Chapter 2

Virtual Data Integration

This chapter discusses the challenges involved in building a data integration system in more concrete terms. To do this, we describe a generic architecture for a data integration system, which we refer to as the *virtual data integration architecture*. We refer to it as virtual because the data remains at the data sources and is only accessed at query time. In the next chapters, we consider each component of the virtual integration architecture and a few variations on it. In order to describe this architecture, we first need to review some basic terms in data management that we use throughout the book.

2.1 Terminology

We first review a few basic terms related to data modeling and querying.

2.1.1 Data model

Data integration systems need to handle data in a variety of data models, be it relational, XML or unstructured data. We review the basics of the relational data model here, and introduce other data models later in the book. In particular, Chapter 11 discusses XML and its underlying data model, as it plays a key role in many data integration scenarios.

A relational database is a set of *relations*, also called *tables* (see Figure 2.1.1). A *database schema* includes a *relational schema* for each of its tables and a set of *integrity constraints* that we describe a bit later.

A *relational schema* specifies the set of attributes in the table and a data type for each attribute. The *arity* of a relation is the number of attributes it has. For

Interview

candidate	date	recruiter	hireDecision	grade
Alan Jones	5/4/2006	Annette Young	No	2.8
Amanda Lucky	8/8/2008	Bob Young	Yes	3.7

EmployeePerformance

empID	name	reviewQuarter	grade	reviewer
2335	Amanda Lucky	1/2007	3.5	Eric Brown
5443	Theodore Lanky	2/2007	3.2	Bob Jones

example, in Figure 2.1.1, the arity of the **Interview** relation is 5 and its schema is:

candidate: string date: date
recruiter: string hireDecision: boolean
grade: float

In a sense, the schema describes how the database author decided the organize the data, what aspects of the data she chose to model, and the distinctions she wished to make. For example, the `Interview` table does not include an attribute for the location of the interview or the position for which the candidate was interviewing. The `EmployeePerformance` table only provides a single grade, and does not model the fact that this grade may be the composition of several more specific performance measures. One of the challenges we face in data integration is that different sources organize their data differently, and these differences need to be reconciled.

A relational table includes a finite number of rows, called *tuples* (or *records*). A tuple assigns a value to each attribute of the table. If the relation being discussed is clear from the context we denote its tuples with its values in parentheses, e.g.,

(Alan Jones, 5/4/2006, Annette Young, No, 2.8).

If not, we denote it as a *ground atom*:

`Interview`(Alan Jones, 5/4/2006, Annette Young, No, 2.8).

In some cases, we describe a tuple with a mapping from attribute names to values, such as:

{candidate→ Alan Jones, date → 5/4/2006, recruiter→ Annette Young,
hireDecision→ No, grade → 2.8}.

We pay special attention to the NULL value in a database. Intuitively, NULL means that the value is not known or may not exist. For example, the value of an age attribute may be NULL if it's not known, whereas the value of spouse could be either unknown or may not exist. The important property to keep in mind about the NULL value is that an equality test involving a NULL returns NULL. In fact, even $NULL = NULL$ returns NULL. This makes intuitive sense because if two values are not known, we certainly do not know that they are equal to each other. We can test explicitly for NULL with the predicate `is NULL`.

A *state* of the database, or *database instance*, is a particular snapshot of the contents of the database. We distinguish between *set* semantics of databases and *multi-set* semantics. In set semantics, a database state assigns a set of tuples to each relation in the database. That is, a tuple can only appear once in a relation. In multi-set semantics, a tuple can appear any number of times in each relation, and hence a database instance is an assignment of a multi-set of tuples to each relation. Unless we state otherwise, our discussion will assume set semantics. Commercial relational databases support both semantics.

We use these notations throughout the book. We denote

- a database instance by D (possibly with subscripts),
- attributes with letters from the beginning of the alphabet, e.g., A, B, C. We denote sets or lists of attributes with overbars, e.g., $\bar{A}$,
- relation names with the letters R, S and T and sets or lists of relations with overbars, e.g., $\bar{R}$.
- tuples with the lowercase letters, e.g., s, t.
- if $\bar{A}$ is a set of attributes and t is a tuple in a relation that has the attributes in $\bar{A}$, then $t^{\bar{A}}$ denotes the restriction of the tuple t to the attributes in $\bar{A}$.

2.1.2 Integrity constraints

Integrity constraints are a mechanism for limiting the possible states of the database. For example, in the employee database, we do not want two rows for the same employee. An integrity constraint would specify that in the employee table the employee ID needs to be unique across the rows. The languages for specifying integrity constraints can get quite involved. Here we focus on the following, most common types of integrity constraints

- **Key constraints:** a set of attributes $\bar{A}$ of a relation R is said to be a key of R if there do not exist a pair of tuples $t_1, t_2 \in R$ such that $t_1^{\bar{A}} = t_2^{\bar{A}}$ and $t_1 \neq t_2$. For example, the attribute candidate can be a key of the **Interview** table.

- **Functional dependencies:** We say that a set of attributes $\bar{A}$ functionally determines a set of attributes $\bar{B}$ in a relation R if for every pair of tuples $t_1, t_2 \in R$, if $t_1^{\bar{A}} = t_2^{\bar{A}}$, then $t_1^{\bar{B}} = t_2^{\bar{B}}$.

  For example, in the **EmployeePerformance** table, emplD and reviewQuarter functionally determine grade. Note that a key constraint is simply a functional dependency where the key attributes determine all of the other attributes in the relation.

- **Foreign key constraints:** let S be a relation with attribute A, and let T be a relation whose key is the attribute B. The attribute A is said to be a foreign key of attribute B of table T if whenever there is a row in S where the attribute A has the value v, then there must exist a row in T where the value of the attribute B is v. For example, the emplD attribute of the **EmployeePerformance** table is a foreign key of the `Employee` table.

2.1.3 Queries and answers

Queries are used for several purposes in data integration systems. As in database systems, queries are used in order to formulate users' information needs. In some cases, we may want to reuse the query expression in other queries, in which case we define a named *view* over the database, defined by the query. If we want the database system to compute the answer to the view and maintain the answer as the database changes, we refer to it as a *materialized view*.

In data integration systems, we also use queries to specify *relationships* between the schemata of data sources. In fact, as we discuss in Chapter 5, queries form the core of the formalisms for specifying semantic mappings.

We distinguish between *structured* queries and *unstructured* queries. Structured queries such as SQL queries over relational databases or XQuery queries over XML databases, are the ones we work hard to support in database systems. Unstructured queries are the ones we're most familiar with on the Web: the most common form of an unstructured query is list of keywords.

We use two different notations for queries over relational databases throughout the book. The first is SQL, which is the language used to query relational data in commercial relational systems. Unfortunately, SQL is not known for its aesthetic aspects and hence not convenient for more formal expositions. Hence, in some of the

more formal discussions, we use the notation of *conjunctive queries* which is based on (a very simple form of) Mathematical Logic.

SQL is a very complex language. For our discussion, we typically use only its most basic features: selecting specific rows from a table, selecting specific columns from a table, combining data from multiple tables using the join operator, taking the union of two tables, and basic aggregation functions. For example, the following queries are typical of the ones we see in the book.

Example 2.1:

SELECT recruiter, candidate
FROM **Interview**, **EmployeePerformance**
WHERE recruiter = name AND **EmployeePerformance.**grade < 2.5

Example 2.2:

SELECT reviewer, Avg(grade)
FROM **EmployeePerformance**
WHERE reviewQuarter ="1/2007"

The query in Example 2.1 asks for pairs of recruiters and candidates where the recruiter got a low grade on their performance review. The answer to this query may reveal candidates who we may want to re-interview. The query in Example 2.2 asks for the average grade that a reviewer gave in the first quarter of 2007.

Given a query Q and a database D, we denote by $Q(D)$ the result of applying the query Q to the database D. Recall that $Q(D)$ is also a relation whose schema is defined implicitly by the query expression.

2.1.4 Conjunctive queries

We briefly review the formalism for conjunctive queries. A conjunctive query has the following form:

$$Q(\bar{X}) \text{:-} R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), c_1, \ldots, c_m$$

In the query, $R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n)$ are the *subgoals* (or *conjuncts*) of the query and together form the *body* of the query. The R_i's are database relations, and the $\bar{X}$'s are tuples of variables and constants. Note that the same database relation can occur in multiple subgoals. Unless we explicitly give the query a different name, we refer to it as Q.

The variables in $\bar{X}$ are called *distinguished variables*, or *head variables*, and the others are *existential variables*. The predicate Q denotes the answer relation of the query. Its arity is the number of elements in $\bar{X}$. We denote by $Vars(Q)$ the set of variables that appear in its head or body.

The c_j's are *interpreted atoms*, and are of the form $X\theta Y$, where X and Y are either variables or constants, and at least one of X or Y is a variable. The operator θ is an interpreted predicate such as $=$, $\leq$, $<$, $!=$, $>$ or $\geq$. We assume the obvious meaning for the interpreted predicates, and unless otherwise stated, we interpreted them over a dense domain.[1]

The semantics of a conjunctive query Q over a database instance D is as follows. Consider any mapping ψ that maps each of the variables in Q to constants in D. Denote by $\psi(R_i)$ the result of applying ψ to $R_i(\bar{X}_i)$, $\psi(c_i)$ the result of applying ψ to c_i, and by $\psi(Q)$ the result of applying ψ to $Q(\bar{X})$, all resulting in ground atoms. If

- each of $\psi(R_1), \ldots, \psi(R_n)$ is in D, and
- for each $1 \leq j \leq m$, $\psi(c_j)$ is satisfied,

then, $\psi(Q)$ is in the answer to Q over D.

To illustrate the correspondence between SQL queries and conjunctive queries, the following conjunctive query is the same as the SQL query in Example 2.1.

Q_1(Y,X) :- `Interview`(X,D,Y,H,F), `EmployeePerformance`(E,Y,T,W,Z), W $\leq$ 2.5.

Note that the join in the conjunctive query is expressed by the fact that the variable Y appears in both subgoals. The predicate on the grade is expressed with an interpreted atom.

Conjunctive queries must be be *safe*; that is, every variable appearing in the head also appears in the body. Otherwise, the set of possible answers to the query may be infinite (i.e., the variable appearing in the head but not in the body can be bound to any value).

We can also express disjunctive queries in this notation. To express disjunction, we write two (or more) conjunctive queries with the same head predicate.

Example 2.3: The following query asks for the recruiters who performed the best *or* the worst:

[1]The alternative would be to interpret them over a discrete domain such as the integers. In that case we need to account for subtle inferences such as implying $X = 4$ from the conjunction $X > 3, X < 5$.

Q_1(E,Y) :- `Interview`(X,D,Y,H,F), `EmployeePerformance`(E,Y,T,W,Z), W $\leq$ 2.5.
Q_1(E,Y) :- `Interview`(X,D,Y,H,F), `EmployeePerformance`(E,Y,T,W,Z), W $\geq$ 3.9.

We also consider conjunctive queries with negated subgoals, of the form:

$$Q(\bar{X}) \text{:-} R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n), \neg S_1(\bar{Y}_1), \ldots, \neg S_m(\bar{Y}_m)$$

For queries with negation, we extend the notion of safety as follows: any variable appearing in the head of the query must also appear in a *positive* subgoal. To produce an answer for the query, the mapping from the variables of Q to the constants in the database must satisfy $\psi(S_1(\bar{Y}_1)), \ldots, \psi(S_m(\bar{Y}_m)) \notin D$.

In our discussions, the term *conjunctive queries* will refer to conjunctive queries *without* interpreted predicates or negation. If we allow other types of atoms, we will explicitly say so.

2.1.5 Datalog programs

A datalog program is a set of rules, each of which is a conjunctive query. Instead of computing a single answer relation, a datalog program computes a set of *intensional* relations (called IDB relations), one of them being designated as the *query predicate*. In datalog, we refer to the database relations as the *EDB relations* (extensional database). Intuitively, the extensional relations are given as a set of tuples (also referred to as ground facts), while the intensional relations are defined by a set of rules. Consequently, the EDB relations can occur only the body of the rules whereas the IDB relations can occur both in the head and in the body.

Example 2.4: Consider a database that includes a simple binary relation representing the edges in a graph: edge(X,Y) holds if there is an edge from X to Y in the graph. The following datalog query computes the paths in the graph. edge is an EDB relation and path is an IDB relation.

r_1 : path(X,Y) :- edge(X,Y)
r_2 : path(X,Y) :- edge(X,Z), path(Z,Y)

The first rule states that all single edges form paths. The second rule computes paths that are composed from shorter ones. The query predicate in this example is path. Note that replacing r_2 by the following rule would produce the same result.

r_3 : path(X,Y) :- path(X,Z), path(Z,Y)

□

The semantics of datalog programs are based on conjunctive queries. We begin with empty extensions for the IDB predicates. We choose a rule in the program and apply it to the current extension of the EDB and IDB relations. We add the tuples computed for the head of the rule to its extension. We continue applying the rules of the program until no new tuples are computed for the IDB relations. The answer to the query is the extension of the query predicate. When the rules do not contain negated subgoals, this process is guaranteed to terminate with a unique answer, independent of the order in which we applied the rules.

Example 2.5: In Example 2.4, suppose we begin with the databases that contains the tuples edge(1,2), edge(2,3), edge(3,4). When we apply r_1 we will obtain path(1,2), path(2,3), and path(3,4). The first time we apply r_2 we obtain path(1,3) and path(2,4). The second time we apply r_2 we obtain path(1,4). Since no new tuples can be derived, the evaluation of the Datalog program terminates. □

In data integration we are interested in datalog programs mostly because they are sometimes needed in order to compute all the answers to a query from a set of data sources (see Sections 5.3 and 5.4). Readers who are familiar with the Prolog programming language will notice that datalog is a subset of Prolog. The reader should also note that not all SQL queries can be expressed in datalog. In particular, there is no support in datalog for grouping and aggregation and for outer-joins. SQL does support limited kinds of recursion but not arbitrary recursion.

As we noted earlier, every query language has its aesthetic limitations. We will not use conjunctive queries or Datalog to express queries with grouping and aggregation except in very limited cases.

2.2 Virtual Data Integration Architecture

We now turn to discussing the elements of the standard virtual data integration architecture. Figure 2.1 shows the logical components of a virtual data integration system. Figure 2.2 shows an example data integration scenario using this architecture.

2.2.1 Components of the data integration system

On the bottom of the figures we see the *data sources*. Data sources can vary on many dimensions, such as the data model underlying them and the ability they have

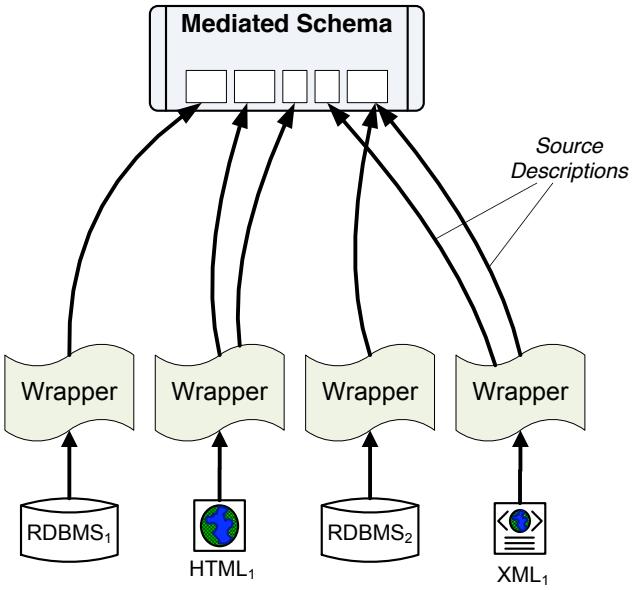

Figure 2.1: Logical components of a virtual data integration system. The data sources can be relational, XML or any store that contains structured data. The wrappers translate queries from the data integration system to the format of the sources and parse the results. The mediated schema, which is used to pose queries to the data integration system, includes the aspects of the domain that are relevant to the data integration scenario. The semantic mappings specify the correspondences between the terms used in the source schemas and the terms of the mediated schema.

to answer queries. Examples of structured sources include database systems with SQL capabilities, XML databases with an XQuery interface, and sources behind web forms that support a limited set of queries (corresponding to the valid combinations of inputs to its fields). In some cases, the source can be an actual application that is driven by a database, such as an accounting system. In such a case, a query to the data source may actually involve an application processing some data stored in the source.

The *wrappers* are (hopefully small) programs whose role is to send queries to a data source, receive answers and possibly apply some basic transformations on the answer. For example, a wrapper to a web form source would accept a query and translate it into the appropriate HTTP request with a URL that poses the query on the source. When the answer comes back in the form of an HTML file, the wrapper would extract the tuples from the HTML file.

On the top of the figure we have the *mediated schema*. The mediated schema is built for the data integration application and contains *only* the aspects of the domain

that are relevant to the application. As such, it does not necessarily contain all the attributes we see in the sources, but only a subset of them. The mediated schema is not meant to store any data. It is purely a logical schema that is used for posing queries by the users (or applications) employing the data integration system.

The key to building a data integration application are the *source descriptions*. These descriptions specify the properties of the sources that the system needs to know in order to use their data. The main component of source descriptions are the *semantic mappings*, that relate the schemata of the data sources to the mediated schema. The semantic mappings specify how attributes in the sources correspond to attributes in the mediated schema (when such correspondences exist), and how the different groupings of attributes into tables are resolved. In addition, the semantic mappings specify how to resolve differences in how data values are specified in different sources. It is important to emphasize that the virtual data integration architecture only requires specifying mappings between the data sources and the mediated schema and not between every pair of data sources. Hence, the number of mappings we specify is the same as the number of sources and not the square of that number. Furthermore, the semantic mappings are specified *declaratively*, which enables the data integration system to reason about the contents of the data sources and their relevance to a given query and optimize the query execution.

Example 2.6: Let us consider each of these components on the example shown in Figure 2.2. In the example, we have five data sources. The first one on the left, S1, stores data about movies, including their names, actors, director and genre. The next three sources, S2-S4, store data about show times. Source S2 covers the entire country, while S3 and S4 consider only cinemas in New York City and San Francisco, respectively. Note that although these three sources store the same type of data, they use different attribute names. The rightmost source, S5, stores reviews about movies.

The mediated schema includes four relations, **Movie**, **Actors**, **Plays** and **Reviews**. Note that the **Review** in the mediated schema does not contain the date attribute, but the source storing reviews does contain it.

The semantic mappings in the source descriptions describe the relationship between the sources and the mediated schema. For example, the mapping of source S1 will state that it contains movies, and that the attribute name in **Movies** maps to the attribute Title in the **Movie** relation of the mediated schema. It will also specify that the **Actors** relation in the mediated schema is a *projection* of the **Movies** source on the attributes name and actors.

Similarly, the mappings will specify that tuples of the **Plays** relation in the mediated schema can be found in either S2, S3 or S4, and that the tuples in S3 have their location city set to New York (and similarly for SF and S4).

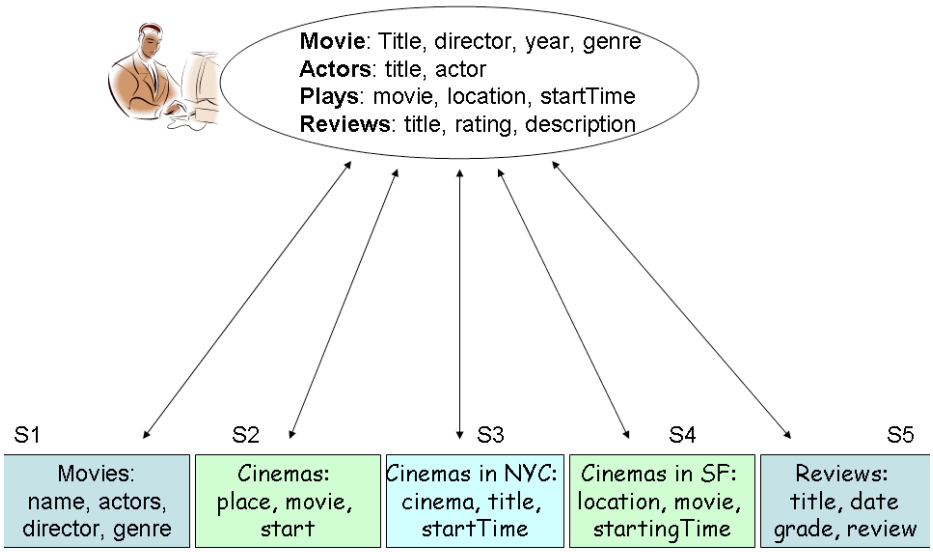

Figure 2.2: An example data integration scenario. Queries about movies are answered using a combination of sources on movie details, cinema listings and review databases.

In addition to the semantic mappings, the source descriptions also describe other aspects of the data sources. First, they specify whether the sources are *complete* or not. For example, source S2 may not contain all the movie showing times in the entire country, while source S3 may be known to contain *all* movie showing times in New York. Second, the source descriptions can specify limited access patterns to the sources. For example, the description of S1 may specify that in order to get an answer from the source, there needs to be an input for at least one of its attributes. Similarly, all the playing times sources also require a movie title as input.

2.2.2 Query processing

Now we consider how query processing proceeds in the virtual data integration architecture, mirroring the different components shown in Figure 2.3.

Query reformulation

As described earlier, the user query is posed in terms of the relations in the mediated schema. Hence, the first step the system must do is *reformulate* the query into queries that refer to the schemas of the data sources. To do so, the system uses the source descriptions. The result of the reformulation is a set of queries that refer to the schemata of the data sources and whose combination will yield the answer to the original query. We refer to the result of reformulation as a *logical query plan*.

In our example, suppose we are searching for show times of movies playing in New York and directed by Woody Allen. The query is posed over the mediated schema as follows.

```
SELECT title, startTime
FROM Movie, Plays
WHERE Movie.title = Plays.movie AND
      location="New York" AND
      director="Woody Allen"
```

The following would be derived during reformulation:

- tuples for **Movie** can be obtained from source S1, but the attribute title needs to be reformulated to name.
- Tuples for **Plays** can be obtained from either source S2 or S3. Since the latter is complete for showings in NYC, we choose it over S2.
- Since source S3 requires the title of a movie as input, and such a title is not specified in the query, the query plan must first access source S1 and then feed the movie titles returned from S1 as inputs to S3.

Hence, the first logical query plan generated by the reformulation engine accesses S1 and S3 to answer the query. However, there is a second logical query plan that is also correct (albeit possibly not complete) and that plan accesses S1 followed by S2.

Query optimization

The next step in query processing is query optimization, as in traditional database systems. Query optimization takes as input a logical query plan and produces a *physical query plan*, which specifies the exact order in which the data sources are accessed, when results are combined, which algorithms are used for performing operations on the data (e.g., join between sources) and the amount of resources allotted to each

operation. As described earlier, the system must also handle the challenges that arise from the distributed nature of the data integration system.

In our example, the optimizer will decide which join algorithm to use to combine results from S1 and S3. For example, the join algorithm may stream movie titles arriving from S1 and input them into S3, or it may batch them up before sending them to S3.

Query execution

Finally, the execution engine is responsible for the actual execution of the physical query plan. The execution engine dispatches the queries to the individual sources through the wrappers and combines the results as specified by the query plan.

Herein lies another significant difference between a data integration system and a traditional database system. Unlike a traditional execution engine that merely executes the query plan given to it by the optimizer, the execution engine of a data integration system may decide to ask the optimizer to reconsider its plan based on its monitoring of the plan's progress. In our example, the execution engine may observe that source S3 is unusually slow and therefore may ask the optimizer to generate a plan that includes an alternate source.

Of course, an alternative would be for the optimizer to already build certain contingencies into the original plan. However, if the number of unexpected execution events is large, the original plan could grow to an enormous size. Hence, one of the interesting technical challenges in designing the query processing engine is how to balance between the complexity of the plan and its ability to respond to unexpected execution events.

2.3 Outline of the Book

The remainder of this book will elaborate on each of the components and processes described above. The following outlines some of the main topics covered by each chapter.

To help motivate the material in the book, we begin in Chapter 3 with an overview of the common ways data integration is done in the enterprise world today when *not* using the declarative techniques espoused in this book. We discuss data warehouses and ETL, along with their limitations. We describe how "NoSQL" systems have been used to bring together data from external files, and how Web mashups can be used to construct integrated Web sites.

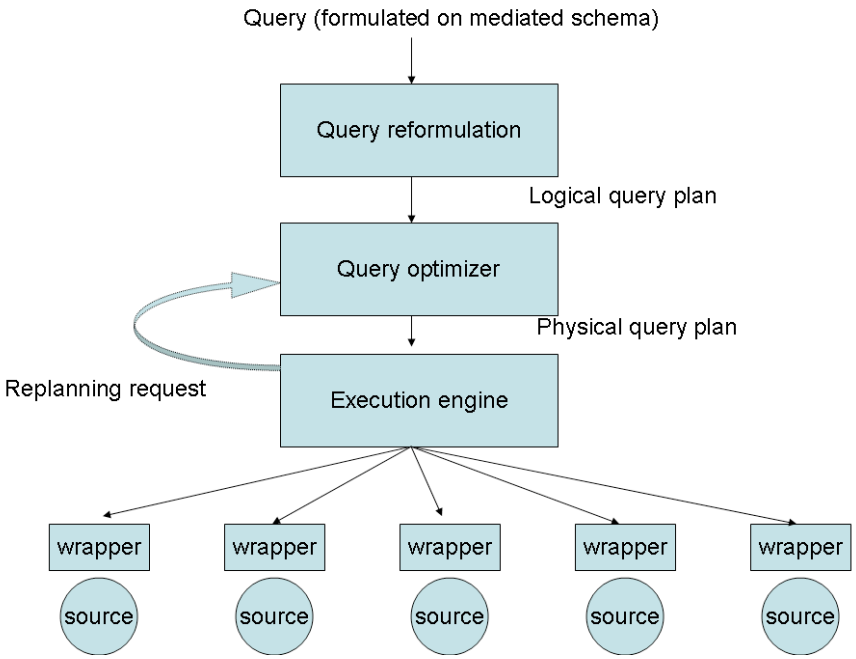

Figure 2.3: Query processing in a data integration system differs from traditional database query processing in two main ways. First, the query needs to be reformulated from the mediated schema to the schema of the sources. Second, query execution may be adaptive in that the query execution plan may change as the query is being executed.

Chapter 4 lays a few theoretical foundations that are needed for later discussions. In particular, the chapter describes algorithms for manipulating and reasoning about query expressions. These algorithms enable us to determine whether one query is *equivalent* to a second query even if they are written differently, and to determine whether a query can be answered from previously computed views on the database. These algorithms will be essential for query reformulation and optimization in data integration, but are also useful in other data management contexts.

Chapter 5 describes the formalisms proposed for specifying source descriptions and in particular the semantic mappings. We describe two languages that use query expressions for specifying semantic mappings: Global-as-View (GAV) and Local-as-View (LAV), and the GLAV language that combines the two. For each of these languages, we describe the appropriate query reformulation algorithms. We also describe how we can handle information about source completeness and about limitations on accessing data in sources.

In Chapter 6 we begin our discussion of techniques for *creating* semantic mappings. As it turns out, creating mappings is one of the main bottlenecks in building data integration applications. Hence, our focus is on techniques that reduce the time required from a person to create mappings. Chapter 6 discusses the fundamental problem of determining whether two strings refer to the same real-world entity. String matching plays a key role in matching data and schema from multiple sources. We describe several heuristics for string matching and methods for scaling up these heuristics to large collections of data. We then discuss the problems of creating mappings at the schema level (Chapter 7), and techniques for creating mappings at the data level (Chapter 8). One of the interesting aspects of these techniques is that they leverage methods from Machine Learning, enabling the system to improve over time as it sees more correct schema and object matchings.

Chapter 9 discusses query processing in data integration systems. The chapter describes how queries are optimized in data integration systems and special query operators that are especially appropriate for this context. The important new concept covered in this chapter is *adaptive query processing*, which refers to the ability of a query processor to change its plan during execution.

Chapter 10 then discusses how Web information extractors or *wrappers* are constructed, in order to acquire the information to be integrated. Wrapper construction is an extremely challenging problem, especially given the idiosyncrasies of real-world HTML. It typically requires a combination of heuristic pattern matching, machine learning, and user interaction.

Part II of the book focuses on richer ways of modeling the data, incorporating hierarchy, class relationships, and annotations. Chapter 11 discusses the role of XML in data integration. XML has played an important role in data integration because it provides a syntax for sharing data. Once the syntax problem was addressed, people's appetites for sharing data in semantically meaningful ways were whetted. This chapter begins by covering the XML data model and query language (XQuery). We then cover the techniques that need to be developed to support query processing and schema mapping in the presence of XML data.

Chapter 12 discusses the role of knowledge representation (KR) in data integration systems. Knowledge Representation is a branch of Artificial Intelligence that develops languages for representing data. These languages enable representing more sophisticated constraints than are possible in models employed by database systems. The additional expressive power enables KR systems to perform sophisticated reasoning on data. Knowledge Representation languages have been a major force behind the development of the *Semantic Web*, a set of techniques whose goal is to enrich data on the web with more semantics in order to ultimately support more complex queries and reasoning. We cover some of the basic formalisms underlying the Semantic Web

and some of the challenges this effort is facing.

Next we describe two ways that data can be annotated. Chapter 13 discusses how to incorporate uncertainty into data integration systems. When data is integrated from multiple autonomous sources, the data may not all be correct or up to date. Furthermore, the schema mappings and the queries may also be approximate. Hence, it is important that a data integration system be able to incorporate these types of uncertainty gracefully. Then Chapter 14 describes another form of annotation: data provenance "explains," for every tuple, how the tuple was obtained or derived. We describe the close relationship between provenance and probabilistic scores.

Finally, Part III discusses some new application contexts in which data integration is being used and the challenges that need to be addressed in them. Chapter 15 discusses the kinds of structured data that are present on the Web and the data integration opportunities that arise there.

Chapter 17 reviews *data warehousing* from Chapter 3: the main advantage of data warehousing is that it can support complex queries more efficiently. We then discuss *data exchange*, an architecture in which we have a *source* database and our goal is to translate the data and store it in a *target* database that has a different schema, and answer queries over the target database.

Chapter 18 describes a peer-to-peer architecture (P2P) for data sharing and integration. In the P2P architecture, we don't have a single mediated schema, but rather a loose collection of collaborating peers. In order to join a P2P data integration system a source would provide semantic mappings to *some* peer already in the system, the one for which it is most convenient. The main advantage of the P2P architecture is that it frees the peers from having to agree on a mediated schema in order to collaborate.

Chapter 19 describes how ideas from Web-based data integration and P2P data sharing have been further extended to support collaborative exchange of data. Key capabilities in collaborative systems include the ability to make annotations and updates to shared views of data.

Finally, Chapter 20 discusses *model management*, which offers an algebra for manipulating schemas and mappings between schemas. Model management represents a formalization of many of the aspects of data transformation and integration described in the book.

Exercises

1. Rewrite the following SQL query as a conjunctive query using the syntax of Section 2.1.4:

```
SELECT candidate, grade, hireDecision
FROM Interview, EmployeePerformance
WHERE recruiter = 'Annette Young'
```

2. Describe the difference in roles between the *wrapper* and *source description* in converting data into a form suitable for a mediated schema.

3. Beyond relational and HTML data, give three examples of data source formats.

Bibliographic Notes

This chapter offered only a very cursory introduction to the relational data model and query languages. We refer the reader to standard database textbooks for further reading on the topic [194, 382]. For a more theoretical treatment of query languages and integrity constraints, we refer the reader to [10] and for a comprehensive treatment of datalog, see [428].