Adaptive Query Processing
Adaptive Query Processing

Amol Deshpande
University of Maryland
USA
amol@cs.umd.edu

Zachary Ives
University of Pennsylvania
USA
zives@cis.upenn.edu

Vijayshankar Raman
IBM Almaden
USA
ravijay@us.ibm.com

Full text available at: http://dx.doi.org/10.1561/1900000001
Foundations and Trends® in Databases
Volume 1 Issue 1, 2007
Editorial Board

Editor-in-Chief:
Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
Berkeley, CA
USA
hellerstein@cs.berkeley.edu

Editors
Surajit Chaudhuri (Microsoft Research)
Ronald Fagin (IBM Research)
Minos Garofalakis (Intel Research)
Johannes Gehrke (Cornell University)
Alon Halevy (Google)
Jeffrey Naughton (University of Wisconsin)
Jignesh Patel (University of Michigan)
Raghu Ramakrishnan (Yahoo! Research)
Editorial Scope

Foundations and Trends® in Databases covers a breadth of topics relating to the management of large volumes of data. The journal targets the full scope of issues in data management, from theoretical foundations, to languages and modeling, to algorithms, system architecture, and applications. The list of topics below illustrates some of the intended coverage, though it is by no means exhaustive:

- Data Models and Query Languages
- Query Processing and Optimization
- Storage, Access Methods, and Indexing
- Transaction Management, Concurrency Control and Recovery
- Deductive Databases
- Parallel and Distributed Database Systems
- Database Design and Tuning
- Metadata Management
- Object Management
- Trigger Processing and Active Databases
- Data Mining and OLAP
- Approximate and Interactive Query Processing
- Data Warehousing
- Adaptive Query Processing
- Data Stream Management
- Search and Query Integration
- XML and Semi-Structured Data
- Web Services and Middleware
- Data Integration and Exchange
- Private and Secure Data Management
- Peer-to-Peer, Sensornet and Mobile Data Management
- Scientific and Spatial Data Management
- Data Brokering and Publish/Subscribe
- Data Cleaning and Information Extraction
- Probabilistic Data Management

Information for Librarians
Foundations and Trends® in Databases, 2007, Volume 1, 4 issues. ISSN paper version 1931-7883. ISSN online version 1931-7891. Also available as a combined paper and online subscription.
Adaptive Query Processing

Amol Deshpande¹, Zachary Ives² and Vijayshankar Raman³

¹ University of Maryland, USA, amol@cs.umd.edu
² University of Pennsylvania, USA, zives@cis.upenn.edu
³ IBM Almaden, USA, ravijay@us.ibm.com

Abstract

As the data management field has diversified to consider settings in which queries are increasingly complex, statistics are less available, or data is stored remotely, there has been an acknowledgment that the traditional optimize-then-execute paradigm is insufficient. This has led to a plethora of new techniques, generally placed under the common banner of adaptive query processing, that focus on using runtime feedback to modify query processing in a way that provides better response time or more efficient CPU utilization.

In this survey paper, we identify many of the common issues, themes, and approaches that pervade this work, and the settings in which each piece of work is most appropriate. Our goal with this paper is to be a “value-add” over the existing papers on the material, providing not only a brief overview of each technique, but also a basic framework for understanding the field of adaptive query processing in general. We focus primarily on intra-query adaptivity of long-running, but not full-fledged streaming, queries. We conclude with a discussion of open research problems that are of high importance.
## Contents

1 Introduction 1

1.1 Query Processing in Relational Database Systems 3
1.2 Motivations for AQP 5
1.3 Road Map 8
1.4 Related Work 8

2 Background: Conventional Optimization Techniques 9

2.1 Query Optimization 9
2.2 Choosing an Effective Plan 20
2.3 Summary 24

3 Foundations of Adaptive Query Processing 27

3.1 New Operators 28
3.2 Adaptivity Loop 35
3.3 Post-mortem Analysis of Adaptive Techniques 40
3.4 Adaptivity Loop and Post-mortem in Some Example Systems 42
3.5 Scope of the Remainder of the Survey 45

4 Adaptive Selection Ordering 47

4.1 Adaptive Greedy 48
4.2 Adaptation using Eddies 52
4.3 Parallel and Distributed Scenarios 57
4.4 Summary 58
One of the fundamental breakthroughs of Codd’s relational data model [33] was the identification of how to take a declarative, logic-based formulation of a query and convert it into an algebraic query evaluation tree. As described in every database textbook, this enabled physical data independence and promised many benefits: the database administrator and the DBMS optimizer became free to choose among many different storage formats and execution plans to answer a declarative query. The challenge, since then, has been how to deliver on these promises — regardless of where or how the data is laid out, how complex the query is, and how unpredictable the operating environment is.

This challenge has spurred 30 years of query processing research. Cost-based query optimization, pioneered by Selinger et al. [102] in System R and refined by generations of database researchers and developers, has been tremendously effective in addressing the needs of relational DBMS query processing: one can get excellent performance for queries over data with few correlations, executed in a relatively stable environment, given sufficient statistical information.

However, when even one of these characteristics is not present, the System R-style optimize-then-execute model begins to break down: as
noted in [69], optimizer error begins to build up at a rate exponential in the size of the query. As the database field has broadened to consider more general data management, including querying autonomous remote data sources, supporting continuous queries over data streams, encoding and retrieving XML data, supporting OLAP and data mining operations, and combining text search with structured query capabilities, the weaknesses of the traditional optimization model have begun to show themselves.

In response, there has been a surge of interest in a broad array of techniques termed adaptive query processing (AQP). AQP addresses the problems of missing statistics, unexpected correlations, unpredictable costs, and dynamic data by using feedback to tune execution. It is one of the cornerstones of so-called autonomic database management systems, although it also generalizes to many other contexts, particularly at the intersection of database query processing and the Web.

The spectrum of adaptive query processing techniques has been quite broad: they may span multiple query executions or adapt within the execution of a single query; they may affect the query plan being executed or the scheduling of operations within the plan; they have been developed for improving performance of local DBMS queries (e.g., [75, 87, 112]), for processing distributed and streaming data (e.g., [6, 72, 88, 92, 101]), and for performing distributed query execution (e.g., [115]).

This survey is an attempt to cover the fundamental issues, techniques, costs, and benefits of adaptive query processing. We begin with a broad overview of the field, identifying the dimensions of adaptive techniques. Then we focus our analysis on the spectrum of approaches available to adapt query execution at runtime — primarily in a non-streaming context. Where possible, we focus on simplifying and abstracting the key concepts of each technique, rather than reproducing the full details available in the papers; we consider generalizations of the specific published implementations. Our goal is to identify the strengths and limitations of the different techniques, demonstrate when they are most useful, and suggest possible avenues of future research.

In the rest of the section, we present a brief overview of query processing in relational database systems (Section 1.1) and elaborate on
the reasons behind the push toward adaptivity (Section 1.2); we then present a road map for the rest of the survey (Section 1.3), and briefly discuss the related surveys of interest (Section 1.4).

1.1 Query Processing in Relational Database Systems

The conventional method of processing a query in a relational DBMS is to parse the SQL statement and produce a relational calculus-like logical representation of the query, and then to invoke the query optimizer, which generates a query plan. The query plan is fed into an execution engine that directly executes it, typically with little or no runtime decision-making (Figure 1.1).

The query plan can be thought of as a tree of unary and binary relational algebra operators, where each operator is annotated with specific details about the algorithm to use (e.g., nested loops join versus hash join) and how to allocate resources (e.g., memory). In many cases the query plan also includes low-level “physical” operations like sorting, network shipping, etc. that do not affect the logical representation of the data.

Certain query processors consider only restricted types of queries, rather than full-blown SQL. A common example of this is select-project-join or SPJ queries: an SPJ query essentially represents a single SQL SELECT-FROM-WHERE block with no aggregation or subqueries.

Fig. 1.1 Query processing in database systems.
An even further restriction is conjunctive queries, which are SPJ queries that only have conjunctive predicates in the WHERE clause; these can be represented as single rules in the Datalog language.

The model of query processing established with the System R project \[102\], which is still followed today, is to divide query processing into three major stages.

Statistics generation is done offline (typically using the RUNSTATS or UPDATE STATISTICS command) on the tables in the database. The system profiles the relation instances, collecting information about cardinalities and numbers of unique attribute values, and often generating histograms over certain fields.

The second stage, which is normally done at runtime\[4\] is query optimization. The optimization stage is very similar to traditional compilation; in fact, in some systems, it generates directly executable code. Optimization uses a combination of cost estimation, where the running times of query subexpressions are estimated (based on known performance characteristics of algorithm implementations, calibration parameters for hardware speed, and the statistics generated for the relations), pruning heuristics (which are necessary to reduce the overall search space), and exhaustive enumeration. For relatively simple queries with good statistics, the plans produced by a query optimizer can be quite good, although as discussed previously, this is less true in more complex settings.

The final stage, query execution, is handled by an engine analogous to a virtual machine or interpreter for the compiled query plan. There are several important aspects of query execution that are of note. The first is that in general it is desirable to pipeline computation, such that each operator processes a tuple at a time from its sub-operators, and also propagates a single tuple to its parent for processing. This leads to better response time in terms of initial answers, and often higher throughput as delays are masked. However, not all operators are naturally amenable to pipelining (e.g., operators like sorting and grouping often must process entire table before they can determine

\[^4\] Except for certain embedded SQL queries, which may be pre-optimized or optimized once for multiple possible input bindings.
what tuple to output next). Also, complex query plans may require too many resources to be fully pipelined. In these settings, the optimizer must break the plan into multiple segments, materializing (storing) intermediate results at the end of each stage and using that as an input to the next stage.

Second, the issue of scheduling computation in a query plan has many performance implications. Traditional query processing makes the assumption that an individual operator implementation (e.g., a nested loops join) should be able to control how CPU cycles are allocated to its child operators. This is achieved through a so-called iterator architecture: each operator has open, close, and getNextTuple methods. The query engine first invokes the query plan root node’s open method, which in turn opens its children, and the process repeats recursively down the plan. Then getNextTuple is called on the root node. Depending on the operator implementation, it will make calls to its children’s getNextTuple methods until it can return a tuple to its parent. The process completes until no more tuples are available, and then the engine closes the query plan.

An alternate approach, so called data-driven or dataflow scheduling, is used in many parallel database systems. Here, in order to allow for concurrent computation across many machines, the data producers — not the consumers — control the scheduling. Each operator takes data from an input queue, processes it, and sends it to an output queue. Scheduling is determined by the rates at which the queues are filled and emptied. In this survey, we will discuss a number of adaptive techniques that in essence use a hybrid of the iterator and data-driven approaches.

1.2 Motivations for AQP

Over the years, many refinements have been made to the basic query processing technology discussed above. Since CPUs are more powerful today and query workloads are much more diverse, query optimizers perform a more comprehensive search of the space of query plans with joins, relying less on pruning heuristics. Selectivity estimation techniques have become more accurate and consider skewed distributions
(and to a limited extent, attribute correlations). However, the System R-style approach has begun to run into its fundamental limits in recent years, primarily due to the emergence of new application domains in which database query processing is being applied. In particular, triggers for this breakdown include the following:

- **Unreliable cardinality estimates**: The cost estimation process depends critically on estimates of the cardinality of various query subexpressions. Despite significant work on building better statistics structures and data collection schemes, many real-world settings have either inaccurate or missing statistics. (In some circumstances, as with remote data sources, statistics may be difficult or impossible to obtain.) Even when base-table statistics are perfect, correlations between predicates can cause intermediate result cardinality estimates to be off by several orders of magnitude [69, 112].

- **Queries with parameter markers**: SQL is not a pleasant language for end users, so most database queries are issued by a user clicking on a form. The SQL for such queries invariably contains parameter markers (for form input), and the pre-computed query plans for such queries can be substantially worse than optimal for some values of the parameters.

- **Dynamically changing data, runtime, and workload characteristics**: In many environments, especially data streams [24 58 92], queries might be long-running, and the data characteristics and hence the optimal query plans might change during the execution of the query. The runtime costs can also change dramatically, especially in wide-area environments. Similarly, fluctuating query workloads can result in variations in the resources available to execute a query (e.g., memory), making it necessary to adapt.

- **Complex queries involving many tables**: Query optimizers typically switch to a heuristic approach when queries become too complex to be optimized using the dynamic programming approach. Such queries are naturally more prone...
to estimation errors [69], and the use of heuristics exacerbates the problem.

- **Interactive querying:** The optimize-then-execute model does not mesh well with an interactive environment where a user might want to cancel or refine a query after a few seconds of execution: the metric changes too quickly for optimization to pay off [61]. Also, pipelined execution and early-result scheduling, even in the presence of slow data sources, becomes paramount.

- **Need for aggressive sharing:** Though there has been much work in multi-query optimization, so far no definitive solutions have emerged in this area. Traditional databases make do with almost no inter-query state sharing because their usage pattern is made up of a small number of queries against large databases. However, sharing the data as well as the computation is critical in environments such as data streams, which feature a very large number of (typically simple) queries over a small set of data streams [28, 86].

There have been two responses to the challenges posed above. The first, a very pragmatic response by application vendors, has been to build domain-specific optimization capabilities outside the DBMS and override its local optimizer. Many commercial DBMSs allow users to specify “hints” on what access methods and join orders to use, via SQL or catalog tables. Recently, SAP has built an application-level query processor that runs only a very limited set of plans (essentially, only table scans), but at very high efficiency [82]. While this achieves SAP’s target of satisfying its users, it runs counter to the database community’s goals of developing high-performance, general-purpose processors for declarative queries.

Our interest in this survey is on the second development, which has been the focus of the academic and commercial DBMS research community: the design and construction of what have come to be known as adaptive (or autonomic) query processing systems, that use runtime feedback to adapt query processing.
1.3 Road Map

We begin with a brief introduction to query optimization in relational database systems (Section 2). We then discuss some of the foundations of AQP, namely, three new operators, and several unifying concepts that we use throughout the survey to illustrate the AQP techniques, to analyze them, and to differentiate between them (Section 3).

We begin our discussion of adaptive query processing by considering a simple class of queries called selection ordering queries (Section 4). The discussion of adaptation techniques for join queries is divided into three parts, roughly based on the space of the query execution plans they explore. We begin with a discussion of techniques for adapting pipelined query execution (Sections 6 and 7), and cover non-pipelined query execution in Section 8. We conclude the survey with a discussion of some of the most important research challenges in adaptive query processing (Section 9).

1.4 Related Work

A number of surveys on query processing are related to this paper. We assume basic familiarity with many of the ideas of Graefe’s survey on query execution techniques [53]. Kossmann’s survey on distributed query processing [79] also provides useful context for the discussion, as do Ioannidis and Chaudhuri’s surveys on query optimization [24, 68]. Babu and Bizarro [8] also present a survey of AQP from a different means of classification from our own (whether the scheme is plan-based, routing-based, or continuous query-based).
References


References


References


Full text available at: http://dx.doi.org/10.1561/1900000001


References


Full text available at: http://dx.doi.org/10.1561/1900000001
References


References


Full text available at: http://dx.doi.org/10.1561/1900000001
References


References

145


146 References


