Adaptive Query Processing: Why, How, When, What Next?

Amol Deshpande
University of Maryland

Zachary Ives
University of Pennsylvania

Vijayshankar Raman
IBM Almaden

ABSTRACT

Adaptive query processing has been the subject of a great deal of recent work, particularly in emerging data management environments such as data integration and data streams. We provide an overview of the work in this area, identifying its common themes, laying out the space of query plans, and discussing open research problems. We discuss *why* adaptive query processing is needed, *how* it is being implemented, *where* it is most appropriately used, and finally, *what next*, i.e., open research problems.

1. INTRODUCTION

In recent years, there has been increasing use of *adaptive query processing* (AQP) as a solution to the problems of query optimization and execution — across relational, text, or XML data, regardless of whether the data is accessed locally, from the Web, or in a continuous stream. Much of this is motivated by the emergence of domains where Selinger-style query optimization fails, either due to insufficient statistics or dynamic data. The result has been a flurry of intriguing new algorithms and systems (e.g., NiagaraCQ, TelegraphCQ, STREAM, Tukwila, YFilter, CAPE, etc). Vendors like IBM, Microsoft, and Oracle are also investigating and deploying adaptivity features for their database products.

In this tutorial we attempt to articulate a "big picture" covering existing techniques, and to understand how techniques generalize beyond their initial implementations. Taking into account time and audience interest, we cover a variety of the major methods developed in the literature, with a focus on techniques used in conventional (non-streaming) queries. Two particular dimensions we consider are the *plan space* explored, and the way execution and optimization are *interleaved* via a measure/analyze/plan/actuate loop. We also discuss the benefits and drawbacks of the various techniques and identify open research problems. Our goal is to simplify and abstract where possible. A more comprehensive survey, which includes a discussion of stream query processing and a full list of references for the work discussed in this tutorial, can be found in our recent survey paper [1].

VLDB '07, September 23-28, 2007, Vienna, Austria.

2. MOTIVATIONS FOR AQP

Declarative queries are a central value proposition of the relational model, letting the users specify only what results they want without having to worry about the strategy (plan) used to access and combine the data. Finding the best plan (query optimization) was addressed in even the first RDBMSs — most successfully by Selinger et al.'s dynamic programming algorithm in System R. System R divided query processing into separate optimization and execution stages and used cost-based enumeration of possible query plans. Over time, this optimization approach has been improved (exploring more exhaustive plans, using histograms, adding cross-block query rewrites), but the basic System R architecture lives on in most query processors. Unfortunately, as database-style query processing has extended to new domains, e.g., data streams, wide-area data sources, and interactive query environments, that approach has run into limitations. Similar problems have been encountered in settings with correlated predicates, query parameters, and self-tuning installations.

Several techniques have been proposed to extend the query optimization process to solve some of these problems: (1) incorporating feedback from previous query executions for better selectivity/cardinality estimation; (2) *parametric* techniques to systematically postpone making certain decisions as late as possible; and (3) *least expected cost* and *robust* optimization techniques that take into account probability distributions that may be associated with certain selectivity estimates. These techniques require minimal changes to the query processor itself; however, they are based on maintaining elaborate statistics on the data, and hence are limited in scope compared to the more ambitious adaptive techniques we discuss in this tutorial.

3. CLASSES OF ADAPTATION

We divide our study of adaptive techniques into two cases.

3.1 Adaptive Selection Ordering

We begin the tutorial with a restricted query processing problem, namely *selection ordering*. Selection ordering determines an order in which to apply a given set of commutative filters (selections) to all the tuples of a relation, so as to find the tuples that *satisfy all the filters*. This is a central problem in query optimization, and it has received renewed attention in the context of environments such as the Web, continuous high-speed data streams, and sensor networks. The problem of selection ordering is now well-understood, with several analytical and theoretical results, in large part due to the "stateless" nature of selection ordering queries.

We first present an adaptive technique called *A-Greedy*

that, at runtime, continuously monitors the properties of the tuples seen recently, and changes (adapts) the query plan in response to changes in these properties. We next consider adapting using the *eddy* operator, which enables fine-grained adaptivity by treating query execution as a process of *routing* tuples through *operators*, and adapts by changing the order in which tuples are routed through the operators (thereby, in effect, changing the query plan being used for the tuple). We briefly discuss extensions of the selection ordering problem to parallel and distributed scenarios.

3.2 Adaptive Join Processing

The design and analysis of adaptive techniques for join queries is more complicated than selection ordering: The space of execution plans is much larger and more complex, due to the large number of possible join orders and algorithms, but more importantly, join operators typically maintain internal state (e.g., hash tables) that depends on the tuples processed by the operator. Due to this internal state, the execution environment itself plays a much larger role in determining the characteristics of query execution: in particular, the type and rate of access to the data can drastically change the nature and trade-offs of query execution. To tackle this complexity, the research community has developed a diverse set of techniques designed for specific execution environments and/or specific join operators. These adaptation techniques apply in different underlying plan spaces, though this space is rarely made explicit. We present these techniques in three parts, roughly based on the space of the query execution plans they explore.

3.2.1 History-Independent Pipelined Execution

We first consider pipelines of non-blocking join and selection operators, with one further restriction: the state built in the operators during execution is largely *independent* of the adaptation choices made by the query processor. This space includes a fairly large class of traditional pipelined query plans, as well as many data stream query processors. This simplifies analysis and design of algorithms, but the algorithms tend to suffer from suboptimal performance because they do not store and reuse any intermediate tuples during query execution. The simplest case is pipelined plans with a single *driver*, and we show how the adaptive selection ordering techniques can be used directly to adapt these queries. We then consider query execution where multiple drivers are permitted, and discuss adaptation using MJoins and unary operators called SteMs (used in conjunction with eddies). We finally discuss a technique called *A-Caching* that uses intermediate result caches to alleviate one of the performance concerns with history-independent execution.

3.2.2 History-Dependent Pipelined Execution

This class of techniques also exclusively uses pipelined operators, but the operators may internalize state that *depends on* the routing choices made by the query processor. This internalized state (the "burden of routing history") makes it hard to change from one query plan to another: an adaptive algorithm must reason about the built-up state and ensure that in switching plans, no output tuples will be lost and no false duplicates will be output. We describe *corrective query processing*, which uses a conventional (binary) query plan at any time, but may use multiple different plans over the entire execution. We then revisit the eddies architecture, and consider adaptation when binary join operators are used with eddies. We next present the STAIR operator, which attempts to lift the burden of routing history by allowing explicit state manipulation. Finally, we discuss how how state manipulation is used in the CAPE system to affect plan changes.

3.2.3 Non-pipelined Execution

Our final segment on adaptive techniques covers plans with blocking operators like SORT — the dominant style of plan considered by most DBMSs today. Blocking operators provide *materialization points* (where an intermediate result is created in entirety before proceeding with further operators of the plan), at which it is easy to re-evaluate query execution decisions and change "downstream" portions of the plan. Techniques that adapt at materialization points are easy to retrofit into existing DBMS engines and some have been prototyped in commercial systems. *Plan staging* simply interleaves optimization and execution: first it optimizes and runs one plan stage to completion, and then, using the first result as input, it optimizes and runs the next stage, etc. The optimization of each stage can use statistics (cardinality, histograms) computed on the outputs of the previous stages. *Mid-query re-optimization*, *progressive optimization*, and *proactive re-optimization* instead initially optimize the entire plan; they monitor the intermediate result sizes during query execution, and re-optimize only if results diverge from the original estimates. *Query scrambling* reacts to address delays in processing data from wide area sources, by rescheduling the order of evaluation of query plan operators and occasionally introducing new operators into the query plan.

4. OPEN PROBLEMS

We conclude with the major tradeoffs in adaptive query processing and present some important open problems in this area:

Parametric query optimization: While this approach has been on the back-burner in recent years, it promises to precompute sets of alternative plans that might avoid the need for adaptive query processing in many cases.

Parallelism: Another open area is extending adaptive techniques to shared-nothing or distributed scenarios: adapting a query plan in a distributed environment can be expensive.

Routing/adaptation policies: Work on eddies and other adaptive techniques often focuses on performance of the tuple routing or plan modification mechanisms, rather than the actual routing or adaptation policies — which could potentially leverage techniques from online learning and related research areas.

Larger-than-memory execution: While adaptive techniques seldom consider paging to disk, this is key to their broader adoption and scale-up.

5. REFERENCES

[1] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007. To appear.