced costs for writing interactions that linearly depend on the data size. Further, also plan P_2 shows good scalability with increasing data size. However, the relative improvement is decreasing because



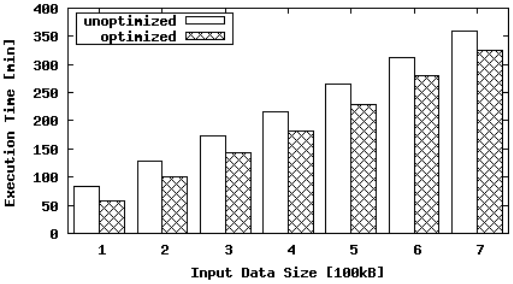
(a) Plan P_1



(b) Plan P_2

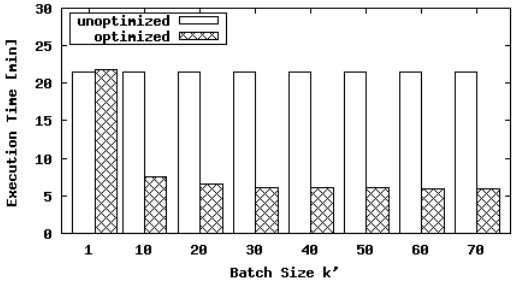


(c) Plan P_5

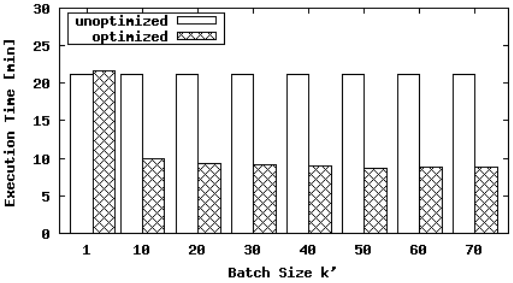


(d) Plan P_7

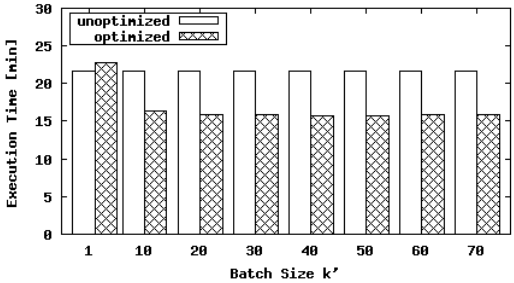
Figure 5.16: Use Case Scalability Comparison with Varying Data Size d



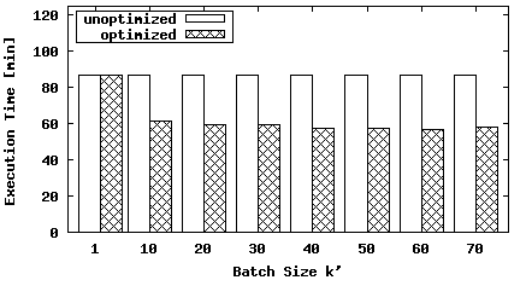
(a) Plan P_1



(b) Plan P_2



(c) Plan P_5



(d) Plan P_7

Figure 5.17: Use Case Scalability Comparison with Varying Batch Size k'

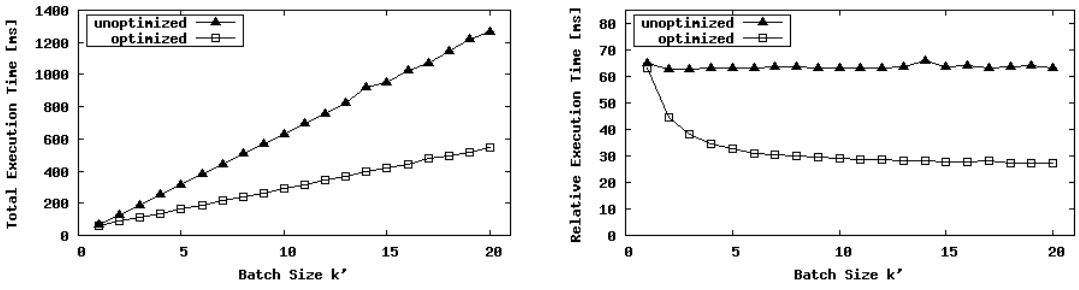
the size of loaded data (that would have an influence on the benefit) was not changed. In contrast, plan P_5 shows only a constant absolute improvement, which resulted in a

decreasing relative improvement because it mainly benefits from the reduced costs for the `Switch` operator. However, this operator is executed after several `Selection` operators that significantly reduced the amount of input data, which led to this almost constant absolute improvement. Similarly, also plan P_7 shows a decreasing relative improvement because the size of external data was not changed. In conclusion, the scalability with increasing data size strongly depends on how a plan benefits from partitioning.

Second, we investigated the scalability with increasing batch sizes k'. There, we reused our example plans P_1, P_2, P_5, and P_7 and the workload configuration from the previous experiment. For each example plan and compared the multi-flow optimization with no-optimization varying the batch size $k' \in \{1, 10, 20, 30, 40, 50, 60, 70\}$. Figure 5.17 shows the results of this experiment. Essentially, we make three major observations. First, the overhead for executing single-message-partitions is marginal. Despite the fact that MFO theoretically cannot decrease the performance, there is some overhead due to horizontal partitioning at the inbound side and additional message abstraction layers. However, the experiments show that this overhead is negligible. As a result the multi-flow optimization is robust in the sense that it ensures predictable performance even in special cases, where we do not benefit from partitioning. Second, the theoretically analyzed monotonically non-increasing total execution time function with increasing batch size and the existence of a lower bound of the total execution time do also hold under experimental evaluation. Third, we observe that the lower this lower bound (the higher the optimization potential), the higher the batch size that is required until we asymptotically tend to this lower bound (e.g., compare Figure 5.17(a) and Figure 5.17(c)). This effect is reasoned by the higher relative amount of time that is logically shared among messages.

Execution Time

Until now we have evaluated the optimization benefit and scalability of multi-flow optimization using restricted batch sizes k'. In this subsection, we investigate in detail the inter-influences between arbitrary message arrival rates R, waiting times Δtw, partitioning attribute selectivities *sel* and the resulting batch size k'. In addition, we evaluate the effects of the resulting batch size k' on the plan execution time $W(P', k')$.



(a) Execution Time $W(P_2', k')$ (b) Relative Execution Time $W(P_2', k')/k'$

Figure 5.18: Execution Time $W(P_2', k')$ with Varying Batch Size k'

First, we evaluated the execution time of the resulting plan instance for message partitions compared to the unoptimized execution. We executed instances of plan P_2 with varying batch size $k' \in [1, 20]$, where we measured the total execution time $W(P_2', k')$ (Figure 5.18(a)) and computed the relative execution time $W(P_2', k')/k'$ (Figure 5.18(b)).

For comparison, the unoptimized plan was executed k' times and we measured the total execution time $W(P_2) \cdot k'$. This experiment has been repeated 100 times. Both unoptimized and optimized execution show a linear scalability with increasing batch size k' with the difference that for optimized execution, we observe a logical y-intercept that is higher than zero. As a result, the unoptimized execution shows a constant relative execution time, while the optimized execution shows a relative execution time that decreases with increasing batch size and that tends towards a lower bound, which is given by the partial plan costs of operators that do not benefit from partitioning. It is important to note that (1) even for one-message-partitions the overhead is negligible and (2) that even small numbers of messages within a batch significantly reduce the relative execution time.

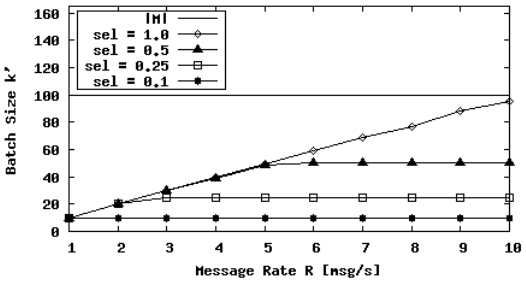


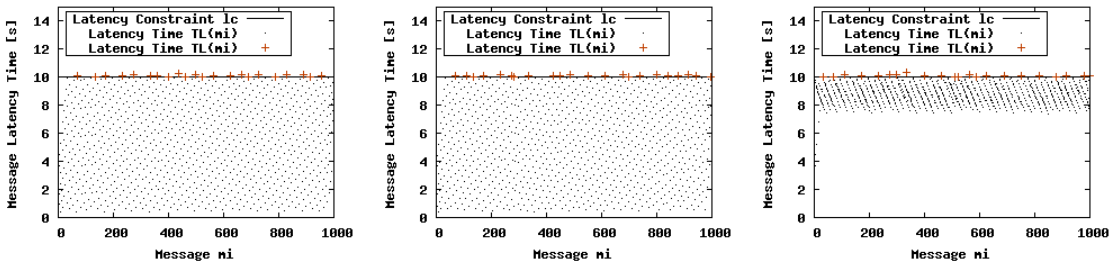
Figure 5.19: Varying R and sel

Second, we evaluated the batch size k' according to different (fixed interval) message rates R, and selectivities sel in order to validate our assumptions about the batch size estimation of $k' = R \cdot \Delta tw$ under the influence of message queue partitioning. We processed $|M| = 100$ messages with plan instances of P_2, where all sub experiments were repeated 100 times with fixed waiting time of $\Delta tw = 10\,\text{s}$. Figure 5.19 shows the influence of the message rate R on the average number of messages per batch. We observe (1) that the higher the message rate, the higher the number of messages per batch, and (2) that the selectivity determines the reachable upper bound. However, until this upper bound, the batch size is independent of the selectivity because for higher selectivities we wait longer ($\Delta tw \cdot 1/sel$) until partition execution. Similarly, also an increasing waiting time showed the expected behavior of linearly increasing batch sizes until the upper bound is reached.

Latency Time

Furthermore, we evaluated the latency influence of MFO. While the total latency time of message sequences is directly related to the throughput and thus reduced anyway, the latency time of single messages needs further investigation. In this subsection, we analyze the given maximum latency guarantee and latency times in overload situations.

In detail, we executed $|M| = 1{,}000$ messages with plan P_2 using a maximum latency constraint of $lc = 10\,\text{s}$ and measured the latency time $T_L(m_i)$ of single messages m_i. There, we fixed a selectivity of $sel = 0.1$, a message arrival rate of $R = 5\,\text{msg/s}$, and used different messages arrival rate distributions (fixed, poisson) as well as analyzed the influence of serialized external behavior. In order to discuss the worst-case consideration, we computed the waiting time $\Delta tw \mid T_L(M' = k'/sel) = lc$ (that is typically only used as a deescalation strategy). Here, Δtw was computed as 981.26 ms because in the worst case there are $1/sel = 10$ different partitions plus the execution time of the last partition. Note that the selectivity has no influence on the variance of message latency times but on



(a) Fixed w/p SEB (b) Poisson w/o SEB (c) Fixed w/ SEB

Figure 5.20: Latency Time of Single Messages $T_L(m_i)$

the computed waiting time (the higher the number of distinct items, the lower the waiting time). For both message arrival rate distribution functions $\mathcal{D} = \texttt{fixed}$ (see Figure 5.20(a)) and $\mathcal{D} = \texttt{poisson}$ (see Figure 5.20(b)), the constraint is rarely exceeded. Essentially, the latency of single messages varies from almost zero to the latency constraint, where the few missed constraints (illustrates as plus in the plots) are caused by variations of the execution time. Note that the latency constraint is explicitly a soft constraint, where we guarantee that it is not exceeded with statistical significance. The reason is that we compute the waiting time based on our cost estimation. If the real execution costs vary slightly around this estimate, there exist cases where the constraint is slightly exceeded as well. Furthermore, the constraint also holds for serialized external behavior (SEB), where all messages show more similar latency times (see Figure 5.20(c)) due to serialization at the outbound side. Thus, there is a lower variance of the latency time of single messages. Note that this is a worst-case scenario. Typically, we use a waiting time Δtw at $W(P', k') = \Delta tw$ that results in much lower latency time for single messages.

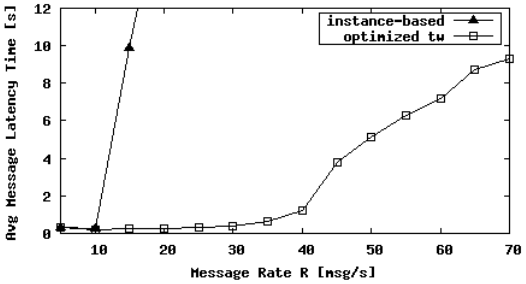


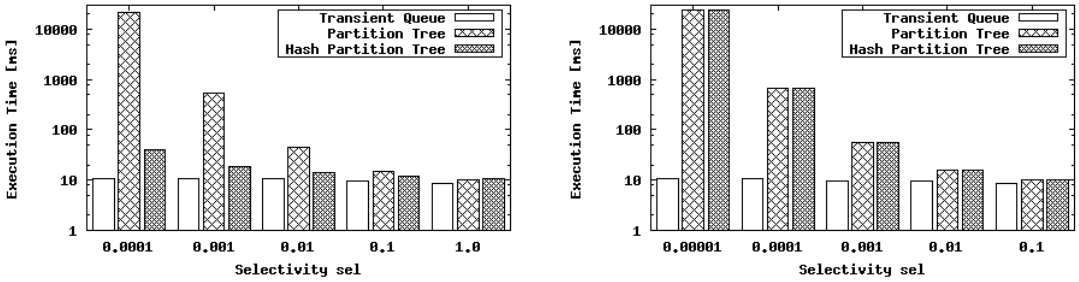
Figure 5.21: Latency in Overload Situations

The maximum latency constraint of single messages cannot be ensured if we are in overload situations. Therefore, we investigated the influence of the message rate R on the average latency time of single messages $T_L(m_i)$. We executed $|M| = 1{,}000$ messages with a selectivity of $sel = 1.0$ for unoptimized and optimized plans varying the message rate $R \in [5\,\text{msg/s}, 70\,\text{msg/s}]$. Figure 5.21 illustrates the results. For low message rates (no overload situation), we observe a minimal average latency of about 220 ms for both plans. For optimized plans, our optimizer achieved this by adjusting the waiting time to the execution time of plan instances with batch size one. After the message rate exceeded the execution time of the unoptimized plan, the average latency rapidly increases due to growing input message queues and the related waiting time. In contrast, for MFO,

the waiting time was adjusted according to the increased message rate, which led to throughput improvements such that the optimized plan was able to process much higher message rates[16]. Due to the lower bound of relative execution times, also the optimized plan cannot cope with the increased message rate after a certain point. However, we observe a slower degradation of the average latency due to more messages per batch, decreasing relative execution time and thus, increased throughput.

Optimization Overhead

In addition to the end-to-end comparison, which already included all optimization overheads, we now investigate the runtime overhead in more detail. Most importantly, the partitioned enqueue operation depends on the selectivity of partitioning attributes. Therefore, we analyzed the overhead of the (hash) partition tree compared to the commonly used transient message queue (no partitioning) with and without serialized external behavior.



(a) Without Serialized External Behavior (SEB) (b) With Serialized External Behavior (SEB)

Figure 5.22: Runtime Overhead for Enqueue Operations with Different Message Queues

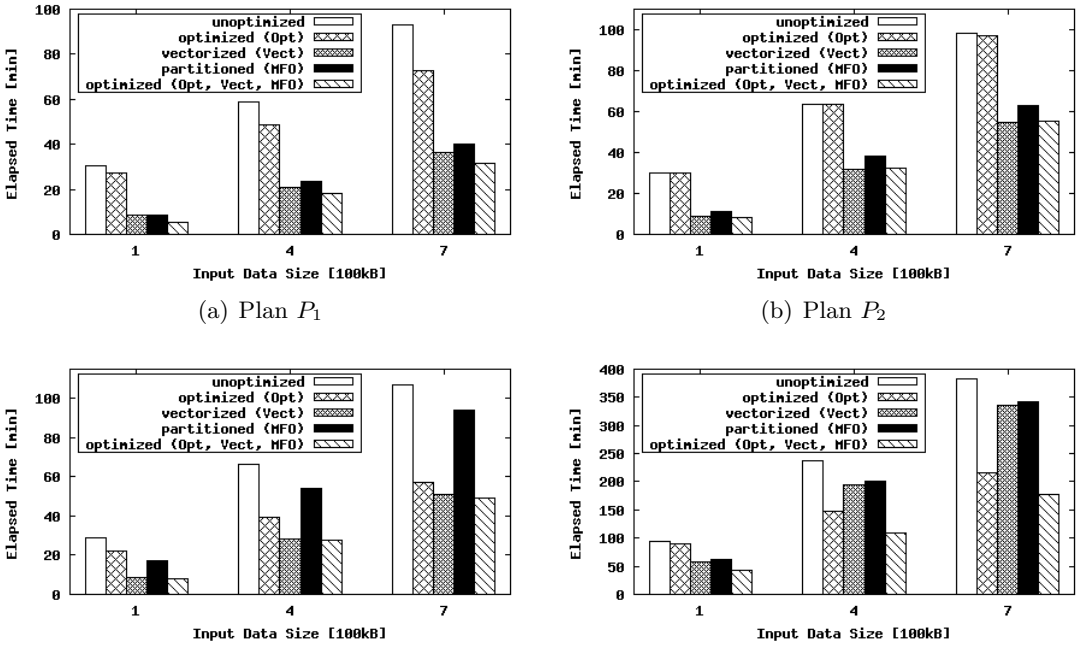
We enqueued 20,000 messages (as required for the end-to-end comparison experiment) with varying selectivities $sel \in \{0.0001, 0.001, 0.01, 0.1, 1.0\}$ of the partitioning attribute and measured the total execution time. This experiment was repeated 100 times. Figure 5.22 illustrates the results of this experiment using a log-scaled y-axis. Essentially, we see that without serialized behavior (SEB), both the transient queue and the hash partition tree show constant execution time with regard to the selectivity and the overhead of the hash partition tree is negligible. In contrast, the execution time of the normal partition tree linearly increases with decreasing selectivity due to the linear probing over all existing partitions for each enqueued message. Furthermore, we observe a similar behavior under serialized external behavior except the fact that also the execution time of the hash partition tree increases linearly with decreasing selectivity. This is caused by the required counting of outrun messages (counter-based serialization strategy), where we need to scan over all partitions in order to determine this counter for each enqueued message. However, for the whole comparison scenario, where 20,000 messages have been enqueued, the overhead for this worst-case was 23.7 s (for 10,000 distinct partitions) which is negligible compared to the achieved throughput. The enqueue operation is executed asynchronously and hence, does not directly reduce the message throughput until a break-even point is

[16] When further increasing the message rate, the average latency of both, unoptimized and optimized execution, tend to different upper bounds that are determined by the case, where all $|M|$ messages arrive simultaneously. For a message rate of 100 $^{msg}/_{s}$, we observed average latency times of 40.2 s (unoptimized) and 11.7 s (optimized).

reached, where the message rate is too high. For example, consider a message rate of $R = 20\,\text{msg/s}$, this break-even point occurs at $\approx 422{,}000$ distinct partitions in the queue.

Side Effects of Optimization Techniques

Putting it all together, we conducted an additional experiment in order to evaluate the influences between MFO, vectorization, and the other cost-based optimization techniques.



(a) Plan P_1 (b) Plan P_2

(c) Plan P_5 (d) Plan P_7

Figure 5.23: Use Case Scalability Comparison with Varying Data Size d

We reused our scalability experiment with increasing data size. In contrast to the already presented scalability results, we now use the different cost-based optimization techniques from Chapter 3-5 in combination with each other. In detail, we executed 20,000 plan instances for the plans P_1, P_2, P_5, and P_7 and compared the optimized plans with their unoptimized versions varying the input data size $d \in \{1, 4, 7\}$ (in 100 kB). In contrast to all other experiments of this chapter, we measured the elapsed scenario time (the total latency time of the message sequence) rather than the total execution time because for vectorized execution, the execution times of single plan instances cannot be aggregated due to overlapping message execution (pipeline semantics). Furthermore, we varied the input data size of these plans (the size of the received message) but did not change the size of externally loaded data. We fixed a batch size of $k' = 10$, an optimization interval of $\Delta t = 5\,\text{min}$, a sliding window size of $\Delta w = 5\,\text{min}$ and `EMA` as the workload aggregation method. The results (total elapsed time) are shown in Figure 5.23. Essentially, we observe two major effects. First, the application of all optimization techniques consistently shows the highest performance compared to single optimization techniques. Only plan P_2 performed slightly worse when using full optimization compared to vectorization only, because the operators that benefit from partitioning are not part of the most time-consuming bucket and thus, MFO introduces additional latency (although it reduces the

total execution time), while not reducing the total latency time. However, for a data size of $d = 1$ as an example, we achieved significant overall relative improvements of 82% (P_1), 72% (P_2), 74% (P_5), and 55% (P_7). Second, we observe that typically, the highest optimization benefits are achieved by vectorization and multi-flow optimization, where these plans (P_1, P_2, and P_5) do not have a high CPU utilization in the unoptimized case. Hence, the optimization benefits are partially overlapping. In contrast, for plans such as P_7, where the local processing steps dominate the execution time (high CPU utilization), the standard cost-based optimiztaion techniques have higher influence. In this case, the different optimization benefits are not overlapping and hence the joint application achieves significant improvements. Furthermore, we see different scalability of the different optimization techniques with incresing data size according to the used plan. The application of all optimization techniques balances these effects such that finally, we observe good scalability with increasing data size for all plans with almost constant improvement. We can conclude that, especially, with regard to the scalability and the maximum benefit, it is advantageous to use all optimization techniques in combination.

Finally, we can state that MFO achieves significant throughput improvement by accepting moderate additional latency time for single messages. Furthermore, the serialized external behavior can be guaranteed as well. Anyway, how much we benefit from MFO depends on the used plans and on the concrete workload. The benefit of MFO is caused by two main facts. First, even for one-message partitions, there is only a moderate runtime overhead (Figures 5.18(b) and 5.22). Second, only a small number of messages is required within one partition to yield a significant speedup (Figure 5.18(b)).

5.6 Summary and Discussion

To summarize, in this chapter, we introduced the data-flow-oriented multi-flow optimization (MFO) technique for throughput maximization of integration flows. Both MFO and the control-flow-oriented vectorization technique achieve throughput improvements. In contrast to vectorization that relies on parallelization, MFO reduces executed work by employing horizontal data partitioning of inbound message queues and executing plans for batches of messages. First, we discussed the plan execution of message partitions that includes the definition of the partition tree as a queue data structure for message partitions as well as the automatic derivation of partitioning attributes, the derivation of partitioning schemes, and the rewriting of plans. Second, we explained the required cost model extensions, the computation of the optimal waiting time with regard to message throughput improvement, and the integration into our overall cost-based optimization framework.

In conclusion of our formal analysis and experimental evaluation, the multi-flow optimization technique achieves significant throughput improvement by accepting moderate additional latency time for single messages. Furthermore, we guarantee constraints of maximum latency for single messages and serialized external behavior. Thus, MFO is applicable for arbitrary asynchronous, data-driven integration flows in many different application areas. Finally, it is important to note that MFO and vectorization can already be applied together in order to achieve the highest throughput due to the integration of both techniques within the surrounding cost-based optimization framework.

Further, MFO opens several opportunities for further optimizations. Future work might consider, for example, (1) the execution of partitions independent of their temporal order,

(2) plan partitioning in the sense of compiling different plans for different partitioning attribute values[17], and (3) MFO for multiple plans by an extended waiting time computation for scheduling overlapping plan executions with regard to the hardware environment.

The main differences of the MFO approach to prior work are twofold: First, the concept of horizontal message queue partitioning simplifies the naïve (time-based) approach because it can be applied also for local operators and no rewriting of queries and local post-processing of query results is required. In addition, horizontal partitioned execution leads to predictable and higher throughput. Second, the approach of computing the optimal waiting time ensures the adaptation to current workload characteristics by adjusting the waiting time according to throughput and latency time. In consequence, the MFO approach can be applied in other domains as well. For example, it might be used for (1) scan sharing, where queries are indexed according to predicate values [UGA+09], or (2) for transient views over equivalent query predicates [ZLFL07].

Beside the achievable throughput improvements, MFO has also some limitations that one should be aware of. First, while caching might lead to using outdated data, the execution of message partitions might cause us to use data that is more current than it was when the message arrived. Despite our guarantee of ensuring eventual consistency as sketched in Subsection 5.1, both caching and MFO, cannot ensure monotonic reads over multiple data objects, which might be a problem if there are hidden dependencies between data objects within the external system. Second, if the number of distinct values is too high, we will not benefit from partitioning due to the additional runtime overhead (partitioned enqueue of messages, serialization at the outbound side) and a fairly low maximum waiting time due to the worst-case latency time consideration according to the number of partitions. However, with regard to the experimental evaluation, there are three facts why we typically benefit from MFO. First, even for one-message partitions, there is only a moderate runtime overhead. Second, throughput optimization is required if and only if high message load (peaks) exists. In such cases, it is very likely that messages with equal attribute values are in the queue. Third, only a small number of messages is required within one partition to yield a significant speedup for different types of operators.

Finally, we consolidate the results from the Chapters 3-5. The general cost-based optimization framework for integration flows, defined in Chapter 3, minimizes the average plan execution time by employing control-flow- and data-flow-oriented optimization techniques but it neglected the alternative optimization objective of throughput improvement. This drawback has been addressed with the integration-flow-specific optimization techniques vectorization (Chapter 4) and multi-flow optimization (Chapter 5). However, the periodical re-optimization algorithm has still several drawbacks. Most importantly, there are the problems of (1) many unnecessary re-optimization steps, where we do not find a new plan if workload characteristics have not changed, and (2) adaptation delays after a workload change, where we use a suboptimal plan until re-optimization and miss optimization opportunities. To tackle these additional problems, in the following Chapter 6, we introduce the concept of on-demand re-optimization.

[17] For example, correlated data inherently leads to data partitions, where each partition has specific statistical characteristics and thus a different optimal plan [Pol05, BBDW05, TDJ10]. MFO in combination with different plans for different data can address this within our cost-based optimization framework. In addition, we can apply plan simplifications (e.g., remove `Switch` operators).

6 On-Demand Re-Optimization

The overall cost-based optimization framework used so far relies on incremental statistic maintenance and periodical re-optimization to meet the high performance demands of integration flows and to overcome the problems of unknown statistics and changing workload characteristics. The potential problems of (1) many unnecessary re-optimization steps and (2) missed optimization opportunities due to adaptation delays are caused by the strict separation of optimization, execution and statistic monitoring. In order to overcome these major drawbacks of periodical re-optimization, in this chapter, we introduce the novel concept of *on-demand re-optimization*.

Our aim is to reduce the overhead for statistics monitoring and re-optimization and at the same time to adapt to changing workload characteristics as fast as possible. We achieve this by extending the optimizer interface of our overall cost-based re-optimization framework in the sense of modeling optimality of a plan by its optimality conditions and triggering re-optimization only if workload changes violate these conditions. First, we define the Plan Optimality Tree (`PlanOptTree`) and describe how to create such a `PlanOptTree` for a given plan to model optimality of this plan by its optimality conditions rather than considering the complete search space. We explain how to use it for statistic maintenance and how this triggers re-optimization if optimality conditions are violated. Second, we exploit these violated conditions for search space reductions during re-optimization. In detail, we explain the directed re-optimization and the update of `PlanOptTrees` after successful re-optimization. Finally, we describe how common optimization techniques are extended in order to enable on-demand re-optimization and we present experimental evaluation results, which compare the periodical re-optimization with this novel on-demand re-optimization approach. According to the experimental evaluation, we achieve improvements concerning re-optimization overhead as well as adaptation sensibility and thus reduce the total execution time.

6.1 Motivation and Problem Description

Efficiency of integration flows is ensured by cost-based optimization in order to (1) exploit the full optimization potential and (2) to adapt to changing workload characteristics [MSHR02, IHW04, BMM+04, DIR07, LSM+07, CC08, NRB09] such as varying cardinalities, selectivities and execution times (for example, reasoned by unpredictable workloads of external systems or temporal variations of network properties). With regard to the low risk of re-optimization overhead and good optimization opportunities, the state-of-the-art cost-based optimization models of integration flows are (1) the periodical re-optimization (see Chapter 3) or (2) the optimize-always optimization model [SVS05, SMWM06]. Within the optimize-always model optimization is triggered for each plan instance, which fails in the case of many plan instances with rather small amounts of data per instance because the optimization time can be even higher than the execution time of a single plan instance.

As mentioned in Subsection 2.2.4, there are fundamental differences to other system

categories that reasoned the use of periodical re-optimization for integration flows. First, integration flows are deployed once and executed *many* times, with rather small amounts of data per instance. Hence, there is no need for mid-instance (inter- or intra-operator) re-optimization. Second, in contrast to continuous-query-based systems, many *independent* instances of an integration flow are executed over time. Thus, there is no need for state migration during plan rewriting. Further advantages are (1) the asynchronous optimization independent of any instance execution, (2) the fact that all subsequent instances (until the next plan change) rather than only the current query benefit from re-optimization, and (3) the efficient inter-instance plan change without state migration. However, this optimization model exhibits also major drawbacks, which we reveal in the following.

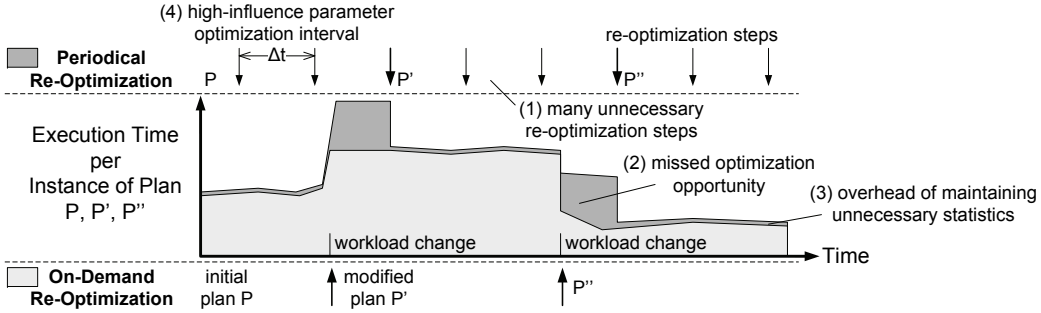


Figure 6.1: Drawbacks of Periodical Re-Optimization

Figure 6.1 shows the execution time of plan instances that have been executed over time in a scenario with two workload shifts. Re-optimization is triggered periodically using a period Δt, where we only find a new plan if a workload shift occurred meanwhile. We observe the potential problems of (1) many unnecessary re-optimization steps, where each step is a full re-optimization and (2) adaptation delays, where we miss optimization opportunities. Furthermore, we might (3) maintain statistics that are not used by the optimizer and (4) the chosen optimization interval has high influence on the execution time. Depending on the optimization interval, periodical re-optimization can even degrade to the unoptimized execution. To tackle these problems, we propose the *on-demand re-optimization* that directly reacts to workload shifts if a new plan is certain to be found. This implies only necessary re-optimization steps and no missed optimization opportunities.

Example 6.1 (Periodical Plan Optimization)**.** *Recall our example plan P_5 that consists of $m = 9$ operators, which is illustrated in Figure 6.2. It receives messages from the system s_3, executes three* `Selection` *operators (according to different attributes). Subsequently, a* `Switch` *operator routes the incoming messages with content-based predicates to schema mapping* `Translation` *operators. Finally, the result is loaded into the system s_6. For each received message, conceptually, an independent instance of this plan is initiated. In order to enable cost-based optimization, statistics are monitored for each operator. We assume that re-optimization is periodically triggered with period Δt, as shown in Figure 6.1. During this re-optimization, all gathered statistics are aggregated and used as cost estimates. However, in this particular example, there are only few rewriting possibilities: In detail, the sequence of* `Selection` *operators can be reordered according to their selectivities (optimality conditions oc_1-oc_3; e.g., $oc_1 : sel(o_2) \leq sel(o_3)$ with $sel = |ds_{out_1}|/|ds_{in_1}|$), and the paths of the* `Switch` *operator can be reordered according to their cost-weighted path probabilities (oc_4). Each single re-optimization is a full optimization, where our transformation-based*

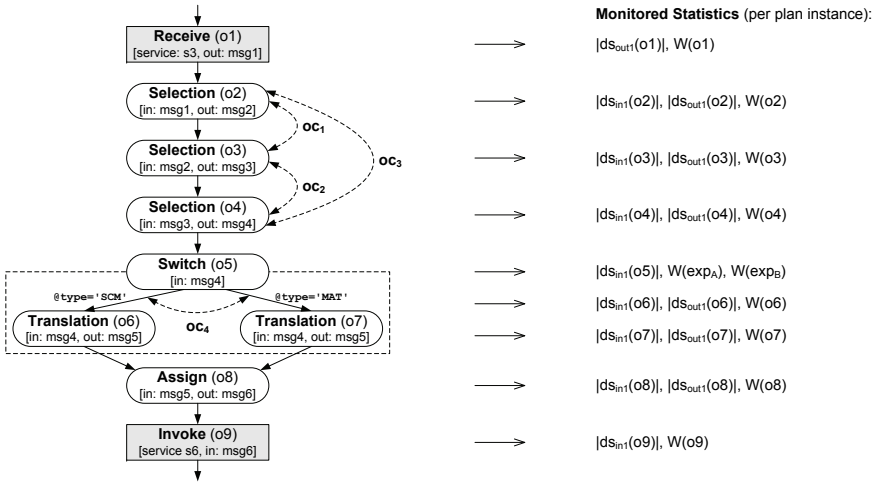


Figure 6.2: Example Plan P_5 and Monitored Statistics

optimization algorithm A-PMO iterates over all operators and applies relevant optimization techniques according to the operator type.

To summarize, the periodical re-optimization has several advantages that reason its application instead of existing approaches from the area of adaptive query processing. However, it exhibits four major drawbacks:

Problem 6.1 (Drawbacks of Periodical Re-Optimization[18])**.** *First, the generic gathering of statistics for all operators leads to the maintenance of statistics that might not be used by the optimizer. Second, periodical re-optimization finds a new plan only if workload characteristics have changed. Otherwise, we trigger many unnecessary invocations of the optimizer that evaluates the complete search space. Third, if a workload change occurs, it takes a while until re-optimization is triggered. Thus, during this adaptation delay, we are using a suboptimal plan and we are missing optimization opportunities. Fourth, the parameter Δt has high influence on optimization and execution times and hence, parameterization requires awareness of changing workloads. We already presented experiments to all of these four drawbacks in Section 3.5. Varying the parameter Δt (e.g., Figure 3.21) showed the spectrum from high re-optimization overheads to a degradation of the execution time to the unoptimized case. In contrast, the generic gathering of statistics (e.g., Figure 3.26) requires further discussion because the overhead of our estimator is negligible. However, including automatic parameter re-estimation techniques (e.g., for continuously computing the optimal smoothing constant α of `EMA`) or using more complex workload aggregation methods would significantly increase this overhead.*

The drawbacks of periodical re-optimization and other optimization models are reasoned by the underlying fundamental problem of the strict separation between optimization, execution and statistics monitoring, which prevents the exchange of detailed information about when and how to re-optimize. This problem of a black-box optimizer was recently reconsidered by Chaudhuri, who argued for rethinking the optimizer contract [Cha09] in the context of DBMS. With regard to related work of the area of adaptive query processing, as

[18] Potential alternatives of periodical re-optimization such as *on-idle re-optimization* (re-optimization on free cycles) or *anticipatory re-optimization* (prediction of workload shifts) also have these drawbacks.

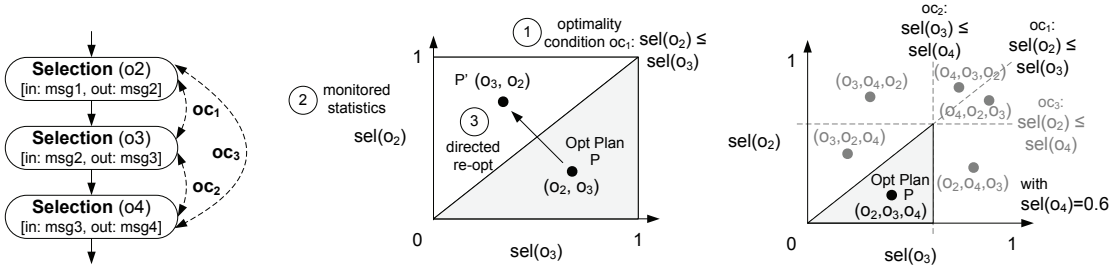
presented in Section 2.2, this underlying problem of a black-box optimizer makes directed re-optimization impossible and the suboptimality of the current plan cannot be guaranteed and thus unnecessary re-optimization steps might be triggered. This problem is also present for approaches of parametric query optimization (PQO). Based on the assumption of unknown query parameters and thus, uncertain statistics during query compilation time (e.g., for stored procedures), PQO [INSS92, HS02, HS03] optimizes a given query into all candidate plans by exhaustively determining the optimal plans for each possible parameter combination (the complete search space) at compile time. The current statistics are used to pick the plan that is optimal for the given parameters at execution time. In contrast to full search space exploration, Progressive PQO (PPQO) [BBD09] aims to iteratively explore the search space over multiple executions of the same query in order to reduce the optimization overhead. Although this approach is advantageous compared to traditional query optimization, it models the complete parameter search space instead of the search space of an individual optimal plan, does not allow for directed re-optimization, does not guarantee to find the optimal plan, and does not address the challenge of when and how to trigger re-optimization.

The major research question is if we could exploit context knowledge from the optimizer, in terms of optimality conditions of a deployed plan, to do just enough for re-optimization. In order to address the mentioned problems, we introduce the *on-demand re-optimization*.

On-Demand Re-Optimization

Our vision of on-demand re-optimization is (1) to model optimality of a plan by its optimality conditions rather than considering the complete search space, (2) to monitor only statistics that are included in these conditions, and (3) to use directed re-optimization if conditions are violated. We use an example to illustrate the idea.

Example 6.2 (On-Demand Re-Optimization)**.** *Assume a subplan $P_5(o_2, o_3)$ (Figure 6.3(a)) and a search space as shown in Figure 6.3(b), where the dimensions represent the selectivities $sel(o_2)$ and $sel(o_3)$. In order to validate the optimality of the plan (o_2, o_3), we continuously observe the optimality condition $oc_1 : sel(o_2) \leq sel(o_3)$, where we only monitor statistics included in such optimality conditions. If an optimality conditions is violated, we can use directed re-optimization in order to determine the new optimal plan.*



(a) Subplan $P_5(o_2, o_3, o_4)$ (b) Plan Diagram of $P_5(o_2, o_3)$ (c) Plan Diagram of $P_5(o_2, o_3, o_4)$

Figure 6.3: Partitioning of a Plan Search Space

Note that we maintain the optimal plan and its optimality conditions (oc_1-oc_3) as shown in Figure 6.3(c) for subplan (o_2, o_3, o_4) but we do not explore the complete search space.

We execute a full optimization only once during the initial deployment of an integration flow. The optimizer contract is changed in a way that it returns the set of optimality conditions. Thus, the resulting research challenge is how to organize these optimality conditions for efficient statistic monitoring, condition evaluation, and directed re-optimization. In contrast to existing *passive* structures such as *matrix views* [BMM+04] that are used and maintained by the re-optimizer for selective operators, we propose an *active* structure, the so-called Plan Optimality Tree (`PlanOptTree`), which is a data structure that models optimality of a plan. It indexes operators and their related statistics, which are included in optimality conditions. As a result, we maintain only required statistics and we can continuously evaluate optimality conditions. Re-optimization is actively triggered only if necessary—in this case, it is guaranteed that we will find a plan with lower costs. Here, directed re-optimization is applied only for operators included in any violated conditions.

Example 6.3 (`PlanOptTree` POT(P_5))**.** *Figure 6.4 shows the* ***`PlanOptTree`*** *of plan* P_5*.*

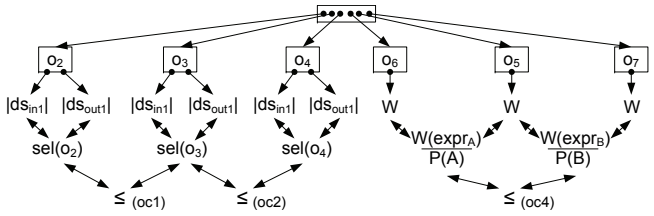


Figure 6.4: `PlanOptTree` of Plan P_5

It includes two optimality conditions (oc_1, oc_2) for expressing the order of the ***`Selection`*** *operators o_2, o_3 and o_4 (see Figure 6.3(c)) according to their selectivities (oc_3 of Example 6.1 is omitted due to transitivity of conditions) and one condition (oc_4) regarding branch prediction of the* ***`Switch`***[19] *operator o_5 according to the weighted path probabilities.*

In the rest of this chapter, we explain in detail how to create, update, and use these `PlanOptTrees` in order to enable the vision of on-demand re-optimization.

6.2 Plan Optimality Trees

In this section, we formally define the `PlanOptTree` and show how to create a `PlanOptTree` for a given plan during the initial optimization of an integration flow. Further, we explain how to use it for statistic maintenance and when to trigger re-optimization.

6.2.1 Formal Foundation

A `PlanOptTree`, which general structure is shown in Figure 6.5, models optimality conditions of a plan and it is defined as follows:

Definition 6.1 (`PlanOptTree`)**.** *Let P denote the* optimal *plan with regard to the current statistics. Further, let m denote the number of operators and let s denote the maximum number of statistic types per operator. Then, the* ***`PlanOptTree`*** *is defined as a graph of five strata representing all optimality conditions of P:*

[19] For the `Switch` operator multiple versions of statistics are monitored. This includes the total execution time $W(o_i)$ as well as the different execution times of all expression evaluations $W(expr_i)$.

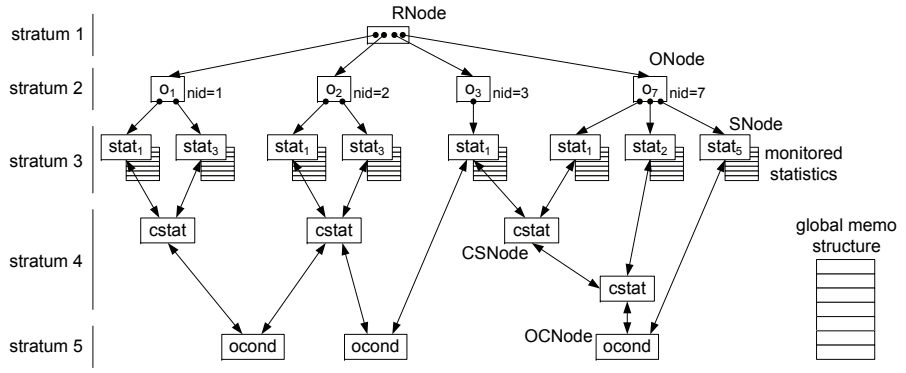


Figure 6.5: General Structure of a `PlanOptTree`

1. RNode: *The single root node refers to m' with $1 \leq m' \leq m$ operator nodes (ONode).*
2. ONode: *An operator node is identified by a node identifier nid and refers to s' with $1 \leq s' \leq s$ statistic nodes (SNode), where s denotes the maximum number of atomic statistic types.*
3. SNode: *A statistic node exhibits one of the s atomic statistic types, where a single type must not occur multiple times for one operator o_i. Further, each SNode contains a list of statistic tuples monitored for o_i, a single aggregate, as well as a reference to a list of CSNodes and a list of OCNodes.*
4. CSNode: *A complex statistic node is a mathematical expression using all referenced parent SNodes or CSNodes as operands, where a CSNode can refer to SNodes of different operators. Further, it refers to a list of complex statistic nodes (CSNode) and a list of optimality condition nodes (OCNode). Hence, arbitrary hierarchies of complex statistics are possible. In addition, CSNodes can be used to represent constant values or externally loaded values.*
5. OCNode: *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.*

The nodes of strata 1 and 2 are reachable over unidirectional references, while nodes of strata 3-5 are defined as bidirectional references (children and parents).

Although the `PlanOptTree` is a graph, we call it a tree, because from the viewpoint of statistic maintenance, only the tree from strata 1 to 3 is relevant, while from the viewpoint of directed optimization, each optimality condition is the root of a tree from strata 5 to 3. All references to children and parents are maintained as sorted lists ordered by their identifier. Conceptually, known index structures can be used instead of lists. Furthermore, each node of stratum 4 and stratum 5 is reachable over multiple paths. For this reason, a `PlanOptTree` includes a `MEMO` structure in order to mark subgraphs that have already been evaluated. Finally, we are able to exploit the following four fundamental properties:

- *Minimal Monitoring:* The `PlanOptTree` includes only operators and statistics that are included in any optimality condition. Thus, we can easily determine the relevant statistics for minimal statistics monitoring (given by stratum 2 and stratum 3).

- *Optimality:* The optimality of a plan is represented by the `PlanOptTree`. If and only if any optimality condition is violated, we will find a plan with lower cost during re-optimization. Thus, there is no need to trigger re-optimization until we detect any violation (break-even point between optimality of different plans).
- *Transitivity:* If a statistic is included in multiple optimality conditions, we can leverage the transitivity of the comparison operators θ. Thus, the total number of required optimality conditions can be reduced.
- *Directed Optimization:* If an optimality condition is violated, we are able to easily determine the involved operators and the related optimization technique that produced this condition. Then, we only need to directly re-optimize those operators.

These properties hold for a complete `PlanOptTree`, while a set of partial `PlanOptTrees` (here, partial is defined as a subset of optimality conditions) might include redundancy, which stands in conflict with *minimal monitoring* and *transitivity*. We will revisit this issue and explain its relevance for further optimization later on.

6.2.2 Creating PlanOptTrees

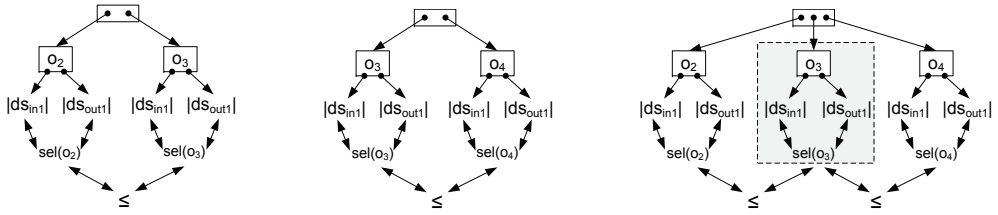
During the initial deployment of an integration flow, the full cost-based optimization is executed once. There, the complete plan search space is evaluated and an initial `PlanOptTree` is created. From this point, the `PlanOptTree` is used for incremental and directed re-optimization only. In this subsection, we explain how to create this initial `PlanOptTree`.

Our standard (transformation-based) optimization algorithm A-PMO recursively iterates over the hierarchy of sequences of atomic and complex operators (internal representation of a plan) and changes the current plan by applying relevant optimization techniques according to the specific types of operators. In contrast, for on-demand re-optimization, we changed the optimizer interface. Now, the optimizer does not only change the current plan but additionally, each applied optimization technique returns also a partial `PlanOptTree` that represents the optimality conditions for the subplan that was considered by this technique. This extension of optimization techniques is straightforward because the existing cost functions and optimality conditions can be reused. For example, the technique *WD4: Early Selection Application* creates a partial `PlanOptTree` when considering two operators. This technique constructs the partial `PlanOptTree` using ONodes, SNodes, a specialized CSNode *Selectivity*, and an OCNode.

The use of the fine-grained partial `PlanOptTrees` at the optimizer interface is advantageous because during directed re-optimization, only subplans are considered and hence, only partial `PlanOptTrees` can be returned. Thus, the solution to the challenge of creating the initial `PlanOptTree` is to merge all partial `PlanOptTrees` to a *minimal* representation.

Example 6.4 (Merging Partial `PlanOptTrees`)**.** *Recall plan P_5 and assume the two partial `PlanOptTrees` (created for the operators o_2 and o_3) shown in Figures 6.6(a) and 6.6(b). When merging those two partial `PlanOptTrees`, we see that operator o_3 and its selectivity (as a CSNode) are used by both partial `PlanOptTrees`. Hence, we add only o_4 and all of its child nodes from POT_2 to POT_1. When doing so, the dangling reference from the new optimality condition to $sel(o_3)$ of POT_2 is modified to refer to the existing selectivity measure $sel(o_3)$ of POT_1. Finally, we created the `PlanOptTree` shown in Figure 6.6(c).*

Algorithm 6.1 describes the creation of a `PlanOptTree` in detail. We iterate over all operators of a given subplan (lines 2-22). If an operator contains a subplan, we recursively



(a) Partial POT_1 (b) Partial POT_2 (c) Merged POT

Figure 6.6: Merging Partial `PlanOptTrees`

invoke this algorithm (line 4), where all subcalls can access the existing `PlanOptTree` *root*. Otherwise, we apply all optimization techniques and obtain the resulting partial `PlanOptTree` for the atomic operator (line 6). If no `PlanOptTree` exists so far, the first partial `PlanOptTree` is used as *root*; otherwise, we merge the partial `PlanOptTree` with the existing *root* (lines 7-22). When merging, we first clear the `MEMO` structure (line 11). Then, we check for the existence of operators as well as statistic nodes, and we add the new nodes if required. At the level of complex statistic nodes, we invoke the `modifyDanglingRefs` in order to change the references of complex statistics and optimality conditions to the existing `PlanOptTree`. We use the `MEMO` structure in order to mark already processed paths of the new partial `PlanOptTree`. As a result, we guarantee a worst-case number

Algorithm 6.1 Initial `PlanOptTree` Creation (A-IPC)

Require: operator *op*, *global variable root* (initially set to `NULL`)

```
1:  o ← op.getSequenceOfOperators()
2:  for i ← 1 to |o| do                                   // for each operator o_i
3:    if type(o_i) ∈ (Plan, Switch, Fork, Iteration, Undefined) then   // complex
4:      o_i ← A-IPC(o_i)
5:    else                                                // atomic
6:      ppot ← getPartialOptTree(o_i)
7:      if root = NULL then
8:        root ← ppot
9:      else
10:       // merge partial PlanOptTrees
11:       clear memo
12:       for all on ∈ ppot.onodes do                     // for each ONode on
13:         if root.containsONode(on.nid) then
14:           eon ← root.getOperator(on.nid)
15:           for all sn ∈ on.snodes do                   // for each SNode sn
16:             if eon.containsSNode(sn.type) then
17:               eson ← eon.getSNode(sn.type)
18:               modifyDanglingRefs(eon, eson, on, sn)
19:             else
20:               eon.snodes.add(sn)
21:         else
22:           root.onodes.add(on)                         // add operator subtree
23: return root
```

of nodes within a `PlanOptTree` for a given plan, which indirectly implies the worst case complexity for any operation that evaluates at most all nodes of such a `PlanOptTree`.

Theorem 6.1 (Worst-Case Complexity)**.** *The worst-case time and space complexity of a `PlanOptTree` for a plan of m operators is $O(m^2)$.*

Proof. Assume a plan P with m operators. 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 in one optimality condition per dependency (in case of a data dependency this subsumes any temporal dependency) and in one additional optimality condition for binary operators. Now, let us assume a sequence of operators o. Then, an arbitrary 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 given by

$$|oc| = \sum_{i=1}^{m}(i-1) + m = \sum_{i=1}^{m-1} i + m = \frac{m \cdot (m+1)}{2}. \tag{6.1}$$

Hence, Theorem 6.1 holds. □

After we have created the initial `PlanOptTree`, we can apply the following two optimizations with an additional single pass over all nodes of the `PlanOptTree`. However, for simplicity of presentation and due to a small impact on our use cases, we did not apply them in the examples.

- *Collapsing Statistic Hierarchies (CSNodes):* The hierarchies of statistic nodes of partial `PlanOptTrees` are defined for each optimization technique with regard to re-usability. Thus, there might be unnecessarily fine-grained CSNodes. We collapse these hierarchies by merging CSNodes with their children if only a single child exists. Similarly to the merging of prefix nodes within a patricia trie [Mor68] this can reduce the number of levels of the `PlanOptTree`.
- *Reusing Atomic Statistic Measures (SNodes):* For operators of the `PlanOptTree` with a data dependency between them, we reuse cardinalities across those operators in order to eliminate redundancy. Due to the data dependency, the output cardinality of operator o_i is equal to the input cardinality of operator o_{i+1} ($|ds_{out}(o_i)| = |ds_{in}(o_{i+1})|$). Hence, we remove the latter SNode ($|ds_{in}(o_{i+1})|$) and modify the references. This requires awareness when updating the `PlanOptTree` after successful re-optimization, because the re-optimization might have changed the ordering of operators and thus, also changed the data dependencies and statistics.

As a result of creating the initial `PlanOptTree`, we obtain a structure that represents optimality of a plan with the properties sketched in Subsection 6.2.1. It includes all cost conditions that must be satisfied for plan optimality with regard to the current statistics. Thus, only the update of statistics can trigger re-optimization.

6.2.3 Updating and Evaluating Statistics

We use the `PlanOptTree` for statistics maintenance and for immediate evaluation of optimality conditions. This triggers re-optimization if optimality conditions are violated.

For on-demand re-optimization, we only maintain statistics that are required to evaluate the optimality conditions. All other statistics are declined by the `PlanOptTree` such that they are not stored and aggregated. When atomic statistics are updated, we also maintain the aggregate, update the hierarchy of complex statistics measures, and evaluate optimality conditions that are reachable children of this statistic node. Arbitrary workload aggregation methods, as described in Subsection 3.3.2, can be used for aggregation of atomic statistics. Due to this incremental maintenance and immediate condition evaluation, the use of Exponential Moving Average (`EMA`) is most suitable because (1) it is incrementally maintained and (2) no negative statistics maintenance (sliding window) is necessary due to the exponentially decaying weights.

The naïve application of triggering re-optimization based on the incrementally monitored statistics could lead to the problem of frequently changing plans.

Example 6.5 (Problem of Frequent Plan Changes)**.** *Assume the optimality condition of* $sel(\sigma_A) \leq sel(\sigma_B)$*. There are two problems that can cause frequent plan changes (instability). First, due to unknown statistics, we are only able to monitor conditional selectivities* $sel(\sigma_A)$ *and* $sel(\sigma_B|\sigma_A)$*. If A and B are correlated, the optimality condition might be violated even after re-optimization. Second, if the selectivities are constant but alternate around equality with* $sel(\sigma_A) \approx sel(\sigma_B)$*, we would also change the plan back and forth. In both cases, we would get frequent re-optimization steps that are not amortized.*

We explicitly address this problem of missing robustness (instability) when triggering re-optimization with the following strategies:

- *Correlation Tables:* As described in Subsection 3.3.4, we explicitly compute conditional selectivities using a lightweight correlation table. Essentially, we maintain selectivities over multiple versions of a plan, where we store and maintain a row of atomic and conditional selectivities for each pair of operators with direct data dependency within the current plan. Unless we see the second operator ordering, we assume statistical independence. However, based on the maintenance of conditional selectivities, we do not make a wrong decision based on correlation twice. For on-demand re-optimization, the use of this correlation table is even more important. The integration into the `PlanOptTree` is realized by a specific complex statistic node (CSNode) *Conditional Selectivity* that maintains and reads the correlation table.
- *Minimal Existence Time:* We use the time period Δt from periodical re-optimization as *minimal existence time* of a plan. This means that no optimality conditions are evaluated during Δt after the last re-optimization. During this interval, we only collect statistics but we do not aggregate and evaluate them. As a result, the adaptation sensibility is reduced in order to avoid that re-optimization is triggered multiple times in case that (1) we have not finished another asynchronous re-optimization step or (2) the workload has changed abruptly and caused the violation of multiple optimality conditions with short delay. However, Δt determines only the minimum period of re-optimization and hence, can be set independently of the workload characteristics (low-influence parameter). After that, we continuously check optimality conditions and adapt faster to workload changes than periodical re-optimization does.
- *Lazy Condition Violation:* When evaluating an optimality condition, we might not have seen all atomic statistics of a plan instance. Similar to known control strategies, re-optimization is lazily triggered if the condition is violated $\sum_1^{m'} s'$ times, which is

at least the number of monitored statistics of one plan instance, i.e., the number of SNodes in the `PlanOptTree` (e.g., six in our example). The condition must be violated by a relative threshold τ and a true evaluation resets this lazy count.

Algorithm 6.2 `PlanOptTree` Insert Statistics (A-PIS)

```
Require: operator id nid, stat type type, statistic value, global variable lastopt
 1: if (on ← root.getOperator(nid)) = NULL or (sn ← on.getSNode(type)) = NULL then
 2:   return                                                  // statistic not required
 3: sn.maintainAggregate(value)
 4: ret ← true
 5: if (time − Δt) > lastopt  then                            // min existence time
 6:   clear memo
 7:   for all cn ∈ sn.csnodes do                              // for each CSNode cn
 8:     if memo.contains(cs) then
 9:       continue
10:     ret ← ret and cs.computeStats(), memo.put(cs)
11:   for all oc ∈ sn.ocnodes do                              // for each OCNode oc
12:     if memo.contains(oc) then
13:       continue
14:     ret ← ret and oc.isOptimal(), memo.put(oc)
15: if  ¬ret  then
16:   A-PTR()                  // actively triggering re-optimization (start reopt thread)
```

Algorithm 6.2 illustrates the statistics maintenance using a `PlanOptTree` and shows how to trigger re-optimization if workload characteristics change over time. Starting from the root, we identify the operator node according to the given *nid*. We decline the maintenance of a statistic if it is not required by the `PlanOptTree` (lines 1-2). Then, we get the specific statistic node and maintain the aggregate (line 3). If the minimum existence time is exceeded, we check the existing optimality conditions. For this purpose, we use our `MEMO` structure, recursively compute complex statistic measures, and check all reachable optimality conditions (lines 6-14). Materialization of complex statistics is applied with awareness of the `PlanOptTree` structure in order to prevent anomalies (e.g., inconsistency of statistics in one optimality condition). We do not miss any violated condition when using the `MEMO` structure because this algorithm is executed for each atomic statistic value. In case there is a violated optimality condition, we trigger re-optimization (line 16).

In conclusion, we evaluate optimality conditions as we gather new statistics, and re-optimization is triggered only if optimality conditions are violated. In contrast to periodical re-optimization, this yields (1) minimal statistics maintenance, (2) the avoidance of unnecessary re-optimization steps, and (3) an immediate adaptation to changing workload characteristics, and hence, it results in a lower overall execution time.

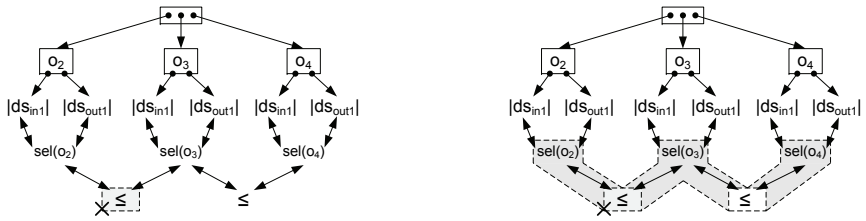
6.3 Re-Optimization

Once re-optimization has been triggered by violated optimality conditions, we re-optimize the current plan. This includes two major challenges. First, we apply directed re-optimization rather than full re-optimization. Second, after successful re-optimization, we incrementally update the existing `PlanOptTree` according to the new conditions.

6.3.1 Optimization Algorithm

Recall our search space shown in Figure 6.3. If certain optimality conditions are violated, we apply directed re-optimization according to those violations. At this point, we know the violated optimality conditions and the optimization techniques, which produced these conditions. For directed re-optimization, we additionally need to determine the minimal set of operators that should be reconsidered by those techniques. In this context, the *transitivity* property of the `PlanOptTree` must be taken into account.

Example 6.6 (Determining Re-Optimization Search Space)**.** *Assume the* `PlanOptTree` *of our example. 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 Figure 6.7(a).*



(a) Initiating (Violated) OCNode (b) Transitive OCNode

Figure 6.7: Determining Re-Optimization Potential

Re-optimization was triggered because the selectivity of operator o_2 *changed to* $sel(o_2) = 0.45$*. Thus, we need to reconsider operators* o_2 *and* o_3 *during re-optimization. However, due to the transitivity of optimality conditions, we evaluate operator* o_4 *as well, because we do not know in advance if the implicit optimality condition of* $sel(o_2) \leq sel(o_4)$ *still holds. The* `PlanOptTree` *is traversed as shown in Figure 6.7(b) to determine that this transitive condition is violated as well.*

From a macroscopic view, we follow a bottom-up approach from the optimality conditions to determine all operators for directed re-optimization. We start at each violated optimality condition and traverse (without cycles) all other 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.

Algorithm 6.3 illustrates how to trigger re-optimization. During statistics maintenance, all violated optimality conditions are added to the set OC. When triggering re-optimization, we traverse the logical transitivity chains for each optimality condition in OC. For the left operator of the optimality condition, we check the transitivity filter (direction of condition operands and its comparison operator) (line 4). If it holds, we check the transitive optimality filter for this reachable optimality condition (line 5). If optimality is violated, we mark this OCNode as a transitively violated condition, add it to OC (line 6) and proceed recursively along the left chain (line 7). This is done similarly for the right operator of the optimality condition (lines 8-12). Finally, we invoke the optimizer with the set OC. The optimizer uses the violated optimality conditions and all involved operators (line 13). Despite the unidirectional references of strata 1 and 2, we can efficiently determine these operators. After successful re-optimization, the `PlanOptTree` is

Algorithm 6.3 `PlanOptTree` Trigger Re-Optimization (A-PTR)

Require: invalid optimality conditions OC

```
1:  clear memo
2:  for all oc ∈ OC do                                          // for each OCNode
3:    for all oc1 ∈ oc.op1.ocnodes do                           // for each OCNode of operand 1
4:      if oc1.θ = oc.θ and oc1.op2 = oc.op1 then
5:        if ¬oc1.isOptimal(oc1.op1.agg, oc.op2.agg) then
6:          OC ← OC ∪ oc1
7:          rCheckTransitivity(oc, oc1.op1, left)
8:    for all oc2 ∈ oc.op2.ocnodes do                           // for each OCNode of operand 2
9:      if oc2.θ = oc.θ and oc2.op1 = oc.op2 then
10:       if ¬oc2.isOptimal(oc2.op2.agg, oc.op1.agg) then
11:         OC ← OC ∪ oc2
12:         rCheckTransitivity(oc, oc2.op2, right)
13: PPOT ← optimizePlan(ptid, OC)                               // apply directed re-optimization
14: A-PPR(PPOT, OC)                                             // update PlanOptTree
```

incrementally updated (line 14). It is important to note that the directed re-optimization of operators involved in violated optimality conditions is equivalent to full re-optimization.

The directed re-optimization relies on monotonic cost functions. This ensures that no local suboptima exist and that we will find the global optimum. The Picasso project [RH05] showed that this assumption holds for complete cost diagrams over multiple alternative plans of *most* queries [HDH07, HDH08, DBDH08]. In contrast, it *always* holds for our cost model of integration flows with regard to a single plan (see Subsection 3.2.2). However, due to the possibility of arbitrarily complex optimality conditions, even without this property, we could still guarantee to trigger full re-optimization if a better plan exists.

Theorem 6.2 (Directed Re-Optimization)**.** *The directed re-optimization for all operators $o' \in P$ that have been identified by violated optimality conditions oc' of a* `PlanOptTree` *is equivalent 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 $G = (V, A)$ of vertexes (operators) and arcs (dependencies). Then, the re-optimization of P is a graph homomorphism $f : G \to H$. In order to prove Theorem 6.2, we show that

$$\forall o_i \notin o' : \quad \left(v_{pre(o_i)} \in G \equiv v_{pre(o_i)} \in H\right) \wedge \left(v_{suc(o_i)} \in G \equiv v_{suc(o_i)} \in H\right), \tag{6.2}$$

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 $f : G \to H$ such that

$$v_j \prec o_i \in G \wedge o_i \prec v_j \in H, \tag{6.3}$$

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 successors of o_i. (2) All used cost functions are known to be monotonically non-decreasing w.r.t. the input statistics. Hence, during re-optimization, $f : G \to H$, the globally optimal solution will be found. (3) Further, all operators o' included in violated optimality conditions $\forall o_i \in oc'$ or transitive optimality conditions $\forall o_i \in toc'$ are used by $f : G \to H$. As a result,

$$\nexists \left(o_i \notin o' \wedge \left(\left(v_{pre(o_i)} \in G \neq v_{pre(o_i)} \in H\right) \vee \left(v_{suc(o_i)} \in G \neq v_{suc(o_i)} \in H\right)\right)\right), \tag{6.4}$$

such that both directed re-optimization and full re-optimization results in the same plan. Hence, Theorem 6.2 holds. □

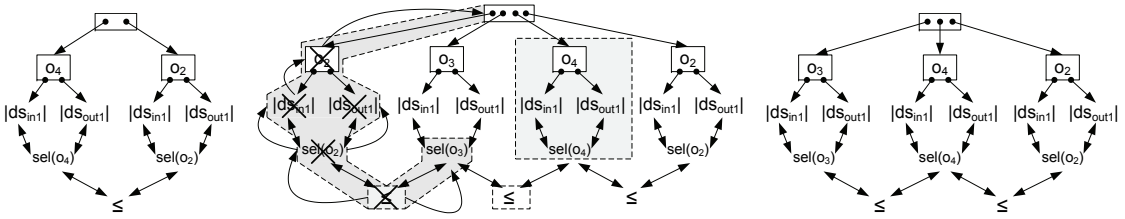
As a result of directed re-optimization, we minimized the re-optimization overhead for large plans and for optimization techniques with a large search space.

6.3.2 Updating PlanOptTrees

Due to directed re-optimization, where we only consider a subset of all plan operators, we need to incrementally update the existing `PlanOptTree` according to the rewritten plan. The extension of the optimizer interface is based on returning partial `PlanOptTrees`. Thus, after successful re-optimization, we incrementally update the `PlanOptTree` with the partial `PlanOptTrees` of the newly created subplans. More precisely, the problem is to update a given `PlanOptTree` that includes violated optimality conditions using a set of given partial `PlanOptTrees`. The result of this update must be semantically equivalent to a full creation from scratch (A-IPC).

We follow an approach called *Partial `PlanOptTree` Replacement*. We start bottom-up from all violated optimality conditions and remove those OCNodes except for transitively violated conditions because the new partial `PlanOptTree` is already aware of them. Subsequently, we traverse up the tree and remove all nodes that do not refer to at least one child any longer. As a result, we remove all nodes included in violated optimality conditions. Finally, we can apply the merging of partial `PlanOptTrees` for all new subplans similar to creating the initial `PlanOptTree`.

Example 6.7 (Partial `PlanOptTree` Replacement)**.** *In Example 6.6, the optimality condition $sel(o_2) \leq sel(o_3)$ and the transitive condition $sel(o_2) \leq sel(o_4)$ were violated. The triggered re-optimization returned the partial `PlanOptTree` shown in Figure 6.8(a).*



(a) New Partial POT (b) POT during Replacement (c) POT after Replacement

Figure 6.8: Partial `PlanOptTree` Replacement

For an incremental update (Figure 6.8(b)), we remove the violated optimality condition $sel(o_2) \leq sel(o_3)$, the selectivity $sel(o_2)$, operator o_2 and its statistics but before deletion, we copy the statistics to the new `PlanOptTree`. The additional arrows illustrate the traversal paths. We then merge the new partial `PlanOptTree` with the remaining `PlanOptTree`. Operator o_4 and all of its conditions are reused and the result is the `PlanOptTree` shown in Figure 6.8(c) representing plan optimality according to the statistics from Example 6.6.

Algorithm 6.4 illustrates the details of the incremental update approach. For each optimality condition in the set of violated optimality conditions, we remove the OCNode and reachable statistic nodes without children (lines 2-7). Then, we clear strata 1-2

Algorithm 6.4 Update via Partial `PlanOptTree` Replacement (A-PPR)

Require: set of new partial `PlanOptTrees` PPOT, invalid optimality conditions OC

```
1:  for all oc ∈ OC do                          // remove invalid optimality conditions
2:    oc.op1.ocnodes.remove(oc)
3:    if oc.op1.|ocnodes| = 0 and oc.op1.|csnodes| = 0 then
4:      rremoveNodes(oc.op1)                                  // recursive bottom-up
5:    oc.op2.ocnodes.remove(oc)
6:    if oc.op2.|ocnodes| = 0 and oc.op2.|csnodes| = 0 then
7:      rremoveNodes(oc.op2)                                  // recursive bottom-up
8:  clearStrata12()
9:  for all ppot ∈ PPOT do                  // merge new partial PlanOptTrees
10:   mergePPOT(root, ppot)                                   // see A-IPC lines 11-22
```

starting from the root with the same concept of removing nodes without any children. Finally, we apply the merge algorithm (line 10) from Subsection 6.2.2.

With the aim of reuse, we could index plans and `PlanOptTrees` created over time by their optimality constraints. This could avoid redundant directed re-optimization and merging `PlanOptTrees` but we would still need to copy statistics. Due to the risk of (1) maintenance overhead (e.g., plans, `PlanOptTrees`), (2) a potentially large search space, as well as (3) low remaining optimization potential, we do not reuse plans. However, future work might investigate this by combining on-demand re-optimization with (progressive) parametric query optimization (PPQO) [BBD09] by iteratively creating possible plans of the search space according to the optimality conditions and subsequently, reusing already created plans.

To summarize, we have shown how to use the `PlanOptTree` for directed re-optimization and how a `PlanOptTree` can be incrementally updated after successful re-optimization. All algorithms presented rely on the extension of the optimizer interface by returning partial `PlanOptTrees` or by directly rearranging the referenced `PlanOptTree`. Either way, existing optimization techniques require modifications. In the following, we will explain these modifications using selected optimization techniques from previous chapters.

6.4 Optimization Techniques

In order to illustrate the applicability of the `PlanOptTree` and the necessary modifications of optimization techniques, we use examples to show how their optimality conditions can be expressed with our approach. First, we describe the on-demand re-optimization for common rewriting techniques (see Chapter 3). Second, we show how the concept of on-demand re-optimization can be applied for cost-based vectorization (see Chapter 4) and multi-flow optimization (see Chapter 5) as well.

6.4.1 Control-Flow- and Data-Flow-Oriented Techniques

For control-flow- and data-flow-oriented rewriting techniques, their already presented optimality conditions can be reused as they are. In this subsection, we discuss the on-demand re-optimization for the common data-flow-oriented example optimization techniques join enumeration, eager group-by, and set operations with distinctness.

Join Enumeration Example

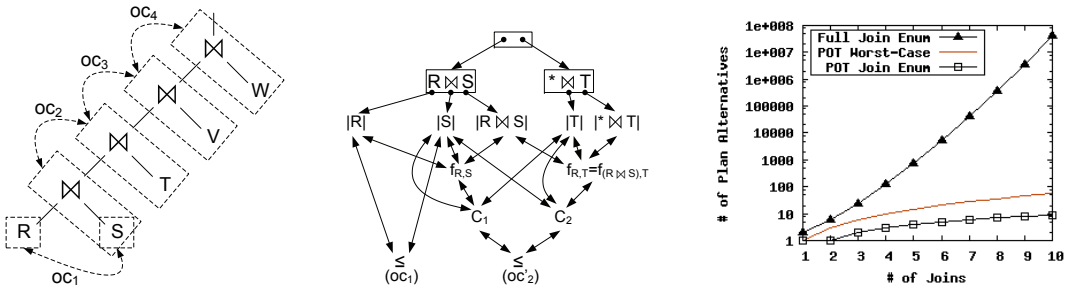
Recall our join enumeration heuristic for the optimization technique *WD10: Join Enumeration*, presented in Subsection 3.3.3, where we assumed a left-deep join tree $(R \bowtie S) \bowtie T$ of $n = 3$ data sets (without cross products and only one join implementation in the form of a nested loop join) with the following $n! = 6$ possible plans:

$$\begin{array}{llll} P_a(opt): & (R \bowtie S) \bowtie T & P_c: & (R \bowtie T) \bowtie S \qquad P_e: \quad (S \bowtie T) \bowtie R \\ P_b: & (S \bowtie R) \bowtie T & P_d: & (T \bowtie R) \bowtie S \qquad P_f: \quad (T \bowtie S) \bowtie R. \end{array}$$

The costs of a (nested loop) join are computed by $C(R \bowtie S) = |R| + |R| \cdot |S|$. Further, the join output cardinality can be derived by $|R \bowtie S| = f_{R,S} \cdot |R| \cdot |S|$ with a join filter selectivity of $f_{R,S} = |R \bowtie S|/(|R| \cdot |S|)$. Thus, the costs of the complete plan $(R \bowtie S) \bowtie T$ are given by $C((R \bowtie S) \bowtie T) = |R| + |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| \cdot |T|$. Assuming variable selectivities and cardinalities, the optimality conditions for arbitrary left-deep join trees (see Figure 6.9(a)) are specified as follows. First, fix two base relations with the commutativity optimality condition of $oc_1: \ |R| \leq |S|$. Second, the optimality of executing $R \bowtie S$ before $* \bowtie T$ is given if the following optimality condition holds:

$$\begin{aligned} oc_2: \quad & |R| + |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| + f_{R,S} \cdot |R| \cdot |S| \cdot |T| \\ & \leq |R| + |R| \cdot |T| + f_{R,T} \cdot |R| \cdot |T| + f_{R,T} \cdot |R| \cdot |T| \cdot |S|. \qquad (6.5) \\ oc_2': \quad & |S| + f_{R,S} \cdot |S| + f_{R,S} \cdot |S| \cdot |T| \leq |T| + f_{R,T} \cdot |T| + f_{R,T} \cdot |T| \cdot |S|, \end{aligned}$$

where oc_2 has been algebraically simplified to oc_2' by subtracting $|R|$ and subsequently dividing by $|R|$. Note that it is possible to monitor all cardinalities $|R|$, $|S|$, and $|T|$ but only the selectivities $f_{R,S}$ and $f_{(R \bowtie S),T}$. To estimate $f_{R,T}$, we need to derive it with $f_{R,T} \ \theta \ f_{(R \bowtie S),T}$, where θ is a mapping function representing the correlation. If we assume statistical independence of selectivities, we can set $f_{R,T} = f_{(R \bowtie S),T}$[20].



(a) Optimality Conditions (b) Example POT (c) Complexity Analysis

Figure 6.9: Example Join Enumeration

The `PlanOptTree` for this example is illustrated in Figure 6.9(b). Here, the `PlanOptTree` contains the two mentioned optimality conditions, which use a hierarchy of atomic and complex statistics. All input cardinalities of base relations and the output cardinalities of join operators are used in the form of atomic statistic nodes. The join selectivities (complex statistic nodes) are computed from the atomic statistics. Both terms of the inequality oc_2' are computed using atomic and complex statistics.

[20] Another approach would be to use serial histograms [Ioa93] or exact frequency matrices [Pol05].

Finally, Figure 6.9(c) compares the number of alternative plans of the full search space with the number of required optimality conditions, using a log-scaled y-axis. The search space reduction (see Theorem 6.1 for the worst case consideration) is achieved by transforming the problem of enumerating all possible plans into binary optimality conditions, where costs in front of and after the considered subplan are equal. Note that the assumption of transitivity does not necessarily require that the used cost model has the adjacent sequence interchange (ASI) property [Moe09]. Figure 6.9(a) shows how this heuristic join enumeration approach works for arbitrarily large left-deep join trees of n input data sets. For this join tree type, the `PlanOptTree` has $n-1$ optimality decisions (shown as arrows).

Due to the restriction of checking only for plan optimality and due to arbitrary complex optimality conditions, this concept of triggering re-optimization leads to the globally optimal solution. In addition, the `PlanOptTree` allows for directed re-optimization. With regard to an equivalent optimization result compared to full join enumeration (e.g., with DPSize), $n(n+1)/2$ optimality conditions are required and only a single reordering of two join operators is applied during one re-optimization step and multiple of these steps are required to find the global optimum. However, in case of heuristic join enumeration, directed re-optimization can be used for all operators in one single re-optimization step but we might not find the global optimum.

Eager Group-By Example

Similarly to the join enumeration example, assume a join of n data sets (with arbitrary multiplicities) and a subsequent group-by, where the join predicate and group-by attributes are equal with $\gamma_{F(X);A_1}(R \bowtie_{R.A_1=S.A_1} S)$. With regard to the optimization technique *WD6: Early Group-by Application*, there are $4n!$ (for $n \geq 2$) possible plans. Without loss of generality, we assume $n = 2$ and concentrate on the $4n! = 8$ possibilities to arrange group-by and join (for an invariant group-by the final γ in P_c-P_f can be omitted):

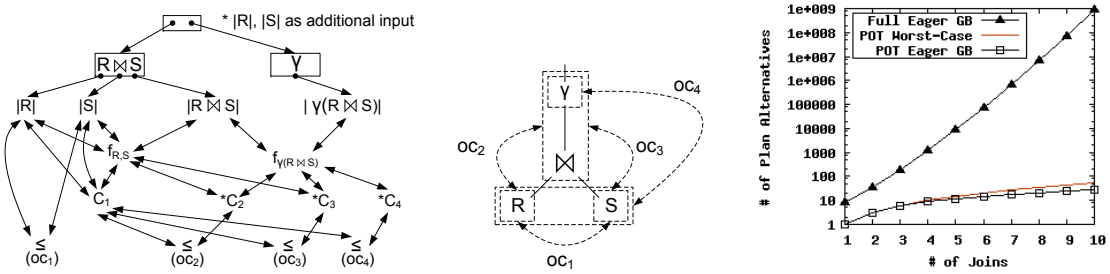
$$\begin{array}{llllllll} P_a(opt): & \gamma(R \bowtie S) & P_c: & \gamma((\gamma R) \bowtie S) & P_e: & \gamma(R \bowtie (\gamma S)) & P_g: & (\gamma R) \bowtie (\gamma S) \\ P_b: & \gamma(S \bowtie R) & P_d: & \gamma(S \bowtie (\gamma R)) & P_f: & \gamma((\gamma S) \bowtie R) & P_h: & (\gamma S) \bowtie (\gamma R). \end{array}$$

In addition to the join costs, the group-by costs are given by $C(\gamma) = |R| + |R| \cdot |R|/2$. Furthermore, the output cardinality in case of a single group-by attribute A_i, with a domain D_{A_i}, is defined as $1 \leq |\gamma R| \leq |D_{A_i}(R)|$, while for an arbitrary number of group-by attributes it is $1 \leq |\gamma R| \leq \prod_{i=1}^{|A|} |D_{A_i}(R)|$. Further, let us denote the group-by selectivity with $f_{\gamma R} = |\gamma R|/|R|$. The optimal plan P_a can then be represented with four optimality conditions. First, the join order is expressed with $oc_1 : |R| \leq |S|$. Second, we use one optimality condition for each single join input (oc_2 and oc_3, where we illustrate oc_2 as an example) and one condition for all join inputs (oc_4):

$$\begin{aligned} oc_2: \; & C(\gamma(R \bowtie S)) \leq \left(|R| + |R|^2/2\right) + \left(f_{\gamma R} \cdot |R| + f_{\gamma R} \cdot |R| \cdot |S|\right) \\ & + \left(f_{(\gamma R),S} \cdot f_{\gamma R} \cdot |R| \cdot |S| + \left(f_{(\gamma R),S} \cdot f_{\gamma R} \cdot |R| \cdot |S|\right)^2/2\right) \\ oc_4: \; & C(\gamma(R \bowtie S)) \leq \left(|R| + |R|^2/2\right) + \left(|S| + |S|^2/2\right) \\ & + \left(f_{\gamma R} \cdot |R| + f_{\gamma R} \cdot |R| \cdot f_{\gamma S} \cdot |S|\right) \\ \text{with } & C(\gamma(R \bowtie S)) = \left(|R| + |R| \cdot |S|\right) + \left(f_{R,S} \cdot |R| \cdot |S| + \left(f_{R,S} \cdot |R| \cdot |S|\right)^2/2\right). \end{aligned} \tag{6.6}$$

Similar to the join example, we compute the join selectivity by $f_{(\gamma R),S} \; \theta \; f_{R,S}$ and assume independence with $f_{(\gamma R),S} = f_{R,S}$. The group-by selectivity is computed by

$f_{\gamma((\gamma R)\bowtie S)}\ \theta\ f_{\gamma(R\bowtie S)}/f_{\gamma(R)}$. If we assume independence for this group-by selectivity, we have $f_{\gamma((\gamma R)\bowtie S)} = 1$ and can set $f_{\gamma(R\bowtie S)} = f_{\gamma(R)}$. As a result, we can derive all variables of the optimality conditions from statistics of the optimal plan.



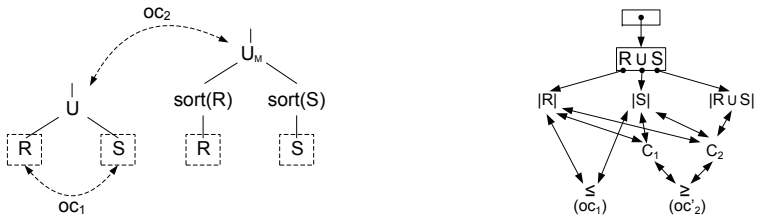
(a) Example POT (b) Optimality Conditions (c) Complexity Analysis

Figure 6.10: Example Eager Group-By

Figure 6.10(a) shows the resulting `PlanOptTree`, where we omitted some connections (*) to atomic statistic nodes for simplicity of presentation. Note that for eager group-by, no transitivity is used. Furthermore, only the plan optimality is modeled rather than the whole plan search space. Hence, only four optimality conditions are required per join operator as shown in Figure 6.10(b). Accordingly, Figure 6.10(c) compares the number of alternative plans of the full search space with the number of required optimality conditions. The improvement is reasoned by the fact that for each join input, we just model if pre-aggregation is advantageous or not.

Union Distinct Example

In contrast to join enumeration or eager group-by, there are many control-flow- and data-flow-oriented optimization techniques with fairly simple optimality conditions and thus, rather small `PlanOptTrees`. An example is the optimization technique *WD11: Setoperation-Type Selection* (set operations with distinctness).



(a) Union Distinct Alternatives (b) Example POT

Figure 6.11: Example Union Distinct

There are three alternative subplans for a union distinct $R \cup S$. First, there is the normal union distinct operator with costs that are given by $C(R \cup S) = |R| + |S| \cdot |R \cup S|/2$ (two plans due to asymmetric costs), where $|R| \leq |R \cup S| \leq |R| + |S|$ holds. Second, we can sort both inputs and apply a merge algorithm with costs of

$$C\left(\texttt{sort}(R) \cup_M \texttt{sort}(S)\right) = |R| + |S| + |R| \cdot \log_2|R| + |S| \cdot \log_2|S|. \tag{6.7}$$

For arbitrary cardinalities, the optimality conditions (Figures 6.11(a) and 6.11(b)) are

$$\begin{aligned} oc_1 &: \ |R| \geq |S| \text{ and} \\ oc_2 &: \ |R| + |S| \cdot \frac{|R \cup S|}{2} \leq |R| + |S| + |R| \cdot \log_2|R| + |S| \cdot \log_2|S|. \end{aligned} \tag{6.8}$$

After simplification of oc_2, we obtain

$$oc_2' : \ |R \cup S| \leq 2\left(1 + \frac{|R| \cdot \log_2|R|}{|S|} + \log_2|S|\right). \tag{6.9}$$

Figure 6.11(b) illustrates the resulting `PlanOptTree`, where we monitor the input and output cardinalities $|R|$, $|S|$, $|R \cup S|$ and we only have to check the two fairly simple optimality conditions oc_1 and oc_2', respectively. A similar concept is also used for example, when deciding on nested-loop or sort-merge joins.

6.4.2 Cost-Based Vectorization

The concept of on-demand re-optimization can also seamlessly be applied to the control-flow-oriented technique cost-based plan vectorization that has been presented in Chapter 4. For this technique, the created `PlanOptTree` depends on the used algorithm and optimization objective. In this subsection, we illustrate the on-demand re-optimization for the, already discussed, constrained optimization objective of

$$\phi_c = \min_{k=1}^{m} k \quad | \quad \forall i \in [1,k] : \left(\sum_{j=1}^{l_{b_i}} W(o_j)\right) \leq W(o_{max}) + \lambda, \tag{6.10}$$

and the typically used heuristic computation approach (A-CPV), whose core idea is to merge buckets in a first-fit (next-fit) manner.

Let o be a sequence of m operators that has been distributed to k execution buckets b_i with $1 \leq k \leq m$. Plan optimality is then represented by $2k-1$ optimality conditions with regard to the heuristic computation approach (A-CPV). First, for each bucket b_i, the total execution time of all included operators must be below the maximum cost constraint with $oc : W(b_i) \leq W(o_{max}) + \lambda$. Second, for each bucket, except the first one, we check if the first operator o_i of this bucket b_i still cannot be assigned to the previous bucket b_{i-1} with $oc_i : W(b_{i-1}) + W(o_i) \geq W(o_{max}) + \lambda$. This optimality condition is reasoned by the first-fit (next-fit) character of the A-CPV.

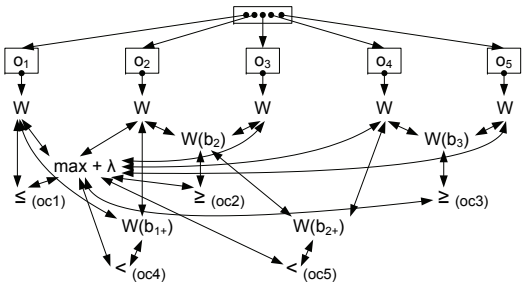


Figure 6.12: Example `PlanOptTree` of Cost-Based Vectorization

The resulting `PlanOptTree` for a sequence of $m = 5$ operators that has been distributed to $k = 3$ execution buckets is illustrated in Figure 6.12. With regard to cost-based vectorization, we need to include operator nodes for all operators, but only the execution time as atomic statistic nodes. Furthermore, we use the complex statistic node $W(o_{max}) + \lambda$ and the aggregated bucket execution costs $W(b_i)$ for each bucket with more than two operators. Then, there are three optimality conditions (oc_1-oc_3), which check that the bucket execution costs (or operator execution costs) are below this maximum. In addition, two optimality conditions (oc_4 and oc_5) are used in order to check if all operators belong to the right bucket with regard to producing the same result as the A-CPV does.

Whenever one of the optimality conditions is violated, we trigger directed re-optimization. In this case we know the operator or execution bucket, respectively, which reasoned this violation. In contrast to the the full A-CPV, we directly start the cost-based plan vectorization at this bucket and hence, might not need to evaluate all operators. However, the worst-case time complexity of $O(m)$ is not changed by the directed cost-based plan vectorization because we might start directed re-optimization at the first operator o_1.

6.4.3 Multi-Flow Optimization

Similarly to the cost-based vectorization, on-demand re-optimization can also be applied to the data-flow-oriented optimization technique multi-flow optimization that has been discussed in Chapter 5. It is based on the algorithms of deriving partitioning schemes (A-DPA) and on the waiting time computation (A-WTC). Furthermore, we follow the optimization objective of minimizing the total latency time with

$$\phi = \max \frac{|M'|}{\Delta t} = \min T_L(M'), \tag{6.11}$$

where $T_L(M')$ is computed by

$$\hat{T}_L(M', k') = \left\lceil \frac{|M'|}{k'} \right\rceil \cdot \Delta tw + \hat{W}(P', k'). \tag{6.12}$$

and $\hat{W}(P', k')$ is computed by $\hat{W}(P', k') = W^-(P') + W^+(P') \cdot k'$ for arbitrary $k' = R \cdot \Delta tw$. Due to our specific cost model extension, this minimum is given at $\Delta tw = W(P', \Delta tw \cdot R)$ such that we can compute Δtw by $\Delta tw = W^-(P')/(1 - W^+(P') \cdot R)$. In order to ensure the latency time constraint at the same time, we additionally evaluate the validity condition of $(0 \leq W(P', k') \leq \Delta tw) \wedge (0 \leq \hat{T}_L \leq lc)$. In general, there are different cases, which reason different optimality conditions. Here, we concentrate on the default case, where the computed waiting time fulfills the validity condition.

For a plan P with $m = 5$ operators and $h = 2$ partitioning attributes, there exist $h + 3 = 5$ optimality conditions. First, $h - 1 = 1$ optimality conditions are required with regard to the derivation of partitioning schemes, where we order the partitioning attributes according to the monitored selectivities with $oc_1 : sel(ba_1) \geq sel(ba_2)$. Second, for this case of a valid waiting time, we require four optimality conditions—independent of the number of partitioning attributes—in order to represent the validity condition with $oc_2 : 0 \leq W(P', k')$, $oc_3 : W(P', k') \leq \Delta tw$, $oc_4 : 0 \leq \hat{T}_L$, and $oc_5 : \hat{T}_L \leq lc$).

The `PlanOptTree` that represents these optimality conditions is shown in Figure 6.13. Essentially, it contains operator nodes for all five operators. Only for operators with partitioning attributes, we monitor the selectivity according to this attribute, while for all

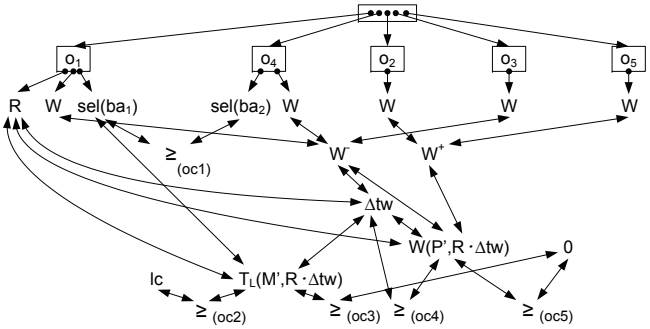


Figure 6.13: Example `PlanOptTree` of Multi-Flow Optimization

operators, we monitor the execution time. In addition, the message rate R is monitored for the first operator of this plan. We use oc_1 to express the ordering of the partition tree. In contrast, for the validity condition, we require a hierarchy of complex statistic nodes. First, we determine the cost components $W^-(P')$ and $W^+(P')$ according to the defined cost model extension. Then, Δtw is computed from these cost components and the message rate. Furthermore, we compute the latency time and the execution time using the determined waiting time Δtw. Finally, we use the optimality conditions oc_2-oc_5 in order to express the mentioned validity condition, where two more complex statistic nodes (latency constraint lc and 0) are used as constant-value operands.

Once we triggered re-optimization, we can use the `PlanOptTree` for directed re-optimization as well. There are three facets, where we can exploit the `PlanOptTree`. First, we use the violated optimality conditions for the directed reordering of partitioning attributes similarly to the ordering of selective operators. Second, with regard to the different cases of computing Δtw (minimum, default, exceeded latency), we can directly derive the case from the violated optimality conditions and compute the waiting time Δtw accordingly. Third, after each executed partitioned plan instance, we determine the current (continuously adapted) waiting time Δtw by querying the corresponding complex statistic node. Most importantly this allows for a workload adaptation with almost no adaptation delay as it was introduced by the optimization interval Δt. Therefore, we also solved the problem that the maximum message latency time cannot be guaranteed if workload characteristics change abruptly and if we use long optimization intervals.

6.4.4 Discussion

We generalize the main findings from the given example optimization techniques. First, even when considering techniques with a large search space, only few optimality conditions are required because we do not model the complete plan search space but only the conditions of the optimal plan. Hence, all conditions are binary decisions of subplans and they exploit the fact that costs of operators in front of and after that subplan are independent of the rewriting decision. Second, only one optimality condition per dependency (plus one condition for binary operators) is required to model plan optimality. As a result, those conditions can be seamlessly integrated into existing memo structures that are typically used during query optimization in DBMS (for an example, see the cascades framework in MS SQL Server [BN08]). Third, due to the possibility of arbitrarily complex optimality conditions, all kinds of optimization techniques can be integrated. For example, our optimizer uses (1) reordering techniques (e.g., switch path reordering, early selection, early

projection, early translation, early group-by, join enumeration), (2) techniques that insert or remove specific operators (e.g., execution pushdown, order-by insertion, rewriting sequences/iteration to parallel flows), and (3) techniques that choose between different physical operator implementations (e.g., join type selection, setoperation type selection). Fourth, existing cost models (e.g., cost formulas and aspects such as correlation) can be reused. Only the optimizer interface and the single optimization techniques require some changes with regard to (1) expressing optimality with partial `PlanOptTrees` and (2) allowing for directed re-optimization of subplans. We explained these modifications for selected optimization techniques. In conclusion, the concept of on-demand re-optimization is generally applicable for existing optimizer architectures as well.

6.5 Experimental Evaluation

In this section, we present selected experimental evaluation results comparing the on-demand re-optimization with the periodical re-optimization that has been evaluated already in Chapter 3. Therefore, we concentrate on the comparison of the different optimization models rather than comparing the on-demand re-optimization approach with the unoptimized execution, or regarding the benefit of individual optimization techniques. In general, the experiments show that:

- Most importantly, the cumulative costs of plan execution are reduced due to the fast adaptation to changing workload characteristics that prevents missed optimization opportunities. In addition to the fast adaptation, the total execution time does not depend on the chosen optimization interval anymore.
- The re-optimization costs are typically reduced because re-optimization is only triggered if required. Re-optimization might be triggered more frequently than for periodical re-optimization in case of continuously changing workload characteristics. For this reason of ensuring robustness, the use of our lightweight correlation table has higher importance as for the periodical re-optimization.
- For complex integration flows, directed re-optimization significantly improves the optimization time. This is especially true for optimization techniques with super-linear time complexity, where the benefit of reducing the number of evaluated combinations has huge impact on the overall optimization time.
- The overhead of evaluating optimality conditions for each monitored atomic statistic is fairly low due to minimal statistic maintenance. Furthermore, this low overhead is negligible compared to the cumulative execution time improvements.

In detail, the description of our experimental results is structured as follows. First, we present the end-to-end comparison of on-demand re-optimization with periodical and no optimization using simple and complex integration flows. We also show the scalability with varying parameters. Second, we discuss the benefit of directed re-optimization and the overhead of maintaining optimality conditions. Third, we explain the effects of correlation and how we achieve robustness of on-demand re-optimization.

Experimental Setting

We implemented the on-demand re-optimization, within our `WFPE` (workflow process engine). This includes the `PlanOptTree` data structure and related algorithms, as well as

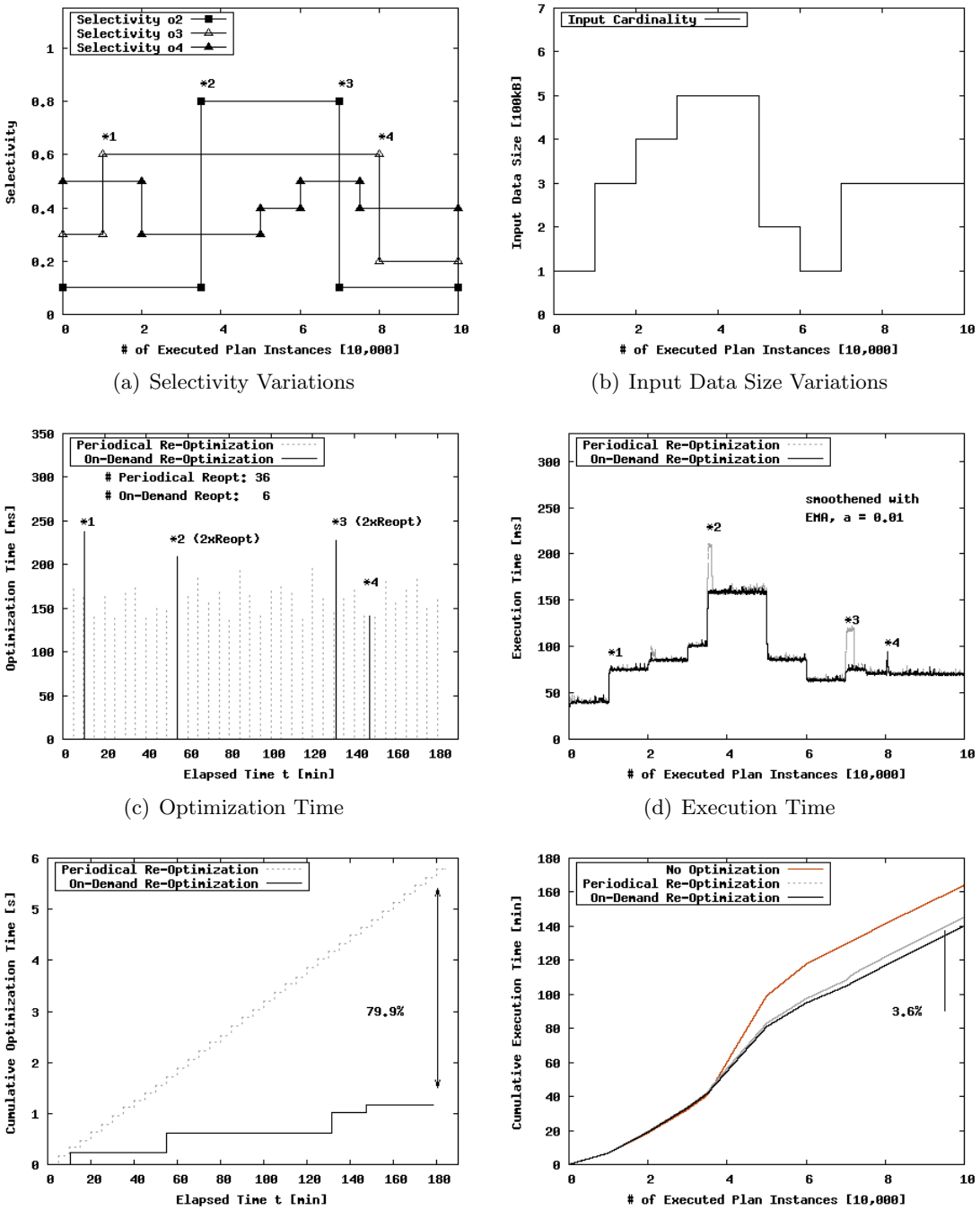
abstractions of statistics maintenance, cost estimation and re-optimization for enabling the use of both alternative optimization models. Most importantly, we extended the optimizer interface in the form of exchanging partial `PlanOptTrees`, which includes the modification of optimization techniques in order to enable directed re-optimization. Furthermore, we ran the experiments on the same platform as used within the rest of this thesis. With the aim of simulating arbitrary changing workload characteristics, we synthetically generated XML data sets with varying selectivities and cardinalities as input for our integration flows.

Simple-Plan End-to-End Comparison

In a first series of experiments, we compared periodical re-optimization with on-demand re-optimization (both asynchronous with inter-instance plan change). In order to conduct a fair evaluation, we used our fairly simple example plan P_5 because the benefit of on-demand re-optimization increases with increasing plan complexity. Essentially, we reused the end-to-end comparison experiment from Chapter 3, where we executed 100,000 instances for the non-optimized plan as well as for both optimization approaches, and we then measured re-optimization and plan execution times. The execution time already includes the synchronous statistic maintenance and the evaluation of optimality conditions in case of on-demand re-optimization. During execution, we varied the selectivities of the three selection operators (see Figure 6.14(a)) and the input cardinality (see Figure 6.14(b)). The input data was generated without correlations. With regard to re-optimization, there are four points ($*1$, $*2$, $*3$, and $*4$) where a workload change (intersection points between selectivities) reasons the change of the optimal plan. For periodical re-optimization, we used a period of $\Delta t = 300\,\text{s}$, while for on-demand re-optimization, we used a minimum existence time of $\Delta t = 1\,\text{s}$ and a lazy condition evaluation count of ten.

Figure 6.14(c) shows the re-optimization times, while Figure 6.14(e) illustrates the cumulative optimization time. There, we used the elapsed scenario time as the x-axis in order to illustrate the characteristics of periodical re-optimization. We see that periodical re-optimization requires many unnecessary optimization steps (36 steps), while on-demand re-optimization is only triggered if a new plan will be found (6 steps). For workload shifts $*2$ and $*3$, two on-demand re-optimizations were triggered due to the two intersection points of selectivities and the used exponential moving average, which caused a small statistics adaptation delay that led to exceeding the lazy count before converging to the final statistic measure. With a different parameterization, only one re-optimization was triggered for each workload shift. Despite directed re-optimization, a single optimization step is, on average, slightly slower than a full re-optimization step due to the small optimization search space of the applied optimization techniques (selection reordering and switch path reordering). The reason is that directed re-optimization requires some constant additional efforts and benefits only if several operators can be ignored during optimization. Further, the re-optimization time of this plan is dominated by the physical plan compilation and the waiting time for the next possible exchange of plans (due to the asynchronous optimization). As shown in Chapter 3, if we would use a larger Δt for periodical re-optimization, we would use suboptimal plans for a longer time and hence, we would miss optimization opportunities. As a result, over time, on-demand re-optimization yields optimization time improvements because it requires fewer re-optimization steps.

Figures 6.14(d) and 6.14(f) show the measured execution times. The different execution times are caused by the changing workload characteristics in the sense of different input



(a) Selectivity Variations

(b) Input Data Size Variations

(c) Optimization Time

(d) Execution Time

(e) Cumulative Optimization Time

(f) Cumulative Execution Time

Figure 6.14: Simple-Flow Comparison Scenario

cardinalities and operator selectivities. When using periodical re-optimization, the often re-occurring small peaks are caused by the numerous asynchronous re-optimization steps. A major characteristic of periodical re-optimization is that after a certain workload shift, it takes a while until periodical re-optimization is triggered. During this time, the plan execution time is much longer than the execution time of the optimal plan. In contrast, on-

demand re-optimization directly reacts to the workload change and switches plans. Hence, over time, execution time reductions are yielded due to the fast adaptation. In order to achieve this with periodical re-optimization, a really small Δt is required. However, this would increase the total re-optimization time. Finally, note that the small differences of absolute execution times compared to Chapter 3 are caused by randomly generated messages for this experiment according to the given selectivities.

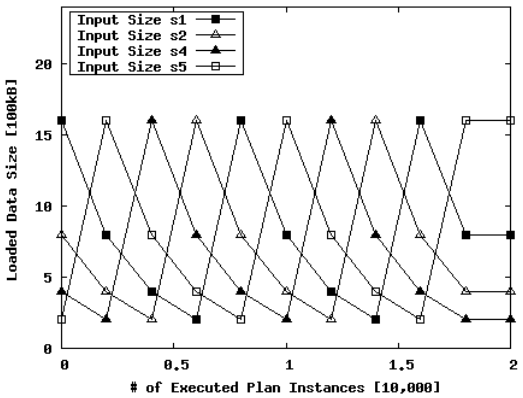
Complex-Plan End-to-End Comparison

In addition to the simple-plan scenario, we executed a second series of experiments using the more complex example plan P_7'. The difference of plan P_7' compared to the already introduced plan P_7 is that we explicitly changed the join query type *chain* to a *clique* type in order to have the possibility of arbitrary join reordering. This plan receives a data set, loads data from four different systems, and executes schema transformations with translation operators (XSLT scripts). Finally, the five data sets are joined using four join operators and the result is sent to a fifth system. We executed 20,000 plan instances using the input cardinalities shown in Figure 6.15(b) and the same parameter configurations as for the simple plan scenario. For the four loaded data sets, we used input cardinalities of $d \in \{2, 4, 8, 16\}$ (in 100 kB). After every 2,000 instances, we changed the input cardinalities of the four external systems round-robin as shown in Figure 6.15(a).

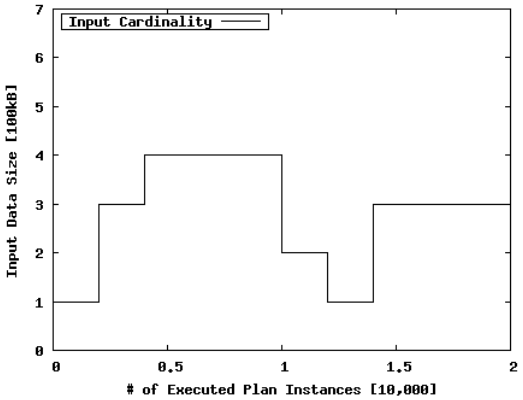
Regarding the results that are shown in Figures 6.15(c)-6.15(f), we observe similar characteristics as within the simple-plan scenario. It is important to note that (1) the higher the optimization opportunities of a plan, and (2) the higher the number of workload changes, the higher the execution time improvements achieved by on-demand re-optimization due to the higher importance of immediate adaptation (see Figures 6.15(d) and 6.15(f)). However, we restricted ourself to moderate differences of input data sizes in order to show only this main characteristic rather than showing arbitrarily high improvements. Further, we also observe, on average, higher re-optimization time improvements as within the simple-plan comparison scenario due to the higher influence of immediate re-optimization and directed re-optimization (see Figures 6.15(c) and 6.15(e)). The outliers for periodical re-optimization have been cause by the Java garbage collection because our implementation of the full join enumeration (DPSize) requires many temporary data objects, which are lazily deleted if space is required. This effect is only visible for periodical re-optimization because there, we used full join enumeration, where more objects have been created and the join enumeration is invoked more often than our on-demand re-optimization. In conclusion, the benefit of on-demand re-optimization increases with increasing plan complexity and increasing frequency of workload changes.

Scalability

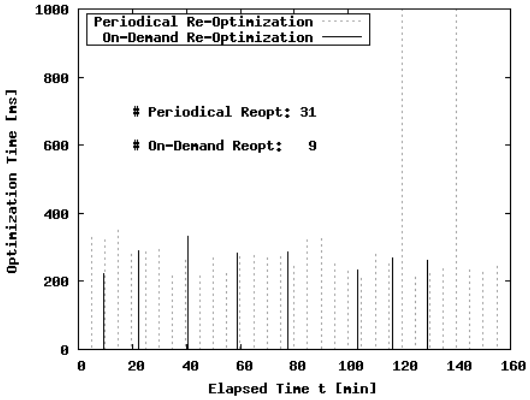
In order to investigate the influencing aspects such as the number of workload shifts wc, the input data size d and the optimization interval Δt in more detail, we conducted an additional series of scalability experiments, where we varied these parameters. We compare the unoptimized case with periodical and on-demand re-optimization using the cumulated plan execution time, the cumulated optimization time, and the number of re-optimizations. As the experimental setup, we executed 5,000 plan instances of P_5 for each configuration, where each workload shift switches between the two selectivity configurations of $(sel(o_2) = 0.8, sel(o_3) = 0.6, sel(o_4) = 0.1)$ and $(sel(o_2) = 0.1, sel(o_3) = 0.6, sel(o_4) = 0.8)$. As



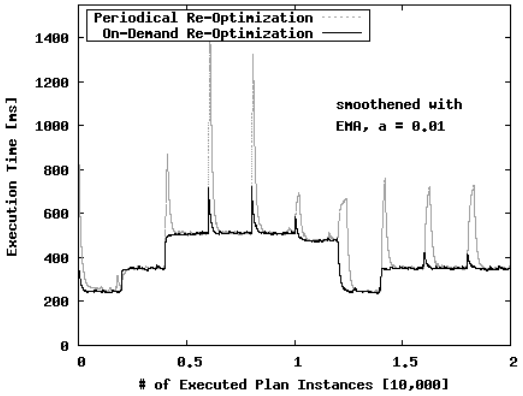
(a) Loaded Data Size Variations



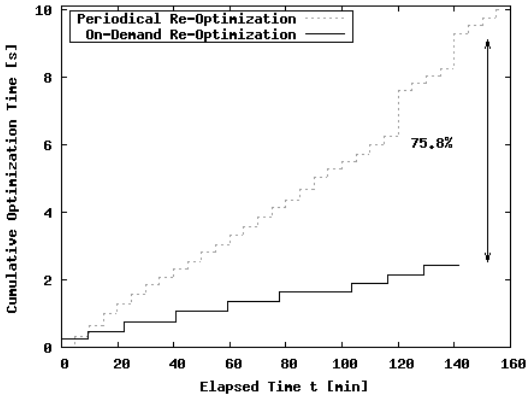
(b) Input Data Size Variations



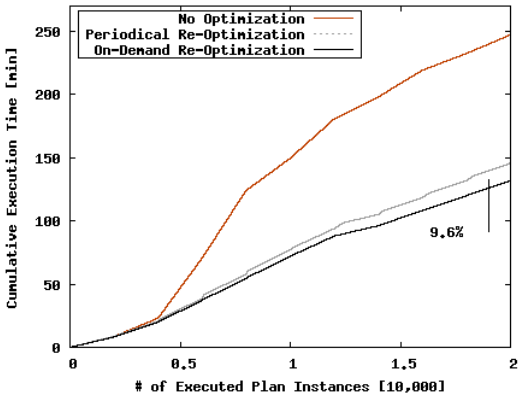
(c) Optimization Time



(d) Execution Time



(e) Cumulative Optimization Time

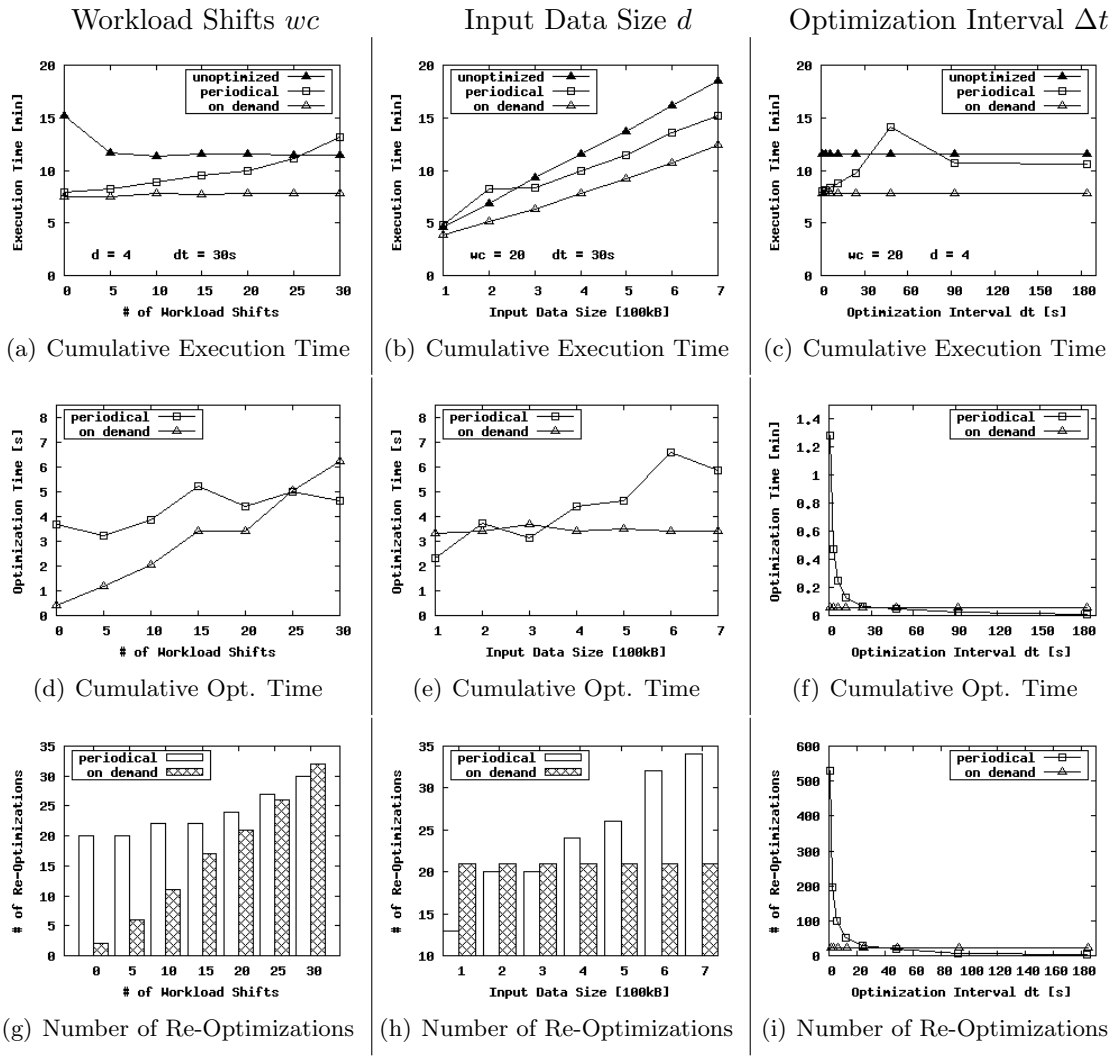


(f) Cumulative Execution Time

Figure 6.15: Complex-Flow Comparison Scenario

default values, we used a number of workload shifts $wc = 20$, an input data size $d = 4$, and an optimization interval of $\Delta t = 30\,\text{s}$. Figure 6.16 illustrates the results.

First, we varied the number of workload shifts $wc \in [0, 30]$. The unoptimized execution shows almost constant execution time (Figure 6.16(a)) except the configuration with no workload shifts because the plan is not optimal for the whole scenario, while for



(a) Cumulative Execution Time (b) Cumulative Execution Time (c) Cumulative Execution Time

(d) Cumulative Opt. Time (e) Cumulative Opt. Time (f) Cumulative Opt. Time

(g) Number of Re-Optimizations (h) Number of Re-Optimizations (i) Number of Re-Optimizations

Figure 6.16: Scalability of Plan P_5 Varying Influencing Parameters

higher numbers of workload shifts the state alternate between optimal and non-optimal. In contrast, on-demand re-optimization shows constant execution time, because it directly reacts to any workload shift and the re-optimization time is no dominating factor, while the execution time of periodical re-optimization degrades for increasing number of workload shifts because we use non-optimal plans more often. Clearly, the optimization time of on-demand re-optimization increases with increasing number of workload shifts (Figures 6.16(d) and 6.16(g)), while periodical re-optimization required an almost constant number of re-optimizations steps. The small increase of optimization steps is reasoned by the increased total execution time.

Second, we varied the input data size $d \in [1, 7]$ (in 100 kB). Due to the increasing absolute optimization benefit per single optimizations step with increasing data size, the absolute improvement of on-demand re-optimization also increases compared to the unoptimized execution and periodical re-optimization (Figure 6.16(b)). On-demand re-optimization shows a constant re-optimization time (Figure 6.16(e)) because the number of workload shifts was fixed to $wc = 20$. In contrast, the optimization time of period-

ical re-optimization increases due to the increasing total execution time because more re-optimization steps have taken place (Figure 6.16(h)).

Third, we varied the optimization interval $\Delta t \in \{1\,\text{s}, 3\,\text{s}, 6\,\text{s}, 12\,\text{s}, 24\,\text{s}, 48\,\text{s}, 92\,\text{s}, 184\,\text{s}\}$. Obviously, the execution time of unoptimized execution, and on-demand re-optimization are independent of this optimization interval (Figure 6.16(c)). The same is also true for the optimization time and the number of optimization steps of on-demand re-optimization (Figures 6.16(f) and 6.16(i)). In contrast, the optimization interval is a high-influence parameter for periodical re-optimization. For an increasing optimization interval, we observe a degradation of the execution time because we miss more optimization opportunities due to longer adaptation delays. Interestingly, for specific configurations, the execution time is even significantly worse than the unoptimized execution because we switch to the optimal plan just before the next workload shift occurs. However, if we further increase the optimization interval, the execution time converges to the unoptimized case. The benefit of on-demand re-optimization mainly depends the benefit of applied optimization techniques. For really short optimization intervals of several seconds, we observe the best configuration of periodical re-optimization, which is still slower than on-demand re-optimization. For these configurations, we observe a significant increase of the cumulative re-optimization time because the number of re-optimization steps increased up to 544.

Finally, we can summarize that on-demand re-optimization always lead to the lowest execution time, where we observed execution time reductions compared to periodical re-optimization of up-to 44.7%. However, the maximum benefit depends on applied optimization techniques, where in this setting we only used selection re-ordering. Furthermore, on-demand re-optimization shows much better robustness of execution time when varying the tested influencing aspects.

Directed Re-Optimization Benefit

In addition, we evaluated the benefit of directed re-optimization, where we re-used the different plans from Chapter 3 that included a varying number of join operators with $m = \{2, 4, 6, 8, 10, 12, 14\}$ as a clique query (all directly connected). We compared the optimization time of (1) the full join enumeration using the standard DPSize algorithm [Moe09], (2) our join enumeration heuristic with quadratic time complexity that we use if the number of joins exceed eight joins (see Section 3.3.2), and (3) our directed re-optimization that only takes operators into account that are included in violated optimality conditions. Before optimization, we randomly generated statistics for input cardinalities and join selectivities. For directed re-optimization, we used an optimized plan and randomly changed the statistics of one join. The experiment was repeated 100 times.

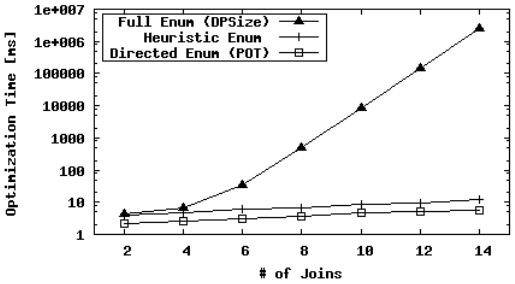
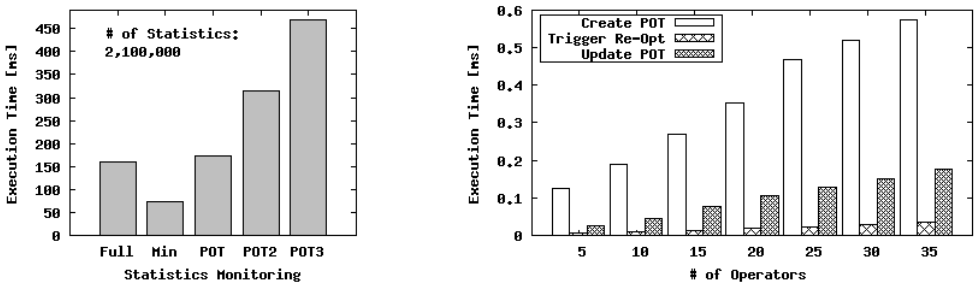


Figure 6.17: Directed Join Enumeration Benefit

Figure 6.17 illustrates the results using a log-scaled y-axis. The optimization time of full join enumeration increases exponentially, while for both heuristic and directed re-optimization, the optimization time increases almost linearly (slightly super-linearly). In addition, directed re-optimization is even faster than the heuristic enumeration because we only reorder quantifiers of violated optimality conditions. Due to randomly generated statistics, on average, we take fewer operators into consideration, while still ensuring to find the global optimal solution over multiple re-optimization steps. As a result, with increasing plan complexity, the relative benefit of directed re-optimization increases.

On-Demand Re-Optimization Overhead

We additionally evaluated the overheads of statistics maintenance and of the `PlanOptTree` algorithms. Here, all experiments were repeated 100 times. It is important to note that both overheads were already included in the end-to-end comparison, where they have been amortized by the execution time improvements in several orders of magnitude.



(a) Statistic Maintenance (b) Algorithm Overhead

Figure 6.18: Overhead of `PlanOptTrees`

Figure 6.18(a) illustrates the statistic maintenance overhead comparing our Estimator component (used for periodical re-optimization) versus our `PlanOptTree` (used for on-demand re-optimization), which both use statistics of all 100,000 plan instances of the simple-plan comparison scenario at the granularity of single operators (2,100,000 atomic statistic tuples). For both models, we used the exponential moving average as aggregation method. The costs include the transient maintenance of aggregates for all operators as well as the aggregation itself. Full Monitoring refers to periodical re-optimization, where all statistics of all operators are gathered by the Estimator. Min Monitoring refers to a hypothetical scenario, where we know the required statistics and maintain only these statistics with the Estimator. The relative improvement of Min Monitoring to Max Monitoring is the benefit we achieve by maintaining only relevant statistics, where the absolute benefit depends on the used workload aggregation method. In contrast, POT Monitoring refers to the use of our `PlanOptTree`. Although the `PlanOptTree` declines unnecessary statistics, it is slower than full monitoring because for each statistic tuple, we compute the hierarchy of complex statistics and evaluate optimality conditions. Therefore, we distinguish three variants of the POT monitoring: POT refers to the monitoring without condition evaluation, while POT2 (without the `MEMO` structure) and POT3 (with the `MEMO` structure) show the overhead of continuously evaluating the optimality conditions. It is important to note that the use of the `MEMO` structure is counterproductive in this scenario due to the rather simple optimality conditions and additional overhead for updating and

evaluating the MEMO structure. However, due to the requirement of robustness, we use the POT3 configurations for all end-to-end comparison scenarios. Furthermore, we show only the total statistic maintenance time because these times scale linear with an increasing number of statistic tuples due to the use of the incremental EMA aggregation method. Finally, compared to the performance benefit of on-demand re-optimization, the overhead for statistic maintenance is negligible.

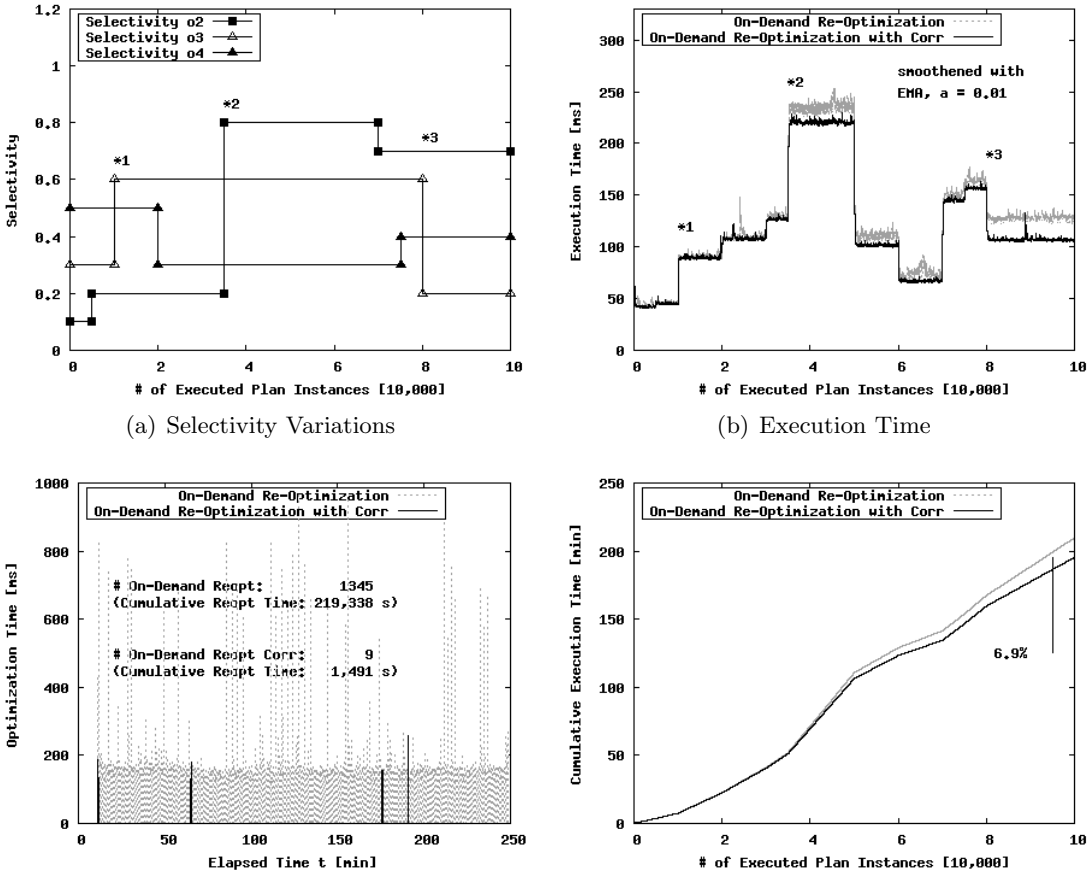
Additionally, we evaluated the algorithm overhead when creating, using and updating PlanOptTrees. There, we used the already described POT3 configuration. Figure 6.18(b) illustrates the results, varying the number of operators m of a plan and hence, indirectly varying the number of optimality conditions. The PlanOptTree is created over all m operators (where we simulated the actual optimization techniques and directly provided the partial PlanOptTrees), while triggering re-optimization and the subsequent update of the PlanOptTree addressed violated optimality conditions only, which depend on the randomly changed statistics. We observe a moderate execution time, where the creation and the update of PlanOptTrees were dominated by the merging of partial PlanOptTrees. Due to the different numbers of addressed optimality conditions the update of PlanOptTrees is more efficient than the creation of an initial PlanOptTree. It is important to note the almost linear scaling of creating PlanOptTree, triggering re-optimization and updating PlanOptTrees. In conclusion, the overhead of PlanOptTree algorithms is fairly low and can be neglected as well because it is only required for initial deployment or during on-demand re-optimization.

Robustness

Compared to periodical re-optimization, on-demand re-optimization is more sensitive with regard to workload changes because it directly reacts on detected violated optimality conditions. According to the robustness of optimization benefits, there are two major effects that are worth mentioning. First, the on-demand re-optimization does not require the specification of an optimization interval. Therefore, it is more robust to arbitrary workload changes compared to periodical re-optimization as shown in Figure 6.16. Second, for on-demand re-optimization there is a higher risk of frequently changing plans due to correlated data or almost equal selectivities that alternate around equality (see Example 6.5). Therefore, we introduced the strategies of the correlation table, minimum existence time and lazy condition evaluation. The minimum existence time simply ensures a lower bound of time between two subsequent re-optimization steps and thus, linearly reduces the number of re-optimization steps. Furthermore, the strategy of lazy condition evaluation overcomes the problem of almost equal selectivities. While the effects of these two strategies are fairly obvious, the use of the correlation table requires a more detailed discussion using a series of experiments.

Therefore, we reused the correlation experiment of Chapter 3 in order to evaluate the on-demand re-optimization on correlated data with and without the use of our lightweight correlation table. We executed 100,000 instances of our example plan P_5 and compared the resulting execution time. We used a minimum existence time of $\Delta t = 5\,\text{s}$, a lazy condition evaluation count of ten, and a re-optimization threshold of $\tau = 0.001$. Figure 6.19(a) recaps the conditional selectivities $P(o_2)$, $P(o_3|o_2)$, and $P(o_4|o_2 \wedge o_3)$ of the three Selection operators (with $P(o_3|\neg o_2) = 1$ and $P(o_4|\neg o_2 \vee \neg o_3) = 1$), which lead to a strong dependence (correlation) of o_3 on o_2 as well as of o_4 on o_2 and o_3.

Figures 6.19(c) and 6.19(b) illustrate the resulting optimization times and execution



(a) Selectivity Variations

(b) Execution Time

(c) Optimization Time

(d) Cumulative Execution Time

Figure 6.19: Simple-Flow Correlation Comparison Scenario

times with and without the use of our correlation table. We observe that without the use of the correlation table, the optimization technique selection reordering assumes statistical independence and thus, changed the plan back and forth, even in case of constant workload characteristics. Due to the direct triggering of re-optimization by on-demand re-optimization, overall 1,345 re-optimization steps have been executed. In addition to this re-optimization overhead, the permanent change between suboptimal and optimal plans led to a degradation of the execution time because non-optimal plans were used for long time horizons. In contrast, when using our correlation table, the required re-optimization steps were reduced to nine. It is important to note that, in case of using the correlation table, a single workload change triggers several re-optimization steps because we allow only one operator reordering per optimization step in order to learn the conditional selectivities stepwise, which reduce the risk for cyclic re-optimizations. However, over time, the conditional selectivity estimates converge to the real selectivities. Finally, as a combination of the reduced number of re-optimization steps and the prevention of using suboptimal plans due to unknown correlation, we achieved a 7% overall improvement with regard to the cumulative execution time (see Figure 6.19(d)). In conclusion, the use of our correlation table ensures robustness in the presence of correlated data or conditional probabilities, which has high importance when using on-demand re-optimization.

6.6 Summary and Discussion

The periodical re-optimization exhibits the drawbacks of (1) full monitoring of all operator statistics regardless if they are used by the optimizer or not, (2) many unnecessary periodical re-optimization steps, where for each step a full optimization is executed, (3) missed optimization opportunities due to potentially slow workload adaptation, and (4) the optimization period Δt as a high-influence parameter, whose configuration requires context knowledge of the integration flows as well as knowledge of the frequency of workload changes. These drawbacks are mainly reasoned by the strict separation between optimizer, statistics monitoring and plan execution.

In this chapter, we presented the concept of *on-demand re-optimization* to overcome these drawbacks. We introduced the `PlanOptTree` that models optimality of the current plan by exploiting context knowledge from the optimizer. With this approach, statistics are monitored according to optimality conditions and directed re-optimization is triggered if and only if such a condition is violated. In conclusion, this approach always reduces the total execution time because (1) if the workload does not change, we avoid unnecessary re-optimization steps, and (2) if there are workload changes, we do not miss any optimization opportunities due to direct re-optimization, while the overhead for evaluating optimality conditions is negligible. In addition, it allows for predictable performance without the need for elaborate parameterization. In conclusion, on-demand re-optimization has the same advantages but overcomes the disadvantages of periodical re-optimization.

However, on-demand re-optimization has also some limitations. Most importantly, the on-demand re-optimization is more sensitive with regard to workload changes than periodical re-optimization is. On the one side this is advantageous because we directly adapt to these changes and therefore, reduce the execution time. On the other side, more care with regard to robustness (stability) is required. For example, correlation-awareness is required because otherwise, we might result in frequent plan changes, which hurt performance. For this reason, we introduce the concepts of *correlation tables*, *minimal existence time*, and *lazy condition violation*, which ensures robustness of on-demand re-optimization.

In conclusion of our experimental evaluation, the on-demand re-optimization, in comparison to periodical re-optimization, achieves additional cumulative execution time improvements, while it requires much less re-optimization steps and therefore, significantly reduces the total optimization overhead. Furthermore, there are plenty issues for future work. This includes, for example, the investigation of (1) the extension of inter-instance re-optimization to intra-instance re-optimization (mid-query re-optimization) in order to support ad-hoc integration flows or long running plan instances, (2) specific approaches for directed re-optimization with regard to complex optimization techniques (e.g., join enumeration, eager group-by), and (3) the combination of progressive parametric query optimization with on-demand re-optimization in order to reuse generated physical plans. Although the on-demand re-optimization approach is tailor-made for integration flows that are deployed once and executed many times, it is also applicable in other areas. Examples for these areas are continuous queries in DSMS, re-occurring queries in DBMS, and incremental maintenance of data mining results.

7 Conclusions

I can't change the direction of the wind,
but I can adjust my sails to always reach my destination.
— Jimmy Dean

From the reactive perspective of an integration platform, we cannot change the workload characteristics (the direction of the wind) in the sense of incoming messages that initiate plan instances of integration flows. However, we can incrementally maintain execution statistics and use these for the cost-based optimization of integration flows (the adjustment of sails) in order to improve their execution time, latency and thus, the message throughput (in order to reach the destination). This allows the workload-based adjustment of the current plan and thus, for continuous adaptation to changing workload characteristics.

Based on emerging requirements of complex integration tasks that (1) stretch beyond simple read-only applications, (2) involve many types of heterogeneous systems and applications, and (3) require fairly complex procedural aspects, typically, imperative integration flows are used for specification and execution of these tasks. In this context, we observe that many independent instances of such integration flows with rather small amounts of data per instance are executed over time in order to achieve (1) high consistency between data of operational systems or (2) high up-to-dateness of analytical query results in data warehouse infrastructures. In addition to this high load of flow instances, the performance of source systems depends on the execution time and availability of synchronous data-driven integration flows. For these reasons, there are high performance demands on integration platforms that execute imperative integration flows.

To tackle this problem of high performance requirements on the execution of integration flows, we introduced the cost-based optimization of these imperative integration flows. In detail, we described the fundamentals of cost-based optimization including novel techniques such as the first entirely defined cost model for imperative integration flows, a transformation-based rewriting algorithm with several search space reduction approaches, the asynchronous periodical re-optimization as well as techniques for workload adaptation and handling of correlated data. Essentially, this cost-based optimizer exploits the major integration flow specific characteristics of being deployed once and executed many times with rather small amounts of data per instance. In addition, we introduced several control-flow and data-flow-oriented optimization techniques, where we adapted on the one side techniques from data management systems, and programming language compilers as well as on the other side, we defined techniques tailor-made for integration flows. Additionally, we described in detail two novel optimization techniques for throughput optimization of integration flows, namely the cost-based vectorization and the multi-flow optimization. Finally, we introduced the novel concept of on-demand re-optimization of integration flows that overcomes the major drawbacks of periodical re-optimization, while still exploiting the major characteristics of integration flows. Our experiments showed that significant execution time and throughput improvements are possible, while only moderate additional optimization overhead is required.

Existing approaches before this thesis either used the rule-based optimize-once model, where the integration flow is only optimized once during the initial deployment, or the optimize-always model, where optimization is triggered for each plan instance. In contrast, we presented the first entire *cost-based* optimizer for *imperative integration flows* using the cost-based optimization models of periodical and on-demand re-optimization. The major advantage of these new optimization models is robustness in terms of (1) high optimization opportunities that allows (2) the adaptation to changing workload characteristics with (3) low risk of optimization overheads.

Beside the investigation of additional optimization techniques, we see four major research challenges regarding future work of the cost-based optimization of integration flows:

- *Mid-Instance Optimization:* Our asynchronous re-optimization model is based on the assumption of many plan instances with rather small amounts of data per instance. In order to make it also suitable for (1) long running plan instances and (2) ad-hoc integration flows (as required for situational BI and mashup integration flows), an extension to synchronous mid-instance re-optimization would be necessary. The challenge is to define a hybrid model for both use cases of integration flows.
- *Multi-Objective Optimization:* We focused on the optimization objectives of execution time and throughput. However, facing new requirements, this might be extended by multiple objectives including for example, monetary measures, execution and latency time, throughput, energy consumption, resiliency or transactional guarantees.
- *Optimization of Multiple Deployed Plans:* The cost-based optimization of integration flows—with few exceptions—aims to optimize a single deployed plan. However, typically, multiple different integration flows are deployed and concurrently executed by the integration platform. The major question is if we could exploit the knowledge about workload characteristics of all plans and their inter-influences for more efficient scheduling of plan instances or influence-aware plan rewriting.
- *Optimization of Distributed Integration Flows:* With regard to load balancing and the emerging trend towards virtualization, distributed integration flows might be used as well. The resulting research challenge is to optimize the entire distributed integration flow, rather than just the subplans of local server nodes. One aspect of this might be the extension of cost-based vectorization to the distributed case.

Thus, we can conclude that there are many directions for future work in this new field of the *cost-based optimization of integration flows*. Similarly to cost-based query optimization, this might be a starting point for the continuous development of new execution and optimization techniques:

> *"In my view, the query optimizer was the first attempt at what we call autonomic computing or self-managing, self-timing technology. Query optimizers have been 25 years in development, with enhancements of the cost-based query model and the optimization that goes with it, and a richer and richer variety of execution techniques that the optimizer chooses from. We just have to keep working on this. It's a never-ending quest for an increasingly better model and repertoire of optimization and execution techniques. So the more the model can predict what's really happening in the data and how the data is really organized, the closer and closer we will come [to the ideal system]."*
>
> — Patricia G. Selinger, IBM Research – Almaden, 2003 [Win03]

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[ABD+10] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. On the Stability of Plan Costs and the Costs of Plan Stability. *Proceedings of the VLDB Endowment*, 3(1):1137–1148, 2010.

[AC99] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD Conference*, pages 181–192, 1999.

[ACG+08] Himanshu Agrawal, Girish Chafle, Sunil Goyal, Sumit Mittal, and Sougata Mukherjea. An Enhanced Extract-Transform-Load System for Migrating Data in Telecom Billing. In *ICDE*, pages 1277–1286, 2008.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*, pages 261–272, 2000.

[AHL+04] Ashraf Aboulnaga, Peter J. Haas, Sam Lightstone, Guy M. Lohman, Volker Markl, Ivan Popivanov, and Vijayshankar Raman. Automated Statistics Collection in DB2 UDB. In *VLDB*, pages 1146–1157, 2004.

[AN08] Alexander Albrecht and Felix Naumann. Managing ETL Processes. In *NTII*, pages 12–15, 2008.

[AN09] Alexander Albrecht and Felix Naumann. METL: Managing and Integrating ETL Processes. In *VLDB Ph.D. Workshop*, pages 1–6, 2009.

[ANY04] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD Conference*, pages 359–370, 2004.

[AT08] David Aumüller and Andreas Thor. Mashup-Werkzeuge zur Ad-hoc Datenintegration im Web. *Datenbankspektrum*, 26:4–10, 2008.

[BABO+09] Manish Bhide, Manoj Agarwal, Amir Bar-Or, Sriram Padmanabhan, Srinivas Mittapalli, and Girish Venkatachaliah. XPEDIA: XML ProcEssing for Data IntegrAtion. *Proceedings of the VLDB Endowment*, 2(2):1330–1341, 2009.

[BB05] Shivnath Babu and Pedro Bizarro. Adaptive Query Processing in the Looking Glass. In *CIDR*, pages 238–249, 2005.

[BBD05a] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive Re-optimization. In *SIGMOD Conference*, pages 107–118, 2005.

[BBD05b] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive Re-optimization with Rio. In *SIGMOD Conference*, pages 936–938, 2005.

[BBD09] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive Parametric Query Optimization. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):582–594, 2009.

[BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator Scheduling for Memory Minimization in Data Stream Systems. In *SIGMOD Conference*, pages 253–264, 2003.

[BBDW05] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-Based Routing: Different Plans for Different Data. In *VLDB*, pages 757–768, 2005.

[BBH+08a] Matthias Böhm, Jürgen Bittner, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Improving Data Independence, Efficiency and Functional Flexibility of Integration Platforms. In *CAiSE Forum*, pages 97–100, 2008.

[BBH+08b] Matthias Böhm, Jürgen Bittner, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Model-Driven Generation of Dynamic Adapters for Integration Platforms. In *MDISIS*, pages 1–15, 2008.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. In *SIGMOD Conference*, pages 211–222, 2001.

[BEA04] BEA and IBM. *BPELJ: BPEL for Java*, 2004.

[BEA08] BEA and Oracle. *Java Process Definition (JPD)*, 2008.

[Beh09] Andreas Behrend. A Magic Approach to Optimizing Incremental Relational Expressions. In *IDEAS*, pages 12–22, 2009.

[Bel34a] E. T. Bell. Exponential Numbers. *Annals of Mathematics*, 35:411–419, 1934.

[Bel34b] E. T. Bell. Exponential Polynomials. *American Mathematical Monthly*, 41:258–277, 1934.

[BGLJ10] Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. Polynomial Heuristics for Query Optimization. In *ICDE*, pages 589–600, 2010.

[BGMP79] Mike W. Blasgen, Jim Gray, Michael F. Mitoma, and Thomas G. Price. The Convoy Phenomenon. *SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[BH08] Philip A. Bernstein and Laura M. Haas. Information Integration in the Enterprise. *Communications of the ACM*, 51(9):72–79, 2008.

[BHL10] Matthias Böhm, Dirk Habich, and Wolfgang Lehner. Multi-Process Optimization via Horizontal Message Queue Partitioning. In *ICEIS*, pages 5–14, 2010.

[BHL11] Matthias Böhm, Dirk Habich, and Wolfgang Lehner. Multi-Flow Optimization via Horizontal Message Queue Partitioning. In *ICEIS Postproceedings*, LNBIP, pages 1–16. Springer, 2011.

[BHLW08a] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. An Advanced Transaction Model for Recovery Processing of Integration Processes. In *ADBIS (local proceedings)*, pages 90–105, 2008.

[BHLW08b] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. DIPBench: An Independent Benchmark for Data-Intensive Integration Processes. In *ICDE Workshops*, pages 214–221, 2008.

[BHLW08c] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. DIPBench Toolsuite: A Framework for Benchmarking Integration Systems. In *ICDE*, pages 1596–1599, 2008.

[BHLW08d] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Message Indexing for Document-Oriented Integration Processes. In *ICEIS (1)*, pages 137–142, 2008.

[BHLW08e] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Model-Driven Development of Complex and Data-Intensive Integration Processes. In *MBSDI*, pages 31–42, 2008.

[BHLW08f] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Model-Driven Generation and Optimization of Complex Integration Processes. In *ICEIS (1)*, pages 131–136, 2008.

[BHLW08g] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Workload-Based Optimization of Integration Processes. In *CIKM*, pages 1479–1480, 2008.

[BHLW09a] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. GCIP: Exploiting the Generation and Optimization of Integration Processes. In *EDBT*, pages 1128–1131, 2009.

[BHLW09b] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Invisible Deployment of Integration Processes. In *ICEIS*, pages 53–65, 2009.

[BHLW09c] Matthias Böhm, Dirk Habich, Wolfgang Lehner, and Uwe Wloka. Systemübergreifende Kostennormalisierung für Integrationsprozesse. In *BTW*, pages 67–86, 2009.

[BHP+09a] Matthias Böhm, Dirk Habich, Steffen Preißler, Wolfgang Lehner, and Uwe Wloka. Cost-Based Vectorization of Instance-Based Integration Processes. In *ADBIS*, pages 253–269, 2009.

[BHP+09b] Matthias Böhm, Dirk Habich, Steffen Preißler, Wolfgang Lehner, and Uwe Wloka. Vectorizing Instance-Based Integration Processes. In *ICEIS*, pages 40–52, 2009.

[BHP+11] Matthias Böhm, Dirk Habich, Steffen Preißler, Wolfgang Lehner, and Uwe Wloka. Cost-Based Vectorization of Instance-Based Integration Processes. *Information Systems*, 36(1):3 – 29, 2011.

[BHW+07] Matthias Böhm, Dirk Habich, Uwe Wloka, Jürgen Bittner, and Wolfgang Lehner. Towards Self-Optimization of Message Transformation Processes. In *ADBIS Research Communications*, pages 1–10, 2007.

[Bit05] Juergen Bittner. *Möglichkeiten zur Integration verschiedener Datenbanken und DB-Applikationen*, 2005. 108. Datenbankstammtisch, HTW Dresden, 15. Juni 2005.

[BJ10] Andreas Behrend and Thomas Jörg. Optimized Incremental ETL Jobs for Maintaining Data Warehouses. In *IDEAS*, pages 1–9, 2010.

[BJR94] George Box, Gwilym M. Jenkins, and Gregory Reinsel. *Time Series Analysis: Forecasting and Control*. Wiley Series in Probability and Statistics, 1994.

[BK09] Alexander Böhm and Carl-Christian Kanne. Processes Are Data: A Programming Model for Distributed Applications. In *WISE*, pages 43–56, 2009.

[BKM07] Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Demaq: A Foundation for Declarative XML Message Processing. In *CIDR*, pages 33–43, 2007.

[BM07] Philip A. Bernstein and Sergey Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD Conference*, pages 1–12, 2007.

[BMI06] BMI. *Business Process Modelling Notation, Version 1.0*, 2006.

[BMK08] Alexander Böhm, Erich Marth, and Carl-Christian Kanne. The Demaq System: Declarative Development of Distributed Applications. In *SIGMOD Conference*, pages 1311–1314, 2008.

[BMM+04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive Ordering of Pipelined Stream Filters. In *SIGMOD Conference*, pages 407–418, 2004.

[BMS05] Jen Burge, Kamesh Munagala, and Utkarsh Srivastava. Ordering Pipelined Query Operators with Precedence Constraints. Technical Report 2005-40, Stanford InfoLab, 2005.

[BN08] Nicolas Bruno and Rimma V. Nehme. Configuration-Parametric Query Optimization for Physical Design Tuning. In *SIGMOD Conference*, pages 941–952, 2008.

[BPA06] Biörn Biörnstad, Cesare Pautasso, and Gustavo Alonso. Control the Flow: How to Safely Compose Streaming Services into Business Processes. In *IEEE SCC*, pages 206–213, 2006.

[Bro09] Juliane Browne. Brewer's CAP Theorem, 2009. `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem`.

[BW04] Shivnath Babu and Jennifer Widom. StreaMon: An Adaptive Engine for Stream Query Processing. In *SIGMOD Conference*, pages 931–932, 2004.

[BWH+07] Matthias Böhm, Uwe Wloka, Dirk Habich, Jürgen Bittner, and Wolfgang Lehner. Ein Nachrichtentransformationsmodell für komplexe Transformationsprozesse in datenzentrischen Anwendungsszenarien. In *BTW*, pages 562–581, 2007.

[CC08] Kajal T. Claypool and Mark Claypool. Teddies: Trained Eddies for Reactive Stream Processing. In *DASFAA*, pages 220–234, 2008.

[CCA08] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *SIGMOD Conference*, pages 739–752, 2008.

[CcR+03] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB*, pages 838–849, 2003.

[CD01] I. Cooper and J. Dilley. *Known HTTP Proxy/Caching Problems*, June 2001. RFC 3143.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD Conference*, pages 379–390, 2000.

[CEB+09] Nazario Cipriani, Mike Eissele, Andreas Brodt, Matthias Großmann, and Bernhard Mitschang. NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing. In *IDEAS*, pages 152–161, 2009.

[Cha09] Surajit Chaudhuri. Query Optimizers: Time to Rethink the Contract? In *SIGMOD Conference*, pages 961–968, 2009.

[CHK+07] Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. In *ICDE Workshops*, pages 624–633, 2007.

[CHX08] Zhao Chenhui, Duan Huilong, and Lu Xudong. An Integration Approach of Healthcare Information System. In *BMEI (1)*, pages 606–609, 2008.

[CKSV08] Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):230–245, 2008.

[CM95] Sophie Cluet and Guido Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *ICDT*, pages 54–67, 1995.

[CNP82] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal Data Partitioning in Database Design. In *SIGMOD Conference*, pages 128–136, 1982.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable Isolation for Snapshot Databases. In *SIGMOD Conference*, pages 729–738, 2008.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including Group-By in Query Optimization. In *VLDB*, pages 354–366, 1994.

[CWGN11] N. Cipriani, M. Wieland, M. Großmann, and D. Nicklas. Tool support for the design and management of context models. *Information Systems*, 36(1):99 – 114, 2011.

[Dad96] Peter Dadam. *Verteilte Datenbanken und Client/Server-Systeme: Grundlagen, Konzepte, Realisierungsformen*. Springer, 1996.

[DB07] Songyun Duan and Shivnath Babu. Processing Forecasting Queries. In *VLDB*, pages 711–722, 2007.

[DBDH08] Atreyee Dey, Sourjya Bhaumik, Harish D., and Jayant R. Haritsa. Efficiently Approximating Query Optimizer Plan Diagrams. *Proceedings of the VLDB Endowment*, 1(2):1325–1336, 2008.

[DCSW09] Umeshwar Dayal, Malú Castellanos, Alkis Simitsis, and Kevin Wilkinson. Data Integration Flows for Business Intelligence. In *EDBT*, pages 1–11, 2009.

[DD99] Ruxandra Domenig and Klaus R. Dittrich. An Overview and Classification of Mediated Query Systems. *SIGMOD Record*, 28(3):63–72, 1999.

[Des04] Amol Deshpande. An Initial Study of Overheads of Eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining Stream Statistics over Sliding Windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[DHR06] Amol Deshpande, Joseph M. Hellerstein, and Vijayshankar Raman. Adaptive Query Processing: Why, How, When, What Next. In *SIGMOD Conference*, pages 806–807, 2006.

[DHW+08] Stefan Dessloch, Mauricio A. Hernández, Ryan Wisnesky, Ahmed Radwan, and Jindan Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.

[dic09] *Digital Imaging and Communications in Medicine*, 2009. http://medical.nema.org/.

[DIR07] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[DS06] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *VLDB*, pages 715–726, 2006.

[EKMR06] Stephan Ewen, Holger Kache, Volker Markl, and Vijayshankar Raman. Progressive Query Optimization for Federated Queries. In *EDBT*, pages 847–864, 2006.

[Fac10] P. L. Fackler. *Generating Correlated Multidimensional Variates*, 2010. `http://www4.ncsu.edu/~fackler/andcorr.ps`.

[Gar09] Gartner. *Market Share: Application Infrastructure and Middleware Software, Worldwide, 2008*. Gartner, 2009.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S Declarative Stream Processing Engine. In *SIGMOD Conference*, pages 1123–1134, 2008.

[GGH00] Jane Grimson, William Grimson, and Wilhelm Hasselbring. The SI Challenge in Health Care. *Communications of the ACM*, 43(6):48–55, 2000.

[GHP+06] Kun Gao, Stavros Harizopoulos, Ippokratis Pandis, Vladislav Shkapenyuk, and Anastassia Ailamaki. Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. In *ICDE*, page 162, 2006.

[GLRG04] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Support for Relaxed Currency and Consistency Constraints in MTCache. In *SIGMOD Conference*, pages 937–938, 2004.

[Gra69] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[Gra90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD Conference*, pages 102–111, 1990.

[Gra94] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[GST96] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *VLDB*, pages 378–389, 1996.

[GYSD08a] Anastasios Gounaris, Christos Yfoulis, Rizos Sakellariou, and Marios D. Dikaiakos. A Control Theoretical Approach to Self-Optimizing Block Transfer in Web Service Grids. *ACM Transactions on Autonomous and Adaptive Systems*, 3(2):1–30, 2008.

[GYSD08b] Anastasios Gounaris, Christos Yfoulis, Rizos Sakellariou, and Marios D. Dikaiakos. Robust Runtime Optimization of Data Transfer in Queries over Web Services. In *ICDE*, pages 596–605, 2008.

[GZ08] Tingjian Ge and Stanley B. Zdonik. A Skip-list Approach for Efficiently Processing Forecasting Queries. *Proceedings of the VLDB Endowment*, 1(1):984–995, 2008.

[HA03] Stavros Harizopoulos and Anastassia Ailamaki. A Case for Staged Database Systems. In *CIDR*, pages 1–12, 2003.

[Haa07] Laura M. Haas. Beauty and the Beast: The Theory and Practice of Information Integration. In *ICDT*, pages 28–43, 2007.

[HAB+05] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael J. Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise Information Integration: Successes, Challenges and Controversies. In *SIGMOD Conference*, pages 778–787, 2005.

[Hab09] Dirk Habich. *Komplexe Datenanalyseprozesse in serviceorientierten Umgebungen*. PhD thesis, Technische Universität Dresden, 2009.

[HDH07] D. Harish, Pooja N. Darera, and Jayant R. Haritsa. On the Production of Anorexic Plan Diagrams. In *VLDB*, pages 1081–1092, 2007.

[HDH08] D. Harish, Pooja N. Darera, and Jayant R. Haritsa. Identifying Robust Plans through Plan Diagram Reduction. *Proceedings of the VLDB Endowment*, 1(1):1124–1140, 2008.

[HKL+08] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing Query Optimization. *Proceedings of the VLDB Endowment*, 1(1):188–200, 2008.

[hl707] *Health Level Seven Version 3*, 2007. http://www.hl7.org/.

[HML09] Thomas Hornung, Wolfgang May, and Georg Lausen. Process Algebra-Based Query Workflows. In *CAiSE*, pages 440–454, 2009.

[HNM03] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Estimating the Output Cardinality of Partial Preaggregation with a Measure of Clusteredness. In *VLDB*, pages 656–667, 2003.

[HNM+07] Wook-Shin Han, Jack Ng, Volker Markl, Holger Kache, and Mokhtar Kandil. Progressive Optimization in a Shared-Nothing Parallel Database. In *SIGMOD Conference*, pages 809–820, 2007.

[HPL+07] Dirk Habich, Steffen Preißler, Wolfgang Lehner, Sebastian Richly, Uwe Aßmann, Mike Grasselt, and Albert Maier. Data-Grey-BoxWeb Services in Data-Centric Environments. In *ICWS*, pages 976–983, 2007.

[HR07] Marc Holze and Norbert Ritter. Towards Workload Shift Detection and Prediction for Autonomic Databases. In *PIKM*, pages 109–116, 2007.

[HR08] Marc Holze and Norbert Ritter. Autonomic Databases: Detection of Workload Shifts with n-Gram-Models. In *ADBIS*, pages 127–142, 2008.

[HRP+07] Dirk Habich, Sebastian Richly, Steffen Preißler, Mike Grasselt, Wolfgang Lehner, and Albert Maier. BPEL-DT - Data-aware Extension of BPEL to Support Data-Intensive Service Applications. In *WEWST*, pages 1–16, 2007.

[HS02] Arvind Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *VLDB*, pages 167–178, 2002.

[HS03] Arvind Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *VLDB*, pages 766–777, 2003.

[HSA05] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD Conference*, pages 383–394, 2005.

[HSF91] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loops. In *SC*, pages 610–632, 1991.

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[IBM05a] IBM. An Architectural Blueprint for Autonomic Computing. Technical report, IBM, 2005.

[IBM05b] IBM. *Information Integration for BPEL (ii4BPEL) on WebSphere Process Server*, 2005.

[IBM10] IBM. *IBM WebSphere Message Broker*, 2010. `http://www-01.ibm.com/software/integration/wbimessagebroker/`.

[IDC08] IDC. *The Diverse and Exploding Digital Universe - An Updated Forecast of Worldwide Information Growth Through 2011*. IDC, 2008. `http://www.emc.com/digital_universe`.

[IDC10] IDC. *The Digital Universe Decade - Are You Ready?* IDC, 2010. `http://www.emc.com/digital_universe`.

[IDR07] Zachary G. Ives, Amol Deshpande, and Vijayshankar Raman. Adaptive Query Processing: Why, How, When, and What Next? In *VLDB*, pages 1426–1427, 2007.

[IHW04] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to Source Properties in Processing Data Integration Queries. In *SIGMOD Conference*, pages 395–406, 2004.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[IKNG09] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD Conference*, pages 309–320, 2009.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. In *VLDB*, pages 103–114, 1992.

[Ioa93] Yannis E. Ioannidis. Universality of Serial Histograms. In *VLDB*, pages 256–267, 1993.

[JC04] Qingchun Jiang and Sharma Chakravarthy. Scheduling Strategies for Processing Continuous Queries over Streams. In *BNCOD*, pages 16–30, 2004.

[JD08] Thomas Jörg and Stefan Deßloch. Towards Generating ETL Processes for Incremental Loading. In *IDEAS*, pages 101–110, 2008.

[JD09] Thomas Jörg and Stefan Deßloch. Formalizing ETL Jobs for Incremental Loading of Data Warehouses. In *BTW*, pages 327–346, 2009.

[Joh74] David S. Johnson. Fast Algorithms for Bin Packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[Jr.68] Henry Sharp Jr. Cardinality of Finite Topologies. *Journal of Combinatorial Theory*, 5(1):82 – 86, 1968.

[JSHL02] Vanja Josifovski, Peter M. Schwarz, Laura M. Haas, and Eileen Tien Lin. Garlic: A New Flavor of Federated Query Processing for DB2. In *SIGMOD Conference*, pages 524–532, 2002.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *VLDB*, pages 128–137, 1986.

[KC04] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data.* WILEY, 2004.

[KD98] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD Conference*, pages 106–117, 1998.

[KE09] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 7. Auflage.* Oldenbourg, 2009.

[KM05] Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.

[Kos00] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[KS04] Jürgen Krämer and Bernhard Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *SIGMOD Conference*, pages 925–926, 2004.

[KS09] Jürgen Krämer and Bernhard Seeger. Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. *ACM Transactions on Database Systems*, 34(1):1–49, 2009.

[KW99] Arnd Christian König and Gerhard Weikum. Combining Histograms and Parametric Curve Fitting for Feedback-Driven Query Result-size Estimation. In *VLDB*, pages 423–434, 1999.

[KZOC09] Hyeonsook Kim, Ying Zhang, Samia Oussena, and Tony Clark. A Case Study on Model Driven Data Integration for Data Centric Software Development. In *CIKM-DSMM*, pages 1–6, 2009.

[Lar02] Per-Åke Larson. Data Reduction by Partial Preaggregation. In *ICDE*, pages 706–715, 2002.

[Lin99] David S. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 1999.

[Lin06] Greg Linden. Marissa Mayer at Web 2.0, 2006. `http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html`.

[Lit61] John D. C. Little. A Proof for the Queueing Formula: $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[LKPMJP05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *SIGMOD Conference*, pages 419–430, 2005.

[Lov93] L. Lovász. *Combinatorial Problems and Exercises*. North Holland, 1993.

[Low09] Greg Low. Plan Caching in SQL Server 2008. Technical report, MS, 2009. `http://msdn.microsoft.com/en-us/library/ee343986(SQL.100).aspx`.

[LRD06] Linh Thao Ly, Stefanie Rinderle, and Peter Dadam. Semantic Correctness in Adaptive Process Management Systems. In *Business Process Management*, pages 193–208, 2006.

[LSM+07] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. Adaptively Reordering Joins during Query Execution. In *ICDE*, pages 26–35, 2007.

[LTS+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.

[LWV03] Lipyeow Lim, Min Wang, and Jeffrey Scott Vitter. SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads. In *VLDB*, pages 369–380, 2003.

[LX09] Rubao Lee and Zhiwei Xu. Exploiting Stream Request Locality to Improve Query Throughput of a Data Integration System. *IEEE Transactions on Computers*, 58(10):1356–1368, 2009.

[LZ05] Haibo Li and Dechen Zhan. Workflow Timed Critical Path Optimization. *Nature and Science*, 3(2):65–74, 2005.

[LZJ+05] Bin Liu, Yali Zhu, Mariana Jbantova, Bradley Momberger, and Elke A. Rundensteiner. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *VLDB*, pages 1338–1341, 2005.

[LZL07] Rubao Lee, Minghong Zhou, and Huaming Liao. Request Window: an Approach to Improve Throughput of RDBMS-based Data Integration System by Utilizing Data Sharing Across Concurrent Distributed Queries. In *VLDB*, pages 1219–1230, 2007.

[Mak07] Mazeyar E. Makoui. *Anfrageoptimierung in objektrelationalen Datenbanken durch kostenbedingte Termersetzungen.* PhD thesis, Universität Hannover, 2007.

[MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[MDK+01] J. T. Mentzer, W. DeWitt, J. S. Keebler, S. Min, N. W Nix, C. D. Smith, and Z. G. Zacharia. Defining Supply Chain Management. *Journal of Business Logistics*, 22:pages 1–26, 2001.

[MHK+07] Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. Consistent selectivity estimation via maximum entropy. *VLDB Journal*, 16(1):55–76, 2007.

[Mic07] Microsoft. *Asklepios: Business Value Case Study - Effizienz statt Turnschuh-Management für die Klinik-IT.* Microsoft, 2007.

[Mic10] Microsoft. *MS BizTalk Server*, 2010. `http://www.microsoft.com/biztalk/en/us/default.aspx`.

[ML02] Volker Markl and Guy M. Lohman. Learning Table Access Cardinalities with LEO. In *SIGMOD Conference*, page 613, 2002.

[MMK+05] Volker Markl, Nimrod Megiddo, Marcel Kutsch, Tam Minh Tran, Peter J. Haas, and Utkarsh Srivastava. Consistently Estimating the Selectivity of Conjuncts of Predicates. In *VLDB*, pages 373–384, 2005.

[MMLW05] Albert Maier, Bernhard Mitschang, Frank Leymann, and Dan Wolfson. On Combining Business Process Integration and ETL Technologies. In *BTW*, pages 533–546, 2005.

[MN06] Guido Moerkotte and Thomas Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*, pages 930–941, 2006.

[Moe09] Guido Moerkotte. *Building query compilers*. online, 2009. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`.

[Mor68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MRS+04] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust Query Processing through Progressive Optimization. In *SIGMOD Conference*, pages 659–670, 2004.

[MS09] Microsoft and SAGA. *Introduction to Financial Messaging Services Bus*. Microsoft and SAGA, 2009.

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD Conference*, pages 49–60, 2002.

[MTSP05] Jose-Norberto Mazón, Juan Trujillo, Manuel A. Serrano, and Mario Piattini. Applying MDA to the Development of Data Warehouses. In *DOLAP*, pages 57–66, 2005.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[MVW00] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Dynamic Maintenance of Wavelet-Based Histograms. In *VLDB*, pages 101–110, 2000.

[Neu09] Thomas Neumann. Query Simplification: Graceful Degradation for Join-Order Optimization. In *SIGMOD Conference*, pages 403–414, 2009.

[NGT98] Hubert Naacke, Georges Gardarin, and Anthony Tomasic. Leveraging Mediator Cost Models with Heterogeneous Data Sources. In *ICDE*, pages 351–360, 1998.

[NRB09] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. Self-Tuning Query Mesh for Adaptive Multi-Route Query Processing. In *EDBT*, pages 803–814, 2009.

[NWRB09] Rimma V. Nehme, Karen Works, Elke A. Rundensteiner, and Elisa Bertino. Query Mesh: Multi-Route Query Processing Technology. *Proceedings of the VLDB Endowment*, 2(2):1530–1533, 2009.

[OAS06] OASIS. *Web Services Business Process Execution Language Version 2.0*, 2006.

[O'C08] William O'Connell. Extreme Streaming: Business Optimization Driving Algorithmic Challenges. In *SIGMOD Conference*, pages 13–14, 2008.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *VLDB*, pages 314–325, 1990.

[OMG03] OMG. *Unified Modeling Language (UML), Version 2.0*, 2003.

[OMG07] OMG. *XML Metadata Interchange (XMI) 2.1*, 2007.

[Ora10] Oracle. *Oracle SOA Suite*, 2010. `http://www.oracle.com/technetwork/middleware/soasuite/overview/index.html`.

[PGVA08] Prasanna Padmanabhan, Le Gruenwald, Anita Vallur, and Mohammed Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *VLDB Journal*, 17(5):1143–1164, 2008.

[PHL10] Steffen Preißler, Dirk Habich, and Wolfgang Lehner. Process-based Data Streaming in Service-Oriented Environments. In *ICEIS*, pages 40–49, 2010.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, 1987.

[Pol05] Neoklis Polyzotis. Selectivity-Based Partitioning: A Divide-and-Union Paradigm For Effective Query Optimization. In *CIKM*, pages 720–727, 2005.

[PVHL09a] Steffen Preißler, Hannes Voigt, Dirk Habich, and Wolfgang Lehner. Stream-Based Web Service Invocation. In *BTW*, pages 407–417, 2009.

[PVHL09b] Steffen Preißler, Hannes Voigt, Dirk Habich, and Wolfgang Lehner. Streaming Web Services and Standing Processes. In *BTW*, pages 608–611, 2009.

[QRR+08] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-Memory Scan Sharing For Multi-Core CPUs. *Proceedings of the VLDB Endowment*, 1(1):610–621, 2008.

[RDS+04] Elke A. Rundensteiner, Luping Ding, Timothy M. Sutherland, Yali Zhu, Bradford Pielech, and Nishant Mehta. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *VLDB*, pages 1353–1356, 2004.

[Rei08] Peter Reimann. Optimization of BPEL/SQL Flows in Federated Database Systems. Master's thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008.

[RH05] Naveen Reddy and Jayant R. Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. In *VLDB*, pages 1228–1240, 2005.

[ROH99] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *VLDB*, pages 599–610, 1999.

[Ros10] Kenneth A. Ross. Optimizing Read Convoys in Main-Memory Query Processing. In *DaMoN*, pages 1–7, 2010.

[RZL04] Amira Rahal, Qiang Zhu, and Per-Åke Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *VLDB Journal*, 13(2):162–176, 2004.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*, pages 23–34, 1979.

[SAP03] SAP. *Energy Data Management*. SAP, 2003.

[SAP10] SAP. *SAP Process Integration*, 2010. `http://help.sap.com/saphelp_nwpi711/helpdata/en/c0/3930405fa9e801e10000000a155106/frameset.htm`.

[SBC06] Piyush Shivam, Shivnath Babu, and Jeffrey S. Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. In *VLDB*, pages 535–546, 2006.

[SBL04] Sven Schmidt, Henrike Berthold, and Wolfgang Lehner. QStream: Deterministic Querying of Data Streams. In *VLDB*, pages 1365–1368, 2004.

[Sch97] August-Wilhelm Scheer. *Wirtschaftsinformatik: Referenzmodelle für industrielle Geschäftsprozesse*. Springer-Verlag, 1997.

[Sch01] August-Wilhelm Scheer. *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen*. Springer-Verlag, 2001.

[Sch07] Sven Schmidt. *Quality-of-Service-Aware Data Stream Processing*. PhD thesis, Technische Universität Dresden, 2007.

[Sim04] Alkis Simitsis. *Modeling and Optimization of Extraction-Transformation-Loading (ETL) Processes in Data Warehouse Environments*. PhD thesis, National Technical University of Athens, 2004.

[Sim05] Alkis Simitsis. Mapping Conceptual to Logical Models for ETL Processes. In *DOLAP*, pages 67–76, 2005.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[SMWM06] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query Optimization over Web Services. In *VLDB*, pages 355–366, 2006.

[SO82] Peter Scheuermann and Aris M. Ouksel. Multidimensional B-trees for associative searching in database systems. *Information Systems*, 7(2):123–137, 1982.

[SS09a] Sattanathan Subramanian and Guttorm Sindre. An Optimization Rule for ActiveXML Workflows. In *ICWE*, pages 410–418, 2009.

[SS09b] Sattanathan Subramanian and Guttorm Sindre. Improving the Performance of ActiveXML Workflows: The Formal Descriptions. In *IEEE SCC*, pages 308–315, 2009.

[Sto02] Michael Stonebraker. Too Much Middleware. *SIGMOD Record*, 31(1):97–106, 2002.

[SVS05] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. Optimizing ETL Processes in Data Warehouses. In *ICDE*, pages 564–575, 2005.

[SW97] Markus Sinnwell and Gerhard Weikum. A Cost-Model-Based Online Method for Ditributed Caching. In *ICDE*, pages 532–541, 1997.

[SWCD09] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. QoX-Driven ETL Design: Reducing the Cost of ETL Consulting Engagements. In *SIGMOD Conference*, pages 953–960, 2009.

[SWDC10] Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, and Malú Castellanos. Optimizing ETL Workflows for Fault-Tolerance. In *ICDE*, pages 385–396, 2010.

[TD03] Feng Tian and David J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *VLDB*, pages 333–344, 2003.

[TDJ10] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Sharing-Aware Horizontal Partitioning for Exploiting Correlations During Query Processing. *Proceedings of the VLDB Endowment*, 3(1):542–553, 2010.

[TP09] Christian Thomsen and Torben Bach Pedersen. pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers. In *DOLAP*, pages 49–56, 2009.

[TSN+08] Jess Thompson, Daniel Sholler, Yefim V. Natis, Massimo Pezzini, Kimihiko Iijima, Raffaella Favata, and Paolo Malinverno. *Magic Quadrant for Application Infrastructure for Back-End Application Integration Projects*. Gartner, 2008.

[TvdAS09] Nikola Trcka, Wil M. P. van der Aalst, and Natalia Sidorova. Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In *CAiSE*, pages 425–439, 2009.

[TVS07] Vasiliki Tziovara, Panos Vassiliadis, and Alkis Simitsis. Deciding the Physical Implementation of ETL Workflows. In *DOLAP*, pages 49–56, 2007.

[UGA+09] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable Performance for Unpredictable Workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717, 2009.

[vdABtHK00] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced Workflow Patterns. In *CoopIS*, pages 18–29, 2000.

[vdAtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[Vog08] Werner Vogels. Eventually Consistent. *ACM Queue*, 6(6):14–19, 2008.

[VSES08] Marko Vrhovnik, Oliver Suhre, Stephan Ewen, and Holger Schwarz. PGM/F: A Framework for the Optimization of Data Processing in Business Processes. In *ICDE*, pages 1584–1587, 2008.

[VSRM08] Marko Vrhovnik, Holger Schwarz, Sylvia Radeschütz, and Bernhard Mitschang. An Overview of SQL Support in Workflow Products. In *ICDE*, pages 1287–1296, 2008.

[VSS02a] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual Modeling for ETL Processes. In *DOLAP*, pages 14–21, 2002.

[VSS02b] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. On the Logical Modeling of ETL Processes. In *CAiSE*, pages 782–786, 2002.

[VSS+07] Marko Vrhovnik, Holger Schwarz, Oliver Suhre, Bernhard Mitschang, Volker Markl, Albert Maier, and Tobias Kraft. An Approach to Optimize Data Processing in Business Processes. In *VLDB*, pages 615–626, 2007.

[Wei11] Andreas M. Weiner. Advanced Cardinality Estimation in the XML Query Graph Model. In *BTW*, pages 207–226, 2011.

[WfM05] WfMC. *Process Definition Interface - XML Process Definition Language 2.0*, 2005.

[WH09] Andreas M. Weiner and Theo Härder. Using Structural Joins and Holistic Twig Joins for Native XML Query Optimization. In *ADBIS*, pages 149–163, 2009.

[Win03] Marianne Winslett. Interview with Pat Selinger. *SIGMOD Record*, 32(4):93–103, 2003.

[WK10] Richard Winter and Pekka Kostamaa. Large Scale Data Warehousing: Trends and Observations. In *ICDE*, page 1, 2010.

[WPSB07] Qinyi Wu, Calton Pu, Akhil Sahai, and Roger S. Barga. Categorization and Optimization of Synchronization Dependencies in Business Processes. In *ICDE*, pages 306–315, 2007.

[XCY06] Zhijiao Xiao, HuiYou Chang, and Yang Yi. Optimal Allocation of Workflow Resources with Cost Constraint. In *CSCWD*, pages 1190–1195, 2006.

[YB08] Qi Yu and Athman Bouguettaya. Framework for Web Service Query Algebra and Optimization. *ACM Transactions on the Web*, 2(1):1–35, 2008.

[YL95] Weipeng P. Yan and Per-Åke Larson. Eager Aggregation and Lazy Aggregation. In *VLDB*, pages 345–357, 1995.

[ZDS+08] Mohamed Ziauddin, Dinesh Das, Hong Su, Yali Zhu, and Khaled Yagoub. Optimizer Plan Change Management: Improved Stability and Performance in Oracle 11g. *Proceedings of the VLDB Endowment*, 1(2):1346–1355, 2008.

[ZLFL07] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD Conference*, pages 533–544, 2007.

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *SIGMOD Conference*, pages 431–442, 2004.

List of Figures

List of Tables

List of Algorithms

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Matthias Böhm
Dresden, 3. Dezember 2010