Zuerst ersch. in : Search Computing : Trends and Developments / Stefano Ceri ... (eds.). - Berlin [u.a.] : Springer, 2011. - S. 156-166. - (Lecture notes in computer science ; 6585 : State-of-the-Art Survey). - ISBN 978-3-642-19667-6
http://dx.doi.org/10.1007/978-3-642-19668-3_15

Run-Time Adaptivity for Search Computing

Daniele Braga[1], Michael Grossniklaus[1], and Norman W. Paton[2]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano
{braga,grossniklaus}@elet.polimi.it
[2] School of Computer Science, University of Manchester
npaton@manchester.ac.uk

Abstract. In Search Computing, queries act over internet resources, and combine access to standard web services with exact results and to ranked search services. Such resources often provide limited statistical information that can be used to inform static query optimization, and correlations between the values and ranks associated with different resources may only become clear at query runtime. As a result, search computing seems likely to benefit from adaptive query processing, where information obtained during query evaluation is used to change the way in which a query is executing. This chapter provides a perspective on how run-time adaptivity can be achieved in the context of Search Computing.

1 Introduction

In contrast to traditional data sources, such as databases and digital libraries, service-based data sources like web services and search engines are more challenging to characterize in terms of average response time, expected result size, data distribution, and similar features that are typically used to perform query planning and optimization according to consolidated techniques. A traditional optimization approach alone is therefore expected to be less effective in a setting where most data is accessed via service invocations and by means of search engines.

Nevertheless, as these data sources are central to Search Computing, it is crucial to have a query processing paradigm that is (a) sophisticated enough to compute an "off-line optimal" query plan at compile-time, and (b) flexible enough to adapt that plan at run-time, in response to deviations from the assumptions made at compile-time on the expected behaviors.

More specifically, a query execution plan embodies the decisions made at compile-time by the query optimizer that generated it. These decisions broadly consist of

1. the order of evaluation of operators, whenever alternative schedules can be considered equivalent;
2. the choice of alternative algorithms and auxiliary data structures to implement the evaluation of operators (as can be the case for different join strategies and different levels of caching);
3. the level of partitioned parallelism, so as to control if the same operation can be performed in parallel by different computational units on different data fragments; and

4. the allocation of plan fragments to available computational resources, so as to balance the computational load and avoid loss of performance due to bottlenecks.

An adaptive query processor may revise any of the above decisions at query run-time, in the light of feedback received, e.g., on actual rather than predicted selectivities, on the presence of delays and idle computational units, or on the correlation between rankings from different searches.

Proposals for adaptive query processing can be classified as *plan preserving*, where adaptation takes place by tuning the strategy used to execute an essentially stable representation of a query, or *plan changing*, where adaptation changes the query plan, and thus involves stopping the execution, re-optimizing the plan, and resuming execution. Plan changing proposals, in particular, must take account of the (partial) work done at the point when re-optimization takes place.

This chapter will discuss the plan preserving and plan changing opportunities for adaptation in Search Computing. Section 2 outlines the key features of plan changing and plan preserving approaches to adaptation, and discusses how these are affected by ranked data. Section 3 outlines opportunities for plan changing adaptivity in the context of the Panta Rhei execution model, described in the previous chapter. Section 4 concludes the chapter with an outlook on future work.

2 Run-Time Adaptive Query Processing

This section details techniques that have been developed to exploit both *plan preserving* and *plan changing* adaptation, with specific reference to adaptation for rank-aware queries. In this section, an approach is considered to be *plan preserving* if the query optimizer is not invoked at query runtime to identify a new plan, and *plan changing* where the optimizer is invoked at query runtime to generate a plan that may then be used to complete query execution.

2.1 Plan Preserving Adaptation

Plan preserving query adaptation modifies some property of query execution at runtime in response to information about the environment in which a query is executing (e.g. machine loads) or some information about the query (e.g. operator selectivity),

Table 1. Plan preserving adaptations

Proposal	Property Adapted
Eddies [1]	Route followed by tuples between operators
Flux [2]	Distribution policy in partitioned parallelism
DITN [3]	Presence of redundant fragments
Scrambling [4]	Query schedule

while leaving the basic structure of the plan unchanged. Table 1 lists several plan preserving approaches along with the property that is adapted. To take these in turn:

Eddies [1]: Unlike classical query plans, in which the order of operator execution is reflected in the structure of the plan, in this proposal an *eddy* operator is introduced that routes data to the operators that actually evaluate the query. The *eddy* then monitors these operators, for example to ascertain their selectivities, and changes the order in which tuples are routed to operators on the basis of this feedback. This has the effect, for example, of changing the join order at runtime.

Flux [2]: In partitioned parallelism, a single operator may be assigned to multiple computational nodes, and a *distribution policy* indicates what fraction of the data should be directed to each node. In this proposal, the distribution policy may be revised at query runtime, with a view to managing load imbalance, and a protocol is provided for moving operator state to reflect changes in distribution.

DITN [3]: In the Data in the Network (DITN) proposal, it is assumed that queries are being evaluated on non-dedicated (e.g. scavaged) resources, and thus that the resources have unpredictable loads and that there may be surplus resource available. In parallel query scheduling, where there is a delay in completing a query fragment, a redundant copy is run on different nodes, and the first of the copies to produce results is preferred over the other.

Scrambling [4]: In query scrambling, adaptations seek to accommodate delays in sources generating tuples. One of the adaptations changes the order in which query fragments are scheduled, by identifying runnable fragments that are then executed and their results materialized.

Although there is considerable diversity in plan preserving techniques, there are a number of recurring themes; the designers of plan preserving adaptive strategies must:

- Identify the problem to be addressed – this is typically much more specific than *improve query response time*, as the adaptation to be made is generally targeted at a particular problem. For example, the detection of load imbalance may result in a change in workload allocation.
- Identify the monitoring information that is required both to diagnose the problem and to parameterize the adaptation. For the most part, monitoring information is that required to parameterize a query cost model [5], but the monitoring information can be also used to select responses as well as to analyse progress (e.g. for learning the properties of different scheduling algorithms [6]).
- Define the adaptation that is to take place, identifying: (i) any constraints on when the adaptation can safely be applied (e.g. moments of symmetry in eddies [1]); (ii) any additional state that is required to support the adaptation (e.g. State Modules that support efficient query processing by eddies [7]); and

(iii) any changes or additions to query compilation or optimization that are required to support adaptation (e.g. a specialized data distribution policy is used by DITN [3]).
- Integrate the adaptation into the query processor; this may be as an adaptive operator (e.g. [1,2]) or a controller that is notified by the query (e.g. [4]).

Overall, the plan preserving approach has been applied in a wide range of contexts to carry out a diverse collection of plan changes. In principle, several different plan preserving approaches can be in play at the same time, although in practice controlling the interplay between different adaptations may be challenging.

2.2 Plan Changing Adaptation

In defining *plan changing* approaches as those in which the optimizer is called at runtime, we note that some adaptive strategies generate multiple plans before query execution, and then may swap between these at runtime (e.g. [8,9]), and thus might be felt to fall within a gap in our classification. Henceforth in this section, however, we consider proposals in which the adaptation involves stopping the plan, re-optimizing the plan and resuming execution.

Because plan changing proposals may make substantial changes to the way in which a query is being evaluated, they must take account of the work done on the evaluation of a query at the point when re-optimization takes place. As such, the designers of plan changing adaptive strategies must:

- Identify the monitoring information that is required both to establish that it may be useful to consider adapting and to inform the construction of a new plan by the optimizer. For example, adaptation may only take place when changes to statistics have been detected beyond some threshold (e.g. [10]), or where the statistics have moved outside the range for which the current plan was optimal when compiled (e.g. [11]).
- Identify when the plan can safely be stopped to support adaptation, and characterize the work done to date so that the new plan does not unnecessarily repeat work. For example, *coarse grained* techniques may reuse complete results of operator evaluation (e.g. [11]), whereas *fine grained* techniques are able to reuse partial results from operators (e.g. [12,13]). Such fine grained techniques must be able to characterize precisely how the partial result of an operator relates to the data consumed, and thus what work remains to be done [13].
- Modify the optimizer so that it takes account of updated statistical data and the state computed by the current plan in exploring new strategies. In doing this, the previous plan reconsidered with the updated statistics may serve as an upper bound to the cost function to guide the new search for optimality.
- Integrate the adaptation into the query processor; this may be as an adaptive operator (e.g. [11]) or a controller (e.g. [13]).

In designing a strategy following the above steps, it may be considered to be a good thing if as many as possible of the following non functional requirements can be satisfied:

- Queries can be stopped for re-optimization at many points in their evaluation; coarse grained techniques are typically only able to reuse work at materialization points when operators have completed execution, whereas fine grained techniques may support a more agile reuse of cached temporary results.
- Auxiliary or repeated work after re-optimization is minimized; some strategies consider discarding work in order to enable adaptation before operators have completed (e.g. [11]), and some use *stitch-up plans* to complete operator evaluation (e.g. [14]).
- A wide variety of operators can be used; some strategies support fine-grained reoptimization by restricting the collection of operators that can participate in adaptive plans (e.g. [12]). A particularly relevant case is that of rank-aware operators, that typically build on thresholds derived from the data already processed and make assumptions on the data that is still to be processed. Rank-aware operators are addressed in the next section.

Another reason for deciding to change the plan could be explicit or implicit feedback from the user. In all settings in which query sessions are interactive and executions can be manually stopped and resumed directly by the users, or users can evaluate the quality of the results obtained to date, there is potential for reoptimization. This is a relevant scenario for Search Computing, as described in detail in Chapter [15] of the present collection, dedicated to the study of the "Search as a Process" paradigm for Search Computing.

2.3 Adaptive Query Processing for Ranked Data

Where query processing includes ranking, standard query processing operators are supplemented with rank-aware operators [16]. For example, a rank-aware join operator consumes ranked inputs from its operands where each input tuple has a *rank score*, applies a *ranking function* to the rank scores of matching input tuples to compute rank scores for matching tuples, and returns matching tuples in order of their scores.

Adaptivity is potentially important for queries that include ranking; the reasons for adapting in non-ranked queries carry forward to rank-aware queries, where additional challenges result from difficulties in predicting the relationships between ranked inputs. For example, are highly ranked tuples more or less likely to match than randomly selected tuples?

Adaptive rank-aware query optimization has been investigated by Ilyas *et al.* [17], where their plan preserving approach is described in most detail. The focus is on adaptations involving a rank-aware symmetric hash join [16]. In this algorithm, when a tuple is read from an operand, it is: (i) stored in the hash table for its operand indexed by join attributes; (ii) probed against the hash table for the other operand, and the rank score is calculated for each matching tuple, which is inserted into a *rank queue*. A result tuple r can be returned from the rank queue when it is known from the properties of the ranking function and the scores of the tuples read from the ranked inputs that no result tuple with a higher score than that of r can be generated from as-yet-unread input tuples.

In the adaptive query processing strategy, runtime monitoring either detects delays or unexpected correlations between scores. When these are detected, the optimizer is rerun to construct a new plan, and the state associated with each operator in the new plan is either copied from the previous plan or reconstructed from scratch. In essence, state can be copied from a node n in the original plan P to a node n' in the new plan P' when n and n' have the same leaves. This ability to reuse some operator state means that work need not be redone during evaluation, but individual adaptations may still be expensive because operator state that did not exist before adaptation must be computed to generate the state that would have existed if P' had been run from the beginning.

3 Adaptation in Search Computing

Adaptive query processing is relevant to Search Computing

(a) because the actual behavior of data sources can diverge from the expected one in many ways (e.g., unexpected delays, or unpredictable cardinality of the results of exact services), but also
(b) because some computations intrinsically need to react and adapt to run-time evidence, as is the case for the generation of results with top-k guarantees in the presence of unpredictable score information for search services, and also
(c) because of the intervention of the end user in the query execution process, which requires the ability to reuse the results available to date and to reshape them to enable the continuation of the search session.

Panta Rhei is the unit-based language used for the specification of physical query plans in Search Computing. We briefly summarize its features so as to make this chapter self-contained. A physical query plan consists of a directed graph of nodes, corresponding to processing units, and edges, forming the data and control flows. Results are progressively created by joining and pruning the data returned by invoking data sources. The most important processing nodes in Panta Rhei are service invocation units, for extracting ranked data from data sources, and strategy units, that are used to control and synchronize the behavior of invocation units that are in parallel and pipe joins.

Join units are rank-preserving according to either *top-k* or *good-k* join strategies. Joins that are regulated by a top-k strategy produce results according to an ordering imposed by a score aggregation function which is a weighted sum of partial scores. Joins that are regulated by a good-k strategy produce results in an order that is as consistent with the ordering given by the aggregation function as can be formed with the best tuples from each source. The actual order in which tuples are output in the latter case is given by combining the best results from each single source as soon as possible, and the more the partial rankings are correlated with the aggregate one the better the good-k results will approximate top-k results. Also, top-k joins are blocking, because a tuple can be output only when it is guaranteed to have a ranking that is at least as high as any that can possibly be formed with data that has yet to be extracted. Good-k joins,

instead, are non-blocking by construction, and therefore their output tuples are produced as soon as they are available.

In a physical plan, operators are assigned a *budget* that expresses the number of service invocations that they are allowed to "spend". More details of Panta Rhei can be found in the previous chapter [18] of this collection.

We can classify the actions performed in order to achieve run-time adaptivity not only as plan-preserving or plan-changing, but also according to the events that trigger the actions. In particular, we distinguish between *user-generated* and *system-generated* actions.

User-generated actions reflect reactions to unsatisfactory results, further classified as:

- *more all* commands, given when the execution halts and the tuples presented in the results are too few with respect to the user's expectations;
- *more one* commands, given when the results associated with one specific source are unsatisfactory, either because they are too few, or because the user believes that there may be additional relevant results which have not yet been extracted, characterized by a lower local ranking but higher capability of forming interesting combinations when joined with results from other services;
- changes to the *score weights in the goal function*, because the current result does not reflect the user's preference, as it over-emphasizes or de-emphasizes one of the services that instead is considered as most relevant for the user;
- *lowering* of the *level of guarantees* that are expected on the result, because a top-k enabled execution leads to marginal results or excessively delays the output of the first combinations, and a good-k approach seems preferable;
- *raising* of the *level of guarantees* that are expected on the result, when a good join execution produces results that cannot be sufficiently trusted, and top-k join strategies should instead be used.

System-generated actions, instead, can be triggered by events and anomalies that are monitored and detected automatically.

3.1 Plan-Preserving Adaptations in Search Computing

In the following cases, plan adaptation occurs in the context of the same plan, whose execution continues after adaptation by reusing the current results.

A. More One and More all Directives. These commands are typically performed by the user when results are perceived as insufficient, and they call for adding more results, either selectively from one service, or globally. It is important to note that Panta Rhei physical plans are intentionally built with the objective of giving few tuples in the result, because users are normally interested only in the top tuples of ranked results. Therefore, allowing within Panta Rhei the structures for a seamless continuation of the execution of a query plan is an important choice, with consequences on the architecture and on the implementation.

B. Change of Weights in the Score Aggregation Function. The change of weights affects top-k strategies, as it changes the threshold values used to assess that a given combination belongs to the top-k results. A change of weights may be followed by a total change of the result which is displayed to the user, but such a change occurs without recomputing the query: a different result set can be selected from the buffered query results by means of a recomputation of threshold values. Such result set may be too small, and in such case the user can continue query execution by means of more one and more all directives.

C. Budget Redistribution. In Panta Rhei, controller units spend their budget of invocations relative to the units they are responsible for, thus implementing the "forward" control logic as decided by compile-time planning and optimization. Then, at run-time, suspend and resume signals are propagated "backward" in order to re-synchronize execution when anomalies are detected, such as delays in one of the controlled units. Therefore, forward controls determine producer-consumer relationships according to the query plan, and backward controls optionally condition those producer-consumer relationships that deviate too much from the optimal plan determined at query optimization time.

The simplest form of system-controlled plan adaptation is a run-time budget redistribution, that consists in moving budget to those operations that have exhausted their budget when execution halts before producing a result with the estimated number of tuples. Redistribution within the same query does not change the global load associated with query execution, and therefore its relationships to other, concurrent query executions. While backward control occurs in the context of the control signals between the units, and is therefore part of the normal adaptive behavior of plan, budget redistribution is properly classified as an adaptation of the plan that occurs on the initiative of a monitoring system module.

D. Service Replacement without Replanning. If a service is unavailable at execution time, the system may detect this fact and autonomously decide to resort to a different service; if the new service has the same signature (or, in SeCo terms, is associated with the same access pattern) then the service can be replaced without changing the plan. Of course the invocation of the new service can have very different delay, cost and output tuples compared to the original service, but such differences can be dealt with by backward controls and budget redistributions.

If the service becomes unavailable during query execution, results presented to the user prior to adaptation are correct, but incomparable with results presented to user after the adaptation, because the first invocation of the replacing service produces ranked results starting from a rank value that has no relationship with the last invocation of the replaced service.

E. Changing Guarantees During Execution. The last plan-preserving adaptation action that we consider here occurs when the user decides to raise or

lower the level of guarantees of the joins of a running plan, e.g. because the user decides to accept all good results in a situation in which the system produces too few top-k results. This is a borderline case, because the plan topology does not change, but the controller of join operations changes substantially.

When the level of guarantee is raised (from good to top), a total change of the result which is displayed to the user is needed, but such a change occurs without recomputing the query: a subset of tuples in top-k order can be selected from the buffered query results by means of the computation of threshold values. Such a result set may be too small, and in such case the user can continue query execution by means of more one and more all directives.

When the level of guarantee is lowered, the system initially presents to the user all the result tuples which are buffered in the order in which they were extracted and generated. Such a result set, although larger than the previous result set, may still be too small, and also in such case the user can continue query execution by means of more one and more all directives.

3.2 Plan-Changing Adaptation in Search Computing

In this subsection we sketch some preliminary cases of plan-changing adaptivity. Supporting this form of adaptation is much more complex, as the new plan can be substantially different from the old plan; the new plan (or a portion of it) needs to be installed as a replacement of the old plan. Moreover, results presented to the users prior to adaptation may be incomparable with results presented to user after adaptation, because the execution of the new plan may restart tuple production for some services participating in the plan.

A. Service Replacement with Replanning. If a service is unavailable at execution time, the system may have to resort to a different service with a different signature. This normally causes a change in the plan, as the service signatures dictate the type (pipe vs parallel) of join that connects the service execution unit to the other units; such changes, however, may be known in advance and rapidly installed with minimum perturbation of the rest of the plan. Of course the new plan can have very different performance, and thus may benefit from reoptimization. If a service becomes unavailable during query execution, results presented to the users prior to adaptation are correct, but incomparable with results presented to user after the adaptation.

B. Selection of Equivalent Alternative Plans. Case A is a particular case of Case B, which occurs when a complete plan is substituted by another equivalent plan. Such a situation may occur when the actual availability and performance of services suggests a radical strategy change. Technically, alternative plans can be determined at execution time by marking certain services as unavailable and then seeing if an alternative plan exists; then, such plan should be installed and executed.

4 Conclusion

In this paper, we considered query adaptation, first by reviewing its general properties, and classifying adaptation into the broad classes of plan-preserving and plan-changing. Such a classification can be applied to explore opportunities for adaptive query processing in Panta Rhei, the SeCo execution engine. We have shown that relevant cases of adaptations may be due to explicit user interactions: users adapt to the progressive presentation of results by issuing simple commands, which may have a great impact upon the result computation strategies. Most cases of plan-preserving adaptation are rather straightforward and will be supported by the first releases of the query engine, while plan-changing adaptation is only sketched in this paper and will be the subject of future work.

References

1. Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. In: SIGMOD Conference, pp. 261–272 (2000)
2. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In: ICDE, pp. 25–36 (2003)
3. Raman, V., Han, W., Narang, I.: Parallel querying with non-dedicated computers. In: Proc. VLDB, pp. 61–72 (2005)
4. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost Based Query Scrambling for Initial Delays. In: SIGMOD Conference, pp. 130–141 (1998)
5. Gounaris, A., Paton, N., Fernandes, A., Sakellariou, R.: Self-monitoring query execution for adaptive query processing. Data Knowl. Eng. 51(3), 325–348 (2004)
6. Sutherland, T.M., Zhu, Y., Ding, L., Rundensteiner, E.A.: An adaptive multi-objective scheduling selection framework for continuous query processing. In: IDEAS, pp. 445–454 (2005)
7. Raman, V., Deshpande, A., Hellerstein, J.M.: Using State Modules for Adaptive Query Processing. In: Proc. ICDE, pp. 353–364 (2003)
8. Babu, S., Bizarro, P., DeWitt, D.: Proactive Re-Optimization. In: Proc. ACM SIGMOD, pp. 107–118 (2005)
9. Bizarro, P., Babu, S., DeWitt, D.J., Widom, J.: Content-based routing: Different plans for different data. In: VLDB, pp. 757–768 (2005)
10. Kabra, N., DeWitt, D.J.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In: SIGMOD Conference, pp. 106–117 (1998)
11. Markl, V., Raman, V., Simmen, D.E., Lohman, G.M., Pirahesh, H.: Robust Query Processing through Progressive Optimization. In: SIGMOD Conference, pp. 659–670 (2004)
12. Li, Q., Shao, M., Markl, V., Beyer, K., Colby, L., Lohman, G.: Adaptively Reordering Joins during Query Execution. In: Proc. ICDE, pp. 26–35 (2007)
13. Eurviriyanukul, K., Paton, N.W., Fernandes, A.A.A., Lynden, S.J.: Adaptive Join Processing in Pipelined Plans. In: Proc. EDBT, pp. 183–194 (2010)
14. Ives, Z., Halevy, A., Weld, D.: Adapting to Source Properties in Data Integration Queries. In: Proc. SIGMOD, pp. 395–406 (2004)
15. Bozzon, A., Brambilla, M., Ceri, S., Fraternali, P.: Exploring the Web with Search Computing. In: Ceri, S., Brambilla, M. (eds.) Search Computing II. LNCS, vol. 6585, pp. 10–25. Springer, Heidelberg (2011)

16. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting top-k join queries in relational databases. VLDB J. 13(3), 207–221 (2004)
17. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K., Elmongui, H.G., Shah, R., Vitter, J.S.: Adaptive Rank-Aware Query Optimization in Relational Databases. ACM Trans. Database Syst. 31(4), 1257–1304 (2006)
18. Braga, D., Corcoglioniti, F., Grossniklaus, M., Vadacca, S.: Efficient Computation of Search Computing Queries. In: Ceri, S., Brambilla, M. (eds.) Search Computing II. LNCS, vol. 6585, pp. 141–155. Springer, Heidelberg (2011)