Chapter 1

Data Integration Services

1 Introduction

... will be written later ...

1.1 Definition of Data Integration

Data integration systems harmonize data from multiple sources into a single coherent representation. The goal is to provide an integrated view over all the data sources of interest and to provide a uniform interface to access all of these data. The access to the integrated data is usually in the form of querying rather than updating the data.

The data sources to be integrated may belong to the same enterprise or may be arbitrary sources on the web. 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 (old and obsolescent systems that are difficult to migrate to a modern technology) or structured/unstructured files with different interfaces. Data integration requires that the differences in modeling, semantics and capabilities of the sources together with the possible inconsistencies be resolved.

1.2 Motivation for Data Integration Systems

- Historical view: integration-by-hand
- Users can focus on specifying *what* they want, not on *how* to obtain what they want. Instead of finding relevant sources, interacting with every source and combining data from different sources, a user can ask queries in a unified way.
  Particular examples:
  - Desire for reports that describe all parts of a merged organization (bank mergers, car dealerships, etc.).

- Facilitates decision support applications (OLAP, Data mining)

  OLAP (On-Line Analytical Processing) is making financial, marketing or business analysis to be able to make business decisions on a collection of detailed data from one or more data sources. The analysis is done through asking large number of aggregate queries on the detailed data.

  Data Mining is discovering knowledge from a large volume of data. Statistical rules or patterns are automatically found from the raw collection of data.

1.3 Major Issues

- Heterogeneity of data sources
- Availability of data sources
- Dynamicity of individual data sources
- Autonomy of data sources
- Correctness of the integrated view of the data
- Query performance

1.4 Summary of State of the Art

... will be written later ...

2 Data Integration Architectures

2.1 Dimensions to categorize architectural models for integrating data sources

There are three orthogonal dimensions which are traditionally used in literature to describe distributed information systems: autonomy, heterogeneity and distribution. Sometimes transparency is considered as the forth parameter. Below we are discussing each of these dimensions.

- Autonomy
  Autonomy refers to the degree to which individual data sources can operate independently. According to Verjalainen and Popescu-Zeletin's classification, 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 part 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 technical level, heterogeneity comes from different hardware platforms, operating systems, networking protocols or similar lower-level concepts. On a conceptual level, heterogeneity comes from different programming and data models as well as different understanding and modeling of the same real-world concepts (ex: naming).
- Distribution
  Distribution refers to the physical distribution of data over multiple sites.
  - Client/Server
    Server does data management, client provides user interface.
  - Peer-to-Peer (fully distributed)
    Each machine has full functionality of data management.
- Distribution transparency
  Transparency refers to the separation of higher-level semantics of a system from lower-level implementation issues. A transparent system hides the implementation details from users.

2.2 Major Approaches to Data Integration

Three common approaches to integrate data sources are the following:

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

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

- Hybrid approach
  Data is selectively materialized, that is, the system is essentially mediator-based where data is extracted from sources on-demand, but the results of the most popular queries are precomputed and stored.

When a number of data sources is very large, and/or the sources are prone to change often (like in the case of web sources), and/or there is no way to predict what kind of queries users will ask, virtual approach is preferable over data warehousing approach. If, however, sources are fixed, don't get upgraded too often and we know what kind of queries are most popular, we can materialize some of them.

2.3 Virtual View Approach

2.3.1 Federated Database Systems

Federated Database System (FDBS) consists of semi-autonomous components (database systems) that operate independently but participate in a federation to partially share data with each other.

This sharing is controlled by each component and not centralized. The components can not be called "fully-autonomous" because each of them is modified by adding interface to communicate with all other components.

Each of the component database systems can be either centralized DBMS, or 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.

FDBS supports local and global operations and treats them differently. Local operations involve only local data access and correspond to the queries submitted directly to this source. Global operations use *FDMS (federated database management system)* to access data from other components. In case of a global operation, each data source whose data is required must allow access to it.

Depending on how the component database systems are integrated, there can be *loosely coupled* FDBSs or *tightly coupled* FDBSs. Tightly coupled FDBS has a unified schema (or several unified schemas) which can be either semi-automatically built by schema integration techniques (see section 3 for details) or created manually by user. To solve logical heterogeneity, a domain expert needs to determine correspondences between schemas of the sources. Tightly coupled FDBS is usually static and difficult to evolve, because schema integration techniques don't allow to add or remove components easily. Examples of this kind of FDBSs are ...

Loosely coupled FDBS does not have a unified schema, but it provides some unified language for querying sources. In this case, component database systems have more autonomy, but user has to resolve all semantic heterogeneities himself. Only technical metadata is needed by loosely coupled FDBS as opposed to tightly coupled one, which requires semantic metadata in addition. 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, Omnibase and Calida.

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

- Data exchange
  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 on its data. Then other components should have 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 simplest way to achieve interoperability 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 1.1. Unfortunately, it requires n*(n-1) schema translations and becomes too tedious with the growth of a number of components in federation. Research is 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 literature: some people call only tightly coupled systems FDBSs, some call only loosely FDBSs, and some take the same approach we did by considering tight and loose architectures be two kinds of federated database system architecture.

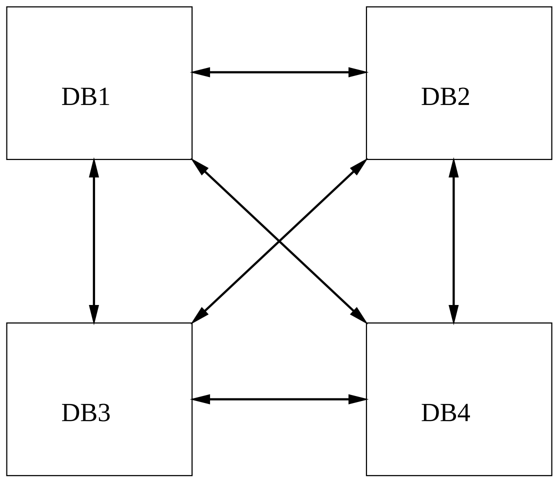

Figure 1.1: Example of federated database architecture (taken from tan book, refer [])

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 to answer the query. It is a good compromise between full integration and no integration.

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

A natural question that arises is how mediation architecture is different from tightly coupled FDBS? Here are the differences between them:

- Mediated architecture may have non-database components
- Query capabilities of sources in mediator-based system can be restricted and the sources don't have to support SQL-querying at all (semistructured data)
- Access to the sources in a mediator-based system is read-only as opposed to read-write access to FDBS
- Development of mediated systems is usually done in top-down way as opposed to bottom-up approach for tightly coupled FDBS

- Sources in mediator-based approach have complete autonomy which means it is easy to add or remove new data sources

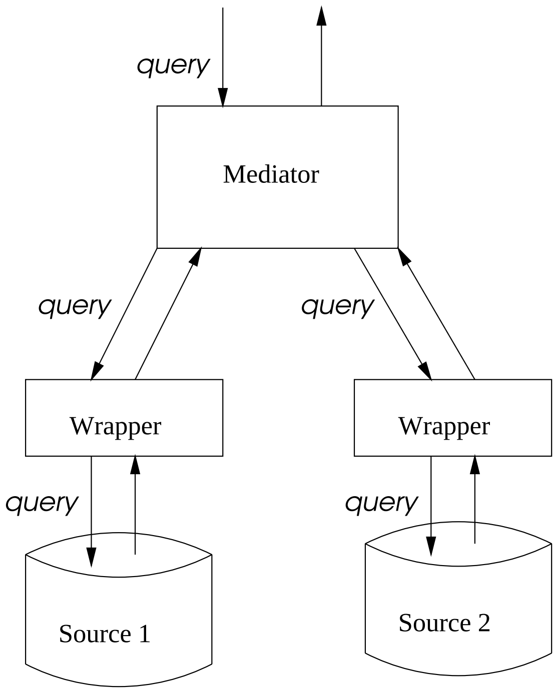

Figure 1.2: Mediated architecture (borrowed with some minor changes from tan book, refer [])

A typical architecture for a mediated system (with two sources) is given in Figure 1.2. Two main components of a mediated system are *mediator* and *wrapper*. Mediator (can be also called *integrator*) performs the following actions in the system:

- Receives a query formulated on the unified (mediated) schema from a user.
- Decomposes this query into sub-queries to individual sources based on *source descriptions* which it has.
- Optimizes execution plan based on source descriptions again.
- Sends sub-queries to wrappers of individual sources, which will transform these sub-queries into queries over sources' local schemas. Then it receives answers to these sub-queries from wrappers, combines them into one answer and sends it to the user.

These steps are described in details in the section on query processing.

Wrapper hides technical and data model heterogeneity from the integrator. It is an important component of both mediator-based architecture and data warehouse, but wrappers for mediated systems are usually more complicated. Please refer to section 5.1 for more information about wrappers.

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

Cars(vin, make, model, color, price, year, mileage)

Dealer two also rents some of his cars, so he has separate relations for cars for sale and for rent. He stores information about cars for sale in two relations:

CarsForSale(vechicalID, carMake, carModel, carColor, carPrice, carYear), CarsSaleMileage(vechicalID, mileage).

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

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

Now if a client of Acme Cars asks a SQL-query:

SELECT vin, autoModel, autoColor, autoYear
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, color to autoColor and year to autoYear.

The wrapper for the second dealer will translate this query to:

SELECT vechicalID, carModel, carColor, carYear
FROM CarsForSale
WHERE carMake = "Honda" **AND** carPrice < 14,000

It also renames vechicalID to vin, carModel to autoModel, carColor to autoColor etc.

Known implementations of mediator-based architecture are: TSIMMIS, Information Manifold, SIMS, Carnot ... Some of them are covered in more details in Systems section.

2.4 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 approach described above, but here this filtered data is actually stored in a single repository (called "data warehouse").
How is a data warehouse different from a traditional databases with OLTP (On-Line Transaction Processing)?

- Warehouse will usually contains terabytes of data and may combine data from many databases, semi-structured and other sources
- Workloads are query intensive; queries are complex and query throughput is more important than transaction throughput
- A data warehouse often contains historical and summarized data which is used for decision support. That also infers that users of a data warehouse are different than users of a traditional DBS: they will be analysts, knowledge workers, executives
- Information is usually read-only as opposed to read/write operations in OLTP.

What is involved in building a data warehouse?

- Modeling and design
  In the stage of designing a warehouse, we need to decide information from which sources we are going to use there, what views (queries) over these sources we want to materialize, and what the global unified schema of the warehouse will be.
- Maintenance (refreshing)
  Maintenance deals with how we create our warehouse from source data and how we refresh it when data in the sources is updated. There are three ways to create a warehouse:
  -Do it periodically when no queries to the system are sent (night time for instance) re-creating a warehouse from scratch from data sources
  -Periodically *incrementally update* it, that is, incorporate changes made to the sources since last update. In this case, only very small amount of data will be touched, so it is more efficient, but it is also more complicated, has a number of issues and is an area of active research.
  -Update it after every change made to any of the sources. This approach does not seem to be too practical though, except for small warehouses with rarely changing data sources [refer to tan book].

View maintenance is the key research topic specific to data warehousing and we discuss it in details in Section 6.

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

Example of a two-source data warehouse is given in Figure 1.3.

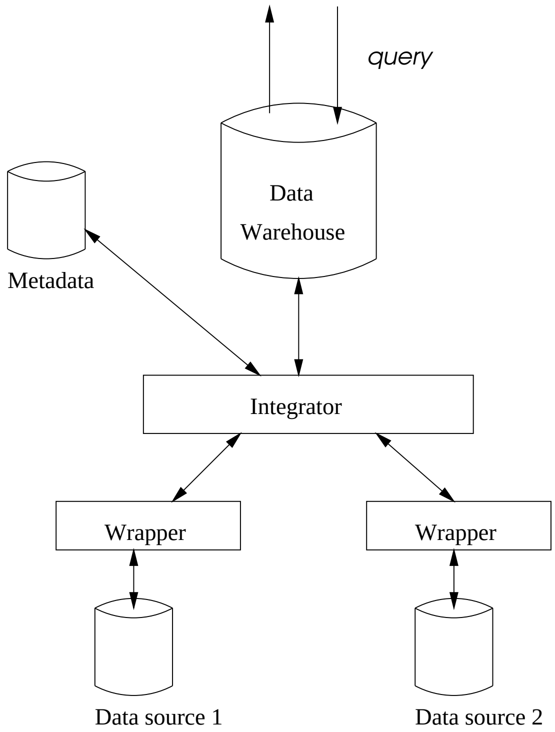

Figure 1.3: A data warehouse

Example: Let's suppose there is a company X that owes two toy stores. Toys there are identical and each store has a database where for each date there us a number of each type of toy (teddy-bear or dog) sold in this store. So store 1 stores relation: **Sales(date, typeToy, numberSold)** and store 2 has two relations: **TeddyBears(date, numberSold)** and **DogsToys(date, number-Sold)**.
Now assume, that company X would like to have the following relation in the data warehouse for decision making purposes (future marketing):
ToySales(date, typeToy, numberSold)
In this case, we need 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. SQL query to the

first source is straightforward, as the relation is exactly the same apart from the name it has. It will look like this:

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

For the second source, we 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

Known implementations of data warehousing approach are Squirrel and WHIPS systems. Overview of WHIPS is given in Systems section of this chapter.

2.5 Hybrid approach

This approach is usually discussed as a way to improve performance of some mediator-based systems. When approach to data integration is virtual, but the queries asked most often are determined and they are materialized in some repository. This repository then can serve as a new source for this mediated system. Issues which arise in this case are some of issues for data warehousing approach:

- What data to materialize?
- How this materialized data is maintained

Such approach discussed in [K.], but otherwise is not too common compared to data warehousing and mediation.

3 Semantic problems in data integration

Different information systems can use different ways of presenting their information to their users. Those differences can make it very difficult for developers to integrate data from the two systems.

This section explores the nature of why this integration can be so difficult, and presents some (partial) solutions the problem.

3.1 The goal of an information system

3.1.1 Information systems as assertions about our world

An information system can be thought of as a record of some facts about the world. (Other functionality might also exist in an information system, but that isn't relevant to this present discussion.)

If an information system is accurate in what it claims about the world, then it's useful. If the information system is inaccurate, the it's much less useful.

3.1.2 System interfaces

An information system makes its information available to users via its *system interfaces*. In modern information systems, these might include any of the following:

- SQL access and the documentation of the database's schema
- A CORBA system with corresponding IDL files
- A C function library
- A set of URLs and associated query parameters
- A set of XML DTDs and the path of the filesystem directory that will contain corresponding XML files

3.1.3 Semantics

The *semantics* of an interface is the specification of how the entities in the interface are supposed to correlate to:

- entities in some other system, or
- entities in the world we live in.

Example

Consider a table in our car dealership's customer database. The table is named *tblCustomers*, and has the following columns:

- cust_num: integer
- street: string(255)

- city: string(60)
- state: string(2)
- car_pref: integer FK(tblCarTypes)

A statement of the table's semantics may look like the following:

"*tblCustomers* contains one record for each customer that our dealership has ever sold a car to. A record is only removed from this table if it is discovered to refer to the same person that another record in this table refers to."

"*cust_num* is a unique identifier for each customer record. No two customers share the same customer number. A customer number is not intended to correspond to any value outside of this database. For a given customer record, this value will never change."

"*street*, *city*, and *state* are the mailing address that the customer has most recently been known to live it. The *street* field includes an apartment number specification if needed."

"*car_pref* is a foreign key into the *tblCarTypes* table. This field shows what type of car the customer has most recently expressed preference for."

3.2 Problems

3.2.1 Simple type differences

System interfaces are usually composed of common-place elements, such as C it unsigned ints, SQL *date*s, and Java Strings.

Sometimes there can be a very simple correspondence between a data source's exposed interface and the integrated system's exposed interface.

For example, both systems might provide a mechanism that when given a car's VIN (Vehicle Identification Number), yields the date that the car was built.

Suppose that the data source's interface is a SQL database with a table that maps cars' VIN to the car's manufacture date (presented as a SQL *datetime*). **TODO: Make sure I've got the right name for the SQL data type**

If the integrated system's exposed interface is written in Java, then it would be desirable for the integrated system to present a car's construction date as a *java.util.Date* object.

Solution

These are perhaps the most benign problems to deal with during integration, because:

- the software needed to implement the conversion can probably appear in a very localized part of the integrated system's source code, and
- this is a kind of conversion that many other software developers are also likely to need to do. This implies that it's quite likely that conversion libraries will be available to the developers of the integrated system.

3.2.2 Unexpressed or under-expressed semantics of data source interfaces

Properly understanding the semantics of the data sources interfaces is vital to integrating the data sources.

To illustrate this, consider what would happen if the developer who were integrating the data sources did *not* understand the semantics of the data sources' interfaces:

The developers, trying to make their integrated information system useful, would try to tell the users of the system how the data coming out of the integrated information system was supposed to correlate to the user's lives.

For example, "The field labelled 'Number sold: ' is the number of cars sold on the date that appears in the field labelled, 'Date', be all of our dealerships combined. [**TODO: Replace with our running example**]

However, in order to be able to make such semantic claims about the outputs of the integrated system, the developers would need to know the semantics of the interfaces that the integrated system got that data from.

If the developers who are writing the software to integrate the data sources aren't able to understand the semantics of the data sources' interfaces, they can't justify any semantic claims about the integrated information system's interface semantics.

Solution

This is a problem that presently requires human involvement to sort out. Confirming the semantics of data sources may involve talking with developers who previously worked with the data sources, talking with users, and some guesswork.

3.2.3 Onto mappings

Sometimes information systems can use different levels of precision to describe the same entities. This causes problems when constructing an integrated system.

Example

Consider the customer databases of two car dealerships. In both databases is a record of the type of car that each customer prefers to drive. This is collected to help the dealership know how to advertise best to each customer.

At one dealership, the customer preference information is very detailed: A customer's preference is expressed in terms of the manufacturer and model line of the car the user likes best. For example, customer 'Charlie Brown' prefers 'Ford F150 pickup truck'.

At the other dealership, the customer preference information is less specific: All that can be specified is the general class of vehicle. For example, 'Lucy Brown' prefers 'pickup truck'.

It happens to be the case that every kind of vehicle described in the first dealer's customer preferences database can be cleanly mapped into a class of vehicle in the second dealer's database. For example, a 'Ford F150 pickup truck' is a 'pickup truck'.

However, the opposite is not true: A vehicle preference from the second dealership's database does not map cleanly into a vehicle preference from the first dealership's database. Therefore, the two databases' vehicle preferences have an *onto mapping*.

Now suppose that an organization tries to create in integrated system that draws customers' vehicle preference information from the two dealerships' customer databases. The developers of the integrated system are confronted with the *onto* relationship described above.

Solution

If uniformity of detail in the integrated system has a high priority, then probably the most reasonable solution is to have the integrated system provide only the subset of information about an entity that is available from all relevant data sources.

A more sophisticated integrated system might allow its interface to provide additional information about an entity in those special cases where the particular data source involved has more information than the common subset. [**TODO: Needs better wording.**]

3.2.4 Different categorizations

Different information systems can record similar information in ways that are so different from each other that integrating the information can be very awkward.

Example

Suppose two car dealerships track the amount of gasoline used at the dealership each month.

One dealership records the monthly use in terms of volume (i.e., gallons) purchased per month.

The other dealership records the monthly use in terms of money spent on gasoline per month.

Now suppose that an integrated system is being developed to show the gasoline use from all car dealers in the larger organization. The developers of the integrated system must wrestle with the difference in measurements.

Note that it could be argued that these two values aren't legitimate candidates for integration, because they actually represent two different details about the dealerships. However, the reality is that the concepts are so similar that a developer might genuinely be asked to provide a (numerically) approximate integration of the values.

Solution

This is a very messy problem. Acceptable solutions are likely to be very application-dependent.

3.2.5 Recognizing object identity

Data sources can have an unstated assumption that there's a one-to-one correspondence between entities in the data source and entities in the outside world.

For example, a car dealership would ideally have only one "customer" record per actual human customer. This is an important quality of the system, because it allows users of the system to perform certain reasoning that otherwise wouldn't be sound.

However, what happens if two data sources being integrated might both have a record for the same customer?

If an integrated system makes a false assumption that the data sources have disjunct sets of customer records, then the integrated system now has duplicate customer records. Reasoning that assumes non-duplicate customer records would be impossible with the integrated system although it would be possible with any of the individual data sources.

Solution

Some data sources might provide enough information to allow the integration software to unambiguously detect matches between entities.

For example, two data sources might both use a customer's Social Security Number as a customer key. This makes duplicate detection trivial.

Often duplication detection involves guesswork. Software systems are available that try to make good guesses about duplicate records based on the information available. A common application of this is removing duplicate entries when large mailing lists are merged.

3.2.6 Conflicting data

Interface semantics often make an implicit claim that data using the interface is absolutely correct. People usually know to take that claim with a grain of salt. When unifying data from two or more data sources, contradictions can be reached, leading to an internally inconsistent view of data.

Example

Suppose that two car dealership have, over time, both sold the same car to a customer.

Each dealership maintains an inventory database that records for each car ever held by the dealer, the following pair of values: Vehicle Identification Number, Date-of-manufacture.

At one dealership, the value was correctly entered:

"123456842", "February 14, 2000"

At the other dealership, the value was incorrectly entered:

"123456842", "April 1, 2000"

When these data are integrated into a single information system, the conflicting values are detected.

Solution

Various approaches might be reasonable depending on the situation:

- When a conflict is detected, bring it to the attention of a human. The human can look for problems such as data entry errors and make a judgement. This audit might also lead to a correction of the original data in one of the data sources.

- If one system is considered more trustworthy than the other, use the answer provided by the more trusted system.
- If more than two systems provided conflicting answers, treat each data source's answer as a vote.
- If the answer is a real number, then allow mathematical interpolation (perhaps the arithmetic mean) to be the final answer presented by the integrated system.

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:

- getting a declarative query from the user and parsing it
- 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
- 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 are 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 **??**, 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 the mediated schema at the integration system (materialized view approach). In both of the approaches, the user query is formulated in terms of the mediated schema. However, in the latter approach, since the data is stored at the integration system according to the mediated schema, query evaluation is no more difficult than traditional way of query processing. The major issue there, is the synchronization of 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 **??**. During maintenance, views defined on the data sources have to be processed on the data sources to rematerialize the updated data. In other words, query processing on the original data sources is realized usually "off-line". [1] On the other hand, in the

[1] For immediate view maintenance policy, it is actually "on-line".

virtual view approach, every time a user asks a query, 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 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

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.

The below figure 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 **??**.

As Figure 1.4 indicates, in addition to modeling the mediated schema, we need to model the sources so that we can establish an association between the relations in the mediated schema and the relations in the source schemas. This

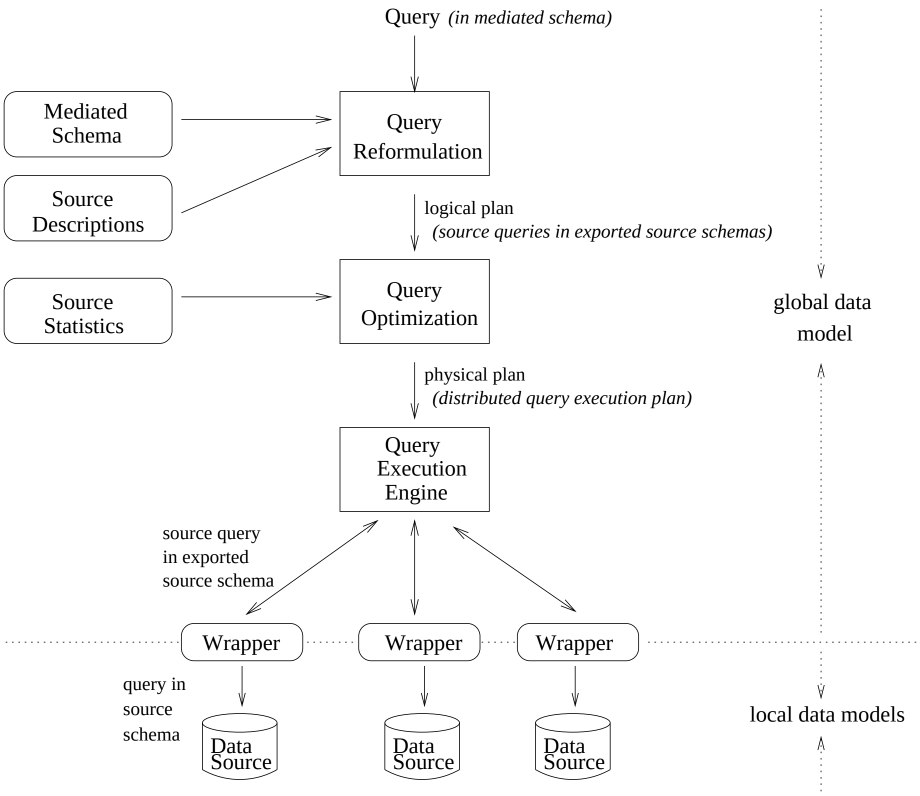

Figure 1.4: Stages of Query Processing

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 relational queries over their relations. 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 [].

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:

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

where q and $r_1, \dots, r_n$ are predicate names and $\bar{X}, \bar{X_1}, \dots, \bar{X_n}$ are either variables or constants. The atom $q(\bar{X})$ is called the *head* of the rule and the atoms $r_1(\bar{X_1}), \dots, r_n(\bar{X_n})$ 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.

.... explain the semantics of the rules, IDB, EDB predicates, conjunctive queries, recursive rules, etc ...

4.1.2 Modeling the Data Sources

To reformulate a query in mediated schema as queries on 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:

Global As View (GAV) Approach

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

example will come here

This approach was taken in the TSIMMIS System [].

Local As View (LAV) Approach

For each data source S, a rule over the relations in the mediated schema is written that describes which tuples are found in S.

example will come here

This 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 [].

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 is also asked in terms of this domain model. In other words, the model provides a language or terminology for accessing the sources.

example will come here

This approach was taken in the SIMS System [].

Each of these approaches has certain advantages and disadvantages over the others. 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 []. 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. As we shall see in the next section, the most important work done on query reformulation focus on the LAV approach. 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 being more flexible.

... need to learn DL more to compare ...

4.1.3 Using Probabilistic Information

... will be written later ...

- for source completeness
- for overlap between parts of the mediated schema
- for overlap between information sources

4.2 Query Reformulation

Using the source descriptions, 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:

- 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 straight forward 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.

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 []. 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 **??**.

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. Note that we do not sacrifice from semantic correctness criterion here, rather we are preferring an incomplete answer to no answer at all. The below definitions formalize these terms:

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'.

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.

4.2.2 Completeness and Complexity of Finding Query Rewritings

... will be written later ...

- source incompleteness
- recursive rewritings

4.2.3 Reformulation Algorithms

Given a query Q and a set of views $V_1 \ldots V_n$, 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 is that the number of query rewritings that need to be considered can be drastically reduced if we first consider each subgoal in the query in isolation, and determine which views may be relevant to each subgoal. Given a query Q, the Bucket Algorithm proceeds in two steps:

1. The algorithm creates a bucket for each subgoal in Q, containing the views (i.e., data sources) that are relevant to answering the particular subgoal. More formally a view V is put in the bucket of a subgoal g in the query if the definition of V contains a subgoal g_1 such that
   - g and g_1 can be unified, and

- after applying the unifier to the query and to the variables of the view that appear in the head, the predicates in Q and in V are mutually satisfiable.

The actual bucket contains the head of the view V after applying the unifier to the head of the view.

2. The algorithm considers 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.

example will come here

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. 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 examples illustrates the algorithm:

example will come here

In general, one function is created for each existential variable that appears in the view definitions. These function symbols are used in the heads of the inverse rules. 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.

The MiniCon Algorithm

MiniCon Algorithm looks at the problem from another perspective. Instead of building rewritings by combining rewritings for each query subgoal or mediated schema relation, we consider how each of the variables in the query can interact with the available views. This way the number of view combinations to be considered can be considerably reduced. The MiniCon Algorithm, like the Bucket Algorithm, first tries to identify which views contain subgoals that correspond to subgoals in the query. However, rather than building buckets, MiniCon Descriptions (MCDs) are built. MCDs are generalized buckets. Each correspond to a set of subgoals from the query mapped to subgoals from a set of views. First the algorithm finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V. Then it looks at the variables that appear in join predicates in the query. The minimal additional set of subgoals that need to be mapped to subgoals in V given the partial mapping between g and

g_1 is found. These subgoals together with their mappings form an MCD. The following example clarifies the algorithm.

example will come here

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. Like the Bucket Algorithm, there are two steps: bucket construction and solution generation.

During the bucket construction step, Shared-Variable-Bucket Algorithm considers the equality constraints introduced by the "shared variables", i.e., variables that occur across multiple subgoals. Additional buckets are constructed called Shared Variable Buckets (SVBs) in order to handle the equality constraints. Each bucket contains only views that cover all the subgoals in which the shared variables representing the bucket appear.

In the solution generation step, a set of buckets is chosen such that each subgoal is represented by a single bucket in the set. From each bucket, a view is selected. Consequently, the solution to the query is expressed as a conjunctive query whose body is the conjunct of the selected views. The extra buckets ensure that the all generated solutions are sound solutions and this way the conjunctive query containment test at the end of the Bucket Algorithm is avoided.

example will come here

The CoreCover Algorithm

In this algorithm, closed-world assumption is taken where views are materialized from base relations. 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. Contrary to the other algorithms, this algorithm aims at finding equivalent rewritings rather than contained rewritings. Three different cost models are considered with the following motivations:

- Cost model M_1 tries to minimize the number of join operations
- Cost model M_2 additionally aims at minimizing the number of disk IO's by minimizing the size of the relations use in the plan
- Cost model M_3 aims at improving M_2 by dropping irrelevant attributes from the intermediate relations during evaluation.

We will be discussing the basic CoreCover Algorithm for the cost model M_1 and refer the interested readers to [] for modified versions developed for M_2 and M_3.

Intuitively, the first step in the algorithm is to find the set of query subgoals that can be covered by a view tuple, called "tuple-core". The second step is to find a minimum number of view tuples to cover query subgoals.

Rather than a technical discussion, we will present the algorithm with the following example:

example will come here

Comparison of the Algorithms

It is important that the algorithm scales well when the number of views increase.

after I write the examples ...

4.2.4 Alternative Query Reformulation for Dynamic Information Integration

... will be written later ...

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. [**?**] proposes an adaptive query execution approach in which query optimization and execution are interleaved. In this section we mainly discuss this approach.

4.3.1 Adaptive Query Execution

In addition to the above listed problems, [**?**] 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 [**?**] 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 [] 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, reoptimization, 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 reoptimization.

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 coming section.

5 Data Extraction

Combines techniques from DB and AI (NLP, Machine learning).

5.1 Techniques for Extracting Data (Wrappers)

... will be written later ...

To access information from different heterogeneous data sources, we have to translate queries and data from one data model to another. This function is provided by wrappers around each individual data source. Wrapper converts queries into one or more queries understandable by the underlying data source and transforms results into the format understood by application (mediator).

5.1.1 Wrapper generation

- Issues
  - refer to the section on Data Models and Schema Integration
  - Mediator systems usually require more complex wrappers than do most warehouse systems

- Ways of creating wrappers

  - **Manual**
    Why is it impractical for some sources?
    In case of Web sources:
    - big number of sources
    - new sources are added frequently
    - format of sources change
    So, high maintenance costs.
  - **Semi-automatic (interactive)**
    Noted that only small part of the code deals with the specific access details of the source. The rest is common among wrappers or data transformation can be expressed in a declarative fashion (high-level). Graphical interface, programming by demonstration.
  - **Automatic**
    * site-specific or generic
    * usually needs training often supervised learning

- Tools for semi-automatic/automatic wrapper construction for structured/semistructured data

  - Template-based wrappers
  - Inductive learning techniques for automatically learning a wrapper (using labeled data)
    Inductive learning - task of computing some generalization from a

set of examples
Methods:

* zero-order (decision tree learners)
* first-order (inductive logic programming)
  - bottom-up/top-down approaches

- Tools for data extraction from unstructured documents
  - Using ontologies and conceptual models to extract and structure information from data-rich, unstructured documents.
  - Using heuristic approaches to find record boundaries in web documents.

5.1.2 Filters

If a wrapper returns a superset of what query wants, we can filter the results of the query.

5.2 Data Source Interfaces

Integrating information systems will almost always result in the development of software that accesses data sources' public interfaces.

A plethora of options is available for how data sources expose their data to other computer systems. This subsection will explore some of the issues that differentiate various types of interfaces.

5.2.1 General issues with interfaces

- **Separation of interface into application-specific and reusable layers**

  Interfaces can often be divided into two parts:

  - a set of primitive components that is used in numerous applications, and
  - a set of application-specific components.

  This distinction is made in the OSI network protocol stack an elsewhere.

  For the remainder of this section, we'll use the term **primitive interface** to refer to these multi-application, reusable interfaces.

  For example, consider a car dealership's customer database system. Suppose that the system exposes its data via a CORBA interface.

  CORBA, and perhaps TCP/IP, could be considered the primitive components of the interface that appear in many applications. Thanks to the support of outside organizations, CORBA and TCP/IP are available for use in many different applications.

However, the set of objects and methods exposed with CORBA for the customer database system is application specific. (For example, a Customer object or a Customer.scheduleForTuneup() method.) Few if any other applications are likely to ever use the same application-specific interface.

- **Resolution of data addressability**

  Primitive interfaces are often ignorant of the structure of the data that the interface helps to transfer between the interface user and the interface implementor.

  For example, one could move car inventory data over FTP in binary transfer mode. The design of FTP is such that the protocol doesn't interpret the structure of the transferred data when using binary transfer mode.

  However, it's possible that the structure of the file being transferred is very rich indeed. For example, the file could be a serialized Java object with a well-defined structure.

  This raises the issue of the resolution of addressability provided by an interface.

  In the example above, the FTP interface was capable of providing data addressability down to the file level.

  However, the FTP interface does not support the selection of a particular XML element from the XML document.

- **Data types**

  Primitive interfaces typically provide a set of one or more primitive data types that the interface explicitly recognizes.

  For example, SQL offers *varchar*, *int*, etc. C APIs offer *int*, *float*, *char**, etc.

  These primitive types may be given special treatment in the language bindings that let a programming language use the interface.

  For example, big-endian and little-endian computers use different bit-level representations of integer data. The SQL bindings for programming environments on various computers will convert the bit-level representations of SQL *int* data into a format that's appropriate for the computer using the interface.

  Application-level constructs must be expressed (directly or indirectly) in terms of these primitive data types.

- **Interface semantics**

  Primitive interfaces are designed with the intention that the primitive interface itself assigns no meaning to the particular data moving across the interface.

A result of this is that primitive interfaces are powerless to express application-level semantics of the data.

See

TO DO: Give a reference to Section 3, once section 3 has a referrable label

for more details on interface semantics.

5.2.2 Network vs. non-network primitive interfaces

In data integration, an important aspect of primitive interfaces is whether or not network communications are used.

- **Non-network interfaces**

  These interfaces do not explicitly support network communications. This is a problem because data integration may involve two or more computers that must communicate.

  If data integration requires that two or more computers communicate, but one or more data source does not offer a network-friendly interface, work-arounds may be painful:

  - Adding network connectivity may introduce yet more software to add network connectivity.
  - Using off-line data transfer mechanisms (tape, CD-RW, etc.) can lead to undesirable latencies in the transfer of data from a data source to the integrating system.

  Examples

  - **Using a removable disk for file transfers ("sneaker net")**

    A data source writes to files on a local disk drive. The files are then copied to a CD-RW disk, which is phyisically transported to a computer that's performing data integration. The files are then read from the CD-RW by the integration software.
  - **Information available only via a local GUI only (i.e., MS Windows)**

    Suppose that a GUI application is the only exposed interface to a particular data source. Developers of an integrated system may have quite a difficult time accessing the data in an automated manner, because poor bindings exist to allow one application to manipulate the GUI of another application intelligently.

- **Network-capable interfaces**

  These interfaces explicitly support network communications.

Unfortunately, many modern networks (both short-distance and long-distance) have a set of problems that can be difficult to deal with from a software level.

- **Failures of network links**

  Various problems can and do occur in the connections between computers. In the modern Internet, it's common to hear of a backhoe operator accidentally digging through a bundle of optic cable used by an Internet service provider, or of a necessary router malfunctioning.

  In naively integrated systems, a network link failure can cripple the entire integrated system or cause the loss of data.

- **Potentially independent failures of communicating applications**

  If the software for integrating data executes on a different computer than one of the data sources, it's possible for only one of the computers to fail.

  This possibility raises very similar problems in system design to those problems arising from fallible network links.

- **Potentially significant communications costs / performance issues**

  In long-distance communications between computer systems, transferring data is often expensive. Businesses that have multiple offices often lease expensive network connections to allow the computers at each site to communicate quickly with computers at other sites.

  Fiscal cost may need to be a consideration in system design when data integration involves computers at separate sites.

Examples

- **FTP** - *File Transfer Protocol*

  This protocol gives one computer access to part of another computer's file-system. It also offers a username/password form of access control.

- **CORBA** - *Common Object Request Broker Architecture*

  This architecture allows software systems to expose objects, methods, and other details to provide object-oriented network services.

- **ODBC** - *Open DataBase Connectivity*

  This protocol is designed to allow applications to access relational database data without regard to the vendor of the database.

6 Materialized View Management

... this whole section will be written later ...

We are going to discuss two major subtopics under this heading with giving higher emphasis to the second one:

- Design and selection of views to materialize
- Maintenance of the materialized views

6.1 Design and Selection of Views to Materialize

Major Goals:

- to minimize total query response time
- to minimize the cost of maintaining the selected views

6.2 The Problem of View Maintenance

6.2.1 Definition

- traditional view update problem and why this one is different and more difficult
- As the database changes because of updates applied to the base relations, the materialized views may also require change. A materialized view can always be brought up to date by re-evaluating the view definition. However, complete re-evaluation is wasteful.
  "heuristic of inertia"- only a part of the view changes in response to changes in the base relations
- steps of view maintenance:
  - propagation (computing changes to the view)
  - refreshing (applying changes to the mat. view)

6.2.2 Dimensions

- Available Information
  materialized view, base relations, other views, integrity constraints, etc.
- Allowable Modifications
  insertions, deletions, updates, sets of each, group updates, change view definition, etc.
- Expressiveness of the View Definition Language
  conjunctive queries, duplicates, aggregation, union, recursion, negation, select, project, join, spj, etc.

- Database Instance
- Complexity
  - Complexity of the View Maintenance Language
  - Complexity of the View Maintenance Algorithm
  - Complexity of Auxiliary Information (in terms of space)

6.2.3 View Maintenance Policies

when to apply maintenance procedures on the materialized views

- Immediate View Maintenance
  Refreshing is done within the transaction that changes the base data. slow transactions; faster queries and always up to date results.
- Deferred View Maintenance
  - lazily, at query time
    fast transactions; slow query speed.
  - forced, after a certain amount of change to the base data
    non up to date results; good transaction and query time.
  - periodically, in certain time intervals
    non up to date results; good transaction and query time.

comparison of all in general; how the decision when to use which one is made.

6.3 Incremental View Maintenance

- pre-update vs post-update algorithms
- various algorithms will be summarized

6.4 View Maintenance Anomalies(Consistency Issues)

- caused by decoupling btw view definition and base data
- definition of correctness and levels of correctness
- solutions:
  - recomputing the views
  - storing copies of base relations
  - ECA (Eager Compensating Algorithm)

6.5 Update Filtering

Detection of base data updates that are irrelevant to the view (i.e., have no effect on the state of the view) wil be discussed. For such updates, we do not need to perform any maintenance. Thus, update filtering makes maintenance more efficient by preventing redundant work.

6.6 View Self-Maintenance

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 avaiable. At minimum, the view update is performed at the integrated system by only knowing the particular base data update that has occurred, the view definitions and the materialied data. We will be describing the alternative notions in detail here.

- Questions
  - Given a view, is it self-maintainable?
  - If it is self-maintainable, how?
- Single-View Self-Maintenance
  does not consider the materialized views
- Multiple-View Self-Maintenance
  makes use of contents of the other materialized views to minimize base data access
- Updates as a whole
  i.e., batch updates; may make maintenance easier and more efficient
- Making views self-maintainable
  When the answer to the first question listed above is No, we can define and materialize a minimal set of auxiliary views to make the original non-maintainable view maintainable. Here, basically we are increasing the amount of information available at the integrated system level.

6.7 Dynamic View Management

- problems of static view selection and maintenance
- dynamic view selection and maintenance
- performance parameters
  - space (taken up by the materialized views)
  - workload (changing query workload)
  - maintenance window (how often we would like to perform maintainance and how long is the system tolerated to be unfunctional)

- the solution in DynaMat

7 Systems

There are a huge bunch of existent systems which intend to develop tools to facilitate the integration of heterogeneous sources. In this section, we present some typical examples. We focus on some university projects, instead of commercial systems.

7.1 Mediated systems

I.TSIMMIS

TSIMMIS stands for "the Stanford-IBM Manager of Multiple Information Sources."

TSIMMIS consists of two main components, one is the source specific translators (wrappers), the other is the "intelligent" mediators. Translators(wrappers) are responsible to convert queries over information in some common model into request that the source can execute, and convert the data returned from the source back into the common model. Mediators are some programs that assemble information from sources, process and combine it, and transmit the final information to the end user.

In Tsimmis, they use a simple-describing object model, the *Object Exchange Model, or OEM.* OEM allows simple objects' nesting and all objects have *labels* to describe their meaning. They also have developed the OEM-QL query language to request OEM objects. OEM-QL is a SQL-like language specified to deal with labels and object nesting. In Tsimmis, both mediators and translators are automatically or semi-automatically generated from their high level request of the information process.

For interface, mediators and translators both take as input OEM-QL queries and return OEM objects. The good point here is that it allows new sources useful once a translator is supplied. There are two ways for end users to get information, one is to write applications that ask for OEM objects, the other is to use their developed browsing tool, named MOBIE(MOsaic Based Information Explorer), to specify queries using OEM-QL.

Another important issue in Tsimmis is that there is no global schema. A mediator does not need to know details all of the data it used. It is not necessary for any person or software component to have a global view of all the information managed by the system.

In Tsimmis, constraint management is more difficult than those centralized systems. Usually they do not have transactions among different sources. Each source may have different policies to those data involved in a constraint. It is not guaranteed that consistent data will be accessed at each time it interacts with the system.

In summary, we list three main difference between Tsimmis and other systems.

- Tsimmis concentrates on providing an integrated system which deals with very diverse and dynamic information.
- In Tsimmis, information access and integration are intertwined.
- Tsimmis requires more human participation in their integration system.

II.SIMS

SIMS stands for the "Services and Information Management for decision Systems".

SIMS is an information mediator for processing queries to multiple information sources. This system takes a domain-level query and dynamically chooses the useful sources, generates a query plan which describes the operations and some specific orders to deal with the data, and performs semantic query optimization.

The application domain models are defined by nodes, representing each class of objects, and their relations, defining relationships between the objects. Queries in SIMS are represented by the general domain model. The system translates the domain-level query into a set of source-level queries. The information source model define both the contents of the objects and their relationship.

To answer a query, SIMS first selects the appropriate information sources. The system provides a set of reformulation operators that are responsible to transform the domain-level concepts into concepts that the information source could accept. The operators include Select-Information-Source, Generalize-Concept, Specialize-Concept, and Decompose-Relation.

The next step is to generate a query plan for the data process. The query plan defines the concrete operations that need to be executed and the order in which they will be executed. The system searches all possible plans with a best-first method until a complete one is found.

Finally, the system performs the semantic query optimization. "A set of applicable rules for the query is constructed. These rules would either be learned by the system or provided as semantic integrity constraints. Based on these rules, the system infers a set of additional constraints and merges them with the input query. The resulting query is semantically equivalent to the input query but is not necessary more efficient. The set of constraints in this resulting query is called the **inferred set**. The system will then select a subset of constraints in the **inferred set** to complete the optimization."

In summary, SIMS provides some ideas which are different to other integration systems.

- In SIMS, the integration problem is shifted from building a single integrated model to how to map between the domain and the information source models.
- The planning in SIMS is performed by an AI planner.

- Compared to other related works to search optimized queries, their algorithm considers "all possible optimizations by firing all applicable rules and collecting candidate constraints in an *inferred set.* Then the system selects the most efficient set of the constraints from the inferred set to form the optimized subqueries".

II.Infomaster

7.2 Data warehousing

I. Whips

The goal of WHIPS (Warehouse Information Prototype at Stanford) is to develop algorithms and tools for the creation and maintenance of data warehouses.

8 Open Questions and Research Issues

... will be formed later from everybody's list of research issues from the above sections ...

9 Concluding Remarks

... will be written later ...

Bibliography