8 Query Processing

One might assume that query processing in a data integration system differs little from query processing in a traditional DBMS. After all, the query language (whether SQL, datalog, or XQuery) is based on standard relational (or extended relational) operations. Its goal remains to find an *efficient* executable plan for the query. While data integration queries often process distributed data, even this problem has been studied in the context of distributed and federated database systems. Despite these cursory similarities, data integration actually offers a number of challenges that require novel solutions.

Example 8.1

To ground this discussion, let us consider, as the central example in this chapter, a query to look for well-reviewed movies with one of the more prolific actors of our time, Morgan Freeman.

```
SELECT title, startTime
FROM Movie M, Plays P, Reviews R
WHERE M.title = P.movie AND M.title = R.movie AND
   P.location="New York" AND M.actor="Morgan Freeman"
   AND R.avgStars >= 3
```

As we execute this query in a data integration context, we may run into a number of challenges that would not emerge in a traditional DBMS setting.

SOURCE QUERY CAPABILITIES AND LIMITED DATA STATISTICS

DBMS performance is enabled by accurate cost estimation, which ultimately means that the DBMS cost estimator needs to have a good model of I/O and processing rates. Even in a distributed or federated architecture, we assume that the query optimizer can obtain good information from the individual sites. However, a data source in the integration scenario might not be a DBMS: it can range from text or HTML sources, to spreadsheets or XML, to a variety of repository formats, to a relational, hierarchical, or XML DBMS. Some of these sources might not have capabilities for keeping statistics on the data or estimating their own performance. As a result, query optimization in data integration may need to be done with no information. Moreover, some data sources might not be able to execute arbitrary full relational algebra expressions, meaning that the (global) query optimizer cannot naively assign portions of the query to each source. For our example, perhaps the `Movies` source provides direct access to an XML file (with no query processing capabilities)

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00008-9

and the `Plays` source provides a forms-based interface that can perform certain selection operations.

INPUT BINDING RESTRICTIONS
Some data sources may need to be given input data in order to return results, as described in Section 3.3. For example, our `Plays` data source may not return all movie information, but only information about movies playing based on movie and director. The `Reviews` site may need the name of a movie to return its reviews.

VARIABLE NETWORK CONNECTIVITY AND SOURCE PERFORMANCE
Especially over Web sources, network bandwidth and round-trip times may be highly variable. This makes it difficult to predict the access cost for each source and may even require failure handling capabilities. Moreover, the data source may itself be a query processor that is shared with other applications and requests, and its performance in returning results might vary even from one data-fetch request to the next.

NEED FOR FAST INITIAL ANSWERS
Finally, many information integration applications have an interactive "flavor," and the goal is to optimize the creation of the first screenful of answers — as opposed to *all* of the answers, as a traditional DBMS emphasizes.

We discuss how each of these requirements impacts the design of a data integration query processor in this chapter. We first briefly review the basics of conventional DBMS query processing in Section 8.1, and then examine how such techniques have been extended to distributed databases in Section 8.2. A reader familiar with relational query engines may be able to skim at least the first of these two sections.

Once the stage has been set, we present the basic architecture of a data integration query processor in Section 8.3, which incorporates query optimization and execution in an *adaptive loop*. This loop consists of several stages. Section 8.4 describes how the initial query plan gets created. Then Section 8.5 presents the query execution runtime system, which is significantly more complex than a conventional engine. The runtime system participates in *adaptive query processing*, described in Section 8.6, which we divide into two main classes: *event-driven adaptivity* (Section 8.7) and *performance-driven adaptivity* (Section 8.8).

8.1 Background: DBMS Query Processing

To understand query processing in data integration, it is useful first to review the architecture and the basic techniques used in a standard DBMS.

An architectural diagram of a basic DBMS query processor is shown in Figure 8.1. A query in SQL is first **parsed** and converted into an *abstract syntax tree* or a conceptual representation. Then, if the query is posed over (virtual) views whose definition is stored in the system catalog it gets **unfolded** and possibly simplified, forming a *logical*

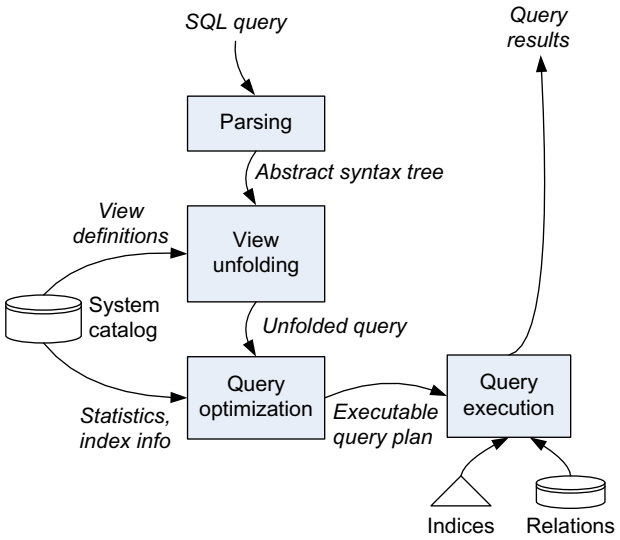

FIGURE 8.1 Modules in a conventional query processor.

query plan. Next, the unfolded query is sent to the **query optimizer**, which consults *statistics* and *index information* from the system catalog in order to choose an *executable query plan* that is predicted to perform well. This plan is fed to the **query execution engine**, which accesses indices and the relation contents, performs a series of operations over this data, and returns query results.

8.1.1 Choosing a Query Execution Plan

Traditional query optimization in a DBMS is based on several fundamental assumptions. First, costs are predictable: we can estimate the number of disk I/Os required to load a table or index, and the access costs are fixed; we can also estimate CPU performance adequately. Second, once data are loaded off disk, pipelined query processing is CPU-bound: the internal implementations of query operators are "tight loops" that have been heavily tuned, and the assumption is that the CPU will be fully occupied in executing them.

Consider how to optimize our movie example from earlier in this chapter. The first step will be to take the query and *unfold* any views — if `Movie`, `Plays`, or `Reviews` were views rather than base relations, their definitions would be expanded. A *query unfolding* stage would create a single logical expression representing the query over the outputs of the views. Then, in some systems, a *query rewrite* stage might attempt to unnest or decorrelate the query expression, such that there is a single select-project-join (SPJ) expression over a series of base tables, rather than an SPJ expression over the output of another SPJ expression.

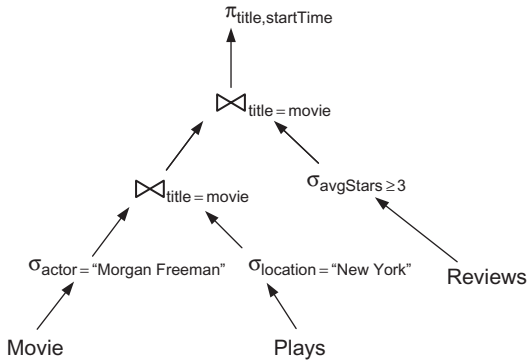

FIGURE 8.2 Example logical query plan for Example 8.1.

In our example, there are no view definitions and hence the logical expression is a single SPJ expression, namely, the simple plan shown in Figure 8.2. This SPJ expression is run through a *cost-based optimizer* that seeks to find the most efficient combination of *physical operators* to produce the expression's output. Cost-based optimization can be divided into two main problems.

Enumeration ("Search")

Plan enumeration iterates over physical query plans equivalent to the original logical plan and estimates their cost. Virtually all optimizers use a combination of *pruning heuristics* to reduce the search space over certain plans that are unlikely to yield good results, plus *exhaustive enumeration* for the remainder of the query plan space.

Modern database optimizers still have options to employ the pruning heuristics used in the original (System R) cost-based query optimizer: searching *left-linear* plans (plans in which every join can only have join expressions on its left input) and only considering cross-products when all join predicates have been evaluated. Within the pruned search space, these optimizers will enumerate all potential join orderings, as well as the different indices available and the different join algorithms.

Optimizers build upon one of the major breakthroughs in the 1970s: the observation that, under certain conditions, query execution costs follow the *principle of optimality*. In general, the optimal join query plan for, say, a 3-way join must make use of the optimal join query plan for a 2-way join. This led to the use of *dynamic programming* when enumerating n-way join plans:

- First (base case), the various access methods to each of the base tables are considered, with selection conditions and projections evaluated as early as possible.
- Next (recursive case), starting with $i = 2$ and increasing to n, possible joins of i relations are considered, by combining optimal $(i-1)$-way join expressions with one additional relation. For each such combination, the most efficient i-way physical plan

is recorded in a dynamic programming table. Again, selection and projection operations, if applicable to the *i*-way join, are applied.
- Once all joins have been computed, similar enumeration is done over cross-products.
- Finally, any remaining grouping and aggregation operations (GROUP BY) and post-aggregation selection conditions (HAVING) are applied.

A variant of the above strategy is to use *top-down* exploration (recursion with memoization) of the same search space. Here, we start with a complete query expression, consider each potential way of decomposing it into substeps (e.g., by finding the different subexpressions that can be joined), and for each such case, we make a recursive call to enumerate each of the subexpressions. Each call will memoize the plan determined to have the lowest cost, such that repeated calls with the same subexpression simply return the memoized results. Recall from your study of algorithms that recursion with memoization and dynamic programming are equivalent. The advantage of the top-down approach is that it enables more possibilities in pruning.

INTERESTING ORDERS

The above strategy ignores the fact that certain operations — sorts (and, as we will discuss later, data shipment across a network) — can be costly but have benefits that are amortized across multiple subsequent operations.

For instance, suppose that `Plays` and `Reviews` returned data sorted by movie title. Then if we were to sort the `Movie` relation by title, we could use the merge join algorithm to perform both joins within the query. Perhaps the sort of the `Movie` relation costs too much to break even with the benefits of doing a merge join with `Plays`. But once we consider the benefits of doing a *second* merge join (with `Reviews`), it is possible that the sort is a good idea after all!

Unfortunately, our dynamic programming strategy as outlined above will not consider this scenario: it will find that the optimal plan for joining `Movie` with `Plays` is *not* to sort and then do a merge join; hence it will discard this plan alternative for the 2-way join. When the 3-way `Movie-Plays-Reviews` join is considered, it will only make use of the 2-way joins in the dynamic programming table.

What is the solution? In essence, we must expand the dynamic programming (or memoization) table to consider sort orders as special cases. The query optimizer first goes over the query to find all potentially useful sort orderings (*interesting orderings*) that might be exploited by join algorithms, the ORDER BY clause, and the GROUP BY clause. In our example, the interesting ordering is on the movie title (called `title` for `Movie` and `movie` for `Plays` and `Reviews`). At each step in plan enumeration, the optimizer will record an entry for the optimal subexpression *for each ordering* (adding a sort operation if necessary to create this ordering) and *independent of ordering*. At the end of the day, the optimizer chooses the cheapest among the different orderings and the order-independent plan (unless, of course, the query had a final ORDER BY clause, in which case it will choose the appropriately ordered plan).

Cost and Cardinality Estimation

Cost and cardinality estimation are closely related: for each algorithm, the query optimizer has a *cost formula* (generally posed in terms of time units) that captures how expensive it is to evaluate a given operation, given its expected input and output sizes. For instance, the cost formula may consider how many disk pages need to be retrieved for a given size of a relation, or how many times a hash function needs to be invoked to look items up in a hash table. (The cost formula usually also has a series of *calibration parameters* that are set during DBMS configuration, to match the speed of the current hardware on which the DBMS is operating.)

Cardinality estimation relies on information gathered by the DBMS offline (and periodically refreshed). The query optimizer can typically look up the cardinality of every base relation (from the system catalog and/or the existing indices), and for indexed attributes, it also can determine the minimum and maximum values. Additionally, the DBMS "tuning wizard" or human administrator may have created *data distribution* information — typically histograms and the number of unique values — over some of the attributes in the tables. Finally, the DBMS integrity constraints (particularly key, unique, and foreign key constraints) may allow the system to infer information about the distributions of certain attribute values.

Based on this information, the query optimizer can attempt to estimate how many values satisfy a selection predicate (by determining which histogram buckets, and subsets thereof, are likely to satisfy the predicate) or a join condition (by intersecting histograms). When multiple predicates exist within a query, the optimizer must typically make simplifying assumptions, e.g., that attributes are independent. In practice, the optimizer's assumptions typically are fairly error prone but the hope is that they at least ensure the optimizer will not choose exceptionally bad plans.

8.1.2 Executing a Query Plan

The execution engine takes the physical plan, which specifies a series of algorithms to apply, and performs the desired operations over the source relations. Here there are several options with respect to *granularity of processing* and *control flow*. We discuss each of these, and their design goals, next.

Granularity of Processing

When we learn the relational algebra, we tend to think of the query operations (select, project, join, etc.) as atomic operations that take in relations and produce relations. In reality, this is seldom the way a query execution engine will do the computation, for two main reasons. First, the intermediate relation produced by each operator can be larger than memory, and it may be quite expensive to buffer this intermediate table and spool it to (and then from) disk. Perhaps more importantly, we might want multiple operations in the query plan to execute *in parallel*, to reduce the latency to the first query answer.

Hence, most query engines support *pipelining* during query processing. Each operator reads enough input tuples to produce a single output tuple. Then it passes this to the next stage (its parent operator in the query evaluation plan). This process repeats as operators successfully produce output. If that operator is able to produce an output tuple, it passes this to *its* parent operator, and so forth. (We discuss control flow among the operators next.)

Of course, it is not always feasible to pipeline all of the operators in a query plan. First, many operators (particularly some join, grouping, and sort implementations) require significant memory, to buffer all of the input tuples they have encountered. There may not be enough memory to give to all of the operators at the same time. Second, some operators (e.g., sorting, grouping over a nonsorted attribute) are naturally *blocking*: they cannot begin producing output data until they have "seen" their input in its entirety. Finally, sometimes the query optimizer decides it is advantageous to compute and save an intermediate result (e.g., to share it across multiple subexpressions or queries). In all of these cases, a *materialization point* appears in the query plan: the intermediate relation at this point is computed, buffered, and, if necessary, saved to disk.

Example 8.2

The query plan of Figure 8.3 contains a hash join operation (`HJoin`), which is a *partly blocking* operator. Hash join consists of two stages: a *build* stage, in which one of its inputs (in our example, the right relation, `Reviews`) is read into a hash table; then the *probe* stage, in which input tuples are pipelined from the left input, joined against the contents of the hash table, and output to the next pipeline stage. The build stage blocks execution of the remainder of the plan until its input has been consumed; the probe stage is fully pipelined.

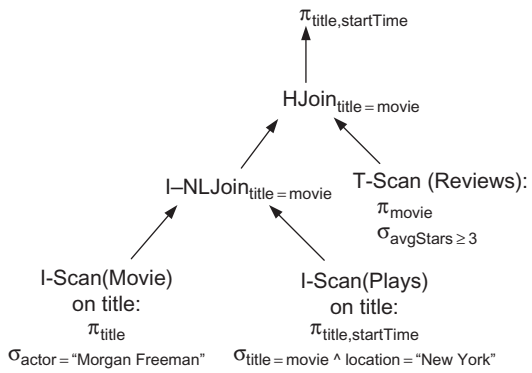

FIGURE 8.3 Example executable (physical) query plan corresponding to Figure 8.2, where **I-Scan** represents an index scan, **T-Scan** represents a table scan, **I-NLJoin** represents an index nested loops join, and **HJoin** represents a hash join.

The materialization point marks the highest point in the query plan where tuples will have been received: the operators above the materialization point will only receive input tuples once the materialized result has been fully computed. Thus one can think of a materialization point as a "barrier" dividing one stage of query processing (where all operators are working in the same pipeline) from the next.

Control Flow

There are a variety of methods by which the query operator algorithms can be scheduled. At one extreme, one could simply place each operator in its own thread (or on a separate processor), and operators could send tuples to one another through queues. At the other extreme, there could be a single execution thread, which carefully controls when each operator gets cycles.

On a single machine, the traditional DBMS architecture more closely resembles the latter: it uses a top-down *iterator*-based architecture, in which each operator calls its child operators to retrieve a tuple at a time, then performs operations over such tuples, and returns the results to its parent operator(s) or to the query output buffer. The iterator model has key advantages in a CPU-bound setting: it eliminates the overhead of context switching among threads, and it generally obviates the need for copying data items into and out of queues.

In distributed and parallel DBMS settings (which we discuss further in the next section), query processing is often I/O-bound: many I/O operations may be occurring simultaneously, and the goal is to free the CPU to work on data values as they come in. For these architectures, the *dataflow* or bottom-up architecture is popular. Here, there may be many query operator threads, each of which is event-driven. As an input tuple becomes available, it is fed to the query operator thread, which consumes it, processes it, and outputs it to some other thread's input queue.

In recent years, hybrid architectures have been extensively studied. Rather than propagating tuples one at a time in the pipelined model, various systems attempt to propagate and operate on *batches* of tuples: this has advantages in terms of cache behavior and CPU scheduling. Additionally, hybrid schemes combining the iterator model with a limited number of threads (e.g., for data prefetching) have been employed for handling external data.

8.2 Background: Distributed Query Processing

Of course, conventional databases were not solely limited to single server settings; distributed DBMSs have been studied for many years. We will highlight here some of the key innovations required to move to the distributed context, as each of these innovations came to influence work in the data integration realm.

PARALLEL VS. DISTRIBUTED DBMSs

On first impression, parallel and distributed DBMSs seem quite similar: both involve taking multiple compute and data storage nodes and coordinating the various machines in

order to handle queries. However, the basic assumptions and approaches used to address these two problem spaces are quite different. In parallel DBMSs the nodes are typically assumed to be homogeneous and connected to a global storage system by a fast network. The main goal is to partition the data across machines and run the same operations in parallel across many machines, such that load is balanced and high speedups are achieved. In distributed DBMSs, different machines may have very different performance characteristics, network performance is likely to be slower and may vary widely across the system, and some data sources may only be accessible from certain machines. The challenge is to determine *which operations* should be executed on *which machines*, in order to optimize for performance. We shall focus our discussion on this latter case.

8.2.1 Data Placement and Shipment

In general, the administrator of a distributed DBMS must consider the problem of *data placement*: how to lay out data across a set of distributed machines. There are a variety of options: data may be partitioned by *relations*, such that certain tables are on one machine and others are on a different machine; by *horizontal partitioning*, meaning that different rows from the same table are placed on different machines; or by *vertical partitioning*, meaning that different columns from the same table are placed on different machines. Each has different performance ramifications: relation partitioning may allow different query subexpressions in the same plan to be computed in parallel on different machines; horizontal partitioning may allow the same query subexpression to be computed in parallel on different data located on different machines; vertical partitioning may allow for computing different query predicates in parallel on different machines. Of course, in addition to partitioning the data, one may also *replicate* and index it: this is likely to improve query performance, but increase the cost and other resources (space, possibly network bandwidth) required by updates.

Centralized DBMSs had one important property that greatly impacted performance: namely, the ordering (or lack thereof) of a query subresult. Distributed databases add a second such property, *location* (or, in the horizontally partitioned case, the set of locations). To compute a join or an aggregate, all data with the same (join or grouping) key must be co-located on the same query processing node.

These requirements have led to two new physical-level query operators: *ship*, which takes the output of a query subexpression and routes it (perhaps using buffering and batching) to the input of a query plan on another node, and *exchange*, which runs across many horizontally partitioned data nodes in parallel and "exchanges" tuples among the set nodes according to a partitioning function over a key, until data with the same key is co-located. The partitioning of data with the exchange operator can, in principle, be done by dividing ranges across the machines, by hashing keys to machines, or by performing some sort of load balancing. A hash-based exchange scheme is sometimes called a *rehash* operation.

Of course, if we are introducing new physical-level query operators, we naturally need to estimate their costs in the query optimizer. Historically, this has proven to be

surprisingly difficult in the real world, as performance on the Internet is often unpredictable. The cost of one link does not necessarily tell us much about the cost of other links (even links to closer sites!). Hence, early work on distributed query processing was primarily targeted at local area or enterprise networks, where costs were fairly uniform. Today, as bandwidth on the Internet has increased both on the backbone and at the end points, performance seems much more stable.

8.2.2 Joining in Two Phases

Given the ship (and possibly exchange) operators, one can build a complete distributed query processor that can execute SQL queries over data that are distributed across machines, even if they are partitioned. However, an efficiency issue arises: if we are given two tables, each located on a different machine, then computing the join is quite expensive. We must ship the entire contents of one table to the other machine (or both to a third machine) to do the computation. In many cases, a significant portion of the data may not even join; yet we still ship it.

This observation has led to work on computing distributed joins in two phases. Here, we ship a "summary" of the join keys for the first relation, R, to the second node. We use this to "prescreen" tuples from the second relation, S: those that appear to join with R (as determined by probing R's summary using the join keys from S) will be collected and sent back to the first node. The first node will then do the actual join of tuples from R and the subset of S it receives.

Using this scheme, we can often significantly reduce network bandwidth utilization, because R's summary is smaller than R itself, and we only transmit the subset of S that appears to join with R. Of course, the summary must satisfy certain properties if we are to use it: specifically, a probe against the summary must never have false negatives, i.e., if the values actually join, the probe must return **true**. Two such summaries have been considered in the literature.

TWO-WAY SEMIJOIN

The *2-way semijoin* operator makes use of the (set-semantics) projection of the join key attributes from R as its summary. The first node sends this collection to the second; when the second node probes against the collection, clearly there will be no false negatives (or false positives!). Thus, it will send back to the first node exactly the set of S tuples that join with R.

BLOOMJOIN

The 2-way semijoin can be expensive if the projection from R is large. The *Bloomjoin* reduces the size of the summary structure by instead using a *Bloom filter* of the join key attributes from R. A Bloom filter can return false positives but never false negatives. It is a bit vector V, accompanied by a set of m hash functions. Initially, all bits are set to zero. For each item x to be inserted into the Bloom filter, we set the bit at $V[h_i(x)]$ for each $1 \leq i \leq m$.

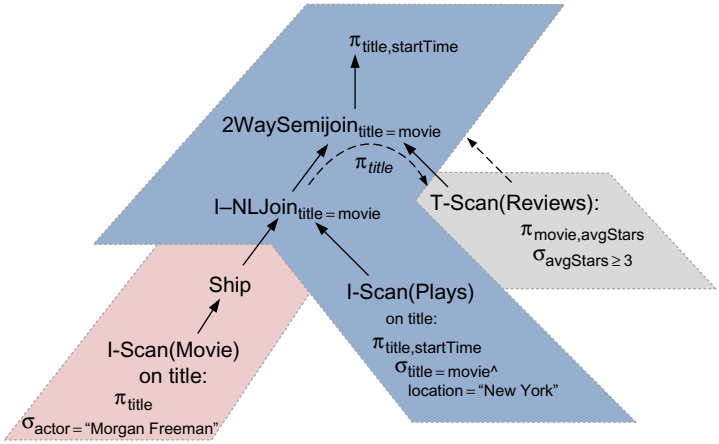

FIGURE 8.4 Example executable (physical) query plan corresponding to Figure 8.2, for query processing across three nodes. Note the use of the ship operator and the dataflow for the 2-way semijoin (indicated by dashed arrows and italicized text).

Later, we probe for an item x by computing $\bigwedge_{i=1}^{m} V[h_i(x)]$. Larger vectors, and larger values of m, result in fewer collisions (and thus fewer false positives).

Cost estimation for the 2-way semijoin and Bloomjoin is done by estimating the size of the summary of R sent in the first step, the size of S with which it joins, and the bandwidth and latency of the network. The end result is a query plan such as the one in Figure 8.4, which shows how data are shipped from the `Movie` node to the node with `Plays`. The `Plays` node also does a 2-way semijoin with the node holding `Reviews`, by sending the set of all titles and receiving the set of all reviews that match those titles. The `Plays` node does the final join and projection, returning results to the user.

8.3 Query Processing for Data Integration

At a very high level, the problem of executing queries in a data integration setting is a special case of the distributed query processing problem, in which distributed DBMS sites are replaced by distributed data sources. However, new challenges are posed by the fact that many of the data sources are located across the Internet and not simply an intranet, sources are *autonomous* and seldom designed to work in a cooperative setting, and information about source data is not readily available.

Figure 8.5 shows the overall query processing architecture that is followed by most data integration systems. As in a conventional DBMS, the first stage is query parsing. Then, rather than simply *unfolding* views, the data integration engine must *reformulate* queries using the schema mappings (see Chapter 2). Once the query has been reformulated, it needs to be optimized and executed — here, we see significant changes, not only within the optimizer and execution engine, but also in the fact that together they form a *loop* to interleave optimization and execution.

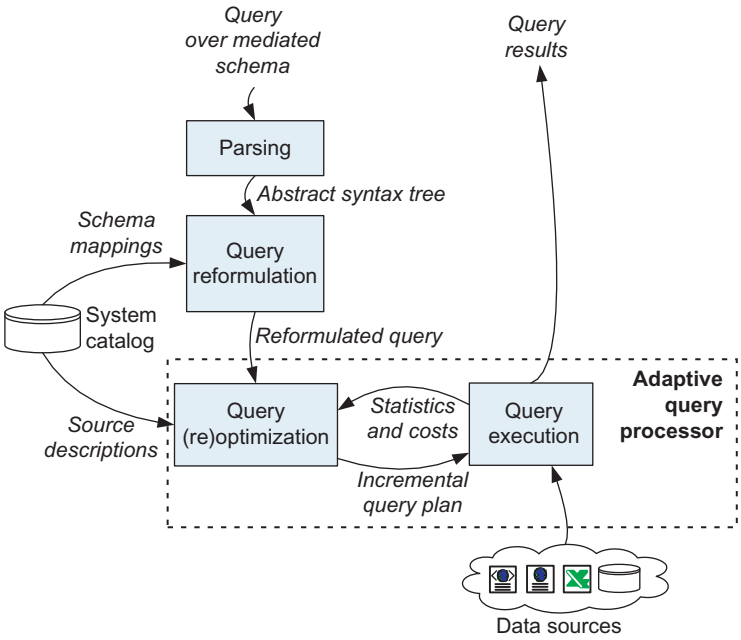

FIGURE 8.5 Modules in a data integration system.

The data sources to be integrated are generally controlled by external organizations (and, in fact, may not be DBMSs). This means that the data integration query processor must accommodate whatever interfaces a source provides to request and receive data. Simple sources may simply provide direct access to XML or HTML documents, from which the required data must be extracted. Other sources may provide a Web forms interface in which data can only be accessed by providing certain input parameters (see Section 3.3). Finally, some sources may in fact provide full SQL support (in some dialect of SQL). All of these constraints require careful treatment in the query optimizer (Section 8.4), as well as the development of custom *wrappers* that translate between a common query request/response format and those of individual sources (Chapter 9).

The fact that data integration occurs over the Internet has several ramifications. First, data transfer rates are more unpredictable, and both burstiness and limited bandwidth may be issues. Second, computation is much more likely to be I/O-bound, rather than CPU-bound, as in a local area network or a local query processor. Finally, in many Internet usage scenarios, we would like the *time to first tuple* to be low, and the output to be produced incrementally. These lead to a new style of query processing architecture and new query operators, as we discuss in Section 8.5.

Finally, in most data integration scenarios, too little information is readily available to make good optimization decisions at compile time. Actual network utilization,

distributions of data at various sources, even the response times of given sources that may be serving many tasks concurrently — all of these are likely to deviate from any original estimates that a query optimizer is given as input. In fact, these characteristics might also change significantly during the execution of the query itself! Hence, as discussed in Section 8.6, for long-running queries it is important to use *adaptive* strategies during query processing, where the query plan may be adjusted in response to actual observed performance characteristics.

8.4 Generating Initial Query Plans

The problem of optimizing a data integration query can actually be extremely challenging if little information is available about costs and data distributions. However, in some settings (e.g., the corporate intranet), such information may be available. In any case, the query processor must begin somewhere, at least generating a partial query plan to start things off. Typically, this is done using a modified version of the traditional query optimizer model, with any available statistics (or ballpark estimates if statistics are unavailable).

A data integration query optimizer must deal with two restrictions that are not present in a traditional RDBMS. First, the presence of *input binding restrictions* will mean that certain query plans will become illegal, simply because they do not bind values for the input variables needed to access a particular source. Second, some data sources are capable of performing part of the query computation (e.g., they may support joins between relations supplied by the source): here it becomes essential that the optimizer be able to determine what is supported, and what cost and cardinality to expect.

The problem of reasoning about input binding restrictions is not tremendously different from what a traditional optimizer already needs to do. The optimizer already needs to reason about when certain physical or even logical plans are illegal (e.g., when input attributes to a predicate are unavailable, or when a Cartesian product is needed, but Cartesian products are only allowed *after* all join conditions are evaluated). However, it has been shown that input binding restrictions sometimes require *exhaustive* evaluation of all query plans ("bushy" enumeration) as opposed to the System-R heuristic of only using left-linear plans. Additionally, the input binding restrictions must be accommodated by using a *dependent join* operator (Section 8.5.2) rather than a more traditional join algorithm.

Optimizing a query with wrappers is somewhat more complex. The query optimizer must determine which operators are to be executed in the wrapper and which in the main query engine, and also the *cost* of the query subexpression being executed in the wrapper. This requires wrappers to have both a query execution component (the actual wrapper operator) and a query optimizer component (a plug-in module that does plan validation and cost estimation for the wrapper). The optimizer can associate with every operator a *property* specifying where the operator is to be executed (by the wrapper or the

engine), and the wrapper's optimizer component will validate that a given query subexpression is actually executable within the wrapper. It will also give a cost estimate for the subexpression to be executed within the wrapper.

8.5 Query Execution for Internet Data

In the Internet setting, the CPU will often be waiting for new data to arrive before it can perform further computation. Moreover, burstiness, delays, and even failures are fairly commonplace. Additionally, a data integration engine must interface with a variety of external sources that support different protocols, and hence there is a need for special join methods that handle access pattern restrictions and wrappers that translate between a standardized request and response format and those of the specific sources.

8.5.1 Multithreaded, Pipelined, Dataflow Architecture

Data from external Internet sources often come at a relatively slow rate: this breaks the core assumption in most DBMS architectures (even those for local area networks), which is that query processing is primarily CPU-bound, and the goal is to have "tight loops" in the code that are carefully optimized. As discussed in Section 8.1.2, most conventional DBMSs use an iterator-based approach, in which the query execution algorithms in the plan determine how and when control is given to their child operators.

A data integration engine, on the other hand, typically uses a hybrid of iterator and dataflow-driven architectures: conceptually, each operator requests one data tuple at a time from its children, as in a conventional iterator model. However, the child operators may execute in threads independent from one another and from the parent (with whom they communicate via queues). Now, if one of the child operators is blocked waiting for input, the CPU can context switch to a different thread, where it may be possible to make forward progress.

THE PIPELINED HASH JOIN

The main multithreaded operator in a data integration plan tends to be the *pipelined hash join*, sometimes called the double pipelined join, the symmetric hash join, or the X-join. This algorithm constructs two hash tables, one for each input relation, and probes tuples from the opposite relation against them.

The pipelined hash join (Figure 8.7) treats each input symmetrically, simply monitoring its two input queues for available tuples. When it finds an input tuple from relation R, it *probes* the tuple against the hash table storing previously read results from the opposite relation S; it outputs each resulting tuple. Simultaneously, it adds the input tuple to the hash table for R (the *build* step). Upon encountering an S tuple, it proceeds symmetrically along the same lines. This join algorithm has several good properties: it immediately

produces output the moment once it has received enough tuples to do so, and it accommodates bursty or unequal transfer rates from the sources. However, a negative property of this algorithm is that it accumulates significantly more in-memory state than many alternative algorithms. A number of sophisticated mechanisms have been developed for swapping out portions of the state from a pipelined hash join, such that it can continue to steadily produce output even for larger-than-memory datasets.

HASH-BASED OPERATORS FOR FASTER INITIAL RESULTS

Beyond the pipelined hash join, most data integration engines tend to use hash-based — as opposed to index- or sort-based — algorithms for most of their processing operations, including aggregation. Indices are seldom available in the data integration context. Sort-based methods require multiple sort passes before they can produce output, and this is contrary to the common goal of producing early output. Hash-based schemes, even in situations where the data are larger than memory, allow for tuples to be lazily swapped to disk on a bucket-by-bucket basis as more memory is needed. In most cases, some buckets do not need to be swapped out, and these can be output directly by the query before it begins swapping in the swapped-out tuples.

8.5.2 Interfacing with Autonomous Sources

At the lowest level of a physical query plan for data integration, we typically have a call to a *wrapper* operator. This operator interfaces with a source-specific wrapper or driver (see Chapter 9 for more details), which is responsible for sending requests to the data source in its native protocol and returning results in the tuple format expected by the data integration engine.

Two "generic" types of wrappers appear in most data integration systems: those for ODBC and those for XML. An ODBC (Open DataBase Connectivity) wrapper typically connects to an ODBC loadable library, which is provided by all of the major relational database vendors for their product and each host operating system. ODBC takes queries posed in XML and returns row sets that are accessed via a cursor interface. The ODBC wrapper has a fairly simple job of translating the pushed-down query operations to SQL, then reading tuples from the row set. An XML wrapper typically converts the pushed-down operations into one or more XPath patterns, makes an HTTP request to fetch the desired XML document, and evaluates the XPaths over data as they stream in. Extracted results are returned to the query plan. We discuss XML processing in more detail in Chapter 11.

WRAPPER OPERATORS

In many cases, the wrapper operator has the ability not only to fetch data, but also to pose queries that are to be executed at the source. These are typically operations that are *pushed down* by the query optimizer: selection conditions, projections, and sometimes even joins

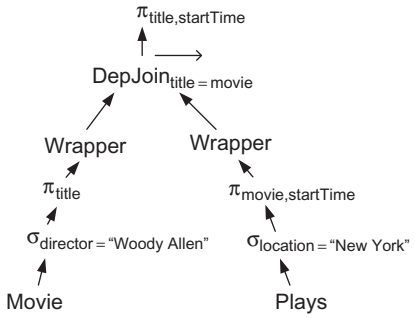

FIGURE 8.6 Example executable (physical) query plan with dependent join.

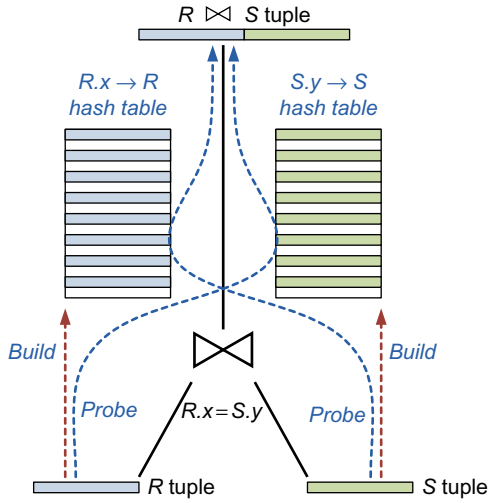

FIGURE 8.7 Internal details of a pipelined hash join. Each tuple from source R is stored in its corresponding hash table (the *build* step), then *probed* against the hash table from S to produce an $R \bowtie S$ result. Source S is processed symmetrically.

between tables that are co-located at the same source. Of course, as discussed previously in Section 8.4, the source must be capable of executing these queries.

HANDLING LIMITED ACCESS PATTERNS WITH DEPENDENT JOIN OPERATORS

As described in Sections 3.3 and 8.4, at times access to a data source must be done by sending inputs (bound variables) to the source, in order for it to return any output data. To do this for multiple sets of input values, we must use a *dependent join* operator: the dependent join fetches tuples from one of its inputs, then sends these as inputs to the source wrapper, and finally returns the cross-product of the results from the two inputs. The dependent join is a special case of the 2-way semijoin described in Section 8.2.2 and can be implemented using the same techniques. Figure 8.6 shows the dataflow of a dependent join, where Movie titles are passed to the Plays source.

8.5.3 Handling Failure

Sometimes, Web sources may become unavailable, e.g., due to a system crash or network partition. Several Web-oriented data integration systems provide for this capability by providing *event handling* capabilities: when a source failure is detected, this can trigger simple modifications to the executing query plan. Often, this involves fetching data from an alternate set of sources (e.g., a mirror or a slightly out-of-date replica). We discuss how these events can trigger adaptive behaviors in Section 8.7.

8.6 Overview of Adaptive Query Processing

Perhaps the main lesson in query processing over the Internet is that conditions constantly change or fail to match expectations: cardinalities or costs may not match optimizer predictions; data streams may get delayed; nodes may fail. Naturally, the approach to handling such issues is to develop a query processor that is capable of *adapting* its strategy in response to such conditions.

In general, this means that the dividing line between the query execution engine (which simply executes query operators) and the query optimizer (which determines strategies) starts to blur. Instead, we have a *feedback loop* in which initial cost and data distribution information is consulted; a preliminary plan is formulated based on comparing predicted costs; the plan begins executing; cost and data distribution information is updated; and we repeat the process.

All adaptive query processors must balance a number of trade-offs.

SPARSITY OF INFORMATION
In general, there are many alternative options for which plan to adapt to — but the actual information available as a basis for decision making (e.g., data distributions, correlations) tends to be limited. This often leads to a trade-off between *exploration* (spending resources to gather information) and *exploitation* (spending resources on producing results).

ANALYSIS TIME VS. EXECUTION TIME
Similarly to the above, we need to determine how many resources to spend on decision making versus execution. Adapting more frequently typically allows us to make the query plan more efficient, but also adds more overhead in continuously reoptimizing. Likewise, searching a more complex space of plan alternatives leads to more optimization time, possibly with the benefit of more efficient execution strategies.

SPACE OF TRANSFORMATIONS
Generally, the more flexibility we would like to enable in changing plans, the more overhead (whether extra memory or extra cost to redundantly compute results) is required.

THE IMPACT OF STATE, NOW AND IN THE FUTURE
Joins are inherently stateful operators, and the cost of a join increases over time (under extreme conditions, quadratically with the size of the input). We must be very careful not

to underassess the cost of a join in a way that leads to *greater* creation of state (e.g., build large intermediate results when it looks like few tuples join, then later find that we are incurring large overheads in processing these results).

We classify adaptive techniques into two broad categories: adaptivity driven by *events* in the query processing environment and adaptivity driven by *performance* monitoring and the determination that query processing should be modified. We begin with the event-driven case.

8.7 Event-Driven Adaptivity

Clearly, one of the major motivations for adaptivity in data integration, especially on the Web, is that unanticipated events may occur. For instance, sources may be unavailable, necessitating a fallback strategy; there may be delays in receiving data from a particular source or set of sources, suggesting that we may wish to prioritize some other portion of query execution; or, at the end of a pipeline stage, we may discover that our intermediate results deviate dramatically from our original expectations, suggesting that it may be beneficial to reoptimize the remainder of the query plan.

All of these can be handled through a general *event-condition-action rule framework*, in which rules may be triggered by runtime events, conditions may be evaluated, and actions may be initiated in response. Event-condition-action rules have the general form **on** *event* **if** *condition* **then** *action sequence*.

Typical events include the completion or initiation of a new pipeline stage, an error or timeout from a wrapper operator, or the processing of a certain number of tuples. For example, the following rule traps a timeout from the query operator named *wrapper1* and in response always activates operator *wrapper2*.

on timeout(wrapper1, 10 msec) **if** true **then** activate(wrapper2)

We can model a rule as a quintuple ⟨*event, condition, action_sequence, owner, active*⟩. The *owner* is the query operator that the rule monitors. If the owner is active, and the *active* flag for the rule is also true, then an *event* can initiate or *trigger* the rule, causing it to test its *condition*. The condition is a Boolean expression with the standard operators, whose terms include integer and Boolean literals, plus predefined functions that poll query operator status: the size of intermediate state produced in an operator, cardinality of tuples read or output by the operator, how long since the operator last received a tuple, etc.

If a rule's conditions are satisfied, then the rule *fires*, i.e., the runtime system executes the actions within the *action sequence*. In an adaptive query processor, these actions may enable or disable portions of a query plan, terminate execution and report errors back to the user, or trigger a reoptimization of a portion of the query plan. A rule's *active* flag is cleared once it fires.

In the subsequent sections, we describe how rules may be used to handle a variety of events and the ensuing actions that may be taken.

8.7.1 Handling Source Failures and Delays

When a data source is completely unavailable, e.g., due to a server crash or a network partition, one option is to simply return an error to the user, stating that the query cannot be completed as specified. However, in certain cases, there may be alternative ways of getting equivalent data to what was at the original source, e.g., if mirrors exist, or if data from other semantically equivalent sources can be combined.

Conceptually, this type of task could be modeled as a relational union operator, whose inputs are the alternative data sources. However, we do not necessarily want to request data from all sources simultaneously. This can be accomplished through a *dynamic collector* operator, which is essentially a union operator with a special *policy* expressed as rules.

Finding Alternative Sources

To handle source failures, the query optimizer generates dynamic collector nodes, with multiple subtrees, each representing a different subplan for acquiring the desired data. The optimizer generates a set of rules for enabling and disabling the sources. Some of the inputs to the collector are enabled upon query start-up; then, based on events (such as timeouts, outright failures, or tuple arrivals), a set of rules associated with the dynamic collector can enable or disable the various inputs.

In the simplest case, the optimizer can generate rules for handling source failures or even timeouts. Consider the following example, where operator *coll*1 has two wrapper operators connecting to sources *A* (the primary source) and *B* (the alternative).

on timeout(A) **if** true **then** activate(coll1,B); deactivate(coll1, A)

More elaborate event-based query processors even generate events on a per-tuple basis, enabling *competitive execution*, where we request data from more than one source in parallel and terminate the request from the slower source.

The following example illustrates a situation where initially we attempt to contact sources *A* and *B*. Whichever source sends 10 tuples earliest "wins" and "kills" the other source.

on opened(coll1) **if** true **then** activate(coll1,A); activate(coll1,B)
on tuplesRead(A,10) **if** true **then** deactivate(coll1,B)
on tuplesRead(B,10) **if** true **then** deactivate(coll1,A)

The basic event handling scheme enables great flexibility in handling network issues, although it requires that the optimizer have good information about alternative means of acquiring the data.

Handling Network Delays with Rescheduling

In a complex query plan that is not fully pipelined, a delay at one of the sources can completely stall execution — even if the other data sources are available, and if some other

portion of the query plan might be executable. A response is to *reschedule* query execution, such that the delayed portion of the plan is suspended but another segment of the plan can be started in the meantime.

In general, the process of rescheduling involves identifying a new *runnable subtree*, and splitting it off from the current plan — into a separate query whose output becomes a materialized table. The current query will be modified "in place" to use the results of this table. This strategy enables us to execute both the main query and the "subtree query" concurrently, and still preserve the correct semantics of operation.

We would like to consider which runnable subtrees to use by looking at their costs. Typically all rescheduling decisions are predetermined at optimization-time, with the decisions encoded as rules to be executed at runtime, specifying which subtree(s) to activate if a given pipeline stage gets stalled. This only requires minor modifications to the optimizer's existing structure: the optimizer already has estimated costs and cardinalities for each subtree, and in assessing the cost of rescheduling, it simply must also account for the costs of materializing and reading from the materialized tables. Alternative runnable subtrees can be prioritized by their *efficiency*: the ratio of how much cost they save from the existing query plan versus the cost of executing them as separate queries with materialized results. Intuitively, the most efficient runnable subtrees are ones that materialize a small amount of data, or materialize data that already gets materialized in the existing query plan.

Once a rescheduling policy has been determined, it is simple to encode it using a rule such as the following.

on waiting(op1, 100 msec) **if** true **then** reschedule(op2)

Results from the research literature show that this rescheduling technique can provide significant speedup when there are multiple pipeline stages in query execution. We note that the use of pipelined hash joins mitigates the need for rescheduling within a select-project-join query; however, for queries with aggregation, or situations in which there is insufficient memory to do a pipelined-hash-join-only query plan, rescheduling is an important technique.

8.7.2 Handling Unexpected Cardinalities at Pipeline End

Sometimes, even in data integration scenarios, query execution is divided into multiple pipeline stages — either due to a lack of memory, or because of blocking operations such as aggregation. At the end of each pipeline stage, the query processor has an opportunity to see whether the overall query plan is progressing as expected and to replace the remaining plan "for free" if not. Of course, *determining* what alternative plan to use may first require reoptimization, which in fact is not free.

The technique of *mid-query reoptimization* attempts to determine whether it is beneficial to reoptimize the remaining pipeline stages of a query plan, based on a combination of runtime statistics, heuristics, and cost estimates. There are three

basic components to this technique: extending query plans with the ability to gather information (such that progress can be monitored), having the optimizer predetermine the settings under which reoptimization should be triggered, and, if necessary, reinvoking the optimizer at runtime to get a new query plan. We discuss each in turn.

Information-Gathering Query Operators

The operators in a traditional query engine do their computation "silently," in the sense that they do not provide feedback on how much data they have received, what the data distributions look like, and so forth. It is essential for an adaptive query processor to provide mechanisms for monitoring status.

In general, these come in two main forms. First, individual operators can track information about their own status and return this upon request. The most common type of status information is how many tuples each operator has processed. Maintaining such per-operator cardinalities adds a small amount of overhead to query execution, but in exchange there is the possibility of determining exact cardinalities (and thus selectivities). Other kinds of information may include execution state (open, closed, failed), and possibly even information about memory consumption.

A second kind of information is *summary* information i.e., histograms or sketches over certain attributes in the data stream. Such summary structures are typically created by special *statistics collection* operators, which are inserted into the query plan by the query optimizer. The placement and use of such operators are typically quite tricky: statistics collection is relatively expensive, and the summary structure typically is only usable *after* the pipeline containing the statistics collector operator has completed. Hence the optimizer attempts to place statistics collectors where critical attributes (e.g., join keys) are uncertain, and where a significant portion of the query plan might still be reoptimize (i.e., a good deal of the remaining query plan must not yet have been started).

Predetermined Reoptimization Thresholds

Given information from the query plan (whether cardinalities of operators or summaries from statistics collection operators), the query engine must be able to make a decision as to whether the query plan is progressing in an acceptable way (i.e., performance will be roughly along the lines of what was predicted by cost-based optimization, or better) or if it is necessary to try to find a new plan.

To change a query plan we must materialize the results of the currently executing pipeline to disk, reinvoke the optimizer over the remaining portions of the query, and execute the new query plan. Clearly, this adds two kinds of additional overhead to the overall query processing running times: that for writing and later reading the materialized results, and that for invoking the query optimizer.

The goal is to only trigger a reoptimization if the total savings will exceed this additional overhead. Of course, the challenge is that at runtime we do not have any simple way of estimating how much savings is possible: this requires the query optimizer! Hence the system must rely on heuristics to determine when to reoptimize.

Typically, the cost of optimization is fairly predictable, as is the cost of materializing and dematerializing a subresult. Let these combined costs be represented by the term C. Then the optimizer will typically be reinvoked if (1) the newly estimated cost of the query, given the new statistics gathered at runtime, exceeds the originally predicted cost by some threshold θ_1, and (2) the predicted cost of the unexecuted portion of the query exceeds C times some coefficient θ_2.

Example 8.3

Refer to Figure 8.8 for an instance of mid-query reoptimization for the SQL query given at the beginning of this chapter. The plan is broken into two pipelines, with a checkpoint between them. The first pipeline contains a pipelined hash join, a statistics collector, and two selection operations. At the end of the pipeline, the checkpoint operator may have a rule such as the following, which calls the optimizer to reoptimize the subsequent fragments if the estimated cardinality is significantly different from the size of the result.

on closed(join1) **if** card(join1) > (est_card(join1) * 1.5) **then** reoptimize

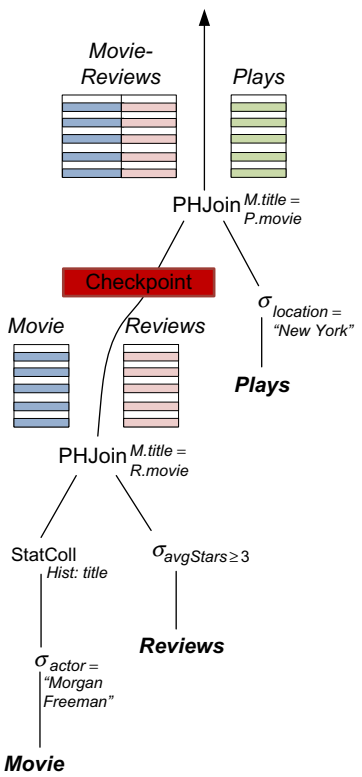

FIGURE 8.8 Example of a query plan with a reoptimization checkpoint, plus a statistics collector on the *title* attribute.

If the output of the first join is no more than 50% larger than what was expected, then the second pipeline continues unaffected. Otherwise, the second join is reoptimized (e.g., to replace the pipelined hash join with a nested loops join over the temporary table, due to lack of memory).

Runtime Reinvocation of the Optimizer

Once reoptimization is to be triggered, this requires suspension of the currently executing plan. This requires the following steps:

- Take the output from the last operator in the pipeline and materialize it to disk as a temporary table.
- Rewrite the original SQL query Q, replacing the already-executed subexpressions of the query with a reference to the temporary table, resulting in a query Q' that encapsulates the not-yet-executed operations from Q.
- Optimize Q', possibly adding statistics collection operators and reoptimization thresholds again, if Q' is to be executed in multiple pipeline stages.

The resulting plan for Q' can now be run to completion, producing the final query results. As an optimization, it is possible to *suspend* execution of the original Q before triggering reoptimization, rather than terminating it — then if Q' has approximately the same costs, it is slightly more efficient to resume computing Q than to start executing Q'.

The mid-query reoptimization approach can be quite useful if there are multiple pipeline stages in a query plan. However, in practice many data integration queries are fully pipelined, rendering the technique ineffective. This motivates the work in our next section.

8.8 Performance-Driven Adaptivity

In data integration scenarios where we do not have good cardinality estimates for base relations, the techniques of the previous subsection are generally inadequate: they can only modify portions of the query plan that are not part of the currently executing pipeline. Given that data integration queries emphasize the use of pipelined hash joins, they typically only *have* one pipeline (possibly followed by an aggregation operation). Hence there has been a great deal of interest in adaptive techniques that allow the query processor to change the execution plan for a currently executing pipeline.

All adaptive query processing systems in this vein assume that past query performance is indicative of future performance and extrapolate the overall cost and cardinality values from this. While this heuristic can be misled by variations in the data, such situations are unlikely to arise in practice: the majority of query cost tends to be in join operations, and the majority of joins are between keys and foreign keys, which may demonstrate skew but are fairly well bounded in the size of their output.

Under the covers, all of these techniques exploit a set of fundamental properties of the relational algebra — namely, the distributivity of union over select, project, and join. In general, we can take any conjunctive query expression over m relations, divide each relation into partitions, and compute the overall result as a union of conjunctive queries over the partitions:

$$\Pi_{\bar{A}}(\sigma_\theta(R_1 \bowtie \ldots \bowtie R_m)) = \bigcup_{1 \le c_1 \le n, \ldots, 1 \le c_m \le n} \Pi_{\bar{A}}(\sigma_\theta((R_1^{c_1} \bowtie \ldots \bowtie R_m^{c_m})))$$

where $R_j^{c_j}$ represents some subset of relation R_j.

This property is actually fundamental to the correctness of the pipelined hash join algorithm. There, at any point during the join of R_1 and R_2, we have an option about whether to read from R_1 and probe against the portion of R_2 read so far, or from R_2 and probe against the portion of R_1 that has been read. Suppose we represent the portion of R_1 that has been read as R_1^0 and, likewise, R_2's portion as R_2^0. Then the pipelined hash join can read more tuples from, say, R_1 (representing a subset R_1^1). It will join this with R_2^0, algebraically performing the operation $(R_1^0 \bowtie R_2^0) \cup (R_1^1 \bowtie R_2^0)$, but returning the same result as $(R_1^0 \cup R_1^1) \bowtie R_2^0$. Similarly, for R_2^1, we would join against R_1^0 *and* R_1^1, and so on for future subsets.

We can further exploit these algebraic equivalences to enable *inter-pipeline* adaptivity, by noting that we can evaluate each of the conjunctive queries within the union above using *any* order of evaluation. In other words, each of the different terms to be unioned might be produced by a *different* query execution plan.

The main question, then, is *which* plan to use for each of the different conjunctive expressions. Several different adaptive techniques have been proposed that opt for different points in the design space. We focus on two endpoints in the spectrum: *eddies*, a technique that enables very frequent adaptivity but makes decisions in a non-cost-based way, and an alternative strategy that makes less frequent, but cost-based decisions, called *corrective query processing*.

8.8.1 Eddies: Queueing-Based Plan Selection

Traditional DBMS query processing puts all of the "intelligence" and decision making into the query optimizer and very little intelligence in the execution engine. Operator scheduling is tightly controlled by the query plan, which consists of a tree (or, occasionally, a directed acyclic graph) of unary and binary operators.

The *eddy* approach in some sense chooses the opposite extreme: the optimizer makes very limited decisions, perhaps performing very high-level heuristics-based query rewrites, but deferring all operator ordering decisions within a select-project-join query block to runtime. Each SPJ expression gets codified into one "super-operator" called an eddy.

The eddy's functionality can be divided into a *tuple router* component and one or more *suboperators*. Suboperators can include selection and projection operators, as well as components that encapsulate join functionality (storing state for a subexpression and probing against state from a different subexpression). Tuples are fed into the eddy from the various inputs. The tuples are all annotated with special Boolean *done* flags, one for each query plan operation, to track their progress. For an initial incoming tuple, all bits are cleared.

Each suboperator has a set of *policy* rules, describing which done flags need to be set or cleared in order for the suboperator to be able to accept a tuple. No suboperator o will accept a tuple whose done bit for o is already set: this means the operator has already been applied to the tuple. Additionally, sometimes there are dependencies in the order of evaluation, and a suboperator may refuse to accept a tuple until that tuple has been processed by other suboperators first.

As a tuple comes into the eddy, its *tuple router* will typically have a choice among several destination suboperators, each of which is willing to accept the tuple. It will use a *routing policy* to choose among these destinations and send an incoming tuple to the appropriate suboperator. Any resulting output tuples from this suboperator o get their done bits for o set and also "carry forward" any set done bits from their inputs; then they are sent back to the eddy. The eddy will then consult the remaining suboperators to see which are willing to accept the tuple, and repeat the process. Once a tuple gets all of its done bits set, it is returned as output from the eddy.

Example 8.4

Refer to Figure 8.9 for an eddy corresponding to the SQL query below (repeated from the beginning of this chapter).

```
SELECT title, startTime
FROM Movie M, Plays P, Reviews R
WHERE M.title = P.movie AND M.title = R.movie AND
  P.location="New York" AND M.actor="Morgan Freeman" AND
  R.avgStars >= 3
```

The different query operations (selections and pipelined hash joins) are each encompassed as a suboperator ($o_1 \ldots o_5$) and the central router is indicated by the "R" node in the middle of the eddy. The routing constraints are indicated by connections in the "R" node between sources and suboperators. Each incoming tuple to the eddy is routed to one of the suboperators; returned results are fed back into the eddy. At each successive step through an operator o_i, the ith "done" bit on the output tuples is set. Once a tuple has all bits set, it is output by the eddy.

If the suboperators consist of select, project, and stateful join operations, then the eddy maintains correctness thanks to the relational algebra distributivity law properties

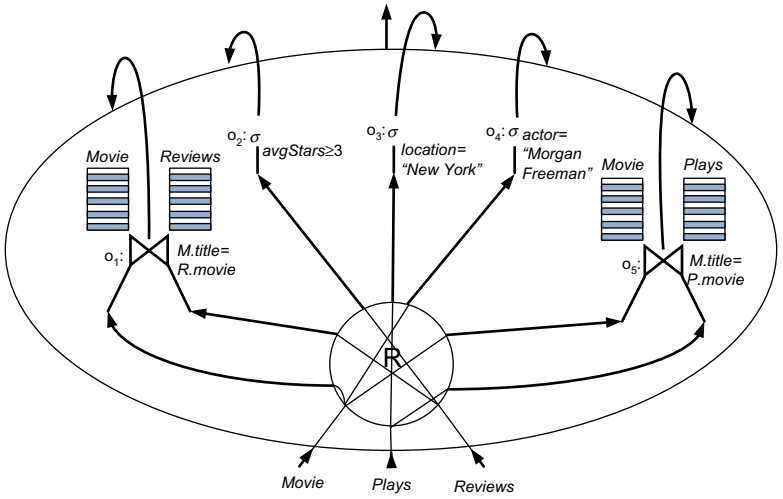

FIGURE 8.9 An eddy for our example query, showing the router ("R"), the various suboperators (o_i), and the internal hash tables in each join operation.

previously described. The major question with the eddy is what *policy* is chosen to select an order of application.

Basic Eddies: "Lottery Scheduling" Routing

The *lottery scheduling* scheme for routing is based on several observations. In general, it is too expensive to make a full cost-based comparison of alternative query plans on a tuple-by-tuple basis. However, we intuitively would prefer to send a tuple to *fast* suboperators over *slow* ones, all other things (in particular, selectivities) being equal; or to *highly selective* operators as opposed to *not very selective* ones, all other things (in particular, computation costs) being equal.

We can bias a tuple router towards fast suboperators as follows. If we place a queue in front of each suboperator, then the queue will quickly fill up for slow operators, but not for fast ones. The eddy's tuple router should *only* be able to send a tuple to a queue that is not empty, and *back pressure* from the queues will therefore bias the eddy to select fast operators.

This scheme can be extended to bias toward more selective operators through a randomized scheme. Each time a suboperator receives a tuple from the tuple router, it is given a "ticket"; each time it outputs a tuple to the router, it loses a ticket. The tuple router will choose the next destination for a tuple by conducting a "lottery" and choosing a ticket from among those for the eligible suboperator destinations.

The lottery scheduling routing scheme adds overhead, but is extremely effective in determining a good order for applying selection predicates. However, it tends to have problems with joins: the selectivity tends to diminish over time, as more tuples are "seen" by the join and added to its internal hash tables. The lottery scheme will often overestimate

the selectivity of a join early in execution. Thus, rather than sending tuples first to, say, a selection predicate, the eddy may first send them to the join — creating large amounts of intermediate state. Later in execution, tuples from the other sources will need to be joined against all of that state, incurring cost that could have been avoided if the selection had been applied first (reducing the state).

Extended Eddies: State Modules

The basic eddy scheme has a bit of a quirk: it does not reflect the "stateful" properties of join operators, e.g., the fact that the rate of tuple production for a join goes up as more input tuples have been consumed. State modules or STeMs "split apart" the internal functionality of the join so an eddy can better track behavior. The STeMs' work creates *two* suboperators for each join within the eddy: one suboperator holds state and the other probes against the opposite relation's state. Now an arriving tuple must pass through both the state and the probe suboperator, in a way that ensures all results are produced: this results in a series of intricate rules on how tuples are propagated into and out of the eddy. Further details on STeMs can be found in the papers listed in the bibliographic notes of this chapter.

Eddies That Migrate State: STAIRs

Eddies, whether using the basic approach or STeMs, assume that current observed operator selectivities are good predictors for future operator selectivities. For joins this is not necessarily the case: early in execution, join operators may be extremely selective; as state is built up, they may become less so. Under extreme circumstances, a join may initially produce no results at all, but may later produce large numbers of tuples. Here, the eddy might be "misled" into accumulating state in a suboptimal way — and once that state has been built up in a suboperator, every arriving tuple must be probed against it.

STAIRs are designed to address this problem, by letting the eddy "disassemble" and move intermediate state to other suboperators (possibly being filtered in the process). This adds some overhead due to wasted work, but it can be less costly than leaving the intermediate state in its suboptimal locations. For more details on STAIRs, please consult the bibliographic references at the end of this chapter.

8.8.2 Corrective Query Processing: Cost-Based Reoptimization

Eddies perform *continuous* query reoptimization, using a data flow heuristic that adds some per-tuple overhead but enables continuous simultaneous exploration of alternative query plans. Eddies will generally avoid "worst-case" query performance, but they spend a significant time doing "exploration" of options, and hence they devote fewer resources to "exploitation" — providing peak performance — than a traditional query engine.

Corrective query processing (CQP) aims instead to dedicate the majority of computation to query answering rather than exploration of potentially better plans. It seeks to be *reactive* to bad plans, making minor course corrections, rather than *proactive* in finding better plans. Unlike eddies, which use a local heuristic, CQP seeks to perform a cost-based

evaluation of plan progress and potentially better plans. To facilitate this, the query processor periodically reruns the query optimizer's cost estimator in a low-priority background thread with the latest runtime statistics, as query execution continues. If the optimizer finds a plan that is substantially better, it can halt the currently executing query plan, allow the plan to reach a *consistent* state (where all computation in the query plan, including blocking operations, has been performed on the source data that has been read), and switch to another plan using the adaptive mechanisms described previously. The new plan is "connected" to the input data streams, which resume where the previous plan left off. Execution continues, potentially switching plans more than once; each sequential change of plans is termed a *phase*. Finally, the system performs a *stitch-up phase* at the end to compute the answers requiring data from *across* plans.

Example 8.5

Refer to Figure 8.10. Given our example query and a set of initial statistics, the query optimizer may start execution using the query plan on the left of the figure, with `Movie` (M) joined with `Reviews` (R) before `Plays` (P). We refer to the initial plan as the *phase 0* plan.

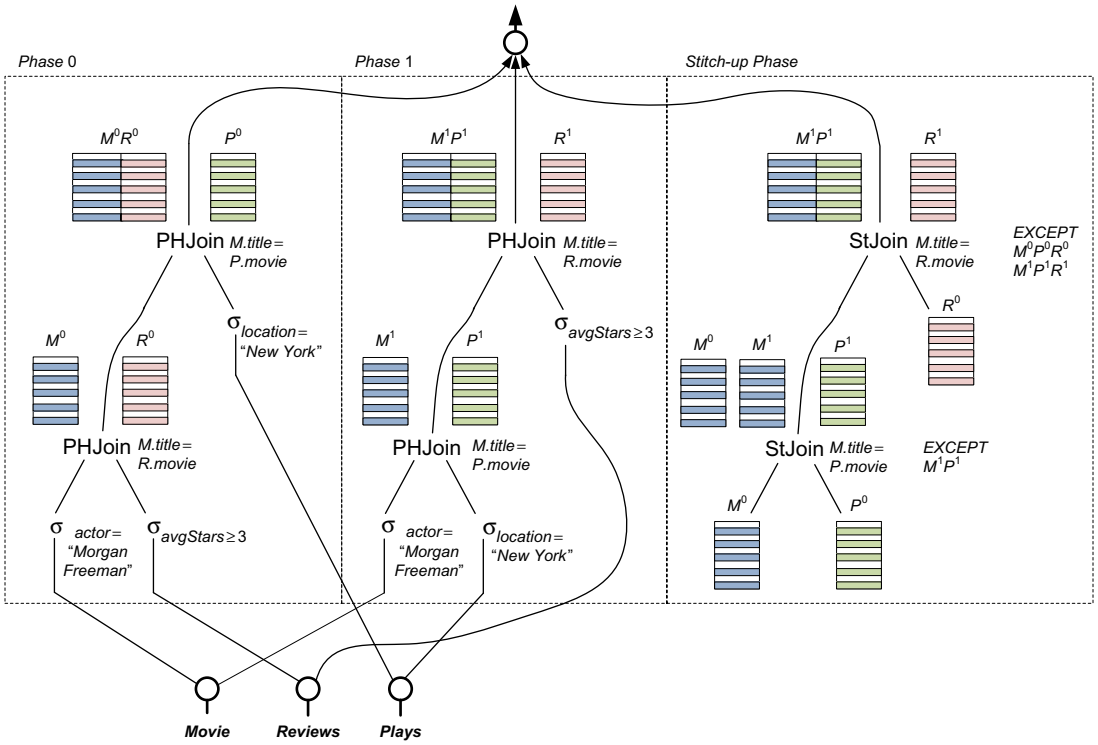

FIGURE 8.10 Corrective query processing example with two normal phases plus a stitch-up phase.

Unlike in a traditional query processor, a CQP-based query processor performs continuous monitoring on the execution of the plan. As statistics are collected and costs are reestimated, the system may trigger the query optimizer in the background, and may determine that an alternative plan is superior. The plan of the first phase is halted, and a new *phase 1* plan is initiated, indicated in the middle of the figure with `Movie` joining with `Plays` before `Reviews`. The new plan executes over the remainder of the data from the three sources: each phase has operated on a disjoint partition of the base data. We designate the first subset of each relation R as R^0 and the second subset as R^1.

The simple union of the 0^{th} and 1^{st} phase results returns only a subset of the desired answers. We must also join all combinations of relations *between* phases, feeding their results into the grouping operator. Omitting the join subscripts for conciseness, the remaining join expression is the following:

$$(R^0 \bowtie P^0 \bowtie R^1) \cup (R^0 \bowtie P^1 \bowtie R^0) \cup (R^0 \bowtie P^1 \bowtie R^1) \cup$$
$$(R^1 \bowtie P^0 \bowtie R^0) \cup (R^1 \bowtie P^0 \bowtie R^1) \cup (R^1 \bowtie P^1 \bowtie R^0)$$

Only when this final expression has been evaluated, in what we call the *stitch-up* phase, will query execution complete. This is done in the last query plan, where a special *stitch-up join* operator (StJoin) takes the intermediate result hash tables from previous execution and tries to use them to produce the remaining answers. This involves iterating through the contents of hash tables and joining them. In order to avoid producing duplicate results, the stitch-up join is given information about which combinations of hash tables *not* to rejoin (the "except" lists).

The CQP problem poses several key challenges, first in terms of how to supplement a query engine with cost estimation and reoptimization capabilities that run during pipelined execution, and later in computing the stitch-up query plans.

Cost reestimation

The corrective query processing approach is based on performing frequent reestimations of query execution cost. The plan status is generally polled every few seconds, as new aggregate performance trends become evident. During the polling step, the system must determine (1) how the remainder of query processing will proceed (especially if the data sources' cardinalities are unknown) and (2) the costs and selectivities of alternative query plans.

Given cardinality and selectivity results from the currently executing plan, the query processor must account for the fact that the current plan only explores a very small piece of the (exponential) search space, and only gives a small amount of real information. The CQP work exploits the fact that *equivalent subexpressions* (regardless of query plan) will always have the same cardinality and selectivity. Even with this observation, the search space remains large, and the CQP cost estimator must make use of a variety of heuristics to make an estimate of the selectivities of subexpressions that it has not yet attempted to execute.

Typically, if the reestimated cost is within a small threshold of the original costs, then no reoptimization needs to be performed, i.e., the space of potential query plans does not need to be searched. However, if the cost exceeds the threshold, then reoptimization is indeed triggered.

Reoptimization

In contrast to other methods such as mid-query reoptimization, the query optimizer in CQP runs "alongside" the query engine. The reoptimizer can be patterned after any established query optimizer model: heuristics-based, randomized, dynamic programming, or top-down with memoization could all be used. The prototypical implementations have used a top-down with memoization architecture, combined with full (bushy) enumeration of alternative join trees. It is critical that the optimizer incorporate the revised cost estimates from runtime, as well as information about how much of each relation has been processed: the goal is to extrapolate the cost of executing the *remainder* of the query.

An additional cost penalty is given to changing the plan versus leaving it the same, because keeping the same plan makes the stitch-up query plan more efficient (since all subexpressions are compatible). This penalty is typically assigned in using a heuristic since the system does not yet know what the cost of the stitch-up plan will be, or even whether there will be additional query execution phases.

Creating Stitch-Up Plans

Once basic query execution has completed, if there were multiple query processing phases, then a stitch-up query plan must combine those subsets of data that had previously been partitioned to different plans. In general, for a join of m relations in n plans, there are $n^m - n$ combinations of subsets that need to be stitched together.

One underlying requirement of corrective query processing is that every plan must "buffer" the source data fed into it at the leaves, so this data can be joined with data in the other plans — this mirrors the requirement of the pipelined hash join, and in fact, since most data integration systems almost exclusively rely on pipelined hash joins, it is often trivially satisfied. Other join forms (nested loops, hybrid hash, and merge) must also be extended to do buffering.

There are several options for when corrective query processing can perform the cross-phase joins required to produce complete answers, including doing them each time a plan is changed. Existing implementations postpone all stitch-up computations to a final *stitch-up phase* at the end, after all prior plans have completed, as outlined in the earlier example.

The stitch-up plan can be computed solely from information stored in the hash tables of the various operators from the previous phases. Rather than rescanning input from the sources, CQP instead scans through the existing hash tables to obtain source tuples. The

query optimizer takes into account all existing state in determining the best query plan to incorporate these hash tables — either for probing against or for scanning as inputs.

For every join operation, it estimates the cost of producing the unavailable intermediate results of the expression, rather than all answers.[1] Next, the optimizer creates an *exclusion list* that specifies which subexpressions have already been computed, can be reused, and should not be recomputed.

Finally, CQP includes a specialized variation of the join operator that efficiently incorporates tuples from multiple existing hash tables and adds lineage information to enable duplicate removal. The *stitch-up join* operator starts with an exclusion list provided by the optimizer (e.g., do not regenerate $R^0 \bowtie P^0$ in the example), plus sets of state structures containing existing results that are to be reused. The stitch-up join iterates over the combinations of existing state structures and decides at a structure-to-structure level (rather than a per-tuple level) whether this combination is in the exclusion list or needs to be generated. Moreover, it decides on a pairwise basis which state structure should be scanned for tuples and which should be probed against; if necessary for performance, it will rehash one of the structures according to the join key. Finally, the stitch-up join combines data from its inputs with that from the existing state structures, checking on a per-tuple basis whether the tuple should be created. The final result is an operator that is much more efficient at producing precisely the results needed.

Example 8.6

Refer to Figure 8.10. The stitch-up join (**StJoin**) operators reuse data from the hash tables of the prior phases, while avoiding the creation of duplicate tuples. The lower-level **StJoin** needs to produce any "missing" tuples for $M \bowtie P$. These include $M^0 \bowtie P^0$, $M^1 \bowtie P^0$, and $M^0 \bowtie P^1$, but not $M^1 \bowtie P^1$. To do this, we take the hash tables for M^0, M^1, P^0, P^1 and "attach" them to the stitch-up join. The stitch-up join operator reads from P^0 and joins against both M^0 and M^1; it reads from M^0 and joins against P^1.

The second stitch-up join performs similarly. It has existing $M^1 \bowtie P^1$ tuples in its hash table and reads the remaining $M \bowtie P$ tuples from the child expression. It also reads the R^0 tuples from a hash table created in the initial phase. The second join performs a join of all combinations of tuples except those already output.

To summarize the key differences in performance-driven adaptivity, corrective query processing takes a more conservative, cost-based approach to modifying query plans, which impacts query execution cost minimally when the plan is relatively good. Eddies take a dataflow-based approach to determining query operator ordering, and they

[1] Because it only considers intermediate results that are part of our current query, and since such results are only a subset of all required data, this problem is somewhat simpler than that for optimizing using materialized views, as discussed in Chapter 2.

constantly explore alternative plans as a result. The CQP approach is perhaps more suited to situations in which selectivities are stable throughout execution but initially unknown, whereas eddies are perhaps more suited to situations in which the selectivities are changing frequently.

Bibliographic Notes

Two excellent surveys of conventional query optimization appear as [121, 318]. These provide details on how plan enumeration is performed, various heuristics for pruning the search space, and the use of histograms and other techniques to estimate cardinalities. A complementary survey on query execution appears in [263], which describes hash, index, and iteration-based schemes for the basic operators.

Distributed databases were explored even in the early days of the relational DBMS field, with projects such as Distributed Ingres [530] and R* [405]. One of the first federated databases, Multi-Base, also dates back to the same time period [525]. Perhaps the most ambitious distributed (and federated) DBMS was Mariposa [532], which attempted to use simple techniques from economics to determine how to place data and where to do computation. Some of these ideas have subsequently been refined in later systems, including data integration systems.

The Bloom filter actually is named after its inventor [90]. Both the Bloomjoin and 2-way semijoin were among the earliest query execution algorithms discussed in the distributed query processing literature [73, 161, 404]. For a more comprehensive overview of distributed query processing, refer to the survey in [350] and the textbook [471]. Recent work has even considered mechanisms for doing distributed processing of correlated subqueries [328] and recursion [395].

Much of the work done on query optimization with sophisticated wrapper cost models was pioneered by IBM Almaden Research Center's Garlic system [281, 498]. Garlic was a data integration system built over the Starburst engine (the research version of DB2 UDB). A major focus of the effort was to extend the optimizer to handle heterogeneous (including nonrelational) data sources. In contrast to most of our discussion in this chapter, Garlic was focused on enterprise information integration (EII) tasks within the corporate intranet. Today, many aspects of Garlic are implemented in IBM's InfoSphere series of products.

Adaptive query processing has been the subject of intense study in the conventional database, data stream, and data integration communities. Two surveys on the topic are [48, 172]. Foundational pieces of work discussed in detail in the surveys include techniques to interleave reoptimization and execution in a conventional database context [150, 336, 556], as well as techniques for building more robust query plans [46, 138, 319]. Eddies, including their STeM and STAIR variations, are presented in [44, 171, 490]. Eddies have even been extended to a distributed context [547]. Corrective query processing was presented in [326], and a closely related technique called CAPE appears in [499]. Within the

stream query processing context, adaptive techniques have been developed for load shedding [47, 543, 552] and minimizing memory usage [45, 50]. Strategies for scheduling and adaptive reordering of windowed operators were developed in [49].

Estimating latencies across the Internet has become less challenging over time as network bandwidth has increased, thus reducing latency due to router contention. The Internet measurement community has done significant work on attempting to predict latencies, and also on instrumenting the Internet with measurement infrastructure [232].