10 Data Warehousing and Caching

Thus far, we have discussed the key approaches to defining schema mappings, composing queries with schema mappings (reformulation) to get queries posed directly over data sources, and query processing over remote data. These basic techniques enable us to fetch data on demand in response to user information needs, using the most up-to-date state available. However, in a variety of circumstances this *fully virtual* data integration architecture may not be ideal: we may be willing to tolerate slightly older data in exchange for better performance, better data quality, or the ability to express more complex queries or perform more sophisticated data transformations. The basic idea is to exploit *materialization* — i.e., precomputed or cached results from the integration process — to meet these goals.

In this chapter we describe various methods for storing and exploiting locally materialized data in order to answer data integrating queries. We start with an overview of the two main approaches to materialization and their respective roles. We then describe the use of *partial* materialization and caching to improve performance. Finally, we discuss an emerging "hybrid" of virtual integration and warehousing, namely, direct querying of externally materialized flat-file data, as in many Web settings today.

There are many applications where the main goal of data integration is to bring all of the data into a single, centralized system where it can be analyzed extensively. As a first example, consider a retailer that is recording each of its transactions (i.e., each purchase of an item at one of its stores) in a set of databases. These databases need to be optimized to support high throughput of transactions, effectively reducing your waiting time at the cashier — and therefore saving you from buying candy that you don't need. In addition to recording the transactions, the retailer would also like to perform deeper analysis of their sales data. For example, they would like to know whether raincoats are selling in higher numbers in Kansas so they can make sure to stock up their stores appropriately. To support such analysis, the retailer creates a separate data store, called a warehouse, that contains data that is cleaned up, aggregated, and in an appropriate form for posing complex decision-support queries. A data warehouse is typically populated not using declarative schema mappings as we have described in this book, but rather through pipelines of procedural ETL (extract/transform/load) tools. In the first phase the system extracts the data from the original sources, be they legacy systems or database systems, and converts them into a format that can be processed further. In the second phase, the system transforms the data into a form appropriate for the data warehouse. This may involve selecting a subset of rows or columns, joining data across tables, aggregating values or applying transformations to data values, applying data cleaning or normalization operations, and so on. In

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00010-7

the third phase, the data are loaded into the warehouse, either replacing existing data or adding to it. We describe data warehouses in Section 10.1.

However, many (though not all) ETL operations can in fact be expressed through declarative means, much as in virtual data integration systems. Consider a set of scientists, each of whom is collecting data about specimens in the field, and each of whom wants to share data with the others. It is common for scientists to store their data in spreadsheets, and perhaps the frequency of their updates will be relatively low. A valuable data integration system for these scientists would be one in which they can all load their data into one repository and query across the data of all the collaborators. The process of *data exchange* results in a materialized central database, much like the data warehouse, except that the data transformations are specified declaratively as in our prior discussions of virtual data integration. As with data warehousing, the heterogeneity is resolved when the data are *uploaded* into the repository. This approach provides some of the benefits of data warehousing — query processing does not require reformulation, the ability to index the materialized data, and the fact that all data are localized to a single DBMS — while also preserving some of the benefits of virtual integration (namely, declarative mappings that the system can reason about). We describe data exchange in Section 10.2.

Of course, there are situations in which one wants some of the performance benefits of materialized data, while also getting “live” data from some of the sources. We briefly discuss opportunities for “hybrid” approaches to integration, relying on partial materialization and caching, in Section 10.3.

We wrap up the chapter with a discussion of problems such as Web traffic analysis, as is done in any advertisement-based Web platform today. Here, the goal is to bring together the Web traffic logs, particularly user click-throughs on advertisements, for every Web server owned by the service provider. Once the data are in one place, the task is to perform a variety of analytics on click-throughs: grouping by customer, by advertiser campaign, by ad type, and so on. Often the goal here is to *directly* process the Web logs “in situ” without having to first load them into a database, and to do very large-scale parallel processing of the data. The standard tools used in this context are based on Google’s MapReduce framework or Apache’s open-source version, Hadoop MapReduce. We discuss these cloud data processing frameworks and their role in doing integrated data analysis over locally materialized data (in delimited files) in Section 10.4.

10.1 Data Warehousing

The original, and still predominant, approach to information integration in the enterprise setting is through the definition and creation of a centralized database called a *data warehouse* (see Figure 10.1). In this model, all data needed by an organization are translated into a target schema and copied into a single (possibly parallel or distributed) DBMS, which gets periodically refreshed. Contrast that with virtual data integration, where data are requested from the *sources* on demand, or with MapReduce, where data are typically

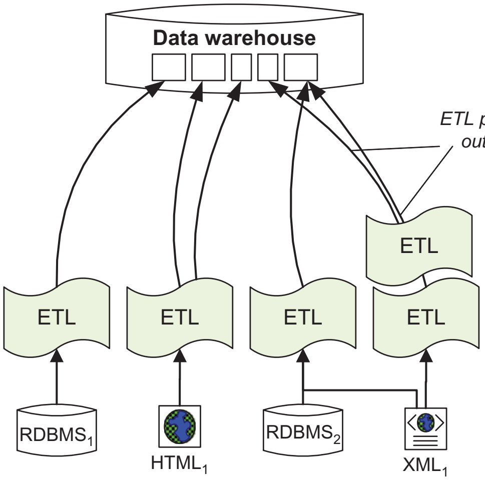

FIGURE 10.1 Logical components of a data warehouse setup. The data are loaded through a pipeline of transformations into a physical data warehouse.

external to the query system and do not support updates or random access. In addition, the transformations that are used to load the data into the warehouse are typically carried out by pipelines of *procedural* code.

Especially in business, a data warehouse serves the natural roles of *archival* and *decision support*. For a variety of reasons, enterprises may need to maintain historical data — for auditing, for analysis, and even for prediction. Simply querying over the existing state of data sources is unlikely to be sufficient — instead, the enterprise needs a master archival copy of the data at different points in time, and the warehouse accomplishes exactly this. More generally, the warehouse, as a consistent "global snapshot" of an enterprise's data with a powerful DBMS, storage system, and CPU, can often be used to perform so-called *decision-support* or *online analytic processing* (OLAP) queries — queries that look at the aggregate characteristics of the data to help form business decisions. For instance, Walmart built a very strong reputation for using sales data to forecast which items, in what quantity, to stock in each store. Such queries typically have multiple levels of aggregation and may involve data mining operations. Similarly, online shopping sites such as Amazon.com, media sites such as Netflix, and even supermarkets such as Safeway all try to build profile information on their customers in order to improve their ability to market to them.

MASTER DATA MANAGEMENT

This central role of the data warehouse has been formalized under the term *master data management* (MDM), which uses a central warehouse as a repository of knowledge about the enterprise's "critical business objects," rules, and processes. Central to MDM is a clean, normalized version of the terms used throughout the enterprise — whether addresses, names, or concepts — and information about the related metadata. Ideally, whenever business objects are used in systems throughout the enterprise, the data values used by these systems can be tied back to the master data. In many ways, a master data repository is merely a data warehouse with a particular role to play.

As with any warehouse, MDM gives the various data owners and stakeholders a bird's-eye view of all of the data entities and a common intermediate representation. However, the master data repository is also intended to be a central repository where relevant properties about data — especially constraints and assumptions — can be captured, making it the home of all *metadata* as well as data. In many cases, the master repository is made queriable to all of the data owners, such that they can directly incorporate it into their systems and processes. It is seen as a way of improving risk management, decision making, and analysis.

Finally, MDM provides a process by which the data can be overseen and managed through *data governance*. In a large enterprise, coordinating the design and evolution of data in accordance with business needs and regulations can be a challenge. Data governance refers to the process and organization put in place to oversee the creation and modification of data entities in a systematic way. Of particular importance are any reporting requirements such as those introduced by the Sarbanes-Oxley Act in the USA, which imposes responsibility and accountability provisions for public corporations that may drive data collection and representation.

Actually defining a data warehouse involves two main tasks: performing central database schema and physical design (Section 10.1.1) and defining a set of extract/transform/load (ETL) operations (Section 10.1.2).

10.1.1 Data Warehouse Design

Designing a data warehouse can be even more involved than designing a mediated schema in a data integration setting because the warehouse must support very demanding queries, possibly over data archived over time. Physical database design becomes critical — effective use of partitioning across multiple machines or multiple disk volumes, creation of indices, definition of materialized views that can be used by the query optimizer. Most data warehouse DBMSs are configured for query-only workloads, as opposed to transaction processing workloads, for performance: this disables most of the (expensive) consistency mechanisms used in a transactional database.

Since the early 2000s, all of the major commercial DBMSs have attempted to simplify the tasks of physical database design for data warehouses. Most tools have "index selection wizards" and "view selection wizards" that take a log of a typical query workload and perform a (usually overnight) search of alternative indices or materialized views, seeking to find the best combination to improve performance. Such tools help, but still there is a need for expert database administrators and "tuners" to obtain the best data warehouse performance.

10.1.2 ETL: Extract/Transform/Load

Once a data warehouse has been designed and configured, obviously it must be initially populated with data and maintained over time. The wide variety of tools used to do this are generically referred to as *ETL*, or *extract/transform/load*, tools. ETL tools address a wide variety of tasks:

IMPORT FILTERS
These include parsers for external file formats, or drivers that interact with third-party systems (whether DBMSs or other applications). Often the external data may not be coming from a relational database, whereas, in almost all cases, a data warehouse is relational.

DATA TRANSFORMATIONS
Data transformation modules typically serve a role very similar to schema mappings in a virtual data integration system: they may join, aggregate, or filter data.

DEDUPLICATION
Deduplication (or record linking) tools seek to determine when multiple records refer to the same entity — often through heuristics. The techniques are often based on the data matching techniques mentioned in Chapter 7.

PROFILING
Data profiling tools typically build up tables, histograms, or other information that summarizes the properties of the data in the warehouse.

QUALITY MANAGEMENT
In addition to deduplication tools, ETL quality management support might include testing against a master list of data values (e.g., a list of legal state/province abbreviations), testing against known business rules (e.g., constraints on combinations of values), standardization tools (e.g., postal address canonicalization), and record merging.

Example 10.1

Consider an e-commerce site scenario: We have a series of invoice line items from customers' purchases, as fulfilled by our warehouse. We wish to bring these into the central data warehouse, while simultaneously filtering for data entry errors.

To do this, we must assemble an ETL pipeline that performs a variety of data splitting, filtering, joining, and grouping operators. See Figure 10.2: the first operator modifies the schema by splitting a single attribute (date/time) into separate date and time attributes. Next, we filter any records with invalid date/time signatures and write them to a log. Next, we join each record with our database of items; again, we filter any invalid entries and write them to a log. Subsequently,

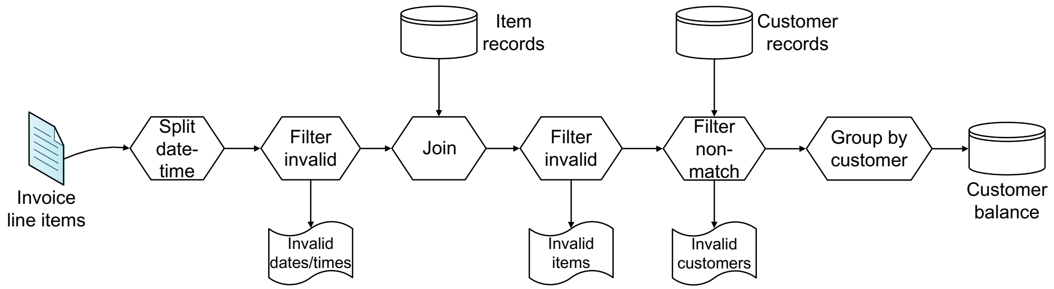

FIGURE 10.2 Example ETL pipeline for importing customer records.

we validate that each record actually corresponds to a valid customer; as has been our pattern, we write any invalid entries to a log. Finally, we group all of the records by customer and use them to update the customer's balance in our data warehouse.

The above example primarily consists of operations that might be captured with declarative schema mappings. However, it should be clear from the preceding list of capabilities that ETL tools can capture functionalities beyond virtual data integration mappings. Tremendous flexibility and expressiveness are provided by the basic architecture, especially since ETL is an offline process that enables long-lived, computationally intensive tasks to be performed. Unfortunately, the flexibility has a drawback, which is that there is very little standardization among ETL tools and approaches. Different vendors' tools have entirely different interfaces and different tools for specifying workflows among the tools.

Moreover, there are cases where one might want to *optimize* the loading process, e.g., by precomputing or caching certain results or sharing work among operators. Most ETL frameworks do not have this flexibility. Hence there has been significant interest in replacing some ETL operations with declarative mappings similar to those we have seen previously in this book. This leads to the problem of computing a data warehouse using declarative mappings, termed *data exchange*, which we discuss next.

10.2 Data Exchange: Declarative Warehousing

As discussed in the previous section, the basic idea of data exchange is to support a warehouse-like setting, with declarative schema mappings specifying the data transformation. Under this model, every data source likely has a wrapper, much as in a conventional data integration scenario, but all data from the sources will be retrieved, transformed according to the schema mappings, and stored in the centralized warehouse or *target data instance* in an offline step. Users directly query the target data instance as in a data warehouse.

In specifying the data-exchange *setting*, we assume that all of the sources have been "wrapped" such that we have access to their data as relations. Moreover, we will consider *all* of these source relations to be part of a single source schema S, and all of the warehouse relations to be part of a target schema T.

Informally, given S and T, an instance, I, of schema S, and a semantic mapping between S and T, the problem of data exchange is to create an appropriate instance of T. In our context, S is the schema of a data source, and T is the schema of the mediated schema of the materialized data store. The challenge in data exchange arises because there are many possible instances of T that we may consider for a given instance, I, of S, but we want the ones that satisfy certain desirable properties. We formalize the problem below.

10.2.1 Data-Exchange Settings

In the discussion of data exchange, we follow the notation of *tuple-generating dependencies*, which were briefly introduced in Chapter 3 and are commonly used in the literature. Recall that tuple-generating dependencies are an alternative formalism for expressing GLAV mappings. Tuple-generating dependencies are formulas of the form

$$(\forall \bar{X}) s_1(\bar{X}_1), \ldots, s_m(\bar{X}_m) \rightarrow (\exists \bar{Y}) t_1(\bar{Y}_1), \ldots, t_l(\bar{Y}_l)$$

where $s_1, \ldots, s_m$ are relations in the source schema S, and $t_1, \ldots, t_l$ are relations in the target schema T. The variables $\bar{X}$ are a subset of $\bar{X}_1 \ldots \cup \ldots \bar{X}_m$, and the variables $\bar{Y}$ are a subset of $\bar{Y}_1 \ldots \cup \ldots \bar{Y}_l$. The variables $\bar{Y}$ do not appear in $\bar{X}_1 \ldots \cup \ldots \bar{X}_m$.

While we focus mostly on data-exchange settings that only involve schema mappings, in some settings we also want to ensure that the target instance satisfies certain constraints on the target schema T. To model such constraints, we introduce *target constraints* into data-exchange settings. Formally, we assume we also have a set of constraints C_T that can be of two forms:

1. Tuple-generating dependencies where the s_i's and the t_j's are relations in T, or
2. Equality-generating dependencies that are formulas of the form

$$(\exists \bar{Y}) t_1(\bar{Y}_1), \ldots, t_l(\bar{Y}_l) \rightarrow (Y_i = Y_j)$$

The formal definition of data-exchange settings is the following.

Definition 10.1 (Data-Exchange Settings). *A data-exchange setting is a tuple* (S, T, M, C_T), *where S is a source schema, T is a target schema, M is a set of tuple-generating dependencies describing a semantic mapping between S and T, and* C_T *is a set of constraints on the target schema T.* □

Example 10.2

Consider the following example, where the source schema S has two relations. The relation Teaches stores pairs of professor and student where the professor teaches a class taken by the student, and the relation Adviser stores pairs of (adviser, student). The target schema T has three relations: Advise is the same as Adviser, TeachesCourse represents which courses professors teach, and Takes stores the student enrollment in courses. The following are the schema mappings between S and T:

r_1 : Teaches(prof, stud) → $\exists D$ Advise(D,stud)
r_2 : Teaches(prof, stud) → $\exists C$ TeachesCourse(prof,C), Takes(C,stud)
r_3 : Adviser(prof, stud) → Advise(prof, stud)
r_4 : Adviser(prof, stud) → $\exists C, D$ TeachesCourse(D, C), Takes(C,stud)

10.2.2 Data-Exchange Solutions

Data-exchange solutions are instances of the target schema that satisfy the constraints in the data-exchange setting.

Definition 10.2 (Data-Exchange Solutions). *Let $D = (S, T, M, C_T)$ be a data-exchange setting. Let I be an instance of S. An instance J of T is said to be a data-exchange solution for D and I if*

1. *the pair (I, J) satisfies the schema mapping M, and*
2. *J satisfies the constraints C_T.* □

Example 10.3

Consider the following instance, I, for the schema S in Example 10.2.

I = {Teaches(Ann, Bob), Teaches(Chloe, David),
Adviser(Ellen, Bob), Adviser(Felicia, David)}

One data-exchange solution for I is the following:

J_0 = {TeachesCourse(Ann, C_1), Takes(C_1, Bob)
TeachesCourse(Chloe, C_2), Takes(C_2, David)
Advise(Ellen, Bob), Advise(Felicia, David)}

As seen in the example, data-exchange solutions include two kinds of values in their tuples: constants from the original instance I and new symbols that did not occur in I. Intuitively, the new symbols, which we refer to as *variables* (sometimes called *labeled nulls*), represent incomplete information we may have about the instance. More precisely, we know that the facts in which they appear hold for some substitution for the variables, but we do not know the exact substitution. For example, in J_0 the symbols C_1 and C_2 represent unknown values, describing the fact that we know there must be courses that Ann and Chloe teach, but we don't know the precise courses.

The following are also data-exchange solutions for I:

J = {TeachesCourse(Ann, C_1), Takes(C_1, Bob)
TeachesCourse(Chloe, C_2), Takes(C_2, David)
Advise(D_1, Bob), Advise(D_2, David)
Advise(Ellen, Bob), Advise(Felicia, David)}

J'_0 = {TeachesCourse(Ann, C_1), Takes(C_1, Bob)
TeachesCourse(Chloe, C_1), Takes(C_1, David)
Advise(Ellen, Bob), Advise(Felicia, David)}

The instance J has two more Advise tuples than J_0. The extra tuples contain new variables. The instance J'_0 is the same as the instance J, except that instead of having two variables C_1 and C_2, it uses the same variable throughout. Later we will see that J_0 has more desirable properties than J or J'_0.

10.2.3 Universal Solutions

As we pointed out earlier, the challenge is to find the data-exchange solutions that are in some sense better than others. In this section we discuss two properties that distinguish data-exchange solutions. The first property, satisfied by *universal* solutions, guarantees that the solution does not *lose* any information. The second property, satisfied by *core universal* solutions, guarantees that the solutions are as small as possible.

Example 10.4

Returning to Example 10.3, we see that solution J'_0 is not completely satisfactory. Specifically, it uses the same variable for the course taught by Ann and the course by Chloe, thereby implying that they teach the same course. However, that equality is not implied by the original instance, I, or from the schema mapping. The instance J'_0 is, in some sense, more specific than the other solutions.

Intuitively, universal solutions are ones that do not lose or add any information. The way we formalize this property is to require that a universal solution can be homomorphically mapped to *any other* universal solution.

In defining homomorphisms, we denote by $\mathcal{C}$ the set of constants appearing in instances of S. In addition to constants in $\mathcal{C}$, instances of T can also include variables taken from an infinite alphabet $\mathcal{V}$ that is assumed to be disjoint from $\mathcal{C}$.

Definition 10.3 (Instance Homomorphism). *Let J_1 and J_2 be two instances of the schema T.*

- *A mapping $h : J_1 \to J_2$ is a homomorphism from J_1 to J_2 if*
  - *$h(c) = c$ for every $c \in \mathcal{C}$,*
  - *for every tuple $R(a_1, \dots, a_n) \in J_1$, the tuple $R(h(a_1), \dots, h(a_n)) \in J_2$.*
- *J_1 and J_2 are homomophically equivalent if there are homomorphisms $h : J_1 \to J_2$ and $h' : J_2 \to J_1$.* □

We can now define universal solutions. In Example 10.3 the solutions J and J_0 are universal solutions, but J_0' is not.

Definition 10.4 (Universal Solutions). *Let $D = (S, T, M, C_T)$ be a data-exchange setting, and let I be an instance of S. We say that J is a universal solution for D and I if J is a data-exchange solution for D and I, and for every other data-exchange solution, J', for D and I, there exists a homomorphism $h : J \to J'$.* □

It can be shown that under very broad conditions on M and C_T, a universal solution exists if and only if there is a data-exchange solution for D and I, and that a universal solution can be constructed in polynomial time in the size of I. It is easy to verify that if C_T is empty, then there always exists a universal solution to a data-exchange problem. When C_T includes equality-generating dependencies, then situations may arise where a universal solution does not exist.

In what follows, we restrict ourselves to the case in which C_T is empty and show that we can create a universal solution using a classical algorithm called the *chase*. Informally, the chase considers every formula r of M in turn. If the algorithm finds a variable substitution for the left-hand side of r for which the right-hand side is not already in the data-exchange solution, then it adds an appropriate tuple to the solution. In creating the new tuple, the algorithm uses fresh variables to substitute for the $\bar{Y}$ variables of r. The chase is shown in Algorithm 10.

Example 10.5

Continuing with our running example, recall the mapping M and the instance I:

r_1 : Teaches(prof, stud) → $\exists D$ Advise(D,stud)
r_2 : Teaches(prof, stud) → $\exists C$ TeachesCourse(prof,C), Takes(C,stud)
r_3 : Adviser(prof, stud) → Advise(prof, stud)
r_4 : Adviser(prof, stud) → $\exists C, D$ TeachesCourse(D, C), Takes(C,stud)

I = {Teaches(Ann, Bob), Teaches(Chloe, David),
Adviser(Ellen, Bob), Adviser(Felicia, David)}

Applying the chase algorithm of Algorithm 10 on I would yield the following instance for J:

Algorithm 10. CreateUniversalSolution

```
Input: data exchange setting $(S, T, M)$ and an instance $I$ of $S$. Output: data instance $J$ containing a
  universal solution.
  Let $J = \emptyset$
  while new tuples can be added to $J$ do
    Let $r \in M$ be of the form $(\forall \bar{X}) s_1(\bar{X}_1), \ldots, s_m(\bar{X}_m) \rightarrow (\exists \bar{Y}) t_1(\bar{Y}_1), \ldots, t_l(\bar{Y}_l)$
    Let $\psi$ be a mapping from $\bar{X}$ to constants in $I$ such that $s_1(\psi(\bar{X}_1)), \ldots, s_m(\psi(\bar{X}_m)) \in S$
    Let $\bar{Y}'_i = \psi(\bar{Y}_i)$ for $1 \leq i \leq l$
    if there does not exist a mapping $\phi$ from $\bar{Y}'$ to the constants in $J$ s.t. $t_1(\phi(\bar{Y}'_1)), \ldots, t_l(\phi(\bar{Y}'_l)) \in J$
    then
      Let $\mu$ be a mapping that maps each variable $Y' \in \bar{Y}'$ to a new variable that does not appear
      in $J$
      Insert $t_1(\mu(\bar{Y}'_1)), \ldots, t_l(\mu(\bar{Y}'_l))$ into $J$
    end if
  end while
  return $J$
```

{Advise(C_1, Bob), Advise(C_2, David)
TeachesCourse(Ann, C_3), Takes(C_3, Bob),
TeachesCourse(Chloe, C_4), Takes(C_4, David),
Advise(Ellen, Bob), Advise(Felicia, David)}

The first two facts are generated by r_1. The next four facts are generated by r_2, and the last two facts are generated by r_3. The mapping formula r_4 does not add new facts to the solution.

10.2.4 Core Universal Solutions

A conceptual problem with universal solutions is that they still may be of arbitrary size. To see why, consider the following set of data-exchange solutions that can be generated in our example.

J_m = {TeachesCourse(Ann, C_1), Takes(C_1, Bob)
TeachesCourse(Chloe, C_2), Takes(C_2, David)
...
TeachesCourse(Ann, C_{2m-1}), Takes(C_{2m-1}, Bob)
TeachesCourse(Chloe, C_{2m}), Takes(C_{2m}, David)
Advise(Ellen, Bob), Advise(Felicia, David)}

The example shows that we can actually create a universal solution of any size we want. Clearly, in a data-exchange scenario we would like to materialize the smallest universal solution. Below we define core universal solutions that are the smallest universal solutions.

To define core universal solutions, we first define subinstances of database schemata.

Algorithm 11. CreateCoreUniversalSolution

```
Input: data exchange setting $D = (S, T, M)$; an instance $I$ of $S$; a universal solution $J$ for $D$ and $I$.
  Output: updated data instance $J$ containing a core universal solution.
  while there is a tuple $\bar{t} \in J$ such that $\{J - \bar{t}\}$ satisfies $M$ do
    Remove $\bar{t}$ from $J$
  end while
  return $J$
```

Definition 10.5 (Subinstances). *Let the relations in the schema T be $T_1, \ldots, T_m$, and let J be an instance of T that for $1 \leq i \leq m$ contains the tuples $\bar{t}_i$ for the relation T_i. Let A be the set of constants and variables mentioned in tuples of J.*

Let J' be an instance of T with relation instances $\bar{t}'_1, \ldots, \bar{t}'_m$ and let B be the constants and variables mentioned in J'.

We say that J' is a subinstance of J if $B \subseteq A$ and $t'_j \subseteq t_j$ for $1 \leq j \leq m$. J' is a proper subinstance of J if one of the inclusions is strict. □

We define core universal solutions to be solutions for which there is no substructure instance that is also a solution.

Definition 10.6 (Core Universal Solutions). *Let $D = (S, T, M, C_T)$ be a data-exchange setting and let I be an instance of S. We say that J is a core universal solution for D and I if J is a universal solution to D and I, and if there is no proper sub-instance of J that is also a universal solution for D and I.* □

It can be shown that the core universal solutions for a data-exchange setting are unique up to an isomorphism. While, in general, finding a core universal solution is intractable, there is a broad range of cases where it can be done in polynomial time in the size of I. One of these cases is when C_T is empty, and Algorithm 11 shows a greedy algorithm for finding the core universal solution in this case. More efficient algorithms for finding core universal solutions have been proposed in the literature, but the greedy one illustrates the key ideas best.

Example 10.6

Consider applying the greedy algorithm in Algorithm 11 to the universal solution created in Example 10.5.

{Advise(C_1, Bob), Advise(C_2, David)
TeachesCourse(Ann, C_3), Takes(C_3, Bob),
TeachesCourse(Chloe, C_4), Takes(C_4, David),
Advise(Ellen, Bob), Advise(Felicia, David)}

The facts Advise(C_1, Bob) and Advise(C_2, David) will be removed because r_1 is satisfied without them. All the other facts must remain in J.

10.2.5 Querying the Materialized Repository

We now have the necessary machinery to create the materialized repository from a set of data sources described by GLAV schema mappings. The materialized repository is the data-exchange solution, where S is a data source and T is the schema of the materialized repository. We can create such as solution for every source in isolation and take their union.

The final problem we need to address is how to *query* the materialized repository. Fortunately, the machinery developed for query reformulation and inverse rules (Section 2.4.5) applies. Specifically, the following theorem shows that we can answer queries by evaluating them on the materialized repository and keeping only the answer tuples that do not contain variables. The theorem follows from Theorem 3.3.

Theorem 10.1. *Let S be a set of data sources and T be the schema of the materialized repository. Let M be a set of GLAV mappings between S and T. Let I be an instance of S and let J be a universal data-exchange solution for (S,T,M) and I.*

Let Q be a union of conjunctive queries without interpreted predicates over T. Let $Q(J)$ be the result of applying Q to J, and let $Q(J)'$ be the subset of $Q(J)$ that includes tuples that contain only constants from I.

Then, $Q(J)'$ is precisely the certain answers to Q. □

10.3 Caching and Partial Materialization

Compared to a virtual integration system, materialization enables us to execute more computationally and disk-intensive queries, over larger amounts of (especially historical) data. However, on the flip side, materialization often imposes a *delay* between when a source is updated and when the updated results appear in the integrated data instance — and hence in users' query results. This may be a problem in settings like *customer relationship management*, where we may need to join the state of multiple internal and external databases, as well as Web services from shipping companies, suppliers, etc.

Naturally, in practice there are many "hybrid" approaches to addressing these problems. Often, some data sources and relations are updated more frequently than others: for instance, the business vocabulary in a Master Data Management system is unlikely to change at a high rate, even if individual line items in a billing system are frequently updated.

Thus, in practice most deployed integration systems — whether designed as warehouses or virtual integration ("Enterprise Information Integration," or EII) systems — exploit *caching* and *partial materialization* over those results that may not need to be 100% up to date or that seldom change and use virtual integration over the other sources. We briefly discuss a variety of potential approaches to "hybrid" models, where queries may be answered over data materialized at the central integration system, rather than being reformulated over remote sources.

CACHING AND DIRECT REUSE IN A VIRTUAL INTEGRATION SCENARIO

A simple, low-maintenance approach to get some of the benefits of materialization in a virtual integration system is to directly *cache* the results of queries and to use answering-queries-using-views algorithms (Chapter 2) to rewrite subsequent queries to reuse the data. One challenge is that the cached entries must be assigned a *time-to-live* after which they should be expired, in the event that the source data have changed. However, this approach otherwise requires little direct intervention from the user or administrator. It is sometimes termed *mid-tier* or *middle-tier caching*.

ADMINISTRATOR-SELECTED VIEW MATERIALIZATION

The database administrator may determine that certain data change slowly and may manually specify how to materialize them — using either ETL or declarative mappings. User queries will be posed over a combination of materialized relations plus virtual relations, and query reformulation will take this into account.

AUTOMATED VIEW SELECTION

A more sophisticated approach is to have the data integration system choose what to materialize, given a set of declarative (not ETL!) mappings, a *query workload*, and perhaps a specification of what source data are relatively static. From this, automated view selection algorithms, using techniques similar to those in the “tuning wizards” of commercial DBMSs, can be used to identify common query expressions and to materialize them.

10.4 Direct Analysis of Local, External Data

Thanks to the preponderance of Web-based applications in which logs are processed to detect patterns — ad click-throughs, site visit patterns, page co-views — and the popularity of Google’s MapReduce programming paradigm for doing distributed computation, there are a wide number of commercial platforms for processing data stored not in database tables but simply in flat files with delimiters. Such data may come from a wide variety of Web server logging platforms and application modules, and they are generally append-only logs describing behavior over time. Thus most of the benefits of a transactional DBMS storage system are less pertinent. In some sense, these logs already represent a locally materialized data warehouse, merely one where the data are stored outside a DBMS. In such settings — unless there is a strong need for indexing — it becomes more desirable to simply process data directly from the files, rather than first importing the data and then querying them. Furthermore, many of the queries of interest are transient, so there is little benefit in setting up a database system to store and query the data.

The integration task in these contexts is generally to perform aggregation queries — somewhat less structurally complex ones than in data warehouse (OLAP) settings, though sometimes computations that require custom logic (what in the database world we would call "user-defined functions" written in a language like Java, C, or Python). For instance, a social network advertiser might want to know the performance of certain ad campaigns, by type of ad, region, demographic, etc., or may want to make friend recommendations based on commonly visited pages.

For such tasks, the Google MapReduce programming model (often implemented in the open-source Apache Hadoop MapReduce platform) is commonly used. MapReduce provides a particular programming "template" for parallel analysis of large-scale data sets; it is roughly analogous to a single SQL query block. The basic programming model is as follows:

- A function called *map* — provided by the user — is given input records (typically in the form of lines of text from a file) one at a time. Its tasks are to (1) parse the records, (2) extract the relevant information from the records, (3) filter data based on the records, and (4) output for each record one or more subresults. One can think of it as primarily serving the role of the WHERE clause in SQL, with user-defined predicates. However, it can also split records, derive new values, and so on.
- The outputs of the map computations are grouped together according to *keys*. This is done through a *shuffle* stage. By default, the shuffle stage *sorts* the data according to the key.
- For each group of map outputs with the same key, a user-provided *reduce* computation is invoked. This computation can output zero or more values. One can think of this as being equivalent to an SQL GROUP BY on the key, followed by a user-defined aggregate function over the group.

Oddly to a database audience, the core MapReduce framework does not directly support the composition of analysis queries with one another. However, in practice one can "string together" multiple MapReduce stages, with each reading the results of the previous, to accomplish composition. For even more complex analysis (performing recursive or iterative computations like link analysis), it is common practice to repeatedly invoke a MapReduce computation within a loop in a shell script or a Java program.

Example 10.7

Suppose we are running a large, multisite Web property, and our goal is to count the number of visits to each Web page. To do this, we need to extract requests from our various Web server logs, which are generally created on a daily basis per machine. We then want to group them together by URL and count them. If the log were a relational table, this query would be trivial to express as a GROUP BY/COUNT query in SQL.

In MapReduce we write two functions, *map*, shown in Algorithm 12, and *reduce*, shown in Algorithm 13. As we can see, these functions are both quite brief, given that the task is so simple. The *map* function is called once per line of a log file; it is also given information about the byte position of the line in the log (which it ignores). It outputs a pair (*key*, *value*), using *url* as the key and the number 1 as the value.

Algorithm 12. User-provided *map* for grouping Web-log URLs

```
Input: string line; logfile position position; target output. Output: writes a record to output.
  Let url := extractUrlFromLine(line)
  Call output.emit(url,1)
```

Algorithm 13. User-provided *reduce* for counting Web-log requests per URL

```
Input: key url; set of counts valueSet; target output. Output: writes a record to output.
  Call output.emit(url,valueSet.size())
```

Now the MapReduce framework performs its *shuffle* stage, sorting all *map* output to group by key. It then invokes the *reduce* code over each key (URL) value, as well as the set of all values matching the key. The cardinality of this set is precisely the count, so we can output that.

One might ask the question: Why is MapReduce so popular in the Web world? In part it is because the framework makes it easy to do large-scale processing of external data, while also implementing custom filtering or aggregation functions. Unlike with SQL, one can seamlessly incorporate user-defined code in C, Java, or Python with little effort. In contrast, many DBMS and data warehouse platforms have fairly awkward and arcane methods of integrating user-defined functions with SQL: often the administrator must first register the functions in SQL DDL, ensure that paths are properly set up, and so on.

Perhaps more importantly, MapReduce provides a level of dependability and scalability that is unmatched in most DBMSs. A MapReduce job can be executed in parallel across thousands of nodes in a cluster (multiple *map* and *reduce* operations run simultaneously over different portions of the data). MapReduce is incredibly resilient to machine *failures*, which are surprisingly likely to occur when thousands of machines are involved. If a node dies, MapReduce will automatically reallocate its tasks to other machines and will recompute any work that was in progress at the node during the crash. MapReduce's absolute performance has sometimes been questioned, but it works quite well if the results are needed, e.g., overnight.

Some developers have felt that MapReduce provides a frustratingly low-level way of writing complex queries. This has led to a series of higher-level, compositional programming languages that compile down to MapReduce: one can get the benefits of MapReduce's "run and forget about it" execution across a cluster, while also having more natural abstractions. The Pig Latin language, for instance, is commonly used to do SQL-like queries, expressed as a nondeclarative pipeline of operators. Going even further, companies like Facebook have developed SQL translation layers (in particular, Apache Hive) to let their customers and developers pose queries in a variant of SQL, but then to run those in Hadoop over flat files.

Clearly, flat-file-based data analysis is only a small subclass of information integration, and systems based on the approach are limited in their ability to manage and integrate heterogeneous data. However, due to their simplicity, MapReduce and its relatives are rather effective in tying together data from multiple autonomous systems, and they are highly scalable. They therefore serve a very important role in the back ends of many Web-based systems and services.

Increasingly, some organizations have begun to use MapReduce not to do the actual data analysis, but to do data *transformation* in order to load the data into a data warehouse: this is a different way of performing ETL.

Bibliographic Notes

The data warehouse has been the traditional approach to data integration from the early days, largely because the early applications revolved around online analytic processing (OLAP), forecasting, and data mining. An excellent survey by Chaudhuri and Dayal appears in [122]. A major challenge in data warehousing is that of incrementally maintaining the data warehouse; this was the focus of the Stanford WHIPS project [129, 447, 486, 596].

Fagin et al. [216] first introduced the notion of data exchange and defined its semantics. An excellent overview of the data-exchange literature appears in the book [36]. The work on data exchange was originally motivated by the Clio System [302], which we discussed in Chapter 5. In [216] they introduced universal solutions and showed that they are homomorphically equivalent to each other and that certain answers can be obtained by any universal solution. Core universal solutions were introduced by Fagin et al. [217]. Both of these papers show precise conditions under which we can find universal and core universal solutions efficiently, and in particular how it depends on the form of the mappings between the source and target schemas and on the constraints present on target schema. They also describe an algorithm for finding the core universal solution that is more efficient than the greedy algorithm we presented. Afrati and Kolaitis showed how to answer aggregation queries in the data-exchange setting in [15], which is important in a data warehouse/OLAP setting.

It is interesting to note that the process of creating a universal solution for a data-exchange problem is similar in spirit to applying the Inverse-Rules Algorithm (Section 2.4.5) on a set of data sources in the context of virtual data integration. The two processes produce a materialized instance of the target schema where some of the tuples have variables. The difference is that in the data-exchange context the goal is to produce the materialized instance of the schema T, whereas in the virtual data integration context the goal is to answer the query that is posed over the mediated schema. Hence, in that context the inverse rules are part of the rewriting of the query using the source descriptions, but the hope is that the query optimizer will find a plan that does not require materializing a complete instance of T.

The problems of caching at an integration or middleware layer have been studied in a variety of contexts and generally are labeled “mid-tier caching” [402]. Commercial implementations are typically based on relatively simple matching templates. One of the first systems to look at a combination of virtual and materialized data integration together was the H2O project [592].

The specific problem of choosing *what* views to materialize, given a workload and a set of database statistics, has been extensively studied in a wide variety of contexts. Among the earliest work on this problem was that of Gupta [276, 277]. Chirkova et al. studied the theoretical complexity of the problem, showing that it can be exponential in the size of the views [134] — but that the problem is often polynomial when standard estimation heuristics are used. More recently, the view selection problem has been incorporated into a multiple query optimizer [437]. The problem of maintaining materialized warehouse views was studied in [149, 398]. View materialization for aggregate queries was considered in [14]. The Active XML [4] project focused on how to decompose integration tasks over Web services into virtual and materialized components.

At the time of the writing of this book, the use of MapReduce in database-style applications was attracting significant interest in the database community. A variety of projects have explored combining MapReduce and SQL database techniques, including HadoopDB [8]. In [531], Stonebraker et al. have also suggested that MapReduce should primarily be considered as complementary to data warehousing, for instance, as an ETL tool framework. In fact, there do exist ETL frameworks written over MapReduce, such as ETLMR [396].