Chapter 1

Data Integration Services

1 Introduction

With the prevalence of the network technology and the Internet, access to data independent of its physical storage location has become highly facilitated. This further has enabled users to access a multitude of data sources that are related in some way and to combine the returned data to come up with useful information which is not physically stored in a single place. For instance, a person who has the intension of buying a car can query several car dealer web sites and then compare the results. He can further query a data source which provides information about car reviews to help his decision about the cars he liked. As another example, imagine a company which has several branches in different cities. Each branch has its own local database recording its sales. Whenever global decisions about the company have to be made, each branch database must be queried and the results must be combined. On the other hand, contacting data sources individually and then combining the results manually every time an information is needed is a very tedious task. Instead, a service is needed which provides transparent access to a collection of related data sources as if these sources as a whole constituted a single data source. We call such a service a *data integration service* and the system that integrates multiple sources to provide this service is usually referred to as a *data integration system* (Figure 1.1).

The main contribution of a data integration system is that users can focus on specifying *what* data they want rather than on describing *how* to obtain it. A data integration system relieves the user from the burden of finding the relevant data sources, interacting with each of them separately, then combining the data they return. To achieve this, the system provides an integrated view of the data stored in the underlying data sources. Users can uniformly access all the data sources as if they were querying a single data source. The access to the integrated data is usually in the form of querying rather than updating the data.

Furthermore, a data integration system facilitates decision support applica-

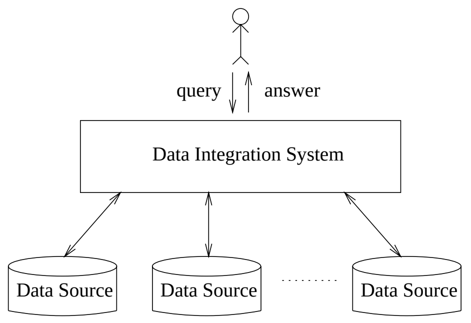

Figure 1.1: Data Integration System

tions like OLAP (On-Line Analytical Processing) and data mining. *OLAP* is to perform financial, marketing or business analysis to be able to make business decisions on a collection of data from one or more data sources. The analysis is done through asking a large number of aggregate queries on the detailed data. For example, the company in our previous example can easily develop OLAP applications once its branch databases are integrated. *Data Mining* is discovering knowledge from a large volume of data. Statistical rules or patterns are automatically found from the raw collection of data. Data integration helps bringing a large body of data together from multiple data sources that can be uniformly queried for knowledge discovery. Detailed information on data mining techniques can be found in the *Customization Chapter*.

In this chapter, we discuss the issues involved in building and operating a data integration system and provide a survey of existing solutions to these issues.

1.1 Major Issues

Let us investigate the main stages involved in building and using a data integration system to comprehend the major issues: design, modeling, and operation.

- A data integration system is basically an information system. Like all computer systems, its architecture has to be designed with data sources to be integrated being the major components. Usually, the data sources must be integrated as they are without making any changes on their design and operation.

- Also, like all information systems, there is an application domain a data integration system has to model. This application domain is determined by the underlying data sources and its modeling should be based on the models of the data sources that make up the integration system.

- After its modeling and design, a data integration system has to be provided with query functionality. This is again highly dependent on the underlying data sources' query capabilities.

Although the contents of the data sources are related in some way, they are likely to show variety in many aspects. These differences make both the design and modeling phase and the operation phase of a data integration system very difficult. The major issue in building a data integration system is resolving these differences between the data sources that may occur at different levels. This issue is generally referred to as *heterogeneity* of the data sources.

The data sources to be integrated may belong to the same enterprise (like the company example), but might also be arbitrary sources on the World Wide Web (like the car buyer example). Most of the time, each of the sources is independently designed for autonomous operation. Also, the sources are not necessarily databases; they may be legacy systems which are old and obsolete systems that are difficult to migrate to a modern technology or they may be structured/unstructured files with different interfaces. Data integration requires that the differences in modeling, semantics and capabilities of the sources, with possible data inconsistencies be resolved. More specifically, the major issues that make integrating such data difficult include:

- Heterogeneity of the data sources
  Each source to be integrated might model the world in its own way. The representation of data of the similar semantics might be quite different in each data source. For example, each might be using different naming conventions to refer to the same real world object. Moreover, they may contain conflicting data. In addition to data representation and modeling differences, heterogeneity may also occur at lower levels including the access methods the sources are using, the operating systems underlying the individual data sources, etc.

- Autonomy of the data sources
  Usually data sources are created in advance of the integrated system. In fact, most of the time they never know that they are part of an integration. They can make decisions independently and they can not be forced to act in certain ways. As a natural consequence of this, they can also change their data or functionality without any announcement to the outside world.

- Query correctness and performance
  Queries to an integrated system are usually formulated according to the unified model of the system. These queries need to be translated into forms that can be understood and processed by the individual data sources. This mapping should not cause incorrectness in query results. Also, query performance needs to be controlled as there are many factors which can degrade it. These include the existence of a network environment which can cause communication delays and the possible unavailability of the data sources for answering queries.

1.2 Chapter Outline

In the rest of this chapter, we discuss the above mentioned issues in more detail. The order of the subsections in the chapter roughly corresponds to the stages involved in building and operating a data integration system. We start out presenting the common approaches to architecting a data integration system in Section 2. Later, we discuss the semantic problems encountered in modeling and data mapping stages of a data integration system in Section 3. Techniques for querying the integrated data are presented in Section 4. The data extraction phase of querying where data is actually obtained from the data sources is detailed in Section 5. We devoted Section 6 to the discussion of an important issue in one particular type of data integration architecture: management of materialized views in datawarehousing systems. This section completes our discussion about the major problems and solutions. Finally, Section 7 concludes the chapter.

2 Data Integration Architectures

The data sources can be organized in the integration system in many ways. In this section we introduce three main architectures of data integration systems: federated databases, mediation, and data warehousing. We group these approaches based on whether the queries to the data sources are sent to the sources when these queries arrive, or the results of the queries are pre-stored. The former approach is a virtual approach and the latter is a materialized approach to data integration. We compare the approaches at the end of this section. We use three parameters to describe the characteristics of the sources of these integration systems: autonomy, heterogeneity, and distribution [Has00, OV99].

- Autonomy
  Autonomy indicates how independent the data sources are from the other sources and from the integrated system. According to Veijalainen and Popescu-Zeletin's classification [MW88], there are three types of autonomy:

  - Design autonomy
    The source is independent in data models, naming of the data elements, semantic interpretation of the data, constraints etc.
  - Communication autonomy
    The source is independent in deciding what information it provides to the other components that are parts of the integrated system and to which requests it responds.
  - Execution autonomy
    The source is independent in execution and scheduling of incoming requests.

- Heterogeneity
  Heterogeneity refers to the degree of dissimilarity between the component data sources that make up the data integration system. It occurs at different levels. On a lower level, heterogeneity comes from different hardware platforms, operating systems, and networking protocols. On a higher level, heterogeneity comes from different programming and data models as well as different understanding and modeling of the same real-world concepts (i.e. naming of relations and attributes).

  Logical heterogeneity can not be resolved automatically as it comes from the fact that different people present the same concept differently. It involves both *schematic* and *semantic heterogeneity*. Schematic problems are the differences in the elements that are used to represent some concept. For example, to store the information about voluntary student positions in the University, one database developer may use the attributes names for each job `(Tea Czar, Hospitality Czar)` with true/false values for each student; the other developer may model these jobs as values of the attribute `Job`. Some of the semantic problems that arise are the interpretation of names and the difference in units used for the attributes. We discuss these issues in Section 3.

- Distribution
  Distribution refers to the physical distribution of data over multiple sites. Creating an integrated system and choosing the appropriate architecture, the designers should take into account the possible latency to communicate with the data sources.

We further consider the most difficult case: fully-distributed and heterogeneous systems with autonomous or semi-autonomous data sources. *Metadata* - the auxiliary data describing the main data - is maintained in the integrated systems to deal with the problems caused by the heterogeneity. It can contain both technical information about the sources (such as query capabilities and access methods), and also semantic information (such as the semantic connections between the relations, the domain dictionary specification) [BKLW99].

We describe the main architectural approaches to the design of the data integration systems, and discuss some solutions to the issues caused by autonomy, distribution and heterogeneity.

2.1 Major Approaches to Data Integration

Two common approaches to integrate data sources are the following:

- Virtual View Approach
  In this case the data is accessed from the sources on-demand when a user submits a query to the information system. This is also called a *lazy* approach to data integration.

- Materialized View/Warehousing Approach
  Some filtered information from data sources is pre-stored (materialized) in a repository (warehouse) and can be queried later by users. This method is also called an *eager* approach to data integration.

Sometimes a hybrid approach is used: integrated data is selectively materialized. The data is extracted from sources on-demand, but the results of some queries are pre-computed and stored. In order to choose what queries to materialize, designers should consider many factors, such as "popularity" of queries and cost of maintenance [Ash00]. These issues are discussed in Section 6.

2.2 Virtual View Approach

Here we discuss two architectures for integrating data sources using a virtual view approach. They are federated database systems and mediated systems.

2.2.1 Federated Database Systems

A *Federated Database System (FDBS)* consists of semi-autonomous components (database systems) that participate in a federation to partially share data with each other [SL90]. Each source in the federation can also operate independently from the others and the federation.

The components can not be called "fully-autonomous" because each component is modified by adding an interface that allows communication with all other databases in the federation.

Each of the component database systems can be either a centralized DBMS, a distributed DBMS, or another federated database management system, and may have any of the three types of autonomy mentioned above (design autonomy, communication or execution autonomy). As a consequence of this autonomy, heterogeneity issues become the main problem.

There are *loosely coupled* FDBSs and *tightly coupled* FDBSs.

A tightly coupled FDBS has a unified schema[1] (or several unified schemas) which can be either semi-automatically built by schema integration techniques (see Section 3 for details) or created manually by the users. To solve the logical heterogeneity, a domain expert needs to determine correspondences between schemas of the sources. A tightly coupled FDBS is usually static and difficult to evolve, because schema integration techniques don't allow to add or remove components easily. An example of this kind of FDBSs is Mermaid [TBC$^+$87].

A loosely coupled FDBS does not have a unified schema, but it provides some unified language for querying sources. In this configuration, component database systems have more autonomy, but humans must resolve all semantic heterogeneities. Requested data comes from the exporter of this data itself and each component can decide how it will view all the accessible data in the federation. As there is no global schema, each source can create its own "federated schema" for its needs. Examples of such systems are MRSDM [Lit85], Omnibase [Rea89] and Calida [JPSL$^+$88].

As pointed out by Heimbigner and McLeod [HM85], in order to remain autonomously functioning systems and provide mutually beneficent sharing of data at the same time, components of FDBS should have facilities to communicate in three ways:

- Data exchange
  The components should be able to access the shared data of the other components of the FDBS. This is the most important purpose of the federation and good mechanisms of data exchange are a must.

- Transaction sharing
  There may be cases where for some reason the component does not want to provide direct access to some of its data, but can share operations

[1]Unified schema is the schema produced out of the schemas of the integration system components, after resolving all syntactic and semantical conflicts between these schemas. This schema allows users to query the integrated system as if it were one database.

on its data. Other components should have the ability to specify which transactions they want to be performed by another component.

- Cooperative activities
  As there is no centralized control, cooperation is the key in federation. Each source should be able to perform a complex query involving accessing data from other components.

The most naive way to achieve interoperability[2] is to map each source's schema to all others' schemas. It is a so-called pair-wise mapping. You can see an example of such federated database system in Figure 1.2. Unfortunately, it requires $n \cdot (n-1)$ schema translations and becomes too tedious with a large number of components in a federation. Research is being done on tools for efficient schema translation (See Section 3 for details).

We should note that the term "Federated Database Systems" is used differently in the literature: some researchers call only tightly coupled systems FDBSs [BKLW99], some call only loosely coupled systems FDBSs [HM85], and some take the same approach we did by considering tight and loose architectures be two kinds of federated database system architecture [SL90].

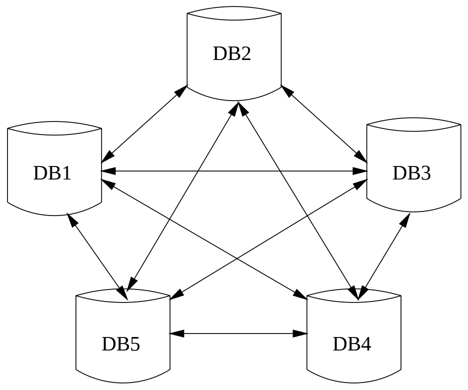

Figure 1.2: Example of federated database architecture

Federated architecture is very appropriate to use when there is a number of autonomous sources, and we want, on one hand, to retain their "independence" allowing user to query them separately, and, on the other hand, allow them to collaborate with each other to answer the query.

2.2.2 Mediated Systems

Mediated system integrates heterogeneous data sources (which can be databases, legacy systems, web sources, etc) by providing virtual view of all this data. Users asking queries to the mediated system do not have to know about data source

[2]Interoperability here means the ability of each source to use the data of the other sources.

location, schemas or access methods, because such system presents one global schema to the user (called *mediated schema*) and users ask their queries in terms of it.

A mediation architecture is different from a tightly coupled federation in the following ways [SL90]:

- A mediated architecture may have non-database components
- The query capabilities of sources in a mediator-based system can be restricted and the sources do not have to support SQL-querying at all
- Access to the sources in a mediator-based system is usually read-only as opposed to read-write access in a FDBS (due to the fact that the sources in the mediator-based system are more autonomous) [BKLW99]
- Sources in a mediator-based approach have complete autonomy which means it is easy to add or remove new data sources

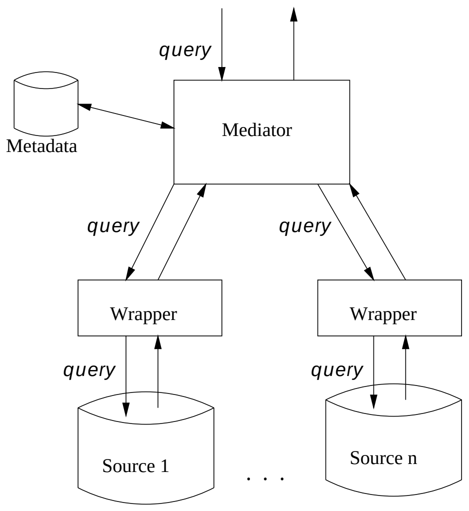

Figure 1.3: Mediated architecture (borrowed with some minor changes from [GMUW00])

A typical architecture for a mediated system is shown in Figure 1.3. The main components of a mediated system are the *mediator* and one *wrapper* per data source. The mediator (sometimes also called an *integrator*) performs the following actions in the system:

1. Receives a query formulated on the unified (mediated) schema from a user.
2. Decomposes this query into sub-queries to individual sources based on *source descriptions*.

3. Optimizes the execution plan based on source descriptions.
4. Sends sub-queries to the wrappers of individual sources, which will transform these sub-queries into queries over sources' local models and schemas. Then the mediator receives answers to these sub-queries from wrappers, combines them into one answer and sends it to the user.

These steps are described in detail in Section 4.

A wrapper hides technical and data model details of the data source from the mediator. It is an important component of both a mediator-based architecture and a data warehouse. Please refer to Section 5 for more information about wrappers.

Example
Let us assume there are two data sources - two car dealer databases which both became parts of Acme Cars company. Each of the car dealers has a separate schema for storing information about cars. Dealer 1 stores it in the relation:

```
Cars(vin, make, model, color, price)
```

Dealer 2 stores information about his cars for sale in the relation:

```
CarsForSale(vehicleID, carMake, carModel, carColor, carPrice).
```

Acme Cars uses a mediated architecture to integrate these two dealers' databases. It does this by providing a mediated schema of the two schemas above. The mediated schema consists of just one relation:

```
Automobiles(vin, autoMake, autoModel, autoColor, autoPrice).
```

Now if a client of Acme Cars submits an SQL-query:

```
SELECT vin, autoModel, autoColor
FROM Automobiles
WHERE autoMake = "Honda" AND autoPrice < 14,000
```

The wrapper for the first database will translate this query to:

```
SELECT vin, model, color, year
FROM Cars
WHERE make = "Honda" AND price < 14,000
```

It also renames `model` to `autoModel` and `color` to `autoColor`. The wrapper for the second dealer will translate this query to:

```
SELECT vehicleID, carModel, carColor
```

```
FROM CarsForSale
WHERE carMake = "Honda" AND carPrice < 14,000
```

The wrapper also renames `vehicleID` to `vin`, `carModel` to `autoModel` and `carColor` to `autoColor`.

Some known implementations of mediator-based architecture are: TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) [CGMH+94], Information Manifold [KLSS95], SIMS [AHK96], and Carnot [HSC+97].

2.3 Materialized View Approach (Data Warehousing)

In a materialized view approach, data from various sources is integrated by providing a unified view of this data, like in a virtual view approach, but here this filtered data is actually stored in a single repository (called *data warehouse*). A data warehouse is different from the traditional databases with OLTP (On-Line Transaction Processing) in the following ways [CD97]:

- It is mainly designed for decision support. As a consequence, a data warehouse often contains historical and summarized data. That also implies that users of a data warehouse are different than users of a traditional DBMS: they will be analysts, knowledge workers, executives

- Workloads in warehouses are query intensive; queries are complex and query throughput is more important than transaction throughput

- Information is usually read-only as opposed to read/write operations in OLTP.

There are three important steps involved in building and maintaining a data warehouse:

- Modeling and design

  In the stage of designing a warehouse, the developers need to decide what information from each source they are going to use in the warehouse, what views (queries) over these sources they want to materialize, and what the global unified schema of the warehouse will be.

- Maintenance (refreshing)

  Maintenance deals with how the warehouse is initially populated from the source data and how it is refreshed when the data in the sources are updated. View maintenance is a key research topic specific to data warehousing and we discuss it in detail in Section 6.

- Operation

  Operation of a data warehouse involves query processing, storage and indexing issues.

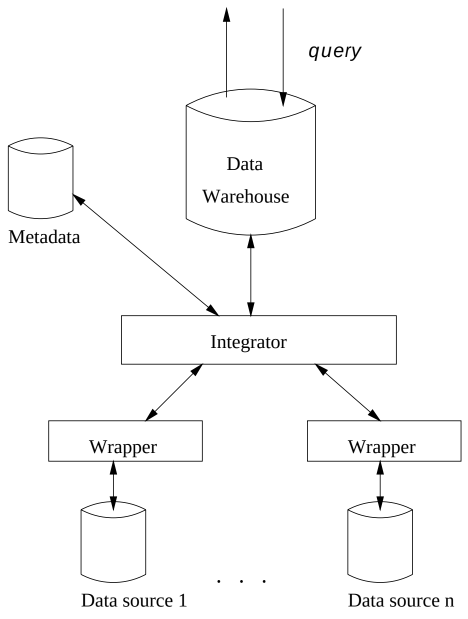

Figure 1.4: Data warehouse architecture

Example of a data warehouse architecture is given in Figure 1.4.

Example

Suppose there is a company Cute Toys that owns two toy stores. There are two types of toys at each store: teddy-bears and dogs. Each store has a database, where they store a number of toys sold on each date, for each kind of a toy. Store 1 stores the relation: `Sales(date, typeToy, numberSold)` and store 2 has two relations: `TeddyBears(date, numberSold)` and `DogsToys(date, numberSold)`.

Now assume that the company would like to have the following relation in the data warehouse for decision making purposes (future marketing):

```
ToySales(date, typeToy, numberSold)
```

In this case, the integrator needs to first select appropriate tuples from each source, take their union and then aggregate, so that for each date and type of a toy we have a total number of toys of this kind sold on a given date. The SQL query to the first source is straightforward, as the relation is exactly the same apart from the name it has. It will look the following:

```
INSERT INTO ToySales1(date, typeToy, numberSold)
SELECT date, typeToy, numberSold
FROM Sales
```

For the second source, the integrator can ask two queries:

```
INSERT INTO ToySales2(date, typeToy, numberSold)
SELECT date, "TeddyBear", numberSold
FROM TeddyBears

INSERT INTO ToySales2(date, typeToy, numberSold)
SELECT date, "Dog", numberSold
FROM DogsToys
```

So, wrappers to sources 1 and 2 will return relations ToySales1 and ToySales2 correspondingly. Now integrator component will join them summing the number of toys of each kind sold on each date:

```
INSERT INTO ToySales(date, typeToy, numberSold)
SELECT date, typeToy, SUM(numberSold)
FROM ToySales1 s1, ToySales2 s2
WHERE s1.typeToy=s2.type AND s1.date = s2.date
```

Some implementations of the data warehousing approach to data integration include the Squirrel [HZ96] and WHIPS (WareHouse Information Prototype at Stanford) [HGMW$^+$95] systems.

We would like to note that the sources that are integrated always retain their execution autonomy.

2.4 Comparison of the architectures

The virtual view approach is preferable to the data warehousing in the following cases:

- the number of data sources in an integrated system is very large and/or the sources are likely to be updated frequently (like in the case of the web sources),
- there is no way to predict what kind of queries the users will ask.

If, however, sources are permanent, don't get upgraded too often and the designers of the integrated system know what kind of queries are to be expected most often, answers to these queries can be materialized. Also, if some sources are physically located far away from the mediator, then accessing them each time a query is formulated may introduce undesired delays in response time. In this case, a data warehousing approach might be chosen to improve the performance.

Among the two architectures based on the virtual view approach (federation and mediation), mediated approach is chosen more often. As for the federation, the systems with this architecture are not very common nowadays due to the large number of interfaces that need to be written for each source to communicate with all the others.

A hybrid approach is usually discussed as a way to improve the performance of some mediator-based systems. The approach to the data integration in this case is virtual, but some selected queries are materialized in a repository. This repository then can serve as a new source for the mediated system. A hybrid approach is proposed in [Ash00], but otherwise is less commonly discussed in literature than are data warehousing and mediation.

3 Schema Integration

A *schema* is a description of how data in a database appears to be structured to users of the database. For example, in a relational database, the schema specifies what relations are in the database, what attributes are defined for each relationship, etc. In an object-oriented database, the schema specifies what classes are defined, what attributes and methods those classes have, etc.

Schema integration is the work that is performed, while constructing an integrated information system, of reconciling the schemas of the different data sources into a single, coherent schema [JLYV00].

The product of schema integration is a (perhaps new) schema that can contain all of the information that is to be available from the integrated information system. Various metrics exist for judging how good the integrated schema is, and are discussed in Section 3.3.

Schema integration can be a very easy or very difficult task, depending on how many data sources are to be integrated, and on how differently their schemas represent information. This section explores the issues that can make schema integration so problematic, and describes what techniques have been developed to deal with those problems.

3.1 Problems in Schema Integration

Schema integration problems can be broadly separated into two categories: the informal problems arising from how humans organize themselves, and problems in the formal realm of how schemas are represented.

3.1.1 Human Organizational Problems

Autonomous Data Sources When performing data integration, it is possible that the people controlling the various data sources act fairly *autonomously* with respect to the people constructing the integrated system. Autonomous data sources seems even more now than before the Internet became so popular, because the range of data sources available for integration is much larger than before.

When a data source is managed by people who are autonomous from the people constructing the integrated system, various problems can arise for the schema integration task:

- Lack of Schema Information Sharing

  The source data administrators might not be interested in, or may not have the resources, to help the integrators to understand how their site's schema relates to the schemas of other sites being integrated.

- Unannounced Schema Changes

  The source data administrators might change their site's schema without forewarning the integrators, leading the integration software to make invalid assumptions about the data source.

- Inconsiderate Schema Design

  The data source administrators might choose a schema that is very difficult to integrate with the other schemas in the integrated system. In tightly controlled organizations, the various data source administrators might be coerced into all having easily integrated schemas. Such coercion is unlikely to be possible in highly autonomous environments.

Complexity of the Set of Data Source Schemas Schema integration is a knowledge-intensive task. It is conceivable that for some large systems, no one human would ever be able to understand the the schemas of all the constituent data sources [Hal95]. This places a limitation on the human-oriented methodologies that can be used to successfully integrate such systems [ND95].

3.1.2 Logical problems

These problems fit squarely in the realm of logics, formal languages, semantics, etc. These problems are the focus of much attention in schema integration research and their formal nature lends them to attempted solutions involving logic, semantics, and knowledge representation.

Numerous incompatible taxonomies have been proposed for describing the problems that can occur in schema integration. Several representative taxonomies appear below.

The Taxonomy from [JLYV00] [3]

- Heterogeneity Conflicts

  Problems with the use of different data models in different schemas. For example, one schema may use an object oriented database, while the integrated schema must be represented with a relational database.

- Naming Conflicts

  Different schemas may use the same term to describe different concepts (*homonyms*) or two different terms to describe the same concept (*synonyms*).

- Semantic Conflicts

  When different schemas use different levels of abstraction are used to model the same entity.

  For example, one database might distinguish between “cars” and “trucks”, whereas another schema in the same integrated system might simply model “automobiles” and fail to store the car/truck distinction.

[3]It is claimed in [JLYV00] that consensus has been reached for using this taxonomy rather than competing taxonomies.

- Structural Conflicts

  Different schemas may represent the same information in different ways.

  For example, one car ownership schema may use a single table that stores car and owner information, while another schema may normalize the same information into a “car” table and an “owner” table.

The Taxonomy from [Var99] This taxonomy is largely a refinement of [JLYV00]'s *Heterogeneity Conflicts* concept, but is still slightly incompatible with the other taxonomy. [Var99] offers this as a taxonomy of *semantic inconsistencies* (e.g., semantic conflicts). However, this taxonomy includes *Naming Conflicts* as a cause of semantic inconsistency, while [JLYV00] considers naming conflicts to be very distinct from semantic inconsistencies.

- Naming Conflicts

  This is the same notion as *Naming Conflicts* from [JLYV00].

- Domain Conflicts

  Different schemas use different simple values to represent data.

  For example, one schema store care price as an integer number, while another might store a textual-rendition of the car's price in a text string.

- Metadata Conflicts

  A concept can be represented with the schema in one data source, but as regular (non-schema) data in another data source.

  For example, one data source may distinguish between cars and trucks by maintaining two separate tables, one for cars and one for trucks. Which table a record appears in specifies whether the vehicle is a car or a truck. Another data source may use a single table, but have a field in that table that indicates whether or not a row in the table represents a car or a truck.

- Structural Conflicts

  This is the same notion as *Structural Conflicts* from [JLYV00].

- Missing Attributes

  One schema may represent a superset of the information available in another schema.

  For example, in two schemas that represent cars for sale, one schema may include an attribute for the date of the car's last oil change, whereas the other schema makes no provision for storing that information.

  This issue is related to [JLYV00]'s *Semantic Conflicts* in the sense that both deal with differences in the level of detail about a the same entity that two schemas can store.

- Different Hardware/Software

  This conflict describes the fact that two information systems that are being integrated can have different hardware, operating systems, communications protocols, etc. Those differences can cause problems when integrating the two systems.

  In our opinion, this is not a cause of *semantic inconsistency* when integrating the information systems. This is a more concrete, low-level issue that has little to do with the semantics of the information systems.

The Taxonomy from [ND95] This work does not offer a full taxonomy of schema integration problems, but does discuss one problem omitted from the two taxonomies listed above: recognition of *object identity* across different data sources/schemas.

Different data sources may attempt to provide information about the same entity. Recognizing the instances where two or more data sources are in fact both describing the same entity can be problematic.

3.2 Representation of the Integrated Schema

The integrated schema will generally be represented in one of the following forms [LSS93].

3.2.1 Common Data Model

This is the design decision to choose a particular data model (such as relational or object-oriented) in which to provide access to data in the integrated system.

Common Data Model(CDM) vs. Homogeneous Descriptions A *homogeneous description* in an integrated system is that system's single, unified schema [JLYV00].

This design choice of whether or not to use a CDM must not be confused with the whether or not to use a homogeneous description for the integrated system.

The concepts are distinct. CDM only specifies that some particular (perhaps unspecified) data model (i.e., object-oriented, or relational) will be used to represent the integrated system. In contrast, a homogeneous description specifies not only the data model to be used, but also the particular schema to be provided by the integrated system.

CDM and homogeneous descriptions are similar, however, because higher-order logics are an alternative to each choice, as we will later see.

Integration Practices Associated with CDM The use of a CDM has traditionally been paired with the development of a homogeneous description for the integrated system in a one-time effort [JLYV00].

The implementation of integrated systems using CDM also have some association with the use of procedural languages, rather than declarative languages [CGL$^+$98].

3.2.2 Description Logics

Description Logics (DLs) are languages used to represent knowledge in a particular structured manner. A DL model uses the notions of *concepts* and *roles* to represent basic ideas about the world [CLN99].

Concepts are unary predicates that specify the subset of some domain. For example, a concept might be a the notion of “car”, “truck”, “automobile”, or “automobile dealership”. Each of those concepts is a definition which includes some objects but excludes others.

Roles are binary predicates that can be used to express relationships between concepts. For example, a role might be “for-sale-by”, that represents the binary relationship that can exist between a “car” and an “automobile dealership”. [CGL$^+$98] describes a DL that also explicitly models n-ary predicates.

Description Logics in Schema Integration DLs can be used by software to reason about the semantics of data for when provided with basic semantic information [Bor95]. This makes them a powerful tool in computer-assisted design of integrated schemas, because DL-based reasoning can make the humans designing the integrated system aware of certain relationships within and between schemas that they otherwise may have gone unnoticed.

The use of DLs for data integration advocated in [CGL$^+$98] uses a DL to not only model each data source, but also to express a model of a *global domain.* The global domain contains the set of concepts and roles that are used in the integrated view of the system.

DL reasoning systems use a set of *intermodel assertions* [CGL$^+$98] that humans can state. These are assertions, expressed in terms of the already-defined concepts and roles, express relationships between the concepts and roles of the data sources in the integrated system, and between the data sources and the global domain model of the integrated system.

DL-based systems can do lots of automatic reasoning as data sources are added to or removed from the integrated system. This automatic reasoning can reduce the effort invested and errors introduced by the humans designing the integrated system.

Schema characteristics that DLs can identify include [Bor95]:

- Coherency of a Concept

  Whether or not any element in a database could ever meet the requirements for inclusion in the concept.

- Subsumption of One Concept by Another

  Identifies which concepts will always have a superset/subset relationship.

- Mutual Disjointness of Two Concepts

  Identifies whether or not the same object could ever meet the requirements for membership in both concepts.

- Equivalence of Two Concepts

  Identifies whether or not two concepts that will always contain the exact same set of elements.

Ability to Represent Schemas from Various Data Models One reason that DLs are a useful tool in reasoning about schemas is that DLs meeting certain criteria are capable of representing the schemas of many popular data models, such as the entity-relation and object-oriented (sans the methods) models [Bor95].

3.2.3 Other Formalisms for Schema Integration

Description Logics are not the only languages that can be used to aid in schema integration. See [HG92] and [ND95] for examples of such formalisms.

3.3 Quality Metrics for Integration Schemas

Various quality metrics for integrated schemas have been proposed:

- **Accessibility** - All data needed from the data sources to provide the integrated view is in fact available from the present set of data sources [CGL$^+$98].
- **Believability** - Warranting confidence that the data provided by the integrated system and/or data sources is consistent (in the Description Logic sense) and complete [CGL$^+$98].
- **Completeness** [JLYV00]
- **Consistency** (in the Description Logic sense) of each data source [CGL$^+$98]
- **Correctness** [JLYV00]
- **Minimality** [JLYV00]
- **Understandability** [JLYV00]
- **Integration Transparency** - In systems that use a Common Data Model, this is the ability of the integrated system to provide views of itself that actually look like one of its constituent data sources [LSS93] [4].
- **Information Capacity** - The ability of an integrated schema to express all of the information that the data source schemas can express [EJ95].

[4] One might consider this to be a feature that is present or absent from an integrated system, rather than a metric that can be given various scores.

- **Readability** - The integrated schema makes clear to humans the important relationships that are implied by the integrated schema [CGL+98].

- **Redundancy** - The recognition of equivalent concepts [CGL+98]

3.4 Steps in Schema Integration

Some attention has been paid to the steps that humans, and their software tools, go through in the design of an integrated schema.

3.4.1 The Overall Schema Integration Process

No general consensus of what the steps are is clear from a survey of academic literature on the subject. Below are two different breakdowns that have been proposed.

From [BF94], we have:

1. Pre-integration

   This step involves:

   - translating the data source schemas into the integrated system's common data model, and
   - *semantic enrichment* [JLYV00] of the source schemas: recording additional semantic information about the schema in a semantic data model (such an entity-relationship model)

   This is done for two reasons:

   - Using one semantic data model for all data sources eliminates issues that arise from the data sources using different data models for their schemas.

     For example, suppose two car dealerships are integrating their customer databases. One dealership's database uses a relational schema, and the other users an object-oriented schema. When integrating the systems, both of those schemas can first be translated into a semantic data model, such as entity-relationship, so simplify reasoning about the integration.

   - The semantic data model can express the relationships between the data source's schema elements and the problem domain that could not be expressed by the data source schema's data model. Having a formal representation of the additional semantic information is helpful, and perhaps necessary, for producing a good integrated schema.

     Note that this additional information must be discovered by humans, since it may be simply absent from some data source schemas.

2. Comparison

   This is the analysis of the the collection of data sources being integrated, looking for relationships between the elements of the various schemas.

   This can be done at two levels: comparison of the schemas, and comparison of the actual data in the data sources. Statistical reasoning techniques, such a fuzzy logic, might be used in these steps to guess at the relationships.

3. Integration

   This is the construction of the integrated schema.

4. Schema Transformation

In contrast to [BF94], [JLYV00] offers the following sequence:

1. Pre-integration

   This includes an early planning phase for the integration project, including selection of the schemas to be integrated, and what order they will be integrated in.

   As with [BF94]'s *pre-integration* step, this step also includes semantic enrichment of the source schemas.

2. Schema Comparison

   This is the analysis of the collection of source schemas to look for correlations and conflicts between them.

   A partial list of conflicts that might be detected at this stage appears in Section 3.1.

3. Schema Conforming

   This is the modification of source schemas to make them more suited for integration with each other.

   This includes the resolution the conflicts that were detected in the schema comparison step, which still remains a partially manual step for humans.

   [JLYV00] suggests that there are other besides conflict resolution might lead to the modification of source schemas, but does not elaborate on what those reasons are.

4. Schema Merging and Restructuring

   This step is where the (conformed) source schemas are finally tied together to form the integrated schema.

   The resulting integrated schema can then be evaluated in terms of the quality metrics described in Section 3.3. The results of that quality analysis can lead to further iteration of the schema integration to improve the quality of the integrated schema.

3.4.2 Processes for Performing Incremental Integration Steps when Using Higher-Order Logics

[CGL$^+$98] describes the steps that can be taken when new data sources or new type of queries are introduced to an integrated system that is integrated using a higher-order logic (i.e., a description logic).

Source-Driven This is when a new data source is to be added to the integrated system. The steps to be taken are as follows.

1. Source Model construction

   The information in the new data source is expressed in terms of the higher-order logic used by the integrated system.

2. Source Model integration

   New intermodel assertions are recorded that relate the new data source to the other data sources and to the global domain model.

   Conflicts that are made apparent after these assertions are recorded are also dealt with at this step.

3. Quality Analysis

   This is the assessment of the quality of the integrated schema. The outcome of this assessment may lead to the repetition of some earlier steps in this sequence, or even in a reconsideration of the global domain model.

4. Source Schema specification

   Recall that description logics may be used only at design time to support the software tools that help humans to design the integrated schema and develop query plans.

   At runtime, the description logics may go unused, and a traditional schema (i.e., relational) must be used to access the data source.

   This step is the construction of a new view of the data source that:

   - is in a schema language usable by the system at runtime, and
   - offers a view of the data source that was designed during the earlier source model integration step.

5. Materialized View Schema restructuring[5]

   The new data source may have introduced new kinds of information to the integrated system. When the integrated system uses materialized views, those views may need to be restructured to be able to express the newly available information.

[5]Only applicable when the integrated system uses materialized views (see Section 6).

Client-Driven Integration This is when a query must be supported by the integrated system, but no execution plan has yet been formulated for that particular query.

To accommodate this event, humans can use software tools that reason about the integrated system's DL to determine whether or not the query can be answered using data source views that are already established.

See [CGL$^+$98] for more specific details on how the reasoning software can help when the integrated system uses materialized views.

3.5 Schema Integration Tools

3.5.1 Available Tools

Based on a survey of academic literature and on the author's familiarity with industrial solutions for data warehousing, the set of tools for assisting with schema integration appears to be largely academic.

An excellent overview of key academic systems for schema integration can be found in [JLYV00].

3.5.2 Benefits of Using Schema Integration Tools

Schema integration tools are good for performing a great deal of reasoning about an integrated system, as long as humans have provided the information that these systems need in an appropriate language.

In particular, the tools can reduce the required human effort needed to integrate schemas by:

- identifying and resolving some schema conflicts [Hal95]
- identifying relationships between the data that are stored in different sources that have different schemas [Hal95]
- optimizing the integrated schema in terms of consistency, redundancy, and type checking [Bor95]
- helping humans know how to rewrite newly-encountered queries [CGL$^+$98]
- determining whether or not existing data sources are capable of answering a query [CGL$^+$98]

3.6 The State of the Art

Schema integration is still an activity that involves humans, primarily at two steps:

- Schema enrichment of data sources

  This activity may involve research by people to add information about source schemas that was never recorded in the schema, or perhaps even in written documents.

- Conflict resolution

  When schema integration tools detect certain conflicts in how data sources and/or the global domain model express information, human judgement is currently needed to decide what to do about the problem.

A trend in research appears to be efforts to reduce the need for human involvement in the process. For the time being, however, schema integration can labor intensive.

4 Querying the Integrated Data

The main purpose of building data integration systems is to facilitate the access to the multitude of data sources. The ability to correctly and efficiently process the queries to the integrated data lies in the heart of the system. The traditional way of query processing involves the following basic steps:

1. getting a declarative query from the user and parsing it

2. passing it through a query optimizer which produces an efficient query execution plan that describes how to exactly evaluate the query, i.e., apply which operators, in what order, using what algorithm

3. executing the plan on the data physically stored on disk

The procedure described above also applies to query processing in data integration systems in general terms. However, the task is more challenging due to the complexities brought by the existence of multiple sources with differing characteristics. First of all, we need to decide which sources are relevant to the query and hence should participate in query evaluation. These chosen data sources will participate in the process by their own query processing mechanisms. Second, due to potential heterogeneity of the sources, there may exist various access methods and query interfaces to the sources. In addition to being heterogeneous, the sources are usually autonomous as well and therefore not all of the them may provide full query capability. Third, the sources might contain inter-related data. There may be both overlapping and inconsistent data. Overlapping data may lead to information redundancy and hence unnecessary computations during query evaluation. Especially in the case where there is a large number of sources and the probability of overlap is high, we may need to choose the most beneficial sources for query evaluation. The last but not the least, the sources may be incomplete in terms of their content. Therefore, it may be impossible to present a complete answer to user's query. This list of complications is extensible.

As discussed in Section 2, a data integration system may be built in two major ways: by defining a mediated schema on the participating data sources without actually storing any data at the integration system (virtual view approach) or by materializing the data defined by a unified schema at the integration system (materialized view approach). In both of the approaches, the user query is formulated in terms of the schema of the integrated system. However, in the latter approach, since the data is stored at the integration system according to the unified schema, query evaluation is no more difficult than traditional way of query processing. The major issue there, is the synchronization of the materialized data with the changes to the original data at the data sources, i.e., maintenance of the materialized views. We discuss this issue in Section 6. During maintenance, views defined on the data sources have to be processed on the data sources to re-materialize the updated data. In other words, query processing on the original data sources is realized usually at a different time than the user's query being processed on the materialized views. On the other hand, in the virtual view approach, every time a user asks a query, data source access is required. Therefore, query processing for the virtual approach includes the issues that would arise for the maintenance stages of the materialized view approach. In this regard, we discuss mainly the query processing problem for the virtual view approach in this section.

In the rest of this section, first we briefly discuss the modeling issues which forms the basis of all the following arguments. Then we present the main stages in query processing in data integration systems in order, namely, query reformulation, query optimization and query execution.

4.1 Data Modeling and Mapping

Traditionally, to build a database system, we first model the requirements of the application and design a schema to support the application. In a data integration system, rather than starting from scratch, we have a set of pre-existing data sources which would form the basis of the application. However, each of these data sources may have different data models and schemas. In other words, each source presents a partial view of the application in its own way of modeling. In fact, if we were to design a database system for the application starting from scratch, we would have another model, which would have the complete and ideal view of the world. To simulate this ideal, we need to design a unifying schema in a single data model based on the schemas of the data sources being integrated. Then each source needs to be mapped to relevant parts of this unified schema. This single schema of the integrated system is called the "mediated schema". Having a mediated schema facilitates the formulation of queries to the integrated system. The users simply pose queries in terms of the mediated schema, rather than directly in terms of the source schemas. Although this is very practical and effective in terms of transparency of the system to the user, it brings the problem of mapping the query in mediated schema to one or more queries in the schemas of the data sources.

Figure 1.5 shows the main stages in query processing in data integration systems. There is a global data model that represents the data integration system and each of the data sources has its own local data model. There are two conceptual translation steps: (i) from the mediated schema to exported source schemas, (ii) from exported source schemas to source schemas. The difference comes from the data models used. In the former one, the user query is reformulated as queries towards individual sources, but they are still in the global data model. In the latter one, source queries are translated into a form that is understandable and processable by the data sources directly, i.e., data model translation is achieved in this latter step. These two steps are performed by the mediator and the wrapper components in the system, respectively. In this section, we will be focusing on the operation of the mediator and the details of the wrapper will be presented in Section 5.

As Figure 1.5 indicates, in addition to modeling the mediated schema, we need to model the sources so that we can establish an association between the

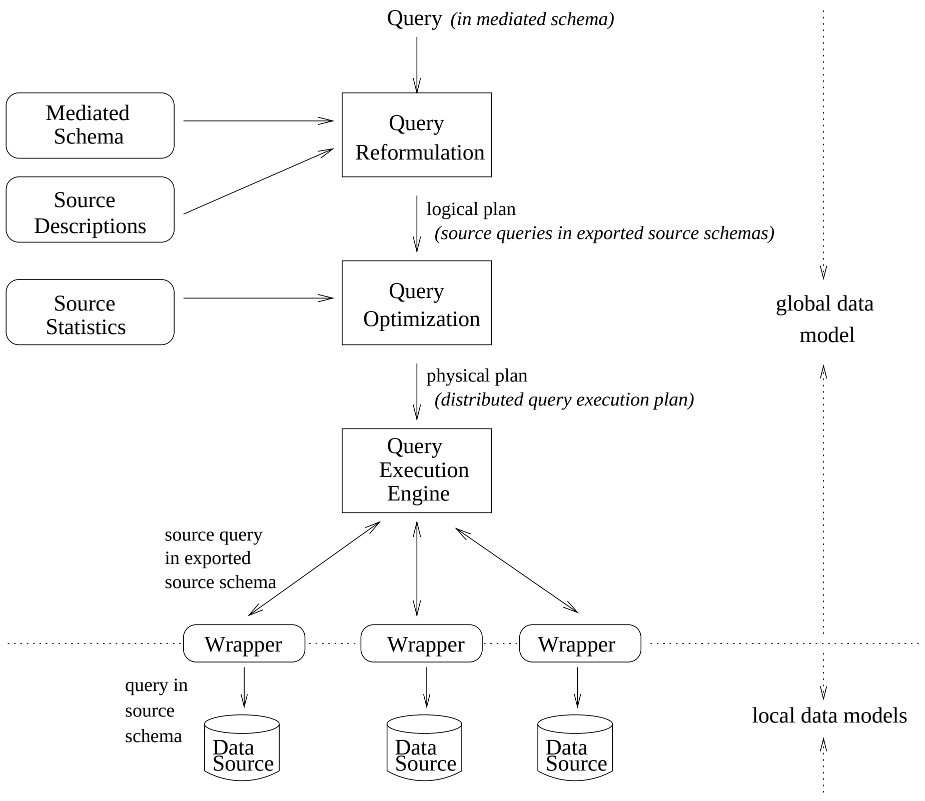

Figure 1.5: Stages of Query Processing [Lev99b]

relations in the mediated schema and the relations in the source schemas. This is achieved through *source descriptions*. The description of a source should specify its contents and constraints on its contents. Moreover, we need to know the query processing capabilities of the data sources. Because in general, information sources may permit only a subset of all possible queries over their schemas. Source capability descriptions include which inputs can be given to the source, minimum and maximum number of inputs allowed, possible outputs of the source, selections the source can apply and acceptable variable bindings [LRO96].

In Figure 1.5, first, using the mediated schema and the source descriptions, user query is reformulated into source queries in exported source schemas. An exported source schema refers to translated source schema in the global data model. These source queries provide a logical plan to the query optimizer which later produces a physical query execution plan using some source statistics. Afterwards, the physical plan is executed by the query execution engine through communicating with the data sources through their wrappers. Although it is not shown in this figure, the query execution engine later collects the results from the sources which are then combined for presentation to the user.

To be able to present the methods for querying the integrated data, we need to choose a data model and language to express the mediated schema, source descriptions and the queries. Due to its simplicity for illustrating the concepts, we will be using relational model as our global data model and Datalog as our language.

4.1.1 Datalog

We can express queries and views as datalog programs. A datalog program consists of a set of rules each having the form:

$$q(\bar{X}) : -r_1(\bar{X_1}), \ldots, r_n(\bar{X_n})$$

where q and $r_1, \ldots, r_n$ are predicate names and $\bar{X}, \bar{X_1}, \ldots, \bar{X_n}$ are either variables or constants. The atom $q(\bar{X})$ is called the *head* of the rule and the atoms $r_1(\bar{X_1}), \ldots, r_n(\bar{X_n})$ are called the *subgoals* in the *body* of the rule. It is assumed that each variable appearing in the head also appears somewhere in the body. That way, the rules are guaranteed to be *safe*, meaning that when we use a rule, we are not left with undefined variables in the head. The variables in $\bar{X}$ are universally quantified and all other variables are existentially quantified. Queries may also contain subgoals whose predicates are arithmetic comparisons. A variable that appears in such a comparison predicate must also appear in an ordinary subgoal so that it has a binding.

Predicates that represent relations stored in the database are called EDB (Extensional DataBase) predicates and predicates whose relation is constructed by the rules are called IDB (Intensional DataBase) predicates. In the above rule, q is an IDB predicate. If all r_i are EDB predicates, then we have a *conjunctive query*. A conjunctive query has the following semantics: We apply the rule for the query to the EDB relations by substituting values for the variables in the body of the rule. If a substitution makes all the subgoals true, then the same substitution applied to the head, is an inferred fact about the head predicate and the answer to the query [Ull97]. In this section, we will be considering conjunctive queries.

4.1.2 Modeling the Data Sources

To reformulate a query in mediated schema as a set of queries that are written in terms of the source schemas, we need the relationship between the relations in the mediated schema and the source relations. This is achieved through modeling the sources using source descriptions.

There are three approaches to describing the sources [Fri99]:

Global As View (GAV) Approach

For each relation R in the mediated schema, a view in terms of the source relations is written which specifies how to obtain R's tuples from the sources.

Example
The following simple example shows how mediated schema relations `CAR` and `REVIEW` can be obtained from the source relations `S1`, `S2` and `S3`.

```
S1(vin, status, model, year) ⇒ CAR(vin, status)
S2(vin, status, make, price) ⇒ CAR(vin, status)
S1(vin, status, model, year) ∧ S3(vin, review) ⇒ REVIEW(vin, review)
S2(vin, status, make, price) ∧ S3(vin, review) ⇒ REVIEW(vin, review)
```

This approach was taken in the TSIMMIS System [CGMH$^+$94].

Local As View (LAV) Approach
For each data source S, a view in terms of the mediated schema relations is written that describes which tuples of the mediated schema relations are found in S.

Example
In LAV, we take an opposite approach to GAV and we describe each source in terms of the mediated schema relations. Assume that source `S1` contains cars produced after 1990 and source `S2` contains cars sold by the dealer `"ACME"`.

```
S1(vin, status, model, year) :− CAR(vin, status),
                               MODEL(vin, model, year), year ≥ 1990
S2(vin, status, make, price) :− CAR(vin, status),
            MODEL(vin, make, year), SELLS(dealer_name, vin, price),
            dealer_name = "ACME"
S3(vin, review) :− REVIEW(vin, review)
```

Query processing using the LAV approach is an application of a much broader problem called "Answering Queries using Views". We will further discuss this problem in the next section.

One of the systems that used this approach was the Information Manifold System [KLSS95].

Description Logics (DL) Approach
Description Logics are languages designed for building schemas based on hierarchies of collections. In this approach, a domain model of the application domain is created. This model describes the classes of information in the domain and the relationships among them. All available information sources are defined in terms of this model. This is done by relating the concepts defining the information sources to appropriate concepts defining the integrated system. Queries to the integrated system are also asked in terms of this domain model. In other words, the model provides a language or terminology for accessing the sources.

DL approach is similar to LAV in that a view that describes each source is written except that views are formulated not in terms of a mediated schema, but on concepts and classes from the application domain model. Queries are also formulated in the same way.

This approach was taken in the SIMS System [AHK96].

Each of these approaches has certain advantages and disadvantages over the others [Lev99b]. The main advantage of GAV is that query reformulation in GAV is very easy. Since the relations in the mediated schema are defined in terms of the source relations, it is enough to unfold the definitions of the mediated schema relations. Another advantage is the reusability of views as if they were sources themselves to construct hierarchies of mediators as in the TSIMMIS System [CGMH$^+$94]. However, it is difficult to add a new source to the system. It requires that we consider the relationship between the new source and all the other sources and the mediated schema and then change the GAV rules accordingly. Query reformulation in LAV is more complex [6]. However, LAV has important advantages compared to GAV: adding new sources and specifying constraints in LAV are easier. To add a new source, all we need to do is describe that source in terms of the mediated schema through one or more views. We do not need to consider the other sources. Moreover, if we want to specify constraints on the sources, we simply add predicates to the source view definitions.

Compared to GAV and LAV approaches, DL approach has the benefit of presenting the user a richer domain model with hierarchical structures. Since the source relations and the mediated schema relations are parts of the same domain model, mapping between them is facilitated. However, DL by itself is not expressive enough to model arbitrary joins of relations [Lev99b]. As in LAV approach, adding new data sources is easy in DL approach. However, if the contents of the new source can not be completely mapped to the domain model, then the domain model has to be extended [AKS96].

4.1.3 Using Probabilistic Information

The source descriptions that have been mentioned up to now consider sources in isolation. However, the sources may be related. Moreover, they have the underlying assumption that sources are complete. For example, in a previous example, we considered that source `S1` contains cars produced after 1990. All the cars in `S1` are produced after 1990 for sure but we do not know whether all the cars produced after 1990 exist there. Therefore, in addition to the *qualitative* source descriptions as discussed in the previous subsection, we also need *quantitative* descriptions about the correlation and incompleteness of the sources [FKL97]. Qualitative descriptions allow us distinguish irrelevant sources. Quantitative descriptions help us distinguish among the relevant sources the ones which have higher probability to contain the answers.

[6]As we shall see in the next section, the most important work done on query reformulation focus on the LAV approach.

[FKL97] categorizes the quantitative information needed into three and presents how each can be specified using probabilities:

- coverage (completeness) of the sources
  It specifies the degree to which sources cover what their qualitative description suggest. This is done through specifying the probability of finding certain data items in the source. For instance, if `S1` is believed to cover 90% of all the cars produced after 1990, then this probability will be 0.9.
- overlap between parts of the mediated schema
  It specifies the degree of overlap between the parts of the mediated schema and hence indirectly the overlap between the data sources. For example, probability that a car is a Japanese car given that it is economic in gas may be assigned a value so that if we know that a car has low gas consumption, then we can infer that it is a Japanese car with some confidence.
- overlap between information sources
  This is to correlate the source contents. It can be derived from the other two categories or can be explicitly stated. For example, the probability that a car contained in `S1` is also contained in `S2` may be 0.9, which is approximately equivalent to saying that `S1` is a subset of `S2`.

This kind of probabilistic information can be very useful to optimize query processing. The sources that have higher probability of containing an answer to a query may be given priority in access. [VP98] also includes a similar study on using probabilistic information in data integration systems.

4.2 Query Reformulation

Using the source descriptions, a user query written in terms of the mediated schema is reformulated into a query that refers directly to the schemas of the sources (but still in the global data model). There are two important criteria to be met in query reformulation [Lev99a]:

- Semantic correctness of the reformulation: The answers obtained from the sources will be correct answers to the original query.
- Minimizing the source access: Sources that can not contribute any answer or partial answer to the query should not be accessed. In addition to avoiding access to redundant sources, we should reformulate the queries as specific as possible to each of the accessed sources to avoid redundant query evaluation.

In this section, we will mainly discuss query reformulation techniques for the LAV approach of source modeling. The reason for this is that query reformulation in LAV is not straightforward and also it is one of the applications of an important problem called "Answering Queries using Views". In what follows, first we briefly summarize this problem together with its other important applications. Then we present various query reformulation algorithms for LAV.

4.2.1 Answering Queries Using Views

Informally, the problem is defined as follows: Given a query Q over a database schema, and a set of view definitions $V_1, \ldots, V_n$ over the same schema, rewrite the query using the views as Q' such that the subgoals in Q' refer only to view predicates. If we can find such a rewriting of Q into Q', then to answer Q, it is enough that we answer Q' using the answers of the views [Lev00].

Interpreted in terms of the query reformulation problem for the LAV approach, this means the following: By using the views describing the sources in terms of the mediated schema, we can answer a user query written in terms of the same schema by rewriting the query as another query referring to the views rather than the mediated schema itself. Each view referred by the new query can be evaluated at the corresponding source this way. Basically we are decomposing the query into several subqueries each of which is referring to a single source.

Answering queries using views has many other important applications which include query optimization, database design, data warehouse design and semantic data caching [Lev00]. For example, query optimization may be achieved by using previously materialized views for answering a query in order to save from recomputation. We are discussing data warehouse design issues in Section 6.

The ideal rewriting we expect to find would be an "equivalent" rewriting. However, this may not always be possible. In data integration systems in particular, source incompleteness and limited source capability would lead to rewritings that approximate the original query. Among the many possible approximate rewritings, we need to find the "best" one. The technical term for this best rewriting is "maximally-contained" rewriting. The below definitions formalize these terms [Lev00]:

Query Containment and Equivalence A query Q' is contained in another query Q if, for all databases D, $Q'(D)$ is a subset of $Q(D)$. A query Q is equivalent another query Q' if Q' and Q are contained in one another.

Equivalent Rewritings Let Q be a query and $V = V_1, \ldots, V_m$ be a set of view definitions. The query Q' is an equivalent rewriting of Q using V if:

- Q' refers only to the views in V, and
- Q' is equivalent to Q.

Maximally-contained Rewritings Let Q be a query and $V = V_1, \ldots, V_m$ be a set of view definitions in a query language L. The query Q' is a maximally-contained rewriting of Q using V with respect to L if:

- Q' refers only to the views in V,
- Q' is contained in Q, and
- there is no rewriting Q_1 such that $Q' \subseteq Q_1 \subseteq Q$ and Q_1 is not equivalent to Q'.

4.2.2 Completeness and Complexity of Finding Query Rewritings

Theoretical issues related to the problem of finding query rewritings using views include completeness and complexity of the query rewriting algorithms. We will briefly touch on these issues here and we refer the interested readers to [Lev00] for a detailed discussion.

Completeness of a query rewriting algorithm is defined as follows in [Lev00]: Given a set of views $\mathcal{V}$ and a query Q, will the query rewriting algorithm always find a rewriting of Q using $\mathcal{V}$ if there exists such a rewriting? The answer to this question also depends on the query language used to express the query rewritings. Sometimes the limited expressiveness of the language may prevent the algorithm from finding a query rewriting although there exists one. In the case that no equivalent query rewriting exists, then we try to find a maximally-contained rewriting. [Lev00] also points out that sometimes we need to use recursive Datalog rules to be able to come up with a maximally-contained rewriting. This exemplifies the dependence of the algorithms on the expressiveness of the query language.

The complexity of the query rewriting algorithms can be discussed under different language and modeling assumptions. In general, they are in NP. Please refer to [Lev00] for a discussion of the specific cases.

4.2.3 Reformulation Algorithms

Given a query Q and a set of views $V_1 \dots V_n$, to rewrite Q in terms of V_is, we have to perform an exhaustive search among all possible conjunctions of m or less view atoms where m is the number of subgoals in the query. The following algorithms propose alternative ways of finding query rewritings to avoid the exhaustive search.

The Bucket Algorithm (Information Manifold)

The main idea underlying the Bucket Algorithm [Lev00] is that we can reduce the number of query rewritings that need to be considered if we consider each subgoal in the query separately to determine which views may be relevant to each subgoal. Given a query Q, the Bucket Algorithm finds a rewriting of Q in two steps:

1. The algorithm creates a bucket for each subgoal in Q which contains the views (i.e., data sources) that are relevant to answering that particular subgoal.
2. The algorithm tries to find query rewritings that are conjunctive queries, each consisting of one conjunct from every bucket. For each possible choice of element from each bucket, the algorithm checks whether the resulting conjunction is contained in the query Q or whether it can be made to be contained if additional predicates are added to the rewriting. If so, the rewriting is added to the answer. Hence, the result of the Bucket Algorithm is a union of conjunctive rewritings.

The following simple example shows how the algorithm works:

Example

Consider the car-dealer example we presented earlier. Assume that there are three data sources `S1`, `S2` and `S3`. `S1` contains information about cars produced after 1990. `S2` contains cars sold by the dealer named `"ACME"`. `S3` contains car reviews. Assume that we have the following relations in the mediated schema:

```
CAR(vin, status)
MODEL(vin, model, year)
SELLS(dealer_name, vin, price)
REVIEW(vin, review)
```

Furthermore, we have the following view definitions for the data sources:

```
S1(vin, status, model, year) :- CAR(vin, status),
                                MODEL(vin, model, year), year ≥ 1990
S2(vin, status, model, price) :- CAR(vin, status),
            MODEL(vin, model, year), SELLS(dealer_name, vin, price),
            dealer_name = "ACME"
S3(vin, review) :- REVIEW(vin, review)
```

Assume that we are looking for used cars produced before 1990, their reviews and where they are sold. We pose the following query to the mediated system:

```
Q(vin, dealer, review) :- CAR(vin, status), MODEL(vin, model, year),
                  SELLS(dealer_name, vin, price), REVIEW(vin, review),
                  year < 1990, status = "used"
```

We will use the initial letters of the fields for ease of presentation. The first step of the Bucket Algorithm constructs the following buckets per subgoal in `Q`:

CAR(V, S)	MODEL(V, M, Y)	SELLS(D, V, P)	REVIEW(V, R)
S2(V, S, M', P')	S2(V, S', M, P')	S2(V, S', M', P)	S3(V, R)

Notice how views are mapped to each query subgoal by the buckets. It is important to note that we did not insert `S1` into buckets `CAR(V, S)` and `MODEL(V, M, Y)` because of the constraint on the `year` attribute in the query. Since `S1` contains cars which are produced after 1990 and the query asks for the ones produced before 1990, `S1` can not answer the query.

The second step of the algorithm chooses one view from each bucket and combines them into a new query. Since for this simple example we have already one entry per bucket, there will be one combination of views. In general, we would have to construct one query per possible combination of the entries and we would test for containment in the original query. Then the result would be the union of all the contained queries.

We obtain the following new query written in terms of the view definitions rather than mediated schema relations:

```
Q'(vin, dealer, review) :− S2(vin, status, model, price),
                     S3(vin, review), year < 1990, status = "used"
```

Note that we eliminated two redundant references to view `S2` and we also added the extra constraints on the `year` and `status` attributes since without these predicates, `Q'` would not be contained in `Q`.

In terms of completeness and complexity, [Lev00] mentions that the Bucket Algorithm is guaranteed to find maximally-contained rewriting of a query if the query does not contain arithmetic comparison predicates. However, the second phase may take exponentially long.

The Inverse-Rules Algorithm (InfoMaster)

The key idea underlying this algorithm is to construct a set of rules that invert the view definitions, i.e., rules that show how to compute tuples for the mediated schema relations from tuples of the views [Lev00]. One can think of this process as obtaining GAV definitions out of LAV definitions. In other words, we are not actually rewriting the query, but we are rewriting the view definitions so that the original query can be easily answered on the rewritten rules.

One inverse rule is constructed for every subgoal in the body of the view. While inverting the view definitions, the existential variables that appear in the view definitions are mapped using Skolem functions to ensure that the value equivalences between the variables are not lost. The following example illustrates the algorithm:

Example
Consider the view definition for `S1` in the previous examples:

```
S1(vin, status, model, year) :− CAR(vin, status),
                               MODEL(vin, model, year), year ≥ 1990
```

Inverse-Rules Algorithm inverts this view definition by writing one inverse rule for every subgoal in the view definition as below:

```
CAR(f1(V, status, model, year), status) :− S1(V, status, model, year)
MODEL(f1(V, status, model, year), model, year) :−
                                           S1(V, status, model, year)
```

As illustrated above, the attribute `vin` is replaced by a skolem function f_1 which takes all the attributes of the view head as input. The attribute corresponding to `vin` is changed to a variable `V`. The reason that we treat `vin` as a special attribute is that it is shared between `CAR` and `MODEL` in the view definition. That is, a `vin` value in `CAR` should also exist in `MODEL` to take place in the view `S1`. f_1 makes sure that they are mapped to the same value.

The rewriting of a query Q using the set of views V is the datalog program that includes the inverse rules for V and the query Q. Below we show how a query is evaluated using these rules.

```
Q(vin) :− CAR(vin, status), MODEL(vin, model, 2000)
```

Assume that the source that is defined by `S1` contains the following data:

```
S1 = {(1, "used", "Honda", 2000), (2, "new", "Toyota", 2001),
      (3, "used", "Subaru", 2000)}
```

Then the algorithm would compute the following tuples:

```
{CAR(f1(1, "used", "Honda", 2000), "used"),
CAR(f1(2, "new", "Toyota", 2001), "new"),
CAR(f1(3, "used", "Subaru", 2000), "used"),
MODEL(f1(1, "used", "Honda", 2000), "Honda", 2000),
MODEL(f1(2, "new", "Toyota", 2001), "Toyota", 2001),
MODEL(f1(3, "used", "Subaru", 2000), "Subaru", 2000)}
```

When Q is evaluated on those tuples, we would obtain the answer {`1, 3`}.

In terms of completeness, this algorithm is guaranteed to find a maximally-contained rewriting in polynomial time in the size of the query and the views [Lev00].

Note that this example also illustrates how the rules in GAV approach can be used to evaluate the queries.

The MiniCon Algorithm

MiniCon Algorithm is an improved version of the Bucket Algorithm. As in the Bucket Algorithm, there are two steps: computing the buckets, one for each subgoal in the query, and then computing the rewritings using the buckets. Additionally, MiniCon Algorithm pays attention to the interaction of the variables in the query and in the view definitions to prune some of the views to be added into the buckets. This way, the number of views to be considered for the rewriting step is reduced, i.e., there will be less number of combinations to check.

The following example clarifies the algorithm.

Example

Consider the following view definitions and the query:

```
S1(vin, status, model, year) :− CAR(vin, status),
                                MODEL(vin, model, year)
S2(vin, status, model, price) :− CAR(vin, status),
           MODEL(vin, model, year), SELLS(dealer_name, vin, price)
S3(vin, review) :− REVIEW(vin, review)


Q(vin, dealer, review) :− CAR(vin, status), MODEL(vin, model, year),
                 SELLS(dealer_name, vin, price), REVIEW(vin, review)
```

The original Bucket Algorithm would put `S1` into buckets of `CAR` and `MODEL`. However, `S1` can not be used in the rewriting of Q for the following reason: Q requires join on `SELLS` and `REVIEW`. To use `S1`, either it should contain also `SELLS` and `REVIEW` or it should have appropriate variables in the head so that it can be joined with other views that contain `SELLS` and `REVIEW`. `S1` must have variable `vin` to be joined with `SELLS` and `REVIEW` on `vin` attributes of the other views. However, `S1`'s head only contains `model` and `year`. Therefore, we can not use `S1` in rewriting. There is no need to put it in the buckets of `CAR` and `MODEL` subgoals. `S2` will go into the buckets of `CAR`, `MODEL` and `SELLS` and `S3` will go into the bucket of `REVIEW`. The new query will be:

```
Q'(vin, dealer, review) :− S2(vin, status, model, price),
                           S3(vin, review)
```

Pruning `S1` in the bucket construction step, we need to check less number of view combinations to rewrite Q' that is contained in Q.

For a detailed discussion on the MiniCon Algorithm, please see [PL00].

The Shared-Variable-Bucket Algorithm

This algorithm, like the MiniCon Algorithm, also aims at recovering the weak aspects of the Bucket Algorithm to obtain a more efficient algorithm. The idea is again to examine the shared variables and reduce the bucket contents so that the number of view combinations to be considered is reduced at the second phase of the algorithm. We are not describing this algorithm in detail. Interested readers should see [Mit99].

The CoreCover Algorithm

In this algorithm, closed-world assumption is taken where views are materialized from base/source relations [ALU01]. Among the possibly infinite number of rewritings, the aim is to find the ones that are guaranteed to produce an optimal physical plan if there exists any. Since the rewriting aims towards query optimization, different cost models are also considered in this algorithm. One particular difference of this algorithm is that, contrary to the previous algorithms, this algorithm aims at finding equivalent rewritings rather than contained rewritings. We will again redirect the interested readers to [ALU01] for a better discussion of the CoreCover Algorithm.

Comparison of the Algorithms

The CoreCover Algorithm is quite different than the other algorithms. First, all the other algorithms aim at finding a maximally-contained rewriting of the query whereas the goal of the CoreCover Algorithm is to find an equivalent rewriting. Second, closed-world assumption is taken which enables the algorithm to find an equivalent rewriting. Third, reformulation stage of query processing is like integrated with the optimization stage since the rewriting has to guarantee an optimal plan for the query.

Of the remaining four algorithms, Bucket, MiniCon and Shared-Variable-Bucket Algorithms share the same spirit in that buckets are constructed and then cartesian product of the buckets are taken to produce the rewritings. The deficiency of the Bucket Algorithm is that the constructed buckets are unnecessarily large and this causes a lot of combinations to be computed and tested for the second phase. MiniCon and Shared-Variable-Bucket Algorithms use a very close approach to prevent this deficiency. The MiniCon Algorithm has been shown to outperform both the Bucket and the Inverse-Rules Algorithms [PL00].

Finally, the Inverse-Rules Algorithm has the advantage that it is query-independent. That is, the rules are computed once and then they apply to any query afterwards. Also, the rules are easy to extend for additional constraints to be added to the system like functional dependencies [Lev00]. On the other hand, the rewriting obtained by the Inverse-Rules Algorithm may contain views that are not relevant to the query because this algorithm ignores the predicates that impose constraints on the variables. An additional phase which removes the irrelevant views may be added to the algorithm, but this is shown to be very inefficient [Lev99b]. Also, Inverse-Rules Algorithm may have to consider a large number of rule unfoldings during query evaluation.

4.3 Query Optimization and Execution

Query optimization refers to the process of translating a declarative query into an efficient query execution plan, i.e., a specific sequence of steps that the query execution engine should follow to evaluate the query. In addition to the operators and their application order specified in the query execution plan, the optimizer should also decide on the specific algorithms that implement the operators and which indices to use with them. There may be many possible execution plans. The best execution plan can be chosen in two ways: cost-based or heuristics-based. In the cost-based approach, the optimizer has to estimate the costs of candidate plans and choose the cheapest of them. Cost estimations are done using statistical information about the underlying data such as sizes of the relations and the selectivity of predicates. Heuristics-based plan generation involves using some rules of thumb like doing selections before joins. Usually heuristics-based technique is easier and cheaper than the cost-based one, because it does not need to consider and evaluate the cost of all possible plans. However, the optimal plan is not guaranteed.

As discussed in the previous section, query reformulation step already provides some optimizations on the query by pruning irrelevant sources and distinguishing the overlapping sources to avoid redundant computation. Furthermore, the rewritten queries are to be as specific as possible. However, these are logical or higher level optimizations. There are still many optimizations to be done when it comes to actually executing the logical plan generated by the reformulator physically on the data.

Query optimization in data integration systems is more difficult than the optimization problem in traditional databases because:

- Sources are autonomous. Optimizer may not have any statistics or either has few or unreliable statistics about the data stored in each of the sources.

- Sources are heterogeneous. They may have different query processing capabilities. The optimizer needs to exploit these capabilities as much as it can. In addition to what kind of queries the sources can process and how they can process them, it is also relevant that what kind of processing power they have underlying their data management system and performance changes due to workload changes.

- In traditional databases, it is easy to estimate the data transfer time since it is between the local disk and the main memory. In data integration systems however, data transfer time is not predictable due to the existence of the network environment. Both delays and bursts may occur.

- On one hand, the sources are overlapping and there is redundancy for most of the time. That is why access to redundant sources should be minimized. On the other hand, some sources may become unavailable without any notice. Query optimizer should be able to handle these cases flexibly by replacing overlapping sources for each other to compensate for unavailability of any of them.

An additional problem that may cause inefficient query execution is that the logical plan produced by the reformulator tends to have a lot of disjunctions, i.e., union operations.

The bottom line is that it is difficult to decide statically what the optimum strategy would be to execute a query due to insufficient information and dynamicity of the environment. Therefore, the traditional approach of first generating a query execution plan and then executing it is no more applicable. [IFF$^+$99] proposes an adaptive query execution approach in which query optimization and execution are interleaved. In the rest of this section we mainly discuss this approach.

4.3.1 Adaptive Query Execution*

[*Note: * refers to that this section is an advanced section and optional to the reader.*]

In addition to the above listed problems, [IFF$^+$99] makes the following observations about query optimization in data integration systems:

- It is more important to aim at minimizing the time to get the first answers to the query rather than trying to minimize the total amount of work to be done to execute the whole query.

- Usually the amount of data coming from the data sources is smaller compared to case of querying a single source as in traditional database systems.

Adaptivity in [IFF$^+$99] exists in two levels:

- interleaved planning and execution

- adaptive operators for execution engine

At a higher level, the former is achieved by creating partial plans called fragments rather than complete plans. The optimizer decides how to proceed next only after executing a fragment. Once a fragment is completed, the optimizer would know more about the sources and the environment so that it could do better planning for the rest of the query.

The latter includes using new operators during execution depending on the observations listed above. Two important operators used in the Tukwila System described in [IFF$^+$99] are double-pipelined hash join and the collector operator.

Double-pipelined hash join is a join implementation that allows Tukwila to quickly return the first answers to the query in spite of the fact that some sources may be responding very slowly. In contrary to the conventional hash join where smaller of the two relations to be joined is chosen as the inner relation to hash by the join attribute, in double-pipelined hash join, both relations are hashed. This way, result tuples are produced as soon as the data from sources arrive. This masks the slow data transmission rates of some sources. The optimizer no longer has to make a decision about which relation should be the inner one (Normally, it would have to know the size of the relations to be able to choose the smaller one as the inner). Also, the processing is not blocked due to delays at the sources.

The *collector operator* is used to facilitate union over large number of overlapping sources. Using the estimates about the overlap relationships between the sources and depending on the run-time behavior of the sources (delays, errors) optimizer adapts its policy about how the unions should be performed and the collector operator achieves the application of this dynamic policy. Policies are specified using rules.

Both levels of adaptivity are realized through *event-condition-action rules.* Events are raised by execution of the operators or completion of some fragments and obtaining some partial results. When an event triggers a rule, first the associated condition is checked. If it is true, then the defined action is executed. Possible actions include reordering of operators, re-optimization, changing the policy of the collector operator and so on. The rules accompany the operator tree generated by the optimizer. They specify how to modify the implementation of some operators (for example, the collector) during run-time if needed and conditions to check at points where fragments complete in order to detect opportunities for re-optimization.

4.3.2 Query Translation

One thing we have treated as a black box until now is how actually the source queries in exported schemas (in schema of the sources but in the global data model) are translated into their actual schemas (in their local data models) and then get executed by their native query processors. This step is called the query translation step. It is achieved by the source-specific wrappers. Data extraction from sources by the wrappers is the topic of the next section.

5 Techniques for Extracting Data: Wrappers

Data extraction deals with the issues that arise during the process of getting data from the different sources to the integration system. It combines techniques from the areas of database systems and artificial intelligence (such as natural language processing and machine learning). In this section, first we discuss *wrappers* that, as we have seen in the previous sections, are important components of a data integration system. Then we review some work on tools for semi-automatic and automatic wrapper generation.

During information integration from heterogeneous data sources, we have to translate queries and data from one data model to another and from one data schema to another. As we mentioned in Section 2, this is done by wrappers that are written for each data source in the integration system. Each wrapper transforms queries in the unified schema to the queries in the format of the the underlying data source and then translates the results back to the unified schema. We would like to note that mediator systems usually require more complex wrappers than do most warehouse systems.

5.1 Wrapper Generation Approaches

Wrapper designers can either construct the wrappers manually, or use some tools facilitating the wrapper code development. Three approaches are usually considered:

- **Manual**
  Hard-coded wrappers are often tedious to create and may be impractical for some sources. For example, in case of web sources: the number of them can be very big, new sources are added frequently, and both the structure and the content of any source may change [AK97]. All these factors lead to the high maintenance costs of manually generated wrappers.

- **Semi-automatic (interactive)**
  It was noted in [HGMN$^+$97] that the part of a wrapper code which deals with the details specific to a particular source is often small. The other part is either the same among wrappers or can be generated semi-automatically based on the declarative description given by a user. Techniques such as *programming by example* can be used for this purpose.

- **Automatic**
  Automatic generation means that there is no human involvement. Tools for automatic wrapper generation can be site-specific or generic. They usually need training in the initial stage and are based on the supervised learning algorithms.

5.2 Tools for Semi-automatic/Automatic Wrapper Construction

Here we review several techniques for semi-automatic and automatic wrapper generation for structured/semistructured data . Most of them are designed for the case of web sources. As we mentioned above, writing the wrapper code for web sources may be especially hard due to the frequent changes of content and structure of the sources. On the other hand, data on the web often has a partial structure. This fact allows us to develop tools for automatically extracting this data.

5.2.1 Using Formatting Information in the Semistructured Pages on the Web

HTML documents often have some internal hierarchy of information, but this hierarchy is not specified explicitly. For example, a site of a travel agency may have information about several countries and hotels in the semistructured format. Some records, such as “a capital”, “money units”, “a language” will appear for all countries, while some others like “states” are country-specific. The presence of a partial structure in many web sources gives an integration system designer an opportunity to generate wrappers for the sources of a particular domain semi-automatically. Often this “semistructured” information may come to the web from the databases underlying the web sources. This raises the question “Why could not we query these databases directly in this case?”. Unfortunately, for a number of subjective reasons, a source may not set permissions for the outside users to query it.

The approach we describe was proposed by [AK97] and is used for semi-automatically generating wrappers for both *multiple-instance sources* and *single-instance sources*. A multiple-instance source contains information on several pages that are all of the same format. An example is CNN’s weather pages[7]: pages for all cities have the same structure (for instance, there is always a `Current Conditions` section with temperature, humidity and wind specified). Wrapper must be able to answer queries about all sections of the individual page. A single-instance source contains a single page with semi-structured information.

The authors identify three steps of a wrapper generation process for the types of sources mentioned above: “structuring the source; building a parser; adding communication capabilities between sources, a wrapper, and a mediator” [AK97] .

1. Structuring the source

   The first step refers to the finding heading tokens on a page, such as “Current Condition”, “Temperature”, “Wind”, and organizing them in a hierarchy tree. Such sections are usually stressed in the document by the size of font (big), the type of font (bold, italic), by noticing a colon following such a token, etc. All these simple heuristics, used by the authors,

[7] *http://www.cnn.com/WEATHER/*

proved to work well for the domains they specified.
After a system has suggested the set of headings, a user may interfere by correcting the output. The hierarchy of the found headings is determined based on indentation spaces and font size. The grammar describing the structure of pages of a web source is produced as the result of this step. Results published by the authors show that usually just few corrections made by the user are needed for a web source.

2. Building a parser

   A parser for extracting any structured portion of data can be generated automatically given the output grammar of the first step.

3. Adding communication capabilities

   First, a wrapper needs some mechanisms to fetch the appropriate pages from the sources. In the case of a single page for each source, it is not a problem as long as URL of this page is known. In the case of multiple pages for a source, we need to map a query to the URL or set of URLs. In the case of the CNN weather site, for example, we can specify that for a given state in the USA and a city in it, the URL of the page containing the weather forecast can be obtained by adding the following to the end of the "*http://www.cnn.com/WEATHER/*" string:
   - the abbreviation of the region (for instance, *ne* stands for the north east);
   - the abbreviation of the state;
   - the name of the city and a 3-letter city abbreviation.
   For example, the URL for Providence, RI is
   http://www.cnn.com/WEATHER/ne/RI/ProvidencePWD.html.
   Second, a wrapper relies on some protocols to deliver data over the network. Authors of the paper [AK97] were using Perl scripts.
   Third, a wrapper and a mediator need to communicate between themselves in the integrated system. In the reviewed system [AK97], KQML (Knowledge Query and Manipulation Language) was used for this.

5.2.2 Template-based Wrappers

The approach proposed by Hammer *et. al.* [HGMN+97] is applicable to several types of data sources: relational databases, legacy systems, and web sources. Their wrapper implementation toolkit is based on the idea of *template-based translation*. A designer of a wrapper uses a declarative language to define the rules (templates) which specify the types of queries handled by a particular source. For each rule he also defines an action to be taken in case a query sent by the mediator of an integration system matches the rule. This action will cause a *native query* - a query in the format of the underlying source - to be executed.

Filter queries are used to extend the set of queries a source can handle. If a source does not support some predicates, the query will be turned into two queries: the native query (that will contain only those predicates that are supported by the source) and the filtered query that will “postprocess” the results of the native query.

The process of query transformation consists of the following steps. First, a query from the mediator is parsed, then it is matched against the templates in the system. If the matching rule was found, the native query is processed by the data source, and the result is filtered with the filter query by the wrapper.

A rule-based language MSL [PGMA96] is used by the authors for query formulation. Below we give an example of an MSL query, a template matching it, and the corresponding native and filter queries. For the purpose of the example (that is based on the example presented in [HGMN^{+}97]), the data source is a relational database.

Example

Let us refer again to the example of Acme Cars company that has a relation

```
Automobiles(vin, autoMake, autoModel, autoPrice, autoYear).
```

This relational database consisting of just one relation, is our data source. We need to write a wrapper supporting MSL queries to this source. We further assume, that the source does not support comparison predicates on the `autoPrice` attribute. Let a user A ask the MSL query about all Honda cars for sale whose price is less than 12,000$:

```
C :-- C:<Automobiles
{<autoMake "Honda"><autoPrice P>}>
AND LessThan(P, 12,000)
```

One of the templates, matching this query is:

```
C :-- C:<Automobiles{<autoMake $A >}>
```

Notice, that the result of this template query is a superset of the results asked by the user query. The action corresponding to this template is to select all automobiles with the `autoMake = $A`. In the system, $A is substituted with `"Honda"` and a native SQL query for the relational database is produced:

```
SELECT *
FROM Automobiles
WHERE autoMake = "Honda"
```

After the wrapper received an answer to this SQL query from the source, the only thing remained, is to postprocess the results of this query using the following filter query:

```
C :-- C:<Automobiles{<autoPrice P>}>
AND LessThan(P, 12,000)
```

This will only leave those `"Honda"` cars, whose price is less than `12,000$`. After that, the result can be returned back to the mediator of the integration system.

5.2.3 Inductive Learning Techniques for Automatically Learning a Wrapper

These techniques are sometimes called *wrapper induction* techniques [Eik99] and are based on inductive learning. According to [Eik99], *inductive learning* is the task of computing a generalization from a set of examples of some unknown concept. This generalization should suggest the model explaining all of the examples.

A very simple example of an inductive inference is when a teacher says a sequence of numbers: 2, 4, 6, 8, 10; and then asks a pupil to guess the rule he used to produce the next number from the previous (in this case $p_{n+1} = p_n + 2$).

[Eik99] points out the following classification of the inductive learning methods used for wrapper induction:

- Zero-order learning
  They are also called *decision tree learners* as their solutions are represented as decision trees. The drawback of these methods is coming from the fact that they are based on the propositional logic that has a number of limitations. For example, they can not deal with several relations in a relational database [Eik99].

- First-order learning
  Methods of this type can deal with first-order logical predicates.
  Inductive logic programming is a method of this class, widely used due to the ability to deal with complex structures such as recursion. Two approaches - *bottom-up* and *top-down* - are often used as a part of the first-order learning.

  The bottom-up approach first suggests a generalization based on few examples. Then this generalized model is corrected based on the other examples.
  The top-down approach starts with a very general hypothesis and then distills it learning from negative examples.

Some known systems for inductive learning of wrappers are STALKER [MMK98] and the system described by Kushmerick *et al.* [KWD97]. An excellent overview of some other systems for information extraction by inductive learning is given in [Eik99].

Here we discuss the system developed by Kushmerick *et al.* [KWD97]. This approach is suited for the text sources (HTML-pages) with a tabulated structure and with the following delimiters: head, right, left and tail.

Example
Let us consider an HTML page with the list of pupils in the class with the corresponding GPA (this example is based on the example in [KWD97]). For the simplicity, we assume that each pupil is identified by only the second name.

```
<HTML>
<TITLE>Current GPA of the students</TITLE>
<BODY><B>Current GPA of the students</B><P>
<B>Simpson</B>  <I>2.72</I><BR>
<B>Johnson</B>  <I>3.5</I><BR>
<B>Peterson</B> <I>4.0</I><BR>
<HR><B>End</B>
</BODY></HTML>
```

`<B>` and `</B>` (as well as `<I>` and `</I>`) are called left and right delimiters and separate the data on the HTML page. However, the first and the last strings also contain these delimiters; so in order to distinguish between tuples and heading/ending of the HTML page, a head tag `<P>` and a tail tag `<HR>` can be used [KWD97]. This set of delimiters makes the job of a wrapper simple: first skip a string with `<P>` in the end; then, till the marker of the end is reached, fill out tuples with the data surrounded by `(<B>, </B>)` and `(<I>, </I>)`.

The algorithm the authors developed is used for automatic generation of wrappers for HTML-sites with such structure. Missing data is not allowed, and the order is also strict. They call this class of wrappers *HLRT-wrappers* (head, left, right and tail).

First, a number of HTML pages is labeled. Labeling in this case means the specification of the tuples, contained in the HTML page. In the example above, we would label that HTML page with {(Simpson, 2.72), (Johnson, 3.5), (Peterson, 4.0)}. The hypothesis in this case is a set of tags used to separate each attribute (in the example, the wrapper should learn that `<B>` and `</B>` are used to separate family names; `<I>` and `</I>` - GPA-s. Labeling by hand is pretty laborious, so the authors described how a set of heuristics (domain-specific) can be used as an input for a labeling algorithm to semi-automate it.

Induction algorithm learns from these labeled data. Iteratively, the algorithm constructs a set of delimiters consistent with the labeled pages. It is done by considering possible combinations of tags present in the pages till the consistent set of tags is found. The question is "How many examples are enough to conclude that the set of delimiters that is consistent with all examples so far, will be consistent with the remaining examples?". More formally, we need to know how many examples are enough to say that with probability ϵ we learned a wrapper correctly with confidence δ. The authors provide the formulas they used to estimate it.

6 Materialized View Management

A view defines a derived relation from a set of database relations. It is actually a query whose result is given a name that can be used like other ordinary relation names stored in the database. When the tuples of the virtual relation defined by a view are physically stored in a database, we call such a view *materialized view*.

Use of materialized views dates back to early 1980s [GM99]. They were first proposed to be used as a tool to speed up queries on views. Then they were also used to maintain integrity constraints and to detect rule violations in active databases. They gained serious reconsideration by the emergence of new applications like data warehousing. In this section, we discuss issues related to use of materialized views in data integration systems.

We presented the materialized view approach to data integration in Section 2. The term comes from the fact that a set of views are derived from the data sources and the answers to those views are actually stored in a repository called a *data warehouse*. The main purpose of this pre-computation is to improve query response time. None of the complex query processing steps described in Section 4 is needed to be able to answer a user query in data warehouses at the time the query is asked. Those steps are completed and the results are collected in advance of the queries. This not only provides the ability to answer many queries very quickly, but also increases the availability of the system since the warehouse continues to answer queries even if the underlying data sources may become inaccessible for some reason at the time of querying.

Of course the benefits mentioned above do not come for free. First, the views to be materialized need to be determined. Usually it is both costly and redundant to materialize all the derived relations defined by the views which constitute the unified schema of the integrated system. The most beneficial views need to be selected based on criteria like frequently asked queries. Second and more importantly, the views that are selected to be materialized need to be *maintained*. View maintenance refers to the process of synchronizing derived data stored at the warehouse with the updates on the *base data*, i.e. data stored at the underlying data sources. The naive way of maintaining views would be to re-materialize the views when a relevant update occurs. However, this is not desirable for good performance.

In this section, we first present the materialized view selection problem together with the proposed solutions. Then we discuss various approaches for maintaining the selected views efficiently.

6.1 Design and Selection of Views to Materialize

The problem of materialized view selection can be defined as follows: Given a set of queries to the integrated system with their access frequencies and a set of source relations with their update frequencies, find a set of views to be materialized such that the total query response time (i.e. query processing time) and the cost of maintaining the selected views are minimized [YKL97]. There may also be other resource constraints to be considered such as disk space, but the most important of all is the maintenance cost/time.

Previous research on this problem has concentrated on Multiple-Query Optimization (MQO) techniques. MQO is the problem of finding an optimal query execution plan for evaluating a set of queries simultaneously. Techniques used involve identifying the common subexpressions among queries, executing those once and reusing later. In general, there may be many possible plans for each query and there may also be many possible ways of combining them. Thus, the search space is really large. Two general approaches are: (i) producing local optimal plans for each query and then merging them, which does not guarantee an optimal solution, and (ii) generating a globally optimal plan, which has a larger search space. [SG90] has proven that MQO problem is NP-complete. Proposed solutions usually make use of heuristics to find a solution as close as possible to the optimal solution. The related work on materialized view selection follows a similar path.

[YKL97] proposes a method where a Multiple View Processing Plan (MVPP) is constructed from the set of queries. Then some parts of this plan are selected to be materialized. The cost comparisons are based on the following cost measures: Cost of a query is the number of rows in the table used to construct that query. Cost of query processing is the frequency of the query multiplied by cost of query access from materialized nodes. Cost of view maintenance is equal to the cost of constructing the view, i.e. re-materialization is assumed. Total cost is equal to the sum of the cost of query processing and the cost of view maintenance.

There are two stages to view selection:

1. finding a good MVPP
   MVPP is the global query execution plan in which local execution plans for individual queries are merged based on shared operations on common sets [YKL97]. There are two ways of finding the MVPP:

   - merging local optimal query plans
     Local optimal plans are computed for each query. Then the queries are ordered in a descending fashion based on their query processing costs multiplied by their access frequencies. If there are k queries, k MVPPs are constructed as follows:

     ```
     for i=1 to k do
         take the ith local query plan and
         incorporate all the others to it in order
     ```

     The view selection algorithm at stage 2 will be run on these k MVPPs and then the least costly one will be chosen. This approach takes linear time in terms of the number of queries.

   - generating a globally optimal plan
     Rather than the locally optimal plans, all possible plans for each query are considered. The problem is mapped to a 0-1 integer linear

programming problem which is stated as follows: Select a subset of the join plan trees such that all queries can be executed and the total query processing cost is minimum [YKL97]. Then the set of join trees found are used to construct the MVPP. Solution to the linear programming problem is the optimal solution. However, solving it is exponential in the number of queries. Therefore, usually near-optimal solution is found.

2. selecting views to materialize from the MVPP
   An execution tree is built for the given MVPP whose nodes correspond to intermediate results to the queries. We can simply choose the complete tree or all the leaf nodes for materialization. These correspond to materializing all the queries and all the base relations, respectively. However, the aim is to find a set of intermediate nodes to materialize such that the total cost for query processing and view maintenance is minimized. The brute force way of finding this set is to compare the cost of every possible combination of nodes. This is not efficient. Some heuristics should be used. The algorithm presented in [YKL97] is based on the following idea: Whenever a new node is considered to be materialized, we calculate the saving it brings in accessing all the queries involved, subtracting the cost for maintaining this node. If the value is positive, then this node will be materialized and added into the solution set.

A somewhat similar approach is presented in [Gup97] which is based on using greedy heuristics and AND-OR graphs. An AND-OR graph represents a set of query plans. AND-OR graphs of the queries are merged to obtain an AND-OR view graph. Each node in the AND-OR view graph represents a view that could be selected for materialization. The problem is to choose among the nodes of the AND-OR view graph such that sum of total query response time and total maintenance time is minimized. [Gup97] states that the minimum set cover problem can be reduced to this problem and it is NP-hard. A near-optimal algorithm is presented which uses greedy heuristics. The set of the selected views has a benefit and at each step views that would increase the benefit of this set would be added to the set. Special cases of AND view graphs, OR view graphs, view graphs with indices are also investigated in [Gup97].

[RSS96] and [MRRS00], which mainly focus on the view maintenance problem, indirectly cover some methods that are applicable to view selection. [RSS96] proposes to augment a given set of materialized views with an additional set of views that may reduce the total maintenance cost. The selection problem here is to determine the additional views. [MRRS00] applies MQO techniques both to view selection and maintenance. Selection comes into play where additional views are to be materialized temporarily for efficient maintenance. The claim is that the same techniques are also applicable to selection of permanent views to materialize.

There are also research studies in materialized view selection for the special case of data cubes [HRU96] and multidimensional datasets [SDN98] in OLAP. We do not present them in detail here.

6.2 The Problem of View Maintenance

Materialized views are derived from data originally stored at multiple data sources. As primary copies of data at the data sources get updated, materialized views become stale or inconsistent with the underlying data. We call the process of bringing the materialized views up-to-date with the changes in the underlying data *view maintenance*. A materialized view can always be brought up-to-date by re-evaluating the view definition. However, recomputing the views every time the base data changes is not very efficient. Besides, [GM95] points out that in general only a part of the view changes in response to changes in the base relations, which is called the *heuristic of inertia*. Thus, only the parts of the views that are affected from the changes need to be computed and updated. This is called *incremental view maintenance*. The following exemplifies incremental view maintenance.

Example

Consider the following base relation stored at some data source:

`Cars(vin, status, model, year, price)`

Assume that the following view defined on `Cars` is materialized at the data warehouse:

```
CheapCars(vin, price) :- Cars(vin, status, model, year, price),
                         price ≤ 3000
```

When a new car tuple <`471, "used", "Mazda", 1992, 2500`> is added into the `Cars` table at the data source, `CheapCars` view needs to be updated. The only modification needed is addition of the tuple <`471, 2500`>. The whole view need not be recomputed. When another car tuple <`839, "used", "LandRover", 1996, 15000`> is added, however, `CheapCars` does not need any modification because the new tuple does not satisfy the view definition.

In this subsection, we present the dimensions of the problem and alternative policies for view maintenance. Incremental view maintenance techniques will be discussed in the following subsections.

6.2.1 Dimensions of the Problem

The following parameters determine the complexity of the view maintenance problem [GM99]:

- Available Information
  It refers to the amount of information available to the view maintenance algorithm. The view definition and the actual update occurred on base data have to be known to the algorithm. In addition to that, information like the content of the materialized views, the contents of the base relations, the definitions of other views and integrity constraints at the data sources might also be accessible to the algorithm. Depending on how much information is available, the task of view maintenance might be facilitated. For example, if we knew that a certain attribute is a key at the underlying data source, then we would also know that every insertion would have a different value for that attribute. Hence, an insertion at the

source would require an insertion at the materialized view that refers to that attribute.

- Allowable Modifications
  It determines what modifications can be handled by the view maintenance algorithm given other parameters. These might include insertions, deletions, updates, group updates, etc.

- Expressiveness of the View Definition Language
  View definition language may also facilitate or complicate the task of view maintenance. Views might be defined at various levels of expressiveness through languages including conjunctive queries, aggregation, recursion, negation, etc.

- Database and Modification Instance
  Current contents of the data sources or the materialized views and the modification may also determine the capabilities of the maintenance algorithm.

- Complexity
  A dimension that is somewhat at a different level than the others is the complexity dimension which refers to the efficiency of the view maintenance algorithm. Complexity can be measured in multiple sub-dimensions including complexity of view maintenance language, view maintenance algorithm or amount of extra information needed.

6.2.2 View Maintenance Policies

There are two main steps in materialized view maintenance: *propagate* and *refresh* [GM99]. Propagate step involves computing changes to be done on the materialized views upon changes to the base data and in refresh step the computed changes are actually applied on the materialized views. Propagate step always precedes the refresh step. The decision of *when* to perform the refresh step is called a *view maintenance policy.* Maintenance policies can be categorized as follows:

- Immediate View Maintenance
  Refreshing is done within the transaction that changes the base data. The advantages of this policy are that queries are processed fast and always return up-to-date results. The reason for this is that materialized views are brought up-to-date in advance of the queries. On the other hand, this policy slows down the transactions at the data sources since propagation and refreshing are to be done in the transaction's scope. Besides, this policy may not always be applicable when the data sources are fully autonomous and their commit decisions can not be delayed by the integrated system.

- Deferred View Maintenance
  Refreshing on the views is done later than the transaction that changes

the base data. Log of changes to the base data are to be kept. This policy allows *batch updates* by applying all the changes collected in the log to the views at the same time. There are three deferred view maintenance policies:

- Views are refreshed lazily at query time. It is guaranteed that query answers will be consistent with the base data and this is done without slowing down the transactions at the sources. However, queries to the integrated system are processed more slowly.
- Changes to views are forced after a certain amount of change to the base data have been done. Both transaction and query performances are good, but queries may return non-up-to-date results.
- Refreshing is done periodically in certain time intervals. This is also called the *snapshot maintenance*. Again, in spite of the good transaction and query time, queries may return non-up-to-date results.

In general, immediate maintenance does not scale with the number of materialized views, but deferred maintenance does. Therefore the decision of which views to maintain immediately has to be made very selectively. If real-time queries are asked on a view for which consistent results are crucial, then that view should be maintained immediately. Views which are queried relatively infrequently can be maintained in a deferred fashion. Usually decision support applications, where a stable copy of the derived data is more important than freshness, use periodical deferred policy. [CKL$^+$97] provides a decent study on consistency and performance issues in supporting multiple view maintenance policies. Materialized views that are related to each other may become inconsistent if they are maintained under different policies. Mutual consistency between views has to be settled.

The next question to ask is *how* view maintenance is applied. In the next subsections, we discuss how actually the maintenance should be performed.

6.3 Incremental View Maintenance

Incremental view maintenance algorithms have been investigated for a long time as an efficient alternative to re-materialization. Most of the work in this area consider the problem for centralized database systems where materialized views are used for purposes like speeding up queries on views or implementing rule checking efficiently. The problem has additional facets when considered in the scope of data integration systems. However, previous work still applies to some cases and form the basis of algorithms for data integration applications. We believe the following categorization of incremental view maintenance algorithms clarifies the link between the two cases:

- pre-update algorithms: maintenance is performed before the base relations have been actually updated, as in the case of immediate maintenance policy where maintenance is performed within the transaction that is updating the source.

- post-update algorithms: maintenance is performed after the transaction that updates the relevant base relations is over.

Previous methods that apply to centralized databases naturally involve pre-update algorithms because the base relations and the materialized views are parts of the same system. However, data integration systems have to use post-update algorithms since the underlying sources are autonomous and they are unaware of the maintenance procedures that are occurring in the integrated system. We can not force them to include maintenance procedures within their update transactions. As stated in [ZGMHW95], information sources are decoupled from the data warehouse. This brings additional problems about consistency.

In this section, our focus is on methods devised for incremental view maintenance in general. We present techniques specifically on data integration systems in the next subsection.

[GM95] provides a survey of incremental view maintenance algorithms classifying them according to view language and available information dimensions. Here we discuss some of them without giving an explicit classification. Our aim is to give a flavor of the important issues that are addressed in many of those algorithms.

[BLT86] handles Select (S), Project (P) and Join (J) views in isolation first and then considers them together as SPJ views. For each case, both insertions and deletions are considered. For S views, inserted tuples are simply unioned and deleted tuples are simply subtracted from the materialized view data set. Updating P views when deletion occurs in the base relation is more complicated. The problem stems from the fact that a tuple in the view that is projected on some particular attribute may be there due to multiple tuples in the base relations. If one of these base tuples is deleted, the derived tuple may not have to be deleted since there are other existing base tuples that it is derived from. This problem is solved by using *counter*s for view tuples. A view tuple would have to be deleted when the counter dropped to 0. Upon insertions of new tuples to one of the join relations, J views should only perform join between the newly added tuples and the other join relation rather than computing the join between two relations from scratch. Deletions are handled in a similar way by only joining the deleted tuples and then subtracting those from the original view. These methods are further combined together for SPJ views [BLT86].

[GMS93] presents two algorithms: *Counting* algorithm and *DRed* (Deletion and Re-derivation) algorithm. In both of these algorithms, the emphasis is on deletion since it is more problematic. Counting algorithm is proposed for non-recursive views with negation and aggregate functions. It is based on the *counter* method of [BLT86], but the view language is more general. For each tuple in the materialized view, number of alternative derivations is stored as the *count*. Relevant insertions increment the count and relevant deletions decrement the count by 1. When the count drops to 0, the tuple need not be stored in the materialized view any more. This algorithm also works with recursive views only if every tuple has finite number of derivations. DRed algorithm works for general recursive views with negation and aggregation. This algorithm involves three basic steps: (i) ignore the alternative derivations and put the view tuple into the delete set if it gets invalidated at least by one of its derivations, (ii) remove the tuples from the delete set if they have other derivations, (iii) compute the tuples to be inserted to the views due to insertions to base relations.

In addition to the algorithmic approaches as summarized above, there are algebraic approaches to incremental view maintenance. [GL95] presents an approach based on multi-set/bag semantics. All the arguments are based on the equivalence of bag-valued expressions. Bag algebra expressions are used to represent the materialized views. Given a transaction that changes the state of the database and a set of bag expressions, they try to derive delta expressions which represent how the bag algebra expressions need to be updated. The goal is to find a minimal set of such delta expressions. [GL95] also emphasizes that proper handling of duplicates is important for computing the aggregate functions (like averaging a list of values) correctly.

[RSS96] explores what additional views should be materialized for optimal incremental maintenance of a given materialized view. [MRRS00] generalizes this idea to how to maintain a set of views efficiently by using additional temporarily or persistently materialized views. Approach involves materializing common subexpressions between view maintenance expressions as in multi query optimization algorithms. We mentioned this approach before in this section as we presented the selection of views to materialize.

6.4 View Maintenance in Data Integration Systems

The main problem in data integration systems in terms of incremental view maintenance is that maintenance has to be done after the updates at the data sources have occurred. Later, when the maintenance has to take place, the integrated system may need to ask additional queries to the data sources if it does not have all the information needed to perform the maintenance. If the data sources continue to change in the time between the updates known by the integrated system and the maintenance time, then the additional queries will be answered according to the new state of the data sources which is different than the one at the time of initial update (i.e., the update which is trying to be fixed at the integrated system). This is called a *state bug* [CGL$^+$96] or *view maintenance anomaly* [ZGMHW95]. This problem stems from the fact that pre-update maintenance algorithms can not be used for data integration systems as they are. [CGL$^+$96] proposes two ways of avoiding the state bug:

- using the pre-update algorithms but restricting the updates and views so that correctness is guaranteed
- developing specific algorithms for the post-update case

[CGL$^+$96] proposes new algorithms for post-update case which are based on the usage of database invariants, i.e. conditions that are guaranteed to hold at every state of the database. These are used to maintain correctness. As [GL95], algebraic approach based on bag semantics is taken. [CGL$^+$96] also emphasizes the minimization of the view down-time. Usually views become inaccessible for queries during maintenance. [QW97] addresses this problem through a two-version no locking (2VNL) algorithm. Two concurrent versions of the materialized views provide continuous and consistent access to the warehouse during maintenance.

[ZGMHW95], on the other hand, is based on a pre-update view maintenance algorithm. The algorithm in [BLT86], which we briefly summarized in the preceding subsection, is used as basis. [ZGMHW95] proposes ECA (Eager Compensating Algorithm) in which extra compensating queries are used to eliminate anomalies. In fact anomalies would not occur if we re-computed the views or stored the copies of base relations referenced in the views, but both of these options are too costly and not good options compared to incremental view maintenance. In ECA, the basic idea is to send compensating queries to the data sources to avoid the potential anomalies that may occur according to query answers coming from the data sources. In other words, the warehouse eagerly forces data sources to send correct information. This is done by anticipating what kind of anomalies can occur beforehand and preparing view maintenance queries which contain compensating expressions in addition to the view maintenance expressions that would avoid the anomalies.

Next subsections discuss how the incremental maintenance process could be made more efficient.

6.5 Update Filtering

Not all the updates at the data sources cause updates at the materialized views. We can speed up the maintenance process if we can detect which base data updates have no effect on the views, and hence need not be maintained. Such updates are called *irrelevant updates* and the procedure of pruning irrelevant updates from the maintenance plan is called *update filtering*. [BCL89] calls queries/views that are not affected from the updates *queries independent of updates*.

Most of the work in this area aims at theoretically defining necessary and sufficient conditions for the detection of irrelevant updates for the cases of insertions, deletions and modifications [BLT86, BCL89, LS93]. [BCL89] defines irrelevant updates as update operations applied to a base relation has no effect on the state of a derived relation independently of the database state. [LS93] reduces the update independence problem to equivalence problem for Datalog programs and provides decidability results for different cases. [BLT86] presents more practical algorithms for detecting irrelevant updates. The views considered are in the form of PSJ queries. Selection condition is the primary determinant for deciding relevance. For insertions at the base relations, we substitute the values of the inserted tuple in the selection condition of the view. If the selection condition becomes unsatisfiable, then the insertion is irrelevant to the view, i.e., no tuple needs to be inserted to the view. Else, the insertion *may* be relevant to the view. Similarly, for deletions, we substitute the values of the deleted tuple in the selection condition of the view. If the selection condition becomes unsatisfiable, then the deletion is irrelevant to the view, i.e., no tuple from the view needs to be deleted. In general, satisfiability of boolean expressions is NP-complete. [BLT86] assumes boolean expressions that are conjunctions of inequalities. Then the problem can be solved in polynomial time. It can also be generalized to disjunctions of conjunctions, which adds a linear factor to the complexity.

In conventional database systems, update filtering can be implemented using integrity constraints or triggers. The base relations are not decoupled from the derived relations. View definitions are known to the whole system. However, in data integration systems, the filtering has to be done at the integration system level. Data sources can not perform filtering since they are not aware of the view definitions.

6.6 View Self-Maintenance

Another way to speed up maintenance is to minimize external data source access. As we mentioned earlier, to maintain a view, we may need to ask queries to the data sources in addition to the update information itself. This requires to communicate with the sources. We should try to exploit the information available at the integrated system (data warehouse) as much as we can to avoid this communication.

In general, *self-maintenance* refers to views being maintained without using all the base data. There exists different notions of its exact meaning depending on how much information is available. The ideal case is that the view update is performed locally at the integrated system by only knowing the particular base data update that has occurred, the view definitions and the materialized data. Whenever this is not possible, we need additional techniques to minimize base data access.

The first thing to do is to decide whether a given view is self-maintainable or not. If it is, then we need to know how to achieve self-maintenance. Otherwise, techniques may be developed to make it self-maintainable. Self-maintainability can be both investigated on a single view or on multiple views. Initially, we can consider each view in isolation. It should also be noted that self-maintainability is an issue specific to data integration systems. In traditional (centralized) databases, since all information is known to the system, there is no context for self-maintainability.

[GJM96] aims at defining self-maintenance rules for SPJ views. Self-maintainability algorithms are highly dependent on the view definition language. Three issues are investigated: (i) which relation is modified, (ii) what type of modification, and (iii) if key information can be exploited. The results they have come up with are as follows:

- For insertions, SP views are self-maintainable. SPJ views are self-maintainable only if join is a self-join (i.e. relation R is joined with itself) and join attribute is the key of R. Other SPJ views are not self-maintainable.

- For deletions, SPJ views are self-maintainable.

- For updates, if modeled as deletion followed by insertion, the rules for insertions and deletions hold. Otherwise, SPJ views are self-maintainable if updates are on non-exposed (i.e. not involved in any predicate in the view definition) attributes.

[BCL89] explores similar conditions for a more general view definition language.

[Huy97] investigates the meaning of self-maintainability at different contexts. They show that self-maintainability can be reduced to the problem of deciding query containment.

There are several techniques to make views locally maintainable [Huy97]:

- Multiple-View Self-Maintenance
  Views that are not self-maintainable when considered in isolation may become collectively maintainable at the integrated system when they are considered together [Huy97]. In other words, the information available to each view is extended to all the materialized views at the warehouse in addition to its own definition and materialization.

- Batch Updates
  Rather than maintaining each update operation separately, if we save the updates and maintain them all together, then the amount of work may be reduced. For example, if an update operation deletes a tuple and a following update inserts the same tuple back, then these two updates have no effect on the state of the materialized views when considered as a whole.

- Auxiliary Materialized Views
  By materializing additional views, other views may become self-maintainable. The basic idea here is to increase the amount of information available at the integrated system level.

Lastly, one important point to note is that self-maintenance also removes the situations where anomalies can occur when the maintenance is totally performed locally at the integrated system. The reason for this is that anomalies are caused by additional queries asked to the data sources some time after the related update has occurred. If a view is self-maintainable, then no additional querying is necessary.

6.7 Dynamic View Management

As stated earlier, materialized view management has two important components: view selection and view maintenance. Until now we assumed that views to materialize are selected once at the beginning according to some statistics on frequently asked queries and base data update frequencies and then the selection is over. From that point on, the system concentrates on the maintenance of those selected views. This kind of view management is called *static view management.* The major problem with this approach is that if the query workload or base data update patterns change, then the decisions about view selection become invalid.

The solution proposed in [KR99] is *dynamic view management* in which view selection and view maintenance stages are unified. The query workload is continuously monitored by the system and view selection decisions are updated dynamically. The constraints to be considered in addition to the changing workload patterns include disk space and maintenance window. Maintenance window has more importance than space because usually the system is unavailable for queries while the maintenance is being carried out. This time window has to be kept as short as possible. The more number of views materialized, the longer the maintenance window is. However, more materialization speeds up the query processing. Therefore, a compromise has to be made.

7 Concluding Remarks

Data integration services are important ingredients of network data services. They present an opportunity for transparent access to many different data sources residing on the network, as if these data sources were in fact a single data source. In other words, they hide from the user the fact that there is a diversity of data sources.

In this chapter, we have presented major problems in building and operating data integration systems together with a survey of proposed solutions. Most of the current data integration methods are still under research and most of the systems mentioned throughout this chapter are research prototypes. There is a need to apply the proposed techniques to real-life systems to improve their performances.

One important point we have not mentioned in the chapter is the standardization issues that are under development to represent and communicate data on the web in an easier way. The most well-known of these standards is the eXtensible Markup Language (XML) [XML00]. As XML or similar standardization efforts became widely used, the data integration services could also benefit from this. The heterogeneity in data representations could decrease and more effective solutions for data integration would be possible.

The amount and availability of data that exist on the network are growing in an increasing speed. The need to bring these data together to infer useful information is also rising. As long as there will be data, there will also be need for data integration services.

Bibliography

[AHK96] Y. Arens, C. Hsu, and C. A. Knoblock. Query Processing in the SIMS Information Mediator. In A. Tate, editor, *Advanced Planning Technology*, pages 61–69. AAAI Press, Menlo Park, CA, 1996.

[AK97] N. Ashish and C. Knoblock. Semi-automatic Wrapper Generation for Internet Information Sources. In *Second IFCIS International Conference on Cooperative Information Systems (CoopIS)*, Charleston, SC, 1997.

[AKS96] Y. Arens, C. A. Knoblock, and W. Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems (JIIS) - Special Issue on Intelligent Information Integration*, 6(2/3):99–130, 1996.

[ALU01] F. N. Afrati, C. Li, and J. D. Ullman. Generating Efficient Plans for Queries Using Views. In *ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, May 2001.

[Ash00] N. Ashish. *Optimizing Information Mediators By Selectively Materializing Data*. PhD thesis, USC, March 2000.

[BCL89] J. A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *Transactions on Database Systems (TODS)*, 14(3):369–400, September 1989.

[BF94] M. Bonjour and G. Falquet. Concept Bases: A Support to Information Systems Integration. *Proceedings of CAiSE94 Conference , Utrecht, 1994*, 1994.

[BKLW99] S. Busse, R. Kutsche, U. Leser, and H. Weber. Federated Information Systems: Concepts, Terminology and Architectures. Technical Report 99-9, Berlin Technical University, 1999.

[BLT86] J. A. Blakeley, P. Larson, and F. Wm. Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD International Con-*

ference on Management of Data, pages 61–71, Washington, DC, May 1986.

[Bor95] A. Borgida. Description Logics in Data Management. *IEEE Transactions on Knowledge and Data Management*, 7(5):671–682, October 1995.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CGL+96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *ACM SIGMOD International Conference on Management of Data*, pages 469–480, Montreal, Canada, June 1996.

[CGL+98] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Description Logic Framework for Information Integration. In *Principles of Knowledge Representation and Reasoning*, pages 2–13, 1998.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *10th Meeting of the Information Processing Society of Japan (IPSJ)*, pages 7–18, Tokyo, Japan, October 1994.

[CKL+97] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting Multiple View Maintenance Policies. In *ACM SIGMOD International Conference on Management of Data*, pages 405–416, Tucson, AZ, June 1997.

[CLN99] D. Calvanese, M. Lenzerini, and D. Nardi. Unifying Class-Based Representation Formalisms. *Journal of Artificial Intelligence Research*, 11:199–240, 1999.

[Eik99] Line Eikvil. Information Extraction from World Wide Web. A Survey, July 1999.

[EJ95] L. Ekenberg and P. Johannesson. Conflictfreeness as a Basis for Schema Integration. In *Conference on Information Systems and Management of Data*, pages 1–13, 1995.

[FKL97] D. Florescu, D. Koller, and A. Levy. Using Probabilistic Information in Data Integration. In *International Conference on Very Large Data Bases (VLDB)*, pages 216–225, Athens, Greece, August 1997.

[Fri99] M. T. Friedman. *Representation and Optimization for Data Integration.* PhD thesis, University of Washington, 1999.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data Integration using Self-Maintainable Views. In *International Conference on Extending Database Technology (EDBT)*, pages 140–144, Avignon, France, March 1996.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *ACM SIGMOD International Conference on Management of Data*, pages 328–339, San Jose, CA, June 1995.

[GM95] A. Gupta and I. S. Mumick. Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GM99] A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, DC, May 1993.

[GMUW00] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*, chapter 11: Information Integration. Prentice Hall, 2000.

[Gup97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *International Conference on Database Theory (ICDT)*, pages 98–112, Delphi, Greece, January 1997.

[Hal95] G. Hall. Negotiation in Database Schema Integration. In *The Inaugural AIS Americas Conference on Information Systems*, Pittsburgh, PA, August 1995.

[Has00] W. Hasselbring. Information System Integration. *Communications of the ACM*, 43(6):33–38, 2000.

[HG92] R. Herzig and M. Gogolla. Transforming Conceptual Data Models into an Object Model. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 280–298, 1992.

[HGMN$^+$97] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breunig, and V. Vassalos. Template-based wrappers in the TSIMMIS system. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[HGMW$^+$95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):41–48, 1995.

[HM85] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, 1985.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD International Conference on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[HSC$^+$97] M. N. Huhns, M. P. Singh, P. E. Cannata, N. Jacobs, T. Ksiezyk, K. Ong, A. P. Sheth, C. Tomlinson, and D. Woelk. The Carnot Heterogeneous Database Project: Implemented Applications. *Distributed and Parallel Databases Journal*, 5(2):207–225, 1997.

[Huy97] N. Huyn. Multiple-View Self-Maintenance in Data Warehousing Environments. In *International Conference on Very Large Data Bases (VLDB)*, pages 26–35, Athens, Greece, August 1997.

[HZ96] R. Hull and G. Zhou. A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In *ACM SIGMOD International Conference on Management of Data*, pages 481–492, 1996.

[IFF$^+$99] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *ACM SIGMOD International Conference on Management of Data*, pages 299–310, Philadelphia, PA, June 1999.

[JLYV00] M. Jarke, M. Lenzerini, and P. Vassiliadis Y. Vassiliou. *Fundamentals of Data Warehouses*. Springer Verlag, 2000.

[JPSL$^+$88] G. Jacobsen, G. Piatetsky-Shapiro, C. Lafond, M. Rajinikanth, and J. Hernandez. CALIDA: A Knowledge–Based System for Integrating Multiple Heterogeneous Databases. In *Third International Conference on Data and Knowledge Bases*, pages 3–18, Jerusalem, Israel, 1988.

[KLSS95] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Symposium on Information Gathering in Distributed Heterogeneous Environments*, 1995.

[KR99] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *ACM SIGMOD International Conference on Management of Data*, pages 371–382, Philadelphia, PA, June 1999.

[KWD97] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint conference on Aritificial Intelligence (IJCAI)*, pages 729–737, 1997.

[Lev99a] A. Y. Levy. Combining Artificial Intelligence and Databases for Data Integration. In *Special issue of LNAI: Artificial Intelligence Today; Recent Trends and Developments*. Springer Verlag, 1999.

[Lev99b] A. Y. Levy. Logic-Based Techniques in Data Integration. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence*, Washington, DC, June 1999.

[Lev00] A. Y. Levy. Answering Queries Using Views: A Survey. Submitted for publication, 2000.

[Lit85] W. Litwin. An Overview of the Multidatabase System MRSDM. In *ACM National Conference*, pages 495–504, October 1985.

[LRO96] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *International Conference on Very Large Data Bases (VLDB)*, pages 251–262, Bombay, India, September 1996.

[LS93] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. In *International Conference on Very Large Data Bases (VLDB)*, pages 171–181, Dublin, Ireland, August 1993.

[LSS93] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the Logical Foundation of Schema Integration and Evolution in Heterogeneous Database Systems. In *2nd International Conference on Deductive and Object-Oriented Databases*, pages 81–100, Phoenix, AZ, 1993.

[Mit99] P. Mitra. Algorithms for Answering Queries Efficiently Using Views. Technical report, Infolab, Stanford University, September 1999.

[MMK98] I. Muslea, S. Minton, and C. Knoblock. Wrapper induction for semistructured web-based information sources. In *Conference on Automatic Learning and Discovery CONALD-98*, 1998.

[MRRS00] H. Mistry, P. Roy, K. Ramamritham, and S. Sudarshan. Materialized View Selection and Maintenance using Multi-Query Optimization. Submitted for publication, March 2000.

[MW88] N. E. Malagardis and T. J. Williams, editors. *Standards in Information Technology and Industrial Control*, chapter Multidatabase Systems in ISO/OSI Environment, pages 83–97. North-Holland, Netherlands, 1988.

[ND95] S. Navathe and M. Donahoo. Towards Intelligent Integration of Heterogeneous Information Sources. In *Proceedings of the 6th International Workshop on Database Re-engineering and Interoperability*, 1995.

[OV99] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, chapter 4: Distributed DBMS Architecture. Prentice Hall, 1999.

[PGMA96] Y. Papakonstantinou, H. Garcia-Molina, and S. Abiteboul. Object fusion in mediator systems. In *International Conference on Very Large Databases*, Bombay, India, September 1996.

[PL00] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. In *International Conference on Very Large Data Bases (VLDB)*, pages 484–495, Cairo, Egypt, September 2000.

[QW97] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *ACM SIGMOD International Conference on Management of Data*, pages 393–404, Tucson, AZ, June 1997.

[Rea89] M. Rusinkiewicz and et. al. OMNIBASE: Design and Implementation of a Multidatabase System. In *1st Annual Symposium in Parallel and Distributed Processing*, Dallas, Texas, May 1989.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *ACM SIGMOD International Conference on Management of Data*, pages 447–458, Montreal, Canada, June 1996.

[SDN98] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized View Selection for Multidimensional Datasets. In *International Conference on Very Large Data Bases (VLDB)*, pages 488–499, New York City, NY, August 1998.

[SG90] T. K. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(2):262–266, June 1990.

[SL90] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[TBC$^+$87] T. Templeton, D. Brill, A. Chen, S. Dao, E. Lund, R. Macgregor, and P. Ward. Mermaid: A Front-End to Distributed Heterogeneous Databases. In *International Conference on Data Engineering*, pages 695–708, 1987.

[Ull97] J. D. Ullman. Information Integration using Logical Views. In *International Conference on Database Theory (ICDT)*, pages 19–40, Delphi, Greece, January 1997.

[Var99] A. Vargun. Semantic Aspects of Heterogeneous Databases, 1999.

[VP98] V. Vassalos and Y. Papakonstantinou. Using Knowledge of Redundancy for Query Optimization in Mediators. In *Workshop on AI and Information Integration (in conjunction with AAAI'98)*, Madison, WI, July 1998.

[XML00] Extensible Markup Language (XML) 1.0. W3C Recommendation, October 2000. http://www.w3.org/TR/REC-xml.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *International Conference on Very Large Data Bases (VLDB)*, pages 136–145, Athens, Greece, August 1997.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, CA, June 1995.